Abstract

Author:	Lei Xiao
Title of the Thesis:	Ad Hoc Routing Framework design and implement
Date:	13th March 2003 Number of pages: 72
Faculty:	Networking Laboratory Helsinki University of Technology (HUT)
Supervisor:	Professor Raimo Kantola
Instructor:	MSc. Jose M. Costa Requena

Ad Hoc network was original designed for military purposes, but it was re-emerging as a hot research topic recent years. There are numerous potential applications for such network: conference, military networks, and personal area network, just to name a few.

There are more research done for Ad hoc routing and has many proposals, but the implementations are relative few. The goal of this master thesis is to develop an Ad Hoc routing Framework on Linux. The framework should support different routing protocols, such as proactive and reactive protocols. Based on it, AODV, as a reactive routing protocol, is deployed.

To achieve this goal, the framework is designed to have two parts: Forwarding Engine and Control Panel. Forwarding Engine resides in Kernel space and adds functionality, which is required by framework, into kernel route function. Control panel contains the detailed routing algorithms and controls different protocols.

The framework consists of many components. Each component has clear defined interface. By keeping the interface consistent, the individual component can change the internal implementation while the whole framework still can be integrated easily.

At last, a test bed with four iPAQs was build, which integrated the framework. Several test cases are performed on test bed and show that the framework works properly for three hops.

Keywords: Ad Hoc networks, AODV, framework, reactive protocol, proactive protocol

Preface

This master's thesis has been written at the Networking Laboratory of Helsinki University of Technology for the MobileMan project funded by the European Union.

I would like to thank the supervisor of the thesis, Professor Raimo Kantola, for the opportunity he gave to me for doing this master thesis, and the instructor, MSc. Jose M. Costa Requena, for his invaluable tutoring and help.

I also thank all the other people in the project team for their supporting so that I could finish the master thesis in six months.

Finally, I would like to express my utmost gratitude to my dear family and friends for all their helps during this busy period.

March 13th 2003
Espoo, Finland

Lei Xiao

Index of contents

ABSTRACT I

PREFACE II

INDEX OF CONTENTS III

LIST OF ABBREVIATIONS AND ACRONYMS VI

LIST OF FIGURES VII

LIST OF TABLES VIII

LIST OF TABLES VIII

CHAPTER 1 1

1. INTRODUCTION 1

1.1. BACKGROUND 1
1.2. TASKS AND STEPS 2
1.3. STRUCTURE OF THE THESIS 2

CHAPTER 2 4

2. AD HOC NETWORKS 4

2.1. OVERVIEW OF AD HOC NETWORKS 4
2.2. DIFFERENT ROUTING PROTOCOLS FOR AD HOC NETWORKS 5
2.3. AD HOC ON-DEMAND DISTANCE-VECTOR PROTOCOL (AODV) 6
2.3.1. Path Discovery 10
2.3.2. Route Table Management 10
2.3.3. Path Maintenance 11
2.3.4. Local Connectivity Management 11

CHAPTER 3 14

3. AD HOC FRAMEWORK DESIGN 14

3.1. ROUTING FUNCTIONALITIES IN MOST UNIX/LINUX OPERATING SYSTEMS 14
3.2. SPECIFIC REQUIREMENTS OF REACTIVE ROUTING PROTOCOL 15
3.3. AD HOC ROUTING FRAMEWORK REQUIREMENTS 15
3.4. EXISTING IMPLEMENTATIONS AND THEIR PROBLEMS 16
3.5. LINUX OPERATING SYSTEM 17
3.5.1. Open Kernel Source 18
3.5.2. Netlink 18
3.5.3. Netfilter 20
3.5.4. Kernel Loadable Module 21

CHAPTER 4 22

4. AD HOC ROUTING FRAMEWORK IMPLEMENTATION 22

4.1. AD HOC ROUTING FRAMEWORK 22
4.2. ODRM MODULE 23
4.2.1. Forwarding Engine 24
- **DUMMY NETWORK DRIVER** 26
- **IPQ** 26
- **RTM** 27

4.2.2. Connector 28
4.2.3. Controller 28
- **LIBIPQ** 30
- **API** 31

4.3. AODV MODULE 33
4.3.1. on_boo_wait 33
4.3.2. active 35

CHAPTER 5 54

5. TESTS AND ANALYSIS 54

5.1. FRAMEWORK INTEGRATION 54
5.2. TEST METHOD AND RESULT 55
5.3. RESULT ANALYSIS 59

CHAPTER 6 61

6. CONCLUSIONS AND FUTURE WORK 61

REFERENCES 62

APPENDIX A. AODV DAEMON CONFIGURATION: 64

List of abbreviations and acronyms

ABR	Associativity-Based Routing
AODV	Ad Hoc On-Demand Distance Vector
CBGR	Cluster Based Gateway Switch Routing
CBRP	Cluster Based Routing Protocol
CP	Control Panel
DSDV	Destination Sequence Distant-Vector
DSR	Dynamic Source Routing
FCS	Future Combat System
FE	Forwarding Engine
FSR	Fisheye State Routing Protocol
GloMo	Global Mobile
GNU	GUN is Not Unix
IETF	Internet Engineering Task Force
KLM	Kernel Loadable Module
MANET	Mobile Ad-hoc NETworks
ODRM	On Demand Routing Module
OSLR	Optimized Link State Routing
PAN	Personal Area Network
PDA	Personal Digital Assistant
POSIX	Portable Operating System Interface
PRNet	Packet Radio Network
QoS	Quality of Service
RLM	Route Lost Module
RTM	Route Timer Module
SSR	Signaling Stability Based adaptive Routing
STAR	Source Tree Adaptive Routing
SURAN	SURvivable Adaptive Network
TORA	Temporally-Ordered Routing Algorithm
WLAN	Wireless Local Area Network
WRP	Wireless Routing Protocol
ZRP	Zone Routing Protocol

List of figures

FIGURE 1 AODV STATE MACHINE..12
FIGURE 2 AD HOC FRAMEWORK CONCEPT DIAGRAM..................................16
FIGURE 3 NETFILTER ARCHITECTURE 20
FIGURE 4 ODRM ARCHITECTURE...24
FIGURE 5 FE_START..24
FIGURE 6 CAPTURE_PKT 25
FIGURE 7 RECEIVE_PKT...26
FIGURE 8 WAIT STATE 28
FIGURE 9 RECEIVE_MSG...29
FIGURE 10 SEND_REQUEST...29
FIGURE 11 KERNEL_RT_MANAGEMENT 30
FIGURE 12 ODRM TIME SEQUENCE..32
FIGURE 13 AODV ON_BOOT_WAIT..34
FIGURE 14 HASH TABLE FOR AODV ROUTING TABLE 35
FIGURE 15 AODV ACTIVE..36
FIGURE 16 ROUTE_DISCOVERY..38
FIGURE 17 PROCESS_NEIGHBOR 39
FIGURE 18 UPDATE_RT 40
FIGURE 19 INSERT_RT...41
FIGURE 20 PROCESS_RREQ 43
FIGURE 21 PROCESS_RREP...43
FIGURE 22 FORWARD_RREP 44
FIGURE 23 PROCESS_RERR 45
FIGURE 24 PROCESS_HELLO...46
FIGURE 25 PROCESS_TIMER_EVENT 47
FIGURE 26 ROUTE_DISCOVERY_TIMEOUT 48
FIGURE 27 ROUTE_EXPIRE_TIMEOUT...49
FIGURE 28 RREP_ACK_TIMEOUT..49
FIGURE 29 RREQ_RECORD_TIMEOUT 50
FIGURE 30 ROUTE_DELETE_TIMEOUT..50
FIGURE 31 LOCAL_REPAIR_TIMOUT 51
FIGURE 32 HELLO_TIMEOUT 51
FIGURE 33 NEIGHBOR_LINK_BREAK 52
FIGURE 34 ONBOOT_WAIT_TIMEOUT 53
FIGURE 35 TEST ENVIRONMENT 56

List of tables

TABLE 1 REACTIVE AND PROACTIVE PROTOCOLS....................6
TABLE 2 ROUTE REQUEST (RREQ) MESSAGE FORMAT....................7
TABLE 3 ROUTE REPLY (RREP) MESSAGE FORMAT....................8
TABLE 4 ROUTE ERROR (RERR) MESSAGE FORMAT....................9
TABLE 5 ROUTE REPLY ACKNOWLEDGEMENT (RREP-ACK) MESSAGE FORMAT..10
TABLE 6 NETLINK ROUTE MESSAGE 19
TABLE 7 NETLINK IPQ MESSAGE 19
TABLE 8 TEST RESULT....................59

Chapter 1

1.Introduction

1.1. Background

The Ad Hoc networks are formed by wireless nodes, which may be mobile. This kind of network may exist without using a pre-existing infrastructure. The paths between nodes may change and potentially contain multiple hops, which makes the routing a problematic area.

The purpose of Ad Hoc network is to ease the deployment. There are many applications, for example: Personal area networking, which includes cell phone, laptop, earphone, and wristwatch; Military environments, which includes soldiers, tanks and planes; Civilian environments, which includes taxi network, meeting room, sports stadiums, boats and aircraft; Emergency operations, which includes search, rescue, policing and fire fighting.

Ad Hoc networks were original designed for military purposes, but now it was re-emerged as the next network generation. There is a lot of research on Ad Hoc networks, but implementations are relatively few. The implementation is the most effective way to validate the design. MobileMan[1] is a EU project that aims to implement a test bed for Ad Hoc network in real life. This test bed reaches all network layers of Ad Hoc networks, from application layer to physical layer.

As a part of MobileMan, our project focuses on routing and service discovery of Ad Hoc networks. The goal of our project is to implement an Ad Hoc routing framework which can support different routing protocols, such as proactive, reactive and hybrid. With this framework, we could add other functionalities, such as service discovery.

This thesis will focus on implementing the basic functionalities for the framework and deploying a reactive module (AODV). These basic functionalities will be organized as a common module and provided as a library for the framework.

1.2. Tasks and steps

The main goal is to design a framework for Ad Hoc routing protocol on Linux operating system. The framework should provide general functionalities for both proactive and reactive routing. As part of this thesis, only a reactive protocol, AODV will be implemented on top of the framework. After implementation, the framework will be integrated into certain number of Personal Digital Assistants (PDA) for testing and validating.

To achieve this goal, we follow these steps:

- Design the whole architecture for Ad Hoc routing framework.
- Write the generic reactive routing module, which is called On Demand Routing Module (ODRM).
- Write the AODV daemon, which is based on Uppsala University implementation.
- Integrate framework on iPAQ
- Have the test bed on iPAQ for obtaining performance result.

1.3. Structure of the thesis

Chapter 2 presents the study of characteristics of Ad Hoc networks. Afterwards, it focuses on different routing protocols of Ad Hoc networks. Finally, it analyzes AODV, as an example of reactive routing protocol, in details.
Chapter 3 describes Ad hoc framework design. Firstly, it analyzes the routing functionalities of UNIX operating system (including Linux). Secondly, based on this analysis, it gives the requirements of our framework. Afterwards, it compares different AODV implementations and summarizes their problems. Finally, the Linux networking and kernel programming facilities are described.
Chapter 4 describes the framework and reactive protocol implementation. It gives ODRM and AODV module implementation in details, which includes system UML/SDL chart, data structure, algorithms, special APIs from Linux and exported by ODRM. ODRM is a generic

module, which provides support for any reactive routing protocol implementation. It also includes a simple kernel route table management API, which can also be used by proactive routing daemon. AODV module uses ODRM functionalities and is rewritten on Uppsala implementation.

Chapter 5 presents the validation and testing of implementation. The framework is integrated into four iPAQs. Different test cases are performed and the results are analyzed.

Chapter 6 comes with some conclusions and future work. It describes the possibilities of adding more functionality into the framework.

Chapter 2

2. Ad Hoc Networks

This chapter presents an overview of Ad Hoc networks and its relevance in actual communication area. Afterwards, we analyze the challenges faced when implementing Ad Hoc networks and the main routing protocols, which can be separated into three categories: proactive, reactive and hybrid. Finally, we describe AODV, as an example of reactive routing algorithm.

2.1. Overview of Ad Hoc networks

Ad Hoc networking is not a new technology. It was originally designed for military purpose in 1970's. The beginning of Ad Hoc networks is from ALOHA system project, which is started in 1968. Based on the experience of ALOHA network, DARPA started the project of Packet Radio Network (PRNET) in 1972. To extend the PRNET technology, DARPA initiated the Survival Adaptive Network (SURAN) project in 1983. In 1994, DARPA continued to develop Ad Hoc networks to satisfy military requirements and started Global Mobile (GloMo) project. Even now DARPA is still supporting various projects for military purpose, such as Future Combat Systems (FCS)[1].

But the commercial Ad Hoc networking research is just starting in recent years. The Internet Engineering Task Force (IETF) MANET [2] working group is formed in June 1997. The aim of MANET working group is to improve routing specification standards within the current Internet protocol stack and lead to an open, flexible and extensible architecture. Besides routing standards, MANET also pays attention to other commercial initiatives, such as IEEE Wireless LAN (WLAN) standard, 802.11and Bluetooth.

[1] The history of Ad Hoc networking is from [3]

In general, Ad Hoc network is a network formed by mobile nodes without central administration. The nodes have to serve both as routers and hosts. They are able to communicate wirelessly with each other by sending and receiving data packets. There are lots of challenges for Ad Hoc network implementation because of its features, such as:

- Dynamic network topology because nodes may join or leave the network at any time.
- Limited wireless transmission rang
- Broadcast nature of the wireless medium
- Packet losses due to transmission error
- Battery constrains
- Ease of snooping on wireless transmissions

The Ad Hoc network attracts more attention in recent years because it is predicted to be next network generation. With the development of personal computing devices, such as smart phone and PDA, people have more powerful devices that have the networking ability. These devices can form Personal Area Networking (PAN) to share the data among them and access other networks, such as Internet. There exists huge market potential for Ah Hoc network because personal computing device is considered as a “Killer Application”.

But the challenges of Ad Hoc network make it difficult to be implemented. Many solutions proposed trying to address a sub-space of the problem domain. It is very hard to have a one-size-fit-all solution.

2.2. Different routing protocols for Ad Hoc networks

Among all the research in Ad Hoc networks, routing protocol are getting most emphasis.

Like other network protocols, Ad Hoc network requires all the protocol layers to be suitable for its characteristics.

There is a lot of research work done on routing protocols, which is focused on network layer. There exits tens of Ad Hoc routing protocol proposals, but relative few proposals on other layers. As the result of routing protocol research shows, more work is needed on other layers.

Ad Hoc routing protocols can be divided into two main categories: proactive and reactive.

Proactive routing protocol is mostly used in today's network. It stores all the route information in route table before hand. When there is route request, it will generate route reply based on the information on route table. This protocol requires refreshing the route table periodically so the router will always keep the latest route information.

Reactive routing protocol is used to fulfill the requirement of Ad Hoc network. Ad Hoc Network doesn't have a stable topology and each node itself is a router. The nodes in the network keep on moving so the routes between them change frequently. It is not feasible to have all the route information before hand. Further more, it is not good for refreshing route information among nodes frequently since this will cause lots of traffic in the network. Reactive routing protocol only tries to find and reply route when there comes route request. So it is also called on demand routing protocol.

It is also possible to combine both proactive and reactive routing to have a hybrid protocol, for example ZRP [4], which could have the benefit of both protocols.

Some proactive and reactive routing are listed below:

Reactive Protocols	Proactive Protocols
ABR	CBGR
AODV	DSDV
TORA	FSR
SSR	OLSR
DSR	STAR
CBRP	WRP

Table 1 Reactive and Proactive Protocols

2.3. Ad Hoc On-Demand Distance-Vector protocol (AODV)

AODV [5] is a reactive routing protocol. It provides quick and efficient route establishment between nodes that desire communication and aims to reduce control overhead and route discovery latency. It uses sequence number to avoid infinitely route loop.

AODV is initialized by Charles E. Perkins and improved from Destination-Sequenced Distance-Vector (DSDV) routing algorithm. AODV does not attempt to maintain routes from every node to every other node in the network, but only the routes that are using. Routes are discovered on demand and are maintained only as long as they are necessary. AODV is able to provide unicast, multicast, and broadcast communication ability. We only discuss unicast and broadcast here.

AODV utilizes four control messages for route discovery, route establishment and route maintenance: Route Request (RREQ), Route Reply (RREP), Route Error (RERR) and Route Reply Acknowledgment (RREP-ACK).

										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							
Destination IP Address																															
Destination Sequence Number																															
Originator IP Address																															
Originator Sequence Number																															

Table 2 Route Request (RREQ) Message Format

The format of RREQ message is illustrated above, and contains the following fields:

Type	1
J	Join flag; reserved for multicast.
R	Repair flag; reserved for multicast.
G	Gratuitous RREP flag; indicates whether a gratuitous RREP should be unicast to the node specified in the Destination IP Address field.
U	Unknown sequence number; indicates the destination sequence number is unknown.
Reserved	Sent as 0; ignored on reception.
Hop Count	The number of hops from the Originator IP Address to the node handling the request.
RREQ ID	A sequence number uniquely identifying the particular RREQ when taken in conjunction with the originating node's IP address.
Destination IP Address	The IP address of the destination for which a route is desired.

Destination Sequence Number

The greatest sequence number received in the past by the originator for any route towards the destination.

Originator IP Address

The IP address of the node which originated the Route Request.

Originator Sequence Number

The current sequence number to be used for route entries pointing to (and generated by) the originator of the route request.

										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								R	A	Reserved														Hop Count							
Destination IP Address																															
Destination Sequence Number																															
Originator IP Address																															
Lifetime																															

Table 3 Route Reply (RREP) Message Format

The format of the Route Reply message is illustrated above, and contains the following fields:

Type	2
R	Repair flag; used for multicast.
A	Acknowledgment required.
Reserved	Sent as 0; ignored on reception.
Prefix Size	If nonzero, the 5-bit Prefix Size specifies that the indicated next hop may be used for any nodes with the same routing prefix (as defined by the Prefix Size) as the requested destination.
Hop Count	The number of hops from the Originator IP Address to the Destination IP Address. For multicast route requests this indicates the number of hops to the multicast tree member sending the RREP.

Destination IP Address

The IP address of the destination for which a route is supplied.

Destination Sequence Number

The destination sequence number associated to the route.

Originator IP Address

The IP address of the node which originated the RREQ for which the route is supplied.

Lifetime	The time in milliseconds for which nodes receiving the RREP consider the route to be valid.

										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								N	Reserved															DestCount							
Unreachable Destination IP Address (1)																															
Unreachable Destination Sequence Number (1)																															
Additional Unreachable Destination IP Addresses (if needed)																															
Additional Unreachable Destination Sequence Numbers (if needed)																															

Table 4 Route Error (RERR) Message Format

The format of the Route Error message is illustrated above, and contains the following fields:

Type	3
N	No delete flag; set when a node has performed a local repair of a link, and upstream nodes should not delete the route.
Reserved	Sent as 0; ignored on reception.
DestCount	The number of unreachable destinations included in the message; MUST be at least 1.

Unreachable Destination IP Address

The IP address of the destination that has become unreachable due to a link break.

Unreachable Destination Sequence Number

The sequence number in the route table entry for the destination listed in the previous Unreachable Destination IP Address field.

0										1					
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
Type								Reserved							

Table 5 Route Reply Acknowledgement (RREP-ACK) Message Format

Type	4
Reserved	Sent as 0; ignored on reception.

AODV algorithm can be summarized as four aspects: path discovery, route table management, path maintenance, and local connectivity management.

2.3.1. Path Discovery

Route discovery is purely on demand and follows a route request/route reply discovery cycle. It includes reverse path setup and forward path setup and described as below:

1. When a node needs a route to a destination, it broadcast a RREQ.
2. Neighbor node gets this RREQ. If it has a path to the destination, the node unicasts a RREP along the reverse path to the source node. Otherwise, it re-broadcasts RREQ and sets up reverse path.
3. When a node receives RREP, it sets up forward path. This path is used for sending packet from the source to the destination.
4. Once a route has been discovered for a given source/destination pair, it is maintained as long as needed by the source node.

2.3.2. Route Table Management

Route table has the information obtained through RREQ and RREP. It includes all the active nodes and active paths. The active nodes are kept in precursor list. They are the nodes from which packets may emanate to be forwarded to the destination. They must be notified when the link to the next hop is broken.
Each route entry has a lifetime. After lifetime is expired, the entry is purges from route table. This makes sure that unused routes grow stale quickly in the network.

2.3.3. Path Maintenance

Path maintenance is only applicable to those active nodes. But lifetime expiration for active nodes doesn't trigger path maintenance. When either the destination or some intermediate node moves, a Route Error (RERR) message is sent to the affected source nodes. It is shown as following:

1. Node upstream of break broadcasts RERR
2. Neighboring nodes propagate RERR if they use source of RERR as next hop for destination.
3. Step 2 is repeated until all the nodes that have this broken destination in their route tables are reached. When a source node receives the RERR, it can reinitiate route discovery if this route is still needed.

.

2.3.4. Local Connectivity Management

Local connectivity is managed by HELLO message and network monitoring. HELLO message is a special RREP message with destination equals source and hop counts to zero. If a node doesn't receive any packet within a certain period from a known neighbor, the connection between them is considered broken and path maintenance might start.
To guarantee the connectivity between neighbors, the node can send out HELLO message if there is no any packet exchanging between them in a certain time.

The route discovery/establishment/maintenance can be drawn as a state machine as follow:

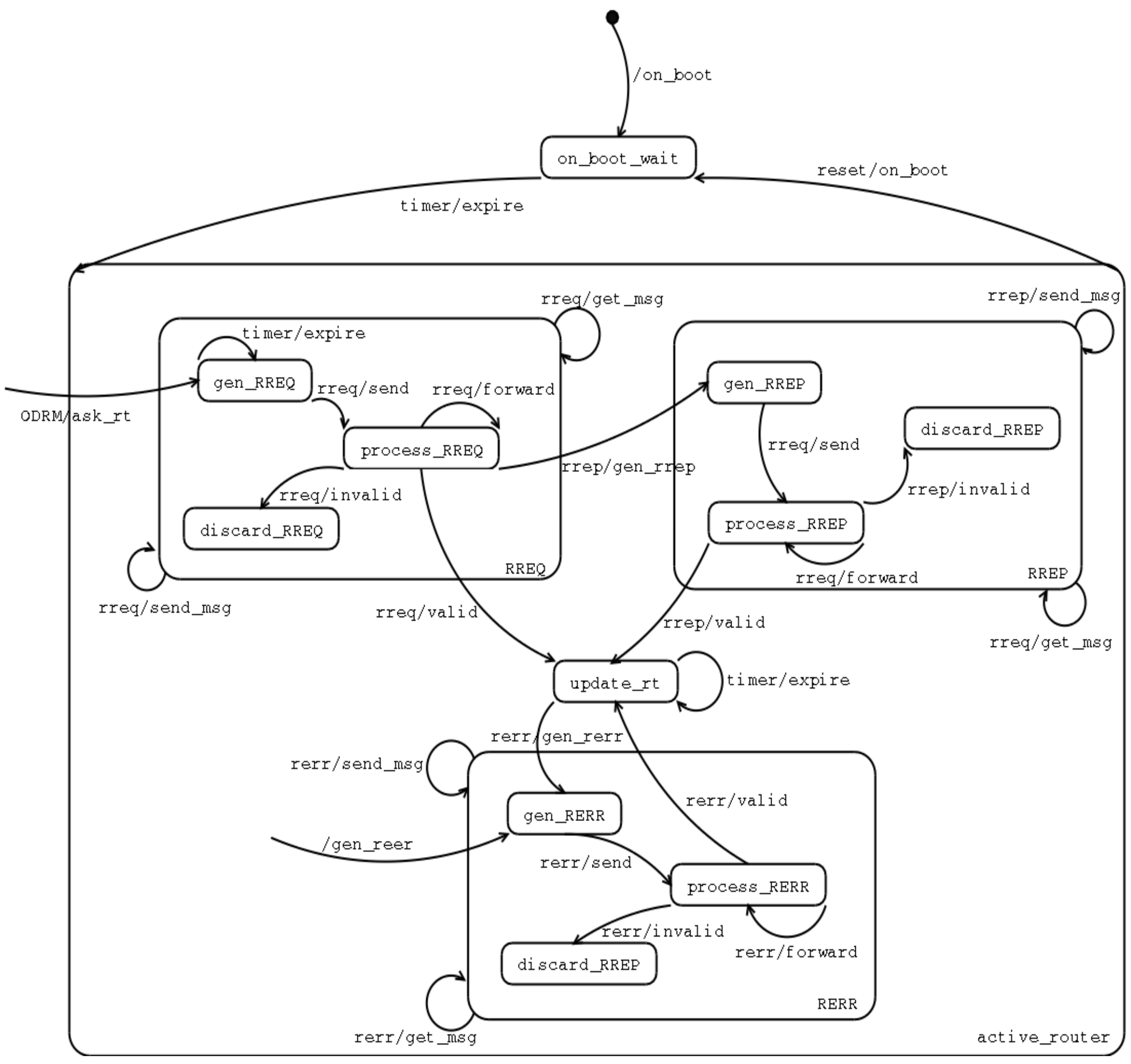

Figure 1 AODV State Machine

There is an "on_boot_wait" state in Figure 1, which is for eliminating potential routing loops. When a node reboots, it may lose all sequence records for all destinations, including its own sequence number. But neighbor nodes might still use this node as active next hop. This can potentially create routing loops.

To prevent this happens, a node must stay in on_boot_wait state for a certain time after it starts/reboots. During this period, it doesn't transmit any RREP message. If it receives any RREQ, RREP or RERR control message, it should create corresponding route entry, but never forward these messages. If it receives any data packet from other nodes, it should broadcast RERR message.

AODV has been proposed as one of the Ad Hoc routing protocol standards [2]. It is evolving further and having more advanced features, such as QoS extensions [6], Service Location [7], IP version 6 support [8], etc.

Chapter 3

3. Ad Hoc Framework Design

This chapter analyzes the common routing functions provided by UNIX/Linux Operating System. Then it describes the characteristics of reactive routing protocol. It finds out that UNIX doesn't support reactive routing. Based on this analysis, it provides an Ad Hoc routing framework design. Afterwards, it analyzes other existing ad hoc routing protocols and their problems, comparing with our Ad Hoc framework design. Finally it describes the functionalities provided by Linux.

3.1. Routing functionalities in most Unix/Linux Operating Systems

Most UNIX/Linux operating systems have build-in network stack and it resides in kernel space of operating system. The routing function is based on network stack so it also resides in the kernel space. But as the network becomes complex, the route calculation algorithm becomes more complex and time consuming. It is not good to put all the functionalities into the kernel since this will hug the kernel too tight to response other requests quickly. Furthermore, users want to access kernel route table from user space for routing management, such as QoS.

For these purposes, most route functions are separated into two parts: one part is in user space, which does the most time consuming calculation, such as route discovery, and passes result to kernel space; one part is in kernel space, which only forwards the packet based on kernel route table. We call kernel space part packet forwarding function and user space part packet routing function. By doing so, the kernel is released from the burden for calculating route and has more time to do other tasks.

3.2. Specific requirements of reactive routing protocol

The kernel will decide the fate of the packet. If the packet destination does not exist in kernel routing table, this packet will be discarded silently by the kernel routing function and an ICMP error message may be sent to the application. This feature is working for traditional routing protocol, e.g. proactive routing protocol, but it cause problems for reactive routing protocol.
Reactive routing protocol does not need to know the routing information before hand. It will try to answer the route request when it arrives. For this reason, reactive routing protocol is also called on demand routing protocol.
On Demand Routing is not supported in most UNIX Operating Systems. To make on demanding routing work, we have to change the behavior of the kernel routing function. The new function will not drop the packet without route at once, but it will queue this packet in some place and query for a valid route. After some time, if a valid route is found, the packet will re-inject into the kernel for normal process. Otherwise, the packet will be dropped finally.

3.3. Ad Hoc Routing Framework requirements

Having the knowledge of how kernel routing function and on demanding routing works, we can decide how to design our Ad Hoc Routing Framework.

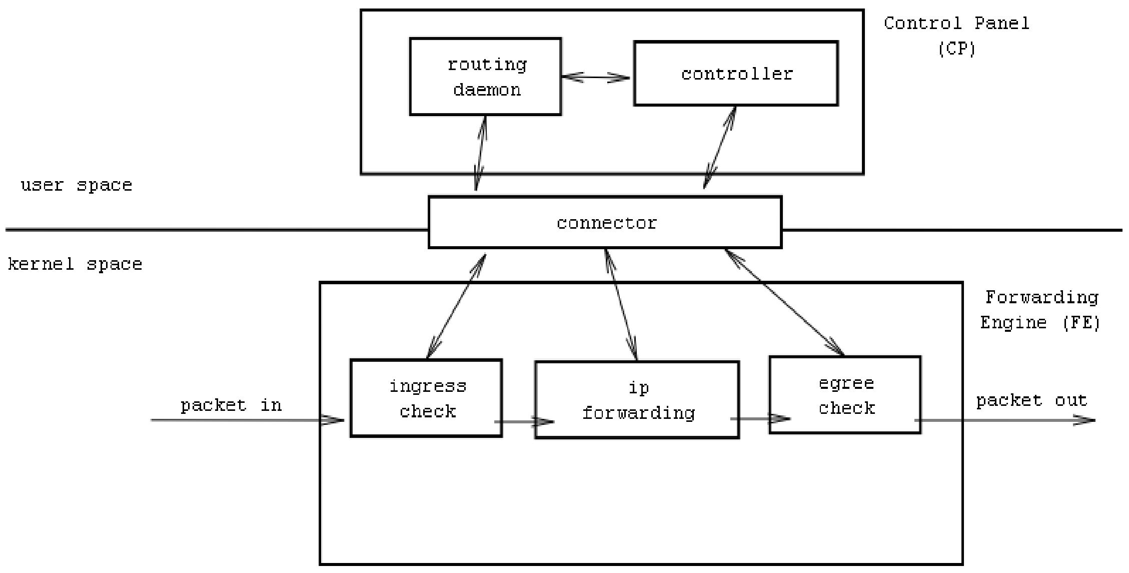

Figure 2 Ad Hoc Framework Concept Diagram

Figure 2 is the concept design of our Ad Hoc routing framework. We separate the framework into two parts: Control Panel, which is in the user space and Forwarding Engine, which is in the kernel space. Because two parts are resident in different privilege spaces, we need a connector to make them communicate with each other.

Forwarding Engine (FE) has three components: ingress check, IP forwarding and egress check. Ingress check is for checking the packet arriving from data link layer; Egress check is for checking the packet leaving for application layer; IP forwarding is for routing the packets. The packet travels in the kernel IP stack must pass these check points so that we can make decision how to deal with the packet. This decision is made by Control Panel (CP) in the user space.

CP has two components: routing daemon and controller. Routing daemon implements the detailed routing algorithm for calculating valid route. Controller is for deciding which algorithm to use for which kind of packet. Controller can register a special kind of packet to be minored. Then it may select a routing algorithm for this kind of packet.

Connector is the communication channel between CP and FE. Applications between user space and kernel space cannot talk to each other directly so we need a link between them. Connector is this kind of link which passes the message between two spaces.

3.4. Existing implementations and their problems

Here, we discuss only AODV implementations. Uppsala University [9] and UCSB [10] implementation are quite similar. They both use Netfilter to capture every packet and send it to user space by Netlink. The daemon in the user space checks this packet for valid route. If the route doesn't exist, the daemon sends route request, and wait for a certain time for route reply. If there is valid route reply, the packet can be re-injected into kernel and routed by newly established route; otherwise, the packet is dropped.

The problem for this kind of implementation is every packet passes to user space, no matter if it has route or not, and then passes back to kernel space. We know the overload of passing across two privilege spaces of operating system is quite high. Since every packet passes between user and kernel spaces twice, this will slow down the process.

NIST [11] implementation is totally done within kernel space. There is no user space daemon running. It also uses Netfilter to capture the packet. This might speed up process, but the kernel daemon may potentially crash the whole system if there has any problem. (Actually there had been a bug in /proc file which crashed the whole operating system.) Also the kernel daemon may consume too much kernel resources and makes operating system respond sluggishly. Even more, this implementation requires more programming skill since it is totally kernel system programming.

Based on the experiences of these existing AODV implementations, we decide our implementation should be:

1. Use Netfilter as packet checkpoint in the kernel.
2. Change kernel as less as possible.
3. Separate implementation into user space and kernel space. Actually this is what we had designed in our concept framework
4. Use Netlink as connector.

3.5. Linux Operating system

From the analysis before, we know that Ad Hoc framework implementation requires the access to the kernel of operating system and even some modification. This requirement leads into the need of having an operating system whose source can be viewed and changed. For this reason, the implementation is based on Linux operating system. Besides open source, Linux also provides other conveniences for implementing.

3.5.1. Open Kernel Source

We choose Linux operating system because its kernel source is available to everyone. Linux [12] is the operating system written from scratch for PC, but now it supports many others CPU architectures. Linux is part of GNU [13] open source project so we can get the source code and change it without any limitation. But we must allow other people to access our changed code and make any changes they want. By this way, software will be improved by many people and the knowledge is share among all programmers.

Linux is a clone of Unix so most Unix programming facilities can be used for it. Furthermore, because of its aim of POSIX compliance (an IEEE standard for operating system), Linux provides us a platform to write programs that are portable to other flavors of UNIX.

In practice, we choose kernel 2.4.18, which is currently used by most Linux distributions, such as Redhat and Debian. From version 2.4 onwards, the network stack of the kernel is changed dramatically. A new framework for packet filtering, which is called Netfilter, is added. This makes IP packet manipulating in a clean and easy way.

We make some changes for ip_queue and libipq, which are from Netfilter framework, so that Netfilter fits into our framework.

3.5.2. Netlink

Netlink [14] is a protocol used by Linux as both an intra-kernel messaging system as well as between kernel and user space.

Netlink acts in the framework as a connector between user and kernel space. It passes asynchronous messages between two spaces. The message is consisted by a message header and a message body. The message header defines what kind of message it is, such as request, respond; and message body contains data, such as IP packet, routing information. The detailed Netlink message for route and ipq are given below:

16	16	
message length		message header
mssage type	flag	
squence number		
process ID		
rtm_family	rtm_dst_len	
tm_src_len	rtm_tos	
rtm_table	rtm_protocol	
rtm_scope	rtm_type	
buffer …		

Table 6 netlink route message

message header …	
packet_id	
packet_mark	
timestamp_sec	
timerstamp_usec	
hook	Indev_name
outdev_name	hw_protocol
hw_type	hw_addlen
hw_addr	date_lenght
payload …	

Table 7 netlink ipq message

3.5.3. Netfilter

Netfilter [15] is a framework inside Linux 2.4.x kernel, which enables packet filtering, network address translation and other packet mangling. It is the re-designed and improved successor of the previous 2.2.x ipchains and 2.0.x ipfwadm systems.

Netfilter is a set of hooks inside the Linux 2.4.x kernel's network stack, which allows kernel modules to register callback functions called every time a network packet traverses one of those hooks.

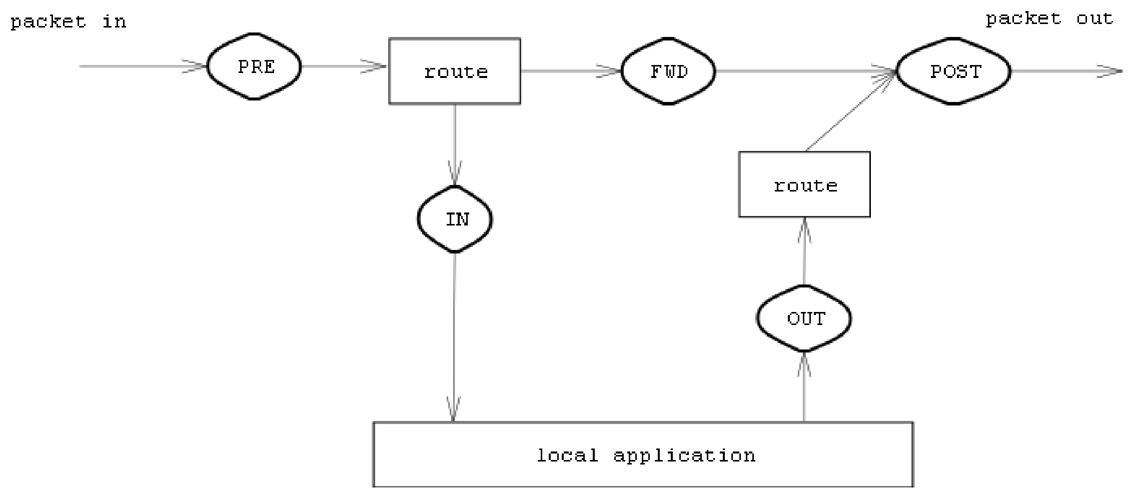

Figure 3 Netfilter Architecture

There are five hooks defined in Netfilter. A packet first pass PRE hook when it just comes from data link layer and passes simple sanity check. After routing, the packet for the local host will pass IN hook and go to upper stack layer, such as application layer. If the packet is for other hosts, it will not go to upper stack layer, but passed FWD hook and forward by the kernel. When packet arrives from upper layer, it passes OUT hook and then go through kernel route function. Finally all packets leaving network layer will pass POST hook.

In our framework, we register three hooks: FWD, POST and OUT. The packet arrives at FWD and POST hooks will be checked whether there is a valid route for it. If there is not, this packet will be queued and a route request is sent to routing daemon in use space. The way for defining packet without route and how the packet is passed to user space will be explained later. All packets with valid route have to pass OUT hook where the time stamp for the route is recorded for the future usage.

3.5.4. Kernel Loadable Module

Linux is a monolith kernel [16] operating system. This makes adding new functionalities into the kernel difficult. To ease this kind of task, Linux has a specific way to extend its kernel functionalities. This is Kernel Loadable Module [17] (KLM). KLM doesn't have to be compiled with kernel and can be loaded or unloaded (provided the module is not being used) at anytime by the user. KLM is mostly used as device driver, but not limited to that.

In our framework, we have two KLMs. One is to register three Netfilter hooks and corresponding callback functions and create Ethernet device driver. Theses functions are used for checking packet without route and recording route time stamp.

The other module provides functionality for communicating with user space. It takes care of sending packet to and receiving packet from user space.

Chapter 4

4. Ad Hoc Routing Framework Implementation

This chapter first describes in detail (UML/SDL, data structure, algorithm and API) how reactive Ad Hoc routing framework is developed. The framework consists of a generic module (ODRM) and a routing module (AODV) that uses ODRM. ODRM includes three parts: FE, connector and controller. FE have RLM, RTM, dummy network device driver and ipq. Connector is implemented by Netlink, ioctl and proc file. Controller exports an API to route daemons. Finally, how AODV module ported from Uppsala implementation is described. This also serves a role for validating ODRM.

4.1. Ad Hoc Routing Framework

As we have known, most UNIX (include Linux) operating systems drop the packet without valid route silently. But reactive routing protocol requires keeping such packet for a certain time while it tries finding valid route. Our design goal is to change the behavior of the operating system and make it work for all reactive routing protocols. Based on this, supporting for other routing protocols, such as OLSR, could be added.

The Ad Hoc routing framework consists of different modules: generic modules and specific modules.

The generic module provides some interfaces for other modules that will implement routing algorithms. In our framework, we define first generic module as On Demand Routing Module (ODRM). It provides basic functionalities for reactive routing protocols, such as queuing packets without route, sending the route request to route daemon and receiving the route reply from route daemon. It also includes kernel route table management functionality, which can be used by any routing daemon.

The specific module implements the concrete Ad Hoc routing algorithm. In our framework, we implement AODV for reactive routing protocol. But we can add any other reactive routing protocol as needed in the future since ODRM makes this very easy.

As shown in our concept design, the AODV daemon is the detailed algorithm implementation. ODRM includes components both in user and kernel space. It has the functionality of CP and FE.

Section 4.2 describes ODRM and section 4.3 analyzes AODV.

4.2. ODRM Module

ODRM includes the core functionalities of the Ad Hoc routing framework. It consists of different components and provides an API that is visible to other modules which will implement the specific routing algorithm.

ODRM includes three parts: FE, connector and controller. FE is in the kernel and changes the behavior of kernel: checks the IP packet, queues packet without valid route, sends message to user space, receives message from user space and re-injects /drops queued packet into kernel.

The connector between user and kernel space is implemented by Netlink, ioctl system call and proc file system.

Controller is in the user space and listens and receives route request from kernel space. It then passes this request to routing daemon. It also listens and receives route reply from routing daemon and passes this reply to kernel space. These functionalities are provided as an API. Controller also includes kernel route table management functionality. This functionality can be used by any routing daemon, both reactive and proactive.

This architecture is shown below:

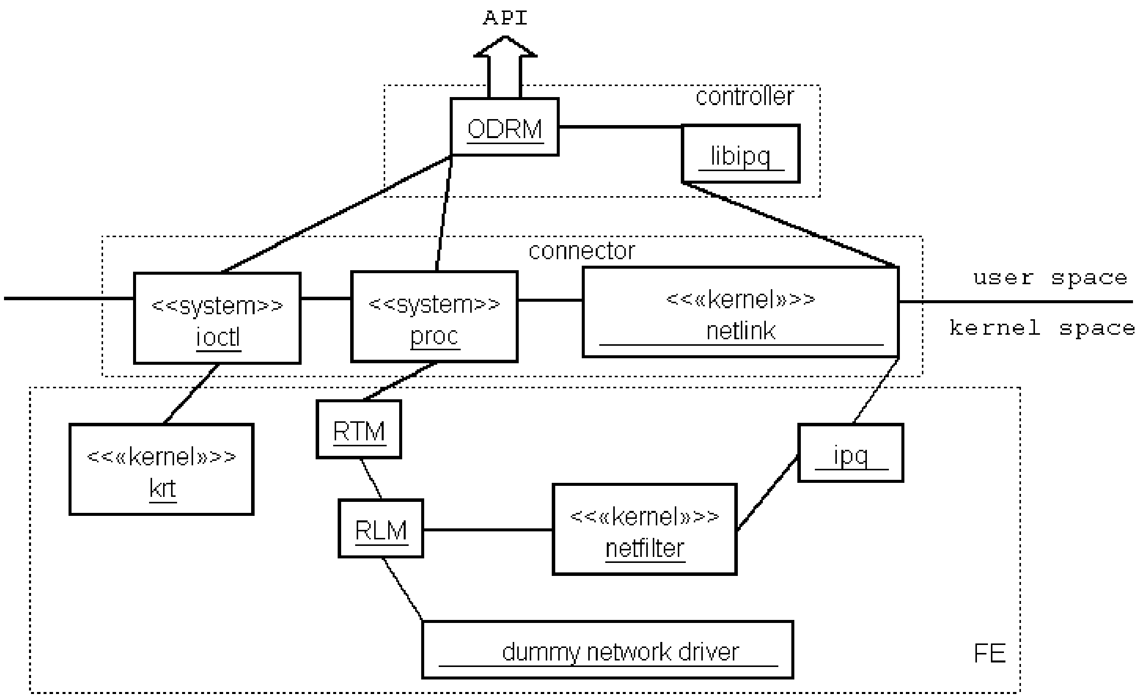

Figure 4 ODRM Architecture

4.2.1. Forwarding Engine

FE resides in kernel. It takes care of capturing and sending packet to user space and receiving packet from user space.

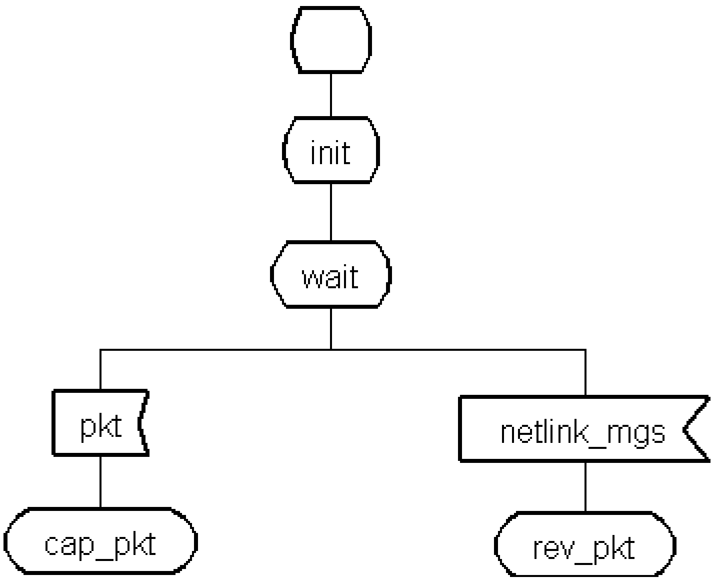

Figure 5 FE_start

When packets traverse three registered hooks, their destinations are checked. If destination is rt_lost, those packets are sent to user space.
If the packet is leaving POST hook, its timestamp is recorded.

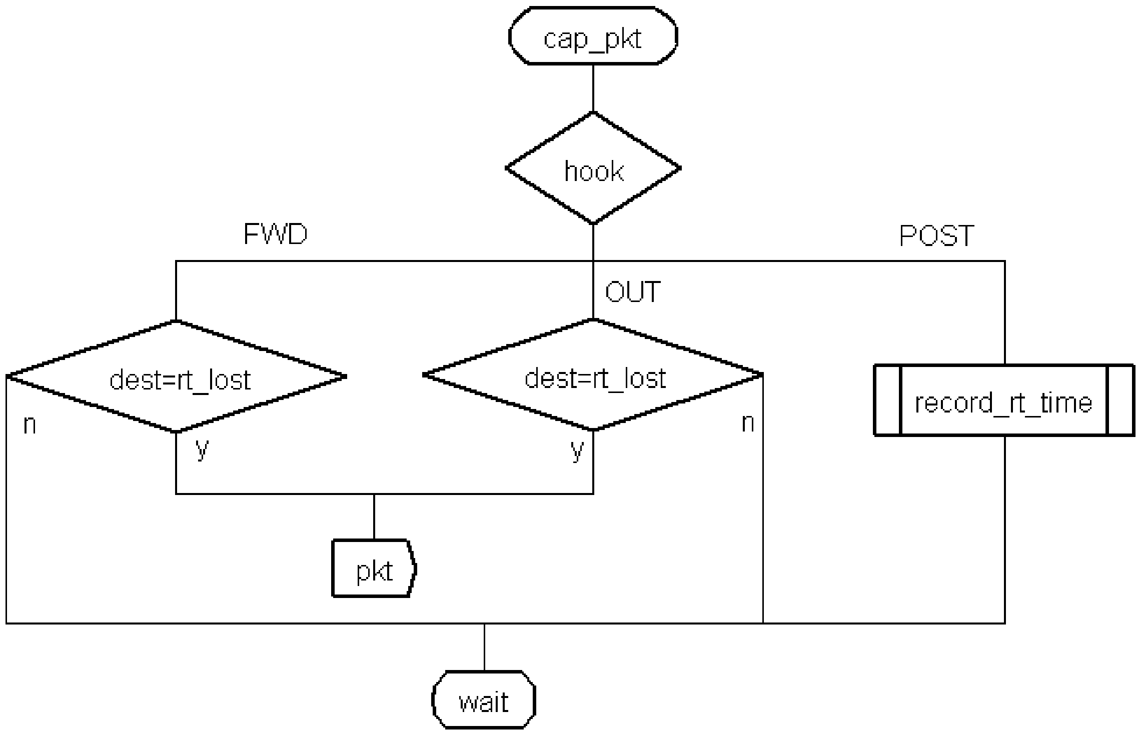

Figure 6 capture_pkt

When the packets come from user space, their type will be checked. If it is NF_DROP, this packet will be dropped. If it is NF_ACCEPT, this packet will be accepted and injected into the kernel.

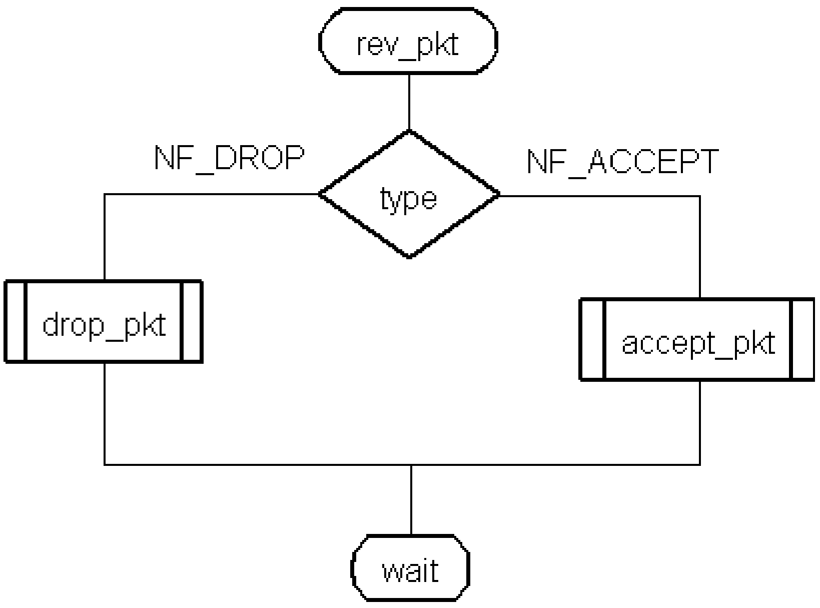

Figure 7 receive_pkt

FE has two components: Route Lost Module (RLM) and ipq, which is a rewritten ip_queue driver of Netfilter framework. They are KLMs.
RLM is the only entry point of FE functionality. It registers and actives Route Time module (RTM), dummy network driver and Netfilter. ipq is from ip_queue driver with a little change. Below is the detailed description of each component.

- Dummy network driver

This is a Linux device driver, particularly an Ethernet device driver. It is called dummy driver because it is not a real physical device, but a virtual device. Further more, it doesn’t include all functionalities of an Ethernet device. It doesn’t send and receive packet because packet never goes through it. It is just used to recognize packet without route. Because it is a simple Ethernet driver, we can use basic network admin tools to configure it. We start this Ethernet interface as rt_lost and set it as default gateway so all packets without route will arrive here at the end. Next we need ipq to capture these packets.

- **ipq**

First, we introduce ip_queue. Ip_queue is used to send and receive message between kernel and user space. The message type could be NF_ACCEPT, NF_DROP, NF_STOLEN and NF_QUEUE. The message body normally includes IP packet. When ip_queue sends a packet to user space, it marks the message header with NF_QUEUE. When it receives a packet from user space, it first checks message header. If message type is NF_DROP, ip_queue will inform kernel drop the queued packet. If message type is NF_ACCEPT, ip_queue will re-inject the packet into kernel.

We could use ip_queue queuing, sending and receiving packet easily. But there is a problem when we receive packet from user space with message type NF_ACCEPT. This means routing daemon has found a valid route and this new route has been established. The normal ip_queue behavior is just re-inject packet into kernel. If the packet is captured at FWD hook, it will never passes kernel routing function (See Figure 3 Netfilter Architecture). We rewrite ip_queue so that after any packet captured at FWD hook and re-inject into kernel with NF_ACCEPT, it must pass kernel route function. By doing so, the packet can be routed correctly by newly established route.

When any packet arrives at three registered hooks, it will be checked if its outgoing device is dummy network interface. If it is, this packet will be treated as the one with invalid route and queued. This is achieved by checking a kernel data structure sk_buff, which includes not only IP packet, but also low layer information, such as Ethernet frame data if the network is Ethernet.

• RTM

RTM is not a KLM module, but just a separate function for recording and retrieving time stamp in proc file system.

All the packets will pass POST hook and from there they leave the network layer. This also means all these packets having valid route. We check every packet’s destination, which is valid at this point, so that we can have a table that includes all used routes and their latest valid time. A time stamp of a valid route is required by reactive routing protocol because a route must be deleted if it is not used for a certain time.

4.2.2.Connector

The connector between user and kernel space is consisted by three parts: Netlink, ioctl system call and proc file system.

Netlink passes asynchronous message between user and kernel space as stated before. The message includes IP packet.

Ioctl is the traditional way for user accessing kernel space. Now many route management tools are written using Netlink because there exists Netlink route library. We use ioctl for managing kernel route table at the moment, but will change to Netlink in the future.

Proc file system likes a window to the kernel. User can access it just as normal file system. Through proc file system, user can easily monitor and even change kernel data structure and service configuration as simply as read and write a file. RTM write time stamp of valid route into /proc/aodv/route.

4.2.3.Controller

Controller waits for messages either from user space or kernel space.

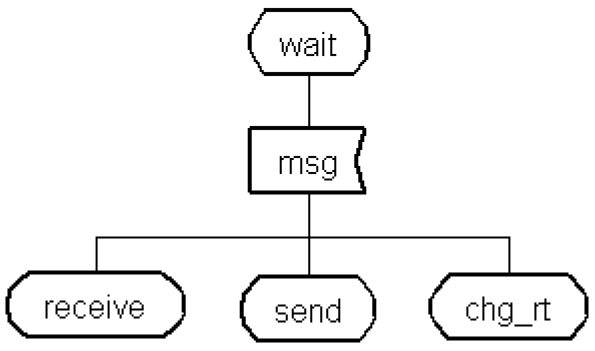

Figure 8 wait state

If the message is from routing daemon, e.g. from user space, CP will first de-queue packet from queue, then send rt_reply to FE, which is in the kernel.

If the message type is FOUND, CP will call chg_krt function to update kernel route table before packet is passed to kernel.

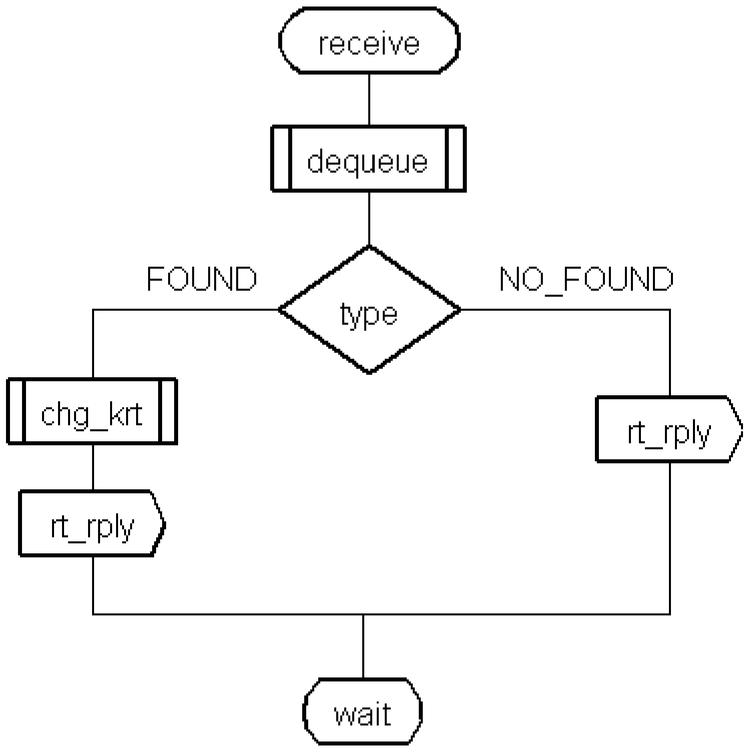

Figure 9 receive_msg

If the message is from FE, e.g. from kernel space, CP will first extract the IP packet from message and insert it into the queue. Then it sends ask_rt to routing daemon and waits for reply.

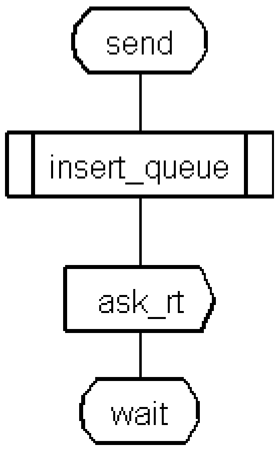

Figure 10 send_request

Controller also receives message from routing daemon about managing kernel routing table. Depending on message type, CP could update/insert/delete kernel routing table.

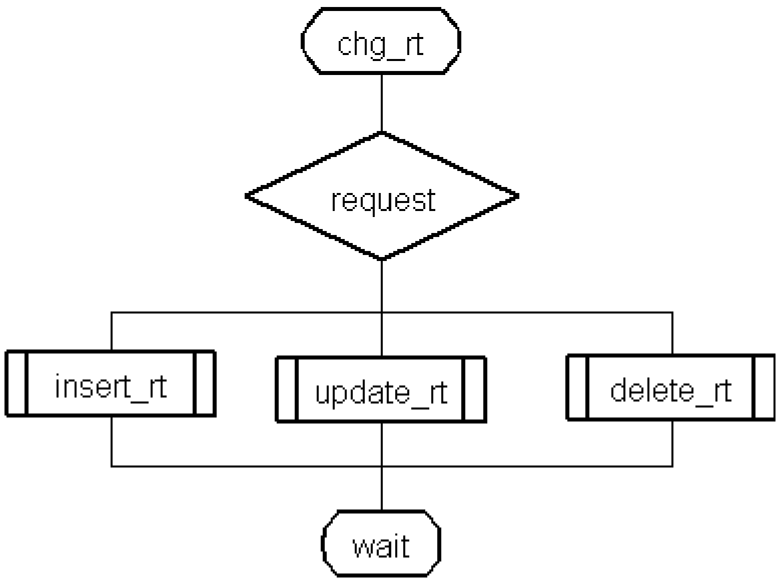

Figure 11 kernel_rt_management

Controller is implemented as a library which exports an API so that routing daemon can use ODRM functionalities.

- libipq

Libipq is just a wrapper function for Netlink in user space. It makes user feel easy and comfortable to use Netlink. With this function, Ipv6 can be also added easily.
After libipq is initialized at first time, it starts a socket for listening request from Netlink, e.g. from kernel space. This socket is used passing message between user and kernel space. Libipq also initializes a queue data structure for queuing packet from kernel space. Netfilter has an elegant implementation to effectively pass packet between user and kernel space. It creates a queue in kernel space for storing all packets and at the same time pass packet with NF_QUEUE into user space. This packet has its own id for identification. In user space, another queue must set up to keep this packet. After certain of time, the packet will be passed back to kernel space. If the packet will be dropped or accepted without any change, only packet id is passed back into kernel. This will save lots of processing time since packet id is very small compared with the whole packet. Only when the packet is changed in user space and accepted by the kernel, the whole packet is needed to pass back to kernel. In most cases, packet doesn’t have any change.

• API

API provided by ODRM is the only interface that routing daemon can use. AS we keep the interface always the same, we can change internal implementation while routing daemon does not know and will not be effected.

```
int start_odrm(void)
```
All the components are initialized properly and ready to serve the request from routing daemon. The return is a socket, which is used by routing daemon for listening and passing route request/reply.

```
void stop_odrm()
```
All the components and data structures are properly released.

```
int wait_rt_request(int fd, struct rt_info* rt)
```
Route daemon continues reading the socket (fd) until there is a request from ODRM.
Route daemon reads the information of route request (rt_info) and tries to find a valid route for it.

```
int reply_rt_request(struct rt_info* rt, int status)
```
Route daemon informs the ODRM the result of route request, which is in the status parameter and is either FOUND or NOT_FOUND. The route information is in rt_info parameter. ODRM will accept or drop the packet according to status.

```
extern int k_add_rte(u_int32_t dest,
                      u_int32_t gw,
                      u_int32_t nm,
                     short int hcnt,
                     char *dev);
```
Add kernel route table

```
extern int k_del_rte(u_int32_t dest,
                     u_int32_t gw,
                     u_int32_t nm,
                     short int hcnt,
                     char *dev);
```
Delete kernel route table

```
extern int k_update_rte(u_int32_t dest,
                        u_int32_t gw,
                        u_int32_t nm,
                        short int hcnt,
                        char *dev);
```

```
int get_rtime(u_int32_t dest)
```
Route daemon gets the time stamp of route, so that it can determine if it is valid or not.

Put these all together, we show here the time sequence of how ODRM works:

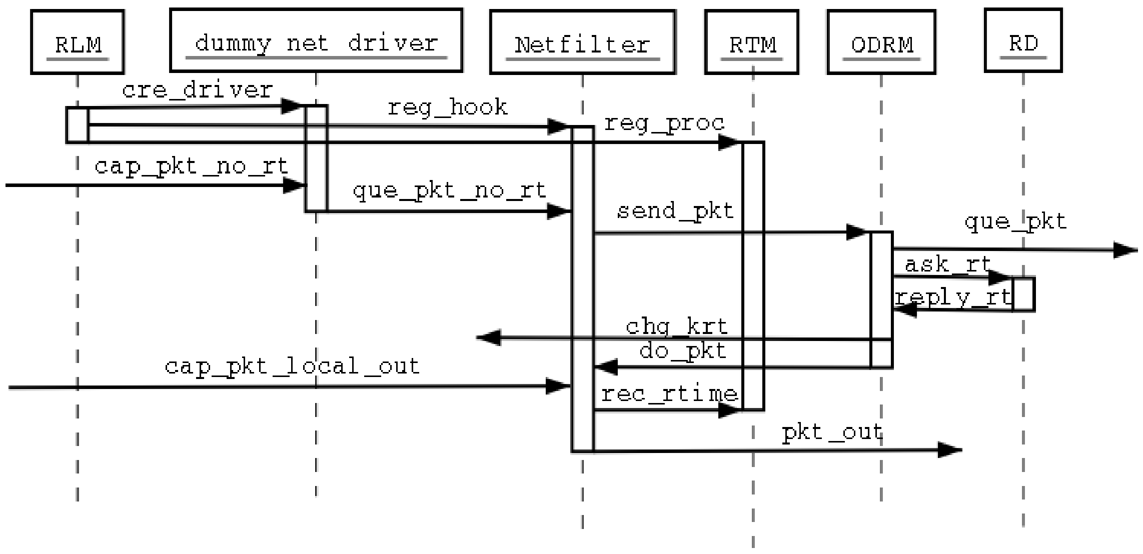

Figure 12 ODRM Time Sequence

ODRM actives RLM and RTM and provides functionalities to routing daemon.
RLM first creates dummy net driver and registers Netfilter with 3 hooks: NF_IP_FORWARD, NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING.

The packet flows like this:

1. Dummy net driver captures packet without routing.
2. Netfilter queues this packet in kernel and sends it to ODRM (by Netlink)
3. ODRM puts this packet into queue in user space.
4. ODRM ask route information about this packet from Routing Daemon
5. Routing Daemon reply with route information
6. ODRM changes kernel routing information based on route information from Routing Daemon
7. ODRM sends back the queued packet to kernel (by Netlink)
8. Before packet leaves for destination, Netfilter gets it and informs RTM to record its time stamp in proc file
9. Finally the packet goes to destination

4.3. AODV Module

This section describes the module that implements the specific routing algorithm – AODV.
We select AODV because it is the most widely implemented reactive algorithm and might become IETF standard.

AODV daemon is based on Uppsala University implementation. We choose it because it has been improved a lot from the first version and now is the sixth version. One of the reasons for porting existing implementations is to test and verify ODRM and to see how well this framework fits to any reactive routing protocol implementation. With some efforts, it is proven that ODRM supports AODV implementation quite well.

AODV consist of two states: on_boot_wait state and active state.

4.3.1. on_boo_wait

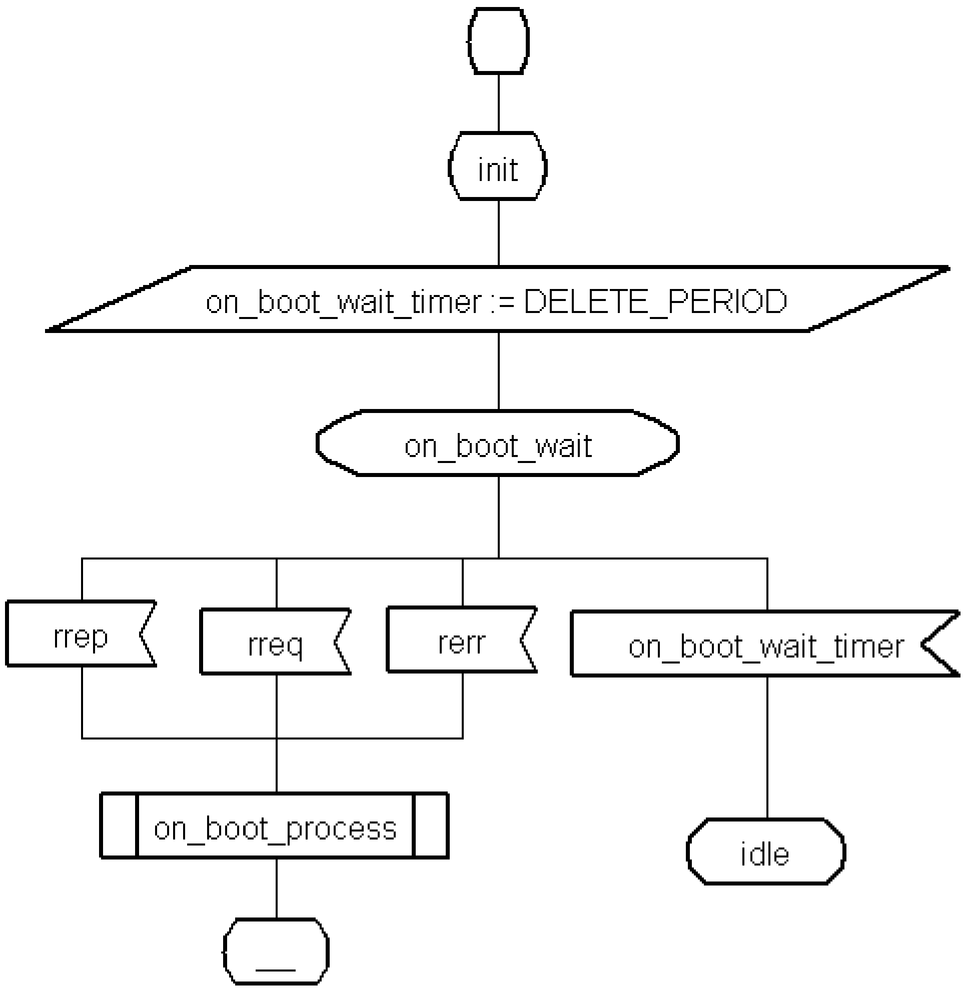

Figure 13 AODV on_boot_wait

On_boot_wait state is the first state that AODV will enter after booting. From AODV draft we know that a node within the ad hoc network must take certain action after reboot. When a node enters this state, it may lose all sequence records for all destinations, include its own sequence number. But the neighbor nodes might still use this node as active next hop. This can potentially create routing loops.

To prevent this happens, a node stay in on_boot_wait state for DELETE_PERIOD after it starts up. During this period, it doesn't transmit any RREP messages. If it receives any RREQ, RREP, or RERR control message, it should create corresponding route entry, but never forward these control messages. If it receives any data packet from other nodes, it should broadcast RERR message.

Just after a node boots, aodv first creates several global data structures. These data structures contain all the states of this node. These states will be updated continually during the life of the node.

- rt_table: includes route information of this node. It is implemented as a linked list hash table, which is used to avoid collapse, as shown below:

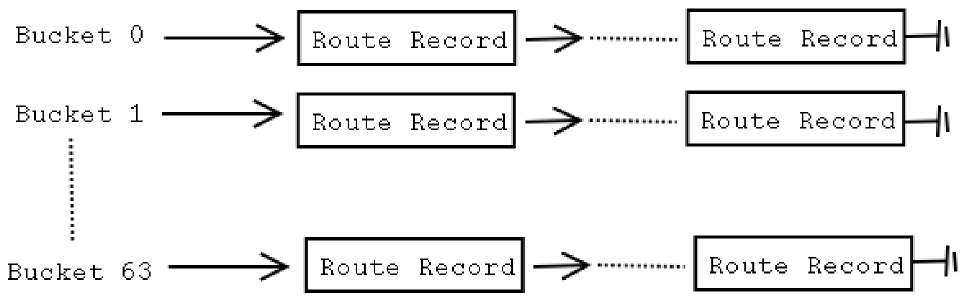

Figure 14 Hash table for AODV routing table

- timer_queue: includes all events which will be active in a certain time. This data structure is very important because it guarantee AODV behave correctly.
- host_info: include node itself information, such as IP address, network interface, etc.
- seek_list: includes nodes which have asked for route and are waiting for the route reply.
- rreq_record_queue: includes all RREQs that have been processed and are still valid.
- blacklist: includes nodes which do not reply RREP_ACK.

After setting up data structures, aodv also starts up some services, such as changing some kernel configurations. Most of them are done by system() function which is not a good implementation because each Linux distribution has different file structure. The command which system() is called will not be in the same directory all the time. For the future porting to other Linux distributions, or even to Unix, we should write a script to manage all the configurations.
At last, aodv creates and starts hello timer and on_boot_wait timer.

4.3.2. active

Active state is another state on which AODV is most working. After a node comes out from on_boot_wait state, it must enter active state and stay there forever if it does not stop. This is achieved by an endless loop. In the loop, the node checks control messages. If it receives any of them, aodv starts corresponding action to process this message.

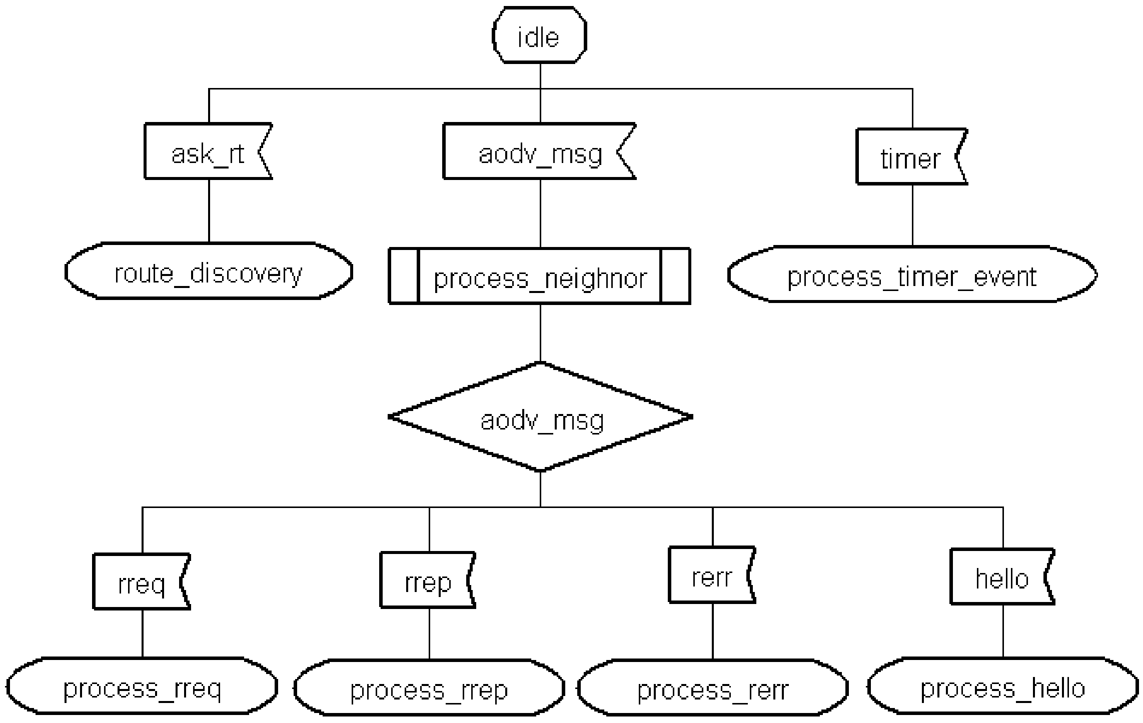

Figure 15 AODV active

There is an implementation issue about how to listen different messages, which can occur at any time, and how to react to it efficiently. The common implementation is using thread. Each thread is listening to one socket. But there are some subtle issues about thread, such as deadlock, synchronization. We must be careful about it.

In Uppsala implementation, select() function [21] is used. Select() can listen to several sockets at the same and respond to any of them quickly and efficiently. It is one of the elegant Unix technologies and used quite widely in the situation where the number of sockets to listen is less than 10. Some very tricky things, which could be difficult to solve by thread, is very easy to avoid by using select() function. Furthermore, select() can be used as a timer and is accurate to millisecond.

In the first version of Uppsala implementation, timer_queue is controlled by alarm signal [22]. When any alarm is due, the corresponding event pops from the queue and executes. At same

time, alarm signal must be rearranged to a new timeout. This implementation is awful and buggy. Actually, timer_queue is the most trouble source of first three versions implementation.
By using select(), we eliminate the need of alarm signal because select() itself can also be used as a timer with accurateness to millisecond, which is enough for AODV. Now the timer_queue is very neat and very stable. UCSB implementation uses the same technology.

To port Uppsala AODV for using functionalities provided by our framework, we let AODV initialize ODRM services by perform start_odrm() call in on_boot_wait state. Then we add a socket into select() loop. This socket is from ODRM which will provide route request from kernel and pass back route reply. After this, we remove all the functions for changing kernel routing behavior. These functionalities are provided by ODRM. At last, we change original route reply function by the one that is provided by ODRM.
Next we analyze how these are achieved.
If the received message is ask_rt, which is from ODRM, AODV daemon goes into route_discovery state.

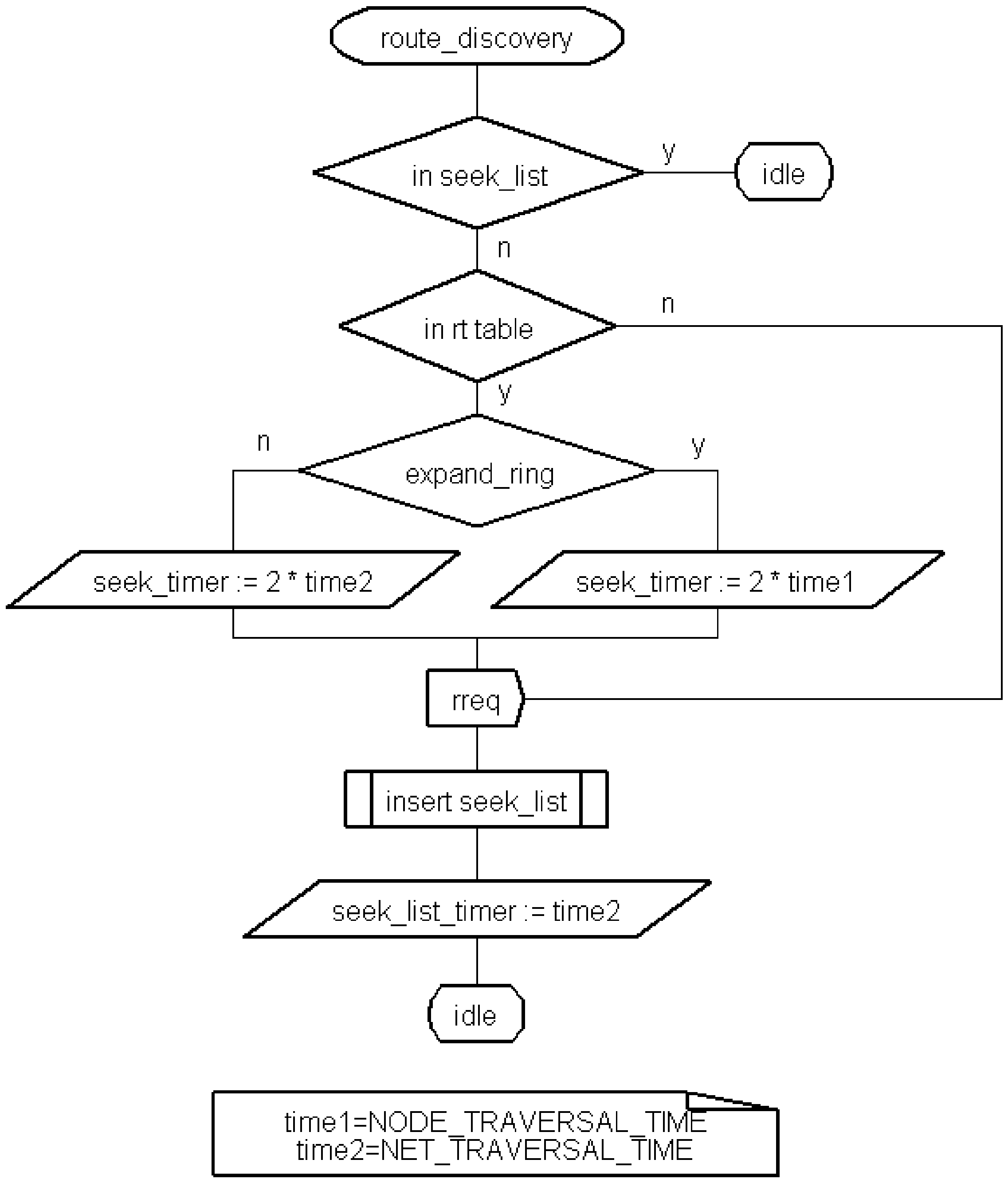

Figure 16 route_discovery

In this state, aodv first checks if the route request is already in process. If it is, aodv returns to idle state at once and waiting for other messages. Otherwise it sets the seeking time for this request and broadcasts RREQ.

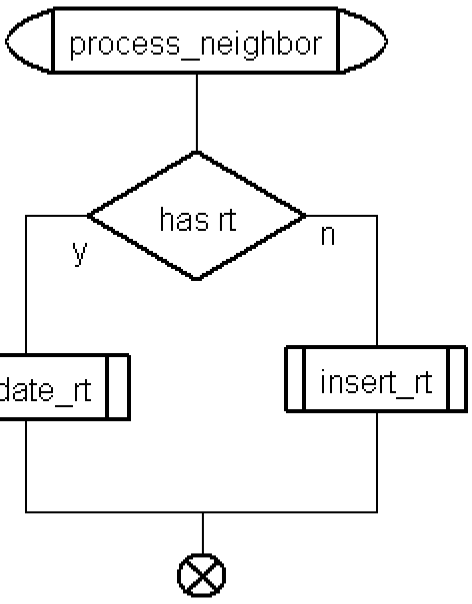

Figure 17 process_neighbor

If the message is AODV control message, aodv will update route table according to the information from the control message.

During the updating and inserting route table, aodv will set up new timer for the route entry and check if there has any suspended route request for this entry. If there has, aodv sends out route reply with FOUND flag to kernel. Those queued packets for this route destination will be de-queued and injected into the kernel.

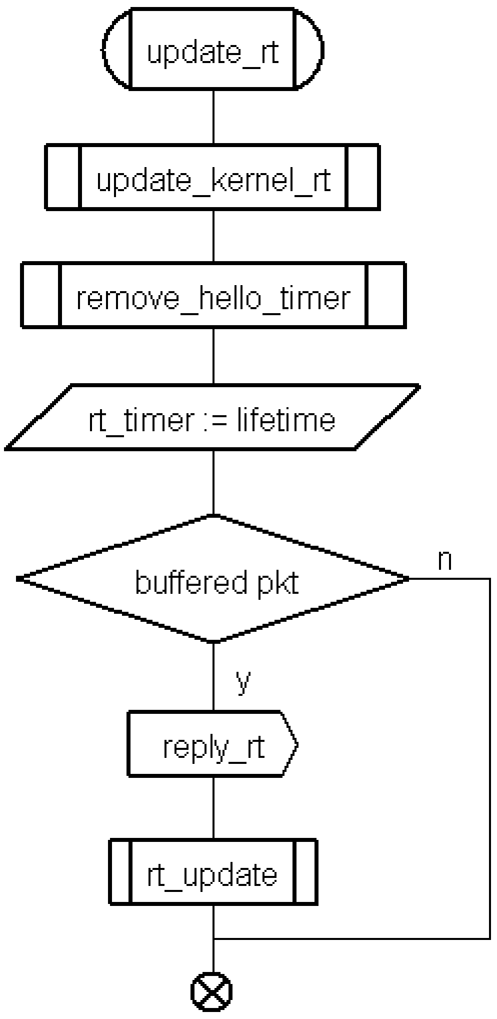

Figure 18 update_rt

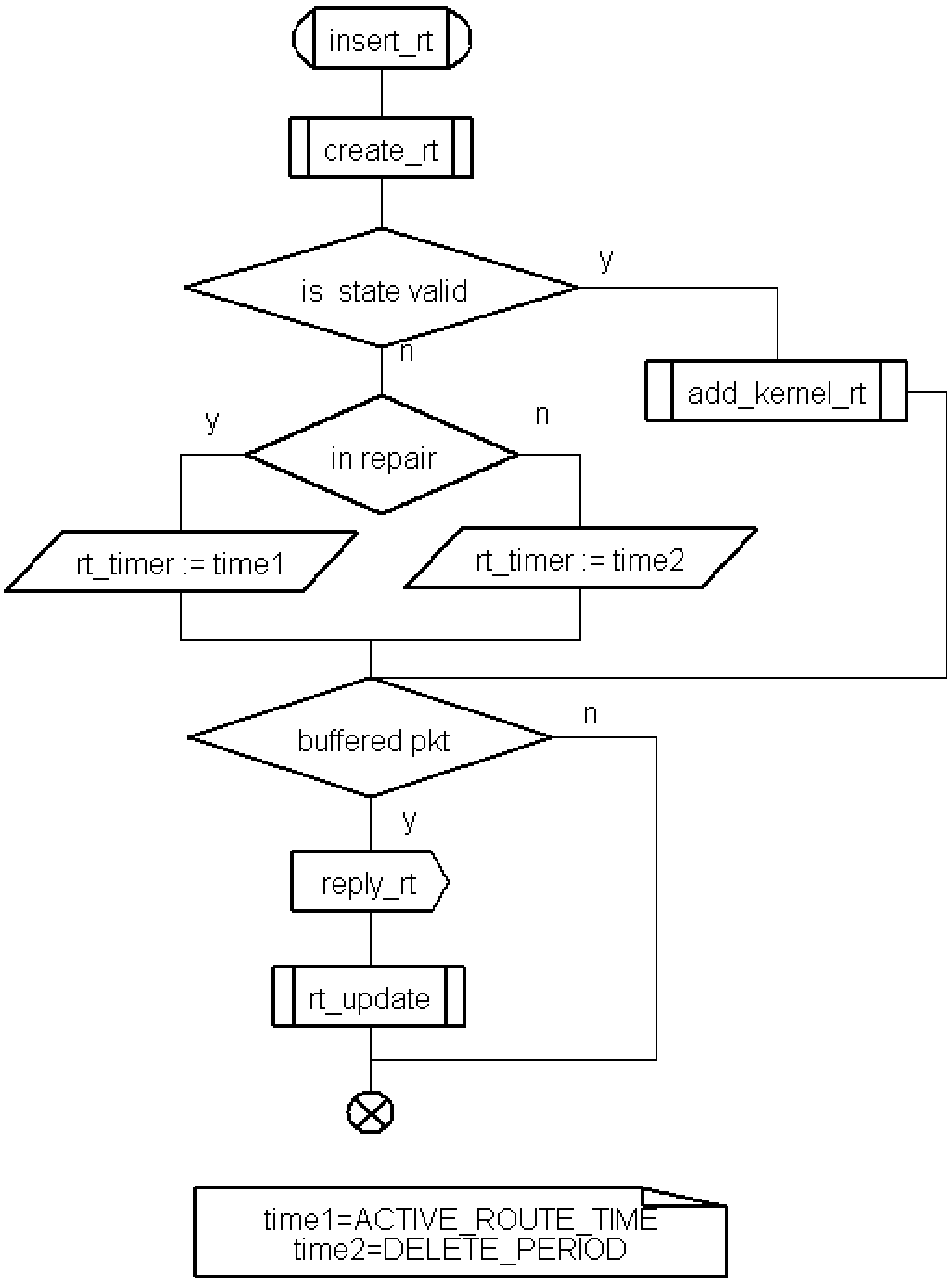

Figure 19 insert_rt

Then according to the message type, aodv will start corresponding process. These messages are: RREQ, RREP, RERR, and HELLO. HELLO is a special RREP message and requires separated processing.

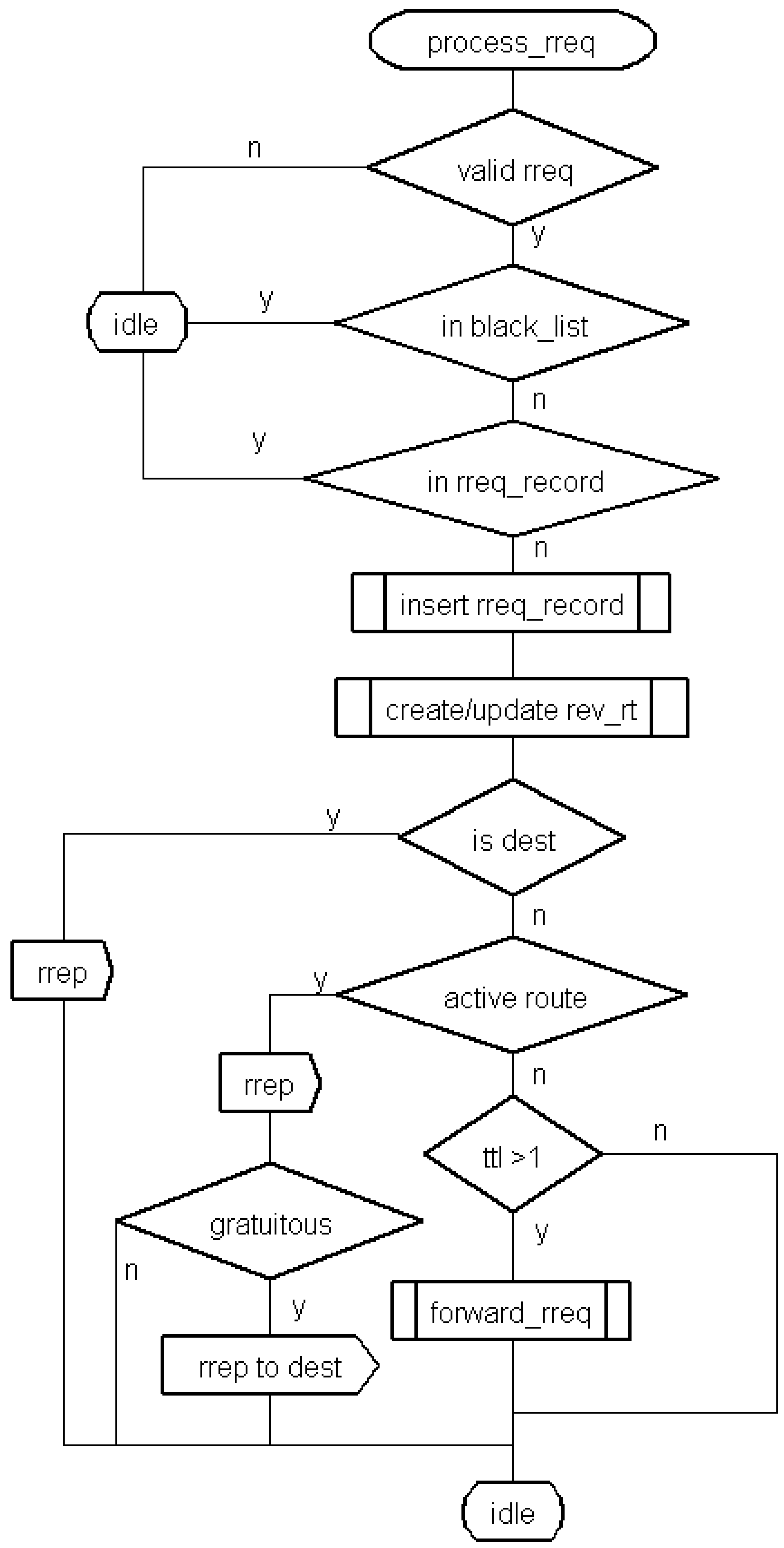
process_rreq
valid rreq
n
y
idle
y
in black_list
n
y
in rreq_record
n
insert rreq_record
create/update rev_rt
y
is dest
n
rrep
y
active route
rrep
n
n
ttl >1
gratuitous
y
n
y
forward_rreq
rrep to dest
idle

Figure 20 process_rreq

When the message is a valid RREQ and not in rreq_record and blacklist, aodv will insert it into rreq_record for avoiding processing duplicated RREQ.

Then aodv updates reverse route table. If the node is the destination of RREQ or it has active route to the destination, it should generate RREP. Otherwise, aodv forwards RREQ. After sending RREP, aodv also sends RREP to the destination whose RREQ's GRATUITOUS flag is set.

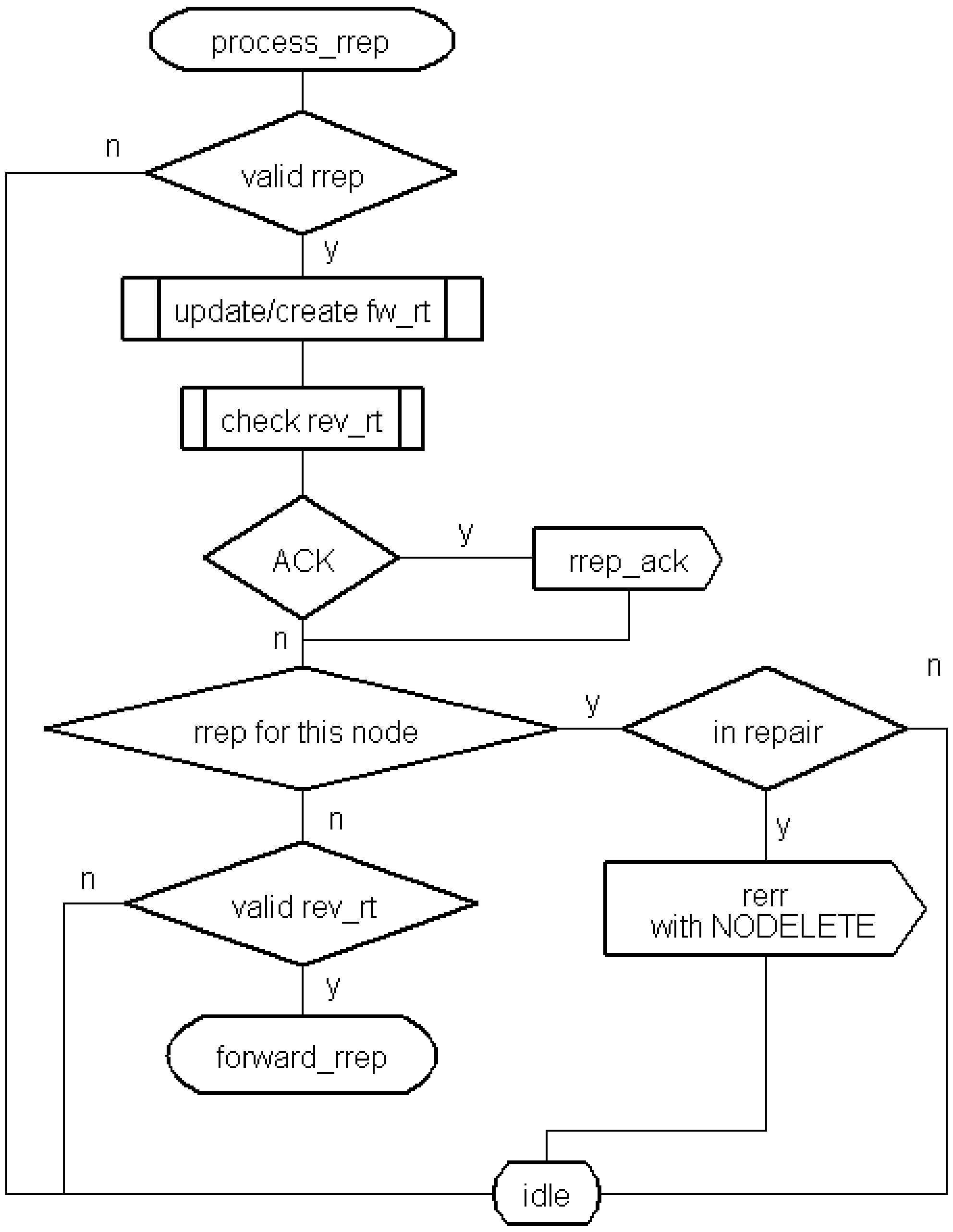

Figure 21 process_rrep

When the message is a valid RREP, aodv will first update corresponding forward route and reverse route. Then aodv sends RREP_ACK if RREP's ACK flag is set. Afterwards, aodv will either forward or accept RREP depending on the RREP destination and valid route entry.

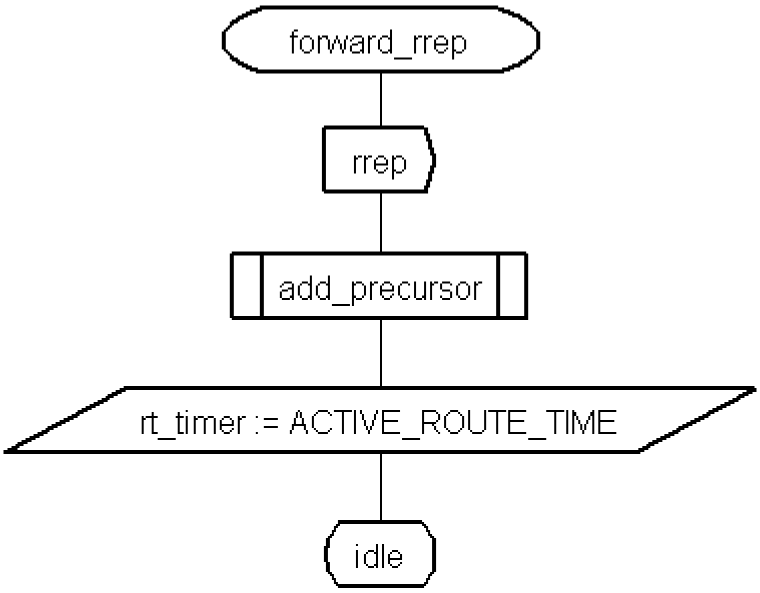

Figure 22 forward_rrep

When aodv is forwarding a RREP message, it must update precursor list based on the information from RREP. By doing so, a reverse route is set up along the path the RREP forwarding.

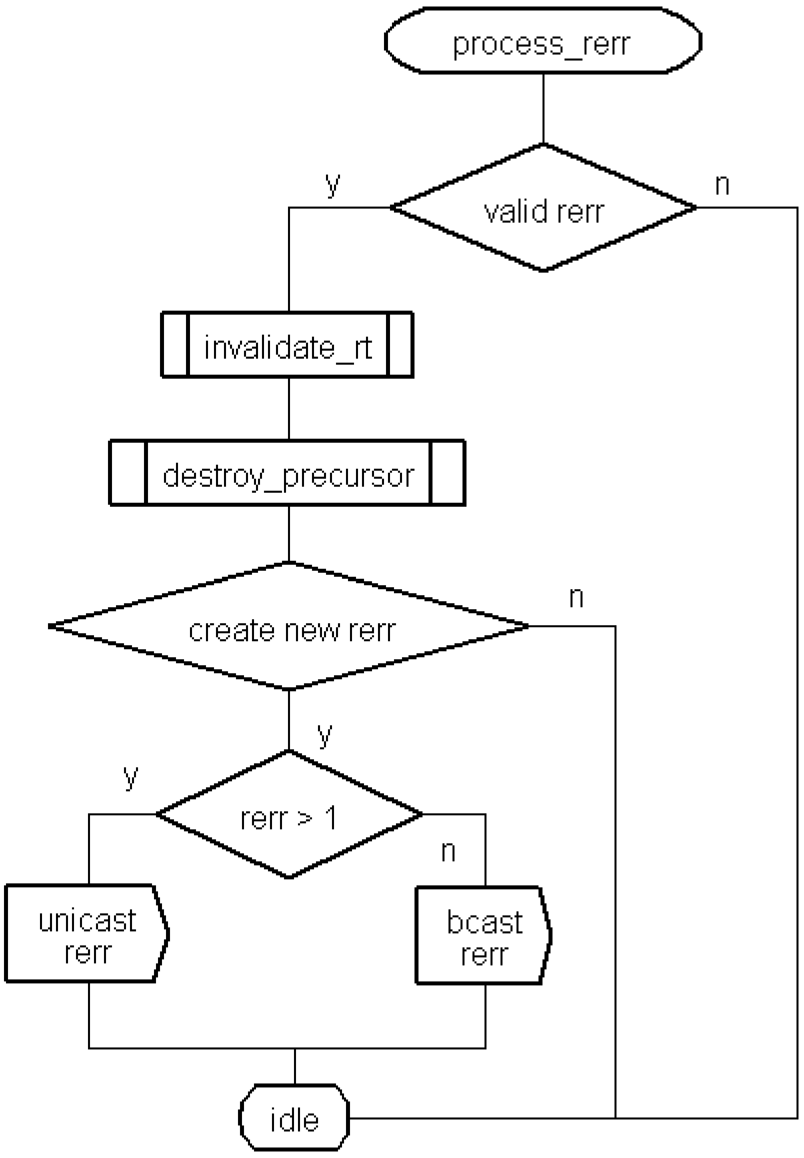

Figure 23 process_rerr

When the message is a valid RERR, aodv invalidates corresponding route entry and destroys precursor list. During destroying precursor list, aodv might create new RERR. At last, aodv sends out RERR, either by unicasting or broadcasting.

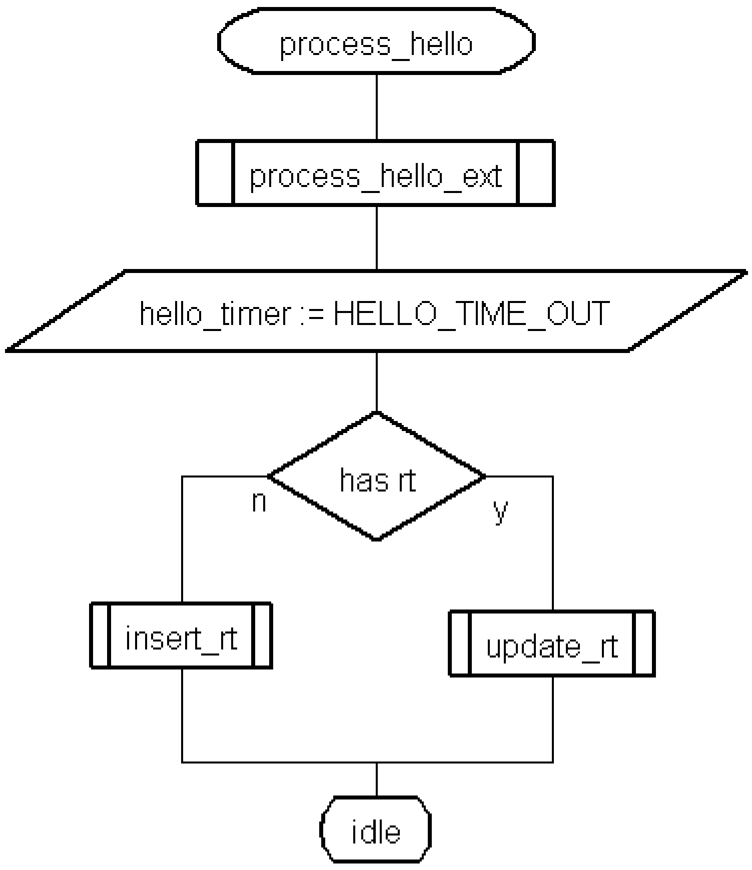

Figure 24 process_hello

When the message is a valid hello, aodv update the route table and setup new timer for next hello.

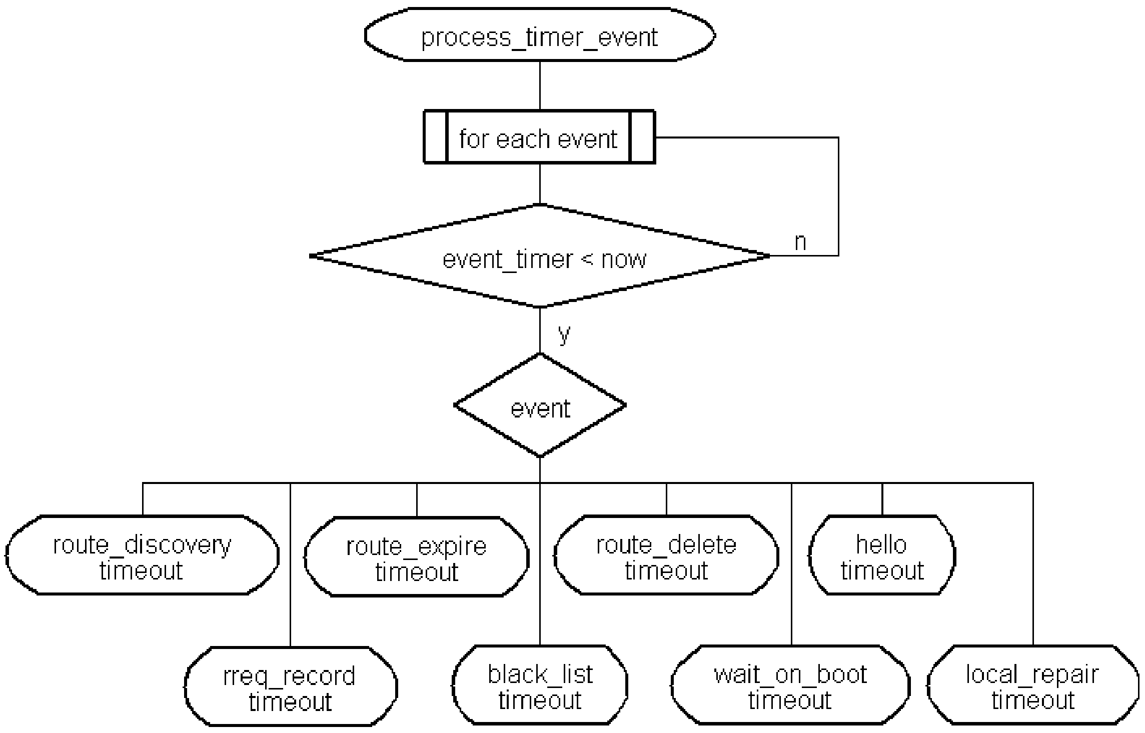

Figure 25 process_timer_event

Another kind of message is timer expiration. It could happen at any time. When aodv receives it, aodv starts to process corresponding timer event and update or insert new timer.

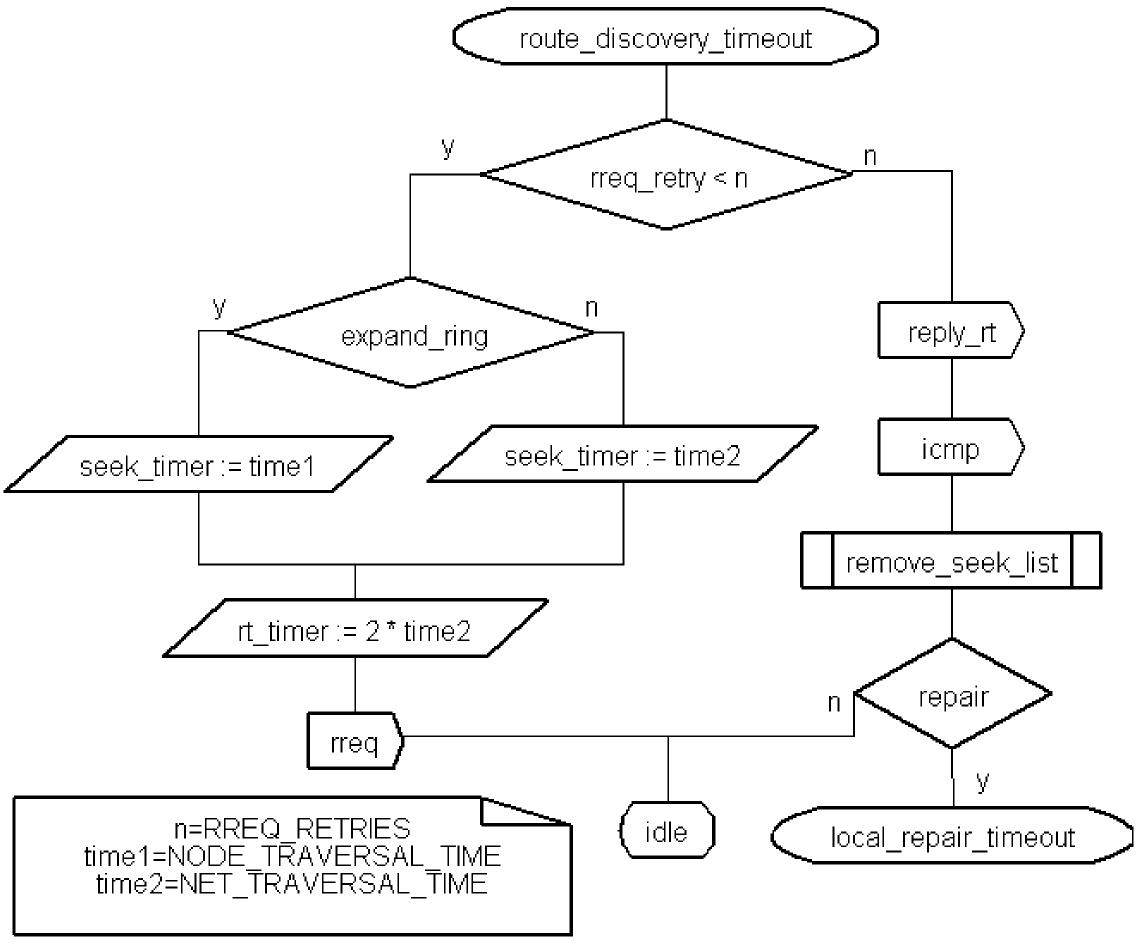

Figure 26 route_discovery_timeout

When route_discovery expires, aodv sends route reply with NO_FOUND flag to kernel if this route has exceeded RREQ_RETTIES limit. When kernel receives this message, it discards all queued packets for this route destination. At the same time, aodv removes seek entry for this route.

If this route request is still within the limit, aodv updates seek entry timer for another seeking and sends out RREQ.

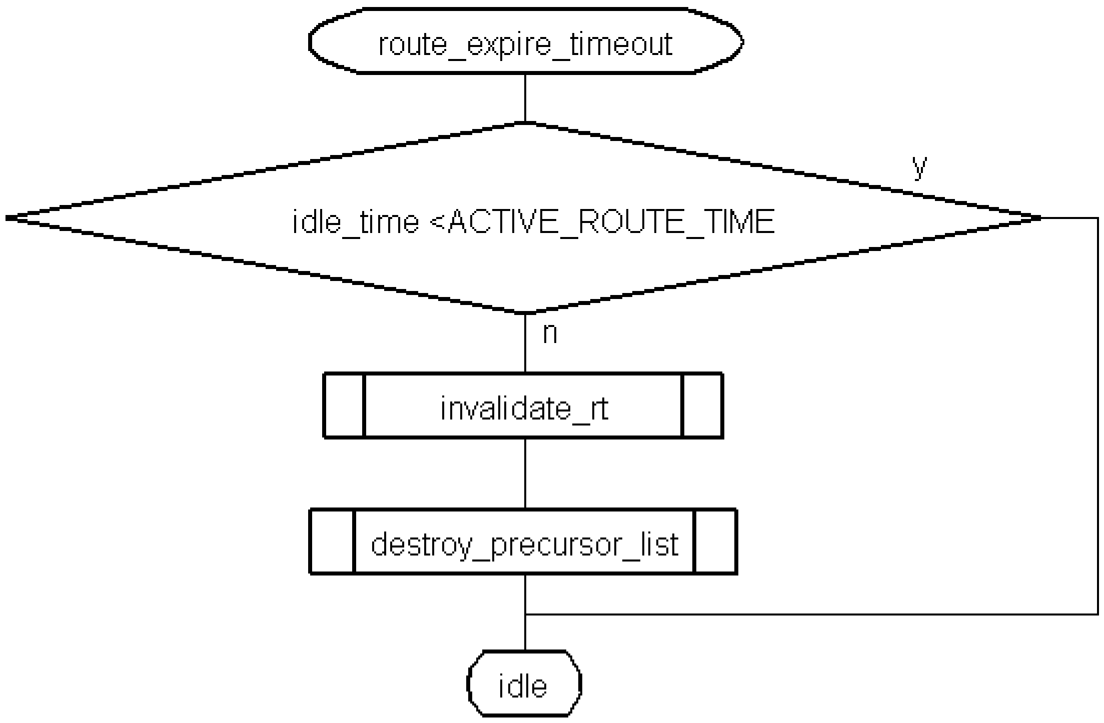

Figure 27 route_expire_timeout

When route_expire expires, aodv will mark this route as invalid if it is idle during ACTIVE_ROUTE_TIME. All the precursors for it will be removed at the same time.

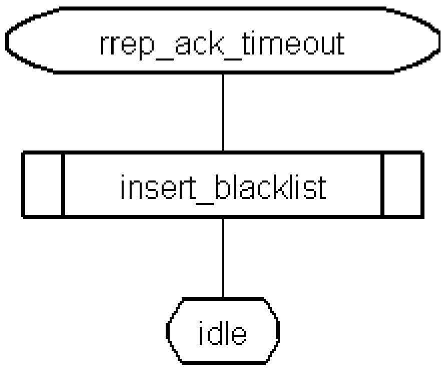

Figure 28 rrep_ack_timeout

After a node sends RREP, it may require other nodes, which receive RREP, to send back RREP_ACK. When rrep_ack expires, aodv will put those nodes into blacklist because it doesn't receive RREP_ACK

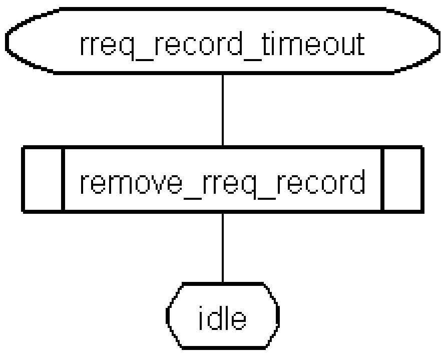

Figure 29 rreq_record_timeout

When a node receives a RREQ, it will put RREQ into the rreq_record queue and set up timer for it. The node will not process the same RREQ during the time. After expiration, this RREQ record will be deleted so that aodv could process it in case there have more.

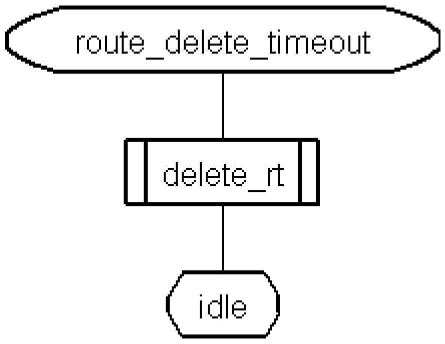

Figure 30 route_delete_timeout

When route_expire expires, the kernel route entry is deleted. But route entries are just marked as invalid. They are really deleted when route_delete timer expires.

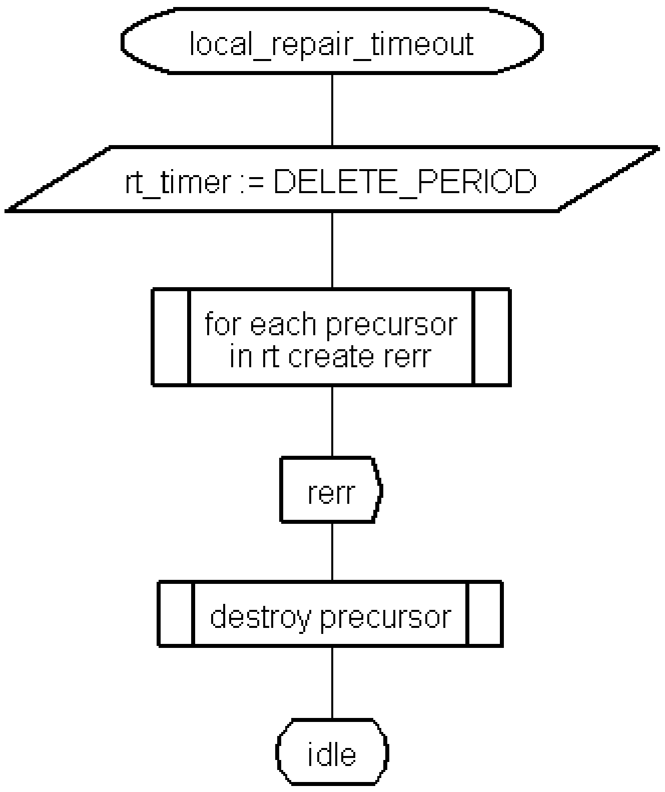
local_repair_timeout
rt_timer := DELETE_PERIOD
for each precursor in rt create rerr
rerr
destroy precursor
idle

Figure 31 local_repair_timout

When local_repair timer expires, aodv generates and sends RERR message for all destinations in precursor list and destroy precursor list. It also sets the route timer to DELETE_PERIOD so that this route will be deleted after timeout.

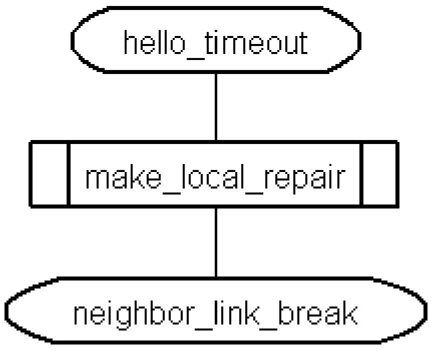

Figure 32 hello_timeout

If hello timer expires, there must be neighbor link break. Aodv tries to make local repair for the break and process neighbor_link_break.

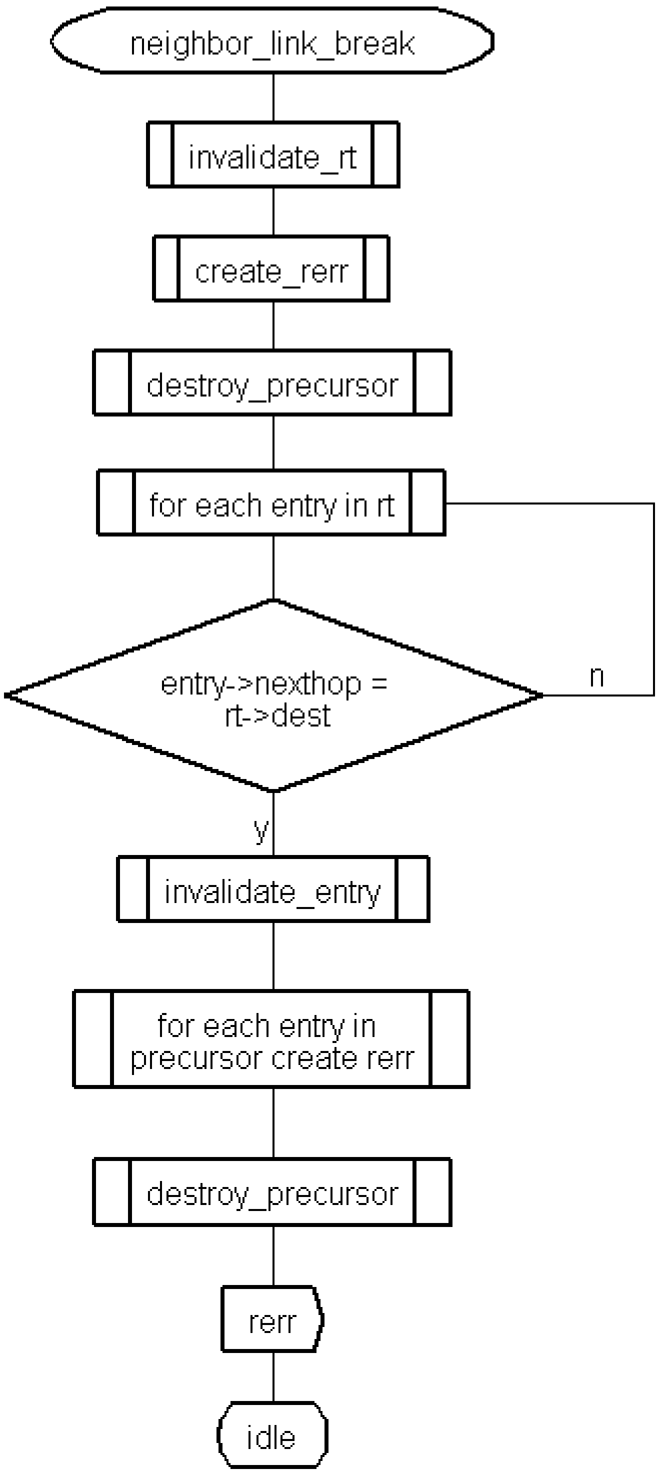

Figure 33 neighbor_link_break

When neighbor link break happens, aodv must go through all the entries in route table and corresponding precursor list. It first invalidates route entries and destroy precursor list and create RERR for it. Then aodv checks routing table for entries, which have the unreachable destination as next hop. These entries must be invalidated and their precursor list must be destroyed and corresponding RERR generated. At last, aodv sends out RERR.

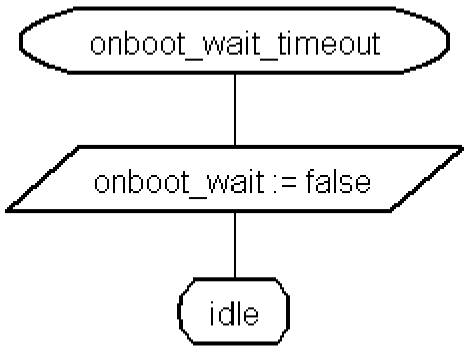

Figure 34 onboot_wait_timeout

When onboot_wait timer expires, on_boot_wait state is terminated and aodv goes into the active state.

Chapter 5

5. Tests and analysis

Chapter 5 deals with the evaluation of the Ad Hoc Framework. Tests are planned and documented to obtain a record of the analysis. The aim of the tests is to analyze how the AODV protocol works when it uses functionalities provided by ODRM module. First we describe how to integrate framework into iPAQ. Then we perform several test cases in different scenario. Finally the result is analyzed.

5.1. Framework integration

ODRM needs a solution to export functionality to routing module. There are several ways to achieve the goal. One approach is to implement ODRM as a library, which export an API; the other is to implement it as a separate daemon, which communicates with the routing daemon by some inter-process communication methods, such as shared memory or socket.

We choose the first approach. Calling library's API is more natural way to access the functionality from routing daemon's point of view. Further more, library call is more efficient because it doesn't invoke the overhead of inter-process communication.

There are two kinds of libraries [18] in Linux: static library (or archive) and dynamically linked library (or shared library). Static library makes program bigger and harder to upgrade, but easier to deploy, while share library makes program smaller, easier to update, but harder to deploy. We use archive for the library, but it is not difficult to change to shared library.

We use iPAQ as mobile node. The original operating system in iPAQ is PocketPC 2002. We have to change it to Linux to integrate the framework. Linux supports ARM architecture, which is used by iPAQ. Also there has ported Linux for iPAQ. This makes integration quite easy. Yet there are some problems when cross compile ARM executable in RedHat host, it is not so difficult to solve them.

The Linux portable on iPAQ is called Familiar [19]. It is a tailored Linux version to fit into the limited resource. Because of very little space for file system, we cannot install Linux kernel source and compile tool chain into iPAQ. This means we cannot build the native executable on iPAQ directly. The default familiar kernel doesn’t include Netfilter, but we need it for the framework. For those reasons, we have to compile everything in Linux PC to get ARM executable. Then we transfer it to iPAQ and run it. This can be achieved by using GNU/gcc compile tool chain for ARM.

5.2. Test method and result

The test is performed in networking lab. We put four iPAQs in four different places so that each one is only within one iPAQ’s wireless signal range. Because the complex building structure and lots of other interferences near 2.4G signal band, such as cordless phone and WLAN network, we have to select these four places by experimenting. They are shown below:

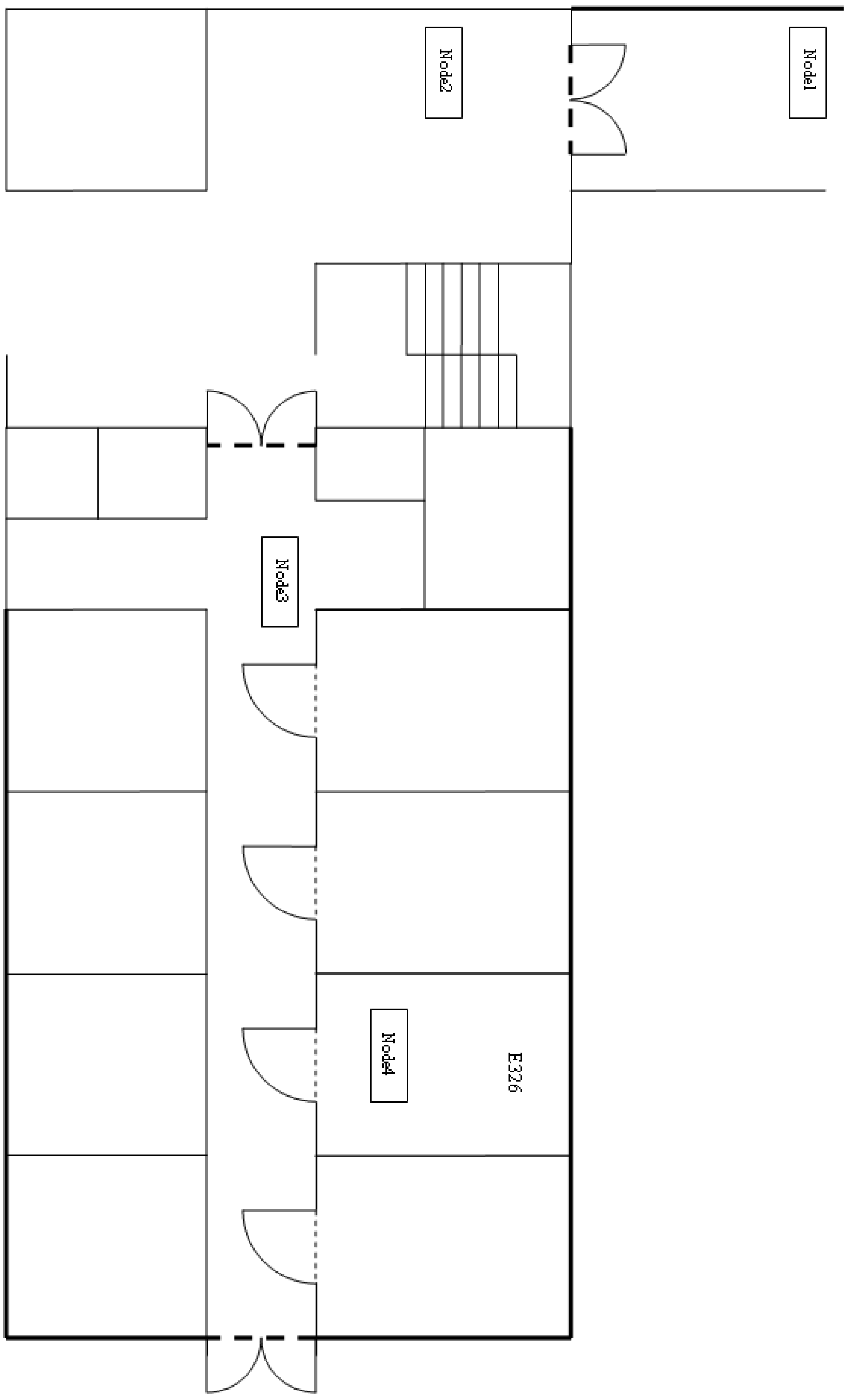

Figure 35 Test Environment

The doors between node 2 and node3, node 2 and node1 are made of iron. They have to be closed all the time. All nodes are within the same floor so the distance between them is the same as shown in Figure 35.

We use simple ping as the test method. The time interval of each ping command is one second. The packet size is 64 bytes. Ping command is issued from node1 to reach node4. We hope node1 should reach node4 in three hops using node2 and node3. The route log file of node1 shows the result. It is the same as we expected.

```
# Time: 02:53:49.461 IP: 10.0.0.1, seqno: 1 entries/active: 2/2
Destination     Next hop        HC  St. Seqno Expire Flags Iface Precursors
10.0.0.2        10.0.0.2        1   VAL 39    2440         eth0
10.0.0.4        10.0.0.2        3   VAL 15    1478         eth0
```

Node1 can only reach node2. To reach node4, it uses node2 as next hop and has three hopes.

The first route request from node1 to node4 can be seen from odrm.log as following:

```
02:53:48.756 pkt_q_insert: buffered pkt to 10.0.0.4
```

Then node1 issues RREQ as shown in aodv.log:

```
02:53:48.784 rreq_create: Assembled RREQ 10.0.0.4
```

After a while, node1 receives RREP from node2:

```
02:53:48.907 rrep_process: from 10.0.0.2 about 10.0.0.1->10.0.0.4
02:53:48.914 rt_table_insert: Inserting 10.0.0.4 (bucket 4) next hop 10.0.0.2
02:53:48.921 rt_table_insert: New timer for 10.0.0.4, life=2012
02:53:48.931 rt_table_insert: ROUTE FOUND for 10.0.0.4
```

The time of setting up route from node1 to node4 is about 170 ms. This is the ideal situation.

When there are RERRs, the situation is more complex.

```
02:53:53.726 aodv_socket_process_packet: Received RERR
02:53:53.733 rerr_process: ip_src=10.0.0.2
02:53:53.740 rerr_process: unreachable dest=10.0.0.4 seqno=16
02:53:53.754 rerr_process: removing rte 10.0.0.4 - WAS IN RERR!!
02:53:53.772 rt_table_invalidate: 10.0.0.4 removed in 15000 msecs
```

RERR is received at `02:53:53.726` and route for 10.0.0.4 is removed.

Then there is a route request form node1 to node4 so a RREQ for 10.0.0.4 is created:

```
02:53:53.816 rreq_create: Assembled RREQ 10.0.0.4
```

After a long time, node1 finally receives RREP from node2:

```
02:53:57.785 aodv_socket_process_packet: Received RREP
02:53:57.794 rrep_process: from 10.0.0.2 about 10.0.0.1->10.0.0.4
```

The time for route setup is nearly 4000 ms. During this period, several RREQs are created. Because we implemented binary exponential backoff algorithms for RREQ timeout, the number of RREQs is reduced. But we have to wait longer for route setup.

From the aodv.log, last RREQ is created at:

```
02:53:57.650 route_discovery_timeout: Seeking 10.0.0.4 ttl=70 wait=5600
```

Its timeout is 5600 ms.

After studying the log file of other nodes, we found out there are lots of neighbor link breaks between node3 and node4. Node3 sends out RERR and node2 and nod1 receive it and purge their route entries. We didn’t show other nodes’ log file because each node has its own system time. It is not feasible to see exactly when a node has neighbor link break and other nodes receive this error.

Node3 and node4 are within the same corridor and their distant is not long. We tested on purpose to find out when the neighbor link break happened. It is shown that when node3 is closer to the iron door, there are lots of neighbor link break errors. It is clear that the iron door reflects the wireless signals, which interfere the original signals.

From [20], broadcast can reach further than unicast signal. Because HELLO message is broadcasted, the ping packet may not reach the other node even if this node is considered as neighbor.

We perform another test that fixes node4 and node3 to make sure they can have stable neighbor link. Then node1 is moving at walking speed. If only node1 is within node2’s signal rang, the ping between node1 and node4 goes well without dropping packets. We also performed other test cases with different aodv options [see Appendix A]. But the results are not always the same as we expected. It is quite difficult to repeat the same test case to get the same result. There are lots of factors effecting AODV behaviors in real running environment. Some results of these test cases are shown below:

	Test 1 [2]	Test 2 [3]	Test 3 [4]
Test period	3 min	5 min 20 sec	4 min
Number of ping	175	316	220
Number of lost packets	50	2	55
Lost packet percentage	28%	2%	25%
Average round trip time	40ms	22ms	35ms
Number of RERR	14	2	10

Table 8 Test result

5.3. Result analysis

With four nodes, our implementation works fine. It is about 10ms for kernel capturing packet without route and sending it to user space and asking for route. This is acceptable comparing with the AODV route request/reply cycle, which is about hundreds of milliseconds.

The overhead of route maintenance is high. When a node receives RERR, it will check all route entries and their precursor lists. If any entry is effected, this entry must be purged and corresponding RERR generated. There could have many RERRs when only one neighbor link breaks.
With LOCAL_REPAIR enabled, the situation would be better.

As [20] shows, HELLO message is not a reliable way to detect neighbor. It is better we could detect the signal strength directly, not depend on IP packet. This means we have to get some helps from data link layer and physical layer, which are lower than IP layer. It is more and more necessary that other network layers have some changes to fit into Ad Hoc network requirement. This will ease implementation a lot.

[2] Test with lots of neighbor link breaks between node3 and node4
[3] Test with fixed node3 and node4 and free moving node1
[4] Test with lots of neighbor link breaks between node3 and node4 and LOCAL_REPAIR enabled

We find there are many other reasons effecting Ad Hoc network besides routing algorithm. Some of them come from physical environment and hardware constrains. The wireless signal range is limited by geography, interference and battery at least.

In networking lab, the range of wireless signal is about ten meters because of many walls and interference from other devices. These devices work near 2.4G band, such as cordless phone, Bluetooth and Aalto wireless network in HUT. When the battery is low, the working range for wireless signal is even shorter.

The direction of antenna also plays a role here. As moving close to the iron door, we notice there are more neighbor link breaks when the antenna of PDA directed to iron door.

Chapter 6

6. Conclusions and future work

In this thesis, we have designed a framework for Ad Hoc routing, which could support both reactive and proactive routing protocol, and implemented reactive routing part for it. Based on the test result, the reactive routing implementation is successful. To achieve the design goal, we separate whole framework into different components. With clearly defined interface for each component, we could integrate them easily. The benefit for component-based design is we could change internal implementation while all components can still work together only if we keep the interfaces the same. Also the complex system can be divided into small modules which are easy to be implemented.

The future work will be to have more tests. Those tests could be data streaming, such as audio, and burst data accessing, such as HTTP request. When we have more nodes, we could perform tests with different network topology.

The kernel route management module is not completely finished since it needs more requirements from proactive routing protocol, e.g. OLSR in our framework. When this information is available, we will add more functionalities into controller component of CP.

After more testing and more bugs fixing, other functionalities may add into the framework. One example is IPv6 support. With our implementation, adding IPv6 is very easy. For example, let's look at ODRM. Dummy network driver doesn't need any changes because it works at data link layer. Netfilter support IPv6 by default; so we just need inform RLM register corresponding hooks, e.g. NF6_IP_FORWARD etc. Then RLM doesn't need more changes. Netlink is also a protocol works lower than IP layer and we just inform ip_queue to register a socket for listening IPv6 packet, then everything is done. The exported interface from ODRM does not change – it is still a socket.

References

[1] MobileMan HUT homepage, http://www.tct.hut.fi/tutkimus/MobileMan/, February 2003

[2] MANET homepage, http://www.ietf.org/html.charters/manet-charter.html, March 2003

[3] Charles E. Perkins, Ad Hoc Networking, Addison-Wesley, December 2000

[4] Zygmunt J. Hass, Marc R. Pearlman and Prince Samar, Zone Routing Protocol (ZRP) for Ad Hoc Networks, *IETF Internet draft* , draft-ietf-manet-zone-zrp-04.txt, July 2002

[5] Charles E. Perkins, Elizabeth M. Belding-Royer and Samir Das, Ad Hoc On Demand Distance Vector (AODV) Routing, *IETF Internet draft*, draft-ietf-manet-aodv-12.txt, November 2002

[6] Charles E. Perkins, Quality of Service for AODV, *IETF Internet draft*, draft-ietf-manet-aodvqos-00.txt, July 2000

[7] Pajeev Koodi and Charles E. Perkins, Service Discovery in On-Demand Ad Hoc Networks, *IETF Internet draft*, draft-ietf-manet-koodi-manet-servicediscovery-00.txt, October 2000

[8] Charles E. Perkins, Elizabeth M. Belding-Royer and Samir Das, Ad Hoc On Demand Distance Vector (AODV) Routing for IP version 6, *IETF Internet draft*, draft-ietf-manet-perkins-aodv6-01.txt, November 200

[9] Uppsala University AODV homepage, http://user.it.uu.se/~henrikl/aodv/, March 2003

[10] UCSB AODV homepage, http://moment.cs.ucsb.edu/AODV/aodv.html

[11] NITS AODV homepage, http://w3.antd.nist.gov/wctg/aodv_kernel/, May 2002

[12] Linux homepage, http://www.linux.org/

[13] GUN's Not UNIX homepage, http://www.gnu.org/, February 2003

[14] Gowri Dhandapani and Anupama Sundaresan, Netlink Sockets Overview, http://qos.ittc.ukans.edu/netlink/html/netlink.html, October 1999

[15] Netfilter homepage, http://www.netfilter.org/

[16] OS kernel design homepage, http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html

[17] Alessandro Rubini and Jonathan Corbet, Linux Device Drivers, 2nd Edition June 2001

[18] Advanced Linux Programming, http://www.advancedlinuxprogramming.com/

[19] Familiar homepage, http://www.handhelds.org, March 2003

[20] H. Lundgren, E. Nordström and C. Tschudin, Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks, WoWMoM 2002.

[21] W. Richard Stevens, UNIX Network Programming, Prentice Hall PTR, 2nd Edition
[22] W. Richard Stevens, Advanced programming in the UNIX environment, Reading (MA) Addison-Wesley, 1992

Appendix A. Aodv daemon configuration:

```
log_level = 6           # 0 = off, default is 3
debug = 0               # use console as log output or not, default is not

rt_log_interval = 1000  # msecs between routing table logging 0 = off
unidir_hack = 0
rreq_gratuitous = 0
expanding_ring_search = 1
internet_gw_mode = 0
local_repair = 0
receive_n_hellos = 0
hello_jittering = 1
optimized_hellos = 0
ratelimit = 1           # Option for rate limiting RREQs and RERRs.

wait_on_reboot = 1
aodv_log = aodv.log            #the location and name of aodv log
aodv_rt_log = aodv_rt.log      #the location and name of aodv route table log

daemon = 1              # startup as daemon or not; default is daemon

#AODV parameters

K = 5
ACTIVE_ROUTE_TIMEOUT = 3000
DELETE_PERIOD = 15000  # K * max(ACTIVE_ROUTE_TIMEOUT,ALLOWED_HELLO_LOSS*HELL
O_INTERVAL)
TTL_START = 2
ALLOWED_HELLO_LOSS = 2
BLACKLIST_TIMEOUT = 5606
HELLO_INTERVAL = 1000
LOCAL_ADD_TTL = 2
MAX_REPAIR_TTL = 10                  # 3 * NET_DIAMETER / 10
MY_ROUTE_TIMEOUT = 6000            # 2 * ACTIVE_ROUTE_TIMEOUT
NET_DIAMETER = 35
NEXT_HOP_WAIT = 50                    # NODE_TRAVERSAL_TIME + 10
NODE_TRAVERSAL_TIME =  40
NET_TRAVERSAL_TIME =  2800             # 2 * NODE_TRAVERSAL_TIME *
NET_DIAMETER
PATH_DISCOVERY_TIME = 5600           # 2 * NET_TRAVERSAL_TIME
RERR_RATELIMIT = 10
RREQ_RETRIES =  2
RREQ_RATELIMIT = 10
TTL_INCREMENT =  2
TTL_THRESHOLD = 7
```