目录

第一章　引言

动态源路由协议（DSR）是一种简单有效的路由协议，专为多跳无线网络设计移动节点的 ad hoc 网络。

DSR 允许网络完全自组织和自配置，而不需要任何现有的网络基础设施或管理。

协议的所有方面完全按需操作，允许 DSR 的路由包开销，自动扩展到只需要对当前使用的路由中的更改作出反应。

该协议允许多个路由到任何目的地，并允许每个发送方选择和控制路由其数据包时使用的路由，例如，用于负载平衡或增强健壮性。DSR 协议的其他优点包括易于保证无环路路由，在包含单向链路的网络中操作，在路由中仅使用"软状态"，并且恢复非常迅速当网络中的路由更改时。

DSR 协议主要是为大约 200 个节点的移动自组织网络设计的，它被设计成即使在非常高的移动率下也能很好地工作。

协议是由两个主要的***"路由发现"*** 机制和***"路由维护"*** 机制实现的，它们协同工作，允许节点发现并维护到 ad hoc 网络中任意目的地的路由。

-路由发现 是节点 S 希望将数据包发送到目标节点 D 时，获得到 D 的源路由的机制。路由发现仅在 S 试图将数据包发送到 D 且到 D 的路由未知时使用。

-路由维护 是指当路由维护表明源路由故障，S 可以尝试使用任何其他到 D 的路由，或可以调用路由发现再次找到一个新的到 D 的路由。路由维护仅在 S 已实际发送数据包到 D 时使用。

第二章　代码介绍

头文件定义及功能

文件	说明
Time.h、List.h、Tbl.h	计时器、链表相关函数
Maint-buf.h、send-buf.h	缓冲区维护相关函数
Link-cache.h	维护缓存表相关函数
Dsr-opt.h	DSR 选项管理、构造 DSR 报文
Dsr.h	设置数据包最大长度、自旋锁、DSR 节点初始化
Dsr-dev.h、dsr-io.h	发送接收 DSR 数据包
Dsr-rerr.h	DSR 路由错误收发
Dsr-rrep.h	DSR 路由回答收发、超时、添加等
Dsr-rreq.h	DSR 路由请求收发、超时、发现等
Dsr-srt.h	DSR 源路由相关
Neigh.h	邻节点信息维护

表 2.1 讨论的文件

第三章　数据结构介绍

3.1 DSR 分组结构

DSR 协议使用一个结构特殊的 DSR 首部携带信息，这个 DSR 首部可以被任何已经存在的 IP 分组携带。DSR 首部由一个固定尺寸的结构体后附加若干个 DSR 选项构成。整个 DSR 首部的结束标志由包含在固定结构体中的首部长度来表示。DSR 首部应该被插入到现存 IP 分组的 IP 头和任何的传输层协议头之间。DSR 分组结构如图所示。

图 3.1.1 DSR 在网络协议栈中的位置

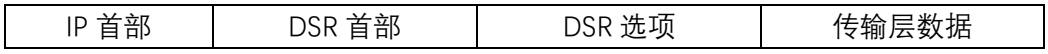


图 3.1.2 DSR 分组

图 3.1.3 DSR 分组结构

插入 DSR 头以后，IP 首部的结构无需改动但 IP 头中的若干个域需要修改以区别 DSR 分组与普通 IP 分组：IP 首部中的 protocol 域需要修改为 DSR 协议标识;由于 DSR 信息的插入，整个分组的长度即 IP 首部中的 total_length 域也要被修改: IP 头的目的地址 addr 应该被修改为 DSR 源路由中的下一跳地址，如果是 DSR 路由请求分组则应该将其置为本地广播地

址:IP 首部中任意其它域的改动都会引起首部校验和（Checksum）的变化，需要用函数 ip_send_check 重新计算。

3.2 DSR 首部

DSR 首部携带了每个 DSR 数据包中所必有的一些信息。格式如下图 3.2.1 所示。

next header	reserved	flag	payload Length
Option			

图 3.2.1 DSR 首部结构

```
01. /* The DSR options header (always comes first) */
02. struct dsr_opt_hdr {
03.     u_int8_t nh;
04. #if defined(__LITTLE_ENDIAN_BITFIELD)
05. 
06.     u_int8_t res:7;
07.     u_int8_t f:1;
08. #elif defined (__BIG_ENDIAN_BITFIELD)
09.     u_int8_t f:1;
10.     u_int8_t res:7;
11. #else
12. #error  "Please fix <asm/byteorder.h>"
13. #endif
14.     u_int16_t p_len;    /* payload length */
15. #ifdef NS2
16.     static int offset_;
17. 
18.     inline static int &offset() {
19.         return offset_;
20.     }
21.     inline static dsr_opt_hdr *access(const Packet * p) {
22.         return (dsr_opt_hdr *) p->access(offset_);
23.     }
24. 
25.     int size() {
26.         return ntohs(p_len) + sizeof(struct dsr_opt_hdr);
27.     }
28. #endif              /* NS2 */
29.     struct dsr_opt option[0];
30. };
```

图 3.2.2 DSR 首部结构定义

第 3 行 *nh* ： next header 为下一个首部，与 IP 首部中的协议字段取值相同。

第 6 行 *res* ： reserved 为保留位占 7 位，置为 0

第 7 行 *f* ： flag 位标志位，占 1 位

第 14 行 *p_len* ： payload Length 为所有 DSR 选项的长度

第 16 行 *offset* 为偏移量

第 29 行 *option[0]* 为 DSR 选项，记录不同 DSR 选项的控制信息

3.3 路由请求选项

路由请求选项的格式如下图 3.3.1 所示。

type	length	id
target address		
address[1]		
address[2]		
……		
address[n]		

图 3.3.1 路由请求选项结构

路由请求选项定义如下:

```
1. struct dsr_rreq_opt {
2.     u_int8_t type;
3.     u_int8_t length;
4.     u_int16_t id;
5.     u_int32_t target;
6.     u_int32_t addrs[0];
7. };
```

图 3.3.2 路由请求选项结构定义

第 2 行 *type* 为选项类型
第 3 行 *length* 为选项长度
第 4 行 *id* 为路由 ID 号，由发起路由时源节点产生一个唯一的 id，如果在路由缓存中存在相同的 id 号，则丢弃防止多次转发；否则转发并记录到缓存的路由请求表中。
第 5 行 *target* 为目的地址
第 6 行 *addrs[0]* 用于记录路由信息，数据报到达第 i 个节点时在 addrs[i]中记录对应的路由信息。

3.4 路由回复选项

路由应答选项的格式如图 3.4.1 所示。

type	length	last hop external	reserved
Address[1]			
Address[2]			
……			

Address[n]

图 3.4.1 路由回复选项结构

```
1.  struct dsr_rrep_opt {
2.      u_int8_t type;
3.      u_int8_t length;
4.  #if defined(__LITTLE_ENDIAN_BITFIELD)
5.      u_int8_t res:7;
6.      u_int8_t l:1;
7.  #elif defined (__BIG_ENDIAN_BITFIELD)
8.      u_int8_t l:1;
9.      u_int8_t res:7;
10. #else
11. #error  "Please fix <asm/byteorder.h>"
12. #endif
13.     u_int32_t addrs[0];
14. };
```

图 3.4.2 路由应答选项结构定义

第 5 行 *res* ： reserved，为保留位，占 7 位，置为 0

第 6 行 / :last hop external，表示节点是否在 DSR 网络内部，0 表示最后一跳节点在 DSR 网络内部，1 表示最后一跳在 DSR 网络外部。选取路由时优先选择在网络内部的节点，因此在路由缓存中优先选择 l 标志位为 0 的路由。

3.5 路由错误选项

路由错误选项的格式如图 3.5.1 所示。

type	length	error_type	reserved	slavage
error source address				
error destination address				
information				

图 3.5.1 路由错误选项结构

```
1.  struct dsr_rerr_opt {
2.      u_int8_t type;
3.      u_int8_t length;
4.      u_int8_t err_type;
5.  #if defined(__LITTLE_ENDIAN_BITFIELD)
6.      u_int8_t res:4;
```

```
7.      u_int8_t salv:4;
8.  #elif defined (__BIG_ENDIAN_BITFIELD)
9.      u_int8_t res:4;
10.     u_int8_t salv:4;
11. #else
12. #error  "Please fix <asm/byteorder.h>"
13. #endif
14.     u_int32_t err_src;
15.     u_int32_t err_dst;
16.     char info[0];
17. };
```

图 3.5.2 路由错误选项结构定义

第 4 行 *err_type* 为错误类型
第 6 行 *res* ：reserved 为保留位，占 4 位
第 7 行 *salv* ：salvage，占 4 位，表示该数据包重新发送的次数，根据次数决定是否丢弃数据包
第 14 行 *err_src*：error source，为路由错误的源地址
第 15 行 *err_dst*：error destination，为路由错误的目的地址
第 16 行 *Info[0]*：information，用于记录路由错误信息

3.6 源路由选项

源路由选项的格式如图 3.6.1 所示。

type	length	First Hop External	Last Hop External	Reserved	Slavage	Segments Left
address[1]						
address[2]						
.....						
address[n]						

图 3.6.1 源路由选项结构

```
1.  struct dsr_srt_opt {
2.      u_int8_t type;
3.      u_int8_t length;
4.  #if defined(__LITTLE_ENDIAN_BITFIELD)
5.  /* TODO: Fix bit/byte order */
6.      u_int16_t f:1;
7.      u_int16_t l:1;
8.      u_int16_t res:4;
9.      u_int16_t salv:4;
10.     u_int16_t sleft:6;
```

```
11. #elif defined (__BIG_ENDIAN_BITFIELD)
12.     u_int16_t f:1;
13.     u_int16_t l:1;
14.     u_int16_t res:4;
15.     u_int16_t salv:4;
16.     u_int16_t sleft:6;
17. #else
18. #error  "Please fix <asm/byteorder.h>"
19. #endif
20.     u_int32_t addrs[0];
21. };
```

图 3.6.2 源路由选项结构定义

第 6 行 *f*： First Hop External，置 0 表示第一跳节点在 DSR 网络内部，置 1 表示第一跳在 DSR 网络外部

第 7 行 *l*：为 Last Hop External，置 0 表示第一跳节点在 DSR 网络内部，置 1 表示第 一跳在 DSR 网络外部

第 10 行 *sleft*： Segments Left，为路由剩余的跳数

第 20 行 *addrs[0]*：维护的源路由路径

3.7 DSR 数据包

```
01. struct dsr_pkt {
02.     struct in_addr src; /* IP level data */   源ip                        struct in_addr {
03.     struct in_addr dst;         目的ip                                in_addr_t s_addr;
04.     struct in_addr nxt_hop;         下一跳ip                    };
05.     struct in_addr prv_hop;     上一跳ip                                    表示一个32位的IPv4地址。in_addr_t一般
06.     int flags;       标记位                                  为32位的unsigned int，其字节顺序为网络字节序，
07.     int salvage;  发送次数限制                                              即该无符号数采用大端字节序。每8位表示一个IP地址中的一个数值。
08. #ifdef NS2
09.     union {
10.         struct hdr_mac *ethh;
11.         unsigned char *raw;
12.     } mac;
13.     struct hdr_ip ip_data;
14.     union {
15.         struct hdr_ip *iph;
16.         char *raw;
17.     } nh;
18. #else
19.     union {
20.         struct ethhdr *ethh;
21.         char *raw;
22.     } mac;
23.     union {
24.         struct iphdr *iph;
25.         char *raw;
26.     } nh;
27.     char ip_data[60];
28. #endif
29.     struct {
30.         union {
31.             struct dsr_opt_hdr *opth;
32.             char *raw;
33.         };
34.         char *tail, *end;
35.     } dh;
36.
37.     int num_rrep_opts, num_rerr_opts, num_rreq_opts, num_ack_opts;
38.     struct dsr_srt_opt *srt_opt;
39.     struct dsr_rreq_opt *rreq_opt;  /* Can only be one */
40.     struct dsr_rrep_opt *rrep_opt[MAX_RREP_OPTS];
41.     struct dsr_rerr_opt *rerr_opt[MAX_RERR_OPTS];
42.     struct dsr_ack_opt *ack_opt[MAX_ACK_OPTS];
43.     struct dsr_ack_req_opt *ack_req_opt;
44.     struct dsr_srt *srt;    /* Source route */
45.
46.     int payload_len;
47. #ifdef NS2
48.     AppData *payload;
49.     Packet *p;
50. #else
51.     char *payload;
52.     struct sk_buff *skb;
53. #endif
```

3.7.1 DSR 数据包结构定义

第 2 行 *struct in_addr src;*：Source，源 IP 地址

第 3 行 *struct in_addr dst;*：Destination，目的端 IP 地址
第 4 行 *struct in_addr nxt_hop;*：Next Hop，下一跳 IP 地址
第 5 行 *struct in_addr prv_hop;*：Previous Hop，上一跳 IP 地址
第 9-12 行 *union{}mac;*：指向以太网帧首部，联合体使我们可以以结构体的方式访问以太网帧首部，或者直接读取以太网帧首部的数据
第 13 行 *struct hdr_ip ip_data;*：IP 数据报中的数据
第 14-17 行 *union{}nh;*：指向 IP 首部，联合体使我们可以以结构体的方式访问 IP 首部，或者直接读取 IP 首部的数据
第 29-35 行 *struct{}dh;*：指向 DSR 首部，联合体使我们可以以结构体的方式访问 DSR 首部，或者直接读取 DSR 首部的数据
第 46 行 *int payload_len;*：负载长度
第 51 行 *char *payload;*：负载
第 52 行 *struct sk_buff *skb;*：分片结构体。其中 struct sk_buff 是 Linux 网络系统中的核心结构体，所有数据包的封装及解封都是在这个结构体的基础上进行的

3.8 源路由结构

源路由是指发送的数据包沿途经过的节点，也就是路由的路径。

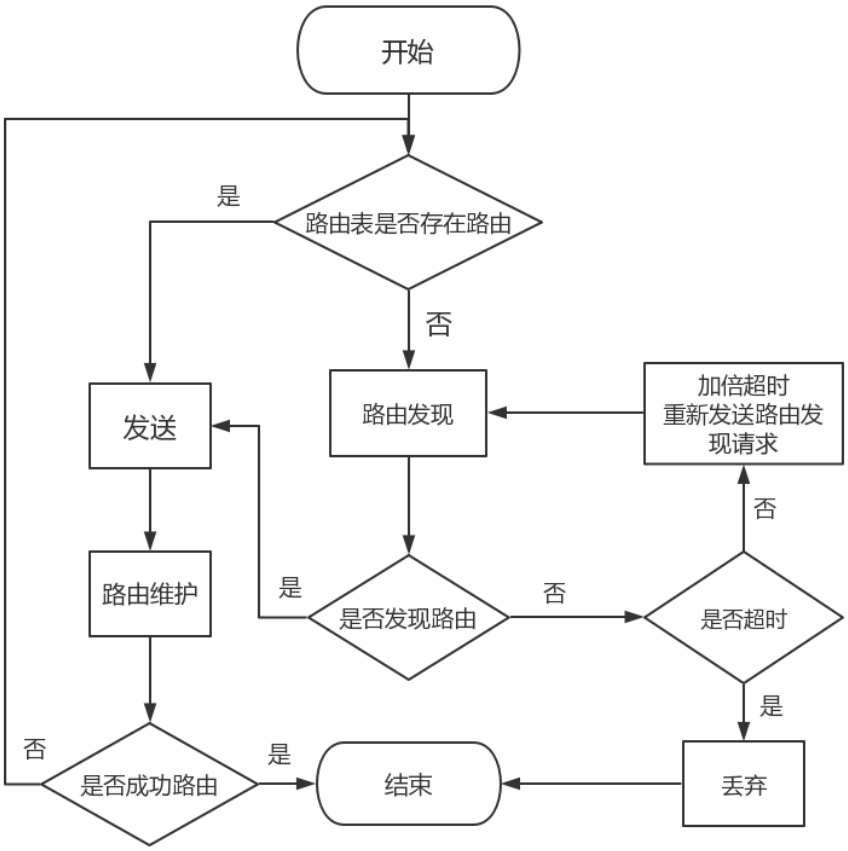
```
1.  struct dsr_srt {
2.      struct in_addr src;
3.      struct in_addr dst;
4.      unsigned short flags;
5.      unsigned short index;
6.      unsigned int laddrs;    /* length in bytes if addrs */
7.      struct in_addr addrs[0];    /* Intermediate nodes */
8.  };
```

3.8.1 源路由结构定义

第 6 行 *unsigned int laddrs;*：Length of Address，表示地址总长度
第 7 行 *struct in_addr addrs[0];*：用于存放中间经过的节点

第四章　源路由协议的实现过程

4.1 流程图



4.2 路由发现

当某些节点产生了一个新的数据包，想要发送给目的节点时，源节点会将源路由放在数据包的头部，给出该数据包到目的节点所遵循的有序跳数。

通常情况下，发送方会从自己的路由缓存中获得一个合适的源路由；如果没有在缓存中找到路由，它就会启动路由发现协议动态地找到一个到达目的节点的新路由。

具体过程如下：

- 路由请求：源节点向邻居节点广播路由请求（RREQ），请求到达目的节点的路由；
- 路由记录：记录从源节点到目的节点中的中间节点请求ID，中间节点接收RREQ后，将自己的地址附在路由记录中，由 addrs[0]数据维护源路由路径；
- 中间节点处理：中间节点维护序列对列表：<源节点地址，请求 ID>；
- 重复 RREQ 检测：如果检测到重复的 RREQ，则中间节点丢弃该消息；
- 路由应答：目的节点收到 RREQ 后，给源节点返回路由应答（RREP）信息，拷贝 RREQ 消息中的路由记录并反向存入 addrs[]数组，源节点收到 RREP 后，在本地路由缓存中缓存路由信息。

生成 DSR 数据包 在 *dsr-pkt.c* 中实现　另有 Packet * p 为参数的不列举

```
01. struct dsr_pkt *dsr_pkt_alloc(struct sk_buff *skb)
02. {
03.     struct dsr_pkt *dp;
04.     int dsr_opts_len = 0;
05. 
06.     dp = (struct dsr_pkt *)MALLOC(sizeof(struct dsr_pkt), GFP_ATOMIC);
07. 
08.     if (!dp)
09.         return NULL;
10. 
11.     memset(dp, 0, sizeof(struct dsr_pkt));
12. 
13.     if (skb) {
14.     /*  skb_unlink(skb); */
15. 
16.         dp->skb = skb;
17. 
18.         dp->mac.raw = skb->mac.raw;
19.         dp->nh.iph = skb->nh.iph;
20. 
21.         dp->src.s_addr = skb->nh.iph->saddr;
22.         dp->dst.s_addr = skb->nh.iph->daddr;
23. 
24.         if (dp->nh.iph->protocol == IPPROTO_DSR) {
25.             struct  *opth;
26.             int n;
27. 
28.             opth = (struct dsr_opt_hdr *)(dp->nh.raw + (dp->nh.iph->ihl << 2));
29.             dsr_opts_len = ntohs(opth->p_len) + DSR_OPT_HDR_LEN;
30. 
31.             if (!dsr_pkt_alloc_opts(dp, dsr_opts_len)) {
32.                 FREE(dp);
33.                 return NULL;
34.             }
35. 
36.             memcpy(dp->dh.raw, (char *)opth, dsr_opts_len);
37. 
38.             n = dsr_opt_parse(dp);
39. 
40.             DEBUG("Packet has %d DSR option(s)n", n);
41.         }
42. 
43.         dp->payload = dp->nh.raw +
44.             (dp->nh.iph->ihl << 2) + dsr_opts_len;
45. 
46.         dp->payload_len = ntohs(dp->nh.iph->tot_len) -
47.             (dp->nh.iph->ihl << 2) - dsr_opts_len;
48. 
49.         if (dp->payload_len)
50.             dp->flags |= PKT_REQUEST_ACK;
51.     }
52.     return dp;
53. }
```

6-11 为数据包申请需要的内存空间，然后将内容全部置零
13-51 如果传入的 skb 指针不为空，则数据包会根据 skb 所指向的内容进行初始化

为 DSR 选项首部分配内存和构造 IP 首部 在 *dsr-opt.c* 中实现

6-7 判断缓冲区能否放下 DSR 选项首部，不能则返回 NULL
9-14 对选项首部信息进行初始化
16　返回指向这个 DSR 选项首部的指针
26-39 构造 IP 首部，如果存在 skb，则赋值，不存在则设为默认值
41-42 将 IP 首部长度设置为网络字节序
44　设置 IP 首部校验和
46　返回指向 IP 首部的指针

```
01. struct dsr_opt_hdr *dsr_opt_hdr_add(char *buf, unsigned int len,
02.                     unsigned int protocol)
03. {
04.     struct dsr_opt_hdr *opt_hdr;
05. 
06.     if (len < DSR_OPT_HDR_LEN)
07.         return NULL;
08. 
09.     opt_hdr = (struct dsr_opt_hdr *)buf;
10. 
11.     opt_hdr->nh = protocol;
12.     opt_hdr->f = 0;
13.     opt_hdr->res = 0;
14.     opt_hdr->p_len = htons(len - DSR_OPT_HDR_LEN);
15. 
16.     return opt_hdr;
17. }
18. 
19. #ifdef __KERNEL__
20. struct iphdr *dsr_build_ip(struct dsr_pkt *dp, struct in_addr src,
21.                 struct in_addr dst, int ip_len, int tot_len,
22.                 int protocol, int ttl)
23. {
24.     struct iphdr *iph;
25. 
26.     dp->nh.iph = iph = (struct iphdr *)dp->ip_data;
27. 
28.     if (dp->skb && dp->skb->nh.raw) {
29.         memcpy(dp->ip_data, dp->skb->nh.raw, ip_len);
30.     } else {
31.         iph->version = IPVERSION;
32.         iph->ihl = 5;
33.         iph->tos = 0;
34.         iph->id = 0;
35.         iph->frag_off = 0;
36.         iph->ttl = (ttl ? ttl : IPDEFTTL);
37.         iph->saddr = src.s_addr;
38.         iph->daddr = dst.s_addr;
39.     }
40. 
41.     iph->tot_len = htons(tot_len);
42.     iph->protocol = protocol;
43. 
44.     ip_send_check(iph);
45. 
46.     return iph;
47. }
48. #endif
```

解析 dsr 的选项　此处对于服务器接收到 SYN 包时根据类型判断包的类型

```
        case DSR_OPT_RREQ:
            if (dp->num_rreq_opts == 0)
                dp->rreq_opt = (struct dsr_rreq_opt *)dopt;
#ifndef NS2
            else
                DEBUG("ERROR: More than one RREQ option!!\n");
#endif
            break;
        case DSR_OPT_RREP:
            if (dp->num_rrep_opts < MAX_RREP_OPTS)
                dp->rrep_opt[dp->num_rrep_opts++] = (struct dsr_rrep_opt *)
#ifndef NS2
            else
                DEBUG("Maximum RREP opts in one packet reached\n");
#endif
            break;
        case DSR_OPT_RERR:
            if (dp->num_rerr_opts < MAX_RERR_OPTS)
                dp->rerr_opt[dp->num_rerr_opts++] = (struct dsr_rerr_opt *)
#ifndef NS2
            else
                DEBUG("Maximum RERR opts in one packet reached\n");
#endif
```

DSR 选项的接收　　在 *dsr-opt.c* 中实现

11 获取 dsr_node 的 ifaddr 即 ipaddr

17-22 如果 datapacket 的目的地址与 destnode 的 addr 相同且 datapacket 的 header 满足要求(DATA_PACKET()与 PT_PING 均没找到)或 datapacket 的目的地址与 destnode 的 addr 相同且 dp 的负载 len 不为 0,action 与 DSR_PKT_DELIVER 进行或运算

26-27 获取 DSR 选项首部长度、选项

31- 如果还有选项，判断选项类型

```
int NSCLASS dsr_opt_recv(struct dsr_pkt *dp)
{
    int dsr_len, l;
    int action = 0;
    struct dsr_opt *dopt;
    struct in_addr myaddr;

    if (!dp)
        return DSR_PKT_ERROR;

    myaddr = my_addr();

    /* Packet for us ? */
#ifdef NS2
    //DEBUG("Next header=%s\n", packet_info.name((packet_t)dp->dh.opth->nh))

    if (dp->dst.s_addr == myaddr.s_addr &&
        (DATA_PACKET(dp->dh.opth->nh) || dp->dh.opth->nh == PT_PING))
        action |= DSR_PKT_DELIVER;
#else
    if (dp->dst.s_addr == myaddr.s_addr && dp->payload_len != 0)
        action |= DSR_PKT_DELIVER;
#endif
    dsr_len = dsr_pkt_opts_len(dp);

    l = DSR_OPT_HDR_LEN;
    dopt = DSR_GET_OPT(dp->dh.opth);

    //DEBUG("Parsing DSR packet l=%d dsr_len=%d\n", l, dsr_len);

    while (l < dsr_len && (dsr_len - l) > 2) {
        //DEBUG("dsr_len=%d l=%d\n", dsr_len, l);
        switch (dopt->type) {
```

处理 DSR 协议数据包　　在 *dsr-io.c* 中实现

```
int NSCLASS dsr_recv(struct dsr_pkt *dp)
```

```
    for (i = 0; i < DSR_PKT_ACTION_LAST; i++) {

        switch (action & mask) {
        case DSR_PKT_NONE:
            break;
        case DSR_PKT_DROP:
        case DSR_PKT_ERROR:
            DEBUG("DSR_PKT_DROP or DSR_PKT_ERROR\n");
            dsr_pkt_free(dp);
            return 0;
        case DSR_PKT_SEND_ACK:
            /* Moved to dsr-ack.c */
            break;
        case DSR_PKT_SRT_REMOVE:
            //DEBUG("Remove source route\n");
            // Hmm, we remove the DSR options when we deliver a
            //packet
            //dsr_opt_remove(dp);
            break;
        case DSR_PKT_FORWARD:

#ifdef NS2
            if (dp->nh.iph->ttl() < 1)
#else
            if (dp->nh.iph->ttl < 1)
#endif
            {
                DEBUG("ttl=0, dropping!\n");
                dsr_pkt_free(dp);
                return 0;
            } else {
                DEBUG("Forwarding %s %s nh %s\n",
                      print_ip(dp->src),
                      print_ip(dp->dst), print_ip(dp->nxt_hop));
                XMIT(dp);
                return 0;
```

17-24 根据选项中的信息，如果数据包是空、丢失、错误，则返回错误信息，并释放该数据包的空间。

34-50 如果是转发的数据包，判断数据包的 ttl，如果 ttl 为 0，输出错误信息并释放空间，否则输出源 IP,目的 IP 及下一跳 IP，调用 XMIT（）发送数据包

发送 DSR 数据包　　在 *dsr-io.c* 中实现

```
01.  void NSCLASS dsr_start_xmit(struct dsr_pkt *dp)
02.  {
03.      int res;
04.
05.      if (!dp) {
06.          DEBUG("Could not allocate DSR packet\n");
07.          return;
08.      }
09.
10.      dp->srt = dsr_rtc_find(dp->src, dp->dst);
11.
12.      if (dp->srt) {
13.
14.          if (dsr_srt_add(dp) < 0) {
15.              DEBUG("Could not add source route\n");
16.              goto out;
17.          }
18.          /* Send packet */
19.
20.          XMIT(dp);
21.
22.          return;
23.
24.      } else {
25.  #ifdef NS2
26.          res = send_buf_enqueue_packet(dp, &DSRUU::ns_xmit);
27.  #else
28.          res = send_buf_enqueue_packet(dp, &dsr_dev_xmit);
29.  #endif
30.          if (res < 0) {
31.              DEBUG("Queueing failed!\n");
32.              goto out;
33.          }
34.          res = dsr_rreq_route_discovery(dp->dst);
35.
36.          if (res < 0)
37.              DEBUG("RREQ Transmission failed...");
38.
39.          return;
40.      }
41.        out:
42.      dsr_pkt_free(dp);
43.  }
```

10 调用 dsr_rtc_find(),判断是否存在到达目的节点的源路由

12-22 如果存在源路由，调用 XMIT（）发送数据包

24-40 如果不存在源路由，则将这个数据包添加到发送队列，进行路由发现

DSR 处理源路由请求

在 *dsr-srt.c* 中实现了对源路由选项的处理功能。

路由缩减

```
01. struct dsr_srt *dsr_srt_shortcut(struct dsr_srt *srt, struct in_addr a1,
02.                  struct in_addr a2)
03. {
04.     struct dsr_srt *srt_cut;
05.     int i, j, n, n_cut, a1_num, a2_num;
06. 
07.     if (!srt)
08.         return NULL;
09. 
10.     a1_num = a2_num = -1;
11. 
12.     n = srt->laddrs / sizeof(struct in_addr);
13.     if (srt->src.s_addr == a1.s_addr)
14.         a1_num = 0;
15. 
16.     /* Find out how between which node indexes to shortcut */
17.     for (i = 0; i < n; i++) {
18.         if (srt->addrs[i].s_addr == a1.s_addr)
19.             a1_num = i + 1;
20.         if (srt->addrs[i].s_addr == a2.s_addr)
21.             a2_num = i + 1;
22.     }
23. 
24.     if (srt->dst.s_addr == a2.s_addr)
25.         a2_num = i + 1;
26. 
27.     n_cut = n - (a2_num - a1_num - 1);
28. 
29.     srt_cut = (struct dsr_srt *)MALLOC(sizeof(struct dsr_srt) +
30.                       (n_cut*sizeof(struct in_addr)),
31.                       GFP_ATOMIC);
32. 
33.     if (!srt_cut)
34.         return NULL;
35. 
36.     srt_cut->src = srt->src;
37.     srt_cut->dst = srt->dst;
38.     srt_cut->laddrs = n_cut * sizeof(struct in_addr);
39. 
40.     if (srt_cut->laddrs == 0)
41.         return srt_cut;
42. 
43.     j = 0;
44. 
45.     for (i = 0; i < n; i++) {
46.         if (i + 1 > a1_num && i + 1 < a2_num)
47.             continue;
48.         srt_cut->addrs[j++] = srt->addrs[i];
49.     }
50. 
51.     return srt_cut;
52. }
```

12 记录总节点数

13-25 a1 为源节点，a2 为目的节点，记录 a1、a2 的下标

27 a1、a2 区间外有多少节点

29 分配空间

36-49 构造削短后的路径

51 返回缩短后的源路由

接收 DSR 源路由选项

```
01. int NSCLASS dsr_srt_opt_recv(struct dsr_pkt *dp, struct dsr_srt_opt *srt_opt)
02. {
03.     struct in_addr next_hop_intended;
04.     struct in_addr myaddr = my_addr();
05.     int n;
06.
07.     if (!dp || !srt_opt)
08.         return DSR_PKT_ERROR;
09.
10.     dp->srt_opt = srt_opt;
11.
12.     /* We should add this source route info to the cache... */
13.     dp->srt = dsr_srt_new(dp->src, dp->dst, srt_opt->length,
14.                           (char *)srt_opt->addrs);
15.
16.     if (!dp->srt) {
17.         DEBUG("Create source route failed\n");
18.         return DSR_PKT_ERROR;
19.     }
20.     n = dp->srt->laddrs / sizeof(struct in_addr);
21.
22.     DEBUG("SR: %s sleft=%d\n", print_srt(dp->srt), srt_opt->sleft);
23.
24.     /* Copy salvage field */
25.     dp->salvage = dp->srt_opt->salv;
26.
27.     next_hop_intended = dsr_srt_next_hop(dp->srt, srt_opt->sleft);
28.     dp->prv_hop = dsr_srt_prev_hop(dp->srt, srt_opt->sleft - 1);
29.     dp->nxt_hop = dsr_srt_next_hop(dp->srt, srt_opt->sleft - 1);
30.
31.     DEBUG("next_hop=%s prev_hop=%s next_hop_intended=%s\n",
32.           print_ip(dp->nxt_hop),
33.           print_ip(dp->prv_hop), print_ip(next_hop_intended));
34.
35.     neigh_tbl_add(dp->prv_hop, dp->mac.ethh);
36.
37.     lc_link_add(my_addr(), dp->prv_hop,
38.             ConfValToUsecs(RouteCacheTimeout), 0, 1);
39.
40.     dsr_rtc_add(dp->srt, ConfValToUsecs(RouteCacheTimeout), 0);
41.
42.     /* Automatic route shortening - Check if this node is the
43.      * intended next hop. If not, is it part of the remaining
44.      * source route? */
45.          对路由缩减的条件进行判断：如果下一跳的地址不是当前地址，且源路由中存在当前地址且其不在缩减列表中，
46.          则可以进行路由缩减。
47.
48.     if (next_hop_intended.s_addr != myaddr.s_addr &&
49.         dsr_srt_find_addr(dp->srt, myaddr, srt_opt->sleft) &&
50.         !grat_rrep_tbl_find(dp->src, dp->prv_hop)) {
51.         struct dsr_srt *srt, *srt_cut;
52.
53.         /* Send Grat RREP */
54.         DEBUG("Send Gratuitous RREP to %s\n", print_ip(dp->src));
55.
56.         srt_cut = dsr_srt_shortcut(dp->srt, dp->prv_hop, myaddr);
57.
58.         if (!srt_cut)
59.             return DSR_PKT_DROP;
60.
61.         DEBUG("shortcut: %s\n", print_srt(srt_cut));
62.
63.         /* srt = dsr_rtc_find(myaddr, dp->src); */
64.         if (srt_cut->laddrs / sizeof(struct in_addr) == 0)
65.             srt = dsr_srt_new_rev(srt_cut);
66.         else
67.             srt = dsr_srt_new_split_rev(srt_cut, myaddr);
68.
69.         if (!srt) {
70.             DEBUG("No route to %s\n", print_ip(dp->src));
71.             FREE(srt_cut);
72.             return DSR_PKT_DROP;
73.         }
74.         DEBUG("my srt: %s\n", print_srt(srt));
75.
76.                 在路由缩减列表中添加本次路由，并且发送一个路由回复用于通知新的路由变更。
77.
78.         grat_rrep_tbl_add(dp->src, dp->prv_hop);
79.
80.         dsr_rrep_send(srt, srt_cut);
81.
82.         FREE(srt_cut);
83.         FREE(srt);
84.     }
85.
86.     if (dp->flags & PKT_PROMISC_RECV)
87.         return DSR_PKT_DROP;
88.
89.     if (srt_opt->sleft == 0)
90.         return DSR_PKT_SRT_REMOVE;
91.
92.     if (srt_opt->sleft > n) {
93.         // Send ICMP parameter error
94.         return DSR_PKT_SEND_ICMP;
95.     }
96.
97.     srt_opt->sleft--;
98.
99.     /* TODO: check for multicast address in next hop or dst */
100.    /* TODO: check MTU and compare to pkt size */
101.
102.        如果不能进行路由缩减，则转发数据包。
103.    return DSR_PKT_FORWARD;
```

48-51 对路由缩减的条件进行判断：如果下一跳的地址不是当前地址，且源路由中存在当前地址且其不在缩减列表中，则可以进行路由缩减。

78-80 在路由缩减列表中添加本次路由，并且发送一个路由回复用于通知新的路由变更。

103 如果不能进行路由缩减，则转发数据包

生成路由请求

节点不知道到目的节点的路由时，通过路由发现来发现路由 在 *dsr-rreq.c* 中实现

路由发现请求

```
01.  int NSCLASS dsr_rreq_route_discovery(struct in_addr target)
02.  {
03.      struct rreq_tbl_entry *e;
04.      int ttl, res = 0;
05.      struct timeval expires;
06.
07.  #define TTL_START 1
08.
09.      DSR_WRITE_LOCK(&rreq_tbl.lock);
10.
11.      e = (struct rreq_tbl_entry *)__tbl_find(&rreq_tbl, &target, crit_addr);
12.
13.      if (!e)
14.          e = __rreq_tbl_add(target);
15.      else {
16.          /* Put it last in the table */
17.          __tbl_detach(&rreq_tbl, &e->l);
18.          __tbl_add_tail(&rreq_tbl, &e->l);
19.      }
20.
21.      if (!e) {
22.          res = -ENOMEM;
23.          goto out;
24.      }
25.
26.      if (e->state == STATE_IN_ROUTE_DISC) {
27.          DEBUG("Route discovery for %s already in progress\n",
28.                print_ip(target));
29.          goto out;
30.      }
31.      DEBUG("Route discovery for %s\n", print_ip(target));
32.
33.      gettime(&e->last_used);
34.      e->ttl = ttl = TTL_START;
35.      /* The draft does not actually specify how these Request Timeout values
36.       * should be used... ??? I am just guessing here. */
37.
38.      if (e->ttl == 1)
39.          e->timeout = ConfValToUsecs(NonpropRequestTimeout);
40.      else
41.          e->timeout = ConfValToUsecs(RequestPeriod);
42.
43.      e->state = STATE_IN_ROUTE_DISC;
44.      e->num_rexmts = 0;
45.
46.      expires = e->last_used;
47.      timeval_add_usecs(&expires, e->timeout);
48.
49.      set_timer(e->timer, &expires);
50.
51.      DSR_WRITE_UNLOCK(&rreq_tbl.lock);
52.
53.      dsr_rreq_send(target, ttl);
```

9 写锁
11 在路由表查找 e
13-19 e 不在表中，添加到 rreq 表
e 在表中，断开连接，把 e 放在表最后
26-31 判断 e 的状态是否在路由 DISC
34-44 设置 ttl 为 1，设置超时，设置计时器发送次数
51 解锁
53 调用 dsr_rreq_send 构造 rreq 包发送 packet

发送路由请求

```
01. int NSCLASS dsr_rreq_send(struct in_addr target, int ttl)
02. {
03.     struct dsr_pkt *dp;
04.     char *buf;
05.     int len = DSR_OPT_HDR_LEN + DSR_RREQ_HDR_LEN;            分配长度
06.
07.     dp = dsr_pkt_alloc(NULL);                                    分配packet
08.
09.     if (!dp) {
10.         DEBUG("Could not allocate DSR packet\n");
11.         return -1;
12.     }
13.     dp->dst.s_addr = DSR_BROADCAST;
14.     dp->nxt_hop.s_addr = DSR_BROADCAST;
15.     dp->src = my_addr();                                  赋源地址、设置广播
16.
17.     buf = dsr_pkt_alloc_opts(dp, len);
18.
19.
20.     if (!buf)
21.         goto out_err;
22.
23.     dp->nh.iph =
24.         dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN, IP_HDR_LEN + len,
25.              IPPROTO_DSR, ttl);                               构造IP
26.
27.     if (!dp->nh.iph)
28.         goto out_err;
29.
30.     dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);        构造选项头
31.
32.     if (!dp->dh.opth) {
33.         DEBUG("Could not create DSR opt header\n");
34.         goto out_err;
35.     }
36.
37.     buf += DSR_OPT_HDR_LEN;
38.     len -= DSR_OPT_HDR_LEN;                                  去选项头
39.
40.     dp->rreq_opt = dsr_rreq_opt_add(buf, len, target, ++rreq_seqno); 添加选项头
41.
42.     if (!dp->rreq_opt) {
43.         DEBUG("Could not create RREQ opt\n");
44.         goto out_err;
45.     }
46. #ifdef NS2
47.     DEBUG("Sending RREQ src=%s dst=%s target=%s ttl=%d iph->saddr()=%d\n",
48.           print_ip(dp->src), print_ip(dp->dst), print_ip(target), ttl,
49.           dp->nh.iph->saddr());
50. #endif
51.
52.      dp->flags |= PKT_XMIT_JITTER;                              与0X08进行或运算
53.
54.      XMIT(dp);                                              发送packet
55.
56.      return 0;
57.
58.        out_err:
59.      dsr_pkt_free(dp);
60.
61.      return -1;
62. }
```

5 分配长度 为 DSR 选项首部长度+路由请求首部长度

7 分配 packet 的内存

13-15 将 packet 的的目的地址和下一跳设置为广播，设置源地址

23-30 构造 IP 首部和选项首部

37-40 将选项首部添加到路由请求选项中

52 与 0x08 进行或运算

54 发送 packet

接收路由请求

```
struct in_addr myaddr;
struct in_addr trg;
struct dsr_srt *srt_rev, *srt_rc;
int action = DSR_PKT_NONE;
int i, n;

if (!dp || !rreq_opt || dp->flags & PKT_PROMISC_RECV)
    return DSR_PKT_DROP;

dp->num_rreq_opts++;

if (dp->num_rreq_opts > 1) {
    DEBUG("More than one RREQ opt!!! - Ignoring\n");
    return DSR_PKT_ERROR;
}
```

丢弃 packet 和 packet 选项过多出错

7-12 判断该路由请求是否已存在在源路由中，若存在就丢弃 packet，不存在就添加到源路由中

14-23 新建源路由及错误处理

26 源路由反转，便于发送路由回复

35-41 设置反向路由前一跳

43 想邻节点添加信息

46 发送缓冲区的 packet

48-61 若路由请求选项的目的地址为当前地址，更新 IP 首部，发送路由回复

67-74 若源路由包含当前地址，丢弃 packet

77 构造路由缓存

85-87 拼接 packet 的源路由和路由缓存

96-99 如果存在重复的源路由，释放拼接的源路由

107 发送缓存的路由回复

路由请求超时处理

```
01. void NSCLASS rreq_tbl_timeout(unsigned long data)
02. {
03.     struct rreq_tbl_entry *e = (struct rreq_tbl_entry *)data;
04.     struct timeval expires;
05. 
06.     if (!e)
07.         return;
08. 
09.     tbl_detach(&rreq_tbl, &e->l);
10. 
11.     DEBUG("RREQ Timeout dst=%s timeout=%lu rexmts=%d \n",
12.           print_ip(e->node_addr), e->timeout, e->num_rexmts);
13. 
14.     if (e->num_rexmts >= ConfVal(MaxRequestRexmt)) {
15.         DEBUG("MAX RREQs reached for %s\n", print_ip(e->node_addr));
16. 
17.         e->state = STATE_IDLE;
18. 
19. /*      DSR_WRITE_UNLOCK(&rreq_tbl); */
20.         tbl_add_tail(&rreq_tbl, &e->l);
21.         return;
22.     }
23. 
24.     e->num_rexmts++;
25. 
26.     /* if (e->ttl == 1) */
27. /*      e->timeout = ConfValToUsecs(RequestPeriod);  */
28. /*  else */
29.     e->timeout *= 2; /* Double timeout */
30. 
31.     e->ttl *= 2;     /* Double TTL */
32. 
33.     if (e->ttl > MAXTTL)
34.         e->ttl = MAXTTL;
35. 
36.     if (e->timeout > ConfValToUsecs(MaxRequestPeriod))
37.         e->timeout = ConfValToUsecs(MaxRequestPeriod);
38. 
39.     gettime(&e->last_used);
40. 
41.     dsr_rreq_send(e->node_addr, e->ttl);
42. 
43.     expires = e->last_used;
44.     timeval_add_usecs(&expires, e->timeout);
45. 
46.     /* Put at end of list */
47.     tbl_add_tail(&rreq_tbl, &e->l);
48. 
49.     set_timer(e->timer, &expires);
50. }
```

11-12 报错目的地址 超时时间 计时器重传次数
14-15 重传>=最大请求重传次数
17 将状态设为闲置
20 将其加到表尾
29-37 加倍超时、加倍 ttl 若超过最大设置为最大
41 再次发送 rreq
43-44 失效时间设置为 e->上一个时间
47 将这个数据包放在表尾
49 设置 table 的定时器

发送数据包 在 *dsr-dev.c* 中实现:

先前的数据包都是通过 XMIT（）函数发送
XMIT（）实际为 dsr_dev_xmit()的宏定义

```
int dsr_dev_xmit(struct dsr_pkt *dp)
```

```
12.     if (dp->flags & PKT_REQUEST_ACK)
13.         maint_buf_add(dp);
14.
15.     dsr_node_lock(dsr_node);
16.
17.     if (dsr_node->slave_dev)
18.         slave_dev = dsr_node->slave_dev;
19.     else {
20.         dsr_node_unlock(dsr_node);
21.         goto out_err;
22.     }
23.     dsr_node_unlock(dsr_node);
24.
25.     skb = dsr_skb_create(dp, slave_dev);
26.
27.     if (!skb) {
28.         DEBUG("Could not create skb!n");
29.         goto out_err;
30.     }
31.
32.     /* Create hardware header */
33.     if (dsr_hw_header_create(dp, skb) < 0) {
34.         DEBUG("Could not create hardware headern");
35.         dev_kfree_skb_any(skb);
36.         goto out_err;
37.     }
38.
39.     len = skb->len;
40.     dst.s_addr = skb->nh.iph->daddr;
41.
42.     DEBUG("Sending %d bytes data_len=%d %s : %sn",
43.           len, skb->data_len,
44.           print_eth(skb->mac.raw),
45.           print_ip(dst));
46.
47.     /* TODO: Should consider using ip_finish_output instead */
48.     res = dev_queue_xmit(skb);
49.
50.     if (res < 0)
51.         goto out_err;
52.
53.     dsr_node_lock(dsr_node);
54.     dsr_node->stats.tx_packets++;
55.     dsr_node->stats.tx_bytes += len;
56.     dsr_node_unlock(dsr_node);
57.
58. out_err:
59.     dsr_pkt_free(dp);
60.
61.     return res;
```

25　根据 DSR 数据包的数据创建一个 sk_buff 数据包
33-45　完成 MAC 首部的封装
48　调用 dev_queue_xmit()将数据包发送出去
54-55 统计发送的数据包信息
59　如果中间某个过程产生错误，则释放该数据包的空间

生成路由回复　　在 *dsr-rrep.c* 中实现

```
01. int NSCLASS dsr_rrep_send(struct dsr_srt *srt, struct dsr_srt *srt_to_me)
02. {
03.     struct dsr_pkt *dp = NULL;
04.     char *buf;
05.     int len, ttl, n;
06.
07.     if (!srt || !srt_to_me)
08.         return -1;
09.
10.     dp = dsr_pkt_alloc(NULL);
11.
12.     if (!dp) {
13.         DEBUG("Could not allocate DSR packet\n");
14.         return -1;
15.     }                                                   判断错误
16.
17.     dp->src = my_addr();
18.     dp->dst = srt->dst;
19.
20.     if (srt->laddrs == 0)                                       源路由为空
21.         dp->nxt_hop = dp->dst;                  packet的下一跳为目的地址
22.     else
23.         dp->nxt_hop = srt->addrs[0];            packet的下一跳为源路由第一跳
24.
25.     len = DSR_OPT_HDR_LEN + DSR_SRT_OPT_LEN(srt) +
26.         DSR_RREP_OPT_LEN(srt_to_me)/*  + DSR_OPT_PAD1_LEN */;
Len为dsr选项首部+源路由选项+RREP选项 /* + PAD选项*/
27.     n = srt->laddrs / sizeof(struct in_addr);           跳数
28.
29.     DEBUG("srt: %s\n", print_srt(srt));
30.     DEBUG("srt_to_me: %s\n", print_srt(srt_to_me));
31.     DEBUG("next_hop=%s\n", print_ip(dp->nxt_hop));
32.     DEBUG
33.         ("IP_HDR_LEN=%d DSR_OPT_HDR_LEN=%d DSR_SRT_OPT_LEN=%d DSR_RREP_OPT_LEN=%d
34.          IP_HDR_LEN, DSR_OPT_HDR_LEN, DSR_SRT_OPT_LEN(srt),
35.          DSR_RREP_OPT_LEN(srt_to_me), DSR_OPT_PAD1_LEN, len);
36.
37.     ttl = n + 1;                                    跳数+1  ttl为1 丢弃
38.
39.     DEBUG("TTL=%d, n=%d\n", ttl, n);
40.
41.     buf = dsr_pkt_alloc_opts(dp, len);          分配buf
42.
43.     if (!buf)
44.         goto out_err;
45.
46.     dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
47.                               IP_HDR_LEN + len, IPPROTO_DSR, ttl);          构造IP header
48.
55.     dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);  构造选项首部
56.
57.     if (!dp->dh.opth) {
58.         DEBUG("Could not create DSR options header\n");
59.         goto out_err;
60.     }
61.
62.     buf += DSR_OPT_HDR_LEN;             加选项首部到buf
63.     len -= DSR_OPT_HDR_LEN;
64.
65.     /* Add the source route option to the packet */
66.     dp->srt_opt = dsr_srt_opt_add(buf, len, 0, dp->salvage, srt);     源路由选项加到packet
67.
68.     if (!dp->srt_opt) {
69.         DEBUG("Could not create Source Route option header\n");
70.         goto out_err;
71.     }
72.
73.     buf += DSR_SRT_OPT_LEN(srt);
74.     len -= DSR_SRT_OPT_LEN(srt);
75.
76.     dp->rrep_opt[dp->num_rrep_opts++] =
77.         dsr_rrep_opt_add(buf, len, srt_to_me);      把源路由选项添加到rrep选项数组
78.
79.     if (!dp->rrep_opt[dp->num_rrep_opts - 1]) {
80.         DEBUG("Could not create RREP option header\n");
81.         goto out_err;
82.     }
83.
84.     /* TODO: Should we PAD? The rrep struct is padded and aligned
85.      * automatically by the compiler... How to fix this? */
86.
87. /*  buf += DSR_RREP_OPT_LEN(srt_to_me); */
88. /*  len -= DSR_RREP_OPT_LEN(srt_to_me); */
89.
90. /*  pad1_opt = (struct dsr_pad1_opt *)buf; */
91. /*  pad1_opt->type = DSR_OPT_PAD1; */
92.
93.     /* if (ConfVal(UseNetworkLayerAck)) */
94. /*      dp->flags |= PKT_REQUEST_ACK; */
95.
96.     dp->flags |= PKT_XMIT_JITTER;               与0x08即1000做或运算
97.
98.     XMIT(dp);                           发送packet
99.
```

7-15 判断错误

20-23 若源路由为空，设置 packet 的下一跳为目的地址，否则设置为源路由第一跳

25-28 获取 packet 长度，dsr 选项首部+源路由选项+RREP 选项 获取跳数 n

47-83 构造 IP 首部、构造选项首部、将选项首部加到 buf、源路由选项加到 packet 中、将源路由选项加入 RREP 选项数组

96 与 0x08 做或运算

98 发送 packet

4.3 路由维护

路由维护会在以下情况下启动：

- 当源节点向目的节点发送数据时，需要对当前路由的可用情况进行监控
- 当网络拓扑变化导致路由故障时，切换到另一条路由或者重新发起路由发现过程

通过逐跳验证机制来判断路由错误信息，从而删除失效的路由：

- 如果数据分组被重发了最大次数仍然没有收到下一跳的确认，则节点向源端发送路由错误信息，并指明中断的链路，源端将该路由从路由缓存中删除
- 如果源端路由缓存中存在另一条到目的节点的路由，则使用该路由重发分组，否则重新开始路由发现过程。

处理确认请求选项和处理确认选项　　在 *dsr-ack.c* 中实现

DSR 使用*确认请求选项(ACK Request)*来判断邻居节点是否存活。当给一个邻居节点重发多次 ACK 请求却仍然没有收到回复的话，该节点就会认为其到邻居节点的链路已经被破坏，从而从路由缓存中删除这条路由，并且给上一次收到 ACK 之后在这条链路上发送过数据包的节点发送路由错误 。

当数据包顺利送达目标节点之后，就会产生一个确认（ACK）选项，而源节点收到这个选项后，就从维护缓存中删除发送给 ACK 源端的数据包。

```
01. int NSCLASS dsr_ack_req_opt_recv(struct dsr_pkt *dp, struct dsr_ack_req_opt *ack_req_opt)
02. {
03.     unsigned short id;
04. 
05.     if (!ack_req_opt || !dp || dp->flags & PKT_PROMISC_RECV)
06.         return DSR_PKT_ERROR;
07. 
08.     dp->ack_req_opt = ack_req_opt;
09. 
10.     id = ntohs(ack_req_opt->id);
11. 
12.     if (!dp->srt_opt)
13.         dp->prv_hop = dp->src;
14. 
15.     DEBUG("src=%s prv=%s id=%un",
16.           print_ip(dp->src), print_ip(dp->prv_hop), id);
17. 
18.     dsr_ack_send(dp->prv_hop, id);
19. 
20.     return DSR_PKT_NONE;
21. }
22. 
23. int NSCLASS dsr_ack_opt_recv(struct dsr_ack_opt *ack)
24. {
25.     unsigned short id;
26.     struct in_addr dst, src, myaddr;
27.     int n;
28. 
29.     if (!ack)
30.         return DSR_PKT_ERROR;
31. 
32.     myaddr = my_addr();
33. 
34.     dst.s_addr = ack->dst;
35.     src.s_addr = ack->src;
36.     id = ntohs(ack->id);
37. 
38.     DEBUG("ACK dst=%s src=%s id=%un", print_ip(dst), print_ip(src), id);
39. 
40.     if (dst.s_addr != myaddr.s_addr)
41.         return DSR_PKT_ERROR;
42. 
43.     /* Purge packets buffered for this next hop */
44.     n = maint_buf_del_all_id(src, id);
45. 
46.     DEBUG("Removed %d packets from maint bufn", n);
47. 
48.     return DSR_PKT_NONE;
49. }
```

8　将 ACK 请求选项赋值给 packet

10　网络字节序转换为主机字节序

12-13　如果没有源路由选项，将数据包的前一跳设置为源节点

18　将数据包发送到前一跳

34-36　将 ACK 源地址、目的地址和 id 号赋值

44　从维护列表中删除 id 为 ACK 数据包 id 的 packet

生成路由错误选项

当发生路由错误时，比如网络拓扑结构发生变化，数据包无法送达，节点需要生成并发送错误通知给数据包的源节点。

发送路由错误报文　　在 *dsr-rerr.c* 中实现

```
01. int NSCLASS dsr_rerr_send(struct dsr_pkt *dp_trigg, struct i
02. {
03.     struct dsr_pkt *dp;
04.     struct dsr_rerr_opt *rerr_opt;
05.     struct in_addr dst, err_src, err_dst, myaddr;
06.     char *buf;
07.     int n, len, i;
08.
09.     myaddr = my_addr();
10.
11.     if (!dp_trigg || dp_trigg->src.s_addr == myaddr.s_addr)
12.         return -1;
13.
14.     if (!dp_trigg->srt_opt) {
15.         DEBUG("Could not find source route option\n");
16.         return -1;
17.     }
18.
19.     if (dp_trigg->srt_opt->salv == 0)
20.         dst = dp_trigg->src;
21.     else
22.         dst.s_addr = dp_trigg->srt_opt->addrs[1];
23.
24.     dp = dsr_pkt_alloc(NULL);
25.
26.     if (!dp) {
27.         DEBUG("Could not allocate DSR packet\n");
28.         return -1;
29.     }
30.
31.     dp->srt = dsr_rtc_find(myaddr, dst);
32.
33.     if (!dp->srt) {
34.         DEBUG("No source route to %s\n", print_ip(dst));
35.         return -1;
36.     }
37.
38.     len = DSR_OPT_HDR_LEN + DSR_SRT_OPT_LEN(dp->srt) +
39.         (DSR_RERR_HDR_LEN + 4) +
40.         DSR_ACK_HDR_LEN * dp_trigg->num_ack_opts;
41.
42.     /* Also count in RERR opts in trigger packet */
43.     for (i = 0; i < dp_trigg->num_rerr_opts; i++) {
44.         if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAG
45.             break;
46.
47.         len += (dp_trigg->rerr_opt[i]->length + 2);
48.     }
49.
50.     DEBUG("opt_len=%d SR: %s\n", len, print_srt(dp->srt));
51.     n = dp->srt->laddrs / sizeof(struct in_addr);
52.     dp->src = myaddr;
53.     dp->dst = dst;
54.     dp->nxt_hop = dsr_srt_next_hop(dp->srt, n);
55.
56.     dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
57.                               IP_HDR_LEN + len, IPPROTO_DSR, IPDEFTTL);
58.
59.     if (!dp->nh.iph) {
60.         DEBUG("Could not create IP header\n");
61.         goto out_err;
62.     }
63.
64.     buf = dsr_pkt_alloc_opts(dp, len);
65.
66.     if (!buf)
67.         goto out_err;
68.
69.     dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);
70.
71.     if (!dp->dh.opth) {
72.         DEBUG("Could not create DSR options header\n");
73.         goto out_err;
74.     }
75.
76.     buf += DSR_OPT_HDR_LEN;
77.     len -= DSR_OPT_HDR_LEN;
78.
79.     dp->srt_opt = dsr_srt_opt_add(buf, len, 0, 0, dp->srt);
80.
81.     if (!dp->srt_opt) {
82.         DEBUG("Could not create Source Route option header\n");
83.         goto out_err;
84.     }
85.
86.     buf += DSR_SRT_OPT_LEN(dp->srt);
87.     len -= DSR_SRT_OPT_LEN(dp->srt);
88.
89.     rerr_opt = dsr_rerr_opt_add(buf, len, NODE_UNREACHABLE, dp->src,
90.                                 dp->dst, unr_addr,
91.                                 dp_trigg->srt_opt->salv);
92.
93.     if (!rerr_opt)
94.         goto out_err;
95.
96.     buf += (rerr_opt->length + 2);
97.     len -= (rerr_opt->length + 2);
98.
99.     /* Add old RERR options */
100.    for (i = 0; i < dp_trigg->num_rerr_opts; i++) {
101.
102.        if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAGE_COUNT))
103.            break;
104.
105.        memcpy(buf, dp_trigg->rerr_opt[i],
106.               dp_trigg->rerr_opt[i]->length + 2);
107.
108.        len -= (dp_trigg->rerr_opt[i]->length + 2);
109.        buf += (dp_trigg->rerr_opt[i]->length + 2);
110.    }
111.
112.    /* TODO: Must preserve order of RERR and ACK options from triggering
113.     * packet */
114.
115.    /* Add old ACK options */
116.    for (i = 0; i < dp_trigg->num_ack_opts; i++) {
117.        memcpy(buf, dp_trigg->ack_opt[i],
118.               dp_trigg->ack_opt[i]->length + 2);
119.
120.        len -= (dp_trigg->ack_opt[i]->length + 2);
121.        buf += (dp_trigg->ack_opt[i]->length + 2);
122.    }
123.
124.    err_src.s_addr = rerr_opt->err_src;
125.    err_dst.s_addr = rerr_opt->err_dst;
126.    DEBUG("Send RERR err_src %s err_dst %s unr_dst %s\n",
127.          print_ip(err_src),
128.          print_ip(err_dst),
129.          print_ip(*((struct in_addr *)rerr_opt->info)));
130.
131.    XMIT(dp);
132.
133.    return 0;
134.
135. out_err:
136.
137.    dsr_pkt_free(dp);
138.
139.    return -1;
140.
141. }
```

11-12 不存在包或包的源地址为当前地址，返回-1

14-16　没有源路由选项 返回-1

19-22　若数据包的重发次数为 0，目的地址设置为数据包的源地址，否则，将目的地址设为 dp_源路由的第二跳地址

31　　在路由缓存中寻找目的地址，赋值给数据包的源路由

38-40　设置数据包长度，为选项头+源路由选项+路由错误选项头+4（自定义 DSR 首部长度）+ACK 头

43-48　遍历路由错误数据包选项的重发次数，判断是否有发送次数大于最大发送次数

51-57　构造 IPheader

64-79　添加源路由选项

81-91　添加路由错误选项

96-110 添加旧路由错误选项

115-122 添加旧 ACK 选项

124-125 赋值路由错误的源地址、目的地址

132　　发送数据包

处理路由错误 在 *dsr-rerr.c* 中实现

```
01. int NSCLASS dsr_rerr_opt_recv(struct dsr_pkt *dp, struct dsr_rerr_opt *rerr_opt)
02. {
03.     struct in_addr err_src, err_dst, unr_addr;
04. 
05.     if (!rerr_opt)
06.         return -1;
07. 
08.     dp->rerr_opt[dp->num_rerr_opts++] = rerr_opt;
09. 
10.     switch (rerr_opt->err_type) {
11.     case NODE_UNREACHABLE:
12.         err_src.s_addr = rerr_opt->err_src;
13.         err_dst.s_addr = rerr_opt->err_dst;
14. 
15.         memcpy(&unr_addr, rerr_opt->info, sizeof(struct in_addr));
16. 
17.         DEBUG("NODE_UNREACHABLE err_src=%s err_dst=%s unr=%s\n",
18.               print_ip(err_src), print_ip(err_dst), print_ip(unr_addr));
19. 
20.         /* For now we drop all unacked packets... should probably
21.          * salvage */
22.         maint_buf_del_all(err_dst);
23. 
24.         /* Remove broken link from cache */
25.         lc_link_del(err_src, unr_addr);
26. 
27.         /* TODO: Check options following the RERR option */
28. /*      dsr_rtc_del(my_addr(), err_dst); */
29.         break;
30.     case FLOW_STATE_NOT_SUPPORTED:
31.         DEBUG("FLOW_STATE_NOT_SUPPORTED\n");
32.         break;
33.     case OPTION_NOT_SUPPORTED:
34.         DEBUG("OPTION_NOT_SUPPORTED\n");
35.         break;
36.     }
37. 
38.     return 0;
39. }
```

8　将路由错误选项添加到数据包的路由错误选项数组尾

10-36 判断错误类型

若不可达，则报错源地址、目的地址，丢弃无 ack 和发送次数超过最多次数的包，将不可达的源地址和目的地址对应路径删除。

流状态不支持和选项不支持则报错。