VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DEVELOPMENT OF A WIRELESS AD-HOC ROUTER

By
Hong Vuong Anh

A thesis submitted to the School of Computer Science and Engineering in partial fulfillment of the requirements for the degree of
Master of Information Technology Management

Ho Chi Minh city, Vietnam

2013

DEVELOPMENT OF A WIRELESS AD-HOC ROUTER

APPROVED BY:

___________________________________ ,

Quan Le-Trung, Dr.techn (Supervisor)

____

____

____

____

THESIS COMMITTEE

ABSTRACT

Mobile ad-hoc networks (MANETs) are subset of wireless network that helps us to overcome the limited freedom of mobility of infrastructure wireless network. MANETs are infrastructure-less, self-created and seft-organized networks that consists of mobile nodes, interconnected by multi-hop wireless paths.

Many routing protocols have been proposed and implementation were developed for routing purposes in MANETs environment. They normally a stand-alone application that serve the main goal – routing function. As for modern router (in term of software router), we need more than just pure routing functionalities such as access control list, route map policy, redistribute mechanism when we have multiple routing protocol implementation running in the network, etc ...

This master thesis deals with the topic to develop a wireless ad-hoc router. With research of existing software router in the market and community, the author chose to integrate the implementation of AODV-UU (AODV routing protocol) to Quagga routing suite for ad-hoc routing concern. The author also update the existing release of OLSRqd (OLSR routing protocol built for Quagga 0.98) in order to work in new Quagga release. With this approach, the author aim to add ad-hoc routing solutions (supports both pro-active and re-active protocols) for Quagga routing framework.

ACKNOWLEGMENTS

This thesis concludes my Master's Degree of Information Technology Management, and is submitted to the School of Computer Science and Engineering at the International University, Vietnam National University – Ho Chi Minh City.

I would like to Dr. Quan Le-Trung for his guidance and helpful advice. His skillful and valuable comments and feedback help me get back on track whenever I lost sight of objectives of this thesis.

Last but not least, I would like to thank my family for unconditional support and encouragement, thank my friends for valuables feedback.

TABLE OF CONTENTS

LIST OF ACRONYMS, ABBREVIATION

MANET(s): Mobile Ad-hoc Network(s)

AODV: Ad-hoc On-demand Distance-Vector routing protocol.

AODV-UU: An AODV implementation developed at Uppsala University, Sweden by Erik Nordström <erik.nordstrom@it.uu.se>.

RERR: AODV Route Error message

RREP: AODV Route Reply message

RREQ: AODV Route Request message

OLSR: Optimized Link State Routing protocol.

OLSRqd: An OLSR implementation for Quagga 0.98.5 by Tudor Golubenco.

TC: OLSR Topology Control message

MID: OLSR Multiple Interface Declaration message

MPR: Multipoint Relays

MPRS: Multipoint Relays Selector

LIST OF TABLES

LIST OF FIGURES

Chapter 1. INTRODUCTION

1.1. Introduction

As wireless communication technology is becoming more and more popular, people expect to be able to use their network terminals anywhere and anytime. Examples of such terminals are PDAs and laptops. Users wish to move around while maintaining connectivity to the network (i.e., Internet), and wireless networks provide them with this opportunity.

Wireless connectivity to the network gives users the freedom of movement they desire. Most wireless networks today require an underlying architecture of fixed-position routers, and are therefore dependent on existing infrastructure. Typically, the mobile nodes in such networks communicate directly with access points (APs), which in turn route the traffic to the corresponding nodes. Today, another type of wireless network is emerging, namely ad hoc wireless networks. These networks consist of mobile nodes and networks which themselves create the underlying architecture for communication. Because of this, no fixed-position routers are needed.

1.2. Statement of problems

Traditional software routing application is made of one program process to provide core routing functionalities – computing its routing table and synchronize with the kernel routing table. As for modern router (in term of software router), we need more than just pure routing functionalities such as access control list, route map policy, redistribute mechanism when we have multiple routing protocol implementation running in the network, etc ...

In this thesis, I aim to do integrating the implementation of AODV-UU 0.9.6 (developed by Erik Nordstrom at the Uppsala University) and OLSRdq 0.1.18 (developed by Tudor Golubenco for Quagga 0.98.5) into the update release of Quagga 0.99.22.1.

1.3. Methodology

The thesis work is done with following approach

- Analysis of source code of AODV-UU 0.9.6, OLSRdq 0.1.18, Quagga 0.99.22.1.

- Update Quagga 0.99.22.1 source code to support the integration with AODV and OLSR daemons.
- Update AODV-UU 0.9.6 source code with sufficient data structure and function modules to establish the communication channel between AODV daemon and Zebra daemon for route add/delete.
- Update OLSRqd 0.1.18 source code to work with Quagga 0.99.22.1 release.
- Perform realistic test scenarios to prove functionality of the integration.

1.4. Results

- AODV daemon can work and communicate with Zebra daemon for kernel routing table update.
- OLSR daemon can work with Zebra daemon in new Quagga release. Add new vtysh command to start OLSR routing on a specific interface.
- Perform successul testcase with three nodes for both AODV and OLSR daemons.

1.5. Scope and limitation

- AODV daemon can now send route add and delete request to Zebra daemon.
- OLSR routing supports only core functionalities from RFC.
- Only supports IPv4 in both daemons.
- Testcase organized under ad-hoc environment. Internet access testcases are being prepared.

1.6. Structure of thesis

The thesis composed of following chapter

Chapter one: Introduction

Chapter two: Literature review.

- Desbribe the overview of routing in MANETs in general with the two represent routing protocols AODV and OLSR.
- Desbribe the overview of AODV-UU 0.9.6 implementation – structures and code analysis of AODV-UU 0.9.6 package includes main data structures, main function calls flow to establish link and message update between AODV daemon and zebra.

- Describe the overview of OLSRdq 0.1.18 implementation - structures and code analysis of OLSRdq 0.1.18 package includes main data structures, main function calls flow to establish link and message update between OLSR daemon and zebra.
- Describe the overview of Quagga 0.99.22.1 release - structures and code analysis of Quagga 0.99.22.1 package includes main data structures, main function calls flow to establish link and message update between Zebra daemon and user-defined routing protocols and kernel routing table.
- Describe main data structures of kernel routing tables in Linux, flows of functions to add/delete/update entries in the routing tables

Chapter three: Methodology

This chapter describes the details design and integration of the two implementations AODV-UU and OLSRdq to Quagga 0.99.22.1. Details regarding function calls flow, packet flow, modification to AODV-UU 0.9.6 and OLSRdq 0.1.18 in order to work with Quagga 0.99.22.1.

This chapter consists of two sub-sections that each of them serves for AODV and OLSR integration.

Chapter four: Testing scenarios and results

Suggest test scenarios to illustrate the integration of AODV-UU and OLSRdq with Quagga 0.99.22.1.

Chapter five: Conclusion and future work

Conclusion the implementation and suggestion for future work.

Chapter 2. LITERATURE REVIEW

2.1. Routing in MANETs

2.1.1. General ad-hoc routing features & classes

a. Routing in ad-hoc wireless networks

MANETs is a type of wireless network where there is no existing infrastructure and devices in this network are free to move to any directions. Because of the infrastructureless nature, each node in MANET is responsible for the routing and forwarding packets. If nodes are within range of each other, then no routing is needed. However, it some nodes move out of range of each other, they can not communicate directly and thus, intermediate nodes must take part in the routing and forwarding task.

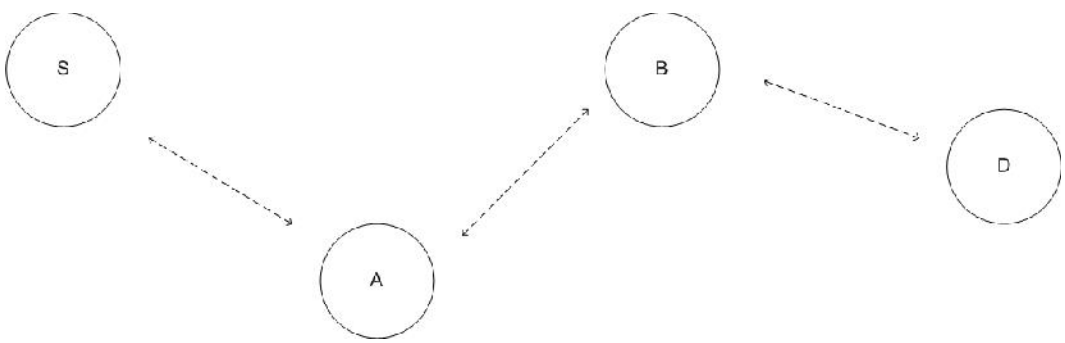


Figure 1: Communication in MANET

In figure 01, intermediate nodes A and B acts as a router to forward packets for the communication of nodes S and D.

MANET routing protocols are typically subdivided into two main categories: proactive routing protocols and reactive on-demand routing protocols. With this approach, we can classify MANET routing protocols into following types:

- Reactive protocols

Unlike proactive routing protocols, reactive routing protocols determine route to a destination node on demand. If a node wants to communicate with another node but there is no available route to destination, the route discovery is initialized. Adhoc On-Demand Distance-Vector routing protocol (AODV) as described in RFC3561 [01] is further discussed in section [2.1.2].

- Proactive protocols

Proactive routing is based upon table driven approach where every node compute and manage routing tables to all other nodes in the network topology. These tables are updated frequently and exchanged to other nodes in order to ensure the up-to-date routing information from each node to others.

Many proactive protocols stem from conventional link state routing, including the Optimized Link State Routing protocol (OLSR) as described in RFC3626 [04] which is discussed in section [2.1.3].

- Hybrid protocols

These types of protocols combine proactive and reactive protocols to exploit their strengths. One approach is to divide the network into zones, and use one protocol within the zones, and another between them.

2.1.2. AODV

AODV is an on-demand routing algorithm that determines a route only when a node wants to send a packet to a destination. It is a relative of the Bellman-Ford distant vector algorithm, but is adapted to work in a mobile environment. Routes are maintained as long as they are needed by the source. AODV is capable of both unicast and multicast routing. [06]

In AODV every node maintains a table containing information about which direction to send the packets in order to reach the destination.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |J|R|G|D|U|   Reserved          |   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RREQ ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination IP Address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator IP Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Originator Sequence Number                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: AODV ROUTE REQUEST message format. [01]

AODV uses sequence numbers to ensure the freshness of routes and avoid loop problem.

a. Control Messages

Three message types are defined by AODV:

RREQ: When a route is not available for the desired destination, a route request packet is flooded throughout the network. Figure [02] shows the format of such a message.

RREP: If a node either is the destination or has a valid route to the destination, it unicasts a route reply message back to the source.

RERR: When a link breaks, the nodes on both sides of the link issue a route error to inform their end nodes of the link break. [06]

b. Sequence numbers

AODV differs from other on-demand routing protocols in that it uses sequence numbers to determine an up-to-date path to a destination. Every entry in the routing table is associated with a sequence number. The sequence numbers act as a route timestamp, ensuring the route remains up-to-date. Upon receiving a RREQ packet, an intermediate node compares its sequence number with the sequence number in the RREQ packet. If the sequence number already registered is greater than that in the packet, the existing route is the most up-to-date. [06]

Counting to infinity problem

The use of sequence numbers for every route also helps AODV avoid the "count to infinity" problem. This problem arises in situations where nodes update each other in a loop. "The core of the problem is that when X tells Y that it has a path somewhere, Y has no way of knowing whether it itself is on the path". So if Y detects that the link to, say, Z is down, but X says it has a valid path, Y assumes X in fact does have a path, thus registering X as the next neighbor toward Z. If the path X assumed is valid runs through Y, X and Y will start updating each other in a loop. [06]

c. Route discovery

Route discovery is initiated by issuing a RREQ message. The route is established when a RREP message is received. However, multiple RREP messages may be received, each suggesting different routes to the destination. The source only updates its path information if the RREP holds information about a more up-to-date route than already registered. Thus, every incoming RREP packet is examined to determine the most current route. [06]

When a intermediate node receives either a RREQ or a RREP packet, information about the previous node from which the packet was received is stored. This way, next time a packet following that route is received, the node knows which node is the next hop toward the source or destination, depending on which end node originated the packet. [06]

The next subsection illustrates route discovery by providing an example.

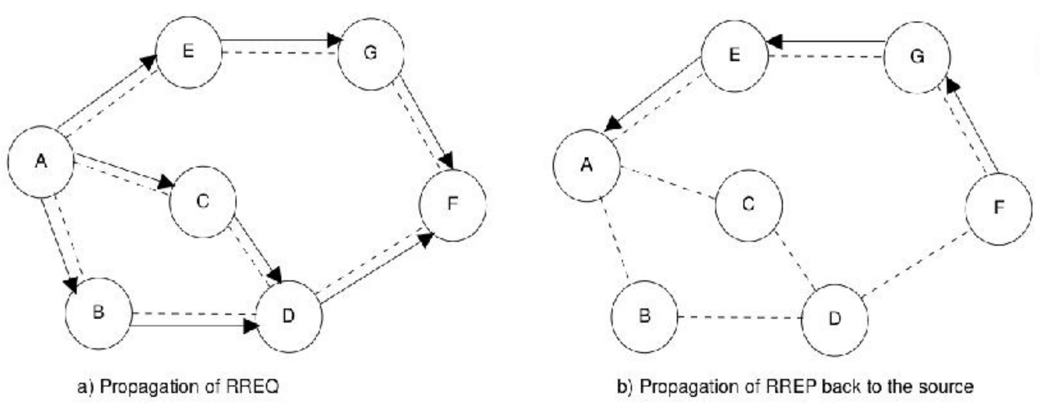


Figure 3: AODV route discovery [06]

Consider the ad-hoc network of Figure 2. In this example, node A wants to send a packet to node F. Suppose A has no table entry for F. A then needs to discover a route to F. In our example, we assume that neither of the nodes knows where F is.

The discovery algorithm works like this:

Node A broadcasts a special ROUTE REQUEST packet on the network. The format of the ROUTE REQUEST (RREQ) packet is shown in figure 3.3 on the preceding page. Upon receiving the RREQ packet, B, C and E check to see if this RREQ packet is a duplicate, and discards it if it is. If not, they proceed to check their tables for a valid route to F. If a valid route is found, a ROUTE REPLY (RREP) packet is sent back to the source. In the case of the destination sequence number in the table being less than the destination sequence number in the RREQ, the route is not considered up-to-date, and thus no RREP packet is sent. Since they don't know where F is, they increment the RREQ packet's hop count, and rebroadcast it. In order to construct a route back to the source in case of a reply, they also make an entry in their reverse route tables containing A's address.

Now, D and G receive the RREQ. These go through the same process as B, C and E. Finally, the RREQ reaches F, which builds an RREP packet and unicasts it back to A. [06]

The Expanding Ring search

Since RREQ packets are flooded throughout the network, this algorithm does not scale well to large networks. If the destination node is located relatively near the source, issuing a RREQ packet that potentially passes through every node in the network is wasteful. The optimization AODV uses is the expanding ring search algorithm. The source node searches successively larger areas until the destination node is found. This is done by incrementing the time to live (TTL) value carried in every RREQ packet for every RREQ retransmission until a route is found thus expanding the "search ring" in which the source is centered. [06]

d. Link Breakage

When a link breaks, a RERR message is propagated to both the end nodes. This implies that AODV does not repair broken links locally, but rather makes the end nodes discover alternate routes to the source. Moreover, link breakage caused by the movement of end nodes also results in initialization of a route discovery process. When an RERR packet is received by intermediate nodes, their cached route entries are removed. [06]

2.1.3. OLSR

The Optimized Link State Routing (OLSR) is a table-driven, proactive routing protocol developed for MANETs. It is an optimization of pure link state protocols that reduces the size of control packets as well as the number of control packet transmissions required. [06]

OLSR reduces the control traffic overhead by using Multipoint Relays (MPR), which is the key idea behind OLSR. An MPR is a node's one-hop neighbor which has been chosen to forward packets. Instead of pure flooding of the network, packets are forwarded by a node's MPRs. This delimits the network overhead, thus being more efficient than pure link state routing protocols. [06]

OLSR is well suited to large and dense mobile networks. Because of the use of MPRs, the larger and more dense a network, the more optimized link state routing is achieved. [06]

MPRs helps providing the shortest path to a destination. The only requirement is that all MPRs declare the link information for their MPR selectors (i.e., the nodes which have chosen them as MPRs). [06]

The network topology information is maintained by periodically exchange link state information. If more reactivity to topological changes is required, the time interval for exchanging of link state information can be reduced. [06]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Reserved             |     Htime     |  Willingness  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Link Code   |   Reserved    |       Link Message Size       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Neighbor Interface Address                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Neighbor Interface Address                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                             .  .  .                           :
:                                                               :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Link Code   |   Reserved    |       Link Message Size       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Neighbor Interface Address                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Neighbor Interface Address                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
:                                       :
(etc.)
```

Figure 4: OLSR HELLO packet format [04]

a. Control messages

OLSR uses three kinds of control messages: HELLO, Topology Information (TC), and Multiple Interface Declaration (MID).

A Hello message is sent periodically to all of a node's neighbors. Hello messages contain information about a node's neighbors, the nodes it has chosen as MPRs (i.e., the MPR Selector set), and a list of neighbors with whom bidirectional links have not yet been confirmed.

Figure 04 above shows the format of the Hello message.

Every node periodically floods the network with TC messages using multipoint relaying mechanism. This message contains the node's MPR Selector set.

MID message is used for announcing that a node is running OLSR on more than one interface.

The MID message is flooded throughout the network by the MPRs. [06]

b. Multipoint Relays

A node N selects an arbitrary subset of its 1-hop symmetric neighbors to forward data traffic. This subset, referred to as an MPR set, covers all the nodes that are two hops away. The MPR set is calculated from information about the node's symmetric one hop and two hop neighbors. This information is extracted from HELLO messages. Similar to the MPR set, an MPR Selectors set is maintained at each node. An MPR Selector set is the set of neighbors that have chosen the node as their MPR.

Upon receiving a packet, a node checks it's MPR Selector set to see if the sender has chosen the node as MPR. If so, the packet is forwarded, else the packet is processed and discarded. [06]

c. Selection of Multipoint Relay Nodes

The MPR set is chosen so that a minimum of one-hop symmetric neighbors are able to reach all the symmetric two-hop neighbors. In order to calculate the MPR set, the node must have link state information about all one-hop and two-hop neighbors. Again, this information is gathered from HELLO messages. Only nodes with willingness different than WILL_NEVER may be considered as MPR. [06]

d. Neighbor discovery

As links in an ad-hoc network can be either unidirectional or bidirectional, a protocol for determining the link status is needed. In OLSR, HELLO messages serve this purpose. HELLO messages are broadcast periodically for neighbor sensing. When a node receives a HELLO message in which it's address is found, it registers the link to the source node as symmetric. [06]

As an example of how this protocol works, consider two nodes A and B which have not yet established links with each other. Firstly, A broadcasts an empty HELLO message. When B receives this message and does not find its own address, it registers in the routing table that the link to A is asymmetric. Then B broadcasts a HELLO message declaring A as an asymmetric neighbor. Upon receiving this message and finding its own address, A registers the link to B as symmetric. A then broadcasts a HELLO message declaring B as a symmetric neighbor, and B registers A as a symmetric neighbor upon reception of this message. [06]

e. Topology Information

Information about the network topology is extracted from topology control (TC) packets. These packets contain the MPR Selector set of a node, and are broadcast by every node in the network, both periodically and when changes in the MPR Selector set are detected. The packets are flooded in the network using the multipoint relaying mechanism. Every node in the network receives such TC packets, from which they extract information to build a topology table. [06]

f. Route Calculation

The shortest path algorithm is used for route calculations, which are initiated when a change is detected in either of the following: the link set, the neighbor set, the two-hop neighbor set, the topology set, or the Multiple Interface Association Information Base. [06]

To calculate the routing table, information is taken from the neighbor set and the topology set. The calculation is an iterative process, in which route entries are added starting from one-hop neighbors, increasing the hop count each time through. [06]

2.2. AODV implementation

In this section, I mention about the software modules for AODV implementation and some main data structure to be used in the release. The details operations of AODV daemon – AODV routing functionalities and the integration with Zebra daemon will be list out later in section 3.3.

Note that the AODV implementation in this writing is referred to the updated version of AODV-UU 0.9.6 in order to work with Quagga 0.99.22.1.

2.2.1. AODV software modules

This section mentions about AODV software modules, this is an updated version of the initial AODV-UU 0.9.6 release to work with Quagga 0.99.22.1. In summary, the source code files can be group into three types:

- AODV group: this software module group handle AODV routing functionality and generate the user-defined routing table (this routing table is synchronize with the kernel routing table via Zebra daemon).

- AODV kernel group: this software module group handle kernel functionality that required by AODV routing protocol, especially the determination to start route request.

- AODV Zebra communication group: this software module group handle the initializing and maintain the communication channel between AODV daemon and Zebra daemon.

a. AODV group

aodv_hello.{c,h}

This module contains HELLO message functionality. It offer functions to generate and send HELLO messages, scheduling of HELLO generation and processing.

aodv_rerr.{c,h}

This module contains RERR message functionality. Moreover, this module defines a data type for RERR messages and unreachable destinations, and offers RERR message creation, addition of unreachable destinations to RERR messages and processing of RERR messages [02]

aodv_rrep.{c, h}

The aodv_rrep module contains RREP message functionality. This file defines the datatype for RREP and RREP-ACK messages. The latter are used for acknowledgment of RREP messages, if the link over which the REP was sent may be unreliable or uni-directional. The module offers creation and processing of messages of these two message types. [02]

aodv_rreq.{c, h}

This module contains RREQ message functionality. It defines the data type for RREQ messages, offers creation and processing of RREQ messages, and allows route discoveries to be performed. It also offers buffering of received RREQs and blacklisting of RREQs from certain nodes. The blacklisting is used for ignoring RREQs from nodes that have previously failed to receive or acknowledge RREPs correctly, e.g. due to a uni-directional link.[02]

aodv_socket.{c, h}

This module contains the socket functionality of AODVd, responsible for handling AODV control messages. It maintains two separate message buffers; a receive buffer and a send buffer, used for storing an incoming or outgoing message until it has been processed or sent out. Each buffer holds only one (1) message at a time. This is not a problem, since packets are handled one by one, and hence safely can be discarded from the buffers after message processing or sending has completed. [02]

The module offers creation of an AODV message (initialization of the send buffer), queuing of an AODV message (copying of message contents to the send buffer) and processing of a received packet. The packet processing function checks the type field of the packet, converts it to the appropriate message type and calls the correct handler function, e.g. rrep_process() of the aodv_rrep module if the message is a RREP message.[02]

To comply with zebra infrastructure, the receiving task is re-implemented. A dedicated thread is initialized when the command "router aodv" is applied (either via user/administrator input from vtysh session or from a saved aodvd.conf file). This thread will listen for incoming packet, check whether it is an AODV control message, check the type field of the packet and then call the correct handler function.

aodv_timeout.{c, h}

This module contains handler functions for timeouts, i.e., functions that are called when certain pre-determined events occur. The handler functions receive a pointer to the affected object, e.g. a routing table entry, as a parameter. Handler functions are defined for the following events:

- Route deletion timeout: This timeout occurs when a route has not been used for a certain period of time. This deletes the route from the internal routing table of AODV-UU.

- Route discovery timeout: This timeout occurs when route discovery was requested, but a route was not found within a certain amount of time. If the expanding ring search option is enabled, this results in a new route discovery with a larger TTL value than in the previous attempt. Otherwise the packet is dropped, an ICMP Destination Host Unreachable message is sent to the application and the destination is removed from the list of destinations for which AODV-UU is seeking routes.

- Route expiry timeout: This timeout occurs when the lifetime of a route has expired. This marks the route as down, creates a RERR message regarding unreachable destinations and sends the RERR message to affected nodes.

- HELLO timeout: This timeout occurs when HELLO messages from a node stop being received. This timeout occurs on an individual basis for each affected route, and is treated as a link failure, i.e., the route expiry timeout handler function is called.

- RREQ record timeout: This timeout occurs when a RREQ message has become old enough that subsequent RREQs from the affected node with a certain RREQ ID should not be considered as duplicates(i.e., being discarded if received) anymore.

- RREQ blacklist timeout: This timeout occurs when RREQs from a certain node should not be ignored anymore. Nodes end up in the blacklist by repeatedly sending RREQs for a certain destination, even though RREPs have been sent back to the requesting node. (The cause for this could e.g. be uni-directional links.) The RREQ blacklist timeout handler function removes a node from this blacklist so that normal operation is resumed.

- RREP-ACK timeout: If RREPs are sent over an unreliable or uni-directional link, a RREP-ACK can be requested by the sending node. If no RREP-ACK is received within a certain time period, this timeout occurs. The result is that the node that failed to reply with a RREP-ACK is added to the RREQ blacklist, described earlier.

- Wait-on-reboot timeout: This timeout occurs when the 15-second wait-on-reboot phase at start-up has elapsed. It resumes active operation of the node, i.e., re-enables transmission of RREP messages.[02]

debug.{c, h}

This module contains the logging functionality of AODVd. It offers logging of general events to a log file as well as routing table logging to a routing table log file. It also contains functions for converting IP addresses and AODV message flags to strings, for the purpose of printing and logging.[02]

Logging of all general events is performed by a special logging function which examines the severity of the log message, checks the current log settings, and writes log messages to the appropriate log files. By default, log messages are also written to the console. [02]

Routing table logging is performed by periodically dumping the contents of the routing table to a routing table log file. Finally, debug-purpose logging is done through a special DEBUG macro in the source code. Such logging will only be effective if AODVd has been compiled with the DEBUG option set. [02]

defs.h

This header file contains macros and data types used throughout AODVd, and hence, almost all the other modules include it. A brief listing of the contents can be found below AODVd version number, log file and routing log file paths, a definition of infinity and infinity checking, maximum number of interfaces, data type for host information, a data type for network devices, macros for retrieving network device information, DEV_IFINDEX (ifindex), DEV_NR (n), AODV message types, macros to access AODV extension, a type definition for callback functions [02]

params.h

This header file contains the following constants, defined by the AODV draft [02]:

ACTIVE_ROUTE_TIMEOUT: The lifetime of an active route.
TTL_START: Initial TTL value to be used for RREQs.
DELETE_PERIOD: Time to wait after expiry of a routing table entry before it is expunged, i.e., deleted.
ALLOWED_HELLO_LOSS: Maximum loss of anticipated HELLO messages before the link to that node is considered to be broken.
BLACKLIST_TIMEOUT: Timeout for nodes that are part of this node's RREQ "blacklist".
HELLO_INTERVAL: Interval between broadcasts of HELLO messages.
LOCAL_ADD_TTL: Used in the calculation of TTL values for RREQs during local repair.
MAX_REPAIR_TTL: Maximum number of hops to a destination for which local repair is to be performed.

MY_ROUTE_TIMEOUT: Time period for which a route, advertised in a RREP message, is valid.
NET_DIAMETER: The maximum diameter of the network. This value will be used as an upper bound for RREQ TTL values.
NEXT_HOP_WAIT: Time to wait for transmission by a next-hop node when attempting to utilize passive acknowledgements (i.e., overhearing of network traffic).
NODE_TRAVERSAL_TIME: Conservative, estimated average of the one-hop traversal time for packets.
NET_TRAVERSAL_TIME: Estimated time that it takes for a packet to traverse the network.
PATH_TRAVERSAL_TIME: Time used for buffering RREQs and waiting for RREPs.
RREQ_RETRIES: Maximum number of RREQ transmission retries to find a route.
TTL_INCREMENT: Increment of the TTL value in each pass of expanding ring search.
TTL_THRESHOLD: When the TTL value for RREQs in expanding ring search has passed this value, it should be set to NET_DIAMETER (instead of continuing the stepwise increment).
K: This constant should be set according to characteristics of the underlying link layer. A node is assumed to invariably receive at least one out of K subsequent HELLO messages from a neighbor if the link is working and the neighbor is sending no other traffic.

routing_table.{c, h}

This module contains routing table functionality. It defines the data types for routing table entries and precursors, both containing pointers to another element of the same type. That way, entries can form a linked list. Each routing table entry contains the following information:

Destination IP Address
Destination Sequence Number
Interface (network interface index)
Hop Count
Last Hop Count
Next Hop

A list of precursors
Lifetime
Routing Flags (forward route, reverse route, neighbor, uni-directional and local repair)
A timer associated with the entry
RREP-ACK timer for the destination
HELLO timer
Last HELLO time
HELLO count
A hash value (for quickly locating the entry)
A pointer to subsequent routing table entries
The operations offered by this module are the initialization and clean-up of the routing table, route addition, route modification, route timeout modification, route lookup, active route lookup, route invalidation, route deletion, precursor addition and precursor removal. As routes are added, updated, invalidated or deleted, the kernel routing table of the system is updated accordingly. [02]

seek_list.{c, h}

This module contains management of the seeking list, i.e., the list of destinations for which AODVd is seeking routes. The seeking list is a linked list of entries, each containing the following information:

The Destination IP address
The Destination Sequence Number
Flags (used for resending RREQs)
Number of RREQs issued
The TTL value to use for RREQs
IP data (for generating an ICMP Destination Host Unreachable message to the application if route discovery fails)
A seeking timer
A pointer to subsequent seeking list entries
Entries may be added to or removed from the seeking list as needed. Seeking list entries can also be searched for, by specifying their destination IP addresses. [02]

timer_queue.{c, h}

This module contains the timer functionality of AODVd. It defines a data type for timers, containing an expiry time, a pointer to a handler function, a pointer to data (used by the handler function), a boolean value indicating timer usage, and a pointer to other timers. [xx]

Timers are added to a timer queue, represented by a linked list. This list is sorted, with the timer that will expire at the head of the list. The timer queue can be "aged", i.e., checked for expired timers. During aging, the handler functions of expired timers are called in sequence, with the specified data pointer (e.g. a pointer to a routing table entry) being passed as an argument. This allows the handler functions to be context-aware. Aging of the timer queue also updates the select() timeout used by AODVd while waiting for incoming messages on sockets. The new timeout of this select() timer is taken from the timer at the head of the timer queue. [02]

b. AODV kernel modules

Kernel module source files locate under folder lnx, includes: kaodv-debug.c, kaodv-expl.c, kaodv-mod.c, kaodv-queue.c, kaodv-netlink.c

AODVd kernel module code to hook into kernel's networking stack at netfilter locations - IP_PRE_ROUTING, IP_LOCAL_OUT, IP_POST_ROUTING. With this approach, AODV daemon may determine when to trigger the route discovery request as of the nature of an on-demand routing protocol. All network interface inbound packets are redirected to AODV kernel code and queued for user-space processing. Details of kernel module operation can be found in later section 3.2.

c. AODV-Zebra modules

For the goal to integrate AODVd into Quagga, certain tasks need to be done

- Maintain the connectivity (communication channel) between AODVd and Quagga.
- Redistribute AODVd route entry actions (Add, update, remove) to Quagga.

AODV-Zebra software modules will handle for creating and maintaining the communication channel between the two processes.

Zebra_prefix.{c, h}

This module contains the data structures to represent for IP address that used in Quagga framework called prefix. This prefix data structure supports for both IPv4 and IPv6.

Zebra_network.{c, h}

Quagga provide a method for zebra to communicate with other routing daemons via UNIX SOCKET (zserv.api). This module contains the functions for reading and writing data from a UNIX socket to a pointer for later processing.

Zebra_stream.{c, h}

Zebra Protocol is used by protocol daemons to communicate with the zebra daemon. Each protocol daemon may request and send information to and from the zebra daemon such as interface states, routing state, nexthop-validation, and so on. Protocol daemons may also install routes with zebra. The zebra daemon manages which route is installed into the forwarding table with the kernel. Zebra Protocol is a streaming protocol with a common header plus the data portion (zebra packet). This module contains a generic data structure for stream data and functions to build up a zebra packet as well as manipulate incoming stream data from zebra.

Zebra_buffer.{c, h}

Stream data from different routing daemons is stored in a buffer data structure before sending to or receiving from zebra. This buffer software module contains the data structure for such buffer and functions to handle buffer operation.

Zebra.{c, h}

This software module contains functions and data structures that routing daemon can utilize to connect/disconnect with zebra. It also contains important default value for zebra UNIX SOCKET path, zebra packet header size, zebra message types and message flags while communicating with zebra. Also in here, we define functions to announce zebra to manage (add/delete) kernel route table, functions to receive interface state and changes.

2.2.2. Main data structure

a. ***Structure of hello_timer***: definition at line 40 of aodv_hello.c

This data structure declare structure of hello message

*static struct timer hello_timer; /*declare hello time*/*

hello_start(), *hello_stop()*, *hello_send()*, *hello_update_timeout()* function will call it to define hello time.

And *struct of timer*: definition at line 37 of timer_queue.h

```
struct timer {
list_t l;
int used;
struct timeval timeout; /*declare time interval for our timer */
timeout_func_t handler; /*function to call when the timer expires*/
void *data; /*data to pass to the handler function*/ };
```

In this structure: timeout declare time interval, handler function call when the timer expires, data define data to pass to the handler.

b. ***Structure of aodv_rerr***: definition at line 35 of aodv_rerr.h

This structure represents information of RERR message and they will be called by rerr_create() to create RERR message in aodv_rerr.c

```
struct {
u_int8_t type; /*message type*/
#if defined(__LITTLE_ENDIAN)
u_int8_t res1:7;
u_int8_t n:1;
#elif defined(__BIG_ENDIAN)
u_int8_t n:1;
u_int8_t res1:7;
#else
#error "Adjust your <bits/endian.h> defines"
#endif
u_int8_t res2; /*reserved*/
u_int8_t dest_count; /*count destination message*/
u_int32_t dest_addr; /*destination address*/
u_int32_t dest_seqno; /*destination sequence number*/ }
```

```
RERR; #define RERR_SIZE sizeof(RERR) /* Extra unreachable destinations... */
typedef struct {
u_int32_t dest_addr; /*destination address*/
u_int32_t dest_seqno; /*destination sequence number*/
} RERR_udest;
```

c. ***Structure of aodv_rrep***: definition at line 37 of aodv_rrep.h

This structure represents information of RREP message and will be called by rrep_create() in aodv_rrep.c to create RREP message, rrep_send() to send RREP message, rrep_forward() to forward RREP message…

```
struct {
u_int8_t type; /*message type*/
#if defined(__LITTLE_ENDIAN)
u_int16_t res1:6;
u_int16_t a:1;
u_int16_t r:1; /*repair flag*/
u_int16_t prefix:5;
u_int16_t res2:3;
#elif defined(__BIG_ENDIAN)
u_int16_t r:1;
u_int16_t a:1;
u_int16_t res1:6;
u_int16_t res2:3;
u_int16_t prefix:5;
#else
#error "Adjust your <bits/endian.h> defines"
#endif
u_int8_t hcnt;
u_int32_t dest_addr; /*destination address*/
u_int32_t dest_seqno; /*destination sequence number*/
u_int32_t orig_addr; /*original address*/
u_int32_t lifetime; /*life time of message*/
} RREP;
#define RREP_SIZE sizeof(RREP) /*Struct of RREP ACK*/
typedef struct {
u_int8_t type; /*message type*/
```

```
u_int8_t reserved; /*reserved*/
} RREP_ack;
```

d. ***Structure of aodv_rreq***: definition at line 39 of aodv_rreq.h

This structure represents information of RREQ message and will be called by rreq_create() to create RREQ message, rreq_send() to send RREQ message, rreq_forward() to forward RREQ message…

```
struct {
u_int8_t type; /*message type*/
#if defined(__LITTLE_ENDIAN)
u_int8_t res1:4;
u_int8_t d:1; /*destination only respond*/
u_int8_t g:1; /*gratuitous RREP flag*/
u_int8_t r:1; /*repair flag*/
u_int8_t j:1; /*join flag (multicast)*/
#elif defined(__BIG_ENDIAN)
u_int8_t j:1; /*join flag (multicast) */
u_int8_t r:1; /*repair flag */
u_int8_t g:1; /*gratuitous RREP flag */
u_int8_t d:1; /*destination only respond */
u_int8_t res1:4;
#else
#error "Adjust your <bits/endian.h> defines"
#endif
u_int8_t res2;
u_int8_t hcnt;/* Distance (in hops) to the destination */
u_int32_t rreq_id; /*route request id*/
u_int32_t dest_addr; /*destination address*/
u_int32_t dest_seqno; /*destination sequence number*/
u_int32_t orig_addr; /*original address*/
u_int32_t orig_seqno; /*original sequence number*/
} RREQ;

#define RREQ_SIZE sizeof(RREQ)

/*A data structure to buffer information about received RREQ's */
struct rreq_record {
```

```
list_t l;
struct in_addr orig_addr; /* Source of the RREQ */
u_int32_t rreq_id; /* RREQ's broadcast ID */
struct timer rec_timer; /*declare receive timer*/
};
```

```
/*Structure of the black list*/
struct blacklist {
list_t l;
struct in_addr dest_addr; /*declare destination address*/
struct timer bl_timer; /*declare black list timer*/
};
```

e. ***Structure of precursor list***: Definition at line 33 of routing_table.h

This structure allocated by precursor_add() in routing_table.c to add some neighbor to precursor list.

```
struct precursor {
list_t l;
struct in_addr neighbor; /*address of neigbor*/
} precursor_t;
```

f. ***Structure of seek_list***: Definition at line 39 of seek_list.h

This structure gives information about seek list and will be called by seek_list_insert() in seek_list.c to add some entry to seek list.

```
/* This is a list of nodes that route discovery are performed for */
typedef struct seek_list {
list_t l;
struct in_addr dest_addr; /*destination address*/
u_int32_t dest_seqno; /*destination sequence number*/
struct ip_data *ipd;
u_int8_t flags; /* The flags we are using for resending the RREQ */
int reqs; int ttl; /*time to live*/
struct timer seek_timer; /*declare time of seek*/
} seek_list_t;
```

g. ***Structure of data for network device dev_info***: Definition at line 88 of defs.h

This structure gives information of device and will be called by aodv_socket_send() in aodv_socket.c to check device.

```
/* Data for a network device */
struct dev_info {
int enabled; /* 1 if struct is used, else 0 */
int sock; /* AODV socket associated with this device */
#ifdef CONFIG_GATEWAY
int psock; /* Socket to send buffered data packets. */
#endif
unsigned int ifindex; /*Interface index*/
char ifname[IFNAMSIZ];
struct in_addr ipaddr; /* The local IP address */
struct in_addr netmask; /* The netmask we use */
struct in_addr broadcast; /*broadcast address*/
};
```

h. ***Structure of host host_info***: Definition at line 101 of defs.h

This structure represents information about host, its latest used sequence number, latest time of broadcast, RREQ ID, Internet gateway mode setting, number of network interfaces…

```
struct host_info {
u_int32_t seqno; /* Sequence number */
struct timeval bcast_time; /* The time of the last broadcast msg sent */
struct timeval fwd_time; /* The time a data packet was last forwarded */
u_int32_t rreq_id; /* RREQ id */
int nif; /* Number of interfaces to broadcast on */
struct dev_info devs[MAX_NR_INTERFACES+1]; /* Add +1 for returning as
"error" in ifindex2devindex. */
};
```

i. ***Structure of route table entries rt_table***: Definition at line 45 of routing_table.h

This contains information about routing table entry and will be used in routing_table.c

```
struct rt_table {
list_t l;
struct in_addr dest_addr; /* IP address of the destination */
u_int32_t dest_seqno; /*destination sequence number*/
unsigned int ifindex; /* Network interface index... */
struct in_addr next_hop; /* IP address of the next hop to the dest */
u_int8_t hcnt; /* Distance (in hops) to the destination */
u_int16_t flags; /* Routing flags */
u_int8_t state; /* The state of this entry */
struct timer rt_timer; /* The timer associated with this entry */
struct timer ack_timer; /* RREP_ack timer for this destination */
struct timer hello_timer; /*define time of hello message*/
struct timeval last_hello_time; /*time of last hello message*/
u_int8_t hello_cnt;
hash_value hash;
int nprec; /* Number of precursors */
list_t precursors;/* List of neighbors using the route */
};
```

j. ***Structure of routing_table***: Definition at line 81 of routing table.h

This structure gives information about routing table. There are: number of entry, number of active entry….

```
struct routing_table {
unsigned int num_entries; /*number entries*/
unsigned int num_active; /*number active entries*/
list_t tbl[RT_TABLESIZE]; /*table entry list*/
};
```

k. ***Structure of AODV zclient***: definition at line 76 of zebra.h

This structure gives information for communication between AODV daemon and Zebra daemon.

```
/* Structure for the zebra client. */
struct zclient
```

```
{
 /* Socket to zebra daemon. */
 int sock;
 /* Flag of communication to zebra is enabled or not.  Default is on.
    This flag is disabled by `no router zebra' statement. */
 int enable;
 /* Connection failure count. */
 int fail;
 /* Input buffer for zebra message. */
 struct stream *ibuf;
 /* Output buffer for zebra message. */
 struct stream *obuf;
 /* Buffer of data waiting to be written to zebra. */
 struct buffer *wb;
 /* Redistribute information. */
 u_char redist_default;
 u_char redist[ZEBRA_ROUTE_MAX];
 /* Redistribute defauilt. */
 u_char default_information;
};
```

The zebra common message header is defined by data structure zserv_headers

```
/* Zserv protocol message header */
struct zserv_header
{
 uint16_t length;
 uint8_t marker;        /* corresponds to command field in old zserv
                         * always set to 255 in new zserv.
                         */
 uint8_t version;
#define ZSERV_VERSION   2
 uint16_t command;
};
```

For route management , we use this zapi_ipv4 data structure to hold information and will then send to zebra for further action (add/delele).

```
/* Zebra IPv4 route message API. */
```

```
struct zapi_ipv4
{
  u_char type;
  u_char flags;
  u_char message;
  safi_t safi;
  u_char nexthop_num;
  struct in_addr **nexthop;
  u_char ifindex_num;
  unsigned int *ifindex;
  u_char distance;
  u_int32_t metric;
};
```

2.3. OLSR implementation

2.3.1. OLSR software module

This section mention about OLSR software modules, this is an updated version of the initial OLSRdq 0.1.18 release to work with Quagga 0.99.22.1. The different between this AODV implementation in section 2.2 is that this OLSR implementation is developed fully from Quagga libraries.

olsr_protocol.h and **olsr_version.h**

The module olsr_protocol.h contains constants and parameters that specified in RFC 3626 for OLSR protocol. OLSR message, message header, hello message data structure are also defined in this module. While module olsr_version.h contains the implementation version information. [07]

olsr_debug.{h,c}

This module contains data structures, parameters, vtysh commands for debug purpose. OLSR debug separated in two types: packet debug (when there are events of reception or error related to OLSR messages – HELLO, MID, TC …) and event debug (when there are events of routing table entries add/update/delete, neighbor sets update, link set update and others). [07]

olsr_dup.{h,c}

This module contains data structures and functionalities to handle olsr duplicate set. Function olsr_dup_add() is called to add a new tuple in duplicate set; olsr_dup_lookup() is used to lookup a duplicate message with a given address and sequence number and function olsr_dup_has_if() return an interface in duplicate set which has a given address. The OLSR default forwarding algorithm is defined in this module. [07]

olsr_linkset.{h,c}

This module handles link set functionalities. A node records a set of “link tuples” what contains the node’s local interface, the correspondent neighbor interface that connects with this node and the link status (symmetric/asymmetric) which defined by a time value. The module has functions to add new link tuple, search for an existing tuple in link set and delete a link tuple when the correspondent link is lost. The link status (symmetric/asymmetric) is updated when node receive information (embedded in control messages) from other nodes. [07]

olsr_neigh.{h,c}

This module handles neighbor set, 2nd hop neighbor and multiple interface association set functionalities. It has functions for initializing task (create new and delete) and searching functions. [07]

olsr_route.{h,c}

This module contains data structures for storing topology and OLSR route information. Topology information base gives OLSR an overview of the whole network topology; TC messages are flooded throughout the network and each node uses TC information to update its topology information base. Function olsr_route_calculation() is called to build (calculate) OLSR routing table. [07]

olsr_mpr.{h,c}

This module handles MPR functionalities. The function olsr_mpr_update_if() is called for MPR calculation, this is a key feature of OLSR protocol to compute and create MPR set. [07]

olsr_interface.{h,c}

This module handles interface functionalities. Zebra interface callback functions also defined here which help OLSR daemon to receive Zebra signal about interface update (interface up/down; interface state up/down; and interface address added/deleted). There are module for handling OLSR daemon vtysh command to enable or disable OLSR routing on a specific interface. [07]

olsr_packet.{h,c}

This module handles packet functionalities in OLSR. Function olsr_read() is trigger in a looped manner to helps OLSR daemon listen for control messages. These control message is classified and sub-functions will be called to handle each message types (HELLO, TC, MID). The module also has functions to build and add control messages (HELLO, TC, MID) to a pre-defined FIFO queue .While olsr_write() function is called to send queued message in FIFO or forward message from other nodes. [modified of 07]

olsr_time.{h,c}

This module hanled timer functionalities. [07]

olsr_vty.{h,c}

This module handle OLSR daemon command line interface functionalities. This enable a control method for network administrator to control OLSR routing process as well as to implement a mechanism for saving and loading a configuration file for OLSR to run. [modified of 07]

List of OLSRd commands:

- Router mode commands:

router olsr : enable OLSR routing process
network IF_NAME : enable OLSR routing on interface IF_NAME
olsr willingness <0-7> : setup node's willingness
olsr neighb-hold-time <1-65535> : setup neighbor tuple holding time
olsr dup-hold-time <1-65535> : setup duplicate tuple holding time
olsr top-hold-time <1-65535> : setup topology tuple holding time
olsr mpr-update-time <1-65535> : MPR update time
olsr rt-update-time <1-65535> : Routing table update time

- Interface mode commands:

olsr hello-interval <1-65535> : Update hello emission interval

olsr mid-interval <1-65535>	: Update mid emission interval
olsr tc-interval <1-65535>	: Update tc emission interval

- "show ip olsr":

show ip olsr	: show OSLR information
show ip olsr neighbor	: show neighbor set
show ip olsr linkset	: show link set
show ip olsr topset	: show topology set
show ip olsr mid	: show mid set
show ip olsr hop2	: show 2nd hop set
show ip olsr routes	: show OLSR routing table

olsr_zebra.{h,c}

This module defines a zclient instance for OLSR daemon to communicate with Zebra daemon. Callback functions used for telling Zebra daemon to add or delete kernel route also implemented in this module. So far OLSR daemon only supports IPv4. Details of kernel route update will be mentioned in Methodology chapter. [07]

olsrd.{h,c}

This module handle OLSR daemon functionalities. A OLSR master process is defined here for overall control of OLSR daemon. The node main address is managed by function olsr_main_addr_update(), the master thread for OLSR daemon is initialized when olsr_master_init() is called. [modified of 07]

olsr_main.c

The main program for OLSRd.

Makefile.{am,in}

This module is used for generating Makefile for OLSR daemon. This OLSR implementation developed on Autotool framework (same with Quagga).

olsr.conf.sample

OLSR daemon load this configuration file (olsr.conf) when it is initialized. While running, user or administrator can make change to OLSR daemon parameter and save changes to olsr.conf at any time via vtysh command – “write”; the changes will be applied the next time when OLSR daemon starts. [modified of 07]

2.3.2. Main data structure

a. **Structure of olsr_master()** : defined in olsrd.h

This is the main data structure for OLSR process, it has the master thread for OLSR daemon while communicating with Zebra daemon.

```
struct olsr_master
{
 struct list *olsr;  /* OLSR instance list. */
 struct thread_master *master;  /* OLSR thread master. */
 struct list *iflist;  /* Zebra interface list. */
 time_t start_time;  /* OLSR start time. */
};
```

This master data structure is initialized when OLSR routing process enabled by function olsr_master_init(). The master thread is used for coordinating other OLSR sub-threads which mentioned later in this section.

b. **Structure of olsr()** : defined olsrd.h

This data structure is initialized for when OLSR daemon started. This contains an OLSR node information.

```
struct olsr
{
 struct in_addr main_addr;              /* Main router address. */
 int main_addr_is_set;
 struct list *oiflist;                  /* List of interfaces enable for OLSR.*/

 u_int16_t port;                        /* Olsr port. */
 float C;                               /* OLSR C constant. */
 u_char will;                           /* The node’s willingness.*/
 float neighb_hold_time;                /* Neighbor Hold Time.*/
 float dup_hold_time;                   /* Duplicate Message Hold Time. */
 float top_hold_time;                   /* Topology Hold Time. */
 float mpr_update_time;                 /* Delay updating mpr amount. */
```

```
 float rt_update_time;                  /* Delay updating routing table amount.*/

 /* Global sequence numbers. */
 u_int16_t msn;                        /* Message Sequence Number.          */
 u_int16_t ansn;                       /* Advertised Neighbor Sequence Number.*/

 /* OLSR sets. */
 struct list *dupset;                  /* Duplicate Message set tuples.      */
 struct list *neighset;                /* Neighbour set tuples.              */
 struct list *n2hopset;                /* 2 Hop Neighbour set tuples.        */
 struct list *midset;                  /* Multiple Interface tuples.         */
 struct list *topset;                  /* Topology tuples.                   */
 struct list *advset;                  /* Advertised set of nodes.           */
 struct route_table *networks;         /* OLSR config networks.              */
 struct route_table *table;            /* The routing table.                 */
 /* Threads. */
 struct thread *t_write;
 struct thread *t_read;

 struct thread *t_delay_stc;
 struct thread *t_mpr_update;
 struct thread *t_rt_update;
 int fd;
};
```

The node main address main_addr is updated by olsr_main_addr_update(). Node maintain a list of interfaces that run OLSR routing with oiflist, whenever user/administrator issue vtysh command to enable OLSR on an interface , then oiflist is updated.

Some parameters specified in RFC3626 is maintain by this data structure – OLSR UDP port 654, the C constant, the node’s willingness and hold times for neighbor, duplicate and topology tuple. The time value for MPR and Route table update also stored in this data structure. These parameters are set with default values from olsr_protocol.h and can be updated or changed via vtysh commands.

The node also has the message sequence number and advertised sequence number, these values will be added to OLSR control messages when the node communicate with other OLSR nodes.pa

Node's information base sets are defined as linked list while node's OLSR routing table is defined as struct route_table from Quagga libraries (see details in section 2.4.2).

A sub-thread *t_read is triggered for reading incoming, function olsr_read() is called for such task, I will mention this task more in section 3.2.

Sub-thread *t_delay_stc handles for stopping TC packet generation that specified in RFC3626 when the advertised link set is empty. It is triggered every topology hold time, function olsr_tc_stop() is called to stop TC packet generation.

For every mpr_update_time the sub-thread *t_mpr_update is triggered to compute MPR set. Function olsr_mpr_update_if() is called to compute MPR set for an OLSR enabled interface.

The sub-thread *t_rt_update handle routing table calculation, in which function olsr_route_calculate() is called to perform such task. The thread is triggered for every rt_update_time.

c. **Structure of olsr_interface()** : defined in olsr_interface.h

This structure hold the system interface information. An output buffer (fifo queue) is defined for each interface to store control messages and forwarded message.

```
struct olsr_interface
{
 struct olsr *olsr;             /* Parent olsr instance. */
 struct interface *ifp;         /* Interface data from zebra. */
 struct olsr_fifo *fifo;        /* Packet send buffer. */
 int sock;                      /* Output socket. */
 struct prefix *address;        /* Interface prefix. */
 struct connected *connected;          /* Pointer to connected. */
 u_int16_t psn;                 /* Packet Sequence Number. */
 struct list *linkset;          /* Links connected to this interface. */
```

```
 struct thread *t_hello;          /* Hello timer thread.*/
 struct thread *t_mid;            /* MID timer thread. */
 struct thread *t_tc;             /* TC timer thread. */
 struct thread *t_write;          /* Write to output socket. */
};
```

When an interface is up and has valid IP address configured, and instance of olsr_interface is initialized by function olsr_if_new(). If the interface is configured for running OLSR, function olsr_if_up() is called to create the fifo buffer olsr_fifo_new(),olsr_output_sock_init() will calculate and assign an UDP socket (socket binded to UDP port 654) for this interface. Other timers sub-threads is triggered to create HELLO, TC, and MID messages and store to FIFO. The sub-thread *t_write also triggered periodically to write fifo message to output socket for sending out.

d. **Structure of olsr_msg()** : defined in olsr_packet.h

This is a generic OLSR message data structure

```
struct olsr_msg
{
 struct in_addr dst;              /* Detination IP. */
 struct stream *obuf;             /* Output buffer. */
 struct olsr_msg *next;           /* Next message in FIFO. */
};
```

olsr_build_and_send_hello(), olsr_build_and_send_tc(), olsr_build_and_send_mid() and olsr_forward() functions will fill the output buffer to construct control messages (HELLO, TC, MID), the message size if determined by *obuf size (size of the stream, details of stream structure is in section 2.4.2). Each message will be push into FIFO queue by olsr_msg_fifo_push().

The structure olsr_fifo represents the FIFO queue for each interface in olsr_packet.h.

```
struct olsr_fifo
{
 unsigned long count;
 struct olsr_msg *head;
 struct olsr_msg *tail;
```

```
};
```

Periodically, the OLSR interface sub-thread t_write is triggered to send all messages in FIFO queue to physical buffer and from there sending out.

e. **Structure of olsr_dup()** : defined in olsr_dup.h

```
struct olsr_dup
{
 struct in_addr addr;            /* D_addr. */
 u_int16_t msn;                  /* D_seq_num. */
 u_char retransmitted;           /* D_retransmitted. */
 struct list *iface_list;        /* D_iface_list. */
 struct thread *t_time;          /* D_time. */
 struct listnode *node;
 struct olsr *olsr;
};
```

This structure represents for OLSR duplicate tuple which is a member of the node's duplicate set (a linked list). Function olsr_dup_add() is called to add a new duplicate tuple to set. olsr_dup_lookup() and olsr_dup_has_if() functions used for search duplicate set. For every duplicate hold time value (duplicate tuple is considered expired), olsr_dup_del() is called to delete the tuple from set.

f. **Structure of olsr_link()** : defined in olsr_linkset.h

This data structure represents a link tuple, a member of the node's link set (a linked list)

```
struct olsr_link
{
 struct olsr_interface *oi;          /* Associated interface. */
 struct in_addr neigh_addr;          /* Link. */
 u_char sym_expired;
 u_char asym_expired;
 struct olsr_neigh *neigh;           /* Associated neighbor. */
 struct thread *t_sym_time;          /* Symetric Timers. */
 struct thread *t_asym_time;         /* Asymetric Timers. */
 struct thread *t_time;              /* Symetric Timers. */
 struct listnode *node;
```

```
};
```

Functions olsr_linkset_add() , olsr_linkset_search() and olsr_linkset_del() are used for link tuple intializing and searching. The link status (symetric, asymetric) is updated when node receives control messages; olsr_sym_expired() and olsr_asym_expired() functions are called to set link status.

g. **Structure of olsr_neigh()** : defined in olsr_neigh.h

This structure represents neighbor tuple, a member of node's neighbor set (linked list).

```
struct olsr_neigh
{
 struct in_addr main_addr;
 u_char status;
 u_char will;
 u_char is_mpr;
 u_char is_mprs;
 struct thread *t_mprs;                    /* Expire time for mprs. */
 struct list *assoc_links;
 struct listnode *node;
 struct olsr *olsr;
};
```

While receiving control messages, the node create its neighbor set that contains all neighbors with their status (willingness, main address). Parameters is_mpr and is_mprs determine whether a neighbor is this node's MPR or is in MRP Selector set.

While a neighbor is still in MRP Selector Set, it is added to this node's Avertised set with function olsr_top_add_adv(). Sub-thread *t_mprs is triggered when a neighbor no-longer in this node's MPR Selector set, function olsr_top_rem_adv() to remove neighbor from Advertised set.

h. **Structure of olsr_top()** : defined in olsr_route.h

This structure represents a topology tuple in node's topology set (linked list). A topology tuple is created for each destination in the network, destination parameteris also recorded – destination node main address dest_addr

```
struct olsr_top
{
  struct in_addr dest_addr;
  struct in_addr last_addr;
  u_int16_t seq;
  struct thread *t_time;
  struct listnode *node;
  struct olsr *olsr;
};
```

Sub-thread *t_time is triggered when a topology tuple expired and need to remove from topology set by function olsr_top_del(). Function olsr_top_add() is called to add new tuple to topology set while olsr_top_lookup() and olsr_top_exists_newer() perform search tasks of a topology set.

i. **Structure of olsr_route() :** defined in olsr_route.h

This structure represents an OLSR route entry which is a member of OLSR route table (a linked list)

```
struct olsr_route
{
  struct in_addr next_hop;
  int dist;
  struct in_addr iface_addr;
};
```

OLSR daemon hold a route update timer to refresh the route table. Function olsr_rt_update() is called when node has updates (neighbor or 2-hop neighbor changes, or multiple interface association set changes) and olsr_rt_timer() is called to t a new routing table to replace the current one.

The OLSRdq maintains its own routing table which is synchronized with kernel routing table via Zebra daemon. More details of the synchronization between OLSRdq routing table and kernel routing table can be found later in chapter 3 Methodology.

2.4. Quagga implementation

2.4.1. An overview of Quagga

Quagga is a routing software package that provides TCP/IP based routing services with routing protocols support such as RIPv1, RIPv2, RIPng, OSPFv2, OSPFv3, IS-IS, BGP-4, and BGP-4+. Quagga also supports special BGP Route Reflector and Route Server behaviour. In addition to traditional IPv4 routing protocols, Quagga also supports IPv6 routing protocols. With SNMP daemon which supports SMUX and AgentX protocol, Quagga provides routing protocol MIBs. Quagga is a fork of Zebra routing suite which developed by Kunihiro Ishiguro, the project is distributed under GNU General Public License. The project is currently active and it was chosen to be the routing protocol stack for Vyatta Core software (award-winning open source network operating system providing advanced IPv4 and IPv6 routing, stateful firewalling, IPSec and SSL OpenVPN, and more) since Vyatta 4.0 release (Glendale). [8]

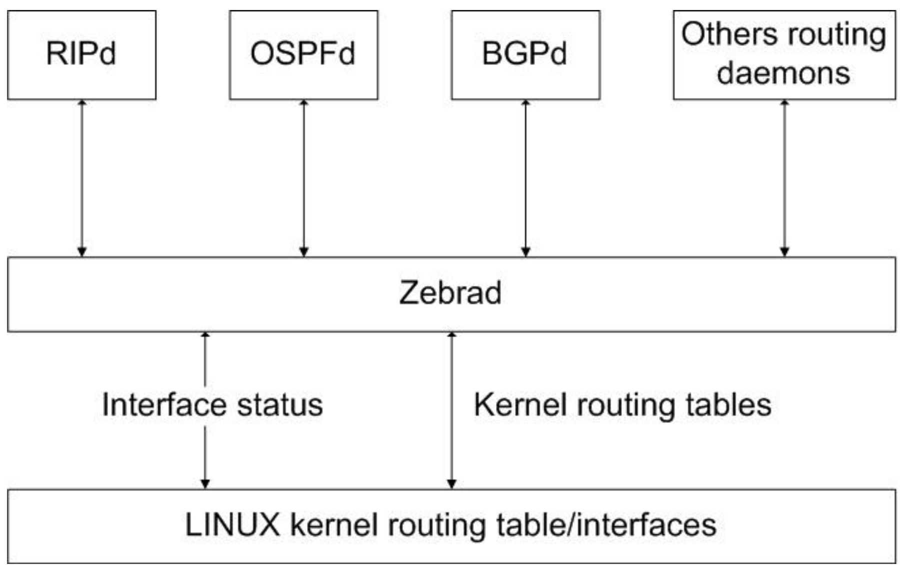


Figure 5: Quagga system architecture

Traditional routing software is made as a one process program which provides all of the routing protocol functionalities. Quagga takes a different approach. It is made from a collection of several daemons that work together to build the routing table. There may be several protocol-specific routing daemons and zebra the kernel routing manager. [8]

The ripd daemon handles the RIP protocol, while ospfd is a daemon which supports OSPF version 2. bgpd supports the BGP-4 protocol. For changing the kernel routing table and for redistribution of routes between different routing protocols, there is a kernel routing table manager zebra daemon. It is easy to add a new routing protocol daemons to the entire routing system without affecting any other software. You need to run only the protocol daemon associated with routing protocols in use. Thus, user may run a specific daemon and send routing reports to a central routing console. Zebra daemon also tell routing protocol daemons for interface status changing, interface IP address update. [8]

There is no need for these daemons to be running on the same machine. You can even run several same protocol daemons on the same machine. This architecture creates new possibilities for the routing system. [8]

Multi-process architecture brings extensibility, modularity and maintainability. At the same time it also brings many configuration files and terminal interfaces. Each daemon has its own configuration file and terminal interface. When you configure a static route, it must be done in zebra configuration file. When you configure BGP network it must be done in bgpd configuration file. This can be a very annoying thing. To resolve the problem, Quagga provides integrated user interface shell called vtysh. vtysh connects to each daemon via UNIX domain socket and then works as a proxy for user input. [8]

In following sections, I mention main (and important) zebra source files, data structures, functions, etc …, that will be used in the implementation of AODV and OLSR daemon.

2.4.2. Quagga libraries

Quagga package comes with rich libraries (data structures, function calls, etc…) that serves for both zebra daemon and user-defined routing daemon. These libraries includes list structures, checksum calculation, log file processing, system signalling handling, virtual terminal interface handling (VTY), customize memory allocation and others. Source files for quagga libraries can be found in lib folder of the package source.

When implement a routing protocol daemon, we can utilize these libraries to our code content. I also mention some parameters defined in Quagga libraries that served for the integration of AODVd and OLSRdq into Quagga 0.99.22.1.

a. Linked list

An implementation of generic linked list can be found in files lib/linklist.{h, c}

```
/* listnodes must always contain data to be valid. Adding an empty node
 * to a list is invalid
 */
struct listnode
{
```

```
 struct listnode *next;
 struct listnode *prev;

 /* private member, use getdata() to retrieve, do not access directly */
 void *data;
};

struct list
{
 struct listnode *head;
 struct listnode *tail;

 /* invariant: count is the number of listnodes in the list */
 unsigned int count;

 /*
  * Returns -1 if val1 < val2, 0 if equal?, 1 if val1 > val2.
  * Used as definition of sorted for listnode_add_sort
  */
 int (*cmp) (void *val1, void *val2);

 /* callback to free user-owned data when listnode is deleted. supplying
  * this callback is very much encouraged!
  */
 void (*del) (void *val);
};
```

And some macros for using:

```
#define listnextnode(X) ((X) ? ((X)->next) : NULL)
#define listhead(X) ((X) ? ((X)->head) : NULL)
#define listtail(X) ((X) ? ((X)->tail) : NULL)
#define listcount(X) ((X)->count)
#define list_isempty(X) ((X)->head == NULL && (X)->tail == NULL)
#define listgetdata(X) (assert((X)->data != NULL), (X)->data)
```

Prototypes of linked list

```
extern struct list *list_new(void); /*encouraged:set list.del callback on new lists */
extern void list_free (struct list *);
extern void listnode_add (struct list *, void *);
extern void listnode_add_sort (struct list *, void *);
extern void listnode_add_after (struct list *, struct listnode *, void *);
extern void listnode_delete (struct list *, void *);
```

```
extern struct listnode *listnode_lookup (struct list *, void *);
extern void *listnode_head (struct list *);
extern void list_delete (struct list *);
extern void list_delete_all_node (struct list *);
```

For list iteration, zebra offers two methods

ALL_LIST_ELEMENTS(list,node,nextnode,data). It is safe to delete the listnode using this method. Here the macro definition for ALL_LIST_ELEMENTS

```
#define ALL_LIST_ELEMENTS(list,node,nextnode,data) \
  (node) = listhead(list), ((data) = NULL); \
  (node) != NULL && \
    ((data) = listgetdata(node),(nextnode) = node->next, 1); \
  (node) = (nextnode), ((data) = NULL)
```

ALL_LIST_ELEMENTS_RO(list,node,data). This is a read-only list iteration method and can be used for traverse the list in loop manner. The definition is as follow:

```
#define ALL_LIST_ELEMENTS_RO(list,node,data) \
  (node) = listhead(list), ((data) = NULL);\
  (node) != NULL && ((data) = listgetdata(node), 1); \
  (node) = listnextnode(node), ((data) = NULL)
```

b. Route entry and route table data structure.

An implementation of route entry and routing table can be found in lib/table.{h,c}. Note that this data structure and functions are used to store user-defined routing protocol and zebra daemon route only and not the actual kernel route table. The synchronization between zebra daemon and user-defined routing protocol daemons route table will be discuss further in Chapter 3 when doing the integration AODV and OLSR routing daemon into Quagga. Note that this route data structure is only used for routing protocol daemon route not the kernel routing table. How Zebra daemon updates the kernel routing table will be mentioned in section 2.4.4.

Some prototypes :

```
extern struct route_table *route_table_init (void);
extern struct route_table *route_table_init_with_delegate(route_table_delegate_t
*);
```

```
extern void route_table_finish (struct route_table *);
extern void route_unlock_node (struct route_node *node);
extern struct route_node *route_top (struct route_table *);
extern struct route_node *route_next (struct route_node *);
extern struct route_node *route_next_until (struct route_node *,
                                            struct route_node *);
extern struct route_node *route_node_get (struct route_table *const,
                                          struct prefix *);
extern struct route_node *route_node_lookup (const struct route_table *,
                                             struct prefix *);
extern struct route_node *route_lock_node (struct route_node *node);
extern struct route_node *route_node_match (const struct route_table *,
                                            const struct prefix *);
extern struct route_node *route_node_match_ipv4 (const struct route_table *,
                                                  const struct in_addr *);
#ifdef HAVE_IPV6
extern struct route_node *route_node_match_ipv6 (const struct route_table *,
                                                  const struct in6_addr *);
#endif /* HAVE_IPV6 */
```

And iteration functions for route table structure

```
extern void route_table_iter_init (route_table_iter_t *iter,
                                   struct route_table *table);
extern void route_table_iter_pause (route_table_iter_t *iter);
extern void route_table_iter_cleanup (route_table_iter_t *iter);
```

c. *Interface and connected address data structure and functions*

Source files in library: lib/if.{h,c}. Interface structure is used to store an machine interface information. Connected address structure is used hold an address with its correspondent interface index.

```
/* Interface structure */
struct interface
{
  char name[INTERFACE_NAMSIZ + 1];
  unsigned int ifindex;
#define IFINDEX_INTERNAL        0
  /* Zebra internal interface status */
  u_char status;
#define ZEBRA_INTERFACE_ACTIVE     (1 << 0)
#define ZEBRA_INTERFACE_SUB        (1 << 1)
```

```
#define ZEBRA_INTERFACE_LINKDETECTION (1 << 2)
  /* Interface flags. */
  uint64_t flags;
  /* Interface metric */
  int metric;
  /* Interface MTU. */
  unsigned int mtu;    /* IPv4 MTU */
  unsigned int mtu6;   /* IPv6 MTU - probably, but not necessarily same as mtu */
  /* Hardware address. */
#ifdef HAVE_STRUCT_SOCKADDR_DL
  union {
    /* note that sdl_storage is never accessed, it only exists to make space.
     * all actual uses refer to sdl - but use sizeof(sdl_storage)!  this fits
     * best with C aliasing rules. */
    struct sockaddr_dl sdl;
    struct sockaddr_storage sdl_storage;
  };
#else
  unsigned short hw_type;
  u_char hw_addr[INTERFACE_HWADDR_MAX];
  int hw_addr_len;
#endif /* HAVE_STRUCT_SOCKADDR_DL */

  /* interface bandwidth, kbits */
  unsigned int bandwidth;
  /* description of the interface. */
  char *desc;
  /* Distribute list. */
  void *distribute_in;
  void *distribute_out;

  /* Connected address list. */
  struct list *connected;
  /* Daemon specific interface data pointer. */
  void *info;
  /* Statistics fileds. */
#ifdef HAVE_PROC_NET_DEV
  struct if_stats stats;
#endif /* HAVE_PROC_NET_DEV */
#ifdef HAVE_NET_RT_IFLIST
  struct if_data stats;
#endif /* HAVE_NET_RT_IFLIST */
};
```

```
/* Connected address structure. */
struct connected
{
  /* Attached interface. */
  struct interface *ifp;
  /* Flags for configuration. */
  u_char conf;
#define ZEBRA_IFC_REAL         (1 << 0)
#define ZEBRA_IFC_CONFIGURED   (1 << 1)
  /*
     The ZEBRA_IFC_REAL flag should be set if and only if this address
     exists in the kernel.
     The ZEBRA_IFC_CONFIGURED flag should be set if and only if this address
     was configured by the user from inside quagga.
   */
  /* Flags for connected address. */
  u_char flags;
#define ZEBRA_IFA_SECONDARY    (1 << 0)
#define ZEBRA_IFA_PEER         (1 << 1)
  /* N.B. the ZEBRA_IFA_PEER flag should be set if and only if
     a peer address has been configured.  If this flag is set,
     the destination field must contain the peer address.
     Otherwise, if this flag is not set, the destination address
     will either contain a broadcast address or be NULL.
   */
  /* Address of connected network. */
  struct prefix *address;
  /* Peer or Broadcast address, depending on whether ZEBRA_IFA_PEER is set.
     Note: destination may be NULL if ZEBRA_IFA_PEER is not set. */
  struct prefix *destination;
  /* Label for Linux 2.2.X and upper. */
  char *label;
};
```

Prototypes and functions for interface data structure

```
extern int if_cmp_func (struct interface *, struct interface *);
extern struct interface *if_create (const char *name, int namelen);
extern struct interface *if_lookup_by_index (unsigned int);
extern struct interface *if_lookup_exact_address (struct in_addr);
extern struct interface *if_lookup_address (struct in_addr);

/* These 2 functions are to be used when the ifname argument is terminated
```

```
   by a '\0' character: */
extern struct interface *if_lookup_by_name (const char *ifname);
extern struct interface *if_get_by_name (const char *ifname);
extern void if_delete (struct interface *);
extern int if_is_up (struct interface *);
extern int if_is_running (struct interface *);
extern int if_is_operative (struct interface *);
extern int if_is_loopback (struct interface *);
extern int if_is_broadcast (struct interface *);
extern int if_is_pointopoint (struct interface *);
extern int if_is_multicast (struct interface *);
extern void if_add_hook (int, int (*)(struct interface *));
extern void if_init (void);
extern void if_terminate (void);
extern void if_dump_all (void);
extern const char *if_flag_dump(unsigned long);
```

Prototypes and functions for connected address data structure :

```
extern struct connected *connected_new (void);
extern void connected_free (struct connected *);
extern void connected_add (struct interface *, struct connected *);
extern struct connected  *connected_add_by_prefix (struct interface *,
                                                   struct prefix *,
                                                   struct prefix *);
extern struct connected  *connected_delete_by_prefix (struct interface *,
                                                      struct prefix *);
extern struct connected  *connected_lookup_address (struct interface *,
                                                    struct in_addr);
```

d. Memory handle

The library offers certain data structure and methods for us to handle memory when coding. For a user-define routing protocol implementation, we have to register the memory type in memtype source files. With these declarations, we can use zebra library functions to allocate / free system memory for user-defined data structure while implementing a routing protocol. Source files in lib/memtypes.{h,c}.

Some prototypes of memory functions can be found in lib/memory.{h,c}

```
/* Prototypes of memory function. */
extern void *zmalloc (int type, size_t size);
```

```
extern void *zcalloc (int type, size_t size);
extern void *zrealloc (int type, void *ptr, size_t size);
extern void  zfree (int type, void *ptr);
extern char *zstrdup (int type, const char *str);
```

And macros for memory functions

```
#define XMALLOC(mtype, size)       zmalloc ((mtype), (size))
#define XCALLOC(mtype, size)       zcalloc ((mtype), (size))
#define XREALLOC(mtype, ptr, size) zrealloc ((mtype), (ptr), (size))
#define XFREE(mtype, ptr)          do { \
                                     zfree ((mtype), (ptr)); \
                                     ptr = NULL; } \
                                   while (0)
#define XSTRDUP(mtype, str)        zstrdup ((mtype), (str))
```

e. Thread handling.

Zebra and routing daemons are designed to run as thread. The library offers thread data structure, functions, macros for purposes such as : schedule for reading incoming packet, schedule for writing built packet to interface output buffer, setup timers in routing protocol , etc… Source files in: lib/thread.{h,c}

Some main prototypes, functions and macros

```
extern struct thread_master *thread_master_create (void);
extern void thread_master_free (struct thread_master *);
/* funcname_thread_add_read is usally used to schedule (or loop) a 'read' action.
Eg: read packets that being send/received on an interface */
extern struct thread *funcname_thread_add_read (struct thread_master *,
                                                int (*)(struct thread *),
                                                void *, int, const char*);
/* funcname_thread_add_write is usally used to schedule (or loop) a 'write'
action. Eg: write packets in buffer to physical interface buffer for sending */
extern struct thread *funcname_thread_add_write (struct thread_master *,
                                                 int (*)(struct thread *),
                                                 void *, int, const char*);
/* These below two thread_add_timer is usally used to initial a timer */
extern struct thread *funcname_thread_add_timer (struct thread_master *,
                                                 int (*)(struct thread *),
                                                 void *, long, const char*);
extern struct thread *funcname_thread_add_timer_msec (struct thread_master *,
                                                      int (*)(struct thread *),
```

```
                                                    void *, long, const char*);
/* macros to simplify usage of thread functions */
#define thread_add_read(m,f,a,v) funcname_thread_add_read(m,f,a,v,#f)
#define thread_add_write(m,f,a,v) funcname_thread_add_write(m,f,a,v,#f)
#define thread_add_timer(m,f,a,v) funcname_thread_add_timer(m,f,a,v,#f)
#define                                         thread_add_timer_msec(m,f,a,v)
funcname_thread_add_timer_msec(m,f,a,v,#f)
```

f. Stream handling

A stream is an arbitrary buffer, whose contents generally are assumed to be in network order. A stream has the following attributes associated with it:

- size: the allocated, invariant size of the buffer.

- getp: the get position marker, denoting the offset in the stream where the next read (or 'get') will be from. This getp marker is automatically adjusted when data is read from the stream, the user may also manipulate this offset as they wish, within limits.

- endp: the end position marker, denoting the offset in the stream where valid data ends, and if the user attempted to write (or 'put') data where that data would be written (or 'put') to.

These attributes are all size_t values.

Stream constraints: getp <= endp <= size

Routing protocol daemons and zebra daemon will construct communication message in stream format. These messages are communicate using Zebra protocol which will be mentioned in section 2.4.3.

Stream data structure

```
/* Stream buffer. */
struct stream
{
  struct stream *next;
  /* Remainder is ***private*** to stream
   * direct access is frowned upon!
   * Use the appropriate functions/macros
   */
  size_t getp;              /* next get position */
```

```
 size_t endp;              /* last valid data position */
 size_t size;              /* size of data segment */
 unsigned char *data; /* data pointer */
};
```

Stream functions which can be categorized in to

- Common functions: create new stream, free memory, copy stream to stream, duplicate a stream and resize
- Stream pointer functions: to get the pointers, set pointer to new position, etc…)
- Stream put actions: to put data to stream in terms of data size
- Stream get actions: to get data from stream in terms of data size.

```
/* Stream prototypes.
 * For stream_{put,get}S, the S suffix mean:
 *
 * c: character (unsigned byte)
 * w: word (two bytes)
 * l: long (two words)
 * q: quad (four words)
 */

/* Common functions */
extern struct stream *stream_new (size_t);
extern void stream_free (struct stream *);
extern struct stream * stream_copy (struct stream *, struct stream *src);
extern struct stream *stream_dup (struct stream *);
extern size_t stream_resize (struct stream *, size_t);

/* Stream pointers functions */
extern size_t stream_get_getp (struct stream *);
extern size_t stream_get_endp (struct stream *);
extern size_t stream_get_size (struct stream *);
extern u_char *stream_get_data (struct stream *);
extern void stream_set_getp (struct stream *, size_t);
extern void stream_set_endp (struct stream *, size_t);
extern void stream_forward_getp (struct stream *, size_t);
extern void stream_forward_endp (struct stream *, size_t);

/* Stream put functions. Note: steam_put: NULL source zeroes out size_t bytes of
stream */
```

```
extern void stream_put (struct stream *, const void *, size_t);
extern int stream_putc (struct stream *, u_char);
extern int stream_putc_at (struct stream *, size_t, u_char);
extern int stream_putw (struct stream *, u_int16_t);
extern int stream_putw_at (struct stream *, size_t, u_int16_t);
extern int stream_putl (struct stream *, u_int32_t);
extern int stream_putl_at (struct stream *, size_t, u_int32_t);
extern int stream_putq (struct stream *, uint64_t);
extern int stream_putq_at (struct stream *, size_t, uint64_t);
extern int stream_put_ipv4 (struct stream *, u_int32_t);
extern int stream_put_in_addr (struct stream *, struct in_addr *);
extern int stream_put_prefix (struct stream *, struct prefix *);

/* Stream get functions */
extern void stream_get (void *, struct stream *, size_t);
extern u_char stream_getc (struct stream *);
extern u_char stream_getc_from (struct stream *, size_t);
extern u_int16_t stream_getw (struct stream *);
extern u_int16_t stream_getw_from (struct stream *, size_t);
extern u_int32_t stream_getl (struct stream *);
extern u_int32_t stream_getl_from (struct stream *, size_t);
extern uint64_t stream_getq (struct stream *);
extern uint64_t stream_getq_from (struct stream *, size_t);
extern u_int32_t stream_get_ipv4 (struct stream *);
```

g. Zebra client data structure

Each routing daemon has to implement a zclient in order to communication with zebra daemon. With this, zebra will tell (announce) routing daemons for any system update: new interface added/deleted, interface up/down, new IP address added/deleted and others. The data structure for zclient can be found in lib/zclient.{h,c}.

```
/* Structure for the zebra client. */
struct zclient
{
  /* Socket to zebra daemon. */
  int sock;
  /* Flag of communication to zebra is enabled or not.  Default is on.
     This flag is disabled by `no router zebra' statement. */
  int enable;
  /* Connection failure count. */
```

```
 int fail;
 /* Input buffer for zebra message. */
 struct stream *ibuf;
 /* Output buffer for zebra message. */
 struct stream *obuf;
 /* Buffer of data waiting to be written to zebra. */
 struct buffer *wb;
 /* Read and connect thread. */
 struct thread *t_read;
 struct thread *t_connect;
 /* Thread to write buffered data to zebra. */
 struct thread *t_write;
 /* Redistribute information. */
 u_char redist_default;
 u_char redist[ZEBRA_ROUTE_MAX];
 /* Redistribute defauilt. */
 u_char default_information;
 /* Pointer to the callback functions. */
 int (*router_id_update) (int, struct zclient *, uint16_t);
 int (*interface_add) (int, struct zclient *, uint16_t);
 int (*interface_delete) (int, struct zclient *, uint16_t);
 int (*interface_up) (int, struct zclient *, uint16_t);
 int (*interface_down) (int, struct zclient *, uint16_t);
 int (*interface_address_add) (int, struct zclient *, uint16_t);
 int (*interface_address_delete) (int, struct zclient *, uint16_t);
 int (*ipv4_route_add) (int, struct zclient *, uint16_t);
 int (*ipv4_route_delete) (int, struct zclient *, uint16_t);
 int (*ipv6_route_add) (int, struct zclient *, uint16_t);
 int (*ipv6_route_delete) (int, struct zclient *, uint16_t);
};
```

Each routing daemon implement a zclient instance and correspondent callback functions to communicate with zebra. Callback functions are in two groups

- Interface callback functions: to receive zebra's signal for new added/deleted interfaces, interface state change (up/down), and for interface address changes.

- Route entry callback functions: to tell zebra to add/delete a Ipv4/Ipv6 route entry.

2.4.3. Zebra protocol

Zebra Protocol is used by protocol daemons to communicate with the zebra daemon. Each protocol daemon may request and send information to and from the zebra daemon such as interface states, routing state, nexthop-validation, and so on. Protocol daemons may also install routes with zebra. The zebra daemon manages which route is installed into the forwarding table with the kernel. [8]

Zebra Protocol is a streaming protocol, with a common header. Two versions of the header are in used. Version 0 is and implicitly versioned. Version 1 has an explicit version field. Version 0 can be distinguished from all other versions by examining the 3rd byte of the header, which contains a marker value for all versions bar version 0. The marker byte corresponds to the command field in version 0, and the marker value is a reserved command in version 0. [8]

Version 1 is used within this writing for integrating AODVd and OLSRd to Quagga 0.99.22.1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+---------------+-------------+
|           Length (2)          |   Marker (1)  | Version (1) |
+-------------------------------+---------------+-------------+
|          Command (2)          |
+-------------------------------+
```

Figure 6: Zebra protocol common header version 1 [08]

‘Length’ : Total packet length including this header. The minimum length is 3 bytes for version 0 messages and 6 bytes for version 1 messages.

‘Marker’ : Static marker with a value of 255 always. This is to allow version 0 Zserv headers (which do not include version explicitly) to be distinguished from versioned headers. Not present in version 0 messages.

‘Version’: Version number of the Zserv message. Clients should not continue processing messages past the version field for versions they do not recognize. Not present in version 0 messages.

‘Command’: The Zebra Protocol command. [08]

List of Zebra Protocol command is specified in follow table

Command	Value
ZEBRA_INTERFACE_ADD	1
ZEBRA_INTERFACE_DELETE	2
ZEBRA_INTERFACE_ADDRESS_ADD	3
ZEBRA_INTERFACE_ADDRESS_DELETE	4
ZEBRA_INTERFACE_UP	5
ZEBRA_INTERFACE_DOWN	6
ZEBRA_IPV4_ROUTE_ADD	7
ZEBRA_IPV4_ROUTE_DELETE	8
ZEBRA_IPV6_ROUTE_ADD	9
ZEBRA_IPV6_ROUTE_DELETE	10
ZEBRA_REDISTRIBUTE_ADD	11
ZEBRA_REDISTRIBUTE_DELETE	12
ZEBRA_REDISTRIBUTE_DEFAULT_ADD	13
ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	14
ZEBRA_IPV4_NEXTHOP_LOOKUP	15
ZEBRA_IPV6_NEXTHOP_LOOKUP	16

Table 1: List of Zebra protocol command [08]

Kernel routing table updating is done via zebra APIs (zapi), routing daemons need to implement correspondent API to perform adding new route to kernel, delete or update kernel route, redistribute zebra routes from other routing daemons.

```
/* Zebra IPv4 route message API. */
struct zapi_ipv4
{
  u_char type;
```

```
  u_char flags;
  u_char message;
  safi_t safi;
  u_char nexthop_num;
  struct in_addr **nexthop;
  u_char ifindex_num;
  unsigned int *ifindex;
  u_char distance;
  u_int32_t metric;
};

/* IPv6 prefix add and delete function prototype. */
struct zapi_ipv6
{
  u_char type;
  u_char flags;
  u_char message;
  safi_t safi;
  u_char nexthop_num;
  struct in6_addr **nexthop;
  u_char ifindex_num;
  unsigned int *ifindex;
  u_char distance;
  u_int32_t metric;
};
```

Important callback functions for zclient:

```
/* Zebra tell routing daemon of an interface in the machine. */
extern struct interface *zebra_interface_add_read (struct stream *);
/* Zebra tell routing daemon of an interface state update in the machine. */
extern struct interface *zebra_interface_state_read (struct stream *s);
/* Zebra tell routing daemon of an interface address update in the machine. */
extern struct connected *zebra_interface_address_read (int, struct stream *);
/* Ask zebra daemon to add/delete an IPv4 address */
extern int zapi_ipv4_route (u_char cmd, struct zclient *, struct prefix_ipv4 *,
                          struct zapi_ipv4 *);
/* Ask zebra daemon to add/delete an IPv6 address. */
extern int zapi_ipv6_route (u_char cmd, struct zclient *zclient,
                     struct prefix_ipv6 *p, struct zapi_ipv6 *api);
```

The supplement parameters for zapi structure must be initialized and constructed by developers before sending to Zebra daemon (via Zebra socket). The details workflow for routing protocol daemon to communicate with zebra daemon will be demonstrated in later chapter 3 for AODV and OLSR daemons.

2.4.4. Zebra daemon

This is the core daemon of Quagga routing suite, it acts as an abstraction layer between other user-defined routing daemons and the underlying Unix kernel. That means, it will send and receive routing requests from routing protocol daemons; add, delete and update kernel routing table. Software module for handling zebra daemon functionalities can be found under folder zebra in Quagga source tree.

a. **Structure of zserv** : defined in zebra/zserv.h

This structure represents the Zebra daemon. It has sock parameter which is the socket to communicate with routing protocol daemons. Zebra daemon supports two types of socket – TCP and UNIX socket.

```
/* Client structure. */
struct zserv
{
  /* Client file descriptor. */
  int sock;

  /* Input/output buffer to the client. */
  struct stream *ibuf;
  struct stream *obuf;

  /* Buffer of data waiting to be written to client. */
  struct buffer *wb;

  /* Threads for read/write. */
  struct thread *t_read;
  struct thread *t_write;

  /* Thread for delayed close. */
  struct thread *t_suicide;

  /* default routing table this client munges */
  int rtm_table;
```

```
 /* This client's redistribute flag. */
 u_char redist[ZEBRA_ROUTE_MAX];

 /* Redistribute default route flag. */
 u_char redist_default;

 /* Interface information. */
 u_char ifinfo;

 /* Router-id information. */
 u_char ridinfo;
};
```

When Zebra daemon starts, zebra_zserv_socket_init() is called to create Zebra server socket, zebra_serv() function is called if Zebra daemon support TCP socket, otherwise, zebra_serv_un() function is called to create Zebra UNIX domain socket with file descriptor at /var/run/zserv.api by default. Routing protocol daemons will connect to this UNIX socket for communicating with Zebra daemon.

Zebra daemon has three buffer; *ibuf and *obuf for input/output buffer to client and *wb buffer for data waiting to be written to client.

Sub-thread *t_read is periodically reschedule to listen to routing protocol daemon requests. zebra_client_read() function is called, it reads messages from UNIX socket, filter command type and perform correspondent functions for handling client requests. In this writing, I want to focus on the kernel routing table update requests - ZEBRA_IPVX_ROUTE_ADD and ZEBRA_IPVX_ROUTE_DELETE (X stands for IP v4 and IP v6).

b. **Structure of rib** : defined at zebra/rib.h

Zebra daemon uses structure rib for storing routing information base.

```
struct rib
{
 struct rib *next;  /* Link list. */
 struct rib *prev;
```

```
 struct nexthop *nexthop;    /* Nexthop structure */
 unsigned long refcnt;    /* Refrence count. */
 time_t uptime;    /* Uptime. */
 int type;  /* Type fo this route. */
 int table;  /* Which routing table */
 u_int32_t metric;   /* Metric */
 u_char distance;  /* Distance. */

 /* Flags of this route.
  * This flag's definition is in lib/zebra.h ZEBRA_FLAG_* and is exposed
  * to clients via Zserv
  */
 u_char flags;

 u_char status;  /* RIB internal status */
#define RIB_ENTRY_REMOVED   (1 << 0)

 /* Nexthop information. */
 u_char nexthop_num;
 u_char nexthop_active_num;
 u_char nexthop_fib_num;
};
```

Because Zebra daemon receives routes from multiple routing protocol daemons which may have same destination address, nexthop address. Each rib entry has parameter 'type' to distinguish route entries. For each routing protocol daemon implementation in Quagga, we must declare a route type for each routing daemon with a administrative distance value (the smaller administrative distance value is the more priority the route entry is) in lib/route_types.h and zebra/zebra.rib.c.

```
 [ZEBRA_ROUTE_SYSTEM]        = {ZEBRA_ROUTE_SYSTEM,    0},
 [ZEBRA_ROUTE_KERNEL]        = {ZEBRA_ROUTE_KERNEL,    0},
 [ZEBRA_ROUTE_CONNECT]       = {ZEBRA_ROUTE_CONNECT,   0},
 [ZEBRA_ROUTE_STATIC]        = {ZEBRA_ROUTE_STATIC,    1},
 [ZEBRA_ROUTE_RIP]           = {ZEBRA_ROUTE_RIP,     120},
 [ZEBRA_ROUTE_RIPNG]         = {ZEBRA_ROUTE_RIPNG,   120},
 [ZEBRA_ROUTE_OSPF]          = {ZEBRA_ROUTE_OSPF,    110},
 [ZEBRA_ROUTE_OSPF6]         = {ZEBRA_ROUTE_OSPF6,   110},
```

[ZEBRA_ROUTE_ISIS]	= {ZEBRA_ROUTE_ISIS, 115},
[ZEBRA_ROUTE_BGP]	= {ZEBRA_ROUTE_BGP, 20 },
[ZEBRA_ROUTE_BABEL]	= {ZEBRA_ROUTE_BABEL, 95},
[ZEBRA_ROUTE_AODV]	= {ZEBRA_ROUTE_AODV, 10},
[ZEBRA_ROUTE_OLSR]	= {ZEBRA_ROUTE_OLSR, 11},

Table 2: Route types and administrative distance value in Zebra libraries.

In this writing, I choose 10 for AODV and 11 for OLSR route administrative distance value (this is for experiment only since there is no specific value for ad-hoc routing protocol).

c. Route update flow

Below is an simple route update flow in Quagga framework.

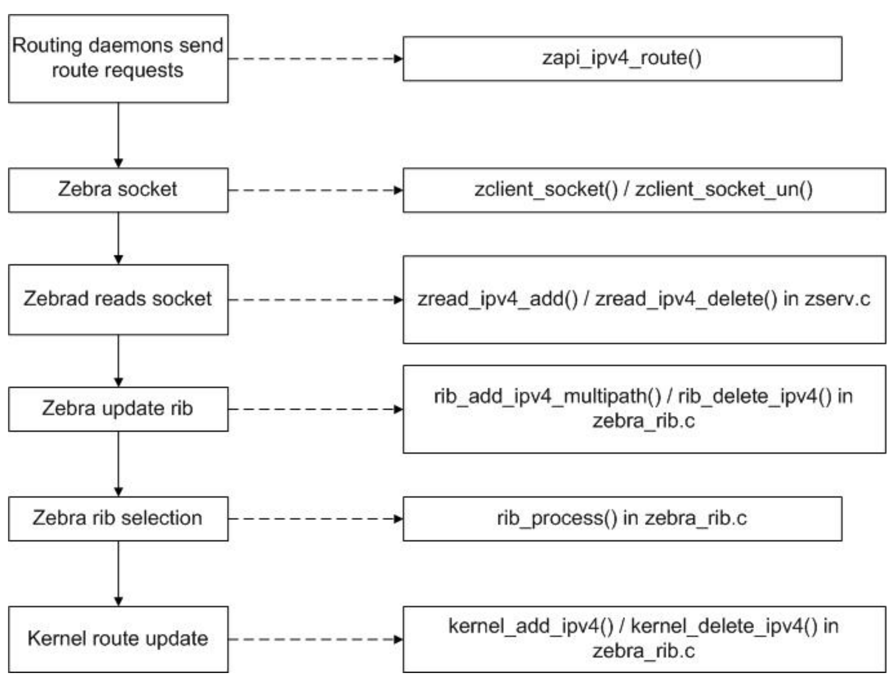


Figure 7: Kernel route update flow in Quagga

Routing daemons operates and send route update to Zebra daemon by calling function zapi_ipv4_route() (for IPv6 route, zapi_ipv6_route() is called instead). Route information (with command for add or delete) is sent to Zebra socket – zclient_socket() / zclient_socket_un() is called to establish the connection. Zebra daemon periodically listen to the socket. Upon receiving route request, functions zread_ipv4_add() / zread_ipv4_delete() is called to handle add or delete request.

Zebra routing information base will then be updated with client's route requests by calling rib_add_ipv4_multipath() (for adding new route) and rib_delete_ipv4() (for deleting a route). After updating, rib_process() is called to process Zebra routing information base, at this stage, Forwarding Information Base (FIB) is calculated (this is where administrative distance value is used for calculating FIB). Zebra feeds the FIB to the kernel, which allows the IP stack in the kernel to forward packets according to the routes computed by Quagga. The kernel FIB is updated in an OS-specific way. For example, the netlink interface is used on Linux, and route sockets are used on FreeBSD. [08]

2.5. Linux kernel routing tables and interfaces

2.5.1. Routing tables

Since linux kernel 2.2, mutiple routing tables is supported. The two common used tables named local and main route table. We can see the list linux kernel of routing tables from /etc/iproute2/rt_tables

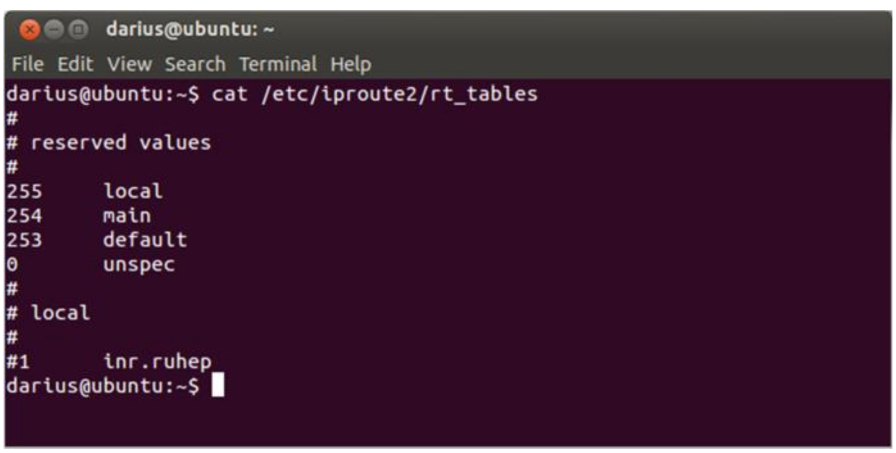


Figure 8: Routing tables on Linux

When a Linux node acting as a router, it supports a flexible route selection capability – route selected based on the packet characteristic; besides the traditional way – hop-by-hop.

Linux kernel makes use of multiple routing tables, the routing policy database (RPDB) and route cache (also referred as forwarding information base). Routing Selection Algorithm in Pseudo-code:

```
if packet.routeCacheLookupKey in routeCache :
   route = routeCache[ packet.routeCacheLookupKey ]
else
   for rule in rpdb :
      if packet.rpdbLookupKey in rule :
         routeTable = rule[ lookupTable ]
         if packet.routeLookupKey in routeTable :
            route = route_table[ packet.routeLookup_key ]
```

When determining the route by which to send a packet, the kernel always consults the routing cache first. The routing cache is a hash table used for quick access to recently used routes. If the kernel finds an entry in the routing cache, the corresponding entry will be used. If there is no entry in the routing cache, the kernel begins the process of route selection.

The kernel begins iterating by priority through the routing policy database. For each matching entry in the RPDB, the kernel will try to find a matching route to the destination IP address in the specified routing table using the aforementioned longest prefix match selection algorithm. When a matching destination is found, the kernel will select the matching route, and forward the packet. If no matching entry is found in the specified routing table, the kernel will pass to the next rule in the RPDB, until it finds a match or falls through the end of the RPDB and all consulted routing tables.

2.5.2. Kernel route update

Linux kernel route can be updated via ip route command line tool or from a programming approach. As mentioned in section 2.4.4, Zebra daemon supports multiple kernel route update methods: ioclt() call, routing socket and netlink. The implementation of functions kernel_add_ipv4() and kernel_delete_ipv4() for above methods can be found in the appendix. The two functions kernel_add_ipv4() and kernel_delete_ipv4() are used for updating kernel routing table when Zebra daemon receives route requests from AODVd and OLSRd in this writing because the two daemons now only support Ipv4 address.

Chapter 3. METHODOLOGY

3.1. Introduction

In this section, I describe the main flows in the implementation of AODVd and OLSRd which integrated into Quagga infrastructure. Details of the synchronization between AODVd and OLSRd routing table with Zebra routing tables and the kernel routing table.

3.2. Integration of AODVd to Quagga 0.99.22.1

3.2.1. Packet handling

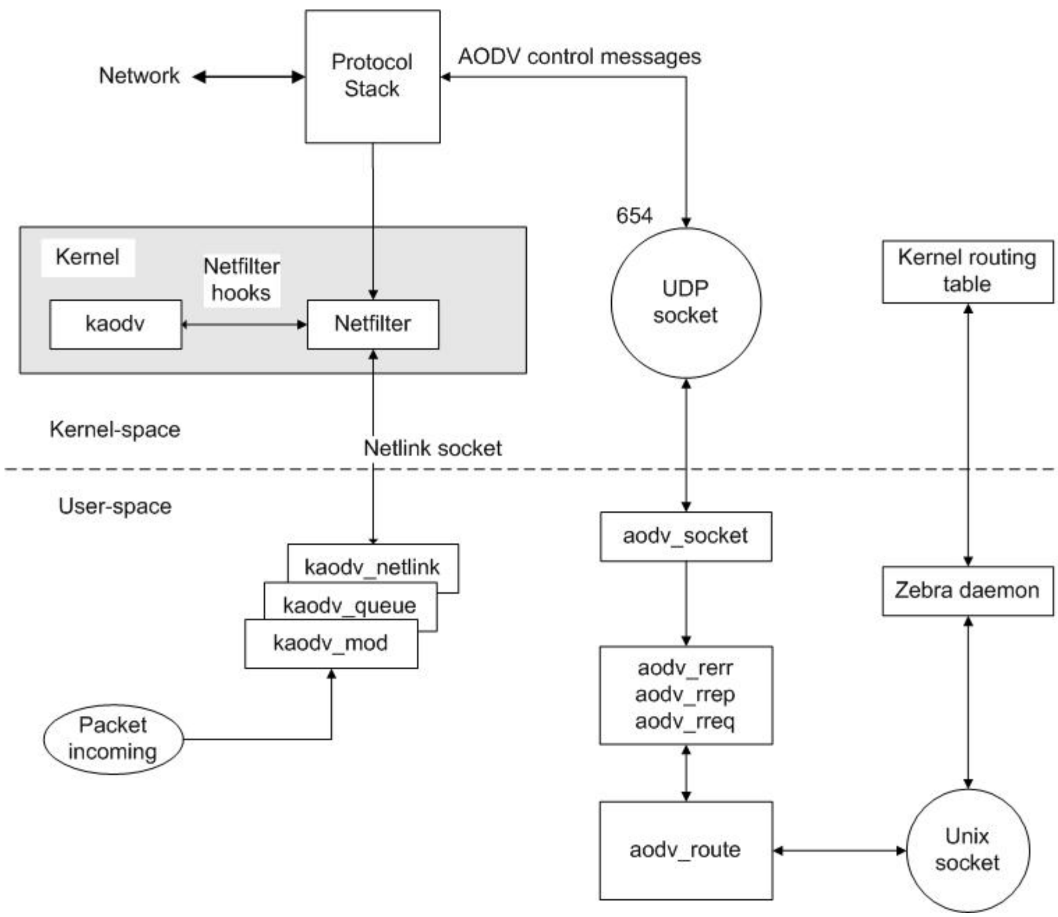


Figure 9: Packet handling of AODVd.

Data packets and AODV control messages are handled separately (Modified from [02])

AODV is a on-demand protocol which means the routing daemon must know when it must trigger route discovery process. In this AODVd implementation, I keep the approach from AODVd build – using netfilter framework, for determining route request. The kernel routing table update job is transfer to Zebra daemon.

The AODVd kernel module setup hooks at netfilter hooks in protocol stacks. These hooks will capture traversed packet, identifies the packet types and tells Netfilter to accept the packet or queue the packet for later processing at user-space. [02]

Function kaodv_netlink_init() in kaodv-netlink.c handle for initializing netlink socket to kernel by calling system function netlink_kernel_create().

```
int kaodv_netlink_init(void)
{
netlink_register_notifier(&kaodv_nl_notifier);
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,14))
kaodvnl = netlink_kernel_create(NETLINK_AODV, kaodv_netlink_rcv_sk);
#elif (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,22))
kaodvnl = netlink_kernel_create(NETLINK_AODV, AODVGRP_MAX,
kaodv_netlink_rcv_sk, THIS_MODULE);
#elif (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24))
kaodvnl = netlink_kernel_create(NETLINK_AODV, AODVGRP_MAX,
kaodv_netlink_rcv_sk, NULL, THIS_MODULE);
#else
kaodvnl = netlink_kernel_create(&init_net, NETLINK_AODV,
AODVGRP_MAX, kaodv_netlink_rcv_skb, NULL, THIS_MODULE);
#endif
}
```

In function kaodv_hook() in kaodv-mod.c defined correspondent actions on a packets. Incoming AODV control messages are accepted and queued for processing on a separated UDP socket. The message is processed by the aodv_socket module. It checks the message type field and call correspondent functions to handle. Please refer to section 3.2.2 for AODV control message processing.

Outgoing AODV message generated by the localhost will also be caught by Netfilter hook (NL_IP_LOCAL_OUT), queued by kaodv-mod module, and received by kaodv-queue module. The kaodvd-queue module will return an accept verdict to kaodv-netlink and the packet will then caught by Netfilter post-routing hook (NL_IP_POST_ROUTING). The packet also updated the most routing information and finally sent out. [02]

For handling data packet, if the destination of the packet (determined by its destination IP address) is the current host, the packet is a broadcast packet, or Internet gateway mode has been enabled and the packet is not a broadcast within the current subnet, the packet is accepted. This means that the packet under these circumstances will be handled as usual by the operating system. [02]

Otherwise, the packet should be forwarded, queued or dropped. The internal routing table of AODV is used for checking whether an active route to the specified destination exists or not. If such a route exists, the next hop of the packet is set and the packet is forwarded. Otherwise, provided that the packet was generated locally, the kaodv-netlink module for indirectly queuing the packet until AODVd has decided on an action, and a route discovery is initiated. If the packet was not generated locally, and no route was found, it is instead dropped and a RERR message is sent to the source of the packet. (Modified from [02]).

When AODVd updating its routing table, changes is synchronized to kernel routing table via Zebra call functions. A Unix socket is initialized to create a communication channel between AODVd and Zebra daemon. Route updates are sent to Zebra daemon (aodv_route) using Zebra protocol, please refer to section 3.2.3 for details. Upon receiving route updates, Zebra will recalculate its routing tables (RIB and FIB), and Zebra FIB is fed to kernel routing table (see the appendix).

3.2.2. AODV flow

The following AODV flow demonstrate how AODV control messages be processed

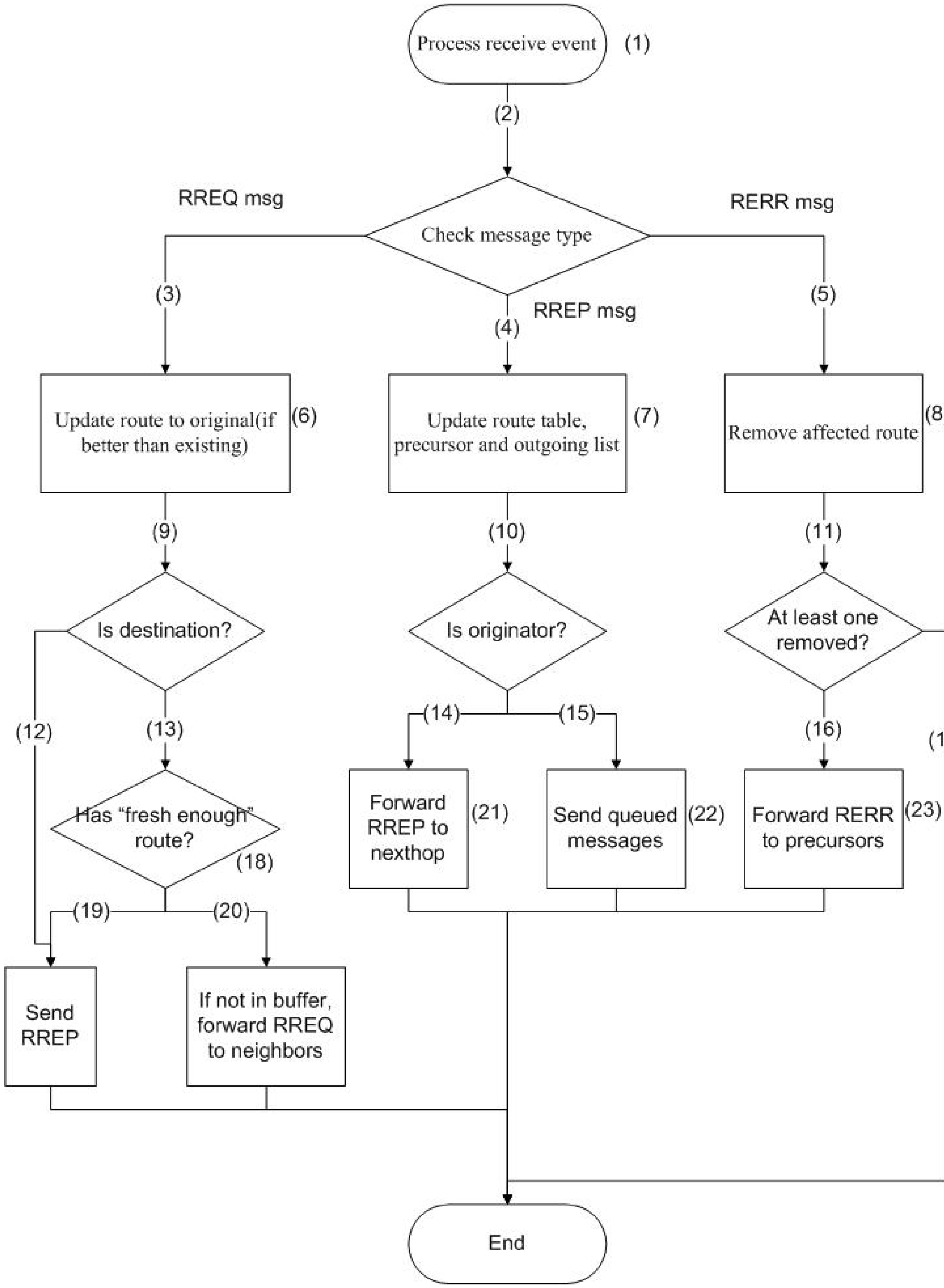


Figure 10: AODV flow (modified from [02])

a. Packet processing in kernel-space

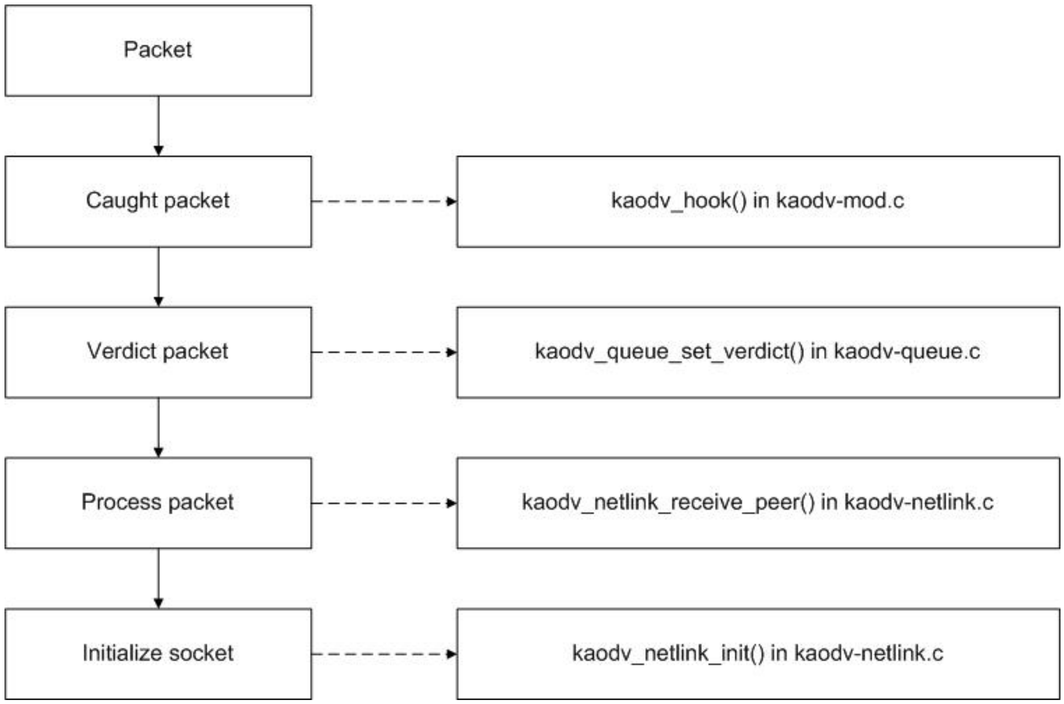


Figure 11: Packet processing in kernel-space of AODVd

Incoming packet will be caught by kaodv_hook() in kaodv-mod.c – a implemented of Netfilter hook. Both AODV control messages and data messages are captured.

```
static unsigned int kaodv_hook(unsigned int hooknum,
                               struct sk_buff *skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn) (struct sk_buff *))
```

sk_buff is data header and net_device stores network device information.

Check if this is AODV control messages

```
/* We want AODV control messages to go through directly to the
         * AODV socket.... */
Check UDP header of the packet
if (iph && iph->protocol == IPPROTO_UDP) {
```

```
	struct udphdr *udph;

udph = (struct udphdr *)((char *)iph + (iph->ihl << 2));
```
The destination and source port in UDP header are AODV_PORT
```
	if (ntohs(udph->dest) == AODV_PORT ||
	    ntohs(udph->source) == AODV_PORT) {
```
Check Netfilter condition, in this case packet is dropped
```
if (qual_th && hooknum == NF_INET_PRE_ROUTING) {

	if (qual && qual < qual_th) {
		pkts_dropped++;
		return NF_DROP;
	}
}
```

Check Netfilter condition, in this case packet is accepted

```
if (hooknum == NF_INET_PRE_ROUTING && in)
		kaodv_update_route_timeouts(hooknum, in, iph);
	return NF_ACCEPT;
}
```

Then packets need processing are queued by kaodv_queue_enqueue_packet()

After that, kaodv_queue_set_verdict() in kaodv-queue.c is called.

If verdict is drop, it does:

```
if (verdict == KAODV_QUEUE_DROP) {

while (1) {
	entry = kaodv_queue_find_dequeue_entry(dest_cmp, daddr);
```
Queue entry found by kaodv_queue_find_dequeue_entry() defined in kaodv-queue.c.
```
static struct kaodv_queue_entry
*kaodv_queue_find_dequeue_entry(kaodv_queue_cmpfn cmpfn, unsigned long data)
{
	struct kaodv_queue_entry *entry;
```

```
	write_lock_bh(&queue_lock);
	entry = __kaodv_queue_find_dequeue_entry(cmpfn, data);
	write_unlock_bh(&queue_lock);
	return entry;
}
```
This function return a queue entry structure
```
struct kaodv_queue_entry {
	struct list_head list;
	struct sk_buff *skb;
	int (*okfn) (struct sk_buff *);
	struct kaodv_rt_info rt_info;
};
```
list_head is list header of entry to be dropped, sk_buff is header data

If verdict is send, packets are accepted and will be sent
```
else if (verdict == KAODV_QUEUE_SEND) {
	struct expl_entry e;

while (1) {
	entry = kaodv_queue_find_dequeue_entry(dest_cmp, daddr);
```
The entry is encapsulated and sent out
```
if (e.flags & KAODV_RT_GW_ENCAP) {
	entry->skb = ip_pkt_encapsulate(entry->skb, e.nhop);
	if (!entry->skb)
		goto next;
}
```
ip_pkt_encapsulate() defined in kaodv-ipenc.c

Allocate new data space at head
```
nskb = skb_copy_expand(skb, skb_headroom(skb),
skb_tailroom(skb) +
sizeof(struct min_ipenc_hdr),
GFP_ATOMIC);
```
Set old owner
```
if (skb->sk != NULL)
skb_set_owner_w(nskb, skb->sk);
iph = SKB_NETWORK_HDR_IPH(skb);
skb_put(nskb, sizeof(struct min_ipenc_hdr));
```

Move the IP header

```
memcpy(nskb->data, skb->data, (iph->ihl << 2));
```

Move the data

```
memcpy(nskb->data + (iph->ihl << 2) + sizeof(struct min_ipenc_hdr),
skb->data + (iph->ihl << 2), skb->len - (iph->ihl << 2));
kfree_skb(skb);
skb = nskb;
```

Update pointers

```
SKB_SET_NETWORK_HDR(skb, 0);
iph = SKB_NETWORK_HDR_IPH(skb);
ipe = (struct min_ipenc_hdr *)(SKB_NETWORK_HDR_RAW(skb) + (iph->ihl
<< 2));
```

Save the old ip header information in the encapsulation header

```
ipe->protocol = iph->protocol;
ipe->s = 0;/* No source address field in the encapsulation header */
ipe->res = 0;
ipe->check = 0;
ipe->daddr = iph->daddr;
```

Update the IP header

```
iph->daddr = dest;
iph->protocol = IPPROTO_MIPE;
iph->tot_len = htons(ntohs(iph->tot_len) + sizeof(struct
min_ipenc_hdr));
```

Recalculate checksums

```
ipe->check = ip_csum((unsigned short *)ipe, 4);
ip_send_check(iph);
if (iph->id == 0)
ip_select_ident(iph, skb_dst(skb), NULL);
```

Then kaodv_netlink_receive_peer() functions defined in kaodv-netlink.c is called to processed the packet. Because of the new injection code for Zebra, some part of this function kaodv_netlink_receive_peer() is not reach – AODV daemon does not send route add/delete to kaodv module but to Zebra daemon instead, more details of route add/delete can be found in next section Kernel route management.

```
case KAODVM_ADDROUTE:
        if (len < sizeof(struct kaodv_rt_msg))
                return -EINVAL;
```

```
        m = (struct kaodv_rt_msg *)msg;

        ret = kaodv_expl_get(m->dst, &e);

        if (ret < 0) {
                ret = kaodv_expl_update(m->dst, m->nhop, m->time,
                                        m->flags, m->ifindex);
        } else {
                ret = kaodv_expl_add(m->dst, m->nhop, m->time,
                                        m->flags, m->ifindex);
        }
        kaodv_queue_set_verdict(KAODV_QUEUE_SEND, m->dst);
        break;
case KAODVM_DELROUTE:
        if (len < sizeof(struct kaodv_rt_msg))
                return -EINVAL;

        m = (struct kaodv_rt_msg *)msg;
        kaodv_expl_del(m->dst);
        kaodv_queue_set_verdict(KAODV_QUEUE_DROP, m->dst);
        break;
```

The remain code is still work :

```
case KAODVM_NOROUTE_FOUND:
        if (len < sizeof(struct kaodv_rt_msg))
                return -EINVAL;

        m = (struct kaodv_rt_msg *)msg;
        KAODV_DEBUG("No route found for %s", print_ip(m->dst));
        kaodv_queue_set_verdict(KAODV_QUEUE_DROP, m->dst);
        break;
case KAODVM_CONFIG:
        if (len < sizeof(struct kaodv_conf_msg))
                return -EINVAL;

        cm = (struct kaodv_conf_msg *)msg;
        active_route_timeout = cm->active_route_timeout;
```

```
        qual_th = cm->qual_th;
        is_gateway = cm->is_gateway;
        break;
default:
        printk("kaodv-netlink: Unknown message type\n");
        ret = -EINVAL;
```

Function kaodv_netlink_init() (defined in kaodv-netlink.c) called to initialize the socket to send packet to AODV control in user-space

```
netlink_register_notifier(&kaodv_nl_notifier);
kaodvnl = netlink_kernel_create(&init_net, NETLINK_AODV,
AODVGRP_MAX, kaodv_netlink_rcv_skb, NULL, THIS_MODULE);
```

b. Packet processing in user-space

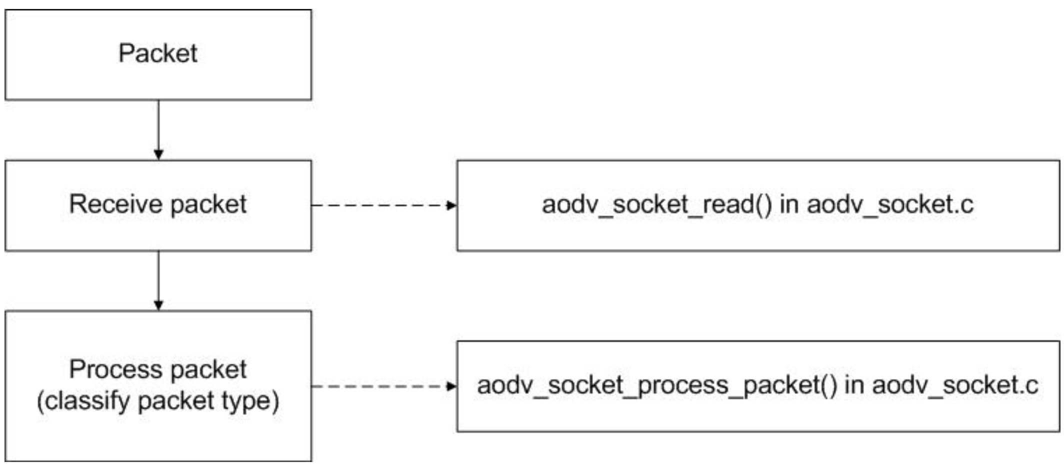


Figure 12: Packet processing in user-space of AODVd

Function aodv_socket_read() in aodv-socket.c will receive packet from upper layer:
The received message stored in data structure msgh. The value fd is the file descriptor of UDP socket that binds to AODV UDP port, fd is calculated in aodv_socket_init() in aodv_socket.c

```
len = recvmsg(fd, &msgh, 0);

  if (len < 0) {
```

```
        alog(LOG_WARNING, 0, __FUNCTION__, "receive ERROR len=%d!",
len);
        return;
    }
```
Get source address from the message
```
    src.s_addr = src_addr.sin_addr.s_addr;
```
Get the ttl and destination address from the message
```
    /* Get the ttl and destination address from the control message */
    for (cmsg = CMSG_FIRSTHDR(&msgh); cmsg != NULL;
         cmsg = CMSG_NXTHDR_FIX(&msgh, cmsg)) {
        if (cmsg->cmsg_level == SOL_IP) {
            switch (cmsg->cmsg_type) {
            case IP_TTL:
                ttl = *(CMSG_DATA(cmsg));
                break;
            case IP_PKTINFO:
              {
                struct in_pktinfo *pi = (struct in_pktinfo *)CMSG_DATA(cmsg);
                dst.s_addr = pi->ipi_addr.s_addr;
              }
            }
        }
    }
```
Prepare the AODV message structure from received message for processing.
```
aodv_msg = (AODV_msg *) recv_buf;

    dev = devfromsock(fd);

    if (!dev) {
        DEBUG(LOG_ERR, 0, "Could not get device info!\n");
        return;
    }
```

Then function aodv_socket_process_packet() is called for processing the control message.

The definition of this function
```
void NS_CLASS aodv_socket_process_packet(AODV_msg * aodv_msg, int len,
```

```
                                          struct in_addr src,
                                          struct in_addr dst,
                                          int ttl, unsigned int ifindex)
```

If the received message is a HELLO message, hello_process() in aodv_hello.c is called

```
if ((aodv_msg->type == AODV_RREP && ttl == 1 &&
         dst.s_addr == AODV_BROADCAST)) {
        hello_process((RREP *) aodv_msg, len, ifindex);
        return;
   }
```

Neighbor is added/updated with function neighbor_add in aodv_socket.c

```
/* Make sure we add/update neighbors */
   neighbor_add(aodv_msg, src, ifindex);
```

Check what type of msg we received and call the corresponding function to handle the msg...

```
switch (aodv_msg->type) {
```

- If this is a RREQ message : call rreq_process() to process (Process 3 in figure 10)

rreq_process() defined in aodv_rreq.c is called to process a RREQ message

```
   case AODV_RREQ:
        rreq_process((RREQ *) aodv_msg, len, src, dst, ttl, ifindex);
        break;
```

- If this is a RREP message : call rrep_process() to process (Process 4 in figure 10)

rrep_process() defined in aodv_rrep.c is called to process a RREP message

```
   case AODV_RREP:
        DEBUG(LOG_DEBUG, 0, "Received RREP");
        rrep_process((RREP *) aodv_msg, len, src, dst, ttl, ifindex);
        break;
```

- If this is a RERR message : call rerr_process() to process (Process 5 in figure 10)

rerr_process() defined in aodv_rerr.c is called to process a RERR message

```
   case AODV_RERR:
        DEBUG(LOG_DEBUG, 0, "Received RERR");
```

```
        rerr_process((RERR *) aodv_msg, len, src, dst);
        break;
```

c. AODV control message processing

AODV is a reactive protocol which only trigger route discovery on demand. Function rreq_route_discovery() is called to check if we have a route to destination, if not start route discovery process

```
void NS_CLASS rreq_route_discovery(struct in_addr dest_addr, u_int8_t flags,
                                   struct ip_data *ipd)
{
    struct timeval now;
    rt_table_t *rt;
    seek_list_t *seek_entry;
    u_int32_t dest_seqno;
    int ttl;
#define TTL_VALUE ttl

    gettimeofday(&now, NULL);
```

Find an entry from seek_list, if found return the entry.

```
if (seek_list_find(dest_addr))
        return;
```

Try finding a route from AODV routing table.

```
/* If we already have a route entry, we use information from it. */
    rt = rt_table_find(dest_addr);

    ttl = NET_DIAMETER;             /* This is the TTL if we don't use expanding
                                   ring search */
```

If we still don’t have a route, send RREQ message for route.

```
rreq_send(dest_addr, dest_seqno, ttl, flags);
```

And update this destination to seek_list with function seek_list_insert() in seek_list.c (process 6 in figure)

```
 /* Remember that we are seeking this destination */
    seek_entry = seek_list_insert(dest_addr, dest_seqno, ttl, flags, ipd);
```

Also we need to update timer for the new seek entry

```
/* Set a timer for this RREQ */
   if (expanding_ring_search)
         timer_set_timeout(&seek_entry->seek_timer,
RING_TRAVERSAL_TIME);
   else
         timer_set_timeout(&seek_entry->seek_timer,
NET_TRAVERSAL_TIME);
```

The function seek_list_insert() definition

```
seek_list_t *NS_CLASS seek_list_insert(struct in_addr dest_addr,
                                        u_int32_t dest_seqno,
                                        int ttl, u_int8_t flags,
                                        struct ip_data *ipd)
{
   seek_list_t *entry;

   if ((entry = (seek_list_t *) malloc(sizeof(seek_list_t))) == NULL) {
         fprintf(stderr, "Failed malloc\n");
         exit(-1);
   }

   entry->dest_addr = dest_addr;
   entry->dest_seqno = dest_seqno;
   entry->flags = flags;
   entry->reqs = 0;
   entry->ttl = ttl;
   entry->ipd = ipd;

   timer_init(&entry->seek_timer, &NS_CLASS route_discovery_timeout, entry);
```

Add an entry

```
list_add(&seekhead, &entry->l);
```

- **Reception of RREQ**

Function rreq_process() in aodv_rreq.c is called to process a RREQ message. It's definition

```
void NS_CLASS rreq_process(RREQ * rreq, int rreqlen, struct in_addr ip_src,
          struct in_addr ip_dst, int ip_ttl, unsigned int ifindex)
```

Perform condition check on RREQ message

```
   /* Ignore RREQ's that originated from this node. Either we do this
      or we buffer our own sent RREQ's as we do with others we
      receive. */
   if (rreq_orig.s_addr == DEV_IFINDEX(ifindex).ipaddr.s_addr)
          return;

   DEBUG(LOG_DEBUG, 0, "ip_src=%s rreq_orig=%s rreq_dest=%s ttl=%d",
           ip_to_str(ip_src), ip_to_str(rreq_orig), ip_to_str(rreq_dest),
           ip_ttl);

   if (rreqlen < (int) RREQ_SIZE) {
          alog(LOG_WARNING, 0,
              __FUNCTION__, "IP data field too short (%u bytes)"
              "from %s to %s", rreqlen, ip_to_str(ip_src), ip_to_str(ip_dst));
          return;
   }
```

If the previous hop of the RREQ is in blacklist or an already processed RREQ, then ignore

```
   if (rreq_blacklist_find(ip_src)) {
          DEBUG(LOG_DEBUG, 0, "prev hop of RREQ blacklisted, ignoring!");
          return;
   }

   /* Ignore already processed RREQs. */
   if (rreq_record_find(rreq_orig, rreq_id))
          return;
```

Update / create a reverse route to the source of the RREQ.

```
    /* The node always creates or updates a REVERSE ROUTE entry to the
       source of the RREQ. */
    rev_rt = rt_table_find(rreq_orig);

    /* Calculate the extended minimal life time. */
    life = PATH_DISCOVERY_TIME - 2 * rreq_new_hcnt *
NODE_TRAVERSAL_TIME;

    if (rev_rt == NULL) {
            DEBUG(LOG_DEBUG, 0, "Creating REVERSE route entry, RREQ orig:
%s",
                  ip_to_str(rreq_orig));

            rev_rt = rt_table_insert(rreq_orig, ip_src, rreq_new_hcnt,
                                     rreq_orig_seqno, life, VALID, 0, ifindex);
    } else {
            if (rev_rt->dest_seqno == 0 ||
                (int32_t) rreq_orig_seqno > (int32_t) rev_rt->dest_seqno ||
                (rreq_orig_seqno == rev_rt->dest_seqno &&
                 (rev_rt->state == INVALID || rreq_new_hcnt < rev_rt->hcnt))) {
                rev_rt = rt_table_update(rev_rt, ip_src, rreq_new_hcnt,
                                         rreq_orig_seqno, life, VALID,
                                         rev_rt->flags);
            }
```

If we are the destination, then we send the RREP (process 12 of figure 10)

```
    /* Are we the destination of the RREQ?, if so we should immediately send a
       RREP.. */
    if (rreq_dest.s_addr == DEV_IFINDEX(ifindex).ipaddr.s_addr) {

            /* WE are the RREQ DESTINATION. Update the node's own
               sequence number to the maximum of the current seqno and the
               one in the RREQ. */
            if (rreq_dest_seqno != 0) {
                if ((int32_t) this_host.seqno < (int32_t) rreq_dest_seqno)
                        this_host.seqno = rreq_dest_seqno;
                else if (this_host.seqno == rreq_dest_seqno)
```

```
                        seqno_incr(this_host.seqno);
            }
            rrep = rrep_create(0, 0, 0, DEV_IFINDEX(rev_rt->ifindex).ipaddr,
                                        this_host.seqno, rev_rt->dest_addr,
                                        MY_ROUTE_TIMEOUT);
            rrep_send(rrep, rev_rt, NULL, RREP_SIZE);
```

If we are not the destination, check if we has “fresh enough” route (process 18 of figure 10), function rt_table_find() is called to lookup a route.

```
    } else {
            /* We are an INTERMEDIATE node. - check if we have an active
             * route entry */
            fwd_rt = rt_table_find(rreq_dest);
```

If we has a “fresh enough” route, then generate and send RREP (process 19 of figure 10)

```
if (fwd_rt && fwd_rt->state == VALID && !rreq->d) {
    struct timeval now;
    u_int32_t lifetime;
    /* GENERATE RREP, i.e we have an ACTIVE route entry that is fresh
       enough (our destination sequence number for that route is
       larger than the one in the RREQ). */
    gettimeofday(&now, NULL);
    /* Respond only if the sequence number is fresh enough... */
    if (fwd_rt->dest_seqno != 0 &&
            (int32_t) fwd_rt->dest_seqno >= (int32_t) rreq_dest_seqno) {
            lifetime = timeval_diff(&fwd_rt->rt_timer.timeout, &now);
            rrep = rrep_create(0, 0, fwd_rt->hcnt, fwd_rt->dest_addr,
                               fwd_rt->dest_seqno, rev_rt->dest_addr,
                               lifetime);
            rrep_send(rrep, rev_rt, fwd_rt, rrep_size);
```

If not, we forward RREQ to neighbors (process 20 of figure 10)

```
  } else {
            goto forward;
            }
```

```
    forward:
        if (ip_ttl > 1) {
            /* Update the sequence number in case the maintained one is
             * larger */
            if (fwd_rt && !(fwd_rt->flags & RT_INET_DEST) &&
                (int32_t) fwd_rt->dest_seqno > (int32_t) rreq_dest_seqno)
                rreq->dest_seqno = htonl(fwd_rt->dest_seqno);

            rreq_forward(rreq, rreqlen, --ip_ttl);
```

- **RREP**

The function rrep_process() in aodv_rrep.c is called to process a received RREP message. It's definition

```
void NS_CLASS rrep_process(RREP * rrep, int rreplen, struct in_addr ip_src,
                           struct in_addr ip_dst, int ip_ttl, unsigned int ifindex)
```

After conditional check, it check if we should make a forward route. Function rt_table_find() is called to find forward route and reverse route.

```
    fwd_rt = rt_table_find(rrep_dest);
    rev_rt = rt_table_find(rrep_orig);
```

If there is no forward route, make a new one by calling rt_table_insert()

```
    if (!fwd_rt) {
        /* We didn't have an existing entry, so we insert a new one. */
        fwd_rt = rt_table_insert(rrep_dest, ip_src, rrep_new_hcnt, rrep_seqno,
                                 rrep_lifetime, VALID, rt_flags, ifindex);
```

If there is a forward route, try updating this route

```
    } else if (fwd_rt->dest_seqno == 0 ||
               (int32_t) rrep_seqno > (int32_t) fwd_rt->dest_seqno ||
               (rrep_seqno == fwd_rt->dest_seqno &&
                (fwd_rt->state == INVALID || fwd_rt->flags & RT_UNIDIR ||
                 rrep_new_hcnt < fwd_rt->hcnt))) {
        pre_repair_hcnt = fwd_rt->hcnt;
        pre_repair_flags = fwd_rt->flags;
```

```
        fwd_rt = rt_table_update(fwd_rt, ip_src, rrep_new_hcnt, rrep_seqno,
                                  rrep_lifetime, VALID,
                                  rt_flags | fwd_rt->flags);
```

If the RREP is for us, accept it and also check condition (process 22 in figure 10)

```
  if (rrep_orig.s_addr == DEV_IFINDEX(ifindex).ipaddr.s_addr) {
        if (pre_repair_flags & RT_REPAIR) {
            if (fwd_rt->hcnt > pre_repair_hcnt) {
                RERR *rerr;
                u_int8_t rerr_flags = 0;
                struct in_addr dest;
                dest.s_addr = AODV_BROADCAST;
                rerr_flags |= RERR_NODELETE;
                rerr = rerr_create(rerr_flags, fwd_rt->dest_addr,
                                   fwd_rt->dest_seqno);
                if (fwd_rt->nprec)
                    aodv_socket_send((AODV_msg *) rerr, dest,
                                     RERR_CALC_SIZE(rerr), 1,
                                     &DEV_IFINDEX(fwd_rt->ifindex));
            }
        }
```

If not , forward RREP to nexthop on the reverse route (process 21 of figure 10)

```
  } else {
        /* --- Here we FORWARD the RREP on the REVERSE route --- */
        if (rev_rt && rev_rt->state == VALID) {
            rrep_forward(rrep, rreplen, rev_rt, fwd_rt, --ip_ttl);
```

- **RERR**

The function rerr_process() in aodv_rerr.c is called for handling RERR message. It’s definition

```
void NS_CLASS rerr_process(RERR * rerr, int rerrlen, struct in_addr ip_src,
                           struct in_addr ip_dst)
```

Check if destination is unreachable in rerr_process()

```
#define RERR_UDEST_FIRST(rerr) ((RERR_udest *)&rerr->dest_addr)
```

```
    /* Check which destinations that are unreachable.  */
    udest = RERR_UDEST_FIRST(rerr);
```

Remove affected route by calling rt_table_invalidate() and also remove precursor list of that route with function precursor_list_destroy() (process 8 of figure 10)

```
rt = rt_table_find(udest_addr);
if (rt && rt->state == VALID && rt->next_hop.s_addr == ip_src.s_addr) {
/* Checking sequence numbers here is an out of draft
     * addition to AODV-UU. It is here because it makes a lot
     * of sense... */
    if (0 && (int32_t) rt->dest_seqno > (int32_t) rerr_dest_seqno) {
            DEBUG(LOG_DEBUG, 0, "Udest ignored because of seqno");
            udest = RERR_UDEST_NEXT(udest);
            rerr->dest_count--;
            continue;
    }
    DEBUG(LOG_DEBUG, 0, "removing rte %s - WAS IN RERR!!",
              ip_to_str(udest_addr));

    /* Invalidate route: */
    if (!rerr->n) {
            rt_table_invalidate(rt);
    }
    /* (a) updates the corresponding destination sequence number
       with the Destination Sequence Number in the packet, and */
    rt->dest_seqno = rerr_dest_seqno;

    /* (d) check precursor list for emptiness. If not empty, include
       the destination as an unreachable destination in the
       RERR... */
    if (rt->nprec && !(rt->flags & RT_REPAIR)) {

            if (!new_rerr) {
                u_int8_t flags = 0;
                if (rerr->n)
                        flags |= RERR_NODELETE;
                            new_rerr = rerr_create(flags, rt->dest_addr,
```

```
                                                rt->dest_seqno);
               DEBUG(LOG_DEBUG, 0, "Added %s as unreachable, seqno=%lu",
                         ip_to_str(rt->dest_addr), rt->dest_seqno);
                           if (rt->nprec == 1)
                       rerr_unicast_dest =
                           FIRST_PREC(rt->precursors)->neighbor;
                       } else {
               /* Decide whether new precursors make this a non unicast RERR */
               rerr_add_udest(new_rerr, rt->dest_addr, rt->dest_seqno);
                           DEBUG(LOG_DEBUG,    0,    "Added    %s    as    unreachable,
seqno=%lu",
                         ip_to_str(rt->dest_addr), rt->dest_seqno);

               if (rerr_unicast_dest.s_addr) {
                       list_t *pos2;
                       list_foreach(pos2, &rt->precursors) {
                           precursor_t *pr = (precursor_t *) pos2;
                           if (pr->neighbor.s_addr != rerr_unicast_dest.s_addr) {
                                   rerr_unicast_dest.s_addr = 0;
                                   break;
                           }
                       }
               }
           }
    } else {
           DEBUG(LOG_DEBUG, 0,
                 "Not sending RERR, no precursors or route in RT_REPAIR");
    }
    /* We should delete the precursor list for all unreachable
       destinations. */
    if (rt->state == INVALID)
           precursor_list_destroy(rt);
```

Send RERR message when at least one route removed (process 23 of figure 10)

```
    /* If a RERR was created, then send it now... */
    if (new_rerr) {
```

```
        rt = rt_table_find(rerr_unicast_dest);

        if (rt && new_rerr->dest_count == 1 && rerr_unicast_dest.s_addr)
            aodv_socket_send((AODV_msg *) new_rerr,
                             rerr_unicast_dest,
                             RERR_CALC_SIZE(new_rerr), 1,
                             &DEV_IFINDEX(rt->ifindex));

        else if (new_rerr->dest_count > 0) {
            /* FIXME: Should only transmit RERR on those interfaces
             * which have precursor nodes for the broken route */
            for (i = 0; i < MAX_NR_INTERFACES; i++) {
                struct in_addr dest;

                if (!DEV_NR(i).enabled)
                    continue;
                dest.s_addr = AODV_BROADCAST;
                aodv_socket_send((AODV_msg *) new_rerr, dest,
                                 RERR_CALC_SIZE(new_rerr), 1, &DEV_NR(i));
            }
        }
    }
```

3.2.3. Kernel route update

The user-space module of AODVd perform kernel route update via three modules

- Function nl_send_add_route_msg() for adding new route to kernel.
- Function nl_send_del_route_msg() for deleting a route from kernel.
- Function nl_send_no_route_found_msg() for annoucement of no route found.

These functions are called when AODVd perform update on its routing table to synchronize AODVd and kernel routing table. With the integration with Quagga, AODVd will tell Zebra daemon for route updates (add/update/delete). This is done by module aodv_route() with definition is file zebra.c. The destination information is stored in data structure prefix_ipv4 p, the next_hop, metric and interface index are used to generate data structure zapi_ipv4 api. Finally, the module zapi_ipv4_route() is called to perform appropriate actions (add /delete route)

a. Adding new route to kernel

Function rt_table_insert() in routing_table.c is called to insert new route to AODV routing table.

```
rt_table_t *NS_CLASS rt_table_insert(struct in_addr dest_addr,
                                     struct in_addr next,
                                     u_int8_t hops, u_int32_t seqno,
                                     u_int32_t life, u_int8_t state,
                                     u_int16_t flags, unsigned int ifindex)
```

First it calculate has key and search for existing route entry for a destination

```
/* Calculate hash key */
index = hashing(&dest_addr, &hash);

/* Check if we already have an entry for dest_addr */
list_foreach(pos, &rt_tbl.tbl[index]) {
        rt = (rt_table_t *) pos;
        if (memcmp(&rt->dest_addr, &dest_addr, sizeof(struct in_addr))
           == 0) {
                DEBUG(LOG_INFO, 0, "%s already exist in routing table!",
                      ip_to_str(dest_addr));
                return NULL;
        }
}
```

If there is no available route then initialize a route table entry rt_table_t() defined in routing_table.h

```
if ((rt = (rt_table_t *) malloc(sizeof(rt_table_t))) == NULL) {
        fprintf(stderr, "Malloc failed!\n");
        exit(-1);
}
memset(rt, 0, sizeof(rt_table_t));
```

Fill route entry information

```
rt->dest_addr = dest_addr;
```

```
rt->next_hop = next;
rt->dest_seqno = seqno;
rt->flags = flags;
rt->hcnt = hops;
rt->ifindex = ifindex;
rt->hash = hash;
rt->state = state;
```

Add timer for new route entry

```
timer_init(&rt->rt_timer, &NS_CLASS route_expire_timeout, rt);
timer_init(&rt->ack_timer, &NS_CLASS rrep_ack_timeout, rt);
timer_init(&rt->hello_timer, &NS_CLASS hello_timeout, rt);
```

Add the rest information and add the first index of routing table

```
rt->last_hello_time.tv_sec = 0;
rt->last_hello_time.tv_usec = 0;
rt->hello_cnt = 0;
rt->nprec = 0;
INIT_LIST_HEAD(&rt->precursors);
/* Insert first in bucket... */
rt_tbl.num_entries++;
DEBUG(LOG_INFO, 0, "Inserting %s (bucket %d) next hop %s",
    ip_to_str(dest_addr), index, ip_to_str(next));
list_add(&rt_tbl.tbl[index], &rt->l);
```

Check state again

```
if (state == INVALID) {
        if (flags & RT_REPAIR) {
                rt->rt_timer.handler = &NS_CLASS local_repair_timeout;
                life = ACTIVE_ROUTE_TIMEOUT;
        } else {
                rt->rt_timer.handler = &NS_CLASS route_delete_timeout;
                life = DELETE_PERIOD;
}
```

And call `nl_send_add_route_msg()` to add new route to kernel

```
#ifndef NS_PORT
        nl_send_add_route_msg(dest_addr, next, hops, life, flags, ifindex);
#endif
}
```

When AODV routing table update in function rt_table_update() (defined in routing_table.c).

In case the existing route entry state change from INVALID to VALID

```
if (rt->state == INVALID && state == VALID) {
/* If this previously was an expired route, but will now be
   active again we must add it to the kernel routing
   table... */
        rt_tbl.num_active++;
        if (rt->flags & RT_REPAIR)
                flags &= ~RT_REPAIR;
#ifndef NS_PORT
        nl_send_add_route_msg(rt->dest_addr, next, hops, lifetime, flags, rt->ifindex);
#endif
```

Or when nexthop address change

```
} else if (rt->next_hop.s_addr != 0 &&   rt->next_hop.s_addr != next.s_addr) {
        DEBUG(LOG_INFO, 0, "rt->next_hop=%s, new_next_hop=%s",
            ip_to_str(rt->next_hop), ip_to_str(next));
#ifndef NS_PORT
        nl_send_add_route_msg(rt->dest_addr, next, hops, lifetime, flags, rt->ifindex);
#endif
}
```

The function nl_send_add_route_msg() does synchronizing AODV route with kernel route which defined in nl.c

```
int nl_send_add_route_msg(struct in_addr dest, struct in_addr next_hop,
                          int metric, u_int32_t lifetime, int rt_flags, int ifindex)
```

It first notify AODV kernel module for update of new route (this is not adding but to refresh AODV kernel module stuff).

```
areq.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct kaodv_rt_msg));
areq.n.nlmsg_type = KAODVM_ADDROUTE;
areq.n.nlmsg_flags = NLM_F_REQUEST;
areq.m.dst = dest.s_addr;
areq.m.nhop = next_hop.s_addr;
areq.m.time = lifetime;
areq.m.ifindex = ifindex;
if (rt_flags & RT_INET_DEST) {
	areq.m.flags |= KAODV_RT_GW_ENCAP;
}
if (rt_flags & RT_REPAIR)
	areq.m.flags |= KAODV_RT_REPAIR;
if (nl_send(&aodvnl, &areq.n) < 0) {
	DEBUG(LOG_DEBUG, 0, "Failed to send netlink message");
	return -1;
}
```

And finally, tell Zebra daemon for adding new route to kernel routing table.

```
return aodv_route(ZEBRA_IPV4_ROUTE_ADD, dest, next_hop, metric, ifindex);
```

Note: the command ZEBRA_IPV4_ROUTE_ADD is sent for adding, the same function aodv_route() called for deleting a route but supplement with other command ZEBRA_IPV4_ROUTE_DELETE.

b. Deleting route from kernel

A route need to be deleted in some situations: during local repair process, when route entry is expired or when the routing table deleted.

During local repair process, local_repair_timeout() function in aodv_timeout.c is called

```
void NS_CLASS local_repair_timeout(void *arg)
{
	rt_table_t *rt;
	struct in_addr rerr_dest;
	RERR *rerr = NULL;
```

```
	rt = (rt_table_t *) arg;

	if (!rt)
		return;

	rerr_dest.s_addr = AODV_BROADCAST;	/* Default destination */

	/* Unset the REPAIR flag */
	rt->flags &= ~RT_REPAIR;

#ifndef NS_PORT
	nl_send_del_route_msg(rt->dest_addr, rt->next_hop, rt->hcnt, rt->ifindex);
#endif
/* Route should already be invalidated. */
```

When the route is expired, rt_table_invalidate() in routing_table.c is called

```
/* Route expiry and Deletion. */
int NS_CLASS rt_table_invalidate(rt_table_t * rt)
```

If route is already invalid, then ignore and do nothing

```
/* If the route is already invalidated, do nothing... */
if (rt->state == INVALID) {
	DEBUG(LOG_DEBUG, 0, "Route %s already invalidated!!!",
	      ip_to_str(rt->dest_addr));
	return -1;
}
```

Remove existing timers belongs to this route

```
/* Remove any pending, but now obsolete timers. */
timer_remove(&rt->rt_timer);
timer_remove(&rt->hello_timer);
timer_remove(&rt->ack_timer);
```

Mark the route as INVALID and reset all parameters

```
/* Mark the route as invalid */
rt->state = INVALID;
rt_tbl.num_active--;
```

```
rt->hello_cnt = 0;
/* When the lifetime of a route entry expires, increase the sequence
  number for that entry. */
seqno_incr(rt->dest_seqno);
rt->last_hello_time.tv_sec = 0;
rt->last_hello_time.tv_usec = 0;
```

Then nl_send_del_route_msg() is called to synchronize deletion action to kernel route

```
#ifndef NS_PORT
	nl_send_del_route_msg(rt->dest_addr, rt->next_hop, rt->hcnt, rt->ifindex);
#endif
```

When need to delete a route entry, rt_table_delete() in routing_table.c is called to perform the task

```
void NS_CLASS rt_table_delete(rt_table_t * rt)
```

First , it detach the entry from the table

```
if (!rt) {
	DEBUG(LOG_ERR, 0, "No route entry to delete");
	return;
}

list_detach(&rt->l);
```

And remove precursor list from the route entry

```
precursor_list_destroy(rt);
```

While route entry state is still VALID, kernel must be synchronized to complete deletion

```
if (rt->state == VALID) {
#ifndef NS_PORT
	nl_send_del_route_msg(rt->dest_addr, rt->next_hop, rt->hcnt, rt->ifindex);
#endif
	rt_tbl.num_active--;
}
```

Remove timers and free memory

```
/* Make sure timers are removed... */
timer_remove(&rt->rt_timer);
timer_remove(&rt->hello_timer);
timer_remove(&rt->ack_timer);
rt_tbl.num_entries--;
free(rt);
return;
```

Function `nl_send_del_route_msg()` defined in nl.c is called for sychronize detetion between AODV route and kernel.

```
int nl_send_del_route_msg(struct in_addr dest, struct in_addr next_hop, int metric, int ifindex)
```

AODV kernel module also notified about the deletion to refresh it's parameters

```
struct {
	struct nlmsghdr n;
		struct kaodv_rt_msg m;
	} areq;

DEBUG(LOG_DEBUG, 0, "Send DEL_ROUTE to kernel: %s:%s metric=%d, ifindex=%d)", ip_to_str(dest),ip_to_str(next_hop),metric,ifindex);

memset(&areq, 0, sizeof(areq));

areq.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct kaodv_rt_msg));
areq.n.nlmsg_type = KAODVM_DELROUTE;
areq.n.nlmsg_flags = NLM_F_REQUEST;

areq.m.dst = dest.s_addr;
areq.m.nhop = next_hop.s_addr;
areq.m.time = 0;
areq.m.flags = 0;

if (nl_send(&aodvnl, &areq.n) < 0) {
	DEBUG(LOG_DEBUG, 0, "Failed to send netlink message");
	return -1;
```

```
}
```

After all, aodv_route() is called to inform Zebra daemon to delete kernel route.

```
return aodv_route(ZEBRA_IPV4_ROUTE_DELETE, dest, next_hop, metric, ifindex);
```

c. Zebra route message

The function aodv_route() is used to ask Zebra daemon for kernel route entry maintenance (add or delete). The command needs to be supplied.

```
int aodv_route(u_char cmd, struct in_addr dest, struct in_addr next_hop, int metric, int ifindex)
{
	struct prefix_ipv4 *p;
	struct zapi_ipv4 api;
	struct in_addr *nexthop_pointer = &next_hop;

	if (! zclient->enable)
		return 0;

	if (zclient->sock <0)
		return 0;

	p = prefix_ipv4_new();
	p->prefixlen = 32;
	p->prefix.s_addr = dest.s_addr;

	if (zclient->redist[ZEBRA_ROUTE_AODV]) {
		api.type = ZEBRA_ROUTE_AODV;
	    api.flags = 0;
	    api.message = 0;
	    api.safi = SAFI_UNICAST;

	    /* Unlike the native Linux and BSD interfaces, Quagga doesn't like
		there to be both and IPv4 nexthop and an ifindex.  Omit the
	    ifindex, and assume that the connected prefixes be set up
	    correctly. */
	    SET_FLAG (api.message, ZAPI_MESSAGE_NEXTHOP);
	    api.ifindex_num = 0;
```

```
                api.nexthop_num = 1;
                api.nexthop = &nexthop_pointer;
                SET_FLAG (api.message, ZAPI_MESSAGE_METRIC);
                api.metric = metric;
            }

            return zapi_ipv4_route (cmd ? ZEBRA_IPV4_ROUTE_ADD :
                                    ZEBRA_IPV4_ROUTE_DELETE, zclient, p, &api);
}
```

In zapi_ipv4_route() module, it will use zclient (which has the socket to zebra) , prefix p (destination information) and zapi (holds next hop , metric and interface index) to generate a stream data that compatible with Zebra protocol.

```
Int zapi_ipv4_route (u_char cmd, struct zclient *zclient, struct prefix_ipv4 *p,
                struct zapi_ipv4 *api)
{
 int i;
 int psize;
 struct stream *s;

 /* Reset stream. */
 s = zclient->obuf;
 stream_reset (s);

/* Create zebra message header */
 zclient_create_header (s, cmd);

 /* Put type and nexthop. */
 stream_putc (s, api->type);
 stream_putc (s, api->flags);
 stream_putc (s, api->message);
 stream_putw (s, api->safi);

 /* Put prefix information. */
 psize = PSIZE (p->prefixlen);
 stream_putc (s, p->prefixlen);
 stream_write (s, (u_char *) & p->prefix, psize);
```

```
/* Nexthop, ifindex, distance and metric information. */
if (CHECK_FLAG (api->message, ZAPI_MESSAGE_NEXTHOP))
  {
    if (CHECK_FLAG (api->flags, ZEBRA_FLAG_BLACKHOLE))
      {
        stream_putc (s, 1);
        stream_putc (s, ZEBRA_NEXTHOP_BLACKHOLE);
        /* XXX assert(api->nexthop_num == 0); */
        /* XXX assert(api->ifindex_num == 0); */
      }
    else
      stream_putc (s, api->nexthop_num + api->ifindex_num);

    for (i = 0; i < api->nexthop_num; i++)
      {
        stream_putc (s, ZEBRA_NEXTHOP_IPV4);
        stream_put_in_addr (s, api->nexthop[i]);
      }
    for (i = 0; i < api->ifindex_num; i++)
      {
        stream_putc (s, ZEBRA_NEXTHOP_IFINDEX);
        stream_putl (s, api->ifindex[i]);
      }
  }

if (CHECK_FLAG (api->message, ZAPI_MESSAGE_DISTANCE))
  stream_putc (s, api->distance);
if (CHECK_FLAG (api->message, ZAPI_MESSAGE_METRIC))
  stream_putl (s, api->metric);

/* Put length at the first point of the stream. */
stream_putw_at (s, 0, stream_get_endp (s));

return zclient_send_message(zclient);
}
```

The generated message is store in stream *buffer of *zclient and is sent to zebra when zclient_send_message() module is called. Module buffer_write is called to write zebra message to zebra UNIX socket. The return value helps us do further actions (announce success, flush the buffer in case there is pending data in buffer, etc…)

```
Int zclient_send_message(struct zclient *zclient)
{
 if (zclient->sock < 0)
   return -1;

 switch (buffer_write(zclient->wb, zclient->sock, STREAM_DATA(zclient->obuf),
                          stream_get_endp(zclient->obuf)))
   {
   case BUFFER_ERROR:
     alog(LOG_NOTICE, 0, __FUNCTION__,
                      "buffer_write failed to zclient fd %d, closing",zclient->sock);
     return zclient_failed(zclient);
     break;
   case BUFFER_EMPTY:
     /*alog(LOG_NOTICE, 0, __FUNCTION__,
                      "buffer empty.");*/
     break;
   case BUFFER_PENDING:
     alog(LOG_NOTICE, 0, __FUNCTION__,
                      "TODO: Buffer not empty. Find a way to handle this case when multiple"
                      "daemons communicate with zebra.");
     break;
   }

 return 0;

}
```

3.3. Integration of OLSRd to Quagga 0.99.22.1

3.3.1. OLSR data flow

The following figure shows how OLSR control messages processed, generating and forwarding. The node's information bases are updated and the route calculation. OLSR is a proactive routing protocol which nodes will actively exchange information (in control messages) and they use the information to compute/update the network topology database and the routing table. OLSRd operates in user-space only (while AODVd needs a kernel space daemon to handle route discovery process).

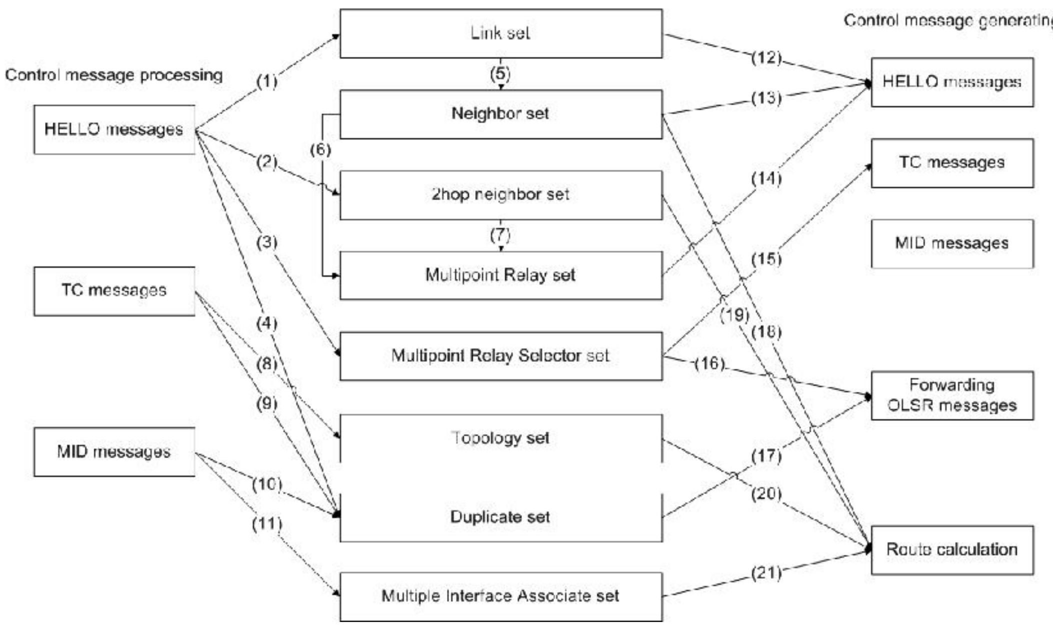


Figure 13: OLSR data flow

When OLSR daemon starts (routing process starts on an interface), function olsr_new() defined in olsrd.c is called to setup OLSR routing process – assign default parameters, initialize node's information bases and routing tables, etc...

```
struct olsr* olsr_new()
{
  zlog_debug("Enable OLSR routing");
```

Allocate memory for data structure

```
  struct olsr *new = XCALLOC(MTYPE_OLSR_TOP, sizeof(struct olsr));
```

Assign default OLSR parameters from RFC

```
  new->port = OLSR_PORT_DEFAULT;
  new->C = OLSR_C_DEFAULT;
  new->will = OLSR_WILL_DEFAULT;
```

```
 new->dup_hold_time = OLSR_DUP_HOLD_TIME_DEFAULT;
 new->top_hold_time = OLSR_TOP_HOLD_TIME_DEFAULT;
 new->mpr_update_time = OLSR_MPR_UPDATE_TIME_DEFAULT;
 new->rt_update_time = OLSR_RT_UPDATE_TIME_DEFAULT;
 new->neighb_hold_time = OLSR_NEIGHB_HOLD_TIME_DEFAULT;
 new->main_addr_is_set = FALSE;
 new->oiflist = list_new();
```

Initialize node's information bases

```
 new->neighset = list_new();
 new->n2hopset = list_new();
 new->dupset = list_new();
 new->midset = list_new();
 new->topset = list_new();
 new->advset = list_new();
```

Initialize routing table

```
 new->networks = route_table_init ();
 new->table = route_table_init ();
```

Create a raw UDP socket for checking control messages

```
 new->fd = olsr_sock_init();
```

Create thread for listening to incoming packets

```
 if (new->fd >= 0)
   new->t_read = thread_add_read (master, olsr_read, new, new->fd);
 return new;
}
```

3.3.2. Control message processing

When the thread for listening incoming packet created, olsr_read() function is called to handle the task. Because of the change in Quagga 0.99.22.1 libraries, the initial OLSRdq source code is updated in order to work with Quagga 0.99.22.1.

```
int  olsr_read (struct thread *thread)
```

The thread reschedule itself for continuously packet listening

```
 /* Fetch packet and reschedule thread. */
 olsr = THREAD_ARG (thread);
```

```
 olsr->t_read = NULL;
 olsr->t_read = thread_add_read (master, olsr_read, olsr, olsr->fd);
```

Incoming packet is stored to buffer

```
 /* read OLSR packet. */
 ibuf = olsr_recv_packet (olsr->fd, &ifp, &iph);
```

Perform condition check, if the packet received from an interface not enabled for OLSR routing, discard it

```
 if (ibuf == NULL)
   return -1;

 /* Packet not from an OLSR enabled IF */
 if (olsr_check_enable_ifp(ifp) < 0)
 {
     stream_free (ibuf);
     return 0;
 }

 /* Self-originated packet should be discarded silently. */
 if (olsr_if_check_address (iph.ip_src))
 {
   stream_free (ibuf);
   return 0;
 }
 oi = olsr_oi_lookup_by_ifp (olsr, ifp);
 if (oi == NULL)
   {
     /* OLSR is not enabled on this interface. */
     stream_free (ibuf);
     return 0;
   }
```

If packet is not an OLSR packets - UDP port is not 698 then discard

```
 stream_set_endp(ibuf, stream_get_size(ibuf));
```

```
/* skip IP header.*/
stream_forward_getp (ibuf, 20);

/* UDP header */
stream_forward_getp (ibuf, 2);        /* skip source port */
port = stream_getw (ibuf);            /* get des port */
len = stream_getw (ibuf) - 8;         /* len = get_len - 8bytes UDP header */
stream_forward_getp (ibuf, 2);        /* skip checksum. */

/* This is not OLSR packet UDP port 698 */
if (port != olsr->port)
{
        stream_free (ibuf);
        return 0;
}
```

Now we receive OLSR control message, function olsr_parse_packet() defined in aodv_packet.c is called to handle. The definition is as follow

```
int olsr_parse_packet (struct olsr *olsr, struct stream *ibuf, struct olsr_interface
*oi,   struct ip *iph, u_int16_t plen)
```

Because the buffer may contain multiple OLSR control messages, we get each valid message from buffer and check the message type in message header. Then call correspondent functions to handle each message type.

```
  p_len = stream_getw (ibuf);
psn = stream_getw (ibuf);

if (p_len != plen)
  zlog_warn ("Packet length from olsr header and the length from the UDP
header are"
           "not the same");

if (plen < OLSR_PACKET_MINSIZE)
  {
    zlog_warn ("packet size %d is smaller than minimum size %d",
               plen, OLSR_PACKET_MINSIZE);
    return plen;
```

```
  }

 /* Get current position in stream. */
 p_offset = stream_get_getp (ibuf) - 4;

 /* For each message. */
 while (stream_get_getp (ibuf) - p_offset < plen)
 {
	/* Process OLSR message") */
  msg_off = stream_get_getp (ibuf);
  oh = olsr_get_header (olsr, ibuf);

  if (IS_DEBUG_OLSR_PACKET (oh->mtype))
	{
	  /* Dump ip and udp headers. */
	  zlog_debug (" ");
	  zlog_debug ("---------------------------------------------");
	  zlog_debug ("ip_src: %s", inet_ntoa (iph->ip_src));
	  zlog_debug ("ip_dst: %s", inet_ntoa (iph->ip_dst));

	  zlog_debug ("mtype: %s vtime: %f  m_size: %d" ,
			OLSR_MSG_TYPE (oh->mtype), oh->vtime, oh->m_size);
	  zlog_debug ("orig: %s ttl: %d hops: %d msn %d", inet_ntoa (oh->oaddr),
			oh->ttl, oh->hops, oh->msn);
	}

  next_msg_off = msg_off + oh->m_size;

  /* Processing conditions. */
  if (oh->ttl == 0)
	{
	  /* Skip message. */
	  stream_set_getp (ibuf, next_msg_off);
	  continue;
	}

  switch (oh->mtype)
```

```
{
```

a. HELLO messages

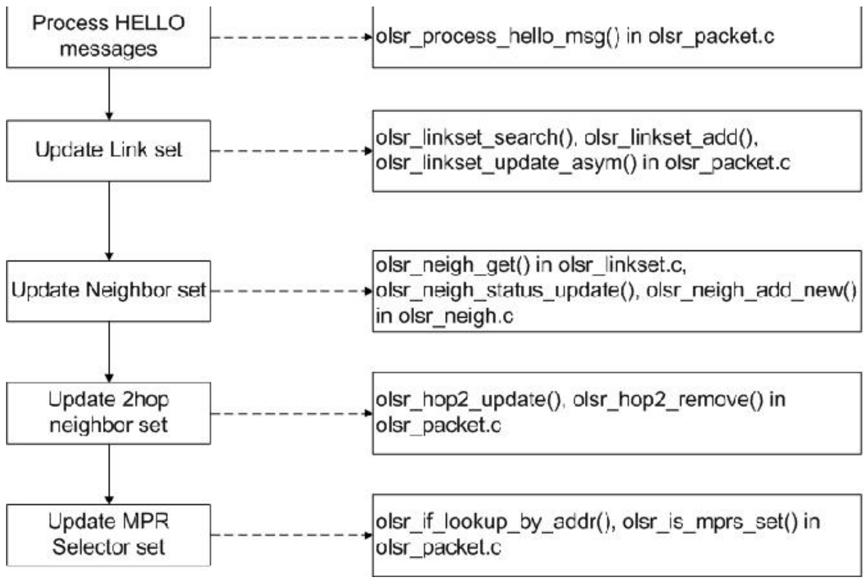


Figure 14: Hello message processing flow

The function olsr_process_hello_msg() defined in aodv_packet.c is called for processing hello message.

```
case OLSR_HELLO_MSG:
/* RFC 3626 8.2.1.   Notice, that a HELLO message MUST neither be forwarded
nor be recorded in the duplicate set.   */
olsr_process_hello_msg (olsr, oi, iph, oh, ibuf);
```

The definition in olsr_packet.c

```
void olsr_process_hello_msg (struct olsr *olsr, struct olsr_interface *oi,
        struct ip *iph, struct olsr_header *oh, struct stream *stream)
```

It lookup for a linkset contain this message originator, if not, then create a new link tuple (process 1 of figure 11)

```
  /* Find if this is form a SYM neighbor. */
  sym_neigh = olsr_neigh_lookup_addr (olsr, &oh->oaddr);

  /* Find a linkset contain this node */
```

```
  ol = olsr_linkset_search (oi, &iph->ip_src);
  if (ol == NULL)
  {
          ol = olsr_linkset_add (olsr, oi, iph, oh, ohh);
  }
```

olsr_linkset_add function defined in olsr_linkset.c is called for adding a new link tuple. It also update the neighbor set with function olsr_neigh_get() and olsr_neigh_add_new() (process 5 of figure)

```
struct olsr_neigh *olsr_neigh_get (struct olsr *olsr, struct olsr_link *ol,
                         struct olsr_header *oh, struct olsr_hello_header *ohh)
{
  struct olsr_neigh *on;
  on = olsr_neigh_lookup (olsr, &oh->oaddr);
  if (on != NULL)
    {
      listnode_add (on->assoc_links, ol);
      olsr_neigh_status_update (on);
      return on;
    }
  else
    return olsr_neigh_add_new (olsr, ol, oh, ohh);
}
```

In case a link tuple for the originator node exists, it is updated

```
  olsr_linkset_update_asym (ol, oh);
  /* Update willingness. */
  ol->neigh->will = ohh->will;
```

Now, information from the HELLO message is extracted for updating the 2hop neighbor set (process 2 of figure 11)

```
  if (sym_neigh && (olh->nt == OLSR_NEIGH_SYM || olh->nt ==
OLSR_NEIGH_MPR))
    olsr_hop2_update (olsr, sym_neigh, &neigh_addr, oh->vtime);
  if (sym_neigh && olh->nt == OLSR_NEIGH_NOT)
    olsr_hop2_remove (olsr, sym_neigh, &neigh_addr);
```

And also update the MPR Selector set (process 3 of figure 11) by marking a neighbor as

```
 if (olh->nt == OLSR_NEIGH_MPR && olsr_if_lookup_by_addr (olsr,
&neigh_addr))
   {
     if (sym_neigh == NULL)
          zlog_warn ("%s is not symetric neighbor but it selected me as MPR.",
                    inet_ntoa (oh->oaddr));
     else
          olsr_is_mprs_set (sym_neigh, oh->vtime);
   }
```

Function olsr_is_mprs_set() defined in olsr_mpr.c, it marks a node in neighbor set as MPR selector by turn on is_mprs flag as TRUE. Set timer for this node to reset MRP selector flag.

```
Void olsr_is_mprs_set (struct olsr_neigh *on, float vtime)
{   on->is_mprs = TRUE;
  THREAD_TIMER_OFF (on->t_mprs);
 OLSR_TIMER_ON (on->t_mprs, olsr_is_mprs_reset, on, vtime);
```

Then function olsr_top_add_adv() defined in olsr_route.c is called to generate and send TC message. This will mentioned later in section 3.3.4

```
 /* Advertise this neighbor.. */
 olsr_top_add_adv (on->olsr, on);
}
```

b. TC messages

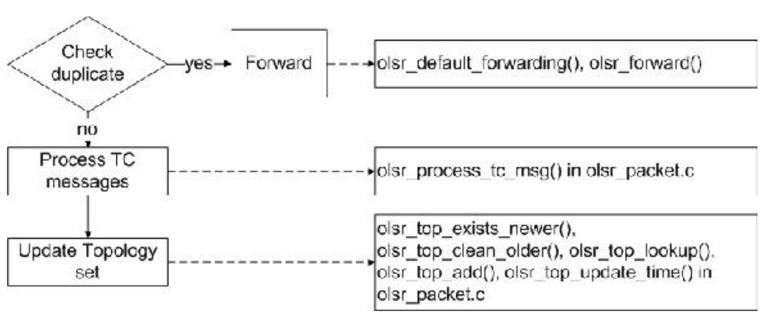


Figure 15: TC message processing flow

When processing TC messages, it first check whether there is a duplicate tuple with same source address and sequence number in the TC message with function olsr_dup_lookup() defined in aodv_dup.c.

```
struct olsr_dup *olsr_dup_lookup (struct olsr *olsr, struct in_addr *addr,
u_int16_t msn)
{
  struct listnode *node;
  struct olsr_dup *dup;

  for (node = listhead (olsr->dupset); node; nextnode (node))
    {
      dup = listgetdata (node);

      if (memcmp (&dup->addr, addr, 4) == 0 && dup->msn == msn)
        return dup;
    }
  return NULL;
}
```

If not then, olsr_process_tc_msg() is called to process the TC message. The definition is olsr_packet.c

```
 void  olsr_process_tc_msg (struct olsr *olsr, struct olsr_header *oh,
                     struct ip *iph, struct stream *stream)
```

Check and discard the message if the sender is not in neighbor set (1hop neighbor)

```
  olsr_mid_get_main_addr (olsr, &iph->ip_src, &addr);
  if (! olsr_neigh_lookup_addr (olsr, &addr))
    {
      if (IS_DEBUG_OLSR_PACKET (oh->mtype))
        zlog_debug ("Ignored because the sender is not my neighbor");
      return;
    }
```

Check and update topology set with information extracted from TC message. First, it discard received TC message is “older” the existing tuples from Topology set (TC message with sequence number less that what is in topology set) and also remove any tuple in topology set with sequence number less than the TC message's sequence number. (process 8 of figure 11)

```
 if (olsr_top_exists_newer (olsr, &oh->oaddr, ansn))
   return;
 olsr_top_cleanup_older (olsr, &oh->oaddr, ansn);
```

Then , it add new topology tuple or update existing ones

```
 while (stream_get_getp (stream) - m_offset < oh->m_size)
   {
     stream_get (&addr, stream, 4);

     if (IS_DEBUG_OLSR_PACKET (oh->mtype))
         zlog_debug ("addr: %s", inet_ntoa (addr));

     top = olsr_top_lookup (olsr, &addr, &oh->oaddr);

     if (top == NULL)
         olsr_top_add (olsr, oh->vtime, ansn, &addr, &oh->oaddr);
     else
         olsr_top_update_time (top, oh->vtime);
   }
}
```

If there is a duplicate tuple, the default forward algorithm must be triggered to determine whether the message should be forwarded or discarded. The definication of default forwarding is in olsr_dup.c

```
int olsr_default_forwarding (struct olsr *olsr, struct olsr_dup *dup, struct olsr_interface *oi, struct olsr_header *oh, struct in_addr *addr)
```

Discard message from a node not a 1hop neighbor

```
 if ((on = olsr_neigh_lookup_addr (olsr, &main_addr)) == NULL)
   return FALSE;
```

Only forward message that has same address and sequence number with an existing duplicate tuple and that tuple must has retransmitted as false the (address of the) interface which received the message is not included among the addresses in D_iface_list otherwise, not forward the message.

```
  if (dup && (dup->retransmitted || olsr_dup_has_if (dup, &OLSR_IF_ADDR
(oi))))
    return FALSE;
```

And update duplicate set (process 9, 16 of figure 11)

```
  ret = on->is_mprs && oh->ttl > 1;
  /* 5    Update Duplicate set. */
  if (dup)
    {
      dup->retransmitted = ret;

      THREAD_TIMER_OFF (dup->t_time);
      OLSR_TIMER_ON (dup->t_time, olsr_dup_del, dup, olsr->dup_hold_time);
      paddr = XCALLOC (MTYPE_PREFIX_IPV4, sizeof (struct in_addr));
      memcpy (paddr, &OLSR_IF_ADDR (oi), sizeof (struct in_addr));
      listnode_add (dup->iface_list, paddr);
    }
  else
    olsr_dup_add (olsr, &oh->oaddr, oh->msn, &OLSR_IF_ADDR (oi), ret);
  return ret;
```

Function olsr_forward() is called to forward the TC message. (process 17 of figure 11)

The definition in olsr_packet.c

```
Void olsr_forward (struct olsr *olsr, struct olsr_header *oh,  struct stream
*stream, u_int16_t msg_off)
```

It reduce ttl by 1 and increase hopcount by 1

```
  stream_putc_at (msg->obuf, 8, oh->ttl - 1);
  stream_putc_at (msg->obuf, 9, oh->hops + 1);
```

After that, the new message is push to all OLSR enabled interfaces output buffer (FIFO) and send out with function olsr_write() in another thread.

```
 for(ALL_LIST_ELEMENTS_RO(olsr->oiflist, node, oi))
   {
     memcpy (&msg->dst, &OLSR_IF_BROADCAST (oi), 4);
     olsr_msg_fifo_push (oi->fifo, msg);
     msg = olsr_msg_clone (msg);
     /* Schedule write to socket. */
     THREAD_WRITE_ON (olm->master, oi->t_write, olsr_write, oi,
                           oi->sock);
   }
```

c. MID messages

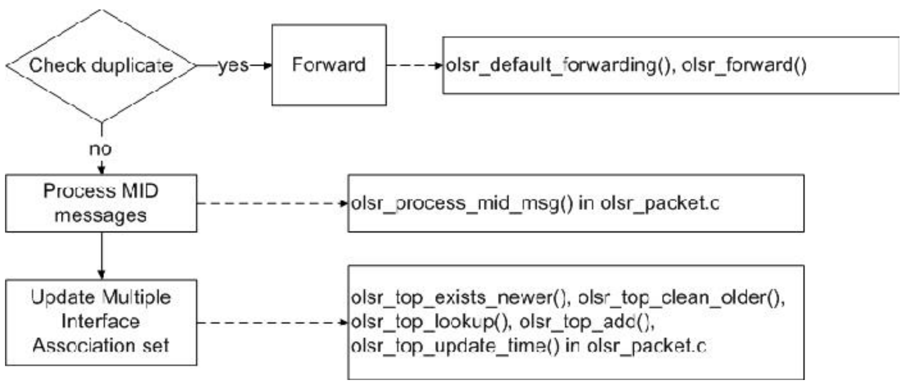


Figure 16: MID message processing flow

The processing of MID message is similar with TC message and starts with checking duplicate set. If there is existing duplicate tuple, the default forward algorithm is applied. (process 10 of figure 11)

If there is no duplicate tuple with same originator's address and sequence number with the MID message, olsr_process_mid_msg() defined in olsr_packet.c is called

```
void olsr_process_mid_msg (struct olsr *olsr, struct olsr_header *oh,
                           struct ip* iph, struct stream *stream)
```

Check and discard if the sender is not a 1hop neighbor

```
 olsr_mid_get_main_addr (olsr, &iph->ip_src, &addr);
 if (! olsr_neigh_lookup_addr (olsr, &addr))
   {
     if (IS_DEBUG_OLSR_PACKET (oh->mtype))
```

```
        zlog_debug ("Ignored because the sender is not my neighbor");
    return;
  }
```

3.3.3. Multipoint relay set generating

The MPR set is generated based on the neighbor set and 2hop neighbor set. Function olsr_mpr_update_if() defined in olsr_mpr.c (process 6, 7 of figure 11)

```
Void olsr_mpr_update_if (struct olsr_interface *oi)
{
 struct olsr *olsr = oi->olsr;
 struct list *N;                        /* Neighbors of interface oi */
 struct list *N2;                       /* 2-hop neigbors reachable via nodes from N */
 struct listnode *node1, *node2;
 struct olsr_1N *n1, *chosen;
 struct olsr_2N *n2;
```

Prepare list of 1-hop neighbor and 2-hop neighbor nodes

```
 N = olsr_mpr_get_if_neigh (olsr, oi);
 N2 = olsr_mpr_get_2hop_neigh (olsr, N);
 /* RFC 3626 8.3.
    1    Start with an MPR set made of all members of N with
    N_willingness equal to WILL_ALWAYS
 */
 for(ALL_LIST_ELEMENTS_RO(N, node1, n1))
   if (n1->on->will == OLSR_WILL_ALWAYS)
     n1->marked = TRUE;

 /*  2    Calculate D(y), where y is a member of N, for all nodes in N.*/
 for(ALL_LIST_ELEMENTS_RO(N, node1, n1))
   n1->D = listcount (n1->hop2lst);

 /*   3     Add to the MPR set those nodes in N, which are the *only* nodes to
provide reachability to a node in N2.  For example, if node b in N2 can be reached
only through a symmetric link to node a in N, then add node a to the MPR set.   */
 for(ALL_LIST_ELEMENTS_RO(N2, node2, n2))
   if (listcount (n2->hop1lst) == 1)
```

```
    {    n1 = listgetdata (listhead (n2->hop1lst));
         n1->marked = TRUE;
    }
 /* Remove nodes from N2 which are now covered by a node in the  MPR set. */
 olsr_mpr_N_N2_cleanup (N, N2);

 /*  4    While there exist nodes in N2 which are not covered by at least one node
in the MPR set. */
 while (! list_isempty (N2))
   {    /* 4.1  For each node in N, calculate the reachability, i.e., the
         number of nodes in N2 which are not yet covered by at
         least one node in the MPR set, and which are reachable
         through this 1-hop neighbor;
         (Done in cleanup.)
    */
    chosen = listgetdata (listhead (N));
    assert (chosen);
    for(ALL_LIST_ELEMENTS_RO(N, node1, n1))
         if (olsr_mpr_neigh_cmp(chosen, n1) < 0)
           chosen = n1;

    olsr_mpr_N_cleanup (N, N2, chosen);
   }
```

The definition of olsr_mpr_N_cleanup()

```
void  olsr_mpr_N_cleanup (struct list *N, struct list *N2, struct olsr_1N *n1)
 struct listnode *node, *next;
 struct olsr_2N *n2;
 /* Delete all n2 nodes referenced by n1. */
 for (node = listhead (n1->hop2lst); node; node = next)
   {
    n2 = listgetdata (node);
    next = nextnode (node);
    olsr_mpr_N2_cleanup (N, N2, n2);
   }
```

A node in neighbor set is marked as MPR.

```
n1->on->is_mpr = TRUE;
```

3.3.4. Control message generating

a. HELLO message

Hello message is generated using information from Link set and neighbor set with MPR flag of a neighbor.

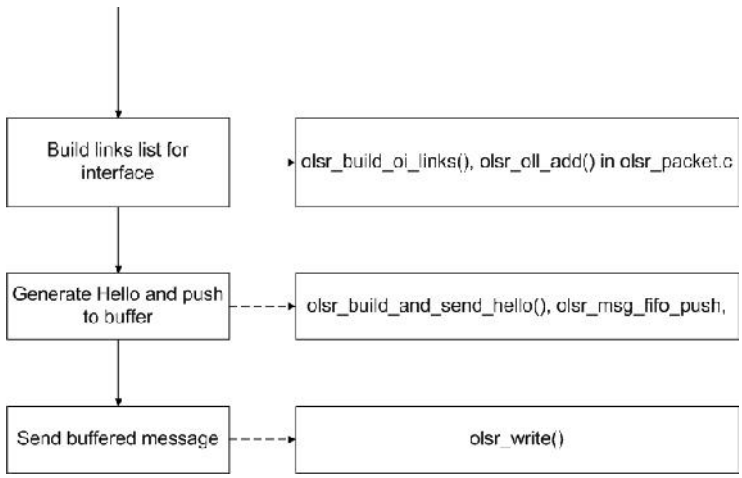


Figure 17: Generate Hello message

The function olsr_hello_timer() is triggered for Hello message generating and sending. For each interface that enabled for OLSR routing, a list of link tuples is formed and sent in Hello message data portion. Function olsr_build_oi_links() is called to build up this list.

```
struct list *olsr_build_oi_links (struct olsr_interface *oi)
```

First it travese the link set and add to the list with Link type, Neighbor type. (process 12 of figure 11)

```
  for (node = listhead (oi->linkset); node; nextnode (node))
  {
   struct olsr_oi_link *new = olsr_oi_link_new ();
   struct olsr_link *ol = listgetdata (node);
   if (!ol->sym_expired)
           new->lt = OLSR_LINK_SYM;
   else
           if (!ol->asym_expired)
```

```
              new->lt = OLSR_LINK_ASYM;
            else
              new->lt = OLSR_LINK_LOST;
      if (ol->neigh->is_mpr)
              new->nt = OLSR_NEIGH_MPR;
      else
              if (ol->neigh->status == OLSR_NEIGH_SYM)
                new->nt = OLSR_NEIGH_SYM;
              else
                if (ol->neigh->status == OLSR_NEIGH_NOT)
                new->nt = OLSR_NEIGH_NOT;
      memcpy (&new->neigh_addr, &ol->neigh_addr, sizeof (struct in_addr));
      olsr_oll_add (list, new);
```

Then, it traverse neighbor set, and add to the list with neighbor type and link type (process 13, 14 in figure 11)

```
  for (node = listhead (oi->olsr->neighset); node; nextnode (node))
  {
    struct olsr_neigh *on = listgetdata (node);
    struct listnode *lnode;
    int found = FALSE;
    for (lnode = listhead (on->assoc_links); lnode; nextnode (lnode)) {
            struct olsr_link *ol = listgetdata (lnode);
            if (ol->oi == oi)           {
              found = TRUE;
              break;
            }
          }
    if (! found)
          {
            struct olsr_oi_link *new = olsr_oi_link_new ();
            new->lt = OLSR_LINK_UNSPEC;
            if (on->is_mpr)
              new->nt = OLSR_NEIGH_MPR;
            else
              if (on->status == OLSR_NEIGH_SYM)
                new->nt = OLSR_NEIGH_SYM;
```

```
          else
            if (on->status == OLSR_NEIGH_NOT)
                    new->nt = OLSR_NEIGH_NOT;
        memcpy (&new->neigh_addr, &on->main_addr, sizeof (struct in_addr));
        olsr_oll_add (list, new); }
```

b. TC message

TC message is sent to neighbor that marked as MPR selector as mentioned in section 3.3.2. That node is added to advertised neighbor set and TC message is periodically sent to node in this advertised neighbor set

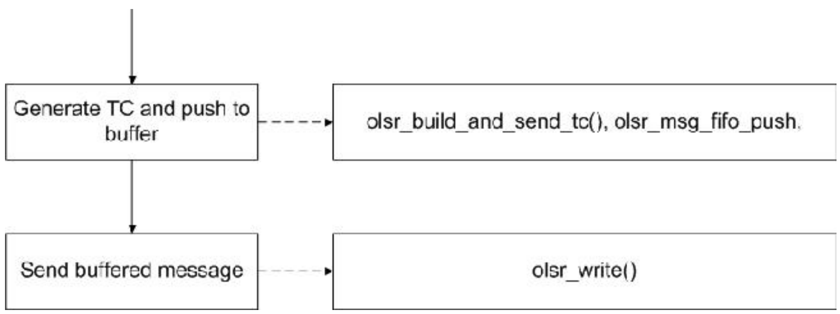


Figure 18: Generate TC message

```
Void olsr_top_add_adv (struct olsr *olsr, struct olsr_neigh *on)
{   struct olsr_interface *oi;
 struct listnode *node;
 if (!listnode_lookup (olsr->advset, on))     {
    listnode_add (olsr->advset, on);
    if (listcount (olsr->advset) == 1)  {
        THREAD_TIMER_OFF (olsr->t_delay_stc);
        /* Start oi tc timers. */
        for(ALL_LIST_ELEMENTS_RO(olsr->oiflist, node, oi))
          OLSR_TIMER_ON (oi->t_tc, olsr_tc_timer, oi, OLSR_IF_TC (oi));
        }

    /* RFC 3626 9.1.
        A sequence number is associated with the advertised neighbor
        set.  Every time a node detects a change in its advertised
        neighbor set, it increments this sequence number
    */
```

```
    olsr->ansn ++;
  }
}
```

The function olsr_build_and_send_tc() is called to construct the TC messages with information extracted from the advertised neighbor set and push into the interface output buffer (FIFO) with olsr_msg_fifo_push()

c. MID message

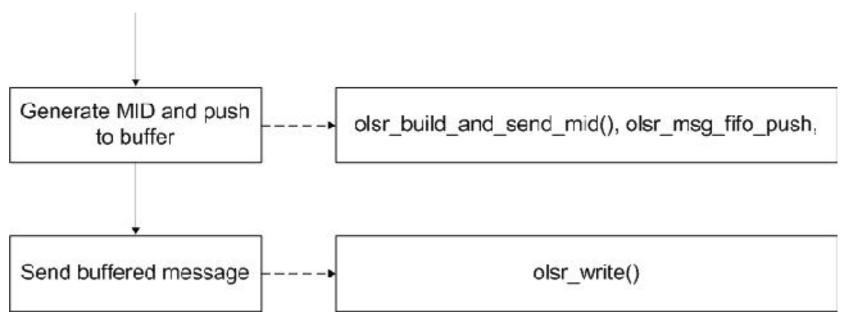


Figure 19: Generate MID message

MID message is generated when there are multiple interfaces take part in OLSR routing. Function olsr_build_and_send_mid() prepares MID message and push to interface output buffer with olsr_msg_fifo_push().

It starts by traverse the interface list, if there is a interface with IP address different with the node's main address.

```
 For(ALL_LIST_ELEMENTS_RO(oi->olsr->oiflist, node, oif))
   if (memcmp (&OLSR_IF_ADDR (oif), &oi->olsr->main_addr, 4) != 0)
     {
         if (STREAM_REMAIN (buf) < 4)
           {
             /* Fill message size. */
             stream_putw_at (buf, msize_off,stream_get_endp(buf) – msize_off +
2);

             /* Push to fifo. */
             olsr_msg_fifo_push (oi->fifo, msg);

             /* Create new message. */
```

```
            msg = olsr_msg_new (&OLSR_IF_BROADCAST (oi), oi->ifp->mtu –
4);
            buf = msg->obuf;
            olsr_put_msg_header (OLSR_MID_MSG, 255, 0,  buf, oi,
&msize_off);
          }

        /* Put Neigbor Interface Address. */
        stream_put (buf, &OLSR_IF_ADDR (oif), 4);
    }
 /* Fill message size. */
 stream_putw_at (buf, msize_off, stream_get_endp(buf) – msize_off + 2);
   /* Push it. */
 olsr_msg_fifo_push (oi->fifo, msg); }
```

3.3.5. Routing table calculation

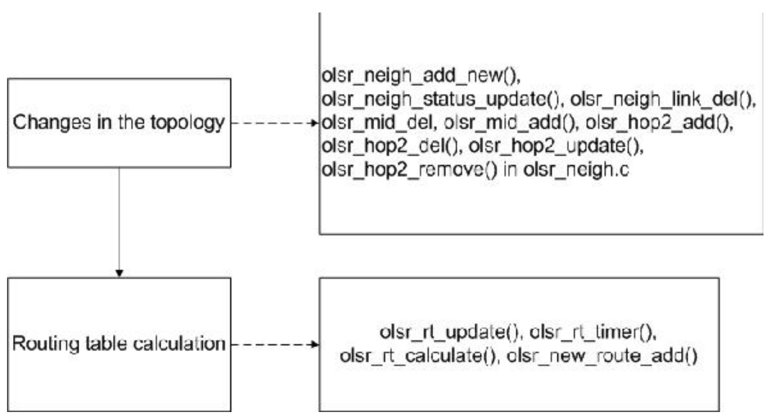


Figure 20: Routing table calculation

For any changes in the topology – update on neighbor set, 2hop neighbor set, topology set and mid set, OLSR needs to re-calculate its routing table.

Function olsr_rt_calculate() is called to compute a new routing table. The definition in olsr_route.c. Note that this is only OLSR routing table, not the kernel table. The synchronization between OLSR table with kernel table (via Zebra is in section 3.3.6)

```
struct route_table * olsr_route_calculate (struct olsr *olsr)
```

It starts by traverse the neighbor set and add symetric neighbor as the destination (process 18 in figure 11)

```
  for(ALL_LIST_ELEMENTS_RO(olsr->neighset, node, on))
   if (on->status == OLSR_NEIGH_SYM || on->status == OLSR_NEIGH_MPR)
     {
          found = FALSE;
          /* For each associated link tuple. */
          for(ALL_LIST_ELEMENTS_RO(on->assoc_links, node2, ol))
            {
              olsr_new_route_add (table,
                                  &ol->neigh_addr,        /* Dest addr.     */
                                  &ol->neigh_addr,        /* Next hop addr. */
                                  1,                      /* Distance.      */
                                  &OLSR_IF_ADDR (ol->oi));  /* Iface addr.    */

              if (memcmp (&ol->neigh_addr, &on->main_addr, 4) == 0)
                found = TRUE;
            }
          if (!found)
            olsr_new_route_add (table,
                                &on->main_addr,         /* Dest addr.     */
                                &ol->neigh_addr,        /* Next hop addr. */
                                1,                      /* Distance.      */
                                &OLSR_IF_ADDR (ol->oi));  /* Iface addr.    */
     }
```

Then it read through 2hop neighbor set (process 19 in figure 11)

```
  for(ALL_LIST_ELEMENTS_RO(olsr->n2hopset, node, hop2))
   {/* Get the route to its neighbor. */
     route = route_node_lookup_ipv4 (table, &hop2->neigh_addr);
     if (route == NULL)          {
     zlog_warn ("olsr_route_calculate: Hop 2 neighbor friend  %s not found in"
                  "the routing table", inet_ntoa (hop2->neigh_addr));
          continue;}
     rinfo = (struct olsr_route*) route->info;
     olsr_new_route_add (table,
                           &hop2->hop2_addr,        /* Dest addr.     */
```

```
                    &rinfo->next_hop,       /* Next hop addr. */
                    2,                      /* Distance.      */
                    &rinfo->iface_addr);    /* Iface addr.    */
  }
```

Next, the topology set is used to add route that further than 2 hop (process 20 in figure 11)

```
 hop = 2;
 while (change)
  {     change = FALSE;
   /* For each topology entry. */
   for(ALL_LIST_ELEMENTS_RO(olsr->topset, node, top))
      {
       /*   if  T_dest_addr does not correspond to R_dest_addr of any
        route entry in the routing table AND its T_last_addr
        corresponds to R_dest_addr
       */
       if (!route_node_lookup_ipv4 (table, &top->dest_addr) &&
         (route = route_node_lookup_ipv4 (table, &top->last_addr)) != NULL)
        {     rinfo = (struct olsr_route*) route->info;
         olsr_new_route_add (table,
                          &top->dest_addr,        /* Dest addr.     */
                          &rinfo->next_hop,       /* Next hop addr. */
                          hop + 1,                /* Distance.      */
                          &rinfo->iface_addr);    /* Iface addr.    */
         change = TRUE;
        }
      }
  }
```

And last, the MID set is used to add new route (process 21 in figure 11)

```
 for(ALL_LIST_ELEMENTS_RO(olsr->midset, node, mid))
  {
   /* if there exists a routing entry such that R_dest_addr  == I_main_addr
     AND there is no routing entry such that R_dest_addr  == I_iface_addr
   */
   if (!route_node_lookup_ipv4 (table, &mid->addr) &&
       (route = route_node_lookup_ipv4 (table, &mid->main_addr)) != NULL) {
```

```
            rinfo = (struct olsr_route*) route->info;
            olsr_new_route_add (table,
                                   &mid->addr,              /* Dest addr.     */
                                   &rinfo->next_hop,        /* Next hop addr. */
                                   rinfo->dist,             /* Distance.      */
                                   &rinfo->iface_addr);     /* Iface addr.    */
        }
  }
```

3.3.6. *Kernel route update*

After OLSR computes a new routing table (in function olsr_rt_timer), it compare with the old routing table as follow. At this stage, we note that all existing routes in the old table already synchronized with kernel routing table.

- All routes in the old table but not in the new table must be removed from kernel table. Function olsr_zebra_ipv4_delete() is called to tell Zebra daemon does kernel route removing.

```
 /* All nodes that are in the old table but not in the new one must be removed. */
 for (rn = route_top (olsr->table); rn; rn = route_next (rn))
   if ((rinfo = (struct olsr_route*)rn->info) != NULL &&
          ! route_node_lookup (table, &rn->p))
     {
          olsr_zebra_ipv4_delete ((struct prefix_ipv4 *)&rn->p, &rinfo->next_hop,
                                        rinfo->dist);
     }
```

- All routes in the new table but not in the old table must be added to kernel routing table. Function olsr_zebra_ipv4_add() is called to tell Zebra daemon does kernel route adding.

```
 /* All nodes that are in the new table but not in the old one must added. */
 for (rn = route_top (table); rn; rn = route_next (rn))
   if ((rinfo = (struct olsr_route*)rn->info) != NULL &&
          ! route_node_lookup (olsr->table, &rn->p))       {
          int ifindex = olsr_get_ifindex_by_addr (olsr, &rinfo->iface_addr);
          if (ifindex == -1)
            zlog_warn ("Can't find ifindex for address %s found in routing table.
```

```
                 Skipping this route", inet_ntoa (rinfo->iface_addr));
        else
          olsr_zebra_ipv4_add ((struct prefix_ipv4 *)&rn->p, &rinfo->next_hop,
                               rinfo->dist, ifindex);
    }
```

Chapter 4. REALISTIC TESTING SCENARIOS AND RESULTS

In this chapter, I first demonstrate the installation procedure for adhoc-iu package as well as running Zebra and configure interfaces with Zebra commands. Later sections are for proposed test-cases of AODVd and OLSRd. Please see a brief summary of this chapter as follow:

Section	Purpose	Details
4.1	Install and run adhoc-iu	- Install adhoc-iu package - Run and configure interface with Zebra daemon
4.2	AODV multi-hop routing	- Routing with AODVd and Zebra. - Three nodes testcase with node movement.
4.3	OLSR multi-hop routing	- Routing with OLSRd and Zebra. - Three nodes testcase with node movement.
4.4	Internet sharing with Zebra	- Default route configuration with Zebra. - Internet access for adhoc nodes.

Table 3: Testing scenario summary

4.1. Installation of adhoc-iu

- Download the package adhoc-iu.0.1.tar.gz and un-compress the package. The package contains updated Quagga 0.99.22.1 source code, updated OLSRdq 0.1.18 under folder olsrd and update AODV-UU 0.9.6 source under folder aodvd.

```
/data/anh/$tar -zxvf adhoc-iu.0.1.tar.gz
/data/anh/$cd adhoc-iu
```

- Configure the package to run with a specific user (this step is optional), by default adhoc-iu is run with root privilege, binary files are installed to /usr/local/sbin/ and configuration files are installed to /usr/local/etc/ .

```
/data/anh/adhoc-iu$./configure --enable-user=[user] --enable-group=[user]
```

- Compile and install adhoc-iu.

```
/data/anh/adhoc-iu$make
/data/anh/adhoc-iu$sudo make install
/data/anh/adhoc-iu$cd aodvd
/data/anh/adhoc-iu/aodvd$make
/data/anh/adhoc-iu/aodvd$sudo make install make
```

- Now adhoc-iu is installed and ready to run. Please note that Zebra daemon must be run before other routing daemons.
- Necessary steps to prepare the interface for adhoc, i.e.: we configure interface wlan0 for adhoc routing.

```
~$sudo stop network-manager
~$sudo ifconfig wlan0 down
~$sudo iwconfig wlan0 mode ad-hoc
~$sudo iwconfig wlan0 essid adhoc
~$sudo ifconfig wlan0 up
```

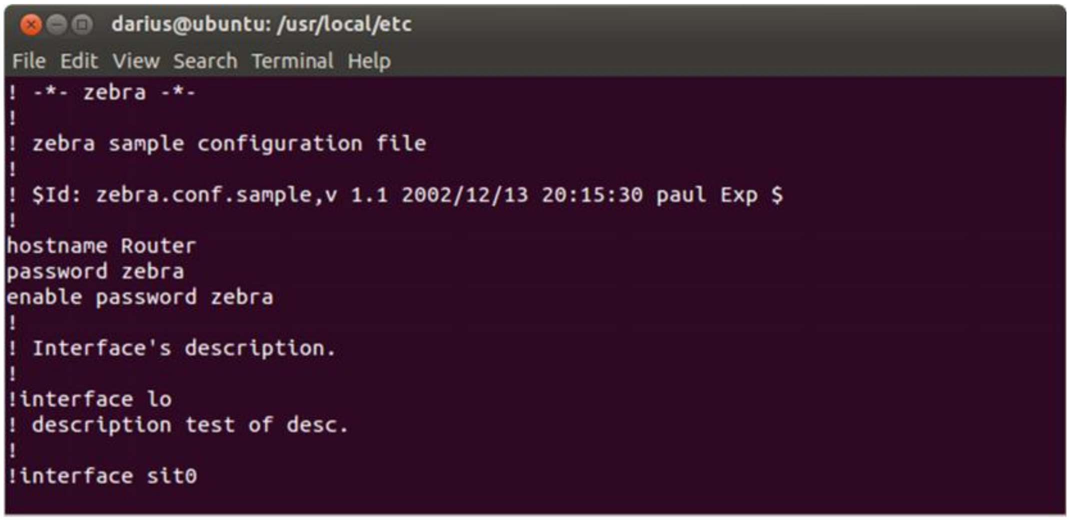
Run Zebra daemon and configure interface IP address

Zebra daemon is run by enter the command zebra from terminal. We can add option "-d" to make it run in daemon mode.

First, we have to prepare the configuration file for Zebra daemon

```
~$cd /usr/local/etc
/usr/local/etc$sudo cp zebra.conf.sample zebra.conf
```

Now, view the default zebra.conf to find the password to telnet to Zebra vtysh interface. By default, the telnet password and the "enable" password is zebra.



```
darius@ubuntu: /usr/local/etc
File Edit View Search Terminal Help
! -*- zebra -*-
!
! zebra sample configuration file
!
! $Id: zebra.conf.sample,v 1.1 2002/12/13 20:15:30 paul Exp $
!
hostname Router
password zebra
enable password zebra
!
! Interface's description.
!
!interface lo
! description test of desc.
!
!interface sit0
```

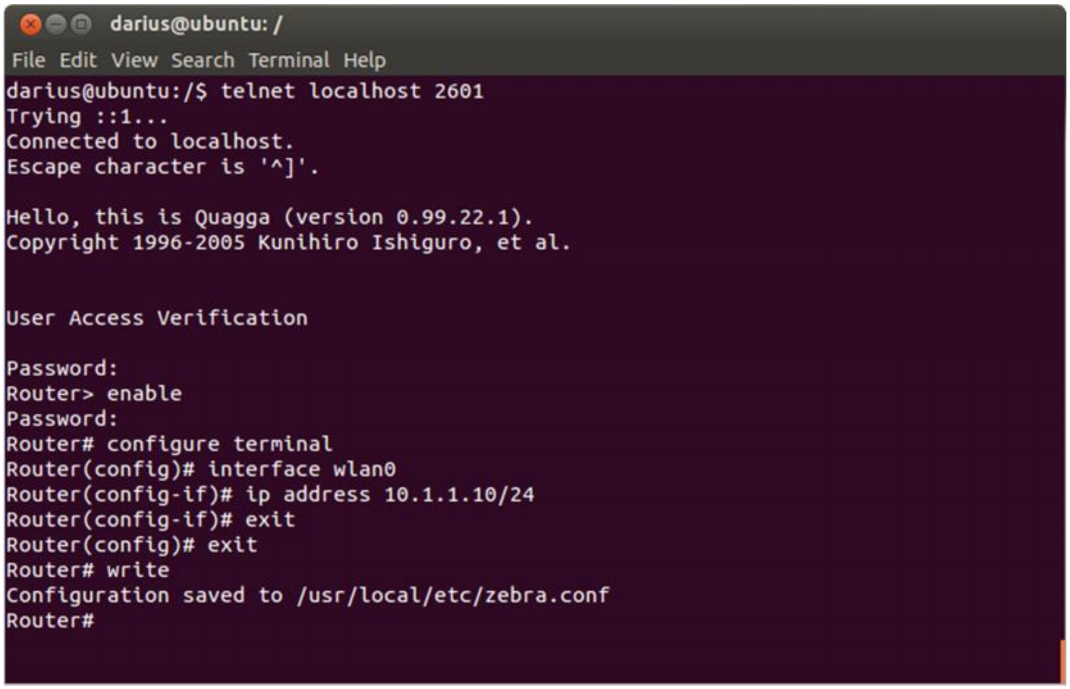
Now start Zebra daemon.

~$zebra –d

Note: if encounter error zebra: error while loading shared libraries: libzebra.so.0: cannot open shared object file: No such file or directory. Try running "ldconfig" with root permission and re-run zebra.

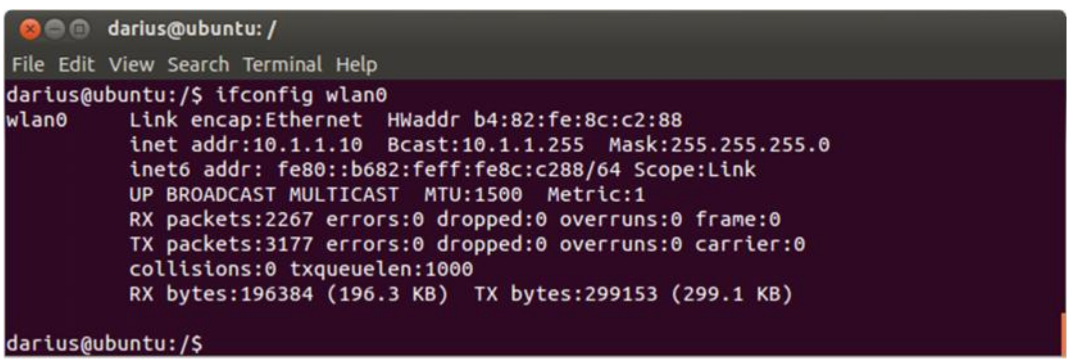
Zebra provides an VTYSH interface (a command line interface) for user and administrator to manage Zebra daemon. It supports a list of commands for manipulate interfaces (turn on/off; update IP address, etc...) , configure static route, configure access control list, route map, etc... Please refer Quagga manual for more information

Try telnet to Zebra VTYSH to update interface wlan0 IP address. By default, Zebra VTYSH listens to TCP port 2601.



```
darius@ubuntu: /
File Edit View Search Terminal Help
darius@ubuntu:/$ telnet localhost 2601
Trying ::1...
Connected to localhost.
Escape character is '^]'.

Hello, this is Quagga (version 0.99.22.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
Router> enable
Password:
Router# configure terminal
Router(config)# interface wlan0
Router(config-if)# ip address 10.1.1.10/24
Router(config-if)# exit
Router(config)# exit
Router# write
Configuration saved to /usr/local/etc/zebra.conf
Router#
```

From zebra VTYSH interface, we configure wlan0 with IP 10.1.1.10 (netmask /24 or 255.255.255.0). The below linux command “ifconfig wlan0” result shows new IP address for interface wlan0.



```
darius@ubuntu: /
File Edit View Search Terminal Help
darius@ubuntu:/$ ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr b4:82:fe:8c:c2:88
          inet addr:10.1.1.10  Bcast:10.1.1.255  Mask:255.255.255.0
          inet6 addr: fe80::b682:feff:fe8c:c288/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:2267 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3177 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:196384 (196.3 KB)  TX bytes:299153 (299.1 KB)

darius@ubuntu:/$
```

4.2. AODV routing test case

4.2.1. Purpose

In this real test case, we run AODV routing with three nodes. Capture the result to see the cooperation between AODVd and Zebra daemon.

4.2.2. Preparation

Setup three machines A, B, C with following:

- OS: Ubuntu 10.04 with kernel 2.6.32-38-generic
- Install adhoc-iu-0.1 on three machines.
- Configure IP address for machines: A (10.1.1.10/24) , B (10.1.1.30/24) and C (10.1.1.20/24). Please refer to section 4.1 for Zebra running and configuring interface's IP address.

4.2.3. Test case

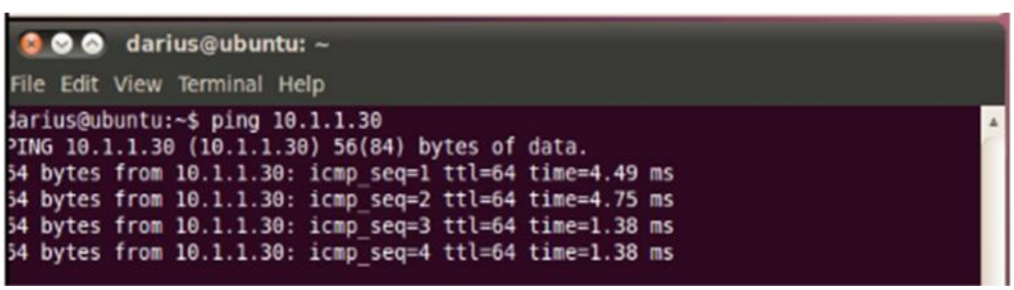
- Machine A and B placed near each other (within coverage). Both of them running AODVd and Zebra.
- Machine A pings machine B: successfully
- Machine B pings machine A: successfully
- Capture Zebra routing table on machine A and B.
- Move machine B away from machine A's coverage
- Machine A pings machine B: fail.
- Machine B pings machine A: fail.
- Now place machine C in the middle of A and B with C is in both A and B's coverage. Run AODVd and Zebra on C.

4.2.4. Expected result

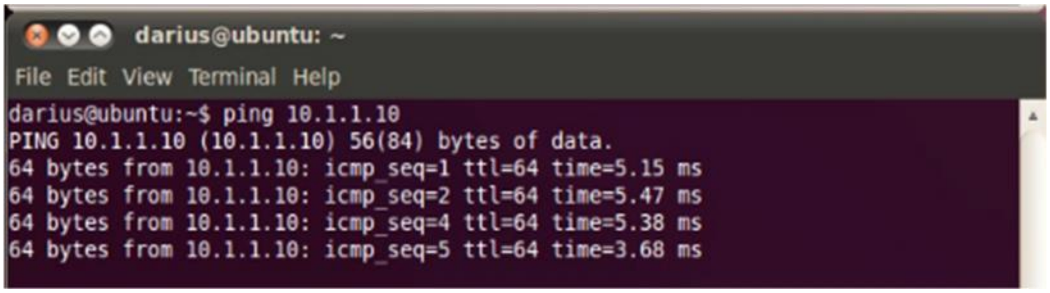
- Machine A can ping machine B.
- Machine A's zebra routing table has route to B via C.
- Machine B can ping machine A
- Machine B's zebra routing table has route to A via C.

4.2.5. Result

Machine A pings machine B

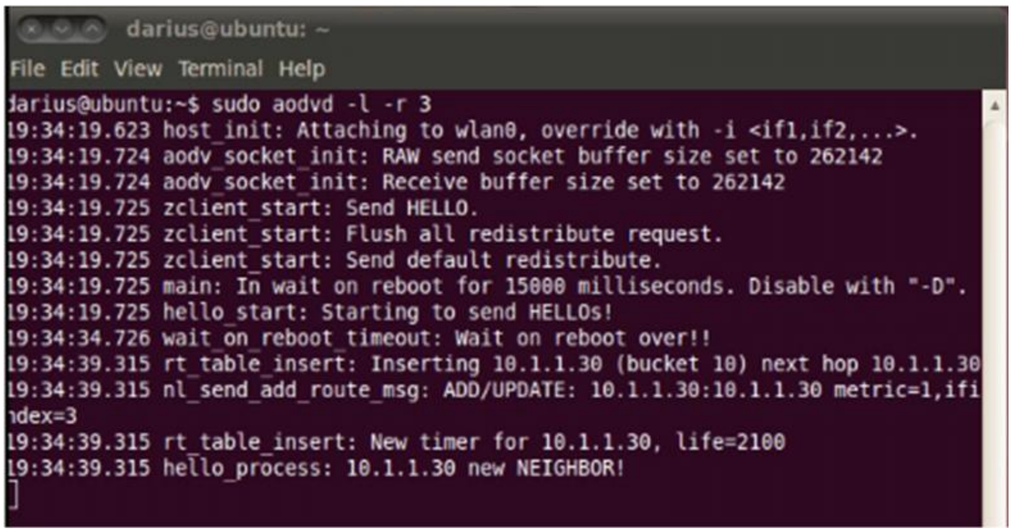


Machine B pings machine A successfully while they are in other's coverage.



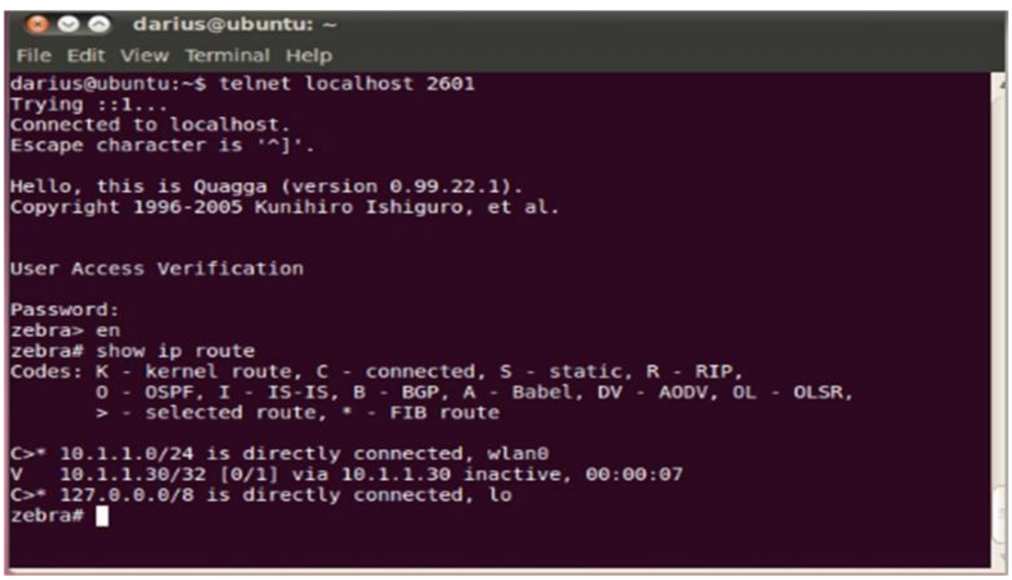
```
darius@ubuntu: ~
File  Edit  View  Terminal  Help
darius@ubuntu:~$ ping 10.1.1.10
PING 10.1.1.10 (10.1.1.10) 56(84) bytes of data.
64 bytes from 10.1.1.10: icmp_seq=1 ttl=64 time=5.15 ms
64 bytes from 10.1.1.10: icmp_seq=2 ttl=64 time=5.47 ms
64 bytes from 10.1.1.10: icmp_seq=4 ttl=64 time=5.38 ms
64 bytes from 10.1.1.10: icmp_seq=5 ttl=64 time=3.68 ms
```

Run AODVd on machine A, it insert neighbor 10.1.1.30 (machine B)



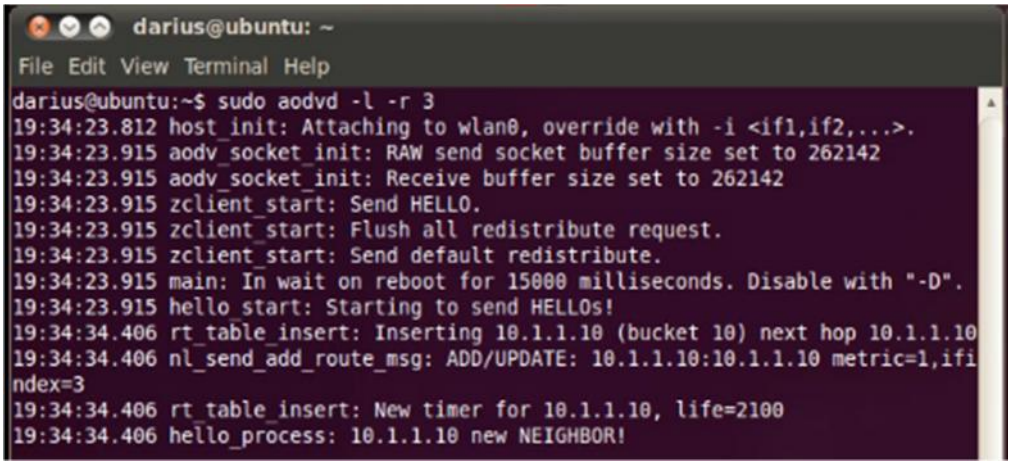
```
darius@ubuntu: ~
File  Edit  View  Terminal  Help
darius@ubuntu:~$ sudo aodvd -l -r 3
19:34:19.623 host_init: Attaching to wlan0, override with -i <if1,if2,...>.
19:34:19.724 aodv_socket_init: RAW send socket buffer size set to 262142
19:34:19.724 aodv_socket_init: Receive buffer size set to 262142
19:34:19.725 zclient_start: Send HELLO.
19:34:19.725 zclient_start: Flush all redistribute request.
19:34:19.725 zclient_start: Send default redistribute.
19:34:19.725 main: In wait on reboot for 15000 milliseconds. Disable with "-D".
19:34:19.725 hello_start: Starting to send HELLOs!
19:34:34.726 wait_on_reboot_timeout: Wait on reboot over!!
19:34:39.315 rt_table_insert: Inserting 10.1.1.30 (bucket 10) next hop 10.1.1.30
19:34:39.315 nl_send_add_route_msg: ADD/UPDATE: 10.1.1.30:10.1.1.30 metric=1,ifi
ndex=3
19:34:39.315 rt_table_insert: New timer for 10.1.1.30, life=2100
19:34:39.315 hello_process: 10.1.1.30 new NEIGHBOR!
```

View machine A's zebra routing table

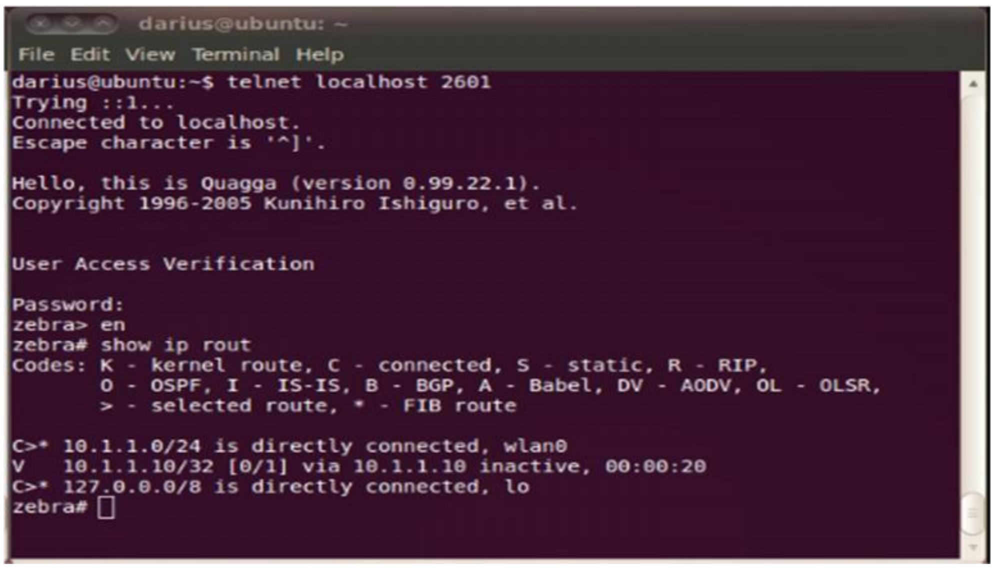


```
darius@ubuntu: ~
File  Edit  View  Terminal  Help
darius@ubuntu:~$ telnet localhost 2601
Trying ::1...
Connected to localhost.
Escape character is '^]'.

Hello, this is Quagga (version 0.99.22.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.


User Access Verification

Password:
zebra> en
zebra# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, A - Babel, DV - AODV, OL - OLSR,
       > - selected route, * - FIB route

C>* 10.1.1.0/24 is directly connected, wlan0
V   10.1.1.30/32 [0/1] via 10.1.1.30 inactive, 00:00:07
C>* 127.0.0.0/8 is directly connected, lo
zebra#
```

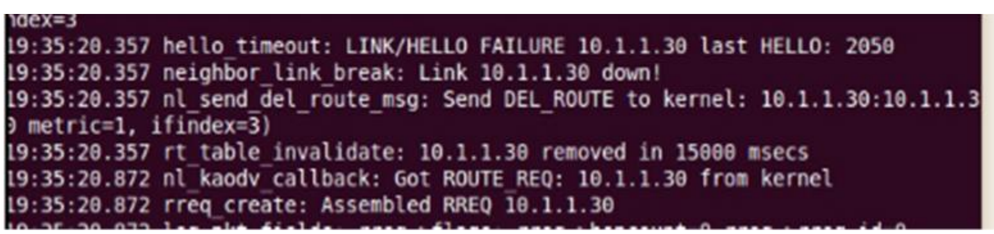
On machine B, it insert new neighbor 10.1.1.10 (machine A)



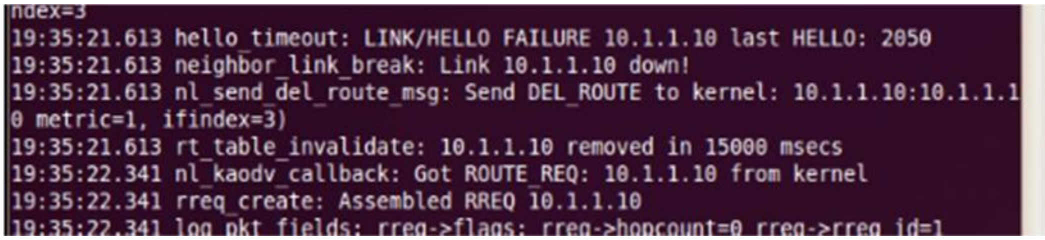
And machine's B zebra routing table has new entry to A



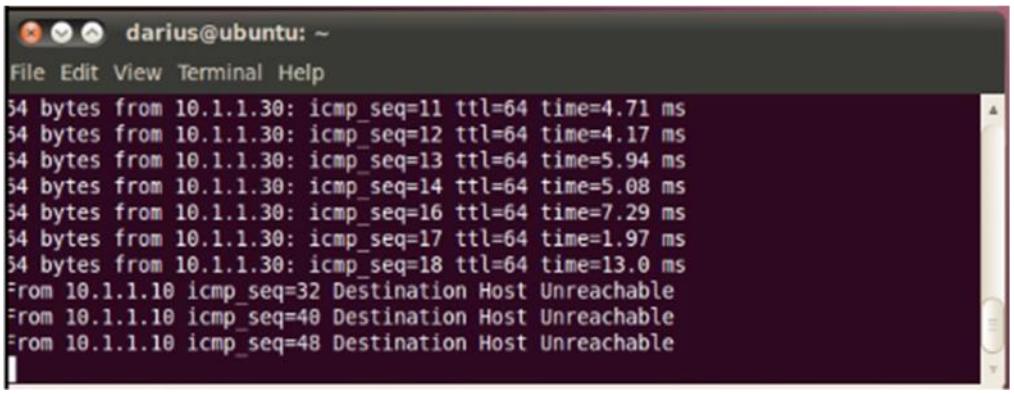
Now, move machine B away from machine A. On machine A, AODVd regconizes a link break and send request to Zebra to delete route



On machine B, AODVd also recognized a link break and send request to Zebra to delete route entry

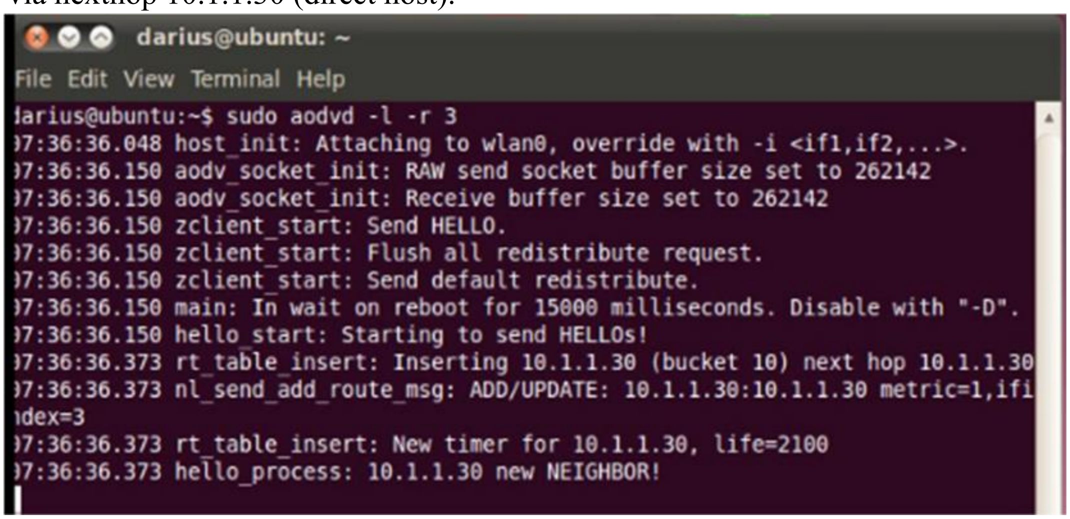


Ping from A to B and vice versa all fail.

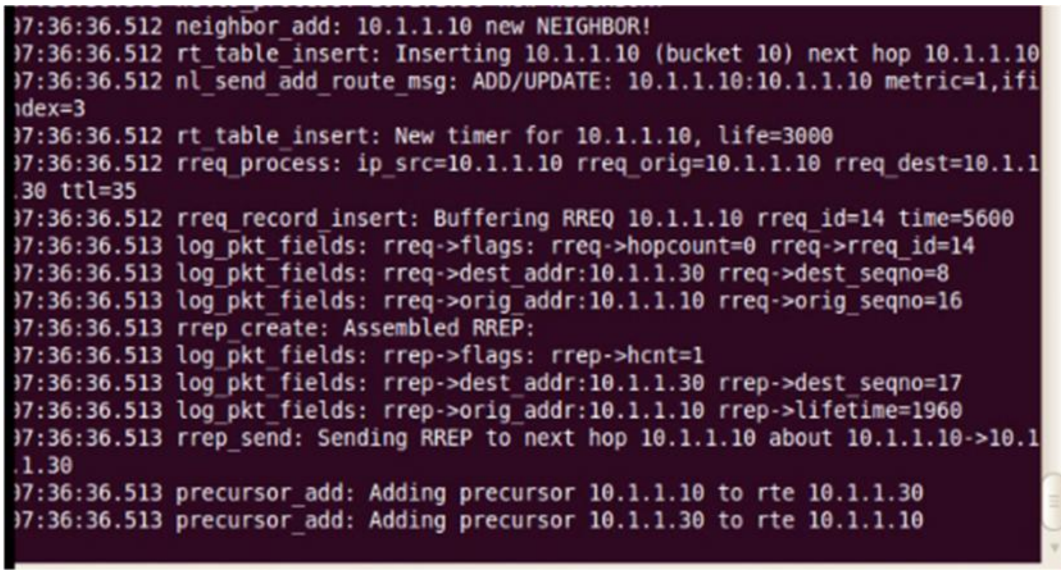


Now, we put machine C between A and B then run AODVd on machine C.

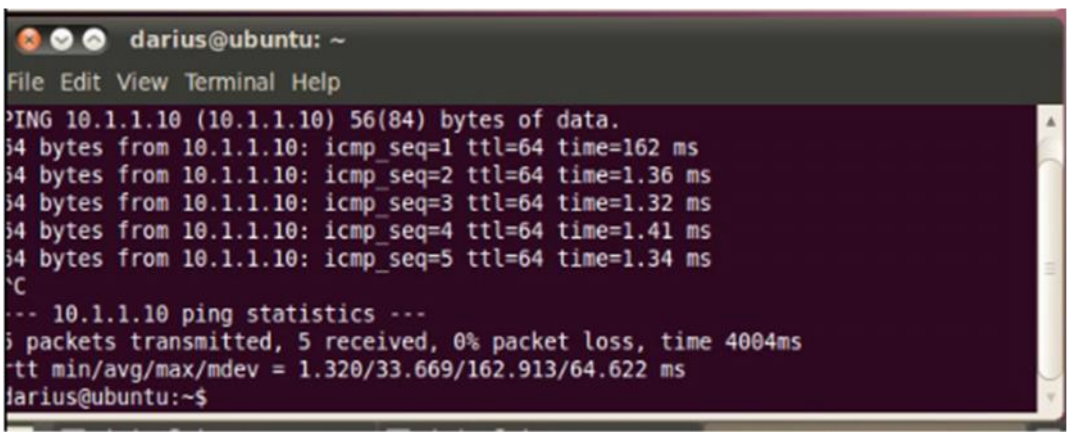
It insert neighbor 10.1.1.30 (machine B), send request to add new route 10.1.1.30 via nexthop 10.1.1.30 (direct host).

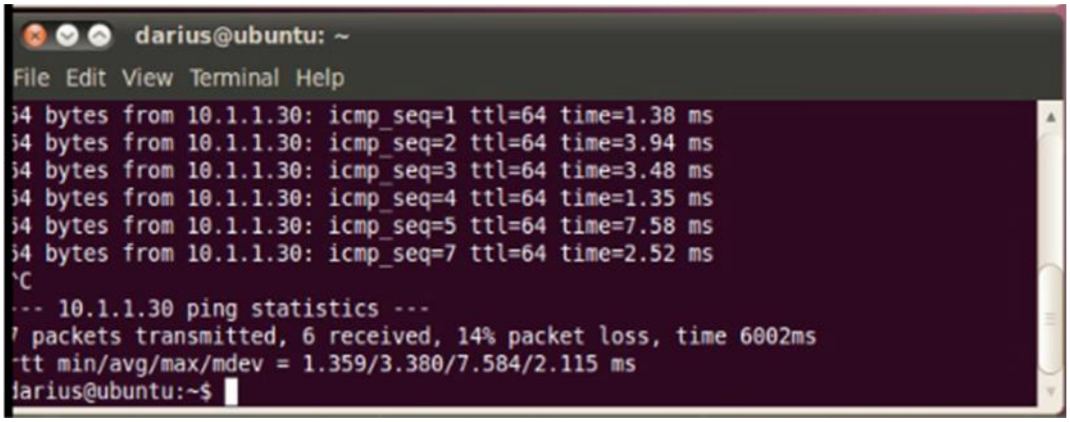


Machine C also inserts neighbor 10.1.1.10 (machine A)., new route 10.1.1.10 via 10.1.1.10 (machine A).

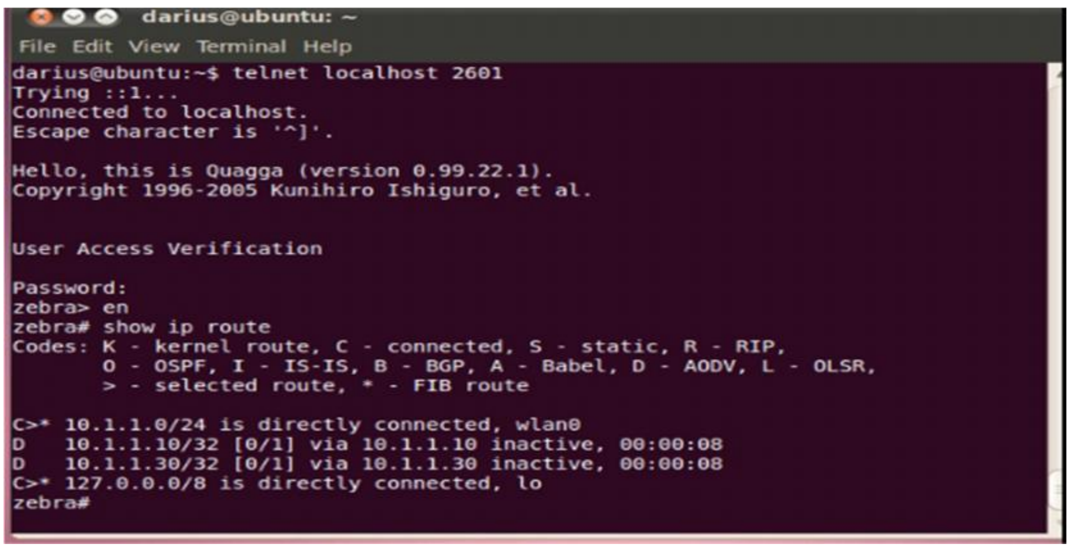


On machine C, ping successfully to both A and B

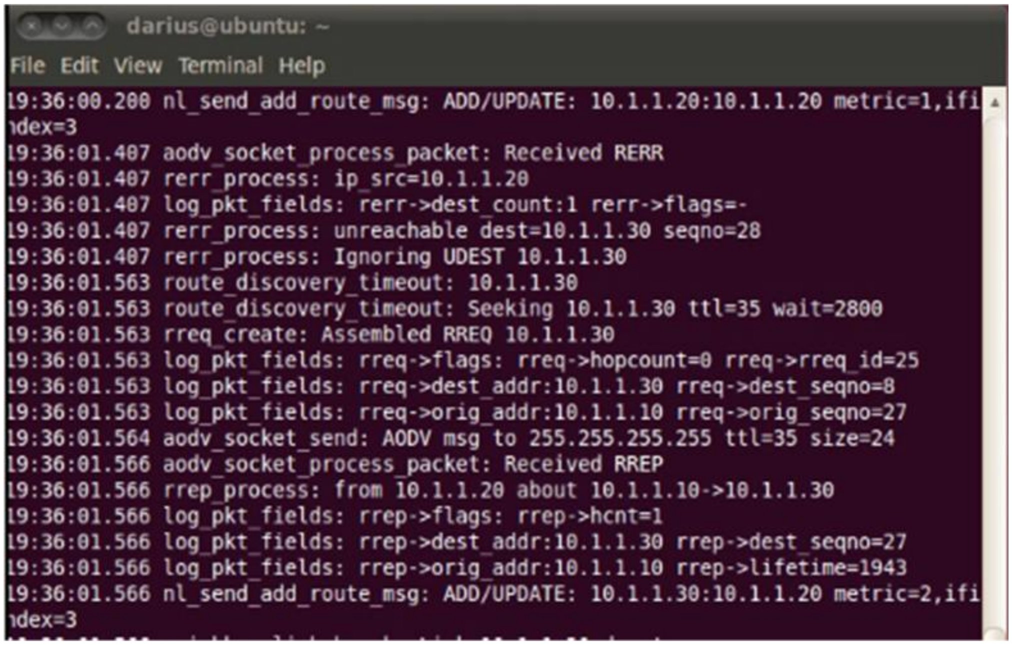




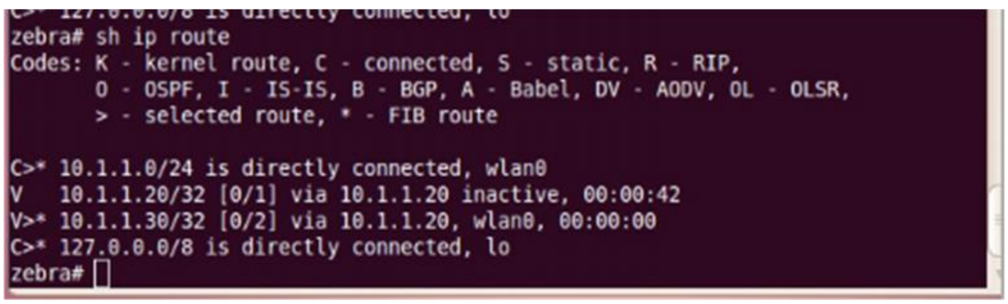
View Zebra routing table on machine C



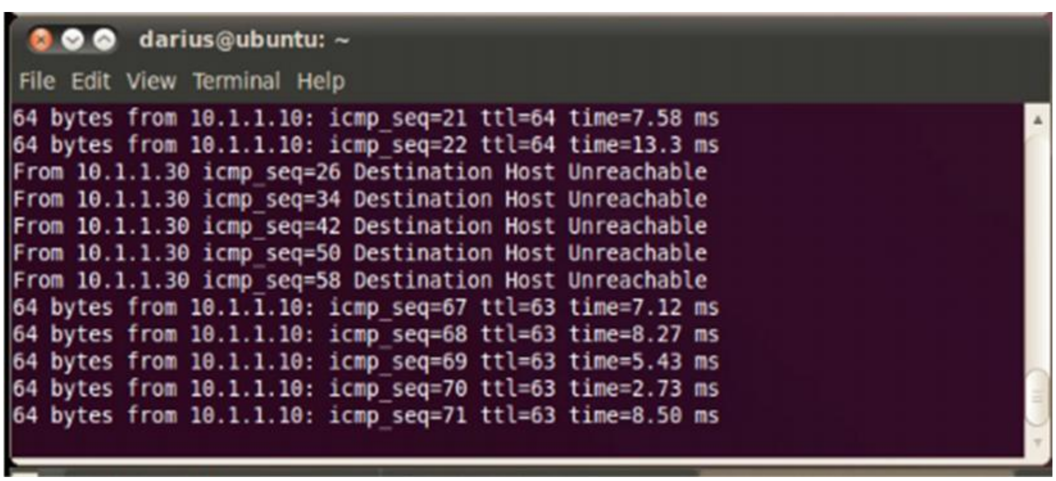
On machine A, it add new neighbor 10.1.1.20 (machine C), update new route to machine B (10.1.1.30) via C (10.1.1.20).



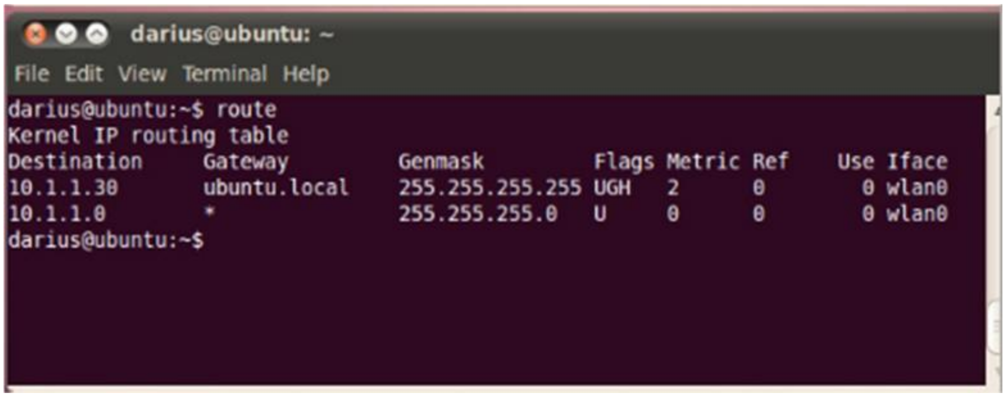
View zebra routing table on machine A



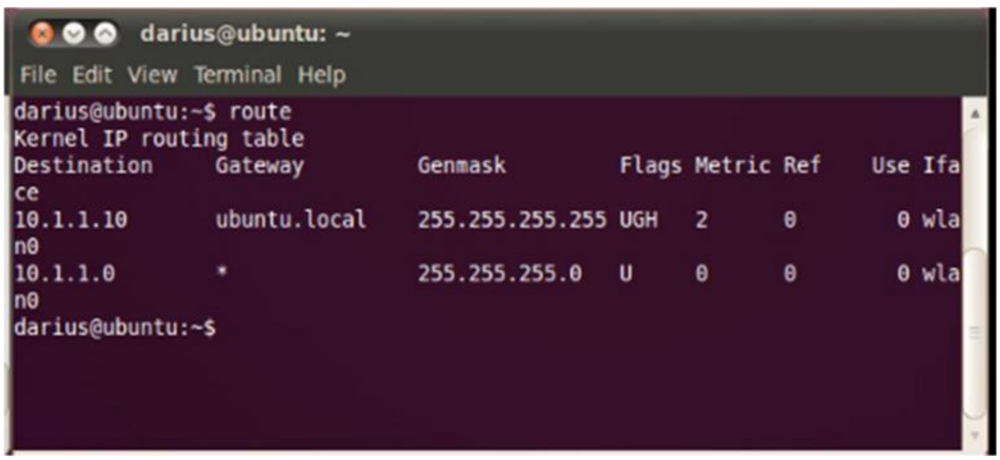
And the ping resume between A and B



Now, we view the kernel routing table on machine A.



And kernel routing table on machine B



4.3. OLSR routing testcase

4.3.1. Purpose

In this real test, we run OLSRd and Zebra on three nodes A, B, C. Capture the OLSR routing result and see the kernel routing table update.

4.3.2. Preparation

Setup the three nodes A, B, C with following:

- OS: Ubuntu 10.04 (kernel 2.6.x) or Ubuntu 12.04 (kernel 3.2).
- Install adhoc-iu-0.1 on the three nodes.
- Configure IP address for three nodes A (10.1.1.10/24), B (10.1.1.20/24) and C (10.1.1.30/24).

4.3.3. *Testcase*

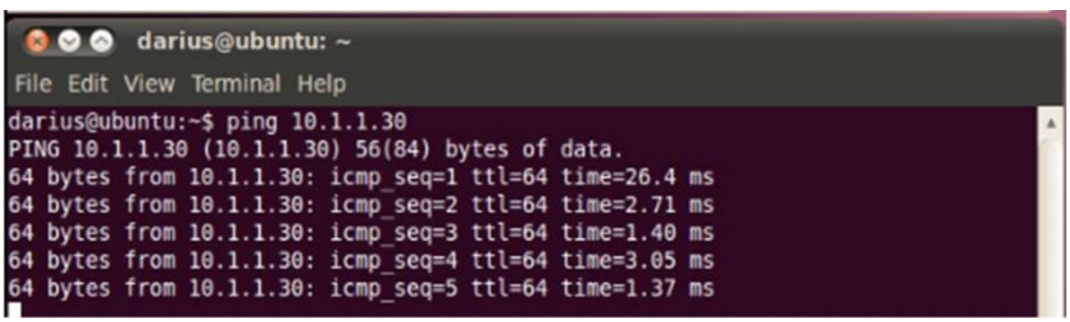
- Put node A near node C.
- Run OLSRd and Zebra on node A and C.
- The ping test from A to C successful.
- Now move C away from node A's coverage
- The ping test from A to C fails.
- Then move B between A and C.
- Run OLSRd and Zebra on node B.

4.3.4. *Expected result*

- The ping test from A to C successful again.
- During the moving, capture the OLSR routing operation result to see a route from A to C via node B
- Capture the kernel routing table to see a route from A to C via node B

4.3.5. *Result*

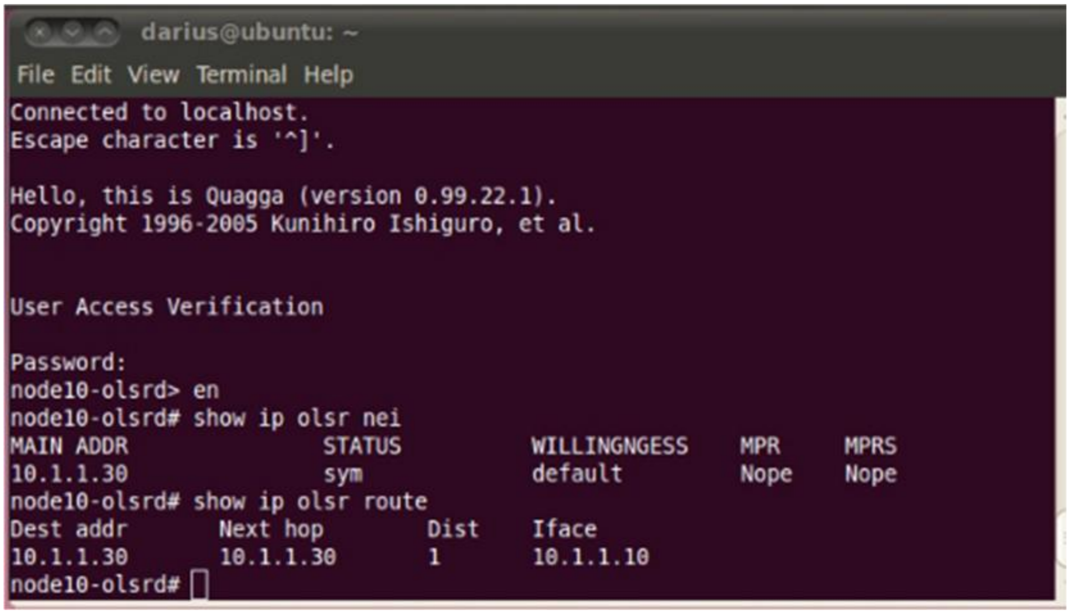
When node A near node C. The ping test successful



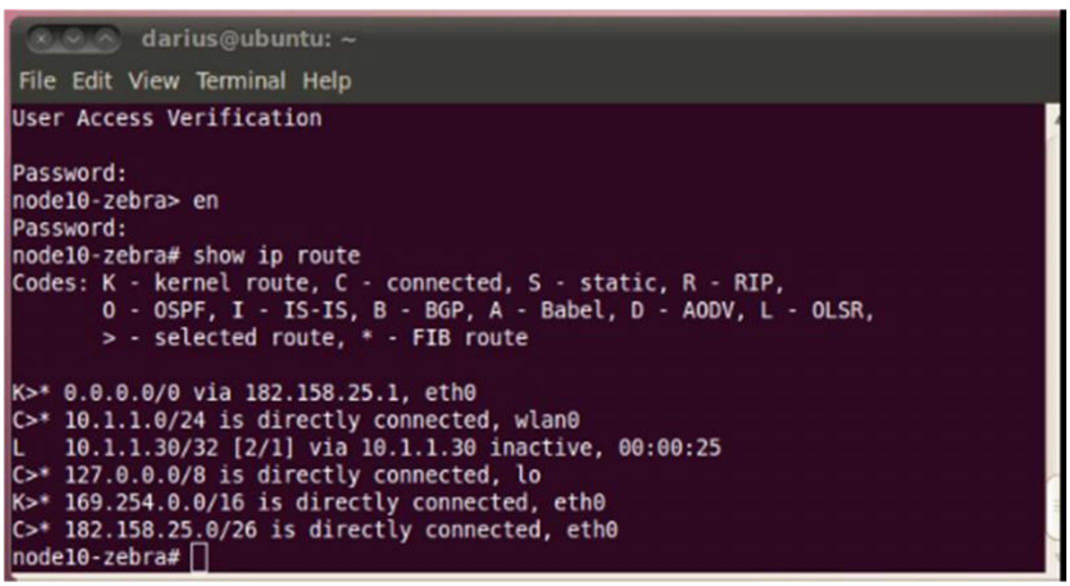
The debug output for OLSRd shows a new route to C.

Try connect to OLSRd and Zebra VTYSH interface to view routing tables

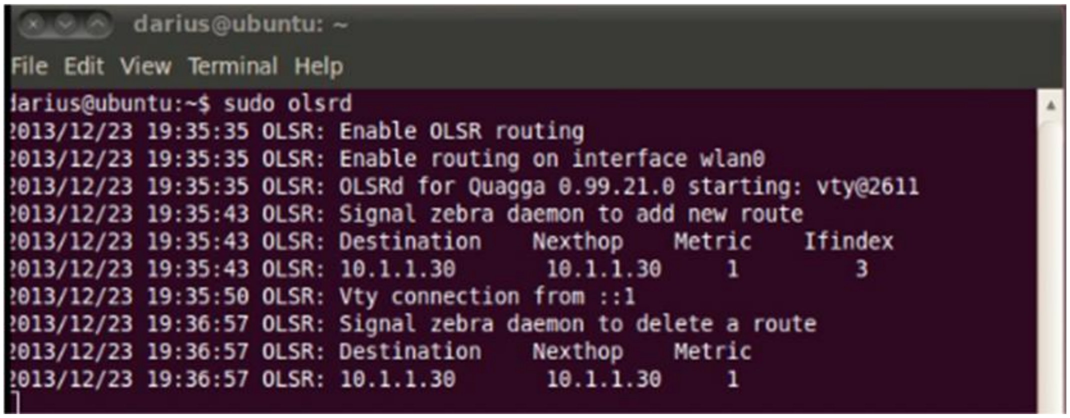
From OLSRd VTYSH session, A has C as a neighbor and a route to C.



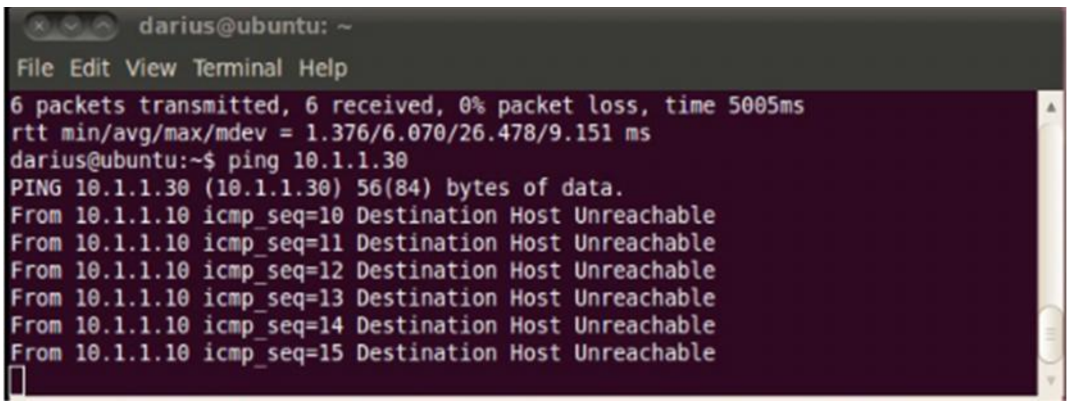
From Zebra VTYSH session, A has a route to C (not a FIB route because A and C are within each other coverage).



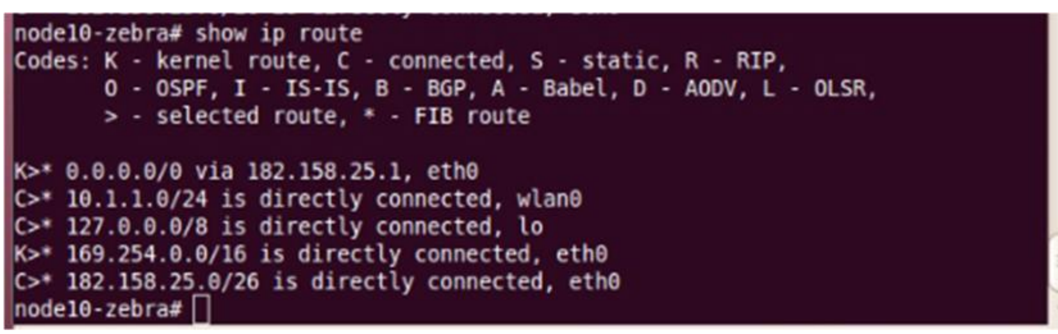
When move C away from A, the OLSRd debug shows a route deleted.



And the ping fails



OLSRd and Zebra daemon route entry to C also deleted



Now, we put node B between node A and C. Run OLSRd and Zebra daemon on node B. From node B's OLSRd debug shows new routes to node A and C. Node A, C select B as MPR (MPRS value of "yes").

```
node20-olsrd> enable
node20-olsrd# show ip olsr nei
MAIN ADDR               STATUS          WILLINGNGESS    MPR     MPRS
10.1.1.30               sym             default         Nope    Yes
10.1.1.10               sym             default         Nope    Yes
node20-olsrd# show ip olsr route
Dest addr       Next hop        Dist    Iface
10.1.1.10       10.1.1.10       1       10.1.1.20
10.1.1.30       10.1.1.30       1       10.1.1.20
node20-olsrd#
```

On node A, the debug shows new routes to node B, and to node C via B.

```
2013/12/23 19:37:58 OLSR: Signal zebra daemon to add new route
2013/12/23 19:37:58 OLSR: Destination    Nexthop    Metric    Ifindex
2013/12/23 19:37:58 OLSR: 10.1.1.20       10.1.1.20     1         3
2013/12/23 19:38:00 OLSR: Signal zebra daemon to add new route
2013/12/23 19:38:00 OLSR: Destination    Nexthop    Metric    Ifindex
2013/12/23 19:38:00 OLSR: 10.1.1.30       10.1.1.30     2         3
```

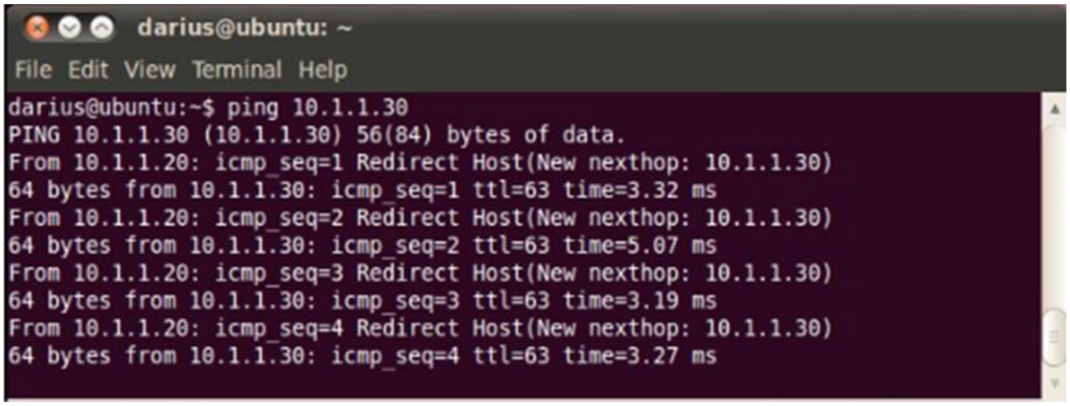
Try look at node A's OLSRd VTYSH session, it has new neighbor B and two routes to B and C.

```
node10-olsrd# show ip olsr nei
MAIN ADDR               STATUS          WILLINGNGESS    MPR     MPRS
10.1.1.20               sym             default         Yes     Nope
node10-olsrd# show ip olsr route
Dest addr       Next hop        Dist    Iface
10.1.1.20       10.1.1.20       1       10.1.1.10
10.1.1.30       10.1.1.20       2       10.1.1.10
node10-olsrd#
```

Node A Zebra's routing table has two entries to node B and to node C via B. The route to C is a FIB and will be synchronized to kernel routing table.

```
node10-zebra# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, A - Babel, D - AODV, L - OLSR,
       > - selected route, * - FIB route

K>* 0.0.0.0/0 via 182.158.25.1, eth0
C>* 10.1.1.0/24 is directly connected, wlan0
L   10.1.1.20/32 [2/1] via 10.1.1.20 inactive, 00:01:24
L>* 10.1.1.30/32 [2/2] via 10.1.1.20, wlan0, 00:01:22
C>* 127.0.0.0/8 is directly connected, lo
K>* 169.254.0.0/16 is directly connected, eth0
C>* 182.158.25.0/26 is directly connected, eth0
node10-zebra#
```

A pings C successfully.

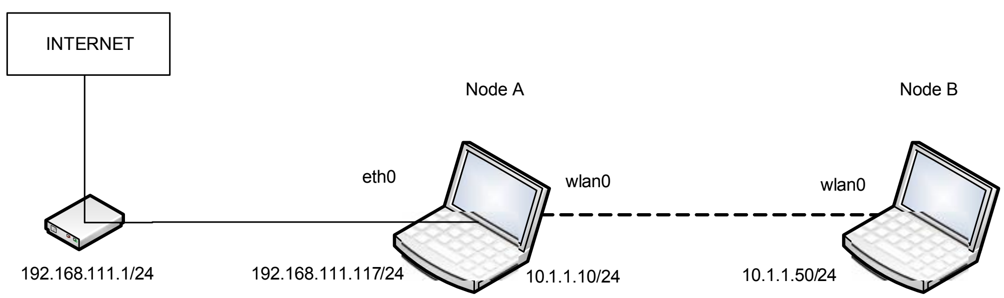


4.4. Internet access test case

4.4.1. Purpose

In this test case, we try share internet access for other MANET node with Zebra routing.

4.4.2. Scenario



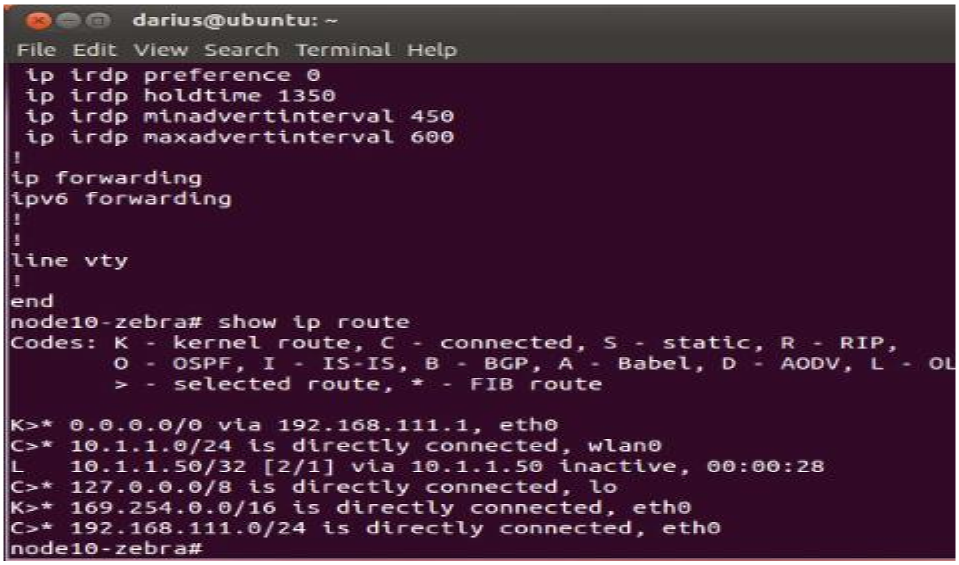
In this scenario, we have one laptop (node A) with two interfaces – wlan0 in an MANET and eth0 connects to a wired network. The IP address configuration is as above diagram. Node B with a wireless wlan0 interface also a member of MANET. Node A has internet access via interface eth0. We will configure node A for sharing internet to node B.

4.4.3. Test and result

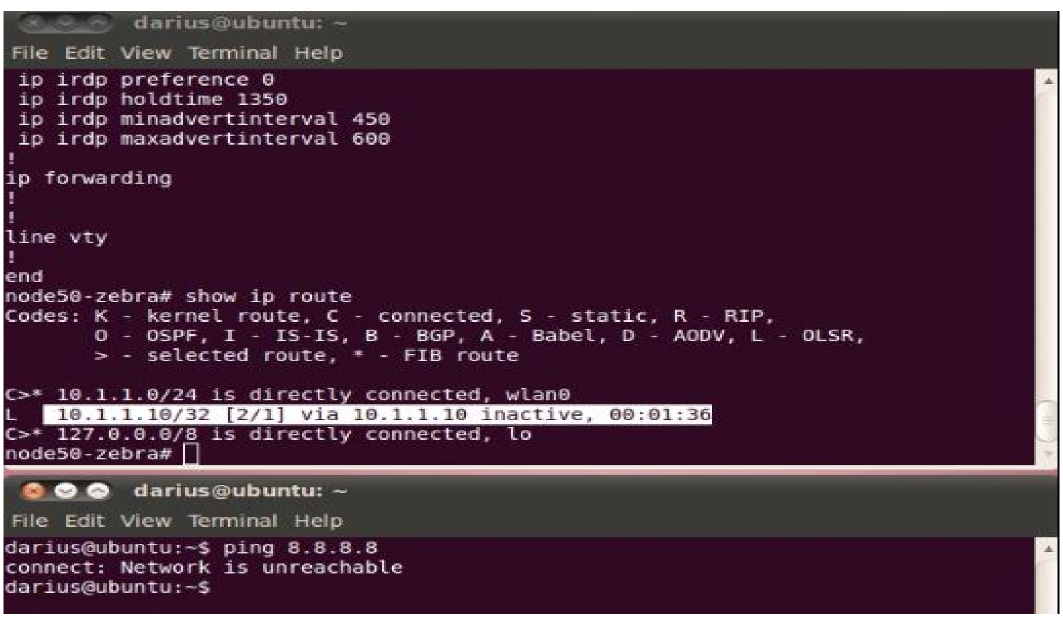
Run Zebra and OLSRd on both two nodes and configure IP address from the diagram.

On node A, “show ip route” command lists all available route entries. Note that: K>* 0.0.0.0/0 via 192.168.111.1, eth0 is a kernel route and it is also the default route for internet access.

L 10.1.1.50/32 [2/1] via 10.1.1.50 is the OLSR route entry to node B



On node B, “show ip route” lists the routing table entries. Node B has a route to node A, however, it still cannot access the internet (the ping 8.8.8.8 failed).



Now, we need to configure for sharing internet access on both node A and B.

On node A, we need to configure NAT translation using following command

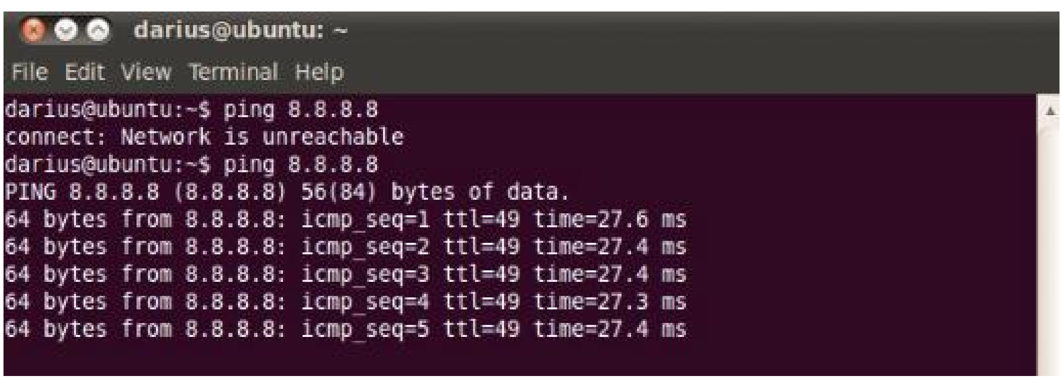
sudo iptables -A FORWARD -o eth0 -i wlan0 -s 10.1.1.0/24 -m conntrack --ctstate NEW -j ACCEPT
#sudo iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
#sudo iptables -t nat -F POSTROUTING
#sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

The first rule allow forwarded packets (initial ones), the second rule allows forwarding of established connection packets and the third rule does the NAT.

On node B, configure default route to forward packet to node A from Zebra.


```
node50-zebra# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, A - Babel, D - AODV, L - OLSR,
       > - selected route, * - FIB route

C>* 10.1.1.0/24 is directly connected, wlan0
L   10.1.1.10/32 [2/1] via 10.1.1.10 inactive, 00:01:36
C>* 127.0.0.0/8 is directly connected, lo
node50-zebra# conf t
node50-zebra(config)# ip route 0.0.0.0/0 10.1.1.10
node50-zebra(config)#
```

Now , try ping 8.8.8.8 on node B again.


```
darius@ubuntu: ~
File  Edit  View  Terminal  Help
darius@ubuntu:~$ ping 8.8.8.8
connect: Network is unreachable
darius@ubuntu:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=49 time=27.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=49 time=27.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=49 time=27.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=49 time=27.3 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=49 time=27.4 ms
```

Because there is no DNS configuration on node B so at this stage we can only locate an internet location with IP address.

Chapter 5. CONCLUSION AND FUTURE WORK

In this thesis, I put a goal to integrate the release of AODV-UU and OLSRdq to work with Zebra daemon (from Quagga 0.99.22.1). This lead to some sub-goals to understand the operation of AODVd, OLSRd and Zebra daemon, the Zebra protocol in which these daemons utilized to create the communication channel.

Further research on existing source code provides more information of different types of messages that routing daemons exchange with Zebra daemon – route add/delete messages, interface status query messages, interface IP address query messages, etc…

Follow the approach, additional code are prepared and update to AODV-UU 0.9.6 and OLSRqd 0.98.5 release to make these two routing daemon able to communication with Zebra daemon for kernel routing table update.

Real testcase with physical machines are also prepared and carried out to test the daemon operation – routing, message exchanging between daemons, kernel routing table updating result. The testcase also come with some movement action to mimic nodes mobility. Log files, screen dumps, screen video records are collected for troubleshooting the source code functionalities.

Due to time constraint and the author limited in programming skill, the release only focus on kernel route update (add and delete actions) for AODVd and core functionalities for OLSRd specified in RFC3626. Further features to be done in future work for this adhoc-iu package

- An vtysh interface for managing AODVd.
- With this release, AODVd does not support all Zebra message types but only add route, update route and delete route messages. The AODVd still not react to interface status changing during its operation which is a drawback to be taken care of in later work.
- Addional functionalities for OLSRd as stated in RFC3626.
- New IPv6 address support is also an important module that the author should target. The security and memory leak are missing in this release and yet to be fulfilled in future work. As for testing purposes, proposed test cases are simple and not cover many aspects in ad-hoc environment – number of nodes, hidden node problem, signal strength, internet gateway,

testing, redistribution between multiple routing protocols are some other approaches to focus on.

LIST OF REFERENCES

[01] C. Perkins, E. Belding-Royer, S. Das, “Ad-hoc On-Demand Distance Vector (AODV) Routing”, IEFT Network Working Group RFC 3561, July 2003.

[02] Bjorn Wiberg, Porting AODV-UU Implementation to ns-2 and Enabling Trace-based Simulation, Master‟s Thesis at Uppsala University, Sweden, 2002.

[03] Clifton Lin, AODV Routing Implementation for Scalable Wireless Ad-hoc Network Simulation, Cornell University, American.

[04] T. Clausen, P. Jacquet, “Optimized Link State Routing Protocol (OLSR)”, IEFT Network Working Group RFC 3626, October 2003.

[05] Optimized Link State Routing Protocol for Ad Hoc Networks, P. Jackquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, L. Viennot*, Hipercom Project, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.

[6] Wireless Extensions to OSPF: Implementation of the Overlapping Relays Proposal, Kenneth Holter, University of Oslo.

[7] OLSRdq - OLSR daemon for Quagga 0.98.5, http://olsrdq.sourceforge.net/ , Tudor Golubenco.

[8] Kunihiro Ishiguro, et al , “Quagga - a routing software package for TCP/IP networks”, Jan 2013,.

[9] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler, "The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel", Prentice Hall, August 2004

Appendix

Zebra daemon has its own routing information base (RIB). While working, Zebra daemon receives route updates from different routing daemons, the module rib_process (defined in zebra_rib.c) is called to update Zebra RIB. There is a comparison between different types of routes based on the administrative value. The result is an FIB database which contains “win” routes is feeded to kernel routing table.

```
/* Core function for processing routing information base. */
static void
rib_process (struct route_node *rn)
{
  struct rib *rib;
  struct rib *next;
  struct rib *fib = NULL;
  struct rib *select = NULL;
  struct rib *del = NULL;
  int installed = 0;
  struct nexthop *nexthop = NULL;
  char buf[INET6_ADDRSTRLEN];

  assert (rn);

  if (IS_ZEBRA_DEBUG_RIB || IS_ZEBRA_DEBUG_RIB_Q)
    inet_ntop (rn->p.family, &rn->p.u.prefix, buf, INET6_ADDRSTRLEN);

  RNODE_FOREACH_RIB_SAFE (rn, rib, next)
    {
      /* Currently installed rib. */
      if (CHECK_FLAG (rib->flags, ZEBRA_FLAG_SELECTED))
        {
          assert (fib == NULL);
          fib = rib;
        }

      /* Unlock removed routes, so they'll be freed, bar the FIB entry,
       * which we need to do do further work with below.
       */
      if (CHECK_FLAG (rib->status, RIB_ENTRY_REMOVED))
        {
          if (rib != fib)
```

```
          {
            if (IS_ZEBRA_DEBUG_RIB)
              zlog_debug ("%s: %s/%d: rn %p, removing rib %p", __func__,
                buf, rn->p.prefixlen, rn, rib);
                rib_unlink (rn, rib);
          }
        else
          del = rib;

        continue;
      }

    /* Skip unreachable nexthop. */
    if (! nexthop_active_update (rn, rib, 0))
      continue;

    /* Infinit distance. */
    if (rib->distance == DISTANCE_INFINITY)
      continue;

    /* Newly selected rib, the common case. */
    if (!select)
      {
        select = rib;
        continue;
      }

    /* filter route selection in following order:
     * - connected beats other types
     * - lower distance beats higher
     * - lower metric beats higher for equal distance
     * - last, hence oldest, route wins tie break.
     */

    /* Connected routes. Pick the last connected
     * route of the set of lowest metric connected routes.
     */
    if (rib->type == ZEBRA_ROUTE_CONNECT)
      {
        if (select->type != ZEBRA_ROUTE_CONNECT
            || rib->metric <= select->metric)
          select = rib;
        continue;
```

```
      }
    else if (select->type == ZEBRA_ROUTE_CONNECT)
      continue;

    /* higher distance loses */
    if (rib->distance > select->distance)
      continue;

    /* lower wins */
    if (rib->distance < select->distance)
      {
        select = rib;
        continue;
      }

    /* metric tie-breaks equal distance */
    if (rib->metric <= select->metric)
      select = rib;
  } /* RNODE_FOREACH_RIB_SAFE */

/* After the cycle is finished, the following pointers will be set:
 * select --- the winner RIB entry, if any was found, otherwise NULL
 * fib    --- the SELECTED RIB entry, if any, otherwise NULL
 * del    --- equal to fib, if fib is queued for deletion, NULL otherwise
 * rib    --- NULL
 */

/* Same RIB entry is selected. Update FIB and finish. */
if (select && select == fib)
  {
    if (IS_ZEBRA_DEBUG_RIB)
      zlog_debug ("%s: %s/%d: Updating existing route, select %p, fib %p",
                  __func__, buf, rn->p.prefixlen, select, fib);
    if (CHECK_FLAG (select->flags, ZEBRA_FLAG_CHANGED))
      {
          zfpm_trigger_update (rn, "updating existing route");

        redistribute_delete (&rn->p, select);
        if (! RIB_SYSTEM_ROUTE (select))
          rib_uninstall_kernel (rn, select);

        /* Set real nexthop. */
        nexthop_active_update (rn, select, 1);
```

```
        if (! RIB_SYSTEM_ROUTE (select))
          rib_install_kernel (rn, select);
        redistribute_add (&rn->p, select);
      }
    else if (! RIB_SYSTEM_ROUTE (select))
      {
        /* Housekeeping code to deal with
           race conditions in kernel with linux
           netlink reporting interface up before IPv4 or IPv6 protocol
           is ready to add routes.
           This makes sure the routes are IN the kernel.
         */

        for (nexthop = select->nexthop; nexthop; nexthop = nexthop->next)
          if (CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB))
            {
              installed = 1;
              break;
            }
        if (! installed)
          rib_install_kernel (rn, select);
      }
    goto end;
  }

/* At this point we either haven't found the best RIB entry or it is
 * different from what we currently intend to flag with SELECTED. In both
 * cases, if a RIB block is present in FIB, it should be withdrawn.
 */
if (fib)
  {
    if (IS_ZEBRA_DEBUG_RIB)
      zlog_debug ("%s: %s/%d: Removing existing route, fib %p", __func__,
        buf, rn->p.prefixlen, fib);

    zfpm_trigger_update (rn, "removing existing route");

    redistribute_delete (&rn->p, fib);
    if (! RIB_SYSTEM_ROUTE (fib))
          rib_uninstall_kernel (rn, fib);
    UNSET_FLAG (fib->flags, ZEBRA_FLAG_SELECTED);
```

```
    /* Set real nexthop. */
    nexthop_active_update (rn, fib, 1);
  }

/* Regardless of some RIB entry being SELECTED or not before, now we can
 * tell, that if a new winner exists, FIB is still not updated with this
 * data, but ready to be.
 */
if (select)
  {
    if (IS_ZEBRA_DEBUG_RIB)
      zlog_debug ("%s: %s/%d: Adding route, select %p", __func__, buf,
        rn->p.prefixlen, select);

    zfpm_trigger_update (rn, "new route selected");

    /* Set real nexthop. */
    nexthop_active_update (rn, select, 1);

    if (! RIB_SYSTEM_ROUTE (select))
      rib_install_kernel (rn, select);
    SET_FLAG (select->flags, ZEBRA_FLAG_SELECTED);
    redistribute_add (&rn->p, select);
  }

/* FIB route was removed, should be deleted */
if (del)
  {
    if (IS_ZEBRA_DEBUG_RIB)
      zlog_debug ("%s: %s/%d: Deleting fib %p, rn %p", __func__, buf,
        rn->p.prefixlen, del, rn);
    rib_unlink (rn, del);
  }

end:
 if (IS_ZEBRA_DEBUG_RIB_Q)
   zlog_debug ("%s: %s/%d: rn %p dequeued", __func__, buf, rn->p.prefixlen,
rn);

 /*
  * Check if the dest can be deleted now.
  */
 rib_gc_dest (rn);
```

```
}
```

When a success RIB (or FIB) updated to kernel, functions defined in zebra/rt.h will be called to perform correspondent actions (add, delete).

```
extern int kernel_add_ipv4 (struct prefix *, struct rib *);
extern int kernel_delete_ipv4 (struct prefix *, struct rib *);

#ifdef HAVE_IPV6
extern int kernel_add_ipv6 (struct prefix *, struct rib *);
extern int kernel_delete_ipv6 (struct prefix *, struct rib *);
#endif /* HAVE_IPV6 */
```

Functions kernel_add_ipv4 handles the adding of new Ipv4 route to kernel routing table while kernel_delete_ipv4 performs deleting an Ipv4 route from kernel. Zebra supports multiple platform thus, the mentioned functions are implemented in multiple approaches – using ioctl , netlink socket or routing socket.

a. For ioctl , the implementation in zebra\rt_ioctl.c

```
/* Interface to ioctl route message. */
static int
kernel_ioctl_ipv4 (u_long cmd, struct prefix *p, struct rib *rib, int family)
{
  int ret;
  int sock;
  struct rtentry rtentry;
  struct sockaddr_in sin_dest, sin_mask, sin_gate;
  struct nexthop *nexthop;
  int nexthop_num = 0;
  struct interface *ifp;

  memset (&rtentry, 0, sizeof (struct rtentry));

  /* Make destination. */
  memset (&sin_dest, 0, sizeof (struct sockaddr_in));
  sin_dest.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
  sin_dest.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
  sin_dest.sin_addr = p->u.prefix4;

  if (CHECK_FLAG (rib->flags, ZEBRA_FLAG_BLACKHOLE))
    {
    SET_FLAG (rtentry.rt_flags, RTF_REJECT);
```

```
    if (cmd == SIOCADDRT)
        for (nexthop = rib->nexthop; nexthop; nexthop = nexthop->next)
          SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);

    goto skip;
  }

  memset (&sin_gate, 0, sizeof (struct sockaddr_in));

  /* Make gateway. */
  for (nexthop = rib->nexthop; nexthop; nexthop = nexthop->next)
    {
    if ((cmd == SIOCADDRT
           && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_ACTIVE))
          || (cmd == SIOCDELRT
             && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB)))
        {
          if (CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_RECURSIVE))
            {
             if (nexthop->rtype == NEXTHOP_TYPE_IPV4 ||
                    nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX)
                   {
                    sin_gate.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
                    sin_gate.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
                    sin_gate.sin_addr = nexthop->rgate.ipv4;
                    rtentry.rt_flags |= RTF_GATEWAY;
                   }
             if (nexthop->rtype == NEXTHOP_TYPE_IFINDEX
                    || nexthop->rtype == NEXTHOP_TYPE_IFNAME)
                   {
                    ifp = if_lookup_by_index (nexthop->rifindex);
                    if (ifp)
                      rtentry.rt_dev = ifp->name;
                    else
                      return -1;
                   }
            }
          else
            {
             if (nexthop->type == NEXTHOP_TYPE_IPV4 ||
```

```
                        nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
                       {
                        sin_gate.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
                        sin_gate.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
                        sin_gate.sin_addr = nexthop->gate.ipv4;
                        rtentry.rt_flags |= RTF_GATEWAY;
                       }
                  if (nexthop->type == NEXTHOP_TYPE_IFINDEX
                        || nexthop->type == NEXTHOP_TYPE_IFNAME)
                       {
                        ifp = if_lookup_by_index (nexthop->ifindex);
                        if (ifp)
                          rtentry.rt_dev = ifp->name;
                        else
                          return -1;
                       }
                }

              if (cmd == SIOCADDRT)
                SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);

              nexthop_num++;
              break;
            }
    }

  /* If there is no useful nexthop then return. */
  if (nexthop_num == 0)
    {
      if (IS_ZEBRA_DEBUG_KERNEL)
            zlog_debug ("netlink_route_multipath(): No useful nexthop.");
      return 0;
    }

 skip:

  memset (&sin_mask, 0, sizeof (struct sockaddr_in));
  sin_mask.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
  sin_mask.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
```

```
  masklen2ip (p->prefixlen, &sin_mask.sin_addr);

  /* Set destination address, mask and gateway.*/
  memcpy (&rtentry.rt_dst, &sin_dest, sizeof (struct sockaddr_in));

  if (rtentry.rt_flags & RTF_GATEWAY)
    memcpy (&rtentry.rt_gateway, &sin_gate, sizeof (struct sockaddr_in));

#ifndef SUNOS_5
  memcpy (&rtentry.rt_genmask, &sin_mask, sizeof (struct sockaddr_in));
#endif /* SUNOS_5 */

  /* Metric.  It seems metric minus one value is installed... */
  rtentry.rt_metric = rib->metric;

  /* Routing entry flag set. */
  if (p->prefixlen == 32)
    rtentry.rt_flags |= RTF_HOST;

  rtentry.rt_flags |= RTF_UP;

  /* Additional flags */
  /* rtentry.rt_flags |= flags; */

  /* For tagging route. */
  /* rtentry.rt_flags |= RTF_DYNAMIC; */

  /* Open socket for ioctl. */
  sock = socket (AF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
    {
      zlog_warn ("can't make socket\n");
      return -1;
    }

  /* Send message by ioctl(). */
  ret = ioctl (sock, cmd, &rtentry);
  if (ret < 0)
    {
      switch (errno)
        {
        case EEXIST:
          close (sock);
```

```
          return ZEBRA_ERR_RTEXIST;
          break;
        case ENETUNREACH:
          close (sock);
          return ZEBRA_ERR_RTUNREACH;
          break;
        case EPERM:
          close (sock);
          return ZEBRA_ERR_EPERM;
          break;
        }

    close (sock);
    zlog_warn ("write : %s (%d)", safe_strerror (errno), errno);
    return ret;
  }
  close (sock);

  return ret;
}
```

b. For netlink socket , the implementation in zebra\rt_netlink.c

```
/* Routing table change via netlink interface. */
static int
netlink_route_multipath (int cmd, struct prefix *p, struct rib *rib,
                         int family)
{
  int bytelen;
  struct sockaddr_nl snl;
  struct nexthop *nexthop = NULL;
  int nexthop_num = 0;
  int discard;

  struct
  {
    struct nlmsghdr n;
    struct rtmsg r;
    char buf[NL_PKT_BUF_SIZE];
  } req;

  memset (&req, 0, sizeof req - NL_PKT_BUF_SIZE);

  bytelen = (family == AF_INET ? 4 : 16);
```

```
 req.n.nlmsg_len = NLMSG_LENGTH (sizeof (struct rtmsg));
 req.n.nlmsg_flags = NLM_F_CREATE | NLM_F_REQUEST;
 req.n.nlmsg_type = cmd;
 req.r.rtm_family = family;
 req.r.rtm_table = rib->table;
 req.r.rtm_dst_len = p->prefixlen;
 req.r.rtm_protocol = RTPROT_ZEBRA;
 req.r.rtm_scope = RT_SCOPE_UNIVERSE;

 if ((rib->flags & ZEBRA_FLAG_BLACKHOLE) || (rib->flags & ZEBRA_FLAG_REJECT))
   discard = 1;
 else
   discard = 0;

 if (cmd == RTM_NEWROUTE)
   {
     if (discard)
       {
         if (rib->flags & ZEBRA_FLAG_BLACKHOLE)
           req.r.rtm_type = RTN_BLACKHOLE;
         else if (rib->flags & ZEBRA_FLAG_REJECT)
           req.r.rtm_type = RTN_UNREACHABLE;
         else
           assert (RTN_BLACKHOLE != RTN_UNREACHABLE);  /* false */
       }
     else
       req.r.rtm_type = RTN_UNICAST;
   }

 addattr_l (&req.n, sizeof req, RTA_DST, &p->u.prefix, bytelen);

 /* Metric. */
 addattr32 (&req.n, sizeof req, RTA_PRIORITY, rib->metric);

 if (discard)
   {
     if (cmd == RTM_NEWROUTE)
       for (nexthop = rib->nexthop; nexthop; nexthop = nexthop->next)
         SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);
     goto skip;
   }
```

```
 /* Multipath case. */
 if (rib->nexthop_active_num == 1 || MULTIPATH_NUM == 1)
   {
     for (nexthop = rib->nexthop; nexthop; nexthop = nexthop->next)
       {

        if ((cmd == RTM_NEWROUTE
            && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_ACTIVE))
           || (cmd == RTM_DELROUTE
               && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB)))
          {

            if (CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_RECURSIVE))
              {
                if (IS_ZEBRA_DEBUG_KERNEL)
                  {
                    zlog_debug
                      ("netlink_route_multipath() (recursive, 1 hop): "
                       "%s %s/%d, type %s", lookup (nlmsg_str, cmd),
#ifdef HAVE_IPV6
                                   (family == AF_INET) ? inet_ntoa (p->u.prefix4) :
                                   inet6_ntoa (p->u.prefix6),
#else
                                   inet_ntoa (p->u.prefix4),
#endif /* HAVE_IPV6 */

                                   p->prefixlen, nexthop_type_to_str (nexthop->rtype));
                  }

                if (nexthop->rtype == NEXTHOP_TYPE_IPV4
                    || nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX)
                       {
                         addattr_l (&req.n, sizeof req, RTA_GATEWAY,
                                    &nexthop->rgate.ipv4, bytelen);
                    if (nexthop->src.ipv4.s_addr)
                           addattr_l(&req.n, sizeof req, RTA_PREFSRC,
                                     &nexthop->src.ipv4, bytelen);
                         if (IS_ZEBRA_DEBUG_KERNEL)
                             zlog_debug("netlink_route_multipath() (recursive, "
                                        "1 hop): nexthop via %s if %u",
                                        inet_ntoa (nexthop->rgate.ipv4),
                                        nexthop->rifindex);
                       }
```

```
#ifdef HAVE_IPV6
          if (nexthop->rtype == NEXTHOP_TYPE_IPV6
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFNAME)
              {
               addattr_l (&req.n, sizeof req, RTA_GATEWAY,
                          &nexthop->rgate.ipv6, bytelen);

               if (IS_ZEBRA_DEBUG_KERNEL)
                   zlog_debug("netlink_route_multipath() (recursive, "
                              "1 hop): nexthop via %s if %u",
                              inet6_ntoa (nexthop->rgate.ipv6),
                              nexthop->rifindex);
              }
#endif /* HAVE_IPV6 */
          if (nexthop->rtype == NEXTHOP_TYPE_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IFNAME
            || nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFNAME)
              {
               addattr32 (&req.n, sizeof req, RTA_OIF,
                          nexthop->rifindex);
            if ((nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX
                || nexthop->rtype == NEXTHOP_TYPE_IFINDEX)
               && nexthop->src.ipv4.s_addr)
             addattr_l (&req.n, sizeof req, RTA_PREFSRC,
                          &nexthop->src.ipv4, bytelen);

               if (IS_ZEBRA_DEBUG_KERNEL)
                   zlog_debug("netlink_route_multipath() (recursive, "
                              "1 hop): nexthop via if %u",
                              nexthop->rifindex);
              }
         }
       else
         {
          if (IS_ZEBRA_DEBUG_KERNEL)
            {
             zlog_debug
              ("netlink_route_multipath() (single hop): "
               "%s %s/%d, type %s", lookup (nlmsg_str, cmd),
#ifdef HAVE_IPV6
```

```
                              (family == AF_INET) ? inet_ntoa (p->u.prefix4) :
                              inet6_ntoa (p->u.prefix6),
#else
                              inet_ntoa (p->u.prefix4),
#endif /* HAVE_IPV6 */
                              p->prefixlen, nexthop_type_to_str (nexthop->type));
                }

               if (nexthop->type == NEXTHOP_TYPE_IPV4
                   || nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
                        {
                         addattr_l (&req.n, sizeof req, RTA_GATEWAY,
                                        &nexthop->gate.ipv4, bytelen);
                         if (nexthop->src.ipv4.s_addr)
                    addattr_l (&req.n, sizeof req, RTA_PREFSRC,
                                        &nexthop->src.ipv4, bytelen);

                         if (IS_ZEBRA_DEBUG_KERNEL)
                              zlog_debug("netlink_route_multipath() (single hop): "
                                         "nexthop via %s if %u",
                                         inet_ntoa (nexthop->gate.ipv4),
                                         nexthop->ifindex);
                        }
#ifdef HAVE_IPV6
               if (nexthop->type == NEXTHOP_TYPE_IPV6
                   || nexthop->type == NEXTHOP_TYPE_IPV6_IFNAME
                   || nexthop->type == NEXTHOP_TYPE_IPV6_IFINDEX)
                        {
                         addattr_l (&req.n, sizeof req, RTA_GATEWAY,
                                        &nexthop->gate.ipv6, bytelen);

                         if (IS_ZEBRA_DEBUG_KERNEL)
                              zlog_debug("netlink_route_multipath() (single hop): "
                                         "nexthop via %s if %u",
                                         inet6_ntoa (nexthop->gate.ipv6),
                                         nexthop->ifindex);
                        }
#endif /* HAVE_IPV6 */
               if (nexthop->type == NEXTHOP_TYPE_IFINDEX
                   || nexthop->type == NEXTHOP_TYPE_IFNAME
                   || nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
                        {
                         addattr32 (&req.n, sizeof req, RTA_OIF, nexthop->ifindex);
```

```
                              if (nexthop->src.ipv4.s_addr)
                        addattr_l (&req.n, sizeof req, RTA_PREFSRC,
                                                 &nexthop->src.ipv4, bytelen);

                              if (IS_ZEBRA_DEBUG_KERNEL)
                                    zlog_debug("netlink_route_multipath() (single hop): "
                                                   "nexthop via if %u", nexthop->ifindex);
                            }
                  else if (nexthop->type == NEXTHOP_TYPE_IPV6_IFINDEX
                      || nexthop->type == NEXTHOP_TYPE_IPV6_IFNAME)
                            {
                              addattr32 (&req.n, sizeof req, RTA_OIF, nexthop->ifindex);

                              if (IS_ZEBRA_DEBUG_KERNEL)
                                    zlog_debug("netlink_route_multipath() (single hop): "
                                                   "nexthop via if %u", nexthop->ifindex);
                            }
                  }

              if (cmd == RTM_NEWROUTE)
                SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);

              nexthop_num++;
              break;
            }
        }
    }
  else
    {
      char buf[NL_PKT_BUF_SIZE];
      struct rtattr *rta = (void *) buf;
      struct rtnexthop *rtnh;
      union g_addr *src = NULL;

      rta->rta_type = RTA_MULTIPATH;
      rta->rta_len = RTA_LENGTH (0);
      rtnh = RTA_DATA (rta);

      nexthop_num = 0;
      for (nexthop = rib->nexthop;
           nexthop && (MULTIPATH_NUM == 0 || nexthop_num <
MULTIPATH_NUM);
```

```
          nexthop = nexthop->next)
        {
         if ((cmd == RTM_NEWROUTE
              && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_ACTIVE))
             || (cmd == RTM_DELROUTE
                 && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB)))
           {
             nexthop_num++;

             rtnh->rtnh_len = sizeof (*rtnh);
             rtnh->rtnh_flags = 0;
             rtnh->rtnh_hops = 0;
             rta->rta_len += rtnh->rtnh_len;

             if (CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_RECURSIVE))
               {
                 if (IS_ZEBRA_DEBUG_KERNEL)
                   {
                     zlog_debug ("netlink_route_multipath() "
                        "(recursive, multihop): %s %s/%d type %s",
                                   lookup (nlmsg_str, cmd),
#ifdef HAVE_IPV6
                                   (family == AF_INET) ? inet_ntoa (p->u.prefix4) :
                                   inet6_ntoa (p->u.prefix6),
#else
                                   inet_ntoa (p->u.prefix4),
#endif /* HAVE_IPV6 */
                                   p->prefixlen, nexthop_type_to_str (nexthop->rtype));
                   }
                 if (nexthop->rtype == NEXTHOP_TYPE_IPV4
                     || nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX)
                   {
                     rta_addattr_l (rta, NL_PKT_BUF_SIZE, RTA_GATEWAY,
                                   &nexthop->rgate.ipv4, bytelen);
                     rtnh->rtnh_len += sizeof (struct rtattr) + 4;

                           if (nexthop->src.ipv4.s_addr)
                       src = &nexthop->src;

                           if (IS_ZEBRA_DEBUG_KERNEL)
                                 zlog_debug("netlink_route_multipath() (recursive, "
                                            "multihop): nexthop via %s if %u",
                                            inet_ntoa (nexthop->rgate.ipv4),
```

```
                              nexthop->rifindex);
            }
#ifdef HAVE_IPV6
          if (nexthop->rtype == NEXTHOP_TYPE_IPV6
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFNAME
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFINDEX)
              {
               rta_addattr_l (rta, NL_PKT_BUF_SIZE, RTA_GATEWAY,
                              &nexthop->rgate.ipv6, bytelen);

               if (IS_ZEBRA_DEBUG_KERNEL)
                   zlog_debug("netlink_route_multipath() (recursive, "
                              "multihop): nexthop via %s if %u",
                              inet6_ntoa (nexthop->rgate.ipv6),
                              nexthop->rifindex);
              }
#endif /* HAVE_IPV6 */
          /* ifindex */
          if (nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX
               || nexthop->rtype == NEXTHOP_TYPE_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IFNAME)
              {
               rtnh->rtnh_ifindex = nexthop->rifindex;
            if (nexthop->src.ipv4.s_addr)
             src = &nexthop->src;

               if (IS_ZEBRA_DEBUG_KERNEL)
                   zlog_debug("netlink_route_multipath() (recursive, "
                              "multihop): nexthop via if %u",
                              nexthop->rifindex);
              }
              else if (nexthop->rtype == NEXTHOP_TYPE_IPV6_IFINDEX
            || nexthop->rtype == NEXTHOP_TYPE_IPV6_IFNAME)
              {
               rtnh->rtnh_ifindex = nexthop->rifindex;

               if (IS_ZEBRA_DEBUG_KERNEL)
                   zlog_debug("netlink_route_multipath() (recursive, "
                              "multihop): nexthop via if %u",
                              nexthop->rifindex);
              }
          else
              {
```

```
                        rtnh->rtnh_ifindex = 0;
                      }
            }
          else
            {
              if (IS_ZEBRA_DEBUG_KERNEL)
                {
                  zlog_debug ("netlink_route_multipath() (multihop): "
                    "%s %s/%d, type %s", lookup (nlmsg_str, cmd),
#ifdef HAVE_IPV6
                              (family == AF_INET) ? inet_ntoa (p->u.prefix4) :
                              inet6_ntoa (p->u.prefix6),
#else
                              inet_ntoa (p->u.prefix4),
#endif /* HAVE_IPV6 */
                              p->prefixlen, nexthop_type_to_str (nexthop->type));
                }
              if (nexthop->type == NEXTHOP_TYPE_IPV4
                  || nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
                {
                        rta_addattr_l (rta, NL_PKT_BUF_SIZE, RTA_GATEWAY,
                                       &nexthop->gate.ipv4, bytelen);
                        rtnh->rtnh_len += sizeof (struct rtattr) + 4;

                        if (nexthop->src.ipv4.s_addr)
                    src = &nexthop->src;

                  if (IS_ZEBRA_DEBUG_KERNEL)
                            zlog_debug("netlink_route_multipath() (multihop): "
                                       "nexthop via %s if %u",
                                       inet_ntoa (nexthop->gate.ipv4),
                                       nexthop->ifindex);
                }
#ifdef HAVE_IPV6
              if (nexthop->type == NEXTHOP_TYPE_IPV6
                  || nexthop->type == NEXTHOP_TYPE_IPV6_IFNAME
                  || nexthop->type == NEXTHOP_TYPE_IPV6_IFINDEX)
                      {
                        rta_addattr_l (rta, NL_PKT_BUF_SIZE, RTA_GATEWAY,
                                       &nexthop->gate.ipv6, bytelen);

                        if (IS_ZEBRA_DEBUG_KERNEL)
                            zlog_debug("netlink_route_multipath() (multihop): "
```

```
                                        "nexthop via %s if %u",
                                        inet6_ntoa (nexthop->gate.ipv6),
                                        nexthop->ifindex);
                            }
#endif /* HAVE_IPV6 */
                /* ifindex */
                if (nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX
                          || nexthop->type == NEXTHOP_TYPE_IFINDEX
                   || nexthop->type == NEXTHOP_TYPE_IFNAME)
                  {
                            rtnh->rtnh_ifindex = nexthop->ifindex;
                            if (nexthop->src.ipv4.s_addr)
                                 src = &nexthop->src;
                            if (IS_ZEBRA_DEBUG_KERNEL)
                                 zlog_debug("netlink_route_multipath() (multihop): "
                                        "nexthop via if %u", nexthop->ifindex);
                            }
                else if (nexthop->type == NEXTHOP_TYPE_IPV6_IFNAME
                   || nexthop->type == NEXTHOP_TYPE_IPV6_IFINDEX)
                            {
                            rtnh->rtnh_ifindex = nexthop->ifindex;

                            if (IS_ZEBRA_DEBUG_KERNEL)
                                 zlog_debug("netlink_route_multipath() (multihop): "
                                        "nexthop via if %u", nexthop->ifindex);
                            }
                else
                            {
                            rtnh->rtnh_ifindex = 0;
                            }
              }
            rtnh = RTNH_NEXT (rtnh);

            if (cmd == RTM_NEWROUTE)
              SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);
          }
        }
      if (src)
        addattr_l (&req.n, sizeof req, RTA_PREFSRC, &src->ipv4, bytelen);

      if (rta->rta_len > RTA_LENGTH (0))
        addattr_l (&req.n, NL_PKT_BUF_SIZE, RTA_MULTIPATH, RTA_DATA
(rta),
```

```
                   RTA_PAYLOAD (rta));
    }

  /* If there is no useful nexthop then return. */
  if (nexthop_num == 0)
    {
      if (IS_ZEBRA_DEBUG_KERNEL)
        zlog_debug ("netlink_route_multipath(): No useful nexthop.");
      return 0;
    }

skip:

  /* Destination netlink address. */
  memset (&snl, 0, sizeof snl);
  snl.nl_family = AF_NETLINK;

  /* Talk to netlink socket. */
  return netlink_talk (&req.n, &netlink_cmd);
}
```

c. For routing socket , the implementation in zebra\rt_socket.c

```
/* Interface between zebra message and rtm message. */
static int
kernel_rtm_ipv4 (int cmd, struct prefix *p, struct rib *rib, int family)

{
  struct sockaddr_in *mask = NULL;
  struct sockaddr_in sin_dest, sin_mask, sin_gate;
  struct nexthop *nexthop;
  int nexthop_num = 0;
  unsigned int ifindex = 0;
  int gate = 0;
  int error;
  char prefix_buf[INET_ADDRSTRLEN];

  if (IS_ZEBRA_DEBUG_RIB)
    inet_ntop (AF_INET, &p->u.prefix, prefix_buf, INET_ADDRSTRLEN);
  memset (&sin_dest, 0, sizeof (struct sockaddr_in));
  sin_dest.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
  sin_dest.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
  sin_dest.sin_addr = p->u.prefix4;
```

```
  memset (&sin_mask, 0, sizeof (struct sockaddr_in));

  memset (&sin_gate, 0, sizeof (struct sockaddr_in));
  sin_gate.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
  sin_gate.sin_len = sizeof (struct sockaddr_in);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */

  /* Make gateway. */
  for (nexthop = rib->nexthop; nexthop; nexthop = nexthop->next)
    {
      gate = 0;
      char gate_buf[INET_ADDRSTRLEN] = "NULL";

      /*
       * XXX We need to refrain from kernel operations in some cases,
       * but this if statement seems overly cautious - what about
       * other than ADD and DELETE?
       */
      if ((cmd == RTM_ADD
	   && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_ACTIVE))
	  || (cmd == RTM_DELETE
	      && CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB)
	      ))
	{
	  if (CHECK_FLAG (nexthop->flags, NEXTHOP_FLAG_RECURSIVE))
	    {
	      if (nexthop->rtype == NEXTHOP_TYPE_IPV4 ||
		  nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX)
		{
		  sin_gate.sin_addr = nexthop->rgate.ipv4;
		  gate = 1;
		}
	      if (nexthop->rtype == NEXTHOP_TYPE_IFINDEX
		  || nexthop->rtype == NEXTHOP_TYPE_IFNAME
		  || nexthop->rtype == NEXTHOP_TYPE_IPV4_IFINDEX)
		ifindex = nexthop->rifindex;
	    }
	  else
	    {
	      if (nexthop->type == NEXTHOP_TYPE_IPV4 ||
		  nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
```

```
                        {
                          sin_gate.sin_addr = nexthop->gate.ipv4;
                          gate = 1;
                        }
                  if (nexthop->type == NEXTHOP_TYPE_IFINDEX
                          || nexthop->type == NEXTHOP_TYPE_IFNAME
                          || nexthop->type == NEXTHOP_TYPE_IPV4_IFINDEX)
                        ifindex = nexthop->ifindex;
                  if (nexthop->type == NEXTHOP_TYPE_BLACKHOLE)
                        {
                          struct in_addr loopback;
                          loopback.s_addr = htonl (INADDR_LOOPBACK);
                          sin_gate.sin_addr = loopback;
                          gate = 1;
                        }
                }

              if (gate && p->prefixlen == 32)
                mask = NULL;
              else
                {
                  masklen2ip (p->prefixlen, &sin_mask.sin_addr);
                  sin_mask.sin_family = AF_INET;
#ifdef HAVE_STRUCT_SOCKADDR_IN_SIN_LEN
                  sin_mask.sin_len = sin_masklen (sin_mask.sin_addr);
#endif /* HAVE_STRUCT_SOCKADDR_IN_SIN_LEN */
                  mask = &sin_mask;
                }

              error = rtm_write (cmd,
                                 (union sockunion *)&sin_dest,
                                 (union sockunion *)mask,
                                 gate ? (union sockunion *)&sin_gate : NULL,
                                 ifindex,
                                 rib->flags,
                                 rib->metric);

            if (IS_ZEBRA_DEBUG_RIB)
            {
              if (!gate)
              {
                zlog_debug ("%s: %s/%d: attention! gate not found for rib %p",
                  __func__, prefix_buf, p->prefixlen, rib);
```

```
      rib_dump (__func__, (struct prefix_ipv4 *)p, rib);
     }
     else
      inet_ntop (AF_INET, &sin_gate.sin_addr, gate_buf,
INET_ADDRSTRLEN);
    }

    switch (error)
    {
     /* We only flag nexthops as being in FIB if rtm_write() did its work. */
     case ZEBRA_ERR_NOERROR:
      nexthop_num++;
      if (IS_ZEBRA_DEBUG_RIB)
       zlog_debug ("%s: %s/%d: successfully did NH %s",
        __func__, prefix_buf, p->prefixlen, gate_buf);
      if (cmd == RTM_ADD)
       SET_FLAG (nexthop->flags, NEXTHOP_FLAG_FIB);
      break;

     /* The only valid case for this error is kernel's failure to install
      * a multipath route, which is common for FreeBSD. This should be
      * ignored silently, but logged as an error otherwise.
      */
     case ZEBRA_ERR_RTEXIST:
      if (cmd != RTM_ADD)
       zlog_err ("%s: rtm_write() returned %d for command %d",
        __func__, error, cmd);
      continue;
      break;

     /* Given that our NEXTHOP_FLAG_FIB matches real kernel FIB, it isn't
      * normal to get any other messages in ANY case.
      */
     case ZEBRA_ERR_RTNOEXIST:
     case ZEBRA_ERR_RTUNREACH:
     default:
      /* This point is reachable regardless of debugging mode. */
      if (!IS_ZEBRA_DEBUG_RIB)
       inet_ntop (AF_INET, &p->u.prefix, prefix_buf,
INET_ADDRSTRLEN);
      zlog_err ("%s: %s/%d: rtm_write() unexpectedly returned %d for
command %s",
       __func__, prefix_buf, p->prefixlen, error, lookup (rtm_type_str, cmd));
```

```
              break;
          }
        } /* if (cmd and flags make sense) */
      else
        if (IS_ZEBRA_DEBUG_RIB)
          zlog_debug ("%s: odd command %s for flags %d",
            __func__, lookup (rtm_type_str, cmd), nexthop->flags);
    } /* for (nexthop = ... */

  /* If there was no useful nexthop, then complain. */
  if (nexthop_num == 0 && IS_ZEBRA_DEBUG_KERNEL)
    zlog_debug ("%s: No useful nexthops were found in RIB entry %p", __func__,
rib);

  return 0; /*XXX*/
}
```