网络协议栈分析与设计课程大作业

AODV 路由协议代码分析

Group 15

姓名	班级	学号	任务分配	成绩
张天尧	软网 1604	201693082	搜集相关的资料，整体代码查看以及对 AODV 的几类报文进行分析以及代码分析撰写	
孙修文	软网 1604	201692214	文档的编写，内核文件、超时管理、日志文件等其他文件阅读，写相应的代码分析	

大连理工大学

Dalian University of Technology

1 概述

The Ad hoc On-Demand Distance Vector (AODV) 路由协议旨在供 ad hoc 网络中的移动节点使用。它可以快速适应动态链路条件，具有低处理和内存开销，低网络利用率的特点。同时可以确定到 ad hoc 网络内目的地的单播路由。 它使用目标序列号来确保循环自由（即使面对路由控制消息的异常传送），也避免了与经典距离矢量协议相关的问题（例如“计数到无穷大”）。

1.1 协议简介

路由请求（RREQ），路由回复（RREP）和路由错误（RERR）是 AODV 定义的消息类型。这些消息类型通过 UDP 接收，并应用正常的 IP 报头处理。因此，例如，期望请求节点使用其 IP 地址作为消息的发起者 IP 地址。对于广播消息，使用 IP 受限广播地址（255.255.255.255）。这意味着不会盲目转发此类消息。然而，AODV 操作确实需要广泛地传播某些消息（例如，RREQ），可能在整个 ad hoc 网络中传播。 IP 报头中的 TTL 表示这种 RREQ 的传播范围。通常不需要碎片。

只要通信连接的端点具有相互有效的路由，AODV 就不起任何作用。当需要到新目的地的路线时，节点广播 RREQ 以找到目的地的路线。当 RREQ 到达目的地本身或具有到目的地的“足够新鲜”路线的中间节点时，可以确定路线。“足够新鲜”路线是目的地的有效路线条目，其相关序列号至少与 RREQ 中包含的序列号一样大。通过将 RREP 单播回到 RREQ 的起源，可以获得该路线。接收请求的每个节点将路由高速缓存回请求的发起者，使得 RREP 可以沿着到该发起者的路径从目的地单播，或者同样地从能够满足该请求的任何中间节点单播。

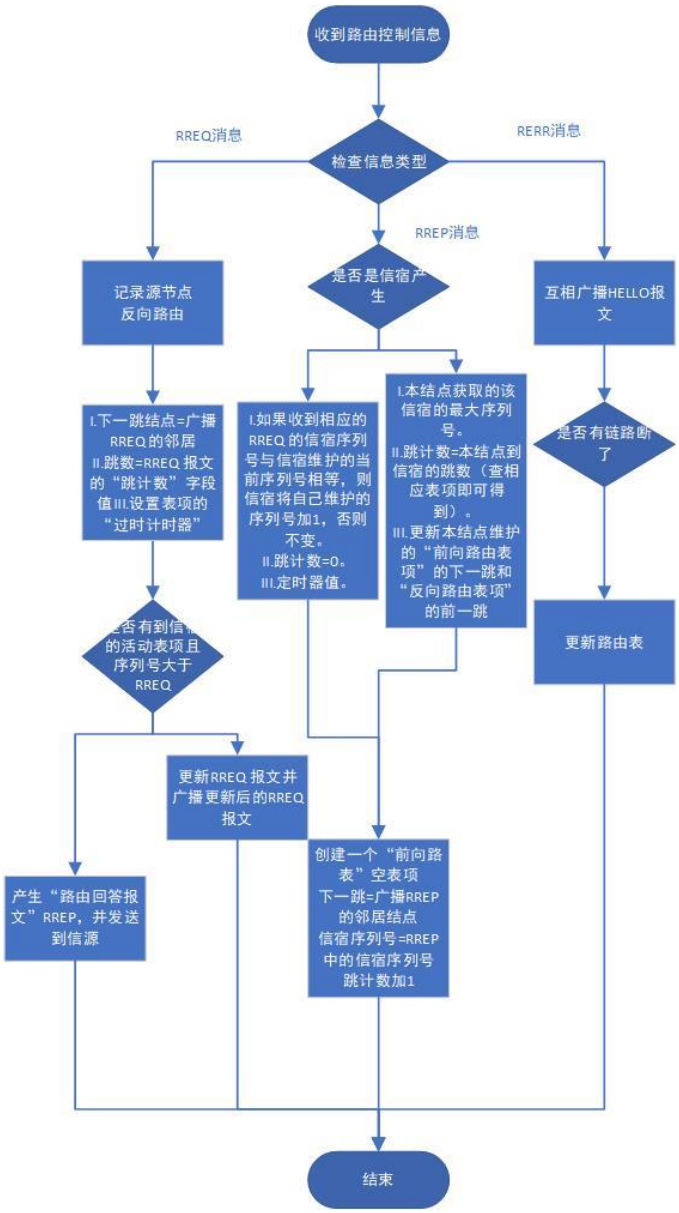
节点监视活动路由中下一跳的链路状态。当检测到活动路由中的链路中断时，RERR 消息用于通知其他节点已发生该链路的丢失。 RERR 消息指示通过断开的链路不再可到达的那些目的地（可能是子网）。为了启用此报告机制，每个节点都保留一个“前导列表”，其中包含每个邻居的 IP 地址，这些邻居很可能将其用作每个目的地的下一跳。在生成 RREP 消息的处理过程中，最容易获取前体列表中的信息，根据定义，该消息必须发送到前体列表中的节点。如果 RREP 具有非零前缀然后，请求 RREP 信息的 RREQ 的发起者包括在子网路由的前体（不是特定于特定目的地）中。还可以接收针对多播 IP 地址的 RREQ。在本文档中，未指定此类消息的完整处理。例如，用于多播 IP 地址的这种 RREQ 的发起者可能必须遵循特殊规则。但是，重要的是通过未启用作为 IP 多播地址的始发节点或目标节点的中间节点启用正确的多播操作，并且同样不适用于任何特殊的多播协议处理。对于这样的多播不知道节点，必须以与任何其他目的地 IP 地址相同的方式执行作为目的地 IP 地址的多播 IP 地址的处理。

AODV 是一种路由协议，它处理路由表管理。即使对于短期路由，也必须保留路由表信息，例如创建用于临时存储发往 RREQ 的节点的反向路径。 AODV 对每个路由表条目使用以下字段：

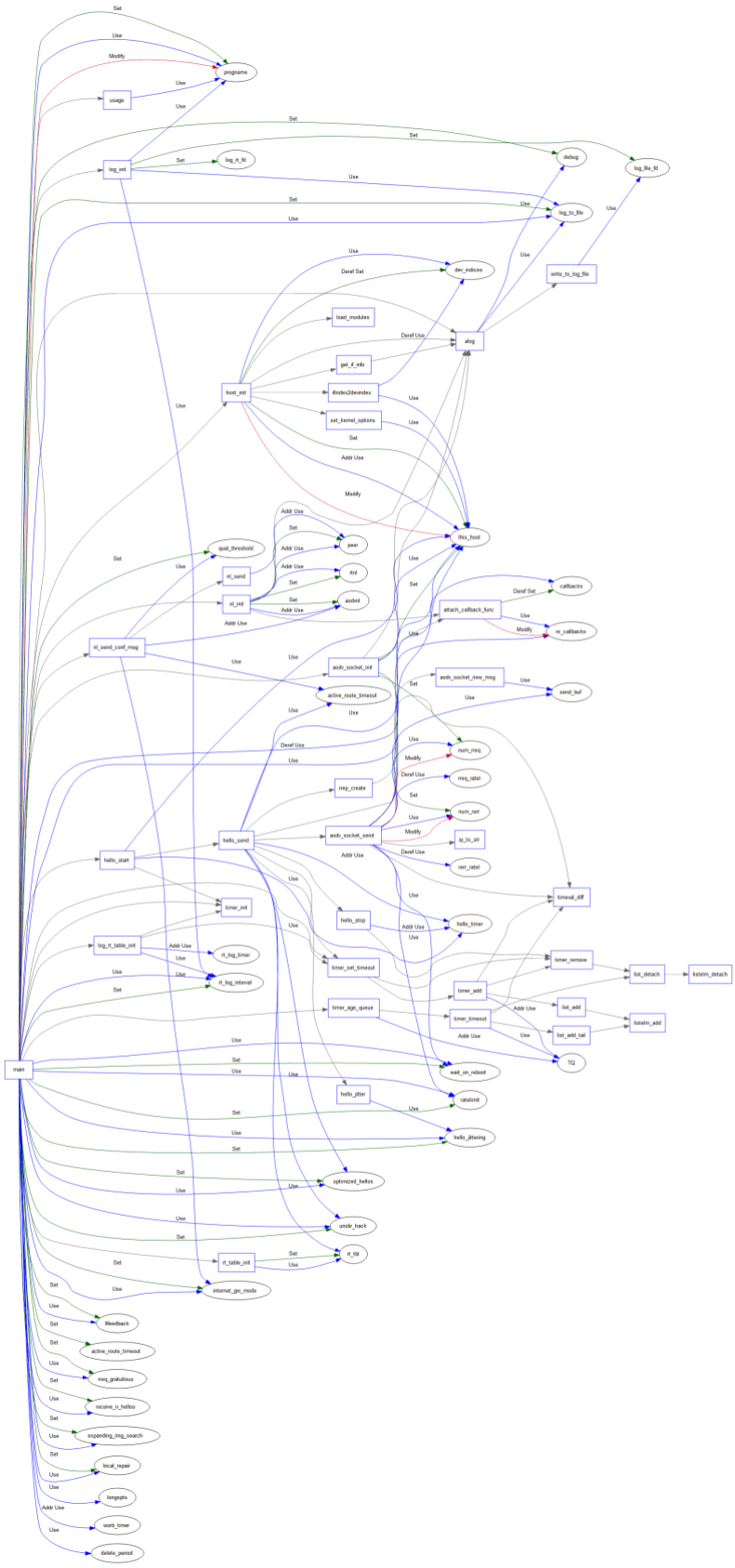
目的地 IP 地址 - 目的地序列号 - 有效目的地序列号标志 - 其他状态和路由标志（例如，有效，无效，可修复，正在修复） - 网络接口 - 跳数（到达目的地所需的跳数） - 下一跳 - 前体列表（在 6.2 节中描述） - 生命周期（路线的到期或删除时间）

管理序列号对于避免路由环路至关重要，即使链路断开且节点不再可达，也无法提供有关其序列号的信息。链接中断或取消激活时，目标无法访问。发生这些情况时，路由将通过涉及序列号的操作无效，并将路由表条目状态标记为无效。

1.2 具体流程

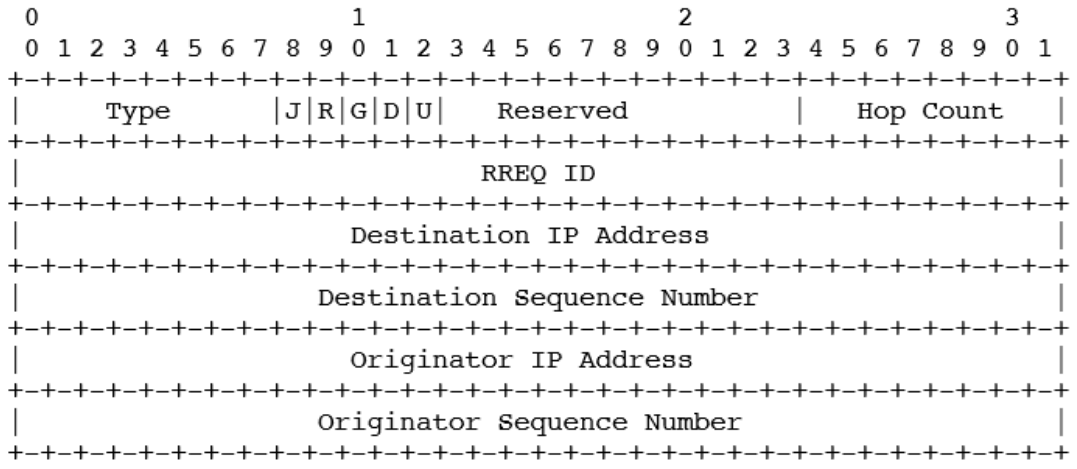


1.3 main 函数的函数调用图



2 报文格式及传输过程

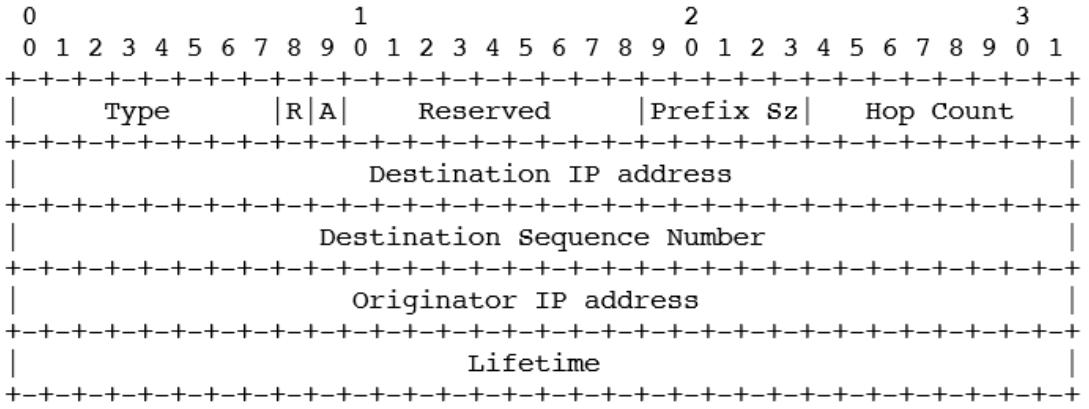
2.1 RREQ 报文格式



Type	**1**
J	加入标志，为多播保留
R	修理标志，为多播保留
G	免费路由回复标志；指示免费路由回复是否应单播到目标 IP 地址字段中指定的节点
D	目的地唯一标志；表示只有目的地可以响应此路由请求
U	未知序列号；表示目标序列号未知
Reserved	为 0，接收端忽略此字段
Hop Count	从发起方 IP 地址到处理请求的节点的跃点数。
RREQ ID	与起始节点的 IP 地址可以唯一标识特定路由请求的序列号
Destination IP Address	目的节点的 IP 地址
Destination Sequence Number	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
Originator IP Address	发起节点 IP 地址

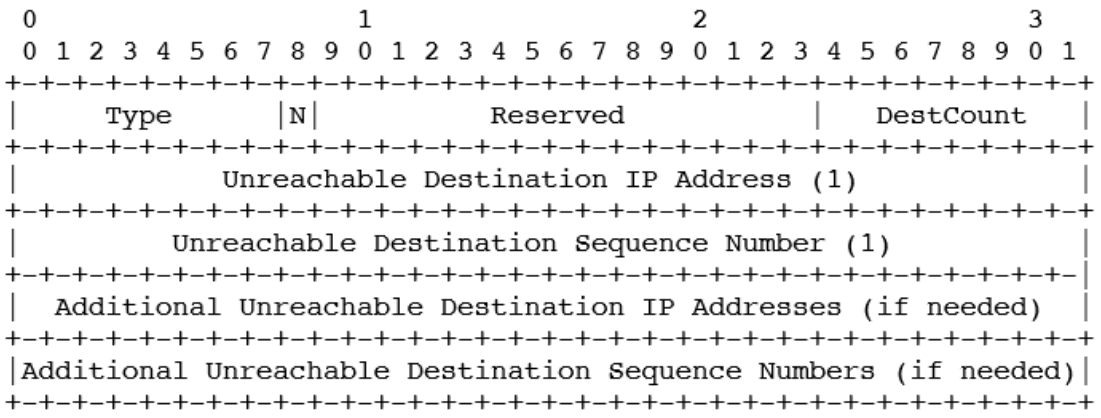
Originator Sequence Number	发起节点序列号

2.2 RREP 报文格式



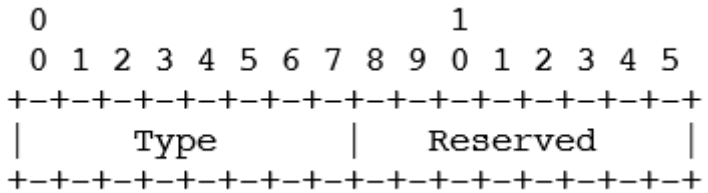
Type	**2**
R	修理标志，为多播保留
A	RREP-ACK
Reserved	为 0，接收端忽略此字段
Prefix Size	前缀长度。这个字段为 5 个 bit（值为 0 到 31），如果非 0， 则代表下一跳节点可以作为任何具有相同路由前缀的节点被请求时的目的节点
Hop Count	从发起节点到目标节点的跳数。对多播路由请求， 这个跳数则是从发起节点到多播节点组里产生 RREP 信息的节点的跳数
Destination IP Address	目标节点的 IP 地址
Destination Sequence Number	相关的目标序列号
Originator IP Address	发起 RREQ 消息的节点的 IP 地址
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

2.3 RERR 报文格式



Type	**3**
N	没有删除标志；当节点已执行链路的本地修复时设置，并且上游节点不应删除该路由
Reserved	为 0，接收端忽略此字段
DestCount	消息中包含的无法到达目的地的数量；必须至少为 1
Unreachable Destination IP Address	由于链接中断而无法访问的目标的 IP 地址
Unreachable Destination Sequence Number	“Unreachable Destination IP Address”字段中列出的目标的路由表条目中的序列号

2.4 RREP-ACK 报文格式



Type	**4**
Reserved	Sent as 0; ignored on reception

3 代码分析

一些比较简单的文件

文件名	作用

Endian.c	判断大小端
List.c	链表的增删改查操作
timer_queue.c	时间队列的增删
seek_list.c	RREQ 寻找目的地链表的增改
locality.c	确认地址
nl.c	套接字的 api 函数

aodv_socket 部分

```
55 /* Seems that some libc (for example ulibc) has a bug in the provided
56  * CMSG_NXTHDR() routine... redefining it here */
57
58 static struct cmsghdr *__cmsg_nxthdr_fix(void *__ctl, size_t __size,
59                                          struct cmsghdr *__cmsg)
60 {
61     struct cmsghdr *__ptr;
62
63     __ptr = (struct cmsghdr *) (((unsigned char *) __cmsg) +
64                                 CMSG_ALIGN(__cmsg->cmsg_len));
65     if ((unsigned long) ((char *) (__ptr + 1) - (char *) __ctl) > __size)
66         return NULL;
67
68     return __ptr;
69 }
70
71 struct cmsghdr *cmsg_nxthdr_fix(struct msghdr *__msg, struct cmsghdr *__cmsg)
72 {
73     return __cmsg_nxthdr_fix(__msg->msg_control, __msg->msg_controllen, __cmsg);
74 }
```

58-74

两个版本的对与 libc 中提供的 CMSG_NXTHDR 函数的修复

79-209

初始化 socket

```
90      /* Create a UDP socket */
91
92      if (this_host.nif == 0) {
93          fprintf(stderr, "No interfaces configured\n");
94          exit(-1);
95      }
```

92-95

创建 udp 套接字

```
97      /* Open a socket for every AODV enabled interface */
98      for (i = 0; i < MAX_NR_INTERFACES; i++) {
99          if (!DEV_NR(i).enabled)
100             continue;
```

98-100

为每一个允许 aodv 的接口开启一个套接字

```
102     /* AODV socket */
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     }
```

103-107

创建 aodv 套接字

```
117     /* Bind the socket to the AODV port number */
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);
122
123     retval = bind(DEV_NR(i).sock, (struct sockaddr *) &aodv_addr,
124             sizeof(struct sockaddr));
```

118-124

将 aodv 端口号与套接字绑定

```
126     if (retval < 0) {
127         perror("Bind failed ");
128         exit(-1);
129     }
130     if (setsockopt(DEV_NR(i).sock, SOL_SOCKET, SO_BROADCAST,
131             &on, sizeof(int)) < 0) {
132         perror("SO_BROADCAST failed ");
133         exit(-1);
134     }
135
136     memset(&ifr, 0, sizeof(struct ifreq));
137     strcpy(ifr.ifr_name, DEV_NR(i).ifname);
138
139     if (setsockopt(DEV_NR(i).sock, SOL_SOCKET, SO_BINDTODEVICE,
140             &ifr, sizeof(ifr)) < 0) {
141         fprintf(stderr, "SO_BINDTODEVICE failed for %s", DEV_NR(i).ifname);
142         perror(" ");
143         exit(-1);
144     }
145
146     if (setsockopt(DEV_NR(i).sock, SOL_SOCKET, SO_PRIORITY,
147             &tos, sizeof(int)) < 0) {
148         perror("Setsockopt SO_PRIORITY failed ");
149         exit(-1);
150     }
151
152     if (setsockopt(DEV_NR(i).sock, SOL_IP, IP_RECVTTL,
153             &on, sizeof(int)) < 0) {
154         perror("Setsockopt IP_RECVTTL failed ");
155         exit(-1);
156     }
157
158     if (setsockopt(DEV_NR(i).sock, SOL_IP, IP_PKTINFO,
159             &on, sizeof(int)) < 0) {
160         perror("Setsockopt IP_PKTINFO failed ");
161         exit(-1);
162     }
```

126-162

设置并检查套接字选项，设置失败就返回错误中止退出

```
183   /* Set max allowable receive buffer size... */
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   }
```

183-196

设置最大允许缓冲区大小

```
211 void NS_CLASS aodv_socket_process_packet(AODV_msg * aodv_msg, int len,
212                                         struct in_addr src,
213                                         struct in_addr dst,
214                                         int ttl, unsigned int ifindex)
215 {
216
217     /* If this was a HELLO message... Process as HELLO. */
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     }
223
224     /* Make sure we add/update neighbors */
225     neighbor_add(aodv_msg, src, ifindex);
226
227     /* Check what type of msg we received and call the corresponding
228        function to handle the msg... */
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 }
```

211-251

套接字处理

218-222

如果是 hello 消息，就使用 hello 消息处理函数

225

需要确保添加/更新了邻居节点信息

229-250

分别处理 rreq、rrep、rerr、rrep_ack 消息，并记录日志内容

```
254 void NS_CLASS recvAODVUUPacket(Packet * p)
255 {
256     int len, i, ttl = 0;
257     struct in_addr src, dst;
258     struct hdr_cmn *ch = HDR_CMN(p);
259     struct hdr_ip *ih = HDR_IP(p);
260     hdr_aodvuu *ah = HDR_AODVUU(p);
261 
262     src.s_addr = ih->saddr();
263     dst.s_addr = ih->daddr();
264     len = ch->size() - IP_HDR_LEN;
265     ttl = ih->ttl();
266 
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);
280 
281     /* Ignore messages generated locally */
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 }
```

254-289

处理收到的数据包

256-265

分配缓存空间

269-279

分类处理数据包

282-286

忽略本地数据包

288

调用套接字处理函数进行处理

aodv_rreq 部分

aodv_rreq.h

```
33  /* RREQ Flags: */
34  #define RREQ_JOIN           0x1
35  #define RREQ_REPAIR         0x2
36  #define RREQ_GRATUITOUS     0x4
37  #define RREQ_DEST_ONLY      0x8
38
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;
```

33-63

rreq 报文定义段

变量	含义	长度
type	Type	8
res1	Flags	8
res2	Reserved	8
hcht	Hop Count	8
rreq_id	RREQ ID	32
dest_addr	Destination IP Address	32

dest_seqno	Destination Sequence Number	32
orig_addr	Originator IP Address	32
orig_seqno	Originator Sequence Number	32

34-37

同时，还在 define 处定义了 J、R、G、D 四个 flag，分别为 0001、0010、0100、1000。

```
67  /* A data structure to buffer information about received RREQ's */
68  struct rreq_record {
69      list_t l;
70      struct in_addr orig_addr;   /* Source of the RREQ */
71      u_int32_t rreq_id;      /* RREQ's broadcast ID */
72      struct timer rec_timer;
73  };
74
75  struct blacklist {
76      list_t l;
77      struct in_addr dest_addr;
78      struct timer bl_timer;
79  };
```

67-79

定义了储存 rreq 报文的结构体以及黑名单节点。

---

aodv_rreq.c

```
61  RREQ *NS_CLASS rreq_create(u_int8_t flags, struct in_addr dest_addr,
62  ▸   ▸   ▸       u_int32_t dest_seqno, struct in_addr orig_addr)
63  {
64      RREQ *rreq;
65
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;
89
90      DEBUG(LOG_DEBUG, 0, "Assembled RREQ %s", ip_to_str(dest_addr));
91  #ifdef DEBUG_OUTPUT
92      log_pkt_fields((AODV_msg *) rreq);
93  #endif
94
95      return rreq;
96  }
```

61-96

创建 rreq 消息包

78-79

增大自身报文序列号，以产生 rreq 洪范

```
116 void NS_CLASS rreq_send(struct in_addr dest_addr, u_int32_t dest_seqno,
117                         int ttl, u_int8_t flags)
118 {
119     RREQ *rreq;
120     struct in_addr dest;
121     int i;
122
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 }
```

116-136

rreq 消息包发送

126-127

发送消息包前检查是否需要将 G 置为 1，也就是需不需要向目标节点 IP 地址域发送一个免费路由回复消息

130-135

向自身节点的所有端口发送 rreq 消息包

```
138 void NS_CLASS rreq_forward(RREQ * rreq, int size, int ttl)
139 {
140     struct in_addr dest, orig;
141     int i;
142 
143     dest.s_addr = AODV_BROADCAST;
144     orig.s_addr = rreq->orig_addr;
145 
146     /* FORWARD the RREQ if the TTL allows it. */
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));
161     }
162 }
```

138-162

rreq 消息包的转发

147-161

如果 TTL 值大于 0，则转发 rreq 消息包

151-153

将 rreq 消息包放进消息队列中，并增加 Hop Count 跳数

157-161

向所有端口广播 rreq 消息包

164-416

处理 rreq 报文

```
203     /* Check if the previous hop of the RREQ is in the blacklist set. If
204        it is, then ignore the RREQ. */
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);
```

205-208

如果该 rreq 的上一跳节点在自身节点的黑名单中，则忽略该 rreq 消息包

可以有效防止网络恶意攻击

211-212

如果该 rreq 消息包已经被处理过，则跳过

216

将该 rreq 消息包放在缓存中

如果在路由发现阶段收到同样的 rreq 消息包，则可以忽略（211-212）

219

检查是否有 rreq 的扩展消息

```
239     /* The node always creates or updates a REVERSE ROUTE entry to the
240        source of the RREQ. */
241     rev_rt = rt_table_find(rreq_orig);
242
243     /* Calculate the extended minimal life time. */
244     life = PATH_DISCOVERY_TIME - 2 * rreq_new_hcnt * NODE_TRAVERSAL_TIME;
245
246     if (rev_rt == NULL) {
247     DEBUG(LOG_DEBUG, 0, "Creating REVERSE route entry, RREQ orig: %s",
248           ip_to_str(rreq_orig));
249
250     rev_rt = rt_table_insert(rreq_orig, ip_src, rreq_new_hcnt,
251                              rreq_orig_seqno, life, VALID, 0, ifindex);
252     } else {
253     if (rev_rt->dest_seqno == 0 ||
254         (int32_t) rreq_orig_seqno > (int32_t) rev_rt->dest_seqno ||
255         (rreq_orig_seqno == rev_rt->dest_seqno &&
256          (rev_rt->state == INVALID || rreq_new_hcnt < rev_rt->hcnt))) {
257         rev_rt = rt_table_update(rev_rt, ip_src, rreq_new_hcnt,
258                                  rreq_orig_seqno, life, VALID,
259                                  rev_rt->flags);
260     }
```

239-259

中间节点更新/创建到原节点的路由，即反向路由。

246-252

rreq 消息包中无原节点，则创建一跳新的反向路由

252-260

rreq 消息包包含原节点，反向更新路由表

中间节点如果有到目的节点的路由时，只有该节点记录的目的节点序列号比 rreq 中的目的节点序列号更大时，才认为这条路由是有效的，才会去更新路由表。

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

    /* WE are the RREQ DESTINATION. Update the node's own
       sequence number to the maximum of the current seqno and the
       one in the RREQ. */
    if (rreq_dest_seqno != 0) {
        if ((int32_t) this_host.seqno < (int32_t) rreq_dest_seqno)
        this_host.seqno = rreq_dest_seqno;
        else if (this_host.seqno == rreq_dest_seqno)
        seqno_incr(this_host.seqno);
    }
    rrep = rrep_create(0, 0, 0, DEV_IFINDEX(rev_rt->ifindex).ipaddr,
                this_host.seqno, rev_rt->dest_addr,
                MY_ROUTE_TIMEOUT);

    rrep_send(rrep, rev_rt, NULL, RREP_SIZE);

    } else {
    /* We are an INTERMEDIATE node. - check if we have an active
     * route entry */

    fwd_rt = rt_table_find(rreq_dest);

    if (fwd_rt && fwd_rt->state == VALID && !rreq->d) {
        struct timeval now;
        u_int32_t lifetime;

        /* GENERATE RREP, i.e we have an ACTIVE route entry that is fresh
           enough (our destination sequence number for that route is
           larger than the one in the RREQ). */

        gettimeofday(&now, NULL);
#ifdef CONFIG_GATEWAY_DISABLED
        if (fwd_rt->flags & RT_INET_DEST) {
        rt_table_t *gw_rt;
        /* This node knows that this is a rreq for an Internet
         * destination and it has a valid route to the gateway */

        goto forward;   // DISABLED

        gw_rt = rt_table_find(fwd_rt->next_hop);

        if (!gw_rt || gw_rt->state == INVALID)
            goto forward;

        lifetime = timeval_diff(&gw_rt->rt_timer.timeout, &now);

        rrep = rrep_create(0, 0, gw_rt->hcnt, gw_rt->dest_addr,
                    gw_rt->dest_seqno, rev_rt->dest_addr,
                    lifetime);

        ext = rrep_add_ext(rrep, RREP_INET_DEST_EXT, rrep_size,
                    sizeof(struct in_addr), (char *) &rreq_dest);

        rrep_size += AODV_EXT_SIZE(ext);

        DEBUG(LOG_DEBUG, 0,
              "Intermediate node response for INTERNET dest: %s rrep_size=%d",
              ip_to_str(rreq_dest), rrep_size);

        rrep_send(rrep, rev_rt, gw_rt, rrep_size);
        return;
        }
#endif              /* CONFIG_GATEWAY_DISABLED */

        /* Respond only if the sequence number is fresh enough... */
        if (fwd_rt->dest_seqno != 0 &&
        (int32_t) fwd_rt->dest_seqno >= (int32_t) rreq_dest_seqno) {
        lifetime = timeval_diff(&fwd_rt->rt_timer.timeout, &now);
        rrep = rrep_create(0, 0, fwd_rt->hcnt, fwd_rt->dest_addr,
                    fwd_rt->dest_seqno, rev_rt->dest_addr,
                    lifetime);
        rrep_send(rrep, rev_rt, fwd_rt, rrep_size);
        } else {
        goto forward;
        }
```

310-387

判断该节点是否为目的节点

314-329

若该节点为目的节点，则回复 rrep 消息，并更新最大的序列号

330-374

若该节点为中间节点

343-373

查看是否包含 rreq 目的节点的路由表信息，若包含则回复 rrep 消息，若不包含则继续向周围广播 rreq 信息

378-387

比较收到的 rreq 消息包的序列号是否大于目的节点序列号，如果不是就表明该消息包已过期，不需要进行转发

```
418 /* Perform route discovery for a unicast destination */
419
420 void NS_CLASS rreq_route_discovery(struct in_addr dest_addr, u_int8_t flags,
421                                    struct ip_data *ipd)
422 {
423     struct timeval now;
424     rt_table_t *rt;
425     seek_list_t *seek_entry;
426     u_int32_t dest_seqno;
427     int ttl;
428 #define TTL_VALUE ttl
429
430     gettimeofday(&now, NULL);
431
432     if (seek_list_find(dest_addr))
433         return;
434
435     /* If we already have a route entry, we use information from it. */
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 */
440     if (!rt) {
441         dest_seqno = 0;
442
443         if (expanding_ring_search)
444             ttl = TTL_START;
445
446     } else {
447         dest_seqno = rt->dest_seqno;
448
449         if (expanding_ring_search) {
450             ttl = rt->hcnt + TTL_INCREMENT;
451         }
452
453 /*      if (rt->flags & RT_INET_DEST) */
454 /*          flags |= RREQ_DEST_ONLY; */
455
456         /* A routing table entry waiting for a RREP should not be expunged
457            before 2 * NET_TRAVERSAL_TIME... */
458         if (timeval_diff(&rt->rt_timer.timeout, &now) <
459             (2 * NET_TRAVERSAL_TIME))
460             rt_table_update_timeout(rt, 2 * NET_TRAVERSAL_TIME);
461     }
462
463     rreq_send(dest_addr, dest_seqno, ttl, flags);
464
465     /* Remember that we are seeking this destination */
466     seek_entry = seek_list_insert(dest_addr, dest_seqno, ttl, flags, ipd);
467
468     /* Set a timer for this RREQ */
469     if (expanding_ring_search)
470         timer_set_timeout(&seek_entry->seek_timer, RING_TRAVERSAL_TIME);
471     else
472         timer_set_timeout(&seek_entry->seek_timer, NET_TRAVERSAL_TIME);
473
474     DEBUG(LOG_DEBUG, 0, "Seeking %s ttl=%d", ip_to_str(dest_addr), ttl);
475
476     return;
477 }
```

418-477

路由发现过程

436-438

自身路由表中若包含目的节点的路由表项，则将 TTL 值设置为 NET_DIAMETER

486

寻路必须寻找目的节点

469-476

为该 rreq 消息包设置计时器

479-540（代码略）

本地修复路由表，与上面的路由发现过程非常相似

---

aodv_rrep 部分

aodv_rrep.h

```
32  /* RREP Flags: */
33
34  #define RREP_ACK        0x1
35  #define RREP_REPAIR     0x2
36
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;
```

37-59

rrep 报文定义段

变量	含义	长度
type	Type	8
res1	Reserved_1	6
a	Acknowledgement required	1
r	Repair flag	1
prefix	Prefix SIze	5

res2	Reserved_2	3
hcnt	Hop Count	8
dest_addr	Destination IP address	32
dest_seqno	Destination Sequence Number	32
orig_addr	Originator IP address	32
lifetime	LIfetime	32

34-35

同时，还在 define 处定义了 ACK、REPAIR 两个 flag，分别为 0001 和 0010。

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

63-66

定义了 RREP_ack 消息包的结构体

aodv_rrep.c

```
43  RREP *NS_CLASS rrep_create(u_int8_t flags,
44  ▸   ▸   ▸       u_int8_t prefix,
45  ▸   ▸   ▸       u_int8_t hcnt,
46  ▸   ▸   ▸       struct in_addr dest_addr,
47  ▸   ▸   ▸       u_int32_t dest_seqno,
48  ▸   ▸   ▸       struct in_addr orig_addr, u_int32_t life)
49  {
50      RREP *rrep;
51
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;
67
68      /* Don't print information about hello messages... *
69  #ifdef DEBUG_OUTPUT
70      if (rrep->dest_addr != rrep->orig_addr) {
71  ▸       DEBUG(LOG_DEBUG, 0, "Assembled RREP:");
72  ▸       log_pkt_fields((AODV_msg *) rrep);
73      }
74  #endif
75
76      return rrep;
77  }
```

43-77

创建 rrep 消息包

```
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  }
```

79-89

创建路由应答，分组信息得到确认并生成 debug 信息“Assembled RREP_ack”

```
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_src));
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 }
```

91-107

处理 rrep_ack 消息

96-98

判断自身节点路由表中有没有之前接受过的 rrep_ack 的源节点 IP 地址

99-101

若没有 IP 地址，那么该节点没有这一路由表项，表明该

rrep_ack 消息非法，中止返回

103-106

包含这一路由表项，那么就删除计时器

```
109 AODV_ext *NS_CLASS rrep_add_ext(RREP * rrep, int type, unsigned int offset,
110                                 int len, char *data)
111 {
112     AODV_ext *ext = NULL;
113 
114     if (offset < RREP_SIZE)
115     return NULL;
116 
117     ext = (AODV_ext *) ((char *) rrep + offset);
118 
119     ext->type = type;
120     ext->length = len;
121 
122     memcpy(AODV_EXT_DATA(ext), data, len);
123 
124     return ext;
125 }
```

109-125

rrep 消息包扩展项

```
127 void NS_CLASS rrep_send(RREP * rrep, rt_table_t * rev_rt,
128                       rt_table_t * fwd_rt, int size)
129 {
130     u_int8_t rrep_flags = 0;
131     struct in_addr dest;
132 
133     if (!rev_rt) {
134     DEBUG(LOG_WARNING, 0, "Can't send RREP, rev_rt = NULL!");
135     return;
136     }
137 
138     dest.s_addr = rrep->dest_addr;
139 
140     /* Check if we should request a RREP-ACK */
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);
144 
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     }
164 
165     DEBUG(LOG_DEBUG, 0, "Sending RREP to next hop %s about %s->%s",
166       ip_to_str(rev_rt->next_hop), ip_to_str(rev_rt->dest_addr),
167       ip_to_str(dest));
168 
169     aodv_socket_send((AODV_msg *) rrep, rev_rt->next_hop, size, MAXTTL,
170             &DEV_IFINDEX(rev_rt->ifindex));
171 
172     /* Update precursor lists */
173     if (fwd_rt) {
174     precursor_add(fwd_rt, rev_rt->next_hop);
175     precursor_add(rev_rt, fwd_rt->next_hop);
176     }
177 
178     if (!llfeedback && optimized_hellos)
179     hello_start();
180 }
```

127-180

rrep 消息发送

133-136

若不包含该路由表项，那么不能发送 rrep 消息包，中止返回

141-143

判断是否需要发送一个 rrep_ack 消息包

145-150

判断 rreq 消息包的目的节点是否为直接邻居节点，如果是的话，很有可能会是单向链路，所以要先发送一个 rrep_ack 消息包

155-161

如果需要忽略 hello 消息包，路由可能会中止。所以在设置单向路由时，所有的期间 hello 计时器都必须要被移除，并将超时时间设置为下一跳

169-170

向下一跳发送 rrep 消息包

173-176

更新 fwd_rt、rev_rt 的前驱路由表

178-179

判断是否有反馈和被处理的 hello 消息，如果没有则开始发送 hello 消息包

```
182 void NS_CLASS rrep_forward(RREP * rrep, int size, rt_table_t * rev_rt,
183                          rt_table_t * fwd_rt, int ttl)
184 {
185     /* Sanity checks... */
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     }
195 
196     DEBUG(LOG_DEBUG, 0, "Forwarding RREP to %s", ip_to_str(rev_rt->next_hop));
197 
198     /* Here we should do a check if we should request a RREP_ACK,
199        i.e we suspect a unidirectional link.. But how? */
200     if (0) {
201     rt_table_t *neighbor;
202 
203     /* If the source of the RREP is not a neighbor we must find the
204        neighbor (link) entry which is the next hop towards the RREP
205        source... */
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;
210 
211     if (neighbor && !neighbor->ack_timer.used) {
212         /* If the node we received a RREQ for is a neighbor we are
213            probably facing a unidirectional link... Better request a
214            RREP-ack */
215         rrep->a = 1;
216         neighbor->flags |= RT_UNIDIR;
217 
218         timer_set_timeout(&neighbor->ack_timer, NEXT_HOP_WAIT);
219     }
220     }
221 
222     rrep = (RREP *) aodv_socket_queue_msg((AODV_msg *) rrep, size);
223     rrep->hcnt = fwd_rt->hcnt;  /* Update the hopcount */
224 
225     aodv_socket_send((AODV_msg *) rrep, rev_rt->next_hop, size, ttl,
226             &DEV_IFINDEX(rev_rt->ifindex));
227 
228     precursor_add(fwd_rt, rev_rt->next_hop);
229     precursor_add(rev_rt, fwd_rt->next_hop);
230 
231     rt_table_update_timeout(rev_rt, ACTIVE_ROUTE_TIMEOUT);
232 }
```

182-232

rrep 消息转发

186-196

做一个“明智“的判断：若 fwd_rt 或 rev_rt 的路由不存在或者 rrep 消息为空时，中止返回

200-201

代码中出现了’if(0)’，也就是说还没有找到合适的条件来做一个判断，当我们发送一个 rrep_ack 请求时

206-209

如果该 rrep 消息的源 IP 地址不是自身节点的邻居，那就必须找到下一跳节点邻居

211-219

如果收到的 rreq 消息来自一个邻居，我们很可能会碰到单向链表的情况，所以最好发送一个 rrep_ack 请求

222-223

发送 rrep 消息包，并更新当前 rrep 值

235-411

rrep 消息处理

```
251     /* Convert to correct byte order on affeected fields: */
252     rrep_dest.s_addr = rrep->dest_addr;
253     rrep_orig.s_addr = rrep->orig_addr;
254     rrep_seqno = ntohl(rrep->dest_seqno);
255     rrep_lifetime = ntohl(rrep->lifetime);
```

252-255

将源 IP 地址、目的 IP 地址、序列号和生命周期转换为正确的字节序

```
256     /* Increment RREP hop count to account for intermediate node... */
257     rrep_new_hcnt = rrep->hcnt + 1;
```

267-268

如果 rrep 的目的节点为自身，则忽略并中止返回

```
310     /* ---------- CHECK IF WE SHOULD MAKE A FORWARD ROUTE ------------ */
311
312     fwd_rt = rt_table_find(rrep_dest);
313     rev_rt = rt_table_find(rrep_orig);
314
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     }
```

312-337

判断是否需要转发路由

315-318

如果该路由表项不存在，则增加一行路由表项

319-329

修复前置节点路由表，并增加一行路由表项

---

aodv_rerr 部分

aodv_rerr.h

```
32  /* RERR Flags: */
33  #define RERR_NODELETE 0x1
34
35  typedef struct {
36      u_int8_t type;
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;
```

32-50

rerr 消息包定义段

变量	含义	长度
type	Type	8
res1	Reserved_1	7
n	N	1
res2	Reserved_2	8
dest_count	Dest Count	8
dest_addr	Unreachable Destination IP Address	32
dest_seqno	Unreachable Destination Sequence Number	32

```
54  /* Extra unreachable destinations... */
55  typedef struct {
56      u_int32_t dest_addr;
57      u_int32_t dest_seqno;
58  } RERR_udest;
```

54-58

额外的不可达节点

aodv_rerr.c

```
38  RERR *NS_CLASS rerr_create(u_int8_t flags, struct in_addr dest_addr,
39  ▸   ▸   ▸       u_int32_t dest_seqno)
40  {
41      RERR *rerr;
42
43      DEBUG(LOG_DEBUG, 0, "Assembling RERR about %s seqno=%d",
44  ▸     ip_to_str(dest_addr), dest_seqno);
45
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;
54
55      return rerr;
56  }
```

38-56

rerr 消息创建，并设置 flags、dest_addr、dest_seqno 的值

```
58  void NS_CLASS rerr_add_udest(RERR * rerr, struct in_addr udest,
59  ▸   ▸   ▸         u_int32_t udest_seqno)
60  {
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++;
67  }
```

58-67

添加不可达节点。每添加一个节点，count 值加 1

70-210

rerr 消息处理过程

```
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      }
```

86-93

检验 rerr 消息的大小是否小于下限值，若小于则报错并中止返回

```
95      /* Check which destinations that are unreachable.  */
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);
```

96-103

循环检验不可达节点，并输出节点 IP 地址和序列号

```
105     rt = rt_table_find(udest_addr);
106
107     if (rt && rt->state == VALID && rt->next_hop.s_addr == ip_src.s_addr) {
108
109         /* Checking sequence numbers here is an out of draft
110          * addition to AODV-UU. It is here because it makes a lot
111          * of sense... */
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     }
118         DEBUG(LOG_DEBUG, 0, "removing rte %s - WAS IN RERR!!",
119       ip_to_str(udest_addr));
```

112-117

比较 rerr 消息中不可达节点的序列号和路由表中保存到的最新同节点的序列号，若小于则说明消息已过期，直接忽略中止返回

```
124             /* Invalidate route: */
125             if (!rerr->n) {
126                 rt_table_invalidate(rt);
127             }
```

125-127

删除无效路由

```
128         /* (a) updates the corresponding destination sequence number
129            with the Destination Sequence Number in the packet, and */
130         rt->dest_seqno = rerr_dest_seqno;
```

130

将路由的目的 IP 序列号更新为 rerr 消息包中对应的目的节点序列号

```
132         /* (d) check precursor list for emptiness. If not empty, include
133            the destination as an unreachable destination in the
134            RERR... */
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;
```

135-150

前驱列表若为空，则在 rerr 消息包中添加目的节点为不可达节点

```
152             } else {
153                 /* Decide whether new precursors make this a non unicast RERR */
154                 rerr_add_udest(new_rerr, rt->dest_addr, rt->dest_seqno);
155
156                 DEBUG(LOG_DEBUG, 0, "Added %s as unreachable, seqno=%lu",
157                       ip_to_str(rt->dest_addr), rt->dest_seqno);
158
159                 if (rerr_unicast_dest.s_addr) {
160                     list_t *pos2;
161                     list_foreach(pos2, &rt->precursors) {
162                         precursor_t *pr = (precursor_t *) pos2;
163                         if (pr->neighbor.s_addr != rerr_unicast_dest.s_addr) {
164                             rerr_unicast_dest.s_addr = 0;
165                             break;
166                         }
167                     }
168                 }
169             }
```

154-168

确定是否是新的前驱节点发送了该非单播 rerr 消息

```
174         /* We should delete the precursor list for all unreachable
175            destinations. */
176         if (rt->state == INVALID)
177             precursor_list_destroy(rt);
```

176-177

为所有的不可达节点删除前驱节点链表

```
185     /* If a RERR was created, then send it now... */
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         }
209     }
```

186-209

如果一个新的 rerr 消息包被创建，那么就立即发送它

190-194

如果 count 值为 1，则单播 rerr 消息包

196-208

若 count 值不为 1，则向包含坏路由的先驱节点发送 rerr 消息包

---

aodv_hello 部分

```
47  long NS_CLASS hello_jitter()
48  {
49      if (hello_jittering) {
50  #ifdef NS_PORT
51      return (long) (((float) Random::integer(RAND_MAX + 1) / RAND_MAX - 0.5)
52              * JITTER_INTERVAL);
53  #else
54      return (long) (((float) random() / RAND_MAX - 0.5) * JITTER_INTERVAL);
55  #endif
56      } else
57      return 0;
58  }
```

47-58

如果 hello 信息有抖动的话，返回一个随机数

```
60  void NS_CLASS hello_start()
61  {
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);
71  }
```

60-71

hello 消息发送开始

62-63

检验 hello 计时器是否开启，若已开启则表明 hello 消息包已发送，直接中止返回

65

获取当前节点时间

67-68

dubug 信息并初始化计时器

70

变量值为 NULL，发送 hello 消息包

```
73  void NS_CLASS hello_stop()
74  {
75      DEBUG(LOG_DEBUG, 0,
76        "No active forwarding routes - stopped sending HELLOs!");
77      timer_remove(&hello_timer);
78  }
```

73-78

hello 消息发送停止

77

移除当前绑定的计时器

80-164

发送 hello 消息

```
91      gettimeofday(&now, NULL);
92
93      if (optimized_hellos &&
94      timeval_diff(&now, &this_host.fwd_time) > ACTIVE_ROUTE_TIMEOUT) {
95      hello_stop();
96      return;
97      }
```

91-97

获取当前时间，并与 hello 消息发送时间做比较，若差值大于路由器活跃超时时间，则停止发送 hello 消息

```
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;
109 #ifdef DEBUG_HELLO
110         DEBUG(LOG_DEBUG, 0, "sending Hello to 255.255.255.255");
111 #endif
112         rrep = rrep_create(flags, 0, 0, DEV_NR(i).ipaddr,
113                            this_host.seqno,
114                            DEV_NR(i).ipaddr,
115                            ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
```

99-104

算出当前时间与最后一个广播消息发送到时间差，并比较该时间差与 hello 消息的周期大小，以确保在时间周期内不重复发送 hello 消息

109-111

若宏定义 DEBUG_HELLO 存在，则显示 hello 广播消息

112-115

创建一个 hello 消息

```
117         /* Assemble a RREP extension which contain our neighbor set... */
118         if (unidir_hack) {
119             int i;
120
121             if (ext)
122                 ext = AODV_EXT_NEXT(ext);
123             else
124                 ext = (AODV_ext *) ((char *) rrep + RREP_SIZE);
125
126             ext->type = RREP_HELLO_NEIGHBOR_SET_EXT;
127             ext->length = 0;
```

118-127

整合包含邻居集合到 rrep 扩展

```
133                 /* If an entry has an active hello timer, we assume
134                    that we are receiving hello messages from that
135                    node... */
136                 if (rt->hello_timer.used) {
137 #ifdef DEBUG_HELLO
138                     DEBUG(LOG_INFO, 0,
139                       "Adding %s to hello neighbor set ext",
140                       ip_to_str(rt->dest_addr));
141 #endif
142                     memcpy(AODV_EXT_DATA(ext), &rt->dest_addr,
143                        sizeof(struct in_addr));
144                     ext->length += sizeof(struct in_addr);
145                 }
```

136-145

如果当前路由有一个正在运行的 hello 计时器，就假设从该节点获得 hello 消息包

167-286

hello 消息处理

```
178     gettimeofday(&now, NULL);
179
180     hello_dest.s_addr = hello->dest_addr;
181     hello_seqno = ntohl(hello->dest_seqno);
```

178-181

获取当前时间，并获取目的节点和序列号，并转换成正确的字节序

```
239     /* This neighbor should only be valid after receiving 3
240        consecutive hello messages... */
241     if (receive_n_hellos)
242     ▸   state = INVALID;
243     else
244     ▸   state = VALID;
```

241-244

该邻居只有在收到 3 个连续的 hello 消息后才有效，并将 state 设置为相对应的值

```
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));
```

248-257

如果路由表中没有活动路由或过期路由。 所以我们添加一个新条目

```
269     ▸   if (receive_n_hellos && rt->hello_cnt < (receive_n_hellos - 1)) {
270     ▸       if (timeval_diff(&now, &rt->last_hello_time) <
271     ▸   ▸   (long) (hello_interval + hello_interval / 2))
272     ▸   ▸   rt->hello_cnt++;
273     ▸       else
274     ▸   ▸   rt->hello_cnt = 1;
275     ▸
276     ▸       memcpy(&rt->last_hello_time, &now, sizeof(struct timeval));
277     ▸       return;
278     ▸   }
```

269-278

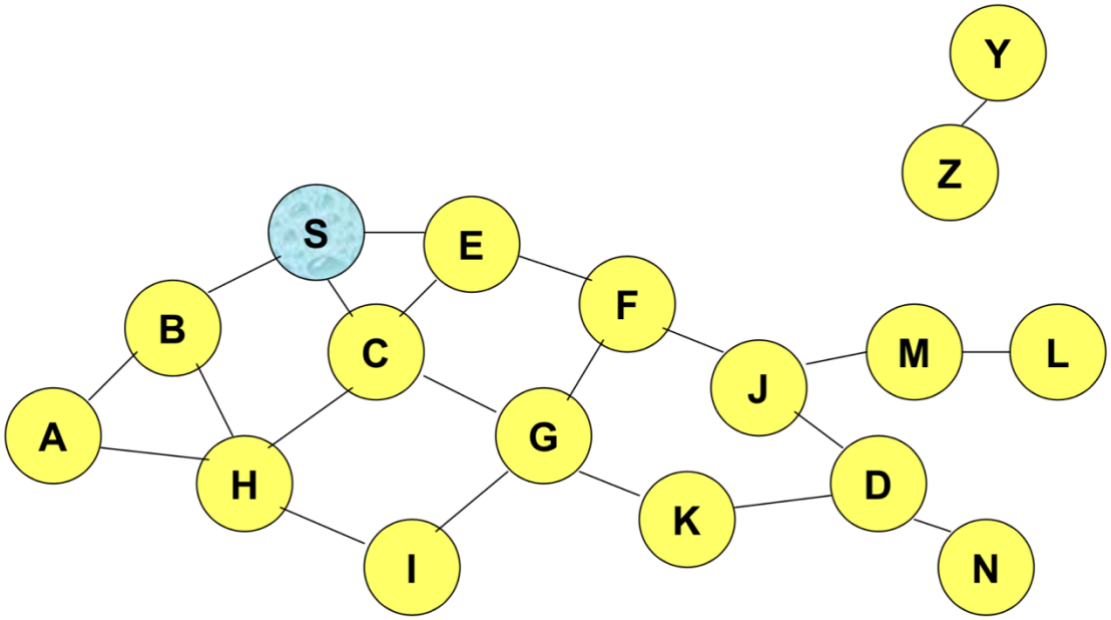
如果路由表中已有活跃路由表项，则增加生命周期，并更新路由表

```
289     #define HELLO_DELAY 50      /* The extra time we should allow an hello
290     ▸   ▸   ▸   ▸       message to take (due to processing) before
291     ▸   ▸   ▸   ▸       assuming lost . */
```

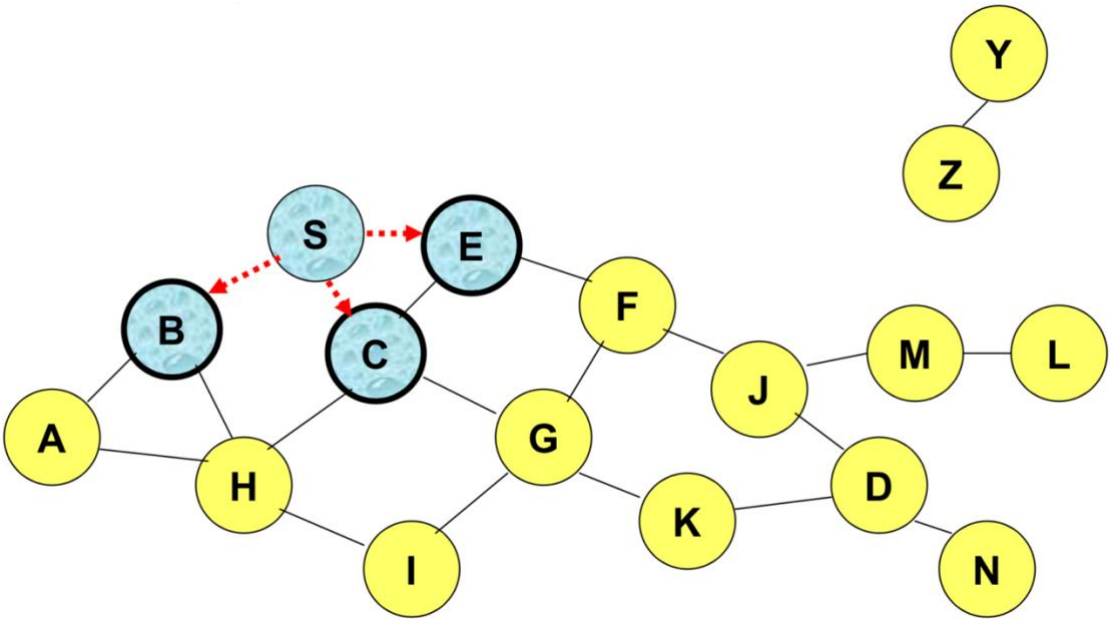
289

设置宏定义 HELLO_DELAY 为 50，防止网络延迟带来的误报

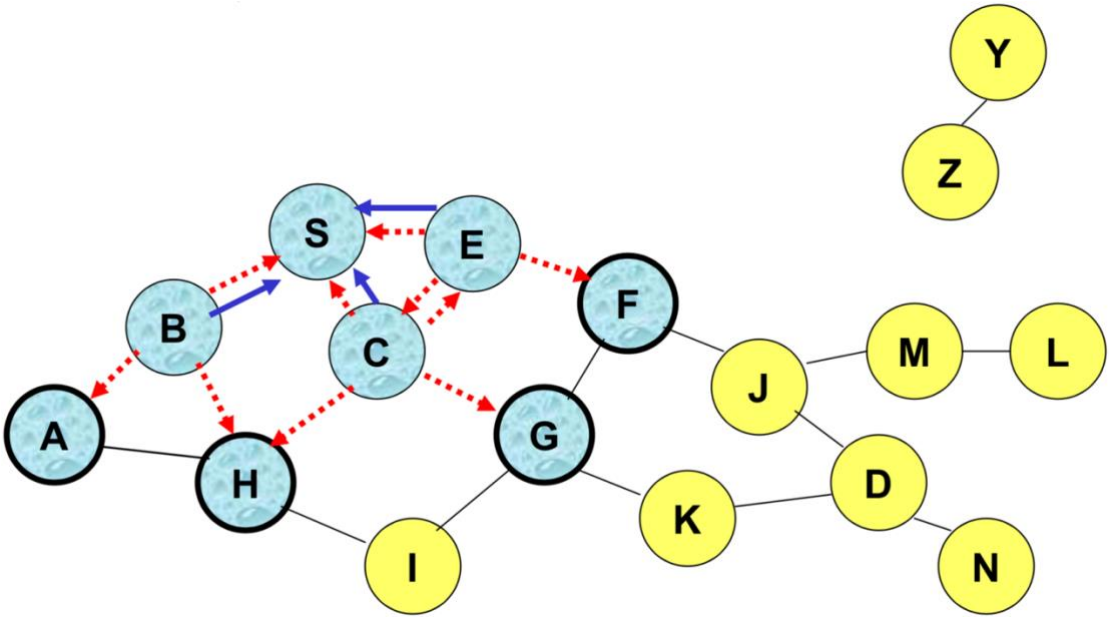
路由应答过程



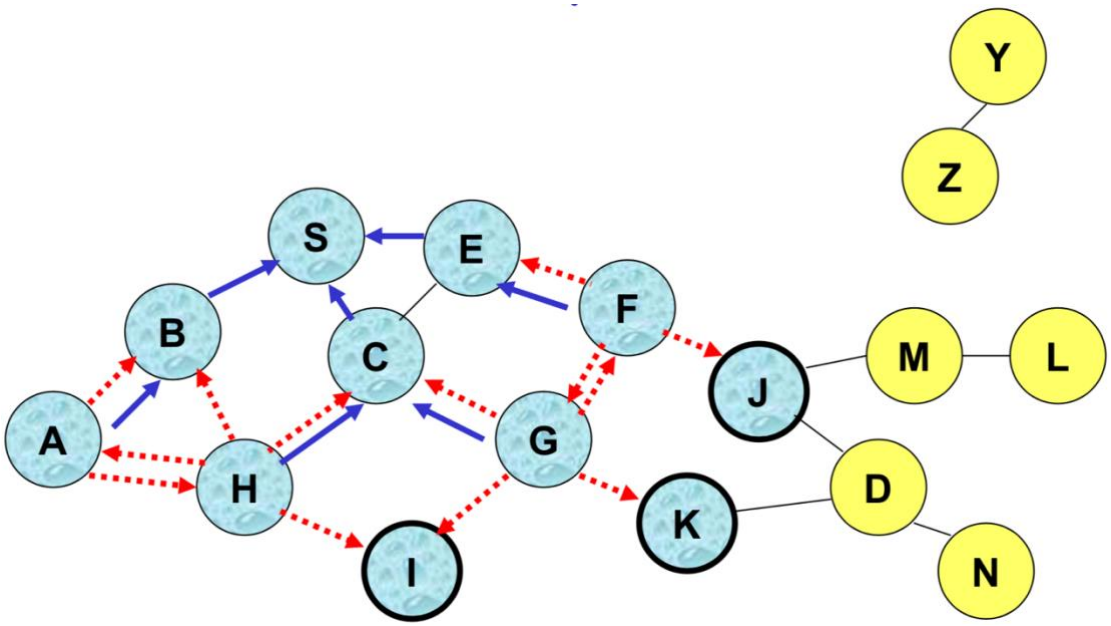
1. 蓝色代表已经收到从 S 发向 D 的 RREQ 报文



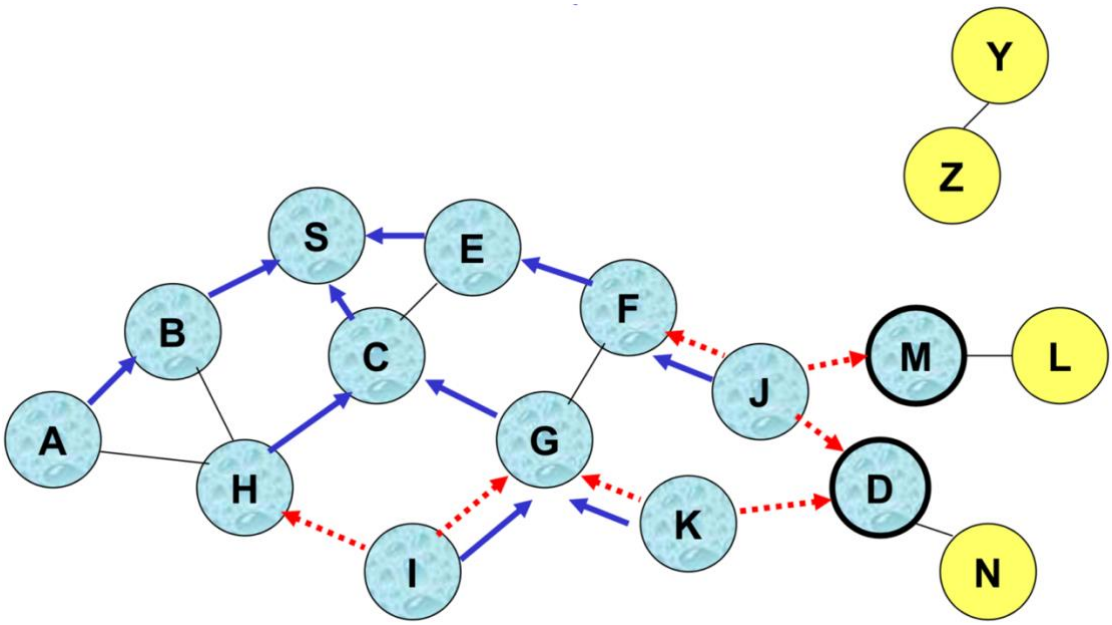
2. 红色箭头表示 RREQ 的传播方向



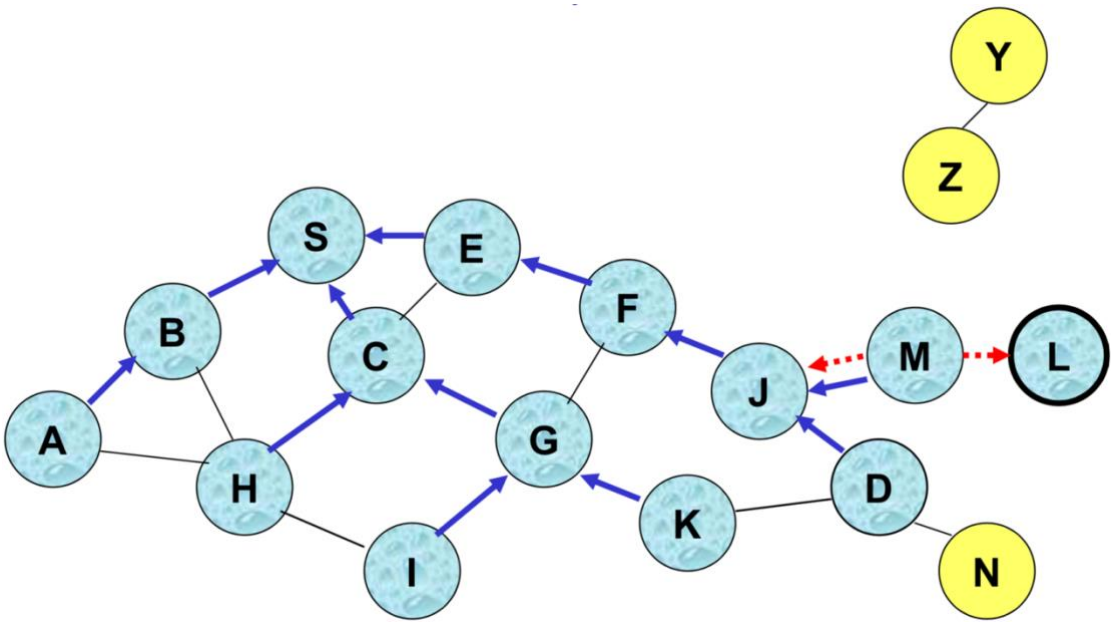
3. 蓝色箭头表示反向路由连接



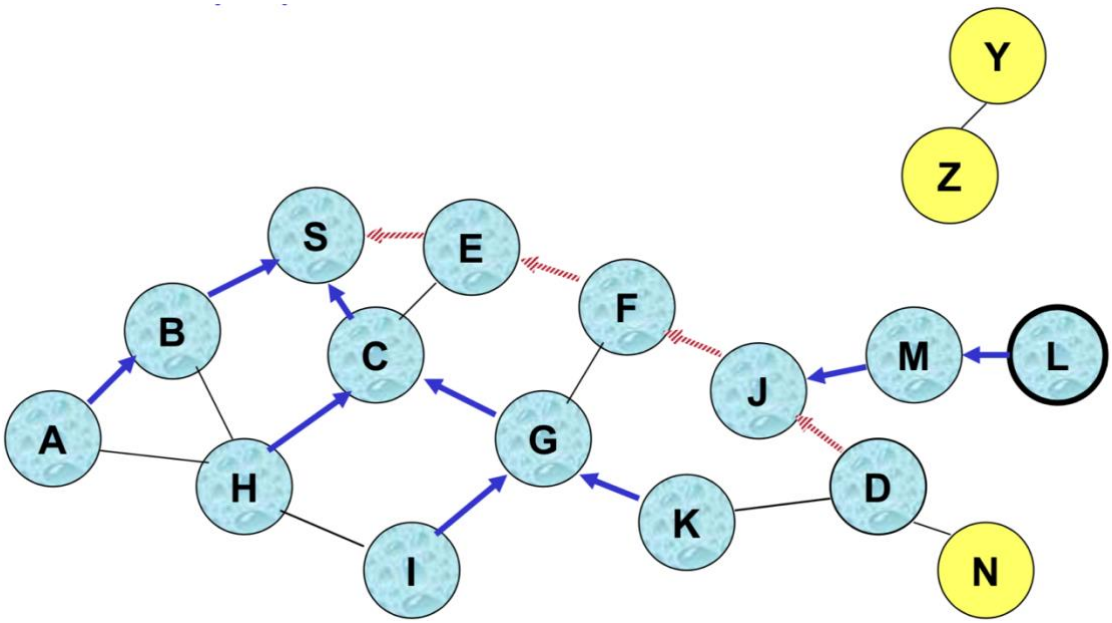
4. C 收到了来自 G 和 H 的 RREQ 报文，但没有再次转发，因为 C 已经转发过一次了



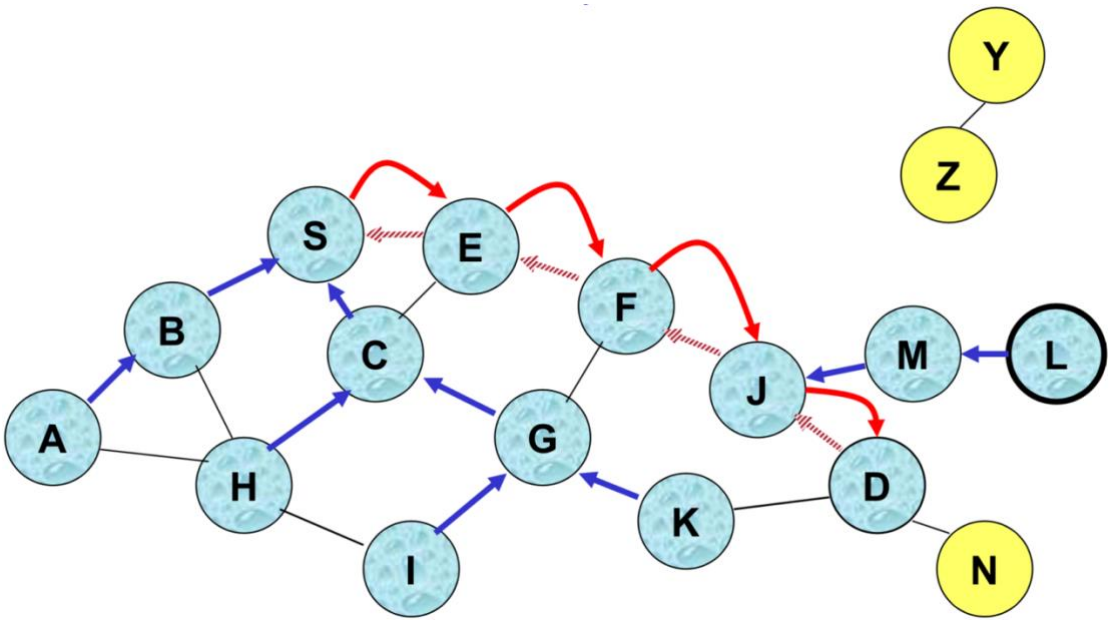
5. 继续转发



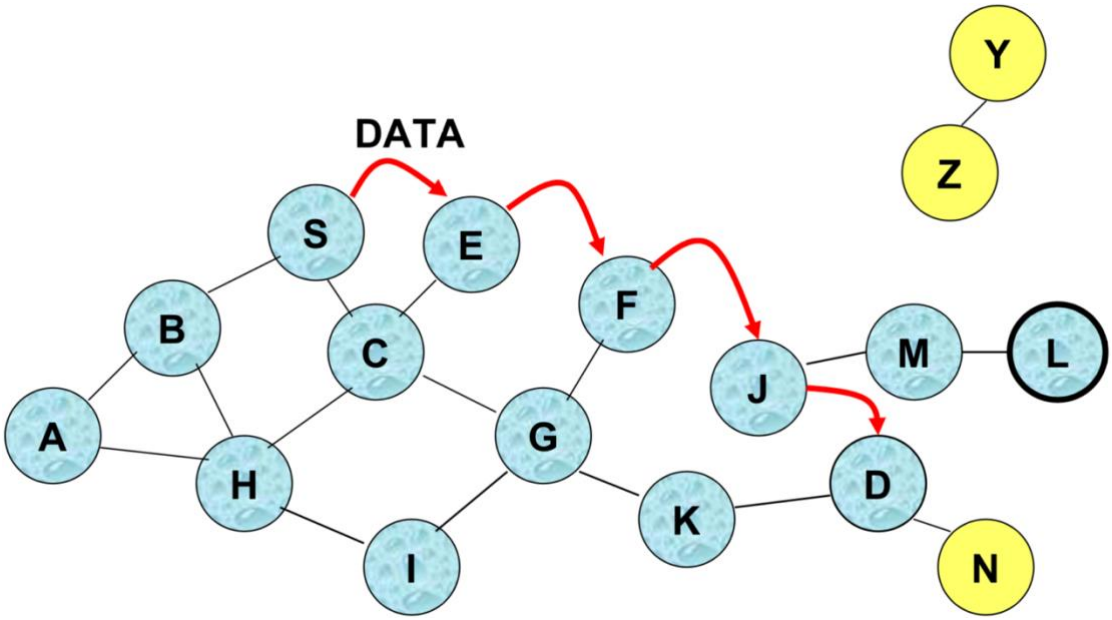
6. D 并未转发 RREQ 报文，因为他是该报文的目的节点



7. 棕色箭头表示 RREP 的路径



8. 当 RREP 沿着反向路由的路径时，转发连接建立了



9. 消息包开头不包含路由

---

aodv_neighbor.c

```
40  void NS_CLASS neighbor_add(AODV_msg * aodv_msg, struct in_addr source,
41                             unsigned int ifindex)
42  {
43      struct timeval now;
44      rt_table_t *rt = NULL;
45      u_int32_t seqno = 0;
46
47      gettimeofday(&now, NULL);
48
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      }
66
67      if (!llfeedback && rt->hello_timer.used)
68      hello_update_timeout(rt, &now, ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
69
70      return;
71  }
```

40-71：首先判断目的节点地址在目前的路由表项中是否存在，如果不存在则添加路由表项，如果是单向链路时则不需要更新任何路由信息。如果路由信息已存在且活跃，则应该增加对应路由的活跃周期。

```
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      }
92
93      DEBUG(LOG_DEBUG, 0, "Link %s down!", ip_to_str(rt->dest_addr));
94
95      /* Invalidate the entry of the route that broke or timed out... */
96      rt_table_invalidate(rt);
97
98      /* Create a route error msg, unless the route is to be repaired */
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     }
107
108     /* Purge precursor list: */
109     if (!(rt->flags & RT_REPAIR))
110     precursor_list_destroy(rt);
```

aodv_neighbor.c

84-110：如果跳数为一，则为直接邻居节点，这里要断开信息并将目的地址传回，用 rerr 信息传回需要修复的路由。

---

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) {
```

```
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);
154             } else {
```

```
159               if (rerr_unicast_dest.s_addr) {
160                   list_t *pos2;
161                   list_foreach(pos2, &rt_u->precursors) {
162                   precursor_t *pr = (precursor_t *) pos2;
163                   if (pr->neighbor.s_addr !=
164                       rerr_unicast_dest.s_addr) {
165                       rerr_unicast_dest.s_addr = 0;
166                       break;
167                   }
168                   }
169               }
```

116-154：如果下一跳是地址不可达节点的表项，则要在 rerr 消息中表示其不可用，其中被修复的链路也要将本来不可达节点的链路标记修复。

159-169：决定是否成为非单播的 RERR 消息。

aodv_timeout

aodv_timeout.c

```
50  void NS_CLASS route_discovery_timeout(void *arg)
51  {
52      struct timeval now;
53      seek_list_t *seek_entry;
54      rt_table_t *rt, *repair_rt;
55      seek_entry = (seek_list_t *) arg;
56
57  #define TTL_VALUE seek_entry->ttl
58
59      /* Sanity check... */
60      if (!seek_entry)
61          return;
62
63      gettimeofday(&now, NULL);
64
65      DEBUG(LOG_DEBUG, 0, "%s", ip_to_str(seek_entry->dest_addr));
66
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          }
```

50-85: 处理路由发现时的超时：记录当前时间，设置一个计时器来检测时间是否超过 2 个 NET_TRAVERSAL_TIME，如果超时就返回没有发现路由信息。

aodv_timeout.c

```
137     rerr_dest.s_addr = AODV_BROADCAST;  /* Default destination */
138
139     /* Unset the REPAIR flag */
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);
```

137-171：处理本地修复时的超时：设置路由的标志为 REPAIR，，使这条路由信息作废并广播 RERR 消息通知其他节点。如果修复超时，就清除队列里所有可能引用处于修复状态的数据包，并记录到日志里。

aodv_timeout.c

```
184  void NS_CLASS route_expire_timeout(void *arg)
185  {
186      rt_table_t *rt;
187
188      rt = (rt_table_t *) arg;
189
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      }
205
206      return;
207  }
208
```

184-208:当不需要某个路由（到期）的时候，将路由断开并从路由表中删去。

aodv_timeout.c

```
209  void NS_CLASS route_delete_timeout(void *arg)
210  {
211      rt_table_t *rt;
212
213      rt = (rt_table_t *) arg;
214
215      /* Sanity check: */
216      if (!rt)
217          return;
218
219      DEBUG(LOG_DEBUG, 0, "%s", ip_to_str(rt->dest_addr));
220
221      rt_table_delete(rt);
222  }
223
```

209-222：路由删除超时，记录日志，从路由表中删除路由信息。

aodv_timeout.c

```
226  void NS_CLASS hello_timeout(void *arg)
227  {
228      rt_table_t *rt;
229      struct timeval now;
230
231      rt = (rt_table_t *) arg;
232
233      if (!rt)
234          return;
235
236      gettimeofday(&now, NULL);
237
238      DEBUG(LOG_DEBUG, 0, "LINK/HELLO FAILURE %s last HELLO: %d",
239            ip_to_str(rt->dest_addr), timeval_diff(&now,
240                                                   &rt->last_hello_time));
241
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);
257      }
258  }
```

226-258：如果长时间没有收到邻居节点的 HELLO 消息，如果需要修复则标记为需要修复然后在队列中删除，邻居链路节点断掉，同时记录日志。

aodv_timeout.c

```
260  void NS_CLASS rrep_ack_timeout(void *arg)
261  {
262      rt_table_t *rt;
263
264      /* We must be really sure here, that this entry really exists at
265         this point... (Though it should). */
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));
276  }
277
278  void NS_CLASS wait_on_reboot_timeout(void *arg)
279  {
280      *((int *) arg) = 0;
281
282      DEBUG(LOG_DEBUG, 0, "Wait on reboot over!!");
283  }
284
285  #ifdef NS_PORT
286  void NS_CLASS packet_queue_timeout(void *arg)
287  {
288      packet_queue_garbage_collect();
289      timer_set_timeout(&PQ.garbage_collect_timer, GARBAGE_COLLECT_TIME)
290  }
291  #endif
```

260-291：rrep_ack_timeout()将传输失败的目的节点放入黑名单中并且记录日志。wait_on_reboot_timeout()重启超时则关闭并记录日志。packet_queue_timeout()数据包超时则重新手机并记录。

debug

debug.c

```
60  #ifdef NS_PORT
61
62      char AODV_LOG_PATH[strlen(AODV_LOG_PATH_PREFIX) +
63                         strlen(AODV_LOG_PATH_SUFFIX) + 16];
64      char AODV_RT_LOG_PATH[strlen(AODV_LOG_PATH_PREFIX) +
65                            strlen(AODV_RT_LOG_PATH_SUFFIX) + 16];
66
67
68      sprintf(AODV_LOG_PATH, "%s%d%s", AODV_LOG_PATH_PREFIX, node_id,
69          AODV_LOG_PATH_SUFFIX);
70      sprintf(AODV_RT_LOG_PATH, "%s%d%s", AODV_LOG_PATH_PREFIX, node_id,
71          AODV_RT_LOG_PATH_SUFFIX);
72  #endif              /* NS_PORT */
73
74      if (log_to_file) {
75      if ((log_file_fd =
76           open(AODV_LOG_PATH, O_RDWR | O_CREAT | O_TRUNC,
77            S_IROTH | S_IWUSR | S_IRUSR | S_IRGRP)) < 0) {
78          perror("open log file failed!");
79          exit(-1);
80      }
81      }
82      if (rt_log_interval) {
83      if ((log_rt_fd =
84           open(AODV_RT_LOG_PATH, O_RDWR | O_CREAT | O_TRUNC,
85            S_IROTH | S_IWUSR | S_IRUSR | S_IRGRP)) < 0) {
86          perror("open rt log file failed!");
87          exit(-1);
88      }
89      }
90      openlog(progname, 0, LOG_DAEMON);
91  }
92  void NS_CLASS log_rt_table_init()
93  {
94      timer_init(&rt_log_timer, &NS_CLASS print_rt_table, NULL);
95      timer_set_timeout(&rt_log_timer, rt_log_interval);
96  }
```

60-71：日志文件为路径前缀+IP 地址+路径后缀。

74-89：打开文件异常处理。

92-96：初始化路由表日志信息。

debug.c

```
98  void NS_CLASS log_cleanup()
99  {
100     if (log_to_file && log_file_fd) {
101     if (NS_OUTSIDE_CLASS close(log_file_fd) < 0)
102         fprintf(stderr, "Could not close log_file_fd!\n");
103     }
104 }
105
106 void NS_CLASS write_to_log_file(char *msg, int len)
107 {
108     if (!log_file_fd) {
109     fprintf(stderr, "Could not write to log file\n");
110     return;
111     }
112     if (len <= 0) {
113     fprintf(stderr, "len=0\n");
114     return;
115     }
116     if (write(log_file_fd, msg, len) < 0)
117     perror("Could not write to log file");
118 }
119
120 char *packet_type(u_int type)
121 {
122     static char temp[50];
123
124     switch (type) {
125     case AODV_RREQ:
126     return "AODV_RREQ";
127     case AODV_RREP:
128     return "AODV_RREP";
129     case AODV_RERR:
130     return "AODV_RERR";
131     default:
132     sprintf(temp, "Unknown packet type %d", type);
133     return temp;
134     }
135 }
```

100-106：关闭日志文件

108-120：将信息输入到日志文件

122-137：由参数 type 返回包的类型

debug.c

```
180     len += sprintf(log_buf + len, "%02d:%02d:%02d.%03ld %s: %s", time->tm_hour,
181             time->tm_min, time->tm_sec, now.tv_usec / 1000, function,
182             msg);
183
184     if (errnum == 0)
185     len += sprintf(log_buf + len, "\n");
186     else
187     len += sprintf(log_buf + len, ": %s\n", strerror(errnum));
188
189     if (len > 1024) {
190     fprintf(stderr, "alog(): buffer to small! len = %d\n", len);
191     goto syslog;
192     }
193
194     /* OK, we are clear to write the buffer to the aodv log file... */
195     if (log_to_file)
196     write_to_log_file(log_buf, len);
197
198     /* If we have the debug option set, also write to stdout */
199     if (debug)
200         printf("%s", log_buf);
201
202     /* Syslog all messages that are of severity LOG_NOTICE or worse */
203   syslog:
204     if (type <= LOG_NOTICE) {
205     if (errnum != 0) {
206         errno = errnum;
207         syslog(type, "%s: %s: %m", function, msg);
208     } else
209         syslog(type, "%s: %s", function, msg);
210     }
211     /* Exit on error */
212     if (type <= LOG_ERR)
213     exit(-1);
214 }
```

180-214:将需要的内容写到缓冲区中方便函数调用写入到文件里面。

debug.c

rreq_flags_to_str()

rrep_flags_to_str()

log_pkt_fields()

rt_flags_to_str()

state_to_str()

devs_ip_to_str()

217-338:将返回的内容按照报文格式存入缓冲区。

debug.c

```
340    void NS_CLASS print_rt_table(void *arg)
341    {
342        char rt_buf[2048], ifname[64], seqno_str[11];
343        int len = 0;
344        int i = 0;
345        struct timeval now;
346        struct tm *time;
347        ssize_t written;
```

340-438：用之前的函数将路由表描述出来。

内核部分：

kaodv-queue

函数名	作用
kaodv_queue_enqueue_entry（）	初始化
kaodv_queue_find_entry（）	寻找队列中的条目
kaodv_queue_find_dequeue_entry（）	删除队列中的条目
kaodv_queue_flush(void)（）	置为空
kaodv_queue_reset（）	重置队列
dest_cmp()	指向目的地址
kaodv_queue_set_verdict()	对相应包进行相应处理
kaodv_queue_get_info()	输出队列各种信息
init_or_cleanup()	对队列进行初始化和清空操作

kaodv-expl

函数名	作用
kaodv_expl_timeout()	超时函数
kaodv_expl_add()	添加路由条目
kaodv_expl_del()	删除路由表项
kaodv_expl_get()	查找路由表项
kaodv_expl_print()	存储到缓冲区
kaodv_expl_proc_info()	缓冲区信息
kaodv_expl_update()	更新路由信息

kaodv-netlink

函数名	作用
kaodv_netlink_build_msg（）	构造消息
kaodv_netlink_send_debug_msg（）	发送日志消息

kaodv_netlink_send_rt_msg（）	路由消息
kaodv_netlink_send_rt_update_msg（）	路由更新消息
kaodv_netlink_send_rerr_msg（）	RERR 消息以表错误
kaodv_netlink_receive_peer（）	根据 type 不同执行不同操作
kaodv_netlink_rcv_nl_event（）	nl 事件
kaodv_netlink_rcv_skb（）	内联函数，接收 skb 包相关

kaodv-mod

函数名	作用
kaodv_update_route_timeouts（）	更新路由的计时器，包括前后一跳
kaodv_hook（）	忽略不符合要求的包
kaodv_proc_info（）	缓冲区内容等过程信息
kaodv_read_proc（）	对过程信息进行阅读
kaodv_init（）	初始化函数

4 总结

AODV 协议是一种距离向量路由协议，使用目的序列号来避免实现路由环路，解决了传统的距离向量路由协议计数到无穷等问题，对出现问题的节点也可以比较快的反应，不活跃的节点和链路都会被删除，良好的日志文件格式让出现的问题可以得到即时纠正。总体而言让我们收获良多。总体学到了很多知识，对协议栈内容也有了新的认识和理解，包括对套接字的掌握，对我们的未来有很大的帮助。