网络协议栈分析与设计课程大作业

AODV 路由协议代码分析

学号	姓名	班级	负责模块	成绩
201692067	郭浩	软网 1601	1. 通读 RFC 2. 分析 RREQ、RREP、RERR、hello 代码 3. 整理文档	
201692050	孙焕纬	软网 1601	1. 通读 RFC 2. 分析 socket、neighbor、time_out、main 代码 3. 总结	
201692110	刘猛	软网 1601	1. 通读 RFC 2. 分析 list、routing_table、time_queue 代码	

目录

第一章 引言

无线自组网按需平面距离向量路由协议（Ad hoc On-Demand Distance Vector Routing，AODV）是应用于无线随意网络（也称作无线 Ad hoc 网络）中进行路由选择的路由协议，它能够实现单播和多播路由。该协议是 Ad Hoc 网络中按需生成路由方式的典型协议。

AODV 协议用于特定网络中的可移动节点。它能在动态变化的点对点网络中确定一条到目的地的路由，并且具有接入速度快，计算量小，内存占用低，网络负荷轻等特点。它采用目的序列号来确保在任何时候都不会出现回环（甚至在路由控制信息出现异常的时候也是如此），避免了传统的距离数组协议中会出现的很多问题（比如无穷计数问题）。

AODV 协议中定义了三种消息种类：路由请求（RREQ）、路由回复（RREP）和路由错误（RERR）。当某节点需要连接到一个新的目的节点时，它将广播一个 RREQ（路由请求消息）来尝试找到一条到目的节点的路由。如果 RREQ 消息到达目的节点，这条路由将被找到。另外一种情况下，路由也可以找到，就是 RREQ 到达了一个中间节点，该中间节点拥有到目的节点的“足够新鲜”的路由（“足够新鲜”的路由首先要是一条到目的地的正确路由，该路由还需要拥有一个足够大的序列号，该序列号不得小于 RREQ 中的序列号）。RREP 就可以通过单播从目的节点返回到发起节点，或者从一个能够找到目的节点的中间节点返回到发起节点。发生以下情况，则广播 RERR 路由错误帧：当传送数据（和尝试路由修复未果）时，如果在节点的路由表中，节点检测到某个活动路由上的下一跳发生了连接中断；如果节点接收到了一个数据包，其目的地是要到本节点没有活动路由而且也没有修复（如果使用了本地修复）；如果节点从邻居节点接收到了一个或多个活动路由的 RERR 消息。此外，AODV 协议可以通过定期广播 hello 报文来维护路由，一旦发现某一个链路断开，节点就发送 RERR 报文通知那些因链路断开而不可达的节点删除相应的记录或者对已存在的路由进行修复。

第二章 代码介绍

2.1 文件介绍

AODV 协议是由 C 语言实现的，源代码中的根目录下有 17 个.c 文件，17 个.h 文件，一个记录更改日志的 Changelog 文件，一个 GNU 通用公共授权文件 GPL，一个用于编译的 Makefile 文件，一个用于简要介绍 AODV 使用方法的 README 文件。根目录包含了两个子目录，分别是 lnx 和 patches。其中 lnx 是 linux 的缩写，里面是 AODV 协议用于 linux 内核实现的代码，包括 6 个.c 文件，7 个.h 文件和一个 Makefile 文件。而 patches 目录下所放置的 8 个文件是 AODV 补丁文件。下表给出了主要的文件并进行了说明：

文件名	说明
defs.h	宏定义
list.h	链表结构定义
params.h	常用参数定义
routing_table.h	路由表定义
timer_queue.h	计时器队列
aodv_hello.c	hello 消息
aodv_neighbor.c	添加活跃邻居节点以及处理邻居节点断开
aodv_rerr.c	RERR 消息
aodv_rrep.c	RREP 消息
aodv_rreq.c	RREQ 消息
aodv_socket.c	管理 aodv 套接字
aodv_timeout.c	超时处理
debug.c	记录日志
endian.c	判断字节序
list.c	链表操作
llf.c	链路层反馈
locality.c	寻找目的地的方位
nl.c	AODV 协议专用的套接字
main.c	协议的初始化和组织运行
routing_table.c	路由表
seek_list.c	RREQ 正要寻找的目的地的链表
time_queue.c	定时器有关的操作
lnx/kaodv-debug.c	记录日志
lnx/kaodv-expl.c	kaodv_expl 结构操作
lnx/kaodv-ipenc.c	ip 数据包操作
lnx/kaodv-mod.c	组件
lnx/kaodv-netlink.c	消息构造和发送
lnx/kaodv-queue.c	内核队列

表 2.1 AODV 路由协议主要文件

2.2 全局变量

变量	数据类型	说明
log_to_file	int	日志到文件
rt_log_interval	int	路由表日志记录间隔
unidir_hack	int	单向链路
rreq_gratuitous	int	是否是免费 RREQ
expanding_ring_search	int	扩展环搜索法支持
internet_gw_mode	int	因特网网关支持
local_repair	int	是否进行本地修复
receive_n_hellos	int	受到 hello 信息
hello_jittering	int	
optimized_hellos	int	最优 hello 消息
ratelimit	int	对 RREQ 和 RERR 消息速度限制的选项
progname	char *	程序名字
wait_on_reboot	int	等待重新启动
qual_threshold	int	准阈值
llfeedback	int	链路层反馈
gw_prefix	int	默认网关前缀
worb_timer	struct timer	等待重新启动的计时器

表 2.2 AODV 路由协议中的全局变量

第三章 AODV 路由协议的实现

3.1 概述

当某节点要给目的节点发送数据包时，要判断是否有现有的到达目的节点的有效路由，如果有，利用现有有效的路由发送数据包；如果没有，以多播形式发出 RREQ 报文，寻找新的路由（如图 3.1）。

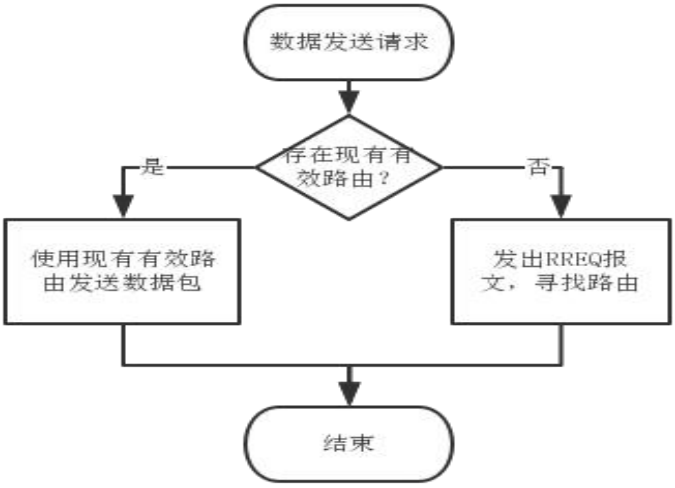

图 3.1 数据发送请求流程图

当节点收到三种种类的消息时，操作过程如图 3.2。

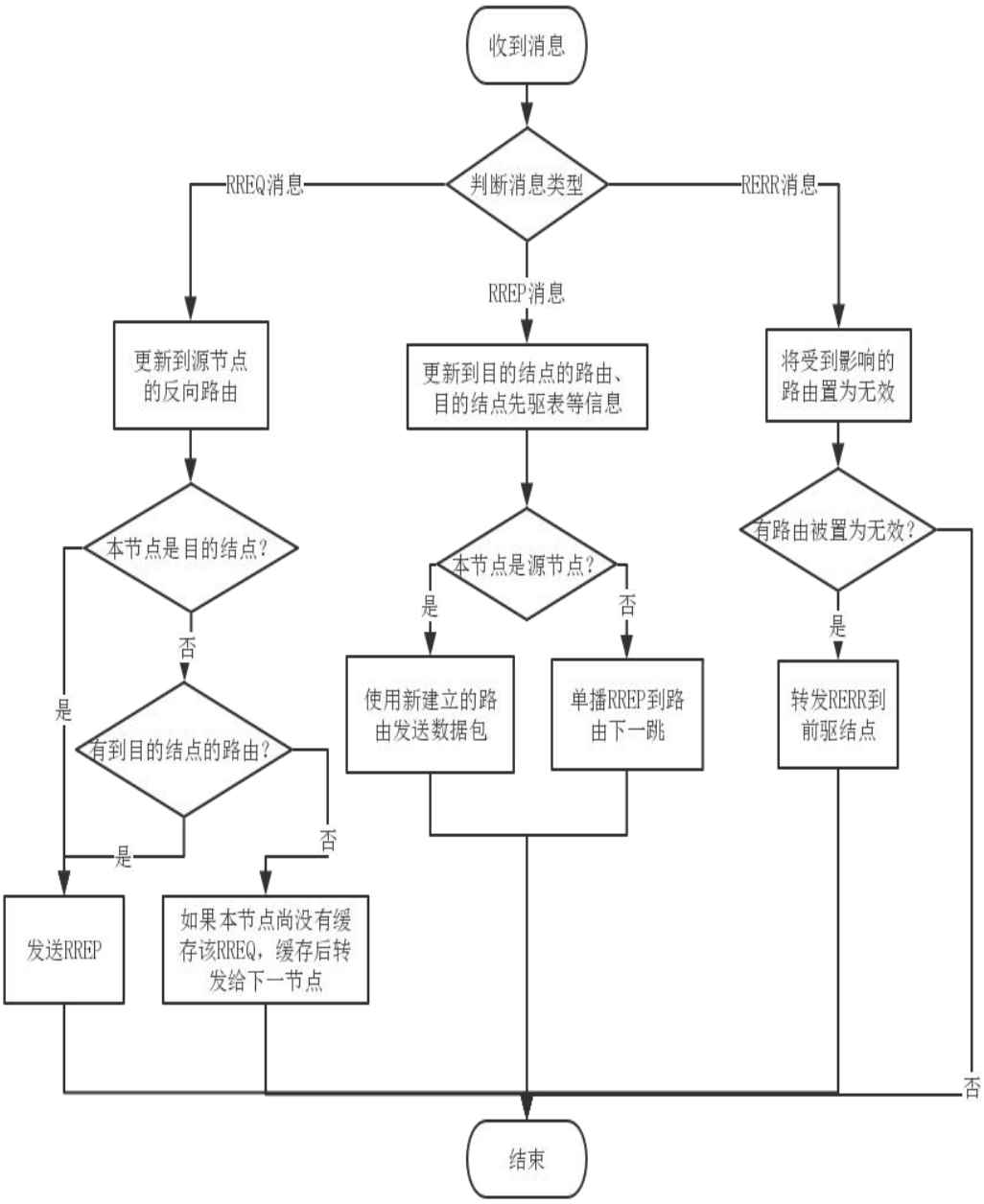

图 3.2 收到消息的操作流程

3.2 消息格式

3.2.1 RREQ 消息格式

aodv_rreq.h

```
39  typedef struct {
40      u_int8_t type;
41  #if defined(__LITTLE_ENDIAN)
42      u_int8_t res1:4;
43      u_int8_t d:1;
44      u_int8_t g:1;
45      u_int8_t r:1;
46      u_int8_t j:1;
47  #elif defined(__BIG_ENDIAN)
48      u_int8_t j:1;        /* Join flag (multicast) */
49      u_int8_t r:1;        /* Repair flag */
50      u_int8_t g:1;        /* Gratuitous RREP flag */
51      u_int8_t d:1;        /* Destination only respond */
52      u_int8_t res1:4;
53  #else
54  #error "Adjust your <bits/endian.h> defines"
55  #endif
56      u_int8_t res2;
57      u_int8_t hcnt;
58      u_int32_t rreq_id;
59      u_int32_t dest_addr;
60      u_int32_t dest_seqno;
61      u_int32_t orig_addr;
62      u_int32_t orig_seqno;
63  } RREQ;
```

aodv_rreq.h

39 - 63 RREQ 消息结构的定义，对应图 3.3。

type——Type（消息格式类型）：RREQ 值为 1。

j——Join flag（加入标志）：为多播保留。

r——Repair flag （修复标志）：为多播保留。

g——Gratuitous RREP flag（免费路由回复标志）：指示是否该向目标节点 IP 地址域指定的节点发送一个免费路由回复消息。

d——Destination only flag（仅允许目的节点回复标志）：标志置位则仅允许目的节点回复本条路由请求。

res——Reserved：Sent as 0 （填充 0），接收端忽略此字段。

hcnt——Hop Count：从发起节点到处理该请求的节点的跳数。

rreq_id——RREQ ID（路由请求标识）：这是一个序列号，用它和发起节点的 IP 就可以唯一标识一个 RREQ 信息。

dest_addr——Destination IP Address（目标节点 IP 地址）：目的节点的 IP 地址，本 RREQ 消息的任务就是想在发起节点和目的节点之间建立一条路由。

dest_seqno——Destination Sequence Number（目标节点序列号）：发起节点在以前通往目标节点的路由信息中能找到的最新的序列号。

orig_addr——Originator IP Address（发起节点 IP 地址）：发起本条路由请求消息的节点的 IP 地址。

orig_seqno——Originator Sequence Number（发起节点序列号）：指向发起者的路由表项中正在使用的序列号。

图 3.3 RREQ 消息格式

3.2.2 RREP 消息格式

aodv_rrep.h

```
37  typedef struct {
38      u_int8_t type;
39  #if defined(__LITTLE_ENDIAN)
40      u_int16_t res1:6;
41      u_int16_t a:1;
42      u_int16_t r:1;
43      u_int16_t prefix:5;
44      u_int16_t res2:3;
45  #elif defined(__BIG_ENDIAN)
46      u_int16_t r:1;
47      u_int16_t a:1;
48      u_int16_t res1:6;
49      u_int16_t res2:3;
50      u_int16_t prefix:5;
51  #else
52  #error "Adjust your <bits/endian.h> defines"
53  #endif
54      u_int8_t hcnt;
55      u_int32_t dest_addr;
56      u_int32_t dest_seqno;
57      u_int32_t orig_addr;
58      u_int32_t lifetime;
59  } RREP;
```

aodv_rrep.h

37 - 59 RREP 消息结构的定义，对应图 3.4。

type——Type（消息格式类型）：RREP 值为 2。

r——Repair flag （修复标志）：为多播保留。

a——Acknowledgment required：需要确认。

res——Reserved：Sent as 0 （填充 0），接收端忽略此字段。

prefix——Prefix Size：前缀长度。这个字段为 5 个 bit（值为 0 到 31），如果非 0，则代表下一跳节点可以作为任何具有相同路由前缀的节点被请求时的目的节点。这个“相同路由前缀”就是 Prefix Size 定义的前缀。

hcnt——Hop Count：从发起节点到目标节点的跳数。对多播路由请求，这个跳数则是从发起节点到多播节点组里产生 RREP 信息的节点的跳数。

dest_addr——Destination IP Address：目标节点的 IP 地址，一条路由将提供给这个节点。

dest_seqno——Destination Sequence Number：和这条路由联系在一起的目标序列号

orig_addr——Originator IP Address：发起 RREQ 消息的节点的 IP 地址，路由将被提供给这个节点。

lifetime——Lifetime：路由生命时间，单位为毫秒。在这段时间内，接收 RREP 的节点会认为这条路由是有效的。

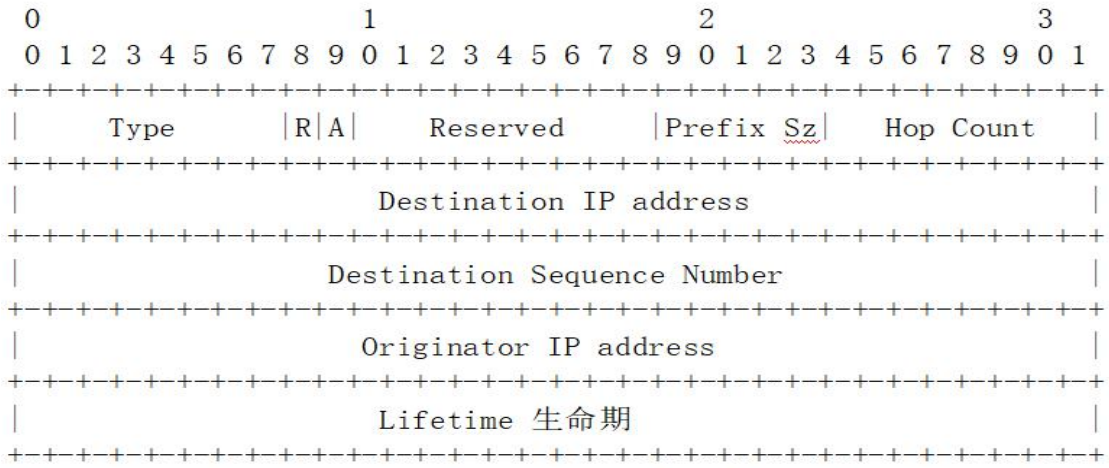

图 3.4 RREP 消息格式

3.2.3 RERR 消息格式

aodv_rerr.h

```
37  #if defined(__LITTLE_ENDIAN)
38      u_int8_t res1:7;
39      u_int8_t n:1;
40  #elif defined(__BIG_ENDIAN)
41      u_int8_t n:1;
42      u_int8_t res1:7;
43  #else
44  #error "Adjust your <bits/endian.h> defines"
45  #endif
46      u_int8_t res2;
47      u_int8_t dest_count;
48      u_int32_t dest_addr;
49      u_int32_t dest_seqno;
50  } RERR;
```

aodv_rerr.h

37 - 50 RERR 消息结构的定义，对应图 3.5。

type——Type（消息格式类型）：RERR 值为 3。

n——No delete flag（不必删除标志）：当一个节点已经对这条连接作了本地修复时，这个

标志位置位，这样上游的节点就不用删除这条路由。
res——Reserved：保留字段，填充 0，接收端不作处理。
dest_count——DestCount：本消息内包含的不可达目的节点的数目，必须至少为 1。
dest_addr——Unreachable Destination IP Address：因为连接断开而不可达的目的节点的 IP 地址。
dest_seqno——Unreachable Destination Sequence Number：路由表项里不可达目的节点的序列号。这个不可达节点的 IP 就是上面那个 Unreachable Destination IP Address。

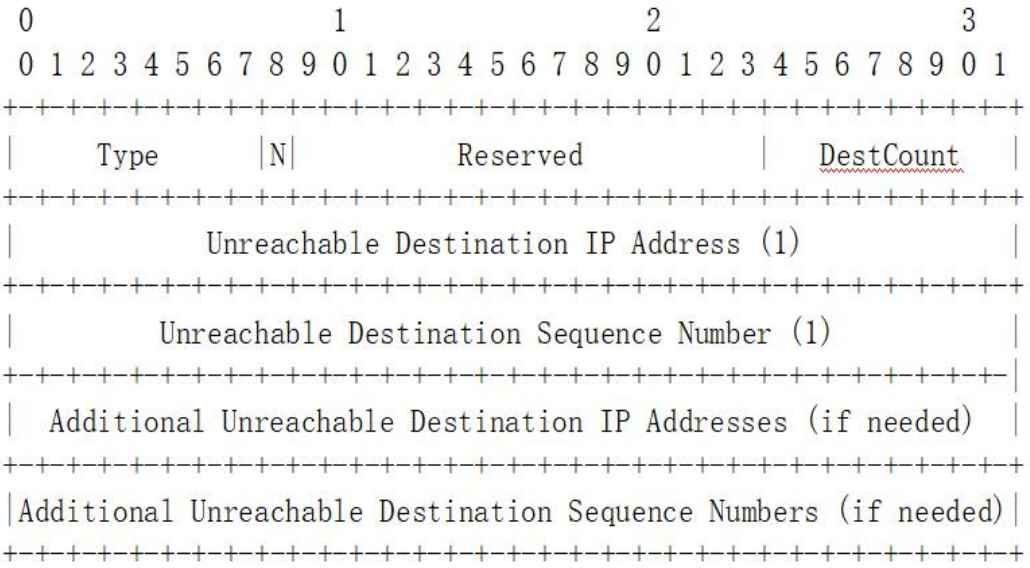

图 3.5 RERR 消息格式

3.2.4 RREP-ACK 消息格式

aodv_rrep.h

```
63  typedef struct {
64      u_int8_t type;
65      u_int8_t reserved;
66  } RREP_ack;
```

aodv_rrep.h

63 - 66 RREP-ACK 消息结构的定义，当收到一条’A’位（‘需要确认’位）被置位的 RREP 消息时，必须回发一条路由回复确认消息作为响应。当网络中存在单向连接而导致路由发现的往返过程（见 6.8 节）无法完成时，这是一个典型的操作。对应图 3.6。
type——Type（消息格式类型）：RREP-ACK 值为 4。
res——Reserved：保留字段，填充 0，接收端不作处理。

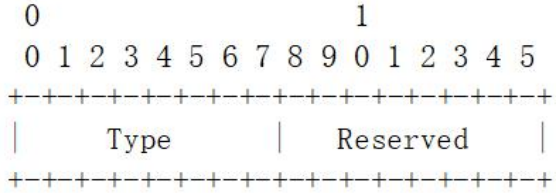

图 3.6 RREP-ACK 消息格式

3.3 链表

对 list.c 中链表操作函数的说明，如表 3.1。

函数	说明
listelm_detach(list_t * prev, list_t * next)	删除特定位置节点
listelm_add(list_t * le, list_t * prev, list_t * next)	在特定位置添加节点
list_add(list_t * head, list_t * le)	在头部添加节点
list_add_tail(list_t * head, list_t * le)	在尾部添加节点
list_detach(list_t * le)	删除链表

表 3.1 list.c 中函数说明

seek_list 中的链表是执行路由发现请求的节点链表。

seek_list.h

```
39  typedef struct seek_list {
40      list_t l;
41      struct in_addr dest_addr;
42      u_int32_t dest_seqno;
43      struct ip_data *ipd;
44      u_int8_t flags;     /* The flags we are using for resending the RREQ */
45      int reqs;
46      int ttl;
47      struct timer seek_timer;
48  } seek_list_t;
```

seek_list.h

39 - 48 对链表结构的定义。l 是节点链表，dest_addr 是目的节点 IP 地址，dest_seqno 是目的节点序列号，ipd 是发送的数据，flags 是确认是否重新发送 RREQ 消息的标志位，ttl 是剩余跳数。

函数	说明
seek_list_insert(struct in_addr dest_addr, u_int32_t dest_seqno,int ttl, u_int8_t flags, struct ip_data *ipd)	在链表中增加一个想要查找的目的节点
seek_list_remove(seek_list_t * entry)	在现有链表中删除一个节点
seek_list_find(struct in_addr dest_addr)	根据目的节点 IP 地址寻找一个链表
seek_list_print()	打印链表信息

表 3.2 seek_list.c 中函数说明

3.4 路由表

在 routing_table.c 和 routing_table.h 中定义了路由表的结构和操作。

routing_table.h

```
45  struct rt_table {
46      list_t l;
47      struct in_addr dest_addr;   /* IP address of the destination */
48      u_int32_t dest_seqno;
49      unsigned int ifindex;   /* Network interface index... */
50      struct in_addr next_hop;    /* IP address of the next hop to the dest */
51      u_int8_t hcnt;      /* Distance (in hops) to the destination */
52      u_int16_t flags;        /* Routing flags */
53      u_int8_t state;     /* The state of this entry */
54      struct timer rt_timer;  /* The timer associated with this entry */
55      struct timer ack_timer; /* RREP_ack timer for this destination */
56      struct timer hello_timer;
57      struct timeval last_hello_time;
58      u_int8_t hello_cnt;
59      hash_value hash;
60      int nprec;          /* Number of precursors */
61      list_t precursors;      /* List of neighbors using the route */
62  };
```

routing_table.h

45 - 62 对路由表项的定义。l 是节点链表，dest_addr 是目的节点 IP 地址，dest_seqno 是目的节点序列号，ifindex 是网络接口，next_hop 是下一跳 IP 地址，hcnt 是到目标节点需要的跳数，flags 和 state 是路由标志和状态，三个定时器分别记录路由建立、ACK、hello 消息的时间，last_hello_time 记录上一次 hello 消息的时间，nprec 是先驱个数，precursors 是先驱表。

函数	说明
rt_table_init()	初始化路由表项
rt_table_destroy()	删除路由表
hashing(struct in_addr *addr, hash_value * hash)	计算哈希值
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)	插入一条新的路由表项
rt_table_update(rt_table_t * rt, struct in_addr next,u_int8_t hops, u_int32_t seqno,u_int32_t lifetime, u_int8_t state, u_int16_t flags)	更新路由表信息
rt_table_update_timeout(rt_table_t * rt, u_int32_t lifetime)	更新路由表中的定时器
rt_table_update_route_timeouts(rt_table_t * fwd_rt, rt_table_t * rev_rt)	更新路由表中定时器以回应发出或接受的数据包
rt_table_find(struct in_addr dest_addr)	根据目的节点 IP 地寻找路由表项
rt_table_find_gateway()	寻找本机的默认网关
rt_table_update_inet_rt(rt_table_t * gw, u_int32_t life)	设置每一个包的下一跳在默认情况是都发给默认网关
rt_table_invalidate(rt_table_t * rt)	将一个路由表项设为无效

rt_table_delete(rt_table_t * rt)	删除一个路由表项
precursor_add(rt_table_t * rt, struct in_addr addr)	在先驱表中增加节点
precursor_remove(rt_table_t * rt, struct in_addr addr)	在先驱表中删除节点
precursor_list_destroy(rt_table_t * rt)	删除先驱表

表 3.3 routing_table.c 中函数说明

3.5 定时器队列

在 timer_queue.h 和 timer_queue.c 定义了定时器队列的结构和操作。

timer_queue.h

```
37  struct timer {
38      list_t l;
39      int used;
40      struct timeval timeout;
41      timeout_func_t handler;
42      void *data;
43  };
```

timer_queue.h

37 - 43 对定时器队列结构的定义。l 是定时器队列，timeout 记录定时器时间。

函数	说明
timer_init(struct timer *t, timeout_func_t f, void *data)	初始化定时器
timer_timeout(struct timeval *now)	初始定时器超时
timer_add(struct timer *t)	添加定时器
timer_remove(struct timer *t)	删除定时器
timer_timeout_now(struct timer *t)	设置一定时器为超时
timer_set_timeout(struct timer *t, long msec)	加快一定时器的时间
timer_left(struct timer *t)	计算定时器剩余时间
timer_age_queue()	计算定时器生存时间
printTQ(list_t * l)	打印定时器队列里每个定时器的信息

表 3.4 timer_queue.c 中函数说明

3.6 主函数

main.c

```
75  struct option longopts[] = {
76      {"interface", required_argument, NULL, 'i'},
77      {"hello-jitter", no_argument, NULL, 'j'},
78      {"log", no_argument, NULL, 'l'},
79      {"n-hellos", required_argument, NULL, 'n'},
80      {"daemon", no_argument, NULL, 'd'},
81      {"force-gratuitous", no_argument, NULL, 'g'},
82      {"opt-hellos", no_argument, NULL, 'o'},
83      {"quality-threshold", required_argument, NULL, 'q'},
84      {"log-rt-table", required_argument, NULL, 'r'},
85      {"unidir_hack", no_argument, NULL, 'u'},
86      {"gateway-mode", no_argument, NULL, 'w'},
87      {"help", no_argument, NULL, 'h'},
88      {"no-expanding-ring", no_argument, NULL, 'x'},
89      {"no-worb", no_argument, NULL, 'D'},
90      {"local-repair", no_argument, NULL, 'L'},
91      {"rate-limit", no_argument, NULL, 'R'},
92      {"version", no_argument, NULL, 'V'},
93      {"llfeedback", no_argument, NULL, 'f'},
94      {0}
95  };
```

main.c

75 - 79 对命令行参数的定义，对应表 3.5，在 usage 函数中打印命令行参数提示，在 main 函数中实现，与对应的全局变量关联。

命令行参数	全称	说明
-i	interface	表示要绑定的接口
-j	hello-jitter	触发 hello-jitter 功能，默认情况下开启
-l	log	输出日志
-n	n-hellos	设置接收到 n 个 hello 消息才当成邻居
-d	daemon	开启守护进程模式，脱离控制台窗口
-g	force-gratuitous	对每个 RREQ 消息强制设置 gratuifous 标记
-o	opt-hellos	设置只在转发数据包时发送 hello 信息
-q	quality-threshold	为控制包设置一个信号质量最小的阈值
-r	log-rt-table	每隔一段时间记录路由表
-u	unidir_hack	侦测并避免单向链路
-w	gateway-mode	开启实验性因特网网关支持
-h	Help	显示所有参数及含义
-x	no-expanding-ring	禁用 RREQ 消息扩展环搜索法支持
-D	no-worb	禁用重启延迟等待
-L	local-repair	开启本地修复
-R	rate-limit	开启 RREQ 和 RERR 消息速率限制
-V	version	输出版本信息
-f	llfeedback	开启链路层反馈

表 3.5 命令行参数说明

main.c

```
607     if (geteuid() != 0) {
608     fprintf(stderr, "must be root\n");
609     exit(1);
610     }
```

main.c

607 - 610 检查是否在 root 权限下运行。

main.c

```
626     rt_table_init();
627     log_init();
628     /*   packet_queue_init(); */
629     host_init(ifname);
630     /*   packet_input_init(); */
631     nl_init();
632     nl_send_conf_msg();
633     aodv_socket_init();
634 #ifdef LLFEEDBACK
635     if (llfeedback) {
636     llf_init();
```

main.c

626 - 623 初始化工作，初始化路由表项、套接字。

main.c

```
641     FD_ZERO(&readers);
642     for (i = 0; i < nr_callbacks; i++) {
643     FD_SET(callbacks[i].fd, &readers);
644     if (callbacks[i].fd >= nfds)
645         nfds = callbacks[i].fd + 1;
646     }
```

main.c

641 - 646 设置套接字。

main.c

```
649     if (wait_on_reboot) {
650     timer_init(&worb_timer, wait_on_reboot_timeout, &wait_on_reboot);
651     timer_set_timeout(&worb_timer, DELETE_PERIOD);
652     alog(LOG_NOTICE, 0, __FUNCTION__,
653          "In wait on reboot for %d milliseconds. Disable with \"-D\".",
654          DELETE_PERIOD);
655     }
```

main.c

649 - 655 设置重启定时器，如果超时重启定时器。

main.c

```
667     timeout = timer_age_queue();
668
669     timeout_spec.tv_sec = timeout->tv_sec;
670     timeout_spec.tv_nsec = timeout->tv_usec * 1000;
671
672     if ((n = pselect(nfds, &rfds, NULL, NULL, &timeout_spec, &origmask))
673         if (errno != EINTR)
674         alog(LOG_WARNING, errno, __FUNCTION__,
675              "Failed select (main loop)");
676         continue;
```

main.c

667 - 676 设置定时器，调用 pselect 函数，用于 IO 复用，监视多个文件描述符的集合，判断是否有符合条件的时间发生。

3.7 套接字管理

3.7.1 套接字初始化

aodv_socket.c

```
 98     for (i = 0; i < MAX_NR_INTERFACES; i++) {
 99     if (!DEV_NR(i).enabled)
100         continue;
```

aodv_socket.c

98 - 100 对每个可用接口打开一个套接字

aodv_socket.c

```
103     DEV_NR(i).sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
104     if (DEV_NR(i).sock < 0) {
105         perror("");
106         exit(-1);
107     }
```

aodv_socket.c

103 - 107 创建 UDP 套接字

aodv_socket.c

```
110     DEV_NR(i).psock = socZket(PF_INET, SOCK_RAW, IPPROTO_RAW);
111
112     if (DEV_NR(i).psock < 0) {
113         perror("");
114         exit(-1);
115     }
```

aodv_socket.c

110 - 115 数据包发送套接字。

aodv_socket.c

```
118     memset(&aodv_addr, 0, sizeof(aodv_addr));
119     aodv_addr.sin_family = AF_INET;
120     aodv_addr.sin_port = htons(AODV_PORT);
121     aodv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

aodv_socket.c

118 - 121 将套接字与 AODV 端口对应。在之后会会设置套接字选项，如果某个选项设置失败会退出。

aodv_socket.c

```
184     for (;; bufsize -= 1024) {
185         if (setsockopt(DEV_NR(i).sock, SOL_SOCKET, SO_RCVBUF,
186                (char *) &bufsize, optlen) == 0) {
187         alog(LOG_NOTICE, 0, __FUNCTION__,
188              "Receive buffer size set to %d", bufsize);
189         break;
190         }
191         if (bufsize < RECV_BUF_SIZE) {
192         alog(LOG_ERR, 0, __FUNCTION__,
193              "Could not set receive buffer size");
194         exit(-1);
195         }
196     }
```

aodv_socket.c

184 - 196 设置接受缓存区允许的最大长度。

下面是 nl.c 文件中 AODV 专用套接字的初始化函数特殊的部分。

nl.c

```
91      aodvnl.sock = socket(PF_NETLINK, SOCK_RAW, NETLINK_AODV);
```

nl.c

91 创建套接字时 socket 函数第一个参数是 PF_NETLINK，表明套接字遵循协议族 NETLINK。第二个参数是 SOCK_RAW，表明采用原始套接字。第三个参数是 NETLINK_AODV，表明采用的协议为 AODV 协议。

nl.c

```
121     memset(&rtnl, 0, sizeof(struct nlsock));
122     rtnl.seq = 0;
123     rtnl.local.nl_family = AF_NETLINK;
124     rtnl.local.nl_groups =
125         RTMGRP_NOTIFY | RTMGRP_IPV4_IFADDR | RTMGRP_IPV4_ROUTE;
126     rtnl.local.nl_pid = getpid();
127
128     rtnl.sock = socket(PF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

nl.c

121 - 128 创建路由套接字时 socket 函数第一个参数是 PF_NETLINK，表明套接字遵循协议族 NETLINK。第二个参数是 SOCK_RAW，表明采用原始套接字。第三个参数是 NETLINK_ROUTE，表明套接字负责路由信息传递。

3.7.2 分类处理消息

aodv_socket_process_packet 函数用于辨别接收到的消息的类型，对不同类型的消息进行不同的操作。

aodv_socket.c

```
218     if ((aodv_msg->type == AODV_RREP && ttl == 1 &&
219      dst.s_addr == AODV_BROADCAST)) {
220     hello_process((RREP *) aodv_msg, len, ifindex);
221     return;
222     }
```

aodv_socket.c

218 - 222 如果接受到了 hello 消息，执行 hello 消息操作。

aodv_socket.c

```
229     switch (aodv_msg->type) {
230
231     case AODV_RREQ:
232     rreq_process((RREQ *) aodv_msg, len, src, dst, ttl, ifindex);
233     break;
234     case AODV_RREP:
235     DEBUG(LOG_DEBUG, 0, "Received RREP");
236     rrep_process((RREP *) aodv_msg, len, src, dst, ttl, ifindex);
237     break;
238     case AODV_RERR:
239     DEBUG(LOG_DEBUG, 0, "Received RERR");
240     rerr_process((RERR *) aodv_msg, len, src, dst);
241     break;
242     case AODV_RREP_ACK:
243     DEBUG(LOG_DEBUG, 0, "Received RREP_ACK");
244     rrep_ack_process((RREP_ack *) aodv_msg, len, src, dst);
245     break;
246     default:
247     alog(LOG_WARNING, 0, __FUNCTION__,
248          "Unknown msg type %u rcvd from %s to %s", aodv_msg->type,
249          ip_to_str(src), ip_to_str(dst));
250     }
251 }
```

aodv_socket.c

229 - 251 判断接收到的消息的类型，对 RREQ、RREP、RERR、RREP-ACK 四种类型的的消息分别执行各自对应的操作，如果接收到的消息类型不在这四种之中，则报错，提示未知消息类型。

3.7.3 接受 AODV 消息

recvAODVUUPacket 函数用于接受 AODV 消息。

aodv_socket.c

```
267     AODV_msg *aodv_msg = (AODV_msg *) recv_buf;
268
269     /* Only handle AODVUU packets */
270     assert(ch->ptype() == PT_AODVUU);
271
272     /* Only process incoming packets */
273     assert(ch->direction() == hdr_cmn::UP);
274
275     /* Copy message to receive buffer */
276     memcpy(recv_buf, ah, RECV_BUF_SIZE);
277
278     /* Deallocate packet, we have the information we need... */
279     Packet::free(p);
```

aodv_socket.c

267 分配缓存空间。

269 - 273 判断该消息是否是 AODV 消息并且是从外部发来的消息。

275 - 279 在上述条件满足后，将该消息复制到接收区缓存中，我们得到需要的信息后就释放掉该消息。

aodv_socket.c

```
282     for (i = 0; i < MAX_NR_INTERFACES; i++)
283     if (this_host.devs[i].enabled &&
284         memcmp(&src, &this_host.devs[i].ipaddr,
285            sizeof(struct in_addr)) == 0)
286         return;
287
288     aodv_socket_process_packet(aodv_msg, len, src, dst, ttl, NS_IFINDEX);
289 }
```

aodv_socket.c

282 - 289 检查是否有本地产生的消息，如果有就忽略掉。之后调用 aodv_socket_process_packet 函数对接受到的消息分类处理。

3.7.4 读取 AODV 套接字

aodv_socket_read 函数用于读取 AODV 套接字

aodv_socket.c

```
325     for (cmsg = CMSG_FIRSTHDR(&msgh); cmsg != NULL;
326      cmsg = CMSG_NXTHDR_FIX(&msgh, cmsg)) {
327     if (cmsg->cmsg_level == SOL_IP) {
328         switch (cmsg->cmsg_type) {
329         case IP_TTL:
330         ttl = *(CMSG_DATA(cmsg));
331         break;
332         case IP_PKTINFO:
333           {
334         struct in_pktinfo *pi = (struct in_pktinfo *)CMSG_DATA(cmsg);
335         dst.s_addr = pi->ipi_addr.s_addr;
336           }
337         }
338     }
339     }
340
341     if (ttl < 0) {
342     DEBUG(LOG_DEBUG, 0, "No TTL, packet ignored!");
343     return;
344     }
```

aodv_socket.c

325 - 339 从接收到的消息中取出 ttl 和目的节点的地址。
341 - 344 判断 ttl 是否小于 0，如果小于 0，表示该消息已不能继续发送，就丢弃掉该消息。

aodv_socket.c

```
347     for (i = 0; i < MAX_NR_INTERFACES; i++)
348     if (this_host.devs[i].enabled &&
349         memcmp(&src, &this_host.devs[i].ipaddr,
350            sizeof(struct in_addr)) == 0)
351         return;
352
353     aodv_msg = (AODV_msg *) recv_buf;
354
355     dev = devfromsock(fd);
356
357     if (!dev) {
358     DEBUG(LOG_ERR, 0, "Could not get device info!\n");
359     return;
360     }
361
362     aodv_socket_process_packet(aodv_msg, len, src, dst, ttl, dev->ifindex);
363 }
```

aodv_socket.c

347 - 351 和上个函数一样，判断消息是否由本地生成，如果是就忽略掉。
355 - 363 判断能否获取到服务信息，如果不能就返回，如果可以，调用 aodv_socket_process_packet 函数对接受到的消息分类处理。

3.7.5 发送 AODV 套接字

aodv_socket_send 函数用于发送 AODV 套接字。

aodv_socket.c

```
386     if (setsockopt(dev->sock, SOL_IP, IP_TTL, &ttl, sizeof(ttl)) < 0) {
387     alog(LOG_WARNING, 0, __FUNCTION__, "ERROR setting ttl!");
388     return;
389     }
```

aodv_socket.c

386 - 389 设置 ttl 值。

aodv_socket.c

```
401     if (wait_on_reboot && aodv_msg->type == AODV_RREP)
402     return;
```

aodv_socket.c

401 - 402 判断是否在等待重启，如果是，不能发送 RREP 消息。

aodv_socket.c

```
420     // Set common header fields
421     ch->ptype() = PT_AODVUU;
422     ch->direction() = hdr_cmn::DOWN;
423     ch->size() += len + IP_HDR_LEN;
424     ch->iface() = -2;
425     ch->error() = 0;
426     ch->prev_hop_ = (nsaddr_t) dev->ipaddr.s_addr;
427
428     // Set IP header fields
429     ih->saddr() = (nsaddr_t) dev->ipaddr.s_addr;
430     ih->daddr() = (nsaddr_t) dst.s_addr;
431     ih->ttl() = ttl;
432
433     // Note: Port number for routing agents, not AODV port number!
434     ih->sport() = RT_PORT;
435     ih->dport() = RT_PORT;
```

aodv_socket.c

420 - 426 设置常用的首部信息。

428 - 431 设置 IP 头部信息。

434 - 435 设置路由端口号。

aodv_socket.c

```
446     if (ratelimit) {
447
448     gettimeofday(&now, NULL);
449
450     switch (aodv_msg->type) {
451     case AODV_RREQ:
452         if (num_rreq == (RREQ_RATELIMIT - 1)) {
453         if (timeval_diff(&now, &rreq_ratel[0]) < 1000) {
454             DEBUG(LOG_DEBUG, 0, "RATELIMIT: Dropping RREQ %ld ms",
455               timeval_diff(&now, &rreq_ratel[0]));
456 #ifdef NS_PORT
457             Packet::free(p);
458 #endif
459             return;
460         } else {
461             memmove(rreq_ratel, &rreq_ratel[1],
462                 sizeof(struct timeval) * (num_rreq - 1));
463             memcpy(&rreq_ratel[num_rreq - 1], &now,
464                sizeof(struct timeval));
465         }
466         } else {
467         memcpy(&rreq_ratel[num_rreq], &now, sizeof(struct timeval));
468         num_rreq++;
469         }
470         break;
471     case AODV_RERR:
472         if (num_rerr == (RERR_RATELIMIT - 1)) {
473         if (timeval_diff(&now, &rerr_ratel[0]) < 1000) {
474             DEBUG(LOG_DEBUG, 0, "RATELIMIT: Dropping RERR %ld ms",
475               timeval_diff(&now, &rerr_ratel[0]));
476 #ifdef NS_PORT
477             Packet::free(p);
478 #endif
479             return;
480         } else {
481             memmove(rerr_ratel, &rerr_ratel[1],
482                 sizeof(struct timeval) * (num_rerr - 1));
483             memcpy(&rerr_ratel[num_rerr - 1], &now,
484                sizeof(struct timeval));
485         }
486         } else {
487         memcpy(&rerr_ratel[num_rerr], &now, sizeof(struct timeval));
488         num_rerr++;
489         }
490         break;
491     }
492     }
```

aodv_socket.c

446 - 492 判断是否有发送信息的速率限制，如果有，则不能超出速率限制发送 RREQ 或 RERR 消息。检查在规定时间内，发送的 RREQ 或 RERR 消息数量超出了规定的 ratelimit，如果超出，将继续收到的 RREQ 或 RERR 消息丢弃。

aodv_socket.c

```
496     if (dst.s_addr == AODV_BROADCAST) {
497
498     gettimeofday(&this_host.bcast_time, NULL);
499
500 #ifdef NS_PORT
501     ch->addr_type() = NS_AF_NONE;
502
503     sendPacket(p, dst, 0.0);
504 #else
505
506     retval = sendto(dev->sock, send_buf, len, 0,
507             (struct sockaddr *) &dst_addr, sizeof(dst_addr));
508
509     if (retval < 0) {
510
511         alog(LOG_WARNING, errno, __FUNCTION__, "Failed send to bc %s",
512          ip_to_str(dst));
513         return;
514     }
515 #endif
516
517     } else {
518
519 #ifdef NS_PORT
520     ch->addr_type() = NS_AF_INET;
521     /* We trust the decision of next hop for all AODV messages... */
522
523     if (dst.s_addr == AODV_BROADCAST)
524         sendPacket(p, dst, 0.001 * Random::uniform());
525     else
526         sendPacket(p, dst, 0.0);
527 #else
528     retval = sendto(dev->sock, send_buf, len, 0,
529             (struct sockaddr *) &dst_addr, sizeof(dst_addr));
530
531     if (retval < 0) {
532         alog(LOG_WARNING, errno, __FUNCTION__, "Failed send to %s",
533          ip_to_str(dst));
534         return;
535     }
536 #endif
537     }
```

aodv_socket.c

496 - 504 判断该消息是否是广播消息，如果是，更新上一次广播的时间以避免对 hello 消息的不必要广播。

3.8 RREQ 消息

3.8.1 创建 RREQ 消息

rreq_create 函数用于创建 RREQ 消息。

aodv_rreq.c

```
66      rreq = (RREQ *) aodv_socket_new_msg();
67      rreq->type = AODV_RREQ;
68      rreq->res1 = 0;
69      rreq->res2 = 0;
70      rreq->hcnt = 0;
71      rreq->rreq_id = htonl(this_host.rreq_id++);
72      rreq->dest_addr = dest_addr.s_addr;
73      rreq->dest_seqno = htonl(dest_seqno);
74      rreq->orig_addr = orig_addr.s_addr;
75
76      /* Immediately before a node originates a RREQ flood it must
77         increment its sequence number... */
78      seqno_incr(this_host.seqno);
79      rreq->orig_seqno = htonl(this_host.seqno);
80
81      if (flags & RREQ_JOIN)
82      rreq->j = 1;
83      if (flags & RREQ_REPAIR)
84      rreq->r = 1;
85      if (flags & RREQ_GRATUITOUS)
86      rreq->g = 1;
87      if (flags & RREQ_DEST_ONLY)
88      rreq->d = 1;
```

aodv_rreq.c

66 - 74 为 RREQ 消息的各个属性赋值。
78 - 79 RREQ 消息增加自身的序列号。
81 - 88 判断 RREQ 消息状态，为标志位赋值。

3.8.2 发送 RREQ 消息

rreq_send 函数用于发送 RREQ 消息。

aodv_rreq.c

```
123     dest.s_addr = AODV_BROADCAST;
124
125     /* Check if we should force the gratuitous flag... (-g option). */
126     if (rreq_gratuitous)
127     flags |= RREQ_GRATUITOUS;
128
129     /* Broadcast on all interfaces */
130     for (i = 0; i < MAX_NR_INTERFACES; i++) {
131     if (!DEV_NR(i).enabled)
132         continue;
133     rreq = rreq_create(flags, dest_addr, dest_seqno, DEV_NR(i).ipaddr);
134     aodv_socket_send((AODV_msg *) rreq, dest, RREQ_SIZE, ttl, &DEV_NR(i));
135     }
136 }
```

aodv_rreq.c

123 设置发送类型为广播。
126 - 127 监测是否需要将 G 标志位置位。
129 - 134 向当前节点的所有接口广播 RREQ 消息

3.8.3 转发 RREQ 消息

rreq_forward 函数用于转发 RREQ 消息。

aodv_rreq.c

```
147     DEBUG(LOG_INFO, 0, "forwarding RREQ src=%s, rreq_id=%lu",
148       ip_to_str(orig), ntohl(rreq->rreq_id));
149
150     /* Queue the received message in the send buffer */
151     rreq = (RREQ *) aodv_socket_queue_msg((AODV_msg *) rreq, size);
152
153     rreq->hcnt++;       /* Increase hopcount to account for
154                  * intermediate route */
155
156     /* Send out on all interfaces */
157     for (i = 0; i < MAX_NR_INTERFACES; i++) {
158     if (!DEV_NR(i).enabled)
159         continue;
160     aodv_socket_send((AODV_msg *) rreq, dest, size, ttl, &DEV_NR(i));
```

aodv_rreq.c

151 - 153 将收到的消息送到发送缓存区队列中，跳数+1。

157 - 160 在 ttl 允许（即 ttl>0）时向所有接口广播 RREQ 消息。

3.8.4 处理 RREQ 消息

rreq_process 函数用于处理接收到的 RREQ 消息。

aodv_rreq.c

```
205     if (rreq_blacklist_find(ip_src)) {
206     DEBUG(LOG_DEBUG, 0, "prev hop of RREQ blacklisted, ignoring!");
207     return;
208     }
209
210     /* Ignore already processed RREQs. */
211     if (rreq_record_find(rreq_orig, rreq_id))
212     return;
213
214     /* Now buffer this RREQ so that we don't process a similar RREQ we
215        get within PATH_DISCOVERY_TIME. */
216     rreq_record_insert(rreq_orig, rreq_id);
217
218     /* Determine whether there are any RREQ extensions */
219     ext = (AODV_ext *) ((char *) rreq + RREQ_SIZE);
```

aodv_rreq.c

205 - 207 检查上一跳的节点是否在当前节点的黑名单中，如果在，则放弃这个消息，可以减少网络攻击。

211 - 212 检查消息是否被处理过，如果是，忽略这个消息。

216 将处理过的消息记录，避免重复处理。

aodv_rreq.c

```
241     rev_rt = rt_table_find(rreq_orig);
```

aodv_rreq.c

241 当一个节点收到一条 RREQ 消息时，它首先创建一个到前一跳节点的路由，或者更新原来已有的，但序列号不对的到上一跳的路由。

aodv_rreq.c

```
312     if (rreq_dest.s_addr == DEV_IFINDEX(ifindex).ipaddr.s_addr) {
313
314     /* WE are the RREQ DESTINATION. Update the node's own
315        sequence number to the maximum of the current seqno and the
316        one in the RREQ. */
317     if (rreq_dest_seqno != 0) {
318         if ((int32_t) this_host.seqno < (int32_t) rreq_dest_seqno)
319         this_host.seqno = rreq_dest_seqno;
320         else if (this_host.seqno == rreq_dest_seqno)
321         seqno_incr(this_host.seqno);
322     }
323     rrep = rrep_create(0, 0, 0, DEV_IFINDEX(rev_rt->ifindex).ipaddr,
324                this_host.seqno, rev_rt->dest_addr,
325                MY_ROUTE_TIMEOUT);
326
327     rrep_send(rrep, rev_rt, NULL, RREP_SIZE);
328
```

aodv_rreq.c

312 - 328 判断当前节点是否是 rreq 的目的节点，若是则回复 rrep 消息，并且更新最大的序列号。

aodv_rreq.c

```
378         if (fwd_rt->dest_seqno != 0 &&
379         (int32_t) fwd_rt->dest_seqno >= (int32_t) rreq_dest_seqno) {
380         lifetime = timeval_diff(&fwd_rt->rt_timer.timeout, &now);
381         rrep = rrep_create(0, 0, fwd_rt->hcnt, fwd_rt->dest_addr,
382                    fwd_rt->dest_seqno, rev_rt->dest_addr,
383                    lifetime);
384         rrep_send(rrep, rev_rt, fwd_rt, rrep_size);
385         } else {
386         goto forward;
```

aodv_rreq.c

378 - 386 判断接收到的 rreq 消息的序列号是不是足够的新，若不是说明是过时的消息，则不用转发。

aodv_rreq.c

```
333     fwd_rt = rt_table_find(rreq_dest);
334
335     if (fwd_rt && fwd_rt->state == VALID && !rreq->d) {
336         struct timeval now;
337         u_int32_t lifetime;
338
339         /* GENERATE RREP, i.e we have an ACTIVE route entry that is fresh
340            enough (our destination sequence number for that route is
341            larger than the one in the RREQ). */
342
343         gettimeofday(&now, NULL);
344 #ifdef CONFIG_GATEWAY_DISABLED
345         if (fwd_rt->flags & RT_INET_DEST) {
346         rt_table_t *gw_rt;
347         /* This node knows that this is a rreq for an Internet
348          * destination and it has a valid route to the gateway */
349
350         goto forward;   // DISABLED
351
352         gw_rt = rt_table_find(fwd_rt->next_hop);
353
354         if (!gw_rt || gw_rt->state == INVALID)
355             goto forward;
356
357         lifetime = timeval_diff(&gw_rt->rt_timer.timeout, &now);
358
359         rrep = rrep_create(0, 0, gw_rt->hcnt, gw_rt->dest_addr,
360                            gw_rt->dest_seqno, rev_rt->dest_addr,
361                            lifetime);
362
363         ext = rrep_add_ext(rrep, RREP_INET_DEST_EXT, rrep_size,
364                            sizeof(struct in_addr), (char *) &rreq_dest);
365
366         rrep_size += AODV_EXT_SIZE(ext);
367
368         DEBUG(LOG_DEBUG, 0,
369               "Intermediate node response for INTERNET dest: %s rrep_size=%d",
370               ip_to_str(rreq_dest), rrep_size);
371
372         rrep_send(rrep, rev_rt, gw_rt, rrep_size);
373         return;
```

aodv_rreq.c

333 - 373 若不是，则查看是否包含 rreq 目的节点的路由信息，若有则回复 rrep 消息，倘若没有则继续广播 rreq 消息。

3.8.5 路由发现

rreq_route_discovery 函数用于 RREQ 消息的路由发现

aodv_rreq.c

```
436     rt = rt_table_find(dest_addr);
437
438     ttl = NET_DIAMETER;     /* This is the TTL if we don't use expanding
439                                ring search */
```

aodv_rreq.c

436 - 439 如果在当前节点的路由表中已经包含目的节点的路由表项，则设置 ttl 为 NET_DIAMETER。

3.9 RREP 消息

3.9.1 创建 RREP 消息

aodv_rrep.c

```
52      rrep = (RREP *) aodv_socket_new_msg();
53      rrep->type = AODV_RREP;
54      rrep->res1 = 0;
55      rrep->res2 = 0;
56      rrep->prefix = prefix;
57      rrep->hcnt = hcnt;
58      rrep->dest_addr = dest_addr.s_addr;
59      rrep->dest_seqno = htonl(dest_seqno);
60      rrep->orig_addr = orig_addr.s_addr;
61      rrep->lifetime = htonl(life);
62
63      if (flags & RREP_REPAIR)
64      rrep->r = 1;
65      if (flags & RREP_ACK)
66      rrep->a = 1;
```

aodv_rrep.c

52 - 66 创建 RREP 消息，为消息格式中提到的各种属性赋值。

3.9.2 RREP-ACK 消息

aodv_rrep.c

```
79  RREP_ack *NS_CLASS rrep_ack_create()
80  {
81      RREP_ack *rrep_ack;
82
83      rrep_ack = (RREP_ack *) aodv_socket_new_msg();
84      rrep_ack->type = AODV_RREP_ACK;
85
86      DEBUG(LOG_DEBUG, 0, "Assembled RREP_ack");
87
88      return rrep_ack;
89  }
```

aodv_rrep.c

79 - 89 创建 RREP-ACK 消息函数。

aodv_rrep.c

```
91  void NS_CLASS rrep_ack_process(RREP_ack * rrep_ack, int rrep_acklen,
92                                 struct in_addr ip_src, struct in_addr ip_dst)
93  {
94      rt_table_t *rt;
95
96      rt = rt_table_find(ip_src);
97
98      if (rt == NULL) {
99      DEBUG(LOG_WARNING, 0, "No RREP_ACK expected for %s", ip_to_str(ip_sr
100
101     return;
102     }
103     DEBUG(LOG_DEBUG, 0, "Received RREP_ACK from %s", ip_to_str(ip_src));
104
105     /* Remove unexpired timer for this RREP_ACK */
106     timer_remove(&rt->ack_timer);
107 }
```

aodv_rrep.c

91 - 107 RREP-ACK 消息处理函数

96 - 103 检查当前节点的路由表中是否有接受到的 RREP-ACK 消息的源 IP 地址，如果没有说明当前节点没有到该 RREP-ACK 源节点的路由，说明这条消息是错误的。如果有，说当前节点向该 RREP-ACK 源节点发送过消息，一出 ACK 计时器。

3.9.3 发送 RREP 消息

rrep_send 用于发送 RREP 消息。

aodv_rrep.c

```
133     if (!rev_rt) {
134     DEBUG(LOG_WARNING, 0, "Can't send RREP, rev_rt = NULL!");
135     return;
136     }
```

aodv_rrep.c

133 - 136 检查路由表项是否为空，如果为空，不能发送 RREP 消息。

aodv_rrep.c

```
141     if ((rev_rt->state == VALID && rev_rt->flags & RT_UNIDIR) ||
142     (rev_rt->hcnt == 1 && unidir_hack)) {
143     rt_table_t *neighbor = rt_table_find(rev_rt->next_hop);
```

aodv_rrep.c

141 - 143 检查是否需要发送 RREP-ACK 消息，如果一个节点转发一个可能出错或是单向的 RREP，这个节点应该将“A”标志位置位以指示此 RREP 包的接受者送回以个 RREP-ACK 包确认已接受到该包。

aodv_rrep.c

```
145     if (neighbor && neighbor->state == VALID && !neighbor->ack_timer.used) {
146         /* If the node we received a RREQ for is a neighbor we are
147            probably facing a unidirectional link... Better request a
148            RREP-ack */
149         rrep_flags |= RREP_ACK;
150         neighbor->flags |= RT_UNIDIR;
151
152         /* Must remove any pending hello timeouts when we set the
153            RT_UNIDIR flag, else the route may expire after we begin to
154            ignore hellos... */
155         timer_remove(&neighbor->hello_timer);
156         neighbor_link_break(neighbor);
157
158         DEBUG(LOG_DEBUG, 0, "Link to %s is unidirectional!",
159           ip_to_str(neighbor->dest_addr));
160
161         timer_set_timeout(&neighbor->ack_timer, NEXT_HOP_WAIT);
162     }
163     }
```

aodv_rrep.c

145 - 150 当接收到的 RREQ 消息目的地址的 IP 地址是当前节点的直接邻居节点，很有可能面临单向链路这种尴尬的情形，所以应该首先发送一个 RREP-ACK 消息。
155 - 159 当设置单向路由时，必须移除期间内的所有 hello 计时器。

aodv_rrep.c

```
174     precursor_add(fwd_rt, rev_rt->next_hop);
175     precursor_add(rev_rt, fwd_rt->next_hop);
```

aodv_rrep.c

174 - 175 更新先驱表。

3.9.4 转发 RREP 消息

rrep_forward 函数用于转发 RREP 消息。

aodv_rrep.c

```
186     if (!fwd_rt || !rev_rt) {
187     DEBUG(LOG_WARNING, 0, "Could not forward RREP because of NULL route!");
188     return;
189     }
190
191     if (!rrep) {
192     DEBUG(LOG_WARNING, 0, "No RREP to forward!");
193     return;
194     }
```

aodv_rrep.c

186 - 194 检查当路由表项或 RREP 消息为空时，停止转发。

aodv_rrep.c

```
206         if (rev_rt->dest_addr.s_addr != rev_rt->next_hop.s_addr)
207             neighbor = rt_table_find(rev_rt->next_hop);
208         else
209             neighbor = rev_rt;
```

aodv_rrep.c

206 - 209 如果 RREP 消息的源节点不是邻居节点，必须要找到一个通往源节点的下一跳邻居节点。

3.9.5 处理 RREP 消息

rrep_process 函数用于处理 RREP 消息。

aodv_rrep.c

```
259     if (rreplen < (int) RREP_SIZE) {
260     alog(LOG_WARNING, 0, __FUNCTION__,
261          "IP data field too short (%u bytes)"
262          " from %s to %s", rreplen, ip_to_str(ip_src), ip_to_str(ip_dst));
263     return;
264     }
```

aodv_rrep.c

259 - 264 判断 RREP 消息的长度是否小于规定值，小于则不能发送。

aodv_rrep.c

```
267     if (rrep_dest.s_addr == DEV_IFINDEX(ifindex).ipaddr.s_addr)
268     return;
```

aodv_rrep.c

267 - 268 如果 RREP 消息时建立到自身路由，则忽略。

aodv_rrep.c

```
315     if (!fwd_rt) {
316     /* We didn't have an existing entry, so we insert a new one. */
317     fwd_rt = rt_table_insert(rrep_dest, ip_src, rrep_new_hcnt, rrep_seqno,
318                  rrep_lifetime, VALID, rt_flags, ifindex);
319     } else if (fwd_rt->dest_seqno == 0 ||
320            (int32_t) rrep_seqno > (int32_t) fwd_rt->dest_seqno ||
321            (rrep_seqno == fwd_rt->dest_seqno &&
322         (fwd_rt->state == INVALID || fwd_rt->flags & RT_UNIDIR ||
323          rrep_new_hcnt < fwd_rt->hcnt))) {
324     pre_repair_hcnt = fwd_rt->hcnt;
325     pre_repair_flags = fwd_rt->flags;
326
327     fwd_rt = rt_table_update(fwd_rt, ip_src, rrep_new_hcnt, rrep_seqno,
328                  rrep_lifetime, VALID,
329                  rt_flags | fwd_rt->flags);
330     } else {
331     if (fwd_rt->hcnt > 1) {
332         DEBUG(LOG_DEBUG, 0,
333           "Dropping RREP, fwd_rt->hcnt=%d fwd_rt->seqno=%ld",
334           fwd_rt->hcnt, fwd_rt->dest_seqno);
335     }
336     return;
337     }
```

aodv_rrep.c

315 - 337 检查是否需要转发路由，如果路由表项不存在，需要插入一项。如果路由表项存在，需要更新路由表项。

aodv_rrep.c

```
342       if (rrep->a) {
343       RREP_ack *rrep_ack;
344
345       rrep_ack = rrep_ack_create();
346       aodv_socket_send((AODV_msg *) rrep_ack, fwd_rt->next_hop,
347                NET_HOP_WAIT, MAXTTL, &DEV_IFINDEX(fwd_rt->ifindex));
348       /* Remove RREP_ACK flag... */
349       rrep->a = 0;
350       }
```

aodv_rrep.c

342 - 350 如果 RREP 消息的“A”标志位被置位，必须发送 RREP-ACK 消息到目的节点。

aodv_rrep.c

```
402     if (rev_rt && rev_rt->state == VALID) {
403         rrep_forward(rrep, rreplen, rev_rt, fwd_rt, --ip_ttl);
```

aodv_rrep.c

402 - 403 沿反向路由转发 RREP 消息。

3.10 RERR 消息

3.10.1 创建 RERR 消息

rerr_create 函数用于创建 RERR 消息。

aodv_rerr.c

```
46      rerr = (RERR *) aodv_socket_new_msg();
47      rerr->type = AODV_RERR;
48      rerr->n = (flags & RERR_NODELETE ? 1 : 0);
49      rerr->res1 = 0;
50      rerr->res2 = 0;
51      rerr->dest_addr = dest_addr.s_addr;
52      rerr->dest_seqno = htonl(dest_seqno);
53      rerr->dest_count = 1;
```

aodv_rerr.c

46 - 53 为 RERR 消息格式中的属性赋初值。

3.10.2 增加不可达节点

rerr_add_udest 函数用于 RERR 消息增加不可达节点。

aodv_rerr.c

```
61      RERR_udest *ud;
62
63      ud = (RERR_udest *) ((char *) rerr + RERR_CALC_SIZE(rerr));
64      ud->dest_addr = udest.s_addr;
65      ud->dest_seqno = htonl(udest_seqno);
66      rerr->dest_count++;
```

aodv_rerr.c

61 - 66 变量 ud 用于记录不可达节点的 IP 地址、序列号。

3.10.3 处理 RERR 消息

rerr_process 函数用于处理 RERR 消息。

aodv_rerr.c

```
86      if (rerrlen < ((int) RERR_CALC_SIZE(rerr))) {
87      alog(LOG_WARNING, 0, __FUNCTION__,
88           "IP data too short (%u bytes) from %s to %s. Should be %d bytes.",
89           rerrlen, ip_to_str(ip_src), ip_to_str(ip_dst),
90           RERR_CALC_SIZE(rerr));
91
92      return;
93      }
```

aodv_rerr.c

86 - 93 检查如果 RERR 消息的长度小于规定值，放弃发送。

aodv_rerr.c

```
96      udest = RERR_UDEST_FIRST(rerr);
97
98      while (rerr->dest_count) {
99
100     udest_addr.s_addr = udest->dest_addr;
101     rerr_dest_seqno = ntohl(udest->dest_seqno);
102     DEBUG(LOG_DEBUG, 0, "unreachable dest=%s seqno=%lu",
103           ip_to_str(udest_addr), rerr_dest_seqno);
```

aodv_rerr.c

96 - 103 打印不可达节点的 IP 地址和序列号

aodv_rerr.c

```
112     if (0 && (int32_t) rt->dest_seqno > (int32_t) rerr_dest_seqno) {
113     DEBUG(LOG_DEBUG, 0, "Udest ignored because of seqno");
114     udest = RERR_UDEST_NEXT(udest);
115     rerr->dest_count--;
116     continue;
117     }
```

aodv_rerr.c

112 - 117 如果 RERR 消息中不可达节点的序列号小于路由表项中最新的对应节点的序列号，说明该消息已过期，丢弃。

aodv_rerr.c

```
125     if (!rerr->n) {
126     rt_table_invalidate(rt);
127     }
```

aodv_rerr.c

125 - 127 如果 RERR 消息中“N”标志位未被置位，说明该路由可以被删除，调用 rt_table_invalidate 函数将该路由置为无效路由。

aodv_rerr.c

```
135         if (rt->nprec && !(rt->flags & RT_REPAIR)) {
136
137         if (!new_rerr) {
138             u_int8_t flags = 0;
139
140             if (rerr->n)
141             flags |= RERR_NODELETE;
142
143             new_rerr = rerr_create(flags, rt->dest_addr,
144                                    rt->dest_seqno);
145             DEBUG(LOG_DEBUG, 0, "Added %s as unreachable, seqno=%lu",
146               ip_to_str(rt->dest_addr), rt->dest_seqno);
147
148             if (rt->nprec == 1)
149             rerr_unicast_dest =
150                 FIRST_PREC(rt->precursors)->neighbor;
```

aodv_rerr.c

135 - 150 如果先驱表不为空，将目的节点作为不可达节点存入 RERR 消息中。

aodv_rerr.c

```
176         if (rt->state == INVALID)
177         precursor_list_destroy(rt);
```

aodv_rerr.c

176 - 177 删除所有不可达节点的先驱表。

aodv_rerr.c

```
186     if (new_rerr) {
187
188     rt = rt_table_find(rerr_unicast_dest);
189
190     if (rt && new_rerr->dest_count == 1 && rerr_unicast_dest.s_addr)
191         aodv_socket_send((AODV_msg *) new_rerr,
192                  rerr_unicast_dest,
193                  RERR_CALC_SIZE(new_rerr), 1,
194                  &DEV_IFINDEX(rt->ifindex));
195
196     else if (new_rerr->dest_count > 0) {
197         /* FIXME: Should only transmit RERR on those interfaces
198          * which have precursor nodes for the broken route */
199         for (i = 0; i < MAX_NR_INTERFACES; i++) {
200         struct in_addr dest;
201
202         if (!DEV_NR(i).enabled)
203             continue;
204         dest.s_addr = AODV_BROADCAST;
205         aodv_socket_send((AODV_msg *) new_rerr, dest,
206                  RERR_CALC_SIZE(new_rerr), 1, &DEV_NR(i));
207         }
208     }
```

aodv_rerr.c

186 - 208 当有有新的 RERR 消息时，如果 dest_count 值为 1，单播该 RERR 消息；如果 dest_count 值>1，仅向有断裂路由先驱节点的接口发送 RERR 消息。

3.11 hello 消息

3.11.1 准备 hello 消息

hello_start 函数用于准备 hello 消息。

aodv_hello.c

```
62      if (hello_timer.used)
63      return;
64
65      gettimeofday(&this_host.fwd_time, NULL);
66
67      DEBUG(LOG_DEBUG, 0, "Starting to send HELLOs!");
68      timer_init(&hello_timer, &NS_CLASS hello_send, NULL);
69
70      hello_send(NULL);
```

aodv_hello.c

62 - 70 如果 hello 消息的计时器已开始使用，直接返回。否则初始化 hello 消息的计算器后发送 hello 消息。

3.11.2 停止发送 hello 消息

hello_stop 函数用于停止发送 hello 消息。

aodv_hello.c

```
75      DEBUG(LOG_DEBUG, 0,
76        "No active forwarding routes - stopped sending HELLOs!");
77      timer_remove(&hello_timer);
```

aodv_hello.c

75 - 77 当当前节点没有到相邻节点的活跃路由是停止发送 hello 消息，移除 hello 消息计时器。

3.11.3 发送 hello 消息

hello_send 函数用于发送 hello 消息。

aodv_hello.c

```
93      if (optimized_hellos &&
94      timeval_diff(&now, &this_host.fwd_time) > ACTIVE_ROUTE_TIMEOUT) {
95      hello_stop();
96      return;
97      }
```

aodv_hello.c

93 - 97 如果当前时间与最后一次发送 hello 消息的间隔大于活跃路由规定时间，说明当前节点没有活跃路由到邻居节点，调用 hello_stop 函数。

aodv_hello.c

```
 99      time_diff = timeval_diff(&now, &this_host.bcast_time);
100      jitter = hello_jitter();
101
102      /* This check will ensure we don't send unnecessary hello msgs, in case
103         we have sent other bcast msgs within HELLO_INTERVAL */
104      if (time_diff >= HELLO_INTERVAL) {
105
106      for (i = 0; i < MAX_NR_INTERFACES; i++) {
107          if (!DEV_NR(i).enabled)
108          continue;
```

aodv_hello.c

99 计算当前时间与最后一次广播时间的时间差
104 - 108 比较计算所得时间差与 hello 消息周期的大小，确保不会重复发送 hello 消息。

3.11.4 处理 hello 消息

hello_process 函数用于处理 hello 消息。

aodv_hello.c

```
241      if (receive_n_hellos)
242      state = INVALID;
243      else
244      state = VALID;
```

aodv_hello.c

241 - 244 邻居节点只有连续收到 3 个 hello 消息才有效，设置 state 值。

aodv hello.c

```
248      if (!rt) {
249      /* No active or expired route in the routing table. So we add a
250         new entry... */
251
252      rt = rt_table_insert(hello_dest, hello_dest, 1,
253                  hello_seqno, timeout, state, flags, ifindex);
254
255      if (flags & RT_UNIDIR) {
256          DEBUG(LOG_INFO, 0, "%s new NEIGHBOR, link UNI-DIR",
257            ip_to_str(rt->dest_addr));
258      } else {
259          DEBUG(LOG_INFO, 0, "%s new NEIGHBOR!", ip_to_str(rt->dest_addr));
260      }
261      rt->hello_cnt = 1;
```

aodv_hello.c

248 - 261 当在路由表中没有一个活跃或过期的路由表项时，要调用 rt_table_insert 函数创建新的路由表项。

3.12 邻居节点处理

3.12.1 增加/更新邻居节点

neighbor_add 函数用于增加/更新邻居节点。

aodv_neighbor.c

```
49      rt = rt_table_find(source);
50
51      if (!rt) {
52      DEBUG(LOG_DEBUG, 0, "%s new NEIGHBOR!", ip_to_str(source));
53      rt = rt_table_insert(source, source, 1, 0,
54                  ACTIVE_ROUTE_TIMEOUT, VALID, 0, ifindex);
55      } else {
56      /* Don't update anything if this is a uni-directional link... */
57      if (rt->flags & RT_UNIDIR)
58          return;
59
60      if (rt->dest_seqno != 0)
61          seqno = rt->dest_seqno;
62
63      rt_table_update(rt, source, 1, seqno, ACTIVE_ROUTE_TIMEOUT,
64              VALID, rt->flags);
65      }
```

aodv_neighbor.c

49 - 55 检查 source 指向的的结点在路由表项中是否存在，不存在调用 rt_table_insert 函数添加新的路由表项。

57 - 58 如果是单向链路不做任何操作。

aodv_neighbor.c

```
67      if (!llfeedback && rt->hello_timer.used)
68      hello_update_timeout(rt, &now, ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
```

aodv_neighbor.c

67 - 68 如果一个路由已经存在，那么这条路由的生命期应该增加，如果必要，至少为 ALLOWED_HELLO_LOSS * HELLO_INTERVAL。

3.12.2 断开邻居节点连接

neighbor_link_break 函数用于断开邻居节点连接。

aodv_neighbor.c

```
84      if (!rt)
85      return;
86
87      if (rt->hcnt != 1) {
88      DEBUG(LOG_DEBUG, 0, "%s is not a neighbor, hcnt=%d!!!",
89            ip_to_str(rt->dest_addr), rt->hcnt);
90      return;
91      }
```

aodv_neighbor.c

84 - 91 如果路由表为空或跳数不等于 1，直接返回。

aodv_neighbor.c

```
 99      if (rt->nprec && !(rt->flags & RT_REPAIR)) {
100      rerr = rerr_create(0, rt->dest_addr, rt->dest_seqno);
101      DEBUG(LOG_DEBUG, 0, "Added %s as unreachable, seqno=%lu",
102            ip_to_str(rt->dest_addr), rt->dest_seqno);
103
104      if (rt->nprec == 1)
105          rerr_unicast_dest = FIRST_PREC(rt->precursors)->neighbor;
106      }
```

aodv_neighbor.c

99 -106 创建 RERR 消息，设置单播的目的结点。

aodv_neighbor.c

```
116      for (i = 0; i < RT_TABLESIZE; i++) {
117      list_t *pos;
118      list_foreach(pos, &rt_tbl.tbl[i]) {
119          rt_table_t *rt_u = (rt_table_t *) pos;
120
121          if (rt_u->state == VALID &&
122          rt_u->next_hop.s_addr == rt->dest_addr.s_addr &&
123          rt_u->dest_addr.s_addr != rt->dest_addr.s_addr) {
124
125          /* If the link that broke are marked for repair,
126             then do the same for all additional unreachable
127             destinations. */
128
129          if ((rt->flags & RT_REPAIR) && rt_u->hcnt <= MAX_REPAIR_TTL
130
131              rt_u->flags |= RT_REPAIR;
132              DEBUG(LOG_DEBUG, 0, "Marking %s for REPAIR",
133                ip_to_str(rt_u->dest_addr));
134
135              rt_table_invalidate(rt_u);
136              continue;
137          }
138
139          rt_table_invalidate(rt_u);
140
141          if (rt_u->nprec) {
142
143              if (!rerr) {
144              rerr =
145                  rerr_create(0, rt_u->dest_addr, rt_u->dest_seqno);
146
147              if (rt_u->nprec == 1)
148                  rerr_unicast_dest =
149                  FIRST_PREC(rt_u->precursors)->neighbor;
150
151              DEBUG(LOG_DEBUG, 0,
152                    "Added %s as unreachable, seqno=%lu",
153                    ip_to_str(rt_u->dest_addr), rt_u->dest_seqno);
```

aodv_neighbor.c

116 - 153 查找路由表中下一跳是不可达节点的表项，这些表项要被加入到 RERR 消息中，如果不可达节点被修复，下一跳为不可达节点的表项也要被修复。

aodv_neighbor.c

```
194         for (i = 0; i < MAX_NR_INTERFACES; i++) {
195         struct in_addr dest;
196
197         if (!DEV_NR(i).enabled)
198             continue;
199         dest.s_addr = AODV_BROADCAST;
200         aodv_socket_send((AODV_msg *) rerr, dest,
201                  RERR_CALC_SIZE(rerr), 1, &DEV_NR(i));
202         }
```

aodv_neighbor.c

194 - 202 仅向有断裂路由先驱节点的接口发送 RERR 消息。

3.13 超时管理

3.13.1 路由发现超时

route_discovery_timeout 函数用于处理路由发现超时。

aodv_timeout.c

```
67      if (seek_entry->reqs < RREQ_RETRIES) {
68
69          if (expanding_ring_search) {
70
71              if (TTL_VALUE < TTL_THRESHOLD)
72                  TTL_VALUE += TTL_INCREMENT;
73              else {
74                  TTL_VALUE = NET_DIAMETER;
75                  seek_entry->reqs++;
76              }
77              /* Set a new timer for seeking this destination */
78              timer_set_timeout(&seek_entry->seek_timer,
79                        RING_TRAVERSAL_TIME);
80          } else {
81              seek_entry->reqs++;
82              timer_set_timeout(&seek_entry->seek_timer,
83                        seek_entry->reqs * 2 *
84                        NET_TRAVERSAL_TIME);
85          }
86          /* AODV should use a binary exponential backoff RREP waiting
87             time. */
88          DEBUG(LOG_DEBUG, 0, "Seeking %s ttl=%d wait=%d",
89                ip_to_str(seek_entry->dest_addr),
90                TTL_VALUE, 2 * TTL_VALUE * NODE_TRAVERSAL_TIME);
91
92          /* A routing table entry waiting for a RREP should not be expunged
93             before 2 * NET_TRAVERSAL_TIME... */
94          rt = rt_table_find(seek_entry->dest_addr);
95
96          if (rt && timeval_diff(&rt->rt_timer.timeout, &now) <
97              (2 * NET_TRAVERSAL_TIME))
98              rt_table_update_timeout(rt, 2 * NET_TRAVERSAL_TIME);
99
100         rreq_send(seek_entry->dest_addr, seek_entry->dest_seqno,
101               TTL_VALUE, seek_entry->flags);
102
103     } else {
104
105         DEBUG(LOG_DEBUG, 0, "NO ROUTE FOUND!");
```

aodv_timeout.c

67 - 105 设置新的计时器记录路由发现所用时间，对每一次的尝试，等待时间都应当是上一次尝试时间的 2 倍。这样就从等待时间上保证遵从了网络协议的二的幂次递减原则。将时间限制设置为 2*NET_TRAVERSAL_TIME，如果仍然超时返回警告没有发现路由。

aodv_timeout.c

```
118         if (repair_rt && (repair_rt->flags & RT_REPAIR)) {
119             DEBUG(LOG_DEBUG, 0, "REPAIR for %s failed!",
120                   ip_to_str(repair_rt->dest_addr));
121             local_repair_timeout(repair_rt);
```

aodv_timeout.c

118 - 121 如果路由正处于修复状态，需要立刻告知超时消息，停止修复。

3.13.2 本地修复超时

local_repair_timeout 函数用于处理本地修复超时。

aodv_timeout.c

```
140     rt->flags &= ~RT_REPAIR;
141
142 #ifndef NS_PORT
143     nl_send_del_route_msg(rt->dest_addr, rt->next_hop, rt->hcnt)
144 #endif
145     /* Route should already be invalidated. */
146
147     if (rt->nprec) {
148
149         rerr = rerr_create(0, rt->dest_addr, rt->dest_seqno);
150
151         if (rt->nprec == 1) {
152             rerr_dest = FIRST_PREC(rt->precursors)->neighbor;
153
154             aodv_socket_send((AODV_msg *) rerr, rerr_dest,
155                      RERR_CALC_SIZE(rerr), 1,
156                      &DEV_IFINDEX(rt->ifindex));
157         } else {
158             int i;
159
160             for (i = 0; i < MAX_NR_INTERFACES; i++) {
161                 if (!DEV_NR(i).enabled)
162                     continue;
163                 aodv_socket_send((AODV_msg *) rerr, rerr_dest,
164                          RERR_CALC_SIZE(rerr), 1,
165                          &DEV_NR(i));
166             }
167         }
168         DEBUG(LOG_DEBUG, 0, "Sending RERR about %s to %s",
169               ip_to_str(rt->dest_addr), ip_to_str(rerr_dest));
170     }
171     precursor_list_destroy(rt);
```

aodv_timeout.c

140 - 171 设置路由的标志位为 REPAIR，将该路由设置为失效，创建 RERR 消息并广播到其它节点。

aodv_timeout.c

```
176     rt->rt_timer.handler = &NS_CLASS route_delete_timeout;
177     timer_set_timeout(&rt->rt_timer, DELETE_PERIOD);
178
179     DEBUG(LOG_DEBUG, 0, "%s removed in %u msecs",
180           ip_to_str(rt->dest_addr), DELETE_PERIOD);
```

aodv_timeout.c

176 - 180 如果修复超时，清除队列中所有可能处于修复状态的数据包。

3.13.3 路由到期超时

route_expire_timeout 函数用于处理路由到期超时。

aodv_timeout.c

```
190     if (!rt) {
191         alog(LOG_WARNING, 0, __FUNCTION__,
192              "arg was NULL, ignoring timeout!");
193         return;
194     }
195
196     DEBUG(LOG_DEBUG, 0, "Route %s DOWN, seqno=%d",
197           ip_to_str(rt->dest_addr), rt->dest_seqno);
198
199     if (rt->hcnt == 1)
200         neighbor_link_break(rt);
201     else {
202         rt_table_invalidate(rt);
203         precursor_list_destroy(rt);
204     }
```

aodv_timeout.c

190 - 204 如果一条路由信息已到期，将路由设置为无效，并在先驱表中删除对应项。

3.13.4 路由删除超时

route_delete_timeout 函数用于处理路由删除超时。

aodv_timeout.c

```
216     if (!rt)
217         return;
218
219     DEBUG(LOG_DEBUG, 0, "%s", ip_to_str(rt->dest_addr));
220
221     rt_table_delete(rt);
```

aodv_timeout.c

216 - 221 从路由表中删除该路由。

3.13.5 hello 消息超时

hello_timeout 函数用于处理 hello 消息超时。

aodv_timeout.c

```
242     if (rt && rt->state == VALID && !(rt->flags & RT_UNIDIR)) {
243
244         /* If the we can repair the route, then mark it to be
245            repaired.. */
246         if (local_repair && rt->hcnt <= MAX_REPAIR_TTL) {
247             rt->flags |= RT_REPAIR;
248             DEBUG(LOG_DEBUG, 0, "Marking %s for REPAIR",
249                   ip_to_str(rt->dest_addr));
250 #ifdef NS_PORT
251             /* Buffer pending packets from interface queue */
252             interfaceQueue((nsaddr_t) rt->dest_addr.s_addr,
253                            IFQ_BUFFER);
254 #endif
255         }
256         neighbor_link_break(rt);
```

aodv_timeout.c

242 - 256 如果可以修复该路由，将其置为 REPAIRED，并将其断开。

3.13.6 RREP-ACK 消息超时

rrep_ack_timeout 函数用于处理 RREP-ACK 消息超时。

aodv_timeout.c

```
266     rt = (rt_table_t *) arg;
267
268     if (!rt)
269         return;
270
271     /* When a RREP transmission fails (i.e. lack of RREP-ACK), add to
272        blacklist set... */
273     rreq_blacklist_insert(rt->dest_addr);
274
275     DEBUG(LOG_DEBUG, 0, "%s", ip_to_str(rt->dest_addr));
```

aodv_timeout.c

266 - 275 发送 RREP 消息确认超时，如果发送失败，将目的结点拉入黑名单。

第四章 总结

我们在之前没有经历过相似的项目，在大作业刚刚开始进行的时候，我们不知从何入手，看不懂代码。经过很多次的尝试，我们结合 RFC 和代码，将 RFC 中提到的功能、流程与代码中的实现函数对应起来，再根据代码中的注释，就慢慢地理解更多的代码，并将各个源代码文件结合起来，对 AODV 无线路由协议的结构更加清楚。

AODV 无线路由协议的基本算法是距离向量路由算法，在此算法基础上进行了改进，解决了许多传统协议中存在的问题。思路简单，易于理解。AODV 作为一种按需路由协议，节点只存储需要的路由表项，减少了空间需求。而且 AODV 协议支持中间节点应答，加快了建立路由的速度，减少了广播数。

在分析过程中，我们也能够很轻易地发现，AODV 的源代码结构清楚，逻辑严谨，对异常情况的处理很完整。代码中的函数和变量命名很清晰，能够根据函数的名字就了解到函数大概的功能。

我们也了解到，AODV 协议路由选择是选择最短路径路由，选择最小跳数的路由，但却忽略了两点之间的传输能力，导致链路吞吐率低、路由不稳定、线路拥塞、延迟甚至数据丢失的问题。随着网络的不断发展和路由协议的不断改进，相信这些问题都能够得到解决，并且性能不断提高。