《网络协议栈分析与设计》课程作业

题目：DSR 路由协议代码分析

姓名：马佳骏　学号：201692058　班级：软网 1602

姓名：赵天阳　学号：201692014　班级：软网 1602

姓名：宋文迪　学号：201692030　班级：软网 1602

目录

成员分工

姓名	代码分工
马佳骏	路由请求部分函数代码分析
赵天阳	协议基本原理、数据结构分析
宋文迪	路由确认部分函数代码分析

1. DSR 简介

1.1 DSR 协议特点

动态源路由协议(Dynamic Source Routing, DSR)是基于源路由概念的按需自适应路由协议，它工作在 TCP/IP 协议族的网际层。它具有以下几个特点：

1、节点动态发现到任何目的地的 Source route。

2、中间节点不必存储转发分组所需的路由信息。

3、采用 Cache 存放路由信息。

4、网络完全自我组织和自我配置。

5、网络开销较小。

6、存在陈旧路由。

7、引起简单有效而成为首选协议。

DSR 基于以下几个假设进行工作：

1、所有节点都愿意参与协议工作。

2、节点的移动速度相对于分组的传输延迟以及底层网络硬件的无线传输范围来说相对温和。

3、节点的无线网络接口以混杂方式工作。（将收到的每个帧都上交给驱动器软件，而不会针对目的地址进行过滤。）

4、任何一对节点之间的通信能力不同。（可支持单向链路）

5、任何一个节点只能宣称一个 IP 地址。

1.2 DSR 协议描述

DSR 协议由两个相互协同的机制构成：路由发现（Route discovery）和路由维护（Route maintenance）。

1.2.1 路由发现

产生路由请求

当源节点要向目标节点发送一个数据包时，首先在自己的路由缓存中查找是否存在到目标节点已知的路由信息。如果不存在这样的路由，源节点调用路由发现。向领节点广播路由请求消息，该消息中包含源节点地址，目标节点地址和路由请求 ID（这个 ID 根据源节点唯一确定，是路由请求消息的身份标志位）。

节点处理路由请求

相邻节点收到路由请求消息后，会进行以下判断和操作：

判断自己是否为目标节点，若该节点即为目标节点则向发起路由请求的源节点发送 RREP 路由应答消息（RREP 中包含 RREQ 中完整的源节点到目标节点的路由）。发送后，节点会删除接收到的路由请求消息。

判断本节点地址是否已经包含在路由请求消息的路由中，若存在表示该路由中存在冗余的路由，为使路由发现的路由信息不重复，该节点会直接删除路由请求消息。

判断这个路由请求消息是否已经超时（TTL 是否超过最大值），若超时则把路由请求消息删除。

当该节点不是目标节点时，判断该节点的路由缓存中是否存在已知的由该节点出发到目标节点的路由信息。若存在，则把路由请求中的路由信息和该节点路由缓存中的路由信息一起封装到路由应答消息中，调用路由应答发送函数把 RREP 发送给源节点，并把路由请求消息删除。若缓存中不存在这样的路由，则该节点把自己的地址添加到路由请求的路选项中并向邻接点广播路由请求。

路由应答

本次分析的 DSR 协议默认网络为双链路，所以由中间节点或者目标节点发起的路由应答，RREP 消息会根据其中包含的源节点到目标节点的路由的逆路由，通过中间的节点依次传送。

1.2.2 路由维护

当源节点正在使用一条到达目的节点的源路由的时候，源节点使用路由维护机制可以检测出因为拓扑变化不能使用的路由，当路由维护指出一条源路由已经中断而不再起作用的时候，为了将随后的数据分组传输到目的节点，源节点能够尽力使用一条偶然获知的到达目的节点的路由，或者重新调用路由寻找机制找到一条新路由；并为了维持整个网络的拓扑结构，一些网络节点会把失效链路有关的源路由从自己的路由缓存中删除，避免自己转发数据分组时再次用到失效的路由而产生转发错误。

1.3 DSR 优劣比较

DSR 协议的优点

1、节点无需周期性地发送路由广播分组。

2、无需维持到全网所有节点的路由信息，进一步节省了电池能量和网络带宽，尤其是当没有节点需要发送数据的时候，网络中没有通信开销。

3、仅需要维护路径上节点之间的连通。

4、能完全消除路由环路。

5、可用于单向信道。

6、中间节点的应答使源节点快速获得路由。

DSR 协议的缺点

1、会引起过时路由问题.

2、每个分组都需要携带完整的路由信息，增大了开销，降低了网络带宽的利用率。

3、不适合直径大的自组网。

4、网络可扩展性不强。

2 代码分析

2.1 协议文件

整个协议文件夹中有各种.c、.h 文件，其中.c 文件为主要的协议分析对象。下表列举了协议主要功能实现文件及其功能简述，后面也将对这些文件代码中的一些进行分析。

文件	功能描述
link-cache.c	对路由链路缓存操作，主要为源路由查找、添加和删除等
maint-buf.c	对维护缓存器操作，主要为其中数据分组的添加、删除等
dsr-rreq.c	主要为路由请求分组的创建、发送和收到后的处理等
dsr-rrep.c	主要为路由应答分组的创建、发送和收到后的处理等
dsr-ack.c	主要为确认分组的创建、发送和收到后的处理等
dsr-rerr.c	主要为路由错误分组的创建、发送和收到后的处理等
send-buf.c	对发送缓存器操作，主要为其中数据分组的添加等
dsr-pkt.c	主要为数据分组的创建和释放等
dsr-opt.c	主要为分组选项的查找、移除和解析等
dsr-io.c	主要为分组的发送和接收等
dsr-srt.c	主要为对源路由的创建和添加等

2.2 DSR 协议数据分组及路由选项格式分析

2.2.1 DSR 选项头部

dsr_opt.h

```
26  /* The DSR options header (always comes first) */
27  struct dsr_opt_hdr {
28      u_int8_t nh;
29  #if defined(__LITTLE_ENDIAN_BITFIELD)
30
31      u_int8_t res:7;
32      u_int8_t f:1;
33  #elif defined (__BIG_ENDIAN_BITFIELD)
34      u_int8_t f:1;
35      u_int8_t res:7;
36  #else
37  #error  "Please fix <asm/byteorder.h>"
38  #endif
39      u_int16_t p_len;     /* payload length */
40  #ifdef NS2
41      static int offset_;
42
43      inline static int &offset() {
44          return offset_;
45      }
46      inline static dsr_opt_hdr *access(const Packet * p) {
47          return (dsr_opt_hdr *) p->access(offset_);
48      }
49
50      int size() {
51          return ntohs(p_len) + sizeof(struct dsr_opt_hdr);
52      }
53  #endif              /* NS2 */
54      struct dsr_opt option[0];
55  };
```

nh:

用于标识紧跟 DSR 首部后面的数据分组头，取值与 IPV4的协议域相同。

res:

必须设置为0发送，并在接收时忽略。

f：

流状态标志，在 DSR 选项头中置0，在 DSR 流状态头中置1。

p_len:

表示整个 DSR 选项头的长度.

option:

dsr 路由选项。其长度可变，每个 DSR 选项被分配有一个唯一的选项码类型(type-length).

2.2.2 DSR 路由选项头部

2.2.2.1 路由请求选项（RREQ）

dsr_rreq.h

```
19  struct dsr_rreq_opt {
20      u_int8_t type;
21      u_int8_t length;
22      u_int16_t id;
23      u_int32_t target;
24      u_int32_t addrs[0];
25  };
```

id:

路由请求消息的 id(身份标志)，由发起路由发现的源节点唯一确定

target：

即 destination，代表目标节点的地址。

addrs：

用来存放由源节点到目标节点路由信息的数组，记录中间经过的节点的地址，源节点的地址不用记录在这里。

2.2.2.2 路由应答选项头部（RREP）

dsr_rrep.h

```
16  struct dsr_rrep_opt {
17      u_int8_t type;
18      u_int8_t length;
19  #if defined(__LITTLE_ENDIAN_BITFIELD)
20      u_int8_t res:7;
21      u_int8_t l:1;
22  #elif defined (__BIG_ENDIAN_BITFIELD)
23      u_int8_t l:1;
24      u_int8_t res:7;
25  #else
26  #error  "Please fix <asm/byteorder.h>"
27  #endif
28      u_int32_t addrs[0];
29  };
```

res ：

7位置零的保留位

l：

状态位，当它置1时，表明路由回复中的最后一跳到达了 DSR 网络的外部。

addrs：

用来存放由源节点到目标节点路由信息的数组。同时也是 RREP 消息转发时依照的路由信息。

2.2.2.3 路由错误选项

dsr_rerr.h

```
19  struct dsr_rerr_opt {
20      u_int8_t type;
21      u_int8_t length;
22      u_int8_t err_type;
23  #if defined(__LITTLE_ENDIAN_BITFIELD)
24      u_int8_t res:4;
25      u_int8_t salv:4;
26  #elif defined (__BIG_ENDIAN_BITFIELD)
27      u_int8_t res:4;
28      u_int8_t salv:4;
29  #else
30  #error  "Please fix <asm/byteorder.h>"
31  #endif
32      u_int32_t err_src;
33      u_int32_t err_dst;
34      char info[0];
35  };
```

err_type:

指明该路由错误的类型，在 dsr_rerr.h 中定义了三种错误类型，分别是：NODE_UNREACHABLE, FLOW_STATE_NOT_SUPPORTED, OPTION_NOT_SUPPORTED.

res：

4位置零的保留位

salv：

表示该数据分组在发生路由错误后被回收重发的次数。

err_src:

发生路由错误的路由的源节点地址。

err_dst:

发生路由错误的路由的目标节点地址。

info：

存放路由错误的详细信息。

2.2.2.4 路由请求确认选项（ack_rep_opt）

dsr_ack.h

```
15  struct dsr_ack_req_opt {
16      u_int8_t type;
17      u_int8_t length;
18      u_int16_t id;
19  };
```

Id:

确认收到的路由请求的 id。

2.2.2.5 确认选项（ack_opt）

dsr_ack.h

```
21  struct dsr_ack_opt {
22      u_int8_t type;
23      u_int8_t length;
24      u_int16_t id;
25      u_int32_t src;
26      u_int32_t dst;
27  };
```

src：

确认的路由的源节点地址。

dst：

确认的路由的目标节点地址。

2.3 DSR 主要功能函数代码分析

2.3.1 link_cache.c 代码分析

2.3.1.1 lc_link_add()函数分析

```
290  int NSCLASS lc_link_add(struct in_addr src, struct in_addr dst,
291                  usecs_t timeout, int status, int cost)
292  {
293      struct lc_node *sn, *dn;
294      int res;
295
296      DSR_WRITE_LOCK(&LC.lock);
297
298      sn = (struct lc_node *)__tbl_find(&LC.nodes, &src, crit_addr);
299
300      if (!sn) {
301          sn = lc_node_create(src);
302
303          if (!sn) {
304              DEBUG("Could not allocate nodes\n");
305              DSR_WRITE_UNLOCK(&LC.lock);
306              return -1;
307          }
308          __tbl_add_tail(&LC.nodes, &sn->l);
309
310      }
311
312      dn = (struct lc_node *)__tbl_find(&LC.nodes, &dst, crit_addr);
```

```
313
314      if (!dn) {
315          dn = lc_node_create(dst);
316          if (!dn) {
317              DEBUG("Could not allocate nodes\n");
318              DSR_WRITE_UNLOCK(&LC.lock);
319              return -1;
320          }
321          __tbl_add_tail(&LC.nodes, &dn->l);
322      }
323
324      res = __lc_link_tbl_add(&LC.links, sn, dn, timeout, status, cost);
325
326      if (res) {
327  #ifdef LC_TIMER
328  #ifdef NS2
329          if (!timer_pending(&lc_timer))
330  #else
331          if (!timer_pending(&LC.timer))
332  #endif
333              lc_garbage_collect_set();
334  #endif
335
336      } else if (res < 0)
337          DEBUG("Could not add new link\n");
```

```
326     if (res) {
327 #ifdef LC_TIMER
328 #ifdef NS2
329         if (!timer_pending(&lc_timer))
330 #else
331         if (!timer_pending(&LC.timer))
332 #endif
333             lc_garbage_collect_set();
334 #endif
335
336     } else if (res < 0)
337         DEBUG("Could not add new link\n");
338
339     DSR_WRITE_UNLOCK(&LC.lock);
340
341     return 0;
342 }
```

lc_link_add()函数一般被 lc_srt_add()函数被调用来向路由链路中建立两个节点之间的链路。

lc_link_add()的形参分别为：src，dst，timeout，status 和 cost，各自指代需要建立链路的起点地址，终点地址，链路超时时间，链路当前状态和链路开销。链路添加成功返回0，不成功返回-1.

296-310

打开路由链路缓存 LC 的写锁。通过调用 tbl_find()函数在路由链路缓存 LC 中查找包含 src 的链路，并将其赋值给 sn。如果 sn 为空，即在路由链路缓存 LC 中没有包含 src 的链路，调用 lc_node_create()函数创建以 src 为起点的链路：如果创建失败，该函数调用 DEBUG 函数输出提示信息，关闭路由链路缓存 LC 的写锁，返回-1；否则，调用 tbl_add_tail()函数将 sn 中的节点添加到路由链路缓存 LC 的 nodes 中。

312-322

通过调用 tbl_find()函数在路由链路缓存 LC 中查找包含 dst 的源路由，并将其赋值给 dn。如果 dn 为空，即在路由链路缓存 LC 中没有包含 dst 的链路，调用 lc_node_create()函数创建以 dst 为起点的链路：如果创建失败，该函数调用 DEBUG 函数输出提示信息，关闭路由链路缓存 LC 的写锁，返回-1；否则，调用 tbl_add_tail（）函数将 dn 中的节点添加到路由链路缓存 LC 的 nodes 中。

324-341

通过调用 lc_link_tbl_add()函数将以 src 为起点、以 dst 终点的路由链路添加到路由链路缓存 LC 的 links 中，关闭路由链路缓存 LC 的写锁，返回0。

2.3.1.2 lc_link_del()函数分析

```
344  int NSCLASS lc_link_del(struct in_addr src, struct in_addr dst)
345  {
346      struct lc_link *link;
347      int res = 1;
348
349      DSR_WRITE_LOCK(&LC.lock);
350
351      link = __lc_link_find(&LC.links, src, dst);
352
353      if (!link) {
354          res = -1;
355          goto out;
356      }
357
358      __lc_link_del(&LC, link);
359
360      /* Assume bidirectional links for now */
361      link = __lc_link_find(&LC.links, dst, src);
362
363      if (!link) {
364          res = -1;
365          goto out;
366      }
367
368      __lc_link_del(&LC, link);
369        out:
370      LC.src = NULL;
371      DSR_WRITE_UNLOCK(&LC.lock);
372
373      return res;
374  }
```

lc_link_del()被调用来删除路由链路缓存中从源节点到目标节点的路由。

lc_link_del()的形参分别为：src，dst，分别指代要删除的路由的源节点地址和目标节点地址。成功执行返回 1。

349-358

打开路由链路缓存 LC 的写锁。通过调用 lc_link_find()函数在路由链路缓存 LC 中查找以 src 为源节点、以 dst 为目的节点的路由，并将函数返回值赋值给 link。如果 link 为空，将 res 赋值为-1，跳转到 out，将路由链路缓存 LC 的 src 赋值为 NULL，关闭链路存储器 LC 的写锁，返回值为-1 的 res；否则，调用 lc_link_del()函数从路由链路缓存 LC 中删除这条路由。

361-363

如果为双向路由，以上述同样步骤处理以 dst 为源节点、以 src 为目的节点的逆向源路由。

2.3.1.3 lc_srt_find()函数分析

```
447  struct dsr_srt *NSCLASS lc_srt_find(struct in_addr src, struct in_addr dst)
448  {
449      struct dsr_srt *srt = NULL;
450      struct lc_node *dst_node;
451
452      if (src.s_addr == dst.s_addr)
453          return NULL;
454
455      DSR_WRITE_LOCK(&LC.lock);
456
457  /*  if (!LC.src || LC.src->addr.s_addr != src.s_addr) */
458      __dijkstra(src);
459
460      dst_node = (struct lc_node *)__tbl_find(&LC.nodes, &dst, crit_addr);
461
462      if (!dst_node) {
463          DEBUG("%s not found\n", print_ip(dst));
464          goto out;
465      }
466
467  /*  lc_print(&LC, lc_print_buf); */
468  /*  DEBUG("Find SR to node %s\n%s\n", print_ip(dst_node->addr), lc_print_buf); */
469
470  /*  DEBUG("Hops to %s: %u\n", print_ip(dst), dst_node->hops); */
472      if (dst_node->cost != LC_COST_INF && dst_node->pred) {
473          struct lc_node *d, *n;
474  /*      struct lc_link *l; */
475          int k = (dst_node->hops - 1);
476          int i = 0;
477
478          srt = (struct dsr_srt *)MALLOC(sizeof(struct dsr_srt) +
479                                         (k * sizeof(struct in_addr)),
480                                         GFP_ATOMIC);
481
482          if (!srt) {
483              DEBUG("Could not allocate source route!!!\n");
484              goto out;
485          }
486
487          srt->dst = dst;
488          srt->src = src;
489          srt->laddrs = k * sizeof(struct in_addr);
490
491          /*      l = __lc_link_find(&LC.links, dst_node->pred->addr, dst_node->addr); */
492
493  /*      if (!l) { */
494  /*          DEBUG("Link not found for timeout update!\n"); */
495  /*      } else { */
496  /*      /\*     DEBUG("Updating timeout for link %s->%s\n",  *\/ */
497  /* /\*                 print_ip(l->src->addr),  *\/ */
498  /* /\*                 print_ip(l->dst->addr)); *\/ */
499  /*          gettime(&l->expires); */
500  /*      } */
501
```

```
501
502         d = dst_node;
503
504         /* Fill in the source route by traversing the nodes starting
505          * from the destination predecessor */
506         for (n = dst_node->pred; (n != n->pred); n = n->pred) {
507
508             /*      l = __lc_link_find(&LC.links, n->addr, d->addr); */
509
510 /*          if (!l) { */
511 /*              DEBUG("Link not found for timeout update!\n"); */
512 /*          } else { */
513 /*          /\*      DEBUG("Updating timeout for link %s->%s\n",  *\/ */
514 /* /\*                    print_ip(l->src->addr),  *\/ */
515 /* /\*                    print_ip(l->dst->addr)); *\/ */
516 /*              gettime(&l->expires); */
517 /*          } */
518             srt->addrs[k - i - 1] = n->addr;
519             i++;
520             d = n;
521         }
522
523         if ((i + 1) != (int)dst_node->hops) {
524             DEBUG("hop count ERROR i+1=%d hops=%d!!!\n", i + 1,
525                   dst_node->hops);
526             FREE(srt);
527             srt = NULL;
528         }
529     }
530       out:
531     DSR_WRITE_UNLOCK(&LC.lock);
532
533     return srt;
534 }
```

lc_srt_find()在源节点要发送新的数据分组到目标节点时被源节点调用，目的是在 节点路由链路缓存 LC 中查询从源节点到目标点的路由链路信息。

lc_srt_find()函数的两个形参分别为 in_addr 类型的 src 和 dst，各自指代要查询的路由信息的源节点地址和目标节点地址。函数返回运行成功的话会返回一个 srt 源路由：查询到的路由信息。

452-455

如果 src 地址等于 dst 地址,即 src 和 dst 是同一个节点,该函数返回 NULL，结束查找从 src 到点 dst 的源路由。否则，打开 src 的链路存储器 LC 的写锁，开始查找从 src 到 dst 的源路由。

458-460

首先，调用 dijkstra（）函数获取以 src 为源节点的源路由，并将这些源路由存入 src 的路由链路缓存 LC 中。然后，调用 tbl_find（）函数在 src 的路由链路缓存 LC 中查找包含 dst 的源路由，并将这条源路由经过的所有节点存入 dst_node 中。

462-465

如果 dst_node 为空，即没有在 src 的路由链路缓存 LC 中查找到包含 dst

的源路由，该函数调用 DEBUG 函数输出提示信息，并跳转到 out 关闭 src 的路由链路缓存 LC 的写锁，返回值为 NULL 的 srt。

472-529

在满足 dst_node 的 cost 不等于 LC_COST_INF 并且 pred 不为空的前提下进入 for 循环，为 srt 分配合适的存储空间：如果分配失败，该函数调用 DEBUG 函数输出提示信息，并跳转到 out 关闭 src 的路由链路缓存 LC 的写锁，返回值为 NULL 的 srt；否则，遍历 dst_node 中的所有节点，并将这些节点添加到 srt 中。在这个 for 循环中会判断 dst_node 的 hops 是否出错：如果出错，调用 DEBUG 函数输出提示信息，释放 srt 的存储空间，并将 srt 重新赋值为 NULL；否则，该函数继续循环，在循环结束后该函数返回从源节点到目的节点的源路由 srt。

2.3.1.4 lc_srt_add()函数

```
536  int NSCLASS
537  lc_srt_add(struct dsr_srt *srt, usecs_t timeout, unsigned short flags)
538  {
539      int i, n, links = 0;
540      struct in_addr addr1, addr2;
541
542      if (!srt)
543          return -1;
544
545      n = srt->laddrs / sizeof(struct in_addr);
546
547      addr1 = srt->src;
548
549      for (i = 0; i < n; i++) {
550          addr2 = srt->addrs[i];
551
552          lc_link_add(addr1, addr2, timeout, 0, 1);
553          links++;
554
555          if (srt->flags & SRT_BIDIR) {
556              lc_link_add(addr2, addr1, timeout, 0, 1);
557              links++;
558          }
559          addr1 = addr2;
560      }
561      addr2 = srt->dst;
562
563      lc_link_add(addr1, addr2, timeout, 0, 1);
564      links++;
565
566      if (srt->flags & SRT_BIDIR) {
567          lc_link_add(addr2, addr1, timeout, 0, 1);
568          links++;
569      }
570      return links;
571  }
```

lc_srt_add()函数被节点调用来向该节点的路由链路缓存中添加新的路由信息。

lc_srt_add()函数的三个形参分别为：srt，timeout 和 flags，各自指代要添加的新路由信息，定义的超时时间和路由标志变量。函数成功运行会返回要添加的路由经过的节点跳数。

545-570

计算这条新源路由经过的节点数，并将节点数赋值给 n。通过调用 lc_link_add（）函数，将这条源路由经过的所有节点添加到这条源路由源节点的路由链路缓存 LC 中，并将跳数 links 加一。在添加节点的过程中，通过 srt->flags & SRT_BIDIR 判断这条源路由是否为双向的：如果是双向的，调用 lc_link_add()函数，将从目标节点到源节点的逆向路由也添加到路由链路缓存 LC 中，跳数 links 也加一。

2.3.2 dsr_rreq.c

2.3.2.1 ds_rreq_route_discovery()函数分析

```
342  int NSCLASS dsr_rreq_route_discovery(struct in_addr target)
343  {
344      struct rreq_tbl_entry *e;
345      int ttl, res = 0;
346      struct timeval expires;
347
348  #define TTL_START 1
349
350      DSR_WRITE_LOCK(&rreq_tbl.lock);
351
352      e = (struct rreq_tbl_entry *)__tbl_find(&rreq_tbl, &target, crit_addr);
353
354      if (!e)
355          e = __rreq_tbl_add(target);
356      else {
357          /* Put it last in the table */
358          __tbl_detach(&rreq_tbl, &e->l);
359          __tbl_add_tail(&rreq_tbl, &e->l);
360      }
361
362      if (!e) {
363          res = -ENOMEM;
364          goto out;
365      }
366
367      if (e->state == STATE_IN_ROUTE_DISC) {
368          DEBUG("Route discovery for %s already in progress\n",
369                print_ip(target));
370          goto out;
371      }
372      DEBUG("Route discovery for %s\n", print_ip(target));
373
374      gettime(&e->last_used);
375      e->ttl = ttl = TTL_START;
376      /* The draft does not actually specify how these Request Timeout values
377       * should be used... ??? I am just guessing here. */
378
379      if (e->ttl == 1)
380          e->timeout = ConfValToUsecs(NonpropRequestTimeout);
381      else
382          e->timeout = ConfValToUsecs(RequestPeriod);
383
384      e->state = STATE_IN_ROUTE_DISC;
385      e->num_rexmts = 0;
386
387      expires = e->last_used;
388      timeval_add_usecs(&expires, e->timeout);
389
390      set_timer(e->timer, &expires);
391
392      DSR_WRITE_UNLOCK(&rreq_tbl.lock);
393
394      dsr_rreq_send(target, ttl);
395
396      return 1;
397        out:
398      DSR_WRITE_UNLOCK(&rreq_tbl.lock);
399
400      return res;
401  }
```

当源节点要发送新数据分组调用 lc_srt_find()函数后没有找到对应到目标节点的路由时，源节点调用 dsr_rreq_routr_discovery()函数来发起从源节点到目标节点的路由发现。

dsr_rreq_routr_discovery()的形参为 target，指代需要路由发现的目标节点地址（由于调用这个函数的节点即为源节点所以参数中不用提现源节点地址）。

350-365

打开路由请求表 rreq_tbl 的写锁。通过调用 tbl_find()函数在路由请求表 rreq_tbl 中查找包含 target 的路由请求，并将函数返回值赋值给 e。如果 e 为空，即没有在路由请求表 rreq_tbl 中查找到包含 target 的路由请求，调用 rreq_tbl_add()函数创建以 target 为目标节点的路由请求并将其添加到路由请求表 rreq_tbl 中，将 res 赋值为-ENOMEM，关闭路由请求表 rreq_tbl 的写锁，返回 res；否则找到包含 target 的路由请求，直接调用 tbl_detach()函数和 tbl_add_tail()函数将查找到的路由请求移到路由请求表 rreq_tbl 的末尾。

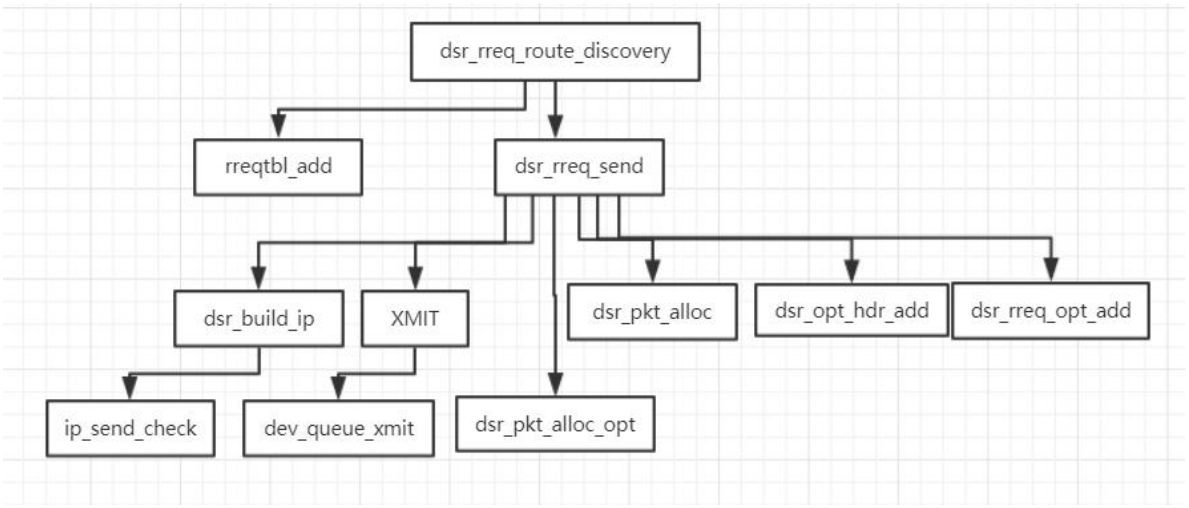
367-371

判断该路由发现 e 的状态是否为 STATE_IN_ROUTE_DISC，即该路由发现是否已经在进行中:如果是,调用 DEBUG 函数输出提示信息,关闭路由请求表 rreq_tbl 的写锁，返回 res.

372-400

如果对应路由请求并没有与在进行中,则更新该路由发现 e 的 ttl、timeout、state、num_rexmts 值。调用 timeval_add_usecs 函数设置该路由发现 e 的超时时间，并调用 set_timer 函数开启定时器。关闭路由请求表 rreq_tbl 的写锁。调用 dsr_rreq_send 函数发送路由请求。返回 res。

2.3.2.2 dsr_rreq_send()函数分析



dsr_rreq_route_discovery()和 dsr_rreq_send()函数调用图

```
438  int NSCLASS dsr_rreq_send(struct in_addr target, int ttl)
439  {
440      struct dsr_pkt *dp;
441      char *buf;
442      int len = DSR_OPT_HDR_LEN + DSR_RREQ_HDR_LEN;
443
444      dp = dsr_pkt_alloc(NULL);
445
446      if (!dp) {
447          DEBUG("Could not allocate DSR packet\n");
448          return -1;
449      }
450      dp->dst.s_addr = DSR_BROADCAST;
451      dp->nxt_hop.s_addr = DSR_BROADCAST;
452      dp->src = my_addr();
453
454      buf = dsr_pkt_alloc_opts(dp, len);
455
456
457      if (!buf)
458          goto out_err;
459
460      dp->nh.iph =
461          dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN, IP_HDR_LEN + len,
462               IPPROTO_DSR, ttl);
463
464      if (!dp->nh.iph)
465          goto out_err;
466
467      dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
468
```

```
469     if (!dp->dh.opth) {
470         DEBUG("Could not create DSR opt header\n");
471         goto out_err;
472     }
473
474     buf += DSR_OPT_HDR_LEN;
475     len -= DSR_OPT_HDR_LEN;
476
477     dp->rreq_opt = dsr_rreq_opt_add(buf, len, target, ++rreq_seqno);
478
479     if (!dp->rreq_opt) {
480         DEBUG("Could not create RREQ opt\n");
481         goto out_err;
482     }
483 #ifdef NS2
484     DEBUG("Sending RREQ src=%s dst=%s target=%s ttl=%d iph->saddr()=%d\n",
485           print_ip(dp->src), print_ip(dp->dst), print_ip(target), ttl,
486           dp->nh.iph->saddr());
487 #endif
488
489     dp->flags |= PKT_XMIT_JITTER;
490
491     XMIT(dp);
492
493     return 0;
494
495       out_err:
496     dsr_pkt_free(dp);
497
498     return -1;
499 }
```

dsr_rreq_send()函数被 dsr_rreq_routr_discovery()函数调用，来以广播的形式向源节点的所有邻接点发送 rreq 消息。

dsr_rreq_send()的形参分别为：target 和 ttl，分别指代路由发现的目标节点地址和路由请求的生存期（即可以经过的节点个数）。函数执行成功返回0，不成功返回-1。

438-449

行代码为数据包分配空间，头部长度应为 dsr 选项的头部长度加上路由请求的头部长度，分配出错返回-1。

450-454

行代码将数据包中的目的地址与下一跳地址填写为广播地址，因为在 dsr 协议中，如果一个节点需要发送数据那么它首先需要寻找路由，它会采取洪泛广播形式，向局域网中发送一个广播来寻找到目的地的路由信息。因此我们需要在数据中的地址信息处填写广播地址。调用 myaddr()函数获取当前节点的 IP 地址为源节点地址。并数据包的选项部分分配空间存到 buf 中。

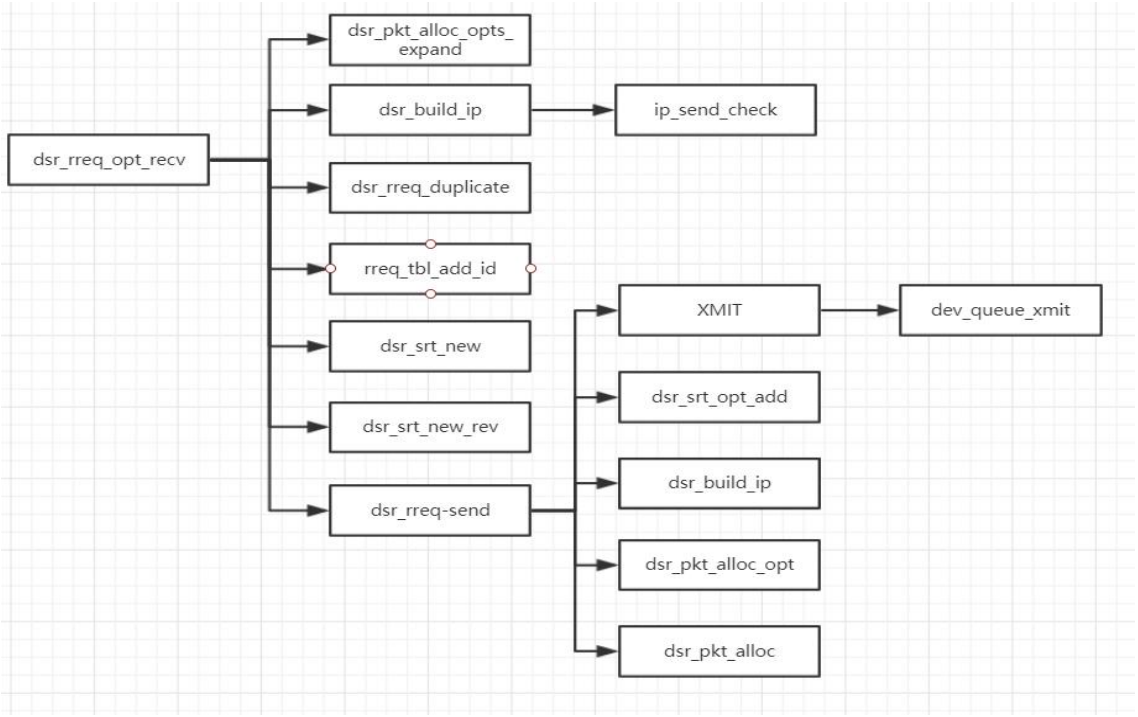
454-458

通过调用 dsr_pkt_alloc_opts 函数为 dp 分配选项空间:如果分配空间失败，跳转到 out_err，调用 dsr_pkt_free 函数释放 dp 空间，该函数返回-1。

460-482

通过调用 dsr_build_ip()函数为 dp 设置 IP，调用 dsr_opt_hdr_add 函数为 dp 设置选项头部，调用 dsr_rreq_opt_add 函数为 dp 设置选项。调用 XMIT 函数初始化并发送路由请求分组 dp。该函数返回 0。

2.3.2.3 dsr_rreq_opt_recv()函数分析



dsr_rreq_opt_recv()函数调用图

```
501  int NSCLASS dsr_rreq_opt_recv(struct dsr_pkt *dp, struct dsr_rreq_opt *rreq_opt)
502  {
503      struct in_addr myaddr;
504      struct in_addr trg;
505      struct dsr_srt *srt_rev, *srt_rc;
506      int action = DSR_PKT_NONE;
507      int i, n;
508
509      if (!dp || !rreq_opt || dp->flags & PKT_PROMISC_RECV)
510          return DSR_PKT_DROP;
511
512      dp->num_rreq_opts++;
513
514      if (dp->num_rreq_opts > 1) {
515          DEBUG("More than one RREQ opt!!! - Ignoring\n");
516          return DSR_PKT_ERROR;
517      }
518
519      dp->rreq_opt = rreq_opt;
520
521      myaddr = my_addr();
522
523      trg.s_addr = rreq_opt->target;
524
525      if (dsr_rreq_duplicate(dp->src, trg, ntohs(rreq_opt->id))) {
526          DEBUG("Duplicate RREQ from %s\n", print_ip(dp->src));
527          return DSR_PKT_DROP;
528      }
529
530      rreq_tbl_add_id(dp->src, trg, ntohs(rreq_opt->id));
531
532      dp->srt = dsr_srt_new(dp->src, myaddr, DSR_RREQ_ADDRS_LEN(rreq_opt),
533                            (char *)rreq_opt->addrs);
534
```

```
535     if (!dp->srt) {
536         DEBUG("Could not extract source route\n");
537         return DSR_PKT_ERROR;
538     }
539     DEBUG("RREQ target=%s src=%s dst=%s laddrs=%d\n",
540           print_ip(trg), print_ip(dp->src),
541           print_ip(dp->dst), DSR_RREQ_ADDRS_LEN(rreq_opt));
542
543     /* Add reversed source route */
544     srt_rev = dsr_srt_new_rev(dp->srt);
545
546     if (!srt_rev) {
547         DEBUG("Could not reverse source route\n");
548         return DSR_PKT_ERROR;
549     }
550     DEBUG("srt: %s\n", print_srt(dp->srt));
551     DEBUG("srt_rev: %s\n", print_srt(srt_rev));
552
553     dsr_rtc_add(srt_rev, ConfValToUsecs(RouteCacheTimeout), 0);
554
555     /* Set previous hop */
556     if (srt_rev->laddrs > 0)
557         dp->prv_hop = srt_rev->addrs[0];
558     else
559         dp->prv_hop = srt_rev->dst;
560
561     neigh_tbl_add(dp->prv_hop, dp->mac.ethh);
562
563     /* Send buffered packets */
564     send_buf_set_verdict(SEND_BUF_SEND, srt_rev->dst);
565
```

```
566     if (rreq_opt->target == myaddr.s_addr) {
567
568         DEBUG("RREQ OPT for me - Send RREP\n");
569
570         /* According to the draft, the dest addr in the IP header must
571          * be updated with the target address */
572 #ifdef NS2
573         dp->nh.iph->daddr() = (nsaddr_t) rreq_opt->target;
574 #else
575         dp->nh.iph->daddr = rreq_opt->target;
576 #endif
577         dsr_rrep_send(srt_rev, dp->srt);
578
579         action = DSR_PKT_NONE;
580         goto out;
581     }
582
583     n = DSR_RREQ_ADDRS_LEN(rreq_opt) / sizeof(struct in_addr);
584
585     if (dp->srt->src.s_addr == myaddr.s_addr)
586         return DSR_PKT_DROP;
587
588     for (i = 0; i < n; i++)
589         if (dp->srt->addrs[i].s_addr == myaddr.s_addr) {
590             action = DSR_PKT_DROP;
591             goto out;
592         }
593
594     /* TODO: Check Blacklist */
595     srt_rc = lc_srt_find(myaddr, trg);
596
```

```
597     if (srt_rc) {
598         struct dsr_srt *srt_cat;
599         /* Send cached route reply */
600 
601         DEBUG("Send cached RREP\n");
602 
603         srt_cat = dsr_srt_concatenate(dp->srt, srt_rc);
604 
605         FREE(srt_rc);
606 
607         if (!srt_cat) {
608             DEBUG("Could not concatenate\n");
609             goto rreq_forward;
610         }
611 
612         DEBUG("srt_cat: %s\n", print_srt(srt_cat));
613 
614         if (dsr_srt_check_duplicate(srt_cat) > 0) {
615             DEBUG("Duplicate address in source route!!!\n");
616             FREE(srt_cat);
617             goto dsr_srt* dsr_rreq_opt_recv::srt_cat
618         }
619 #ifdef NS2
620         dp->nh.iph->daddr() = (nsaddr_t) rreq_opt->target;
621 #else
622         dp->nh.iph->daddr = rreq_opt->target;
623 #endif
624         DEBUG("Sending cached RREP to %s\n", print_ip(dp->src));
625         dsr_rrep_send(srt_rev, srt_cat);
626 
627         action = DSR_PKT_NONE;
628 
629         FREE(srt_cat);
630     } else {
631 
```

```
632     rreq_forward:
633         dsr_pkt_alloc_opts_expand(dp, sizeof(struct in_addr));
634 
635         if (!DSR_LAST_OPT(dp, rreq_opt)) {
636             char *to, *from;
637             to = (char *)rreq_opt + rreq_opt->length + 2 +
638                 sizeof(struct in_addr);
639             from = (char *)rreq_opt + rreq_opt->length + 2;
640 
641             memmove(to, from, sizeof(struct in_addr));
642         }
643         rreq_opt->addrs[n] = myaddr.s_addr;
644         rreq_opt->length += sizeof(struct in_addr);
645 
646         dp->dh.opth->p_len = htons(ntohs(dp->dh.opth->p_len) +
647                                    sizeof(struct in_addr));
648 #ifdef __KERNEL__
649         dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
650                      ntohs(dp->nh.iph->tot_len) +
651                      sizeof(struct in_addr), IPPROTO_DSR,
652                      dp->nh.iph->ttl);
653 #endif
654         /* Forward RREQ */
655         action = DSR_PKT_FORWARD_RREQ;
656     }
657       out:
658     FREE(srt_rev);
659     return action;
660 }
```

当某个节点收到来自邻接点的 rreq 路由请求消息后会调用 dsr_rreq_opt_recv()函数来对 rreq 消息进行处理:转发,丢弃或启动路由应答。

dsr_rreq_opt_recv()的形参分别为：dp 和 rreq_opt，分别指代收到的路由请求分组和路由请求选项。函数执行返回处理类型。

509-519

如果dp为空,或者rreq_opt为空,或者dp的flags值为 PKT_PROMISC_RECV，该函数返回 DSR_PKT_DROP，表示要把路由请求分组丢弃；否则，将 dp 的 num_rreq_opts 值加一。这时，判断 dp 的 num_rreq_opts 值是否大于 1：如果是，调用 DEBUG 函数输出错误信息，该函数返回 DSR_PKT_ERROR，表示收到的路由请求分组存在问题；否则，将 dp 分组中读取的 rreq_opt 赋值为参数二 rreq_opt。

525-528

通过调用 dsr_rreq_duplicate()函数判断该节点是否已经收到过和路由请求分组 dp 具有相同目的节点地址和 ID 的路由请求分组：如果是，调用 DEBUG 函数输出提示信息，该函数返回 DSR_PKT_DROP；否则，将该路由请求加入到自己的路由请求表，并更新该路由请求分组 dp。

543-554

向节点路由缓存中添加逆向路由信息，以便以后发送路由请求回复报文，函数执行成功打印出相关地址信息并之后在路由缓存中添加与时间相关的信息,否则函数返回 DSR_PKT_DROP。

566-581

判断 rreq_out 的目的节点地址是否是该节点的地址(即判断自己是否为目标节点)：如果是，调用 dsr_rrep_send（）函数向源节点回复路由应答，并将 action 值设置为 DSR_PKT_NONE，跳转到 out，释放 str_rev 空间，返回值为 DSR_PKT_NONE 的 action。

585-592

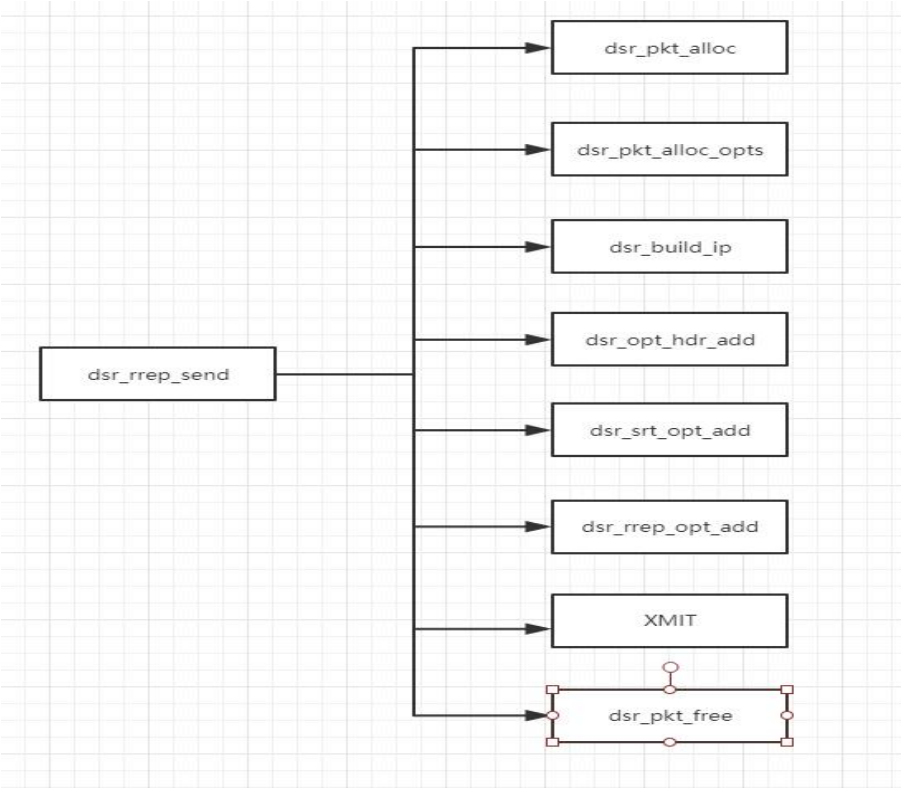
判断 dp 的源节点地址是否是该节点自己的地址：如果是，该函数返回 DSR_PKT_DROP。否则，判断该节点自己的地址是否已经列在该路由请求 dp 的路由记录列表中：如果是，该函数返回 DSR_PKT_DROP。

595-656

该节点检查自己的链路存储器中是否有到达目的节点的路由：如果存在，调用 dsr_rrep_send 函数向源节点回复包含到达目的节点的路由的路由应答；否则，该节点将自己的地址添加到路由请求 dp 的路由记录列表中，按与源节点同样的方式转发该路由请求分组 dp，并将 action 值设置为 DSR_PKT_FORWARD_RREQ 并返回。

2.3.3 dsr_rrep.c

2.3.3.1 dsr_rrep.send()函数分析



dsr_rrep.send()函数调用图

```
221  int NSCLASS dsr_rrep_send(struct dsr_srt *srt, struct dsr_srt *srt_to_me)
222  {
223      struct dsr_pkt *dp = NULL;
224      char *buf;
225      int len, ttl, n;
226
227      if (!srt || !srt_to_me)
228          return -1;
229
230      dp = dsr_pkt_alloc(NULL);
231
232      if (!dp) {
233          DEBUG("Could not allocate DSR packet\n");
234          return -1;
235      }
236
237      dp->src = my_addr();
238      dp->dst = srt->dst;
239
240      if (srt->laddrs == 0)
241          dp->nxt_hop = dp->dst;
242      else
243          dp->nxt_hop = srt->addrs[0];
244
245      len = DSR_OPT_HDR_LEN + DSR_SRT_OPT_LEN(srt) +
246          DS#define DSR_OPT_HDR_LEN  sizeof(struct dsr_opt_hdr)
247
248      n = srt->laddrs / sizeof(struct in_addr);
249
```

```
250     DEBUG("srt: %s\n", print_srt(srt));
251     DEBUG("srt_to_me: %s\n", print_srt(srt_to_me));
252     DEBUG("next_hop=%s\n", print_ip(dp->nxt_hop));
253     DEBUG
254         ("IP_HDR_LEN=%d DSR_OPT_HDR_LEN=%d DSR_SRT_OPT_LEN=%d DSR_RREP_OPT_LEN=%d DSR_OPT_PAD1_LEN=%d RREP len=%d\n",
255          IP_HDR_LEN, DSR_OPT_HDR_LEN, DSR_SRT_OPT_LEN(srt),
256          DSR_RREP_OPT_LEN(srt_to_me), DSR_OPT_PAD1_LEN, len);
257
258     ttl = n + 1;
259
260     DEBUG("TTL=%d, n=%d\n", ttl, n);
261
262     buf = dsr_pkt_alloc_opts(dp, len);
263
264     if (!buf)
265         goto out_err;
266
267     dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
268                               IP_HDR_LEN + len, IPPROTO_DSR, ttl);
269
270     if (!dp->nh.iph) {
271         DEBUG("Could not create IP header\n");
272         goto out_err;
273     }
274
275     dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
276
277     if (!dp->dh.opth) {
278         DEBUG("Could not create DSR options header\n");
279         goto out_err;
280     }
281
282         buf += DSR_OPT_HDR_LEN;
283         len -= DSR_OPT_HDR_LEN;
284
285         /* Add the source route option to the packet */
286         dp->srt_opt = dsr_srt_opt_add(buf, len, 0, dp->salvage, srt);
287
288         if (!dp->srt_opt) {
289             DEBUG("Could not create Source Route option header\n");
290             goto out_err;
291         }
292
293         buf += DSR_SRT_OPT_LEN(srt);
294         len -= DSR_SRT_OPT_LEN(srt);
295
296         dp->rrep_opt[dp->num_rrep_opts++] =
297             dsr_rrep_opt_add(buf, len, srt_to_me);
298
299         if (!dp->rrep_opt[dp->num_rrep_opts - 1]) {
300             DEBUG("Could not create RREP option header\n");
301             goto out_err;
302         }
303
304         /* TODO: Should we PAD? The rrep struct is padded and aligned
305          * automatically by the compiler... How to fix this? */
306
307 /*      buf += DSR_RREP_OPT_LEN(srt_to_me); */
308 /*      len -= DSR_RREP_OPT_LEN(srt_to_me); */
309
310 /*      pad1_opt = (struct dsr_pad1_opt *)buf; */
311 /*      pad1_opt->type = DSR_OPT_PAD1; */
312
313         /* if (ConfVal(UseNetworkLayerAck)) */
314 /*          dp->flags |= PKT_REQUEST_ACK; */
315
316         dp->flags |= PKT_XMIT_JITTER;
317
318         XMIT(dp);
319
320         return 0;
321           out_err:
322         if (dp)
323             dsr_pkt_free(dp);
324
325         return -1;
326 }
```

dsr_rrep_send()函数被目标节点或满足条件的中间节点在接收 rreq 消息后调用，向源节点发送路由应答消息。

dsr_rrep_send()的形参分别为：srt 和 srt_to_me,分别指代从接收到的 rreq 消息中读取的路由信息和自己路由缓存中读取的信息（这里需要注意，如果该函数由目标节点用掉，srt_to_me 应该为空值）。函数执行成功返回 0，不成功返回-1.

223-225

函数申请变量，dp 即为要发送的 rrep 消息，buf 用来存放 dsr 各个路由选项信息。

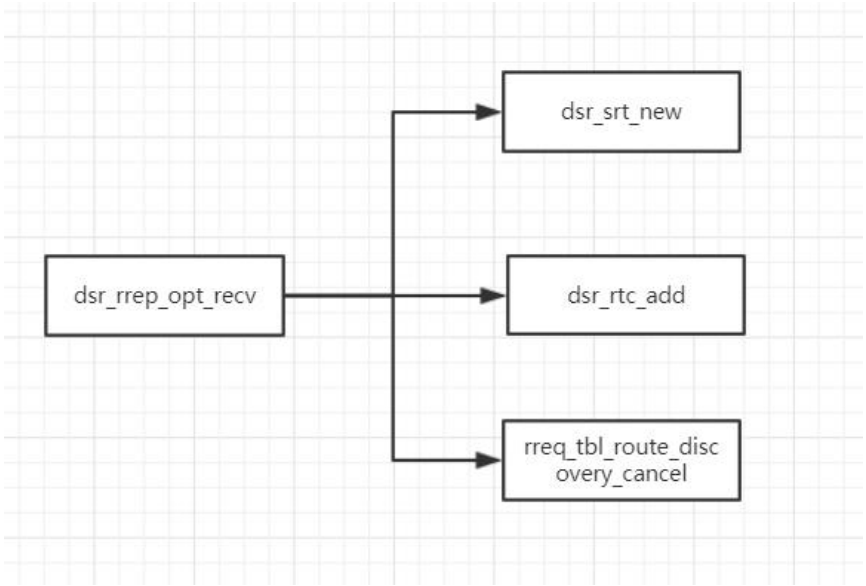
227-238

判断 srt、srt_to_me 是否为在空值：如果都为空，该函数返回-1；否则，为路由应答分组 dp 分配空间。然后，判断为 dp 分配空间是否成功：如果失败，调用 DEBUG 函数输出提示信息，该函数返回-1；否则，将 dp 的 src 赋值为该节点自己的地址，dst 赋值为 srt 的目的节点地址。

240-325

调用 dsr_pkt_alloc_opts()函数为 dp 选项分配空间：如果失败，跳转到 out_err，调用 dsr_pkt_free()函数释放 dp 空间，该函数返回-1。调用 dsr_build_ip()函数为 dp 设置 IP：如果失败，调用 DEBUG 函数输出提示信息，跳转到 out_err。调用 dsr_opt_hdr_add()函数为 dp 添加 dsr 选项头部信息：如果失败,调用 DEBUG 函数输出提示信息,跳转到 out_err。调用 dsr_srt_opt_add()函数添加源路由选项头部,把 rreq 中读取的路由信息封装,如果失败,调用 DEBUG 函数输出提示信息，跳转到 out_err。调用 dsr_rrep_opt_add()函数添加路由应答选项头部，把 srt_to_me 中的路由信息封装：如果失败，调用 DEBUG 函数输出提示信息，跳转到 out_err。如果以上函数均执行成功，调用 XMIT()函数初始化并发送路由应答分组 dp，该函数返回 0。

2.3.3.2 dsr_rrep_opt_recv()函数分析



dsr_rrep_opt_recv()函数调用图

```
328  int NSCLASS dsr_rrep_opt_recv(struct dsr_pkt *dp, struct dsr_rrep_opt *rrep_opt)
329  {
330      struct in_addr myaddr, srt_dst;
331      struct dsr_srt *rrep_opt_srt;
332
333      if (!dp || !rrep_opt || dp->flags & PKT_PROMISC_RECV)
334          return DSR_PKT_ERROR;
335
336      if (dp->num_rrep_opts < MAX_RREP_OPTS)
337          dp->rrep_opt[dp->num_rrep_opts++] = rrep_opt;
338      else
339          return DSR_PKT_ERROR;
340
341      myaddr = my_addr();
342
343      srt_dst.s_addr = rrep_opt->addrs[DSR_RREP_ADDRS_LEN(rrep_opt) / sizeof(struct in_addr)];
344
345      rrep_opt_srt = dsr_srt_new(dp->dst, srt_dst,
346                     DSR_RREP_ADDRS_LEN(rrep_opt),
347                     (char *)rrep_opt->addrs);
348
349      if (!rrep_opt_srt)
350          return DSR_PKT_ERROR;
351
352      dsr_rtc_add(rrep_opt_srt, ConfValToUsecs(RouteCacheTimeout), 0);
353
354      /* Remove pending RREQs */
355      rreq_tbl_route_discovery_cancel(rrep_opt_srt->dst);
356
357      FREE(rrep_opt_srt);
358
```

```
359      if (dp->dst.s_addr == myaddr.s_addr) {
360          /*RREP for this node */
361
362          DEBUG("RREP for me!\n");
363
364          return DSR_PKT_SEND_BUFFERED;
365      }
366
367      DEBUG("I am not RREP destination\n");
368
369      /* Forward */
370      return DSR_PKT_FORWARD;
371  }
```

如同 dsr_rreq_send()函数，当某个节点收到来自邻接点的 rrep 路由应答消息后会调用 dsr_rrep_opt_recv()函数来对 rrep 消息进行处理。

dsr_rrep_send()的形参分别为：dp 和 rrep_opt，分别指代收到的 rrep 路由应答消息和存放路由应答选项的指针。函数执行会返回处理操作类型。

341-357

调用 dsr_rtc_add()函数将路由应答分组中的路由添加到该节点自己的路由缓存中。然后，调用 rreq_tbl_route_discovery_cancel()函数取消此次到达目的节点的路由发现，调用 FREE()函数释放 rrep_opt_srt 空间。

359-370

如果该路由请求分组 dp 的目的节点地址是该节点自己的地址，调用 DEBUG 函数输出提示信息，该函数返回 DSR_PKT_SEND_BUFFERED，缓存操作类型；否则，调用 DEBUG 函数输出提示信息，该函数返回 DSR_PKT_FORWARD，转发操作类型。

2.3.4 dsr_ack.c

2.3.4.1 dsr_ack_send()和 dsr_req_ack_send()函数分析

```
42  int NSCLASS dsr_ack_send(struct in_addr dst, unsigned short id)
43  {
44      struct dsr_pkt *dp;
45      struct dsr_ack_opt *ack_opt;
46      int len;
47      char *buf;
48
49      /* srt = dsr_rtc_find(my_addr(), dst); */
50
51  /*  if (!srt) { */
52  /*      DEBUG("No source route to %s\n", print_ip(dst.s_addr)); */
53  /*      return -1; */
54  /*  } */
55
56      len = DSR_OPT_HDR_LEN + /* DSR_SRT_OPT_LEN(srt) +  */ DSR_ACK_HDR_LEN;
57
58      dp = dsr_pkt_alloc(NULL);
59
60      dp->dst = dst;
61      /* dp->srt = srt; */
62      dp->nxt_hop = dst;  //dsr_srt_next_hop(dp->srt, 0);
63      dp->src = my_addr();
64
65      buf = dsr_pkt_alloc_opts(dp, len);
66
67      if (!buf)
68          goto out_err;
69
70      dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
71                    IP_HDR_LEN + len, IPPROTO_DSR, IPDEFTTL);
72
73      if (!dp->nh.iph) {
74          DEBUG("Could not create IP header\n");
75          goto out_err;
76      }
77
78      dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
79
80      if (!dp->dh.opth) {
81          DEBUG("Could not create DSR opt header\n");
82          goto out_err;
83      }
84
85      buf += DSR_OPT_HDR_LEN;
86      len -= DSR_OPT_HDR_LEN;
87
88      /* dp->srt_opt = dsr_srt_opt_add(buf, len, dp->srt); */
89
90  /*  if (!dp->srt_opt) { */
91  /*      DEBUG("Could not create Source Route option header\n"); */
92  /*      goto out_err; */
93  /*  } */
94
95  /*  buf += DSR_SRT_OPT_LEN(dp->srt); */
96  /*  len -= DSR_SRT_OPT_LEN(dp->srt); */
97
98      ack_opt = dsr_ack_opt_add(buf, len, dp->src, dp->dst, id);
99
100     if (!ack_opt) {
101         DEBUG("Could not create DSR ACK opt header\n");
102         goto out_err;
103     }
```

```
104
105         DEBUG("Sending ACK to %s id=%u\n", print_ip(dst), id);
106
107         dp->flags |= PKT_XMIT_JITTER;
108
109         XMIT(dp);
110
111         return 1;
112
113           out_err:
114         dsr_pkt_free(dp);
115         return -1;
116     }
```

dsr_ack_send()函数由节点成功收到数据分组时调用，向发送数据分组的节点发送一个确认收到分组的信息。

dsr_ack_send()的形参分别为:dst 和 id,分别指代 ack 信息的目标地址（即向该节点发送分组的节点地址）和消息的 id。成功执行返回 1，不成功返回-1.

44-63

变量初始化，dp 即为这个函数要发送的 ack 确认消息。为 dp 申请内存，申请失败打印失败信息并返回-1。指定 dp 的下一跳地址，原地址和目标地址。

70-103

分别利用 dsr_build_ip() 函数， dsr_opt_hdr_add() 函数和 dsr_ack_opt_add()函数为 dp 消息添加 IP 头部，dsr 选项头部和 ack 选项头部信息。如果失败，利用 DEBUG 打印错误信息，释放 dp 并返回-1。

107-111

根据上面指定 dp 的信息，转发 dp 消息。

```
209  int NSCLASS dsr_ack_req_send(struct in_addr neigh_addr, unsigned short id)
210  {
211      struct dsr_pkt *dp;
212      struct dsr_ack_req_opt *ack_req;
213      int len = DSR_OPT_HDR_LEN + DSR_ACK_REQ_HDR_LEN;
214      char *buf;
215
216      dp = dsr_pkt_alloc(NULL);
217
218      dp->dst = neigh_addr;
219      dp->nxt_hop = neigh_addr;
220      dp->src = my_addr();
221
222      buf = dsr_pkt_alloc_opts(dp, len);
223
224      if (!buf)
225          goto out_err;
226
227      dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
228                    IP_HDR_LEN + len, IPPROTO_DSR, 1);
229
230      if (!dp->nh.iph) {
231          DEBUG("Could not create IP header\n");
232          goto out_err;
233      }
234
235      dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
236
237      if (!dp->dh.opth) {
238          DEBUG("Could not create DSR opt header\n");
239          goto out_err;
240      }
241
242      buf += DSR_OPT_HDR_LEN;
243      len -= DSR_OPT_HDR_LEN;
244
245      ack_req = dsr_ack_req_opt_create(buf, len, id);
246
247      if (!ack_req) {
248          DEBUG("Could not create ACK REQ opt\n");
249          goto out_err;
250      }
251
252      DEBUG("Sending ACK REQ for %s id=%u\n", print_ip(neigh_addr), id);
253
254      XMIT(dp);
255
256      return 1;
257
258        out_err:
259      dsr_pkt_free(dp);
260      return -1;
261  }
```

dsr_req_ack_send()函数在节点收到 rreq 消息后调用，向上一节点发送收到 rreq 消息的 req_ack 确认消息。

由于 dsr_req_ack_send()和 dsr_ack_send()基本相同，这里不再进行分析。

2.3.4.2 dsr_ack_opt_recv()和 dsr_ack_req_opt_recv()函数分析

```
285  int NSCLASS dsr_ack_opt_recv(struct dsr_ack_opt *ack)
286  {
287      unsigned short id;
288      struct in_addr dst, src, myaddr;
289      int n;
290
291      if (!ack)
292          return DSR_PKT_ERROR;
293
294      myaddr = my_addr();
295
296      dst.s_addr = ack->dst;
297      src.s_addr = ack->src;
298      id = ntohs(ack->id);
299
300      DEBUG("ACK dst=%s src=%s id=%u\n", print_ip(dst), print_ip(src), id);
301
302      if (dst.s_addr != myaddr.s_addr)
303          return DSR_PKT_ERROR;
304
305      /* Purge packets buffered for this next hop */
306      n = maint_buf_del_all_id(src, id);
307
308      DEBUG("Removed %d packets from maint buf\n", n);
309
310      return DSR_PKT_NONE;
311  }
```

dsr_ack_opt_recv()在节点收到分组确认信息后调用，对收到的 ack 分组确认消息进行处理。

dsr_ack_opt_recv()的形参为：ack，指代该节点收到的 ack 消息。

302-303

判断 ack 分组确认消息的目的节点地址是否为该节点自己的地址，，如果不是，该函数返回 DSR_PKT_ERROR。

306-310

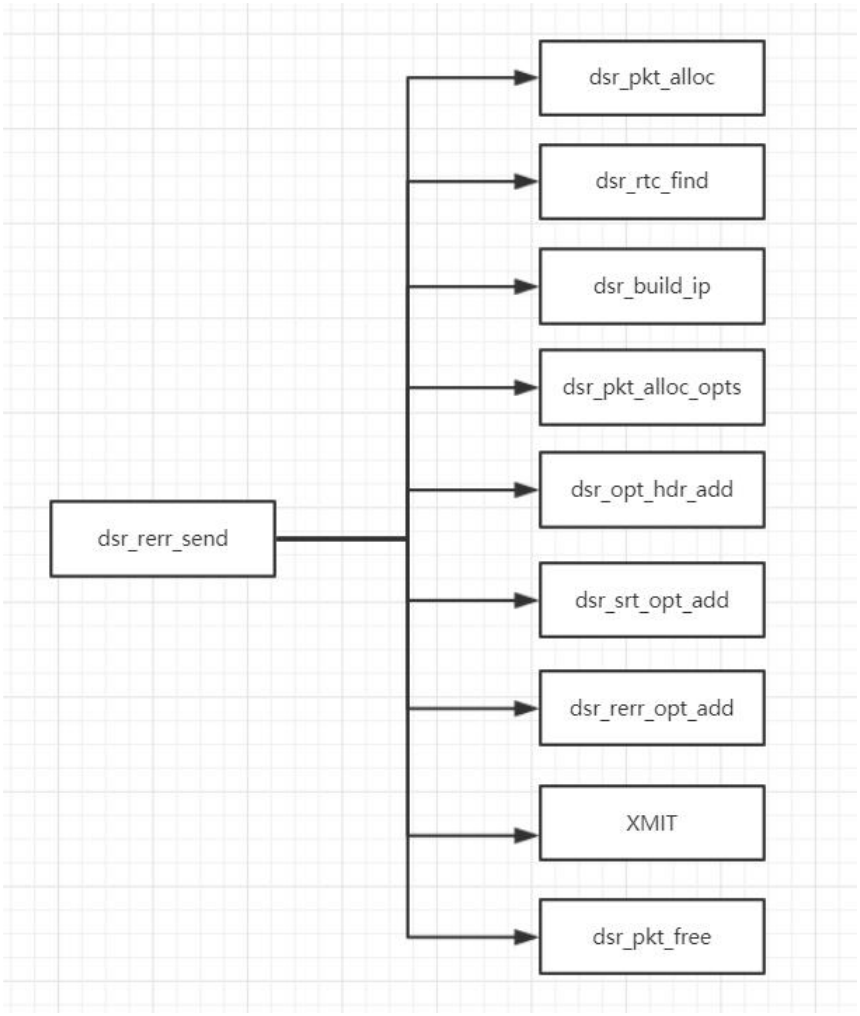
调用 maint_buf_del_all_id()函数将该 ack 确认分组所对应的数据分组从维护缓存器 maint_buf 中删除，该函数返回 DSR_PKT_NONE。

```
263  int NSCLASS dsr_ack_req_opt_recv(struct dsr_pkt *dp, struct dsr_ack_req_opt *ack_req_opt)
264  {
265      unsigned short id;
266
267      if (!ack_req_opt || !dp || dp->flags & PKT_PROMISC_RECV)
268          return DSR_PKT_ERROR;
269
270      dp->ack_req_opt = ack_req_opt;
271
272      id = ntohs(ack_req_opt->id);
273
274      if (!dp->srt_opt)
275          dp->prv_hop = dp->src;
276
277      DEBUG("src=%s prv=%s id=%u\n",
278            print_ip(dp->src), print_ip(dp->prv_hop), id);
279
280      dsr_ack_send(dp->prv_hop, id);
281
282      return DSR_PKT_NONE;
283  }
```

dsr _ack_req _opt_recv()在节点收到 ack_req 消息后调用。向上一节点发送一个 ack 消息。

2.3.5 dsr_rerr.c

2.3.5.1 dsr_rerr_send()函数分析



```c
int NSCLASS dsr_rerr_send(struct dsr_pkt *dp_trigg, struct in_addr unr_addr)
{
    struct dsr_pkt *dp;
    struct dsr_rerr_opt *rerr_opt;
    struct in_addr dst, err_src, err_dst, myaddr;
    char *buf;
    int n, len, i;

    myaddr = my_addr();

    if (!dp_trigg || dp_trigg->src.s_addr == myaddr.s_addr)
        return -1;

    if (!dp_trigg->srt_opt) {
        DEBUG("Could not find source route option\n");
        return -1;
    }

    if (dp_trigg->srt_opt->salv == 0)
        dst = dp_trigg->src;
    else
        dst.s_addr = dp_trigg->srt_opt->addrs[1];

    dp = dsr_pkt_alloc(NULL);

    if (!dp) {
        DEBUG("Could not allocate DSR packet\n");
        return -1;
    }

    dp->srt = dsr_rtc_find(myaddr, dst);

    if (!dp->srt) {
        DEBUG("No source route to %s\n", print_ip(dst));
        return -1;
    }

    len = DSR_OPT_HDR_LEN + DSR_SRT_OPT_LEN(dp->srt) +
        (DSR_RERR_HDR_LEN + 4) +
        DSR_ACK_HDR_LEN * dp_trigg->num_ack_opts;

    /* Also count in RERR opts in trigger packet */
    for (i = 0; i < dp_trigg->num_rerr_opts; i++) {
        if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAGE_COUNT))
            break;

        len += (dp_trigg->rerr_opt[i]->length + 2);
    }

    DEBUG("opt_len=%d SR: %s\n", len, print_srt(dp->srt));
    n = dp->srt->laddrs / sizeof(struct in_addr);
    dp->src = myaddr;
    dp->dst = dst;
    dp->nxt_hop = dsr_srt_next_hop(dp->srt, n);

    dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
                  IP_HDR_LEN + len, IPPROTO_DSR, IPDEFTTL);

    if (!dp->nh.iph) {
        DEBUG("Could not create IP header\n");
        goto out_err;
    }

    buf = dsr_pkt_alloc_opts(dp, len);
```

```
128     if (!buf)
129         goto out_err;
130
131     dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
132
133     if (!dp->dh.opth) {
134         DEBUG("Could not create DSR options header\n");
135         goto out_err;
136     }
137
138     buf += DSR_OPT_HDR_LEN;
139     len -= DSR_OPT_HDR_LEN;
140
141     dp->srt_opt = dsr_srt_opt_add(buf, len, 0, 0, dp->srt);
142
143     if (!dp->srt_opt) {
144         DEBUG("Could not create Source Route option header\n");
145         goto out_err;
146     }
147
148     buf += DSR_SRT_OPT_LEN(dp->srt);
149     len -= DSR_SRT_OPT_LEN(dp->srt);
150
151     rerr_opt = dsr_rerr_opt_add(buf, len, NODE_UNREACHABLE, dp->src,
152                     dp->dst, unr_addr,
153                     dp_trigg->srt_opt->salv);
154
155     if (!rerr_opt)
156         goto out_err;
157
158     buf += (rerr_opt->length + 2);
159     len -= (rerr_opt->length + 2);
160
```

```
161     /* Add old RERR options */
162     for (i = 0; i < dp_trigg->num_rerr_opts; i++) {
163
164         if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAGE_COUNT))
165             break;
166
167         memcpy(buf, dp_trigg->rerr_opt[i],
168                dp_trigg->rerr_opt[i]->length + 2);
169
170         len -= (dp_trigg->rerr_opt[i]->length + 2);
171         buf += (dp_trigg->rerr_opt[i]->length + 2);
172     }
173
174     /* TODO: Must preserve order of RERR and ACK options from triggering
175      * packet */
176
177     /* Add old ACK options */
178     for (i = 0; i < dp_trigg->num_ack_opts; i++) {
179         memcpy(buf, dp_trigg->ack_opt[i],
180                dp_trigg->ack_opt[i]->length + 2);
181
182         len -= (dp_trigg->ack_opt[i]->length + 2);
183         buf += (dp_trigg->ack_opt[i]->length + 2);
184     }
185
186     err_src.s_addr = rerr_opt->err_src;
187     err_dst.s_addr = rerr_opt->err_dst;
```

```
188
189         DEBUG("Send RERR err_src %s err_dst %s unr_dst %s\n",
190               print_ip(err_src),
191               print_ip(err_dst),
192               print_ip(*((struct in_addr *)rerr_opt->info)));
193
194         XMIT(dp);
195
196         return 0;
197
198     out_err:
199
200         dsr_pkt_free(dp);
201
202         return -1;
203
204     }
```

当节点向下一个节点多次转发数据分组后仍然没有收到下一节点的 ack 确认信息时，说明这两个节点之间的链路已经断了，这时要启动 DSR 的路由维护机制，该节点调用 dsr_reer_send()函数，创建并向这个分组的源节点发送 rerr 路由错误消息。

dsr_reer_send()的形参分别为：dp_trigg 和 unr_addr，分别指代转发出错的数据分组和无法到达的节点地址。函数执行成功返回 0，不成功返回-1。

73-74

如果 dp_trigg 为空，或者 dp_trigg 的源节点地址为该节点自己的地址，该函数返回-1。

76-79

如果 dp_trigg 的源路由选项为空，调用 DEBUG 函数输出提示信息，该函数返回-1。

86-91

调用 dsr_pkt_alloc 函数为路由错误分组 dp 分配空间：如果失败，调用 DEBUG 函数输出提示信息，该函数返回-1。

93-98

调用 dsr_rtc_find 函数在路由缓存 LC 中查找以该节点自己的地址为源节点、以 dp_trigg 的源节点为目的节点的源路由，以便 rerr 消息的转发：如果失败，调用 DEBUG 函数输出提示信息，该函数返回-1。

100-159

更新路由错误分组 dp 的 src、dst、nxt_hop、nh.iph、dh.opth、srt_opt 等消息转发信息和各 dsr 选项，并调用 dsr_rerr_opt_add()函数为路有错误分组 dp 添加选项信息。

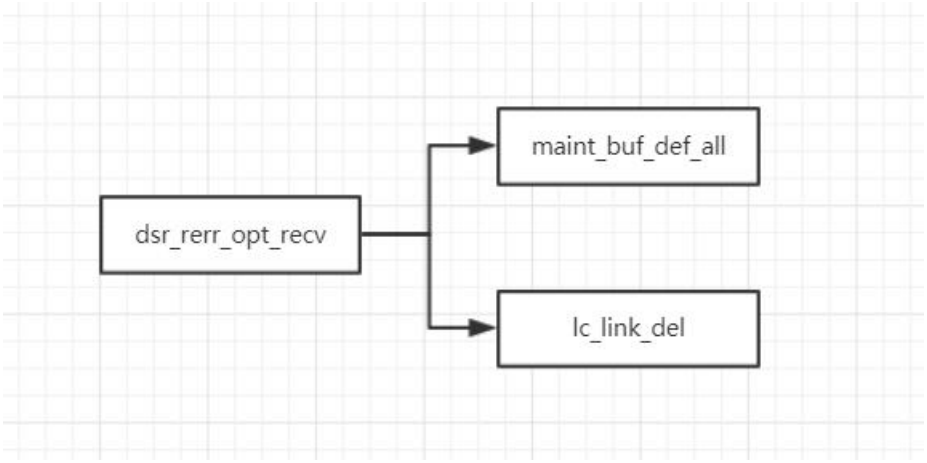
162-184

将与 dp_trigg 相关的旧路由错误分组选项信息和旧确认分组选项信息添加到路由错误分组 dp 中。

194-195

调用 XMIT 函数初始化并发送路由错误分组 dp，该函数返回 0。

2.3.5.2 dsr_rerr_opt_recv()函数分析



dsr_rerr_opt_recv()函数调用图

```
206  int NSCLASS dsr_rerr_opt_recv(struct dsr_pkt *dp, struct dsr_rerr_opt *rerr_opt)
207  {
208      struct in_addr err_src, err_dst, unr_addr;
209
210      if (!rerr_opt)
211          return -1;
212
213      dp->rerr_opt[dp->num_rerr_opts++] = rerr_opt;
214
215      switch (rerr_opt->err_type) {
216      case NODE_UNREACHABLE:
217          err_src.s_addr = rerr_opt->err_src;
218          err_dst.s_addr = rerr_opt->err_dst;
219
220          memcpy(&unr_addr, rerr_opt->info, sizeof(struct in_addr));
221
222          DEBUG("NODE_UNREACHABLE err_src=%s err_dst=%s unr=%s\n",
223                print_ip(err_src), print_ip(err_dst), print_ip(unr_addr));
224
225          /* For now we drop all unacked packets... should probably
226           * salvage */
227          maint_buf_del_all(err_dst);
228
229          /* Remove broken link from cache */
230          lc_link_del(err_src, unr_addr);
231
```

```
232             /* TODO: Check options following the RERR option */
233     /*      dsr_rtc_del(my_addr(), err_dst); */
234             break;
235         case FLOW_STATE_NOT_SUPPORTED:
236             DEBUG("FLOW_STATE_NOT_SUPPORTED\n");
237             break;
238         case OPTION_NOT_SUPPORTED:
239             DEBUG("OPTION_NOT_SUPPORTED\n");
240             break;
241     }
242
243     return 0;
244 }
245
```

当源节点或中间节点收到 rerr 路由错误消息时，节点调用 dsr_rerr_opt_recv()函数对 rerr 进行处理，对于节点不可达错误则删除自己的路由缓存中所有包括失效链路的路由。

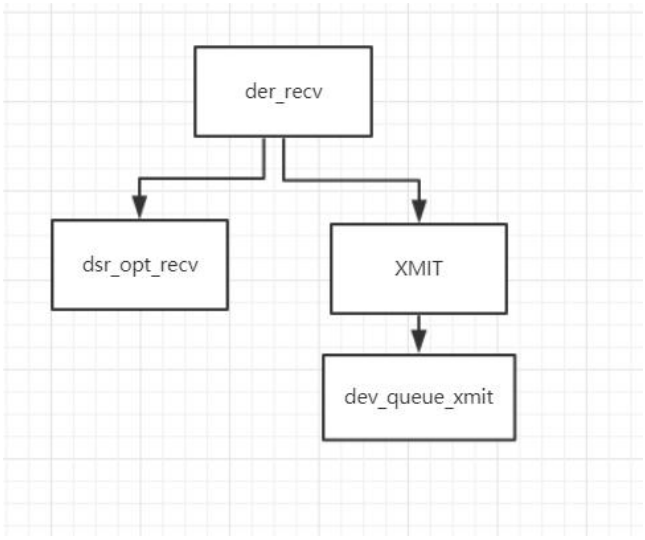
dsr_rerr_opt_recv()的形参分别为：dp 和 rerr_opt，分别指代收到的 rerr 路由错误消息和存放路由错误选项的指针。函数成功执行返回 0，不成功返回-1。

215-241

判断接收的 rerr 路由错误消息的错误类型。对于 NODE_UNREACHABLE(节点不可达)错误，就调用 maint_buf_del_all()函数从维护缓存器 maint_buf 中删除以该路由错误分组目的节点地址为下一跳节点的所有数据分组，并调用 lc_link_del()函数从路由链路缓存 LC 中删除以该路由错误分组源节点地址为源节点、不可达下一跳节点地址为目的节点的所有源路由；对于 FLOW_STATE_NOT_SUPPORTED(流状态不支持)和 OPTION_NOT_SUPPORTED(选项不支持)两种错误类型，直接打印错误信息。

2.3.6 dsr_io.c

2.3.6.1 dsr_recv()函数分析



dsr_recv 函数调用图

```
29  int NSCLASS dsr_recv(struct dsr_pkt *dp)
30  {
31      int i = 0, action;
32      int mask = DSR_PKT_NONE;
33
34      /* Process DSR Options */
35      action = dsr_opt_recv(dp);
36
37      /* Add mac address of previous hop to the neighbor table */
38
39      if (dp->flags & PKT_PROMISC_RECV) {
40          dsr_pkt_free(dp);
41          return 0;
42      }
43      for (i = 0; i < DSR_PKT_ACTION_LAST; i++) {
44
45          switch (action & mask) {
46          case DSR_PKT_NONE:
47              break;
48          case DSR_PKT_DROP:
49          case DSR_PKT_ERROR:
50              DEBUG("DSR_PKT_DROP or DSR_PKT_ERROR\n");
51              dsr_pkt_free(dp);
52              return 0;
53          case DSR_PKT_SEND_ACK:
54              /* Moved to dsr-ack.c */
55              break;
56          case DSR_PKT_SRT_REMOVE:
57              //DEBUG("Remove source route\n");
58              // Hmm, we remove the DSR options when we deliver a
59              //packet
60              //dsr_opt_remove(dp);
61              break;
62          case DSR_PKT_FORWARD:
63
```

```
64  #ifdef NS2
65              if (dp->nh.iph->ttl() < 1)
66  #else
67              if (dp->nh.iph->ttl < 1)
68  #endif
69              {
70                  DEBUG("ttl=0, dropping!\n");
71                  dsr_pkt_free(dp);
72                  return 0;
73              } else {
74                  DEBUG("Forwarding %s %s nh %s\n",
75                        print_ip(dp->src),
76                        print_ip(dp->dst), print_ip(dp->nxt_hop));
77                  XMIT(dp);
78                  return 0;
79              }
80              break;
81          case DSR_PKT_FORWARD_RREQ:
82              XMIT(dp);
83              return 0;
84          case DSR_PKT_SEND_RREP:
85              /* In dsr-rrep.c */
86              break;
87          case DSR_PKT_SEND_ICMP:
88              DEBUG("Send ICMP\n");
89              break;
90          case DSR_PKT_SEND_BUFFERED:
91              if (dp->rrep_opt) {
92                  struct in_addr rrep_srt_dst;
93                  int i;
94
```

```
95                  for (i = 0; i < dp->num_rrep_opts; i++) {
96                      rrep_srt_dst.s_addr = dp->rrep_opt[i]->addrs[DSR_RREP_ADDRS_LEN(dp->rrep_opt[i]) / sizeof(struct in_addr)];
97
98                      send_buf_set_verdict(SEND_BUF_SEND, rrep_srt_dst);
99                  }
100             }
101                 break;
102         case DSR_PKT_DELIVER:
103             DEBUG("Deliver to DSR device\n");
104             DELIVER(dp);
105             return 0;
106         case 0:
107             break;
108         default:
109             DEBUG("Unknown pkt action\n");
110         }
111         mask = (mask << 1);
112     }
113
114     dsr_pkt_free(dp);
115
116     return 0;
117 }
```

协议动用 dsr_rec()函数来规定对于接收数据分组的处理，转发或者丢弃。dsr_recv()的形参为:dp，指代接收打数据分组。成功执行返回 0。

35-63

调用 dsr_opt_recv()函数将协议对数据分组的操作类型赋值给 action，并针对 action 的不同操作类型，对数据分组进行不同操作。比如对于 DSR_PKT-ERROR,DSR_PKT_DROP 直接调用 dsr_pkt_free()函数把数据分组释放并丢弃，对于 DSR_PKTFORWARD，给 dp 的邻接点地址添加 mac 地址。

65-110

对于已经到达 ttl 生命周期的数据分组协议也会将它释放丢弃。而针对其他

的操作类型，协议就会调用 XMIT()或 DELIVER()函数转发该数据分组。

114　在该函数的最后，还是会释放这个数据分组。

2.3.6.2 dsr-start_xmit()函数分析

```
119   void NSCLASS dsr_start_xmit(struct dsr_pkt *dp)
120   {
121       int res;
122
123       if (!dp) {
124           DEBUG("Could not allocate DSR packet\n");
125           return;
126       }
127
128       dp->srt = dsr_rtc_find(dp->src, dp->dst);
129
130       if (dp->srt) {
131
132           if (dsr_srt_add(dp) < 0) {
133               DEBUG("Could not add source route\n");
134               goto out;
135           }
136           /* Send packet */
137
138           XMIT(dp);
139
140           return;
141
142       } else {
143   #ifdef NS2
144           res = send_buf_enqueue_packet(dp, &DSRUU::ns_xmit);
145   #else
146           res = send_buf_enqueue_packet(dp, &dsr_dev_xmit);
147   #endif
148           if (res < 0) {
149               DEBUG("Queueing failed!\n");
150               goto out;
151           }
152           res = dsr_rreq_route_discovery(dp->dst);
153
154           if (res < 0)
155               DEBUG("RREQ Transmission failed...");
156
157           return;
158       }
159     out:
160       dsr_pkt_free(dp);
161   }
```

dsr_start_xmit()函数被调用来给要发送的数据分组添加源路由头部信息。dsr_start_xmit()的形参为：dp，指代源节点要发送的数据分组。无返回值。

128-142

调用 dsr_rtc_find()函数在本节点路由缓存中查询是否存在满足数据分组转发的路由信息。找到对应路由调用 dsr_srt_add()函数把该源路由信息添加到数据分组的头部，添加失败报错并丢地数据分组，添加成功则依照源路由转发数据分组。

143-157

如果没有查询到信息，调用 send_buf_enqueue_packet() 和 dsr_rrep_route_discovery()函数把该数据分组暂时放到发送缓冲中，并开始路由发现。