Dsr.c

Ack 发送、选项添加、选项接收

Ack-request 创建、添加、发送、接收

Dsr-dev.c

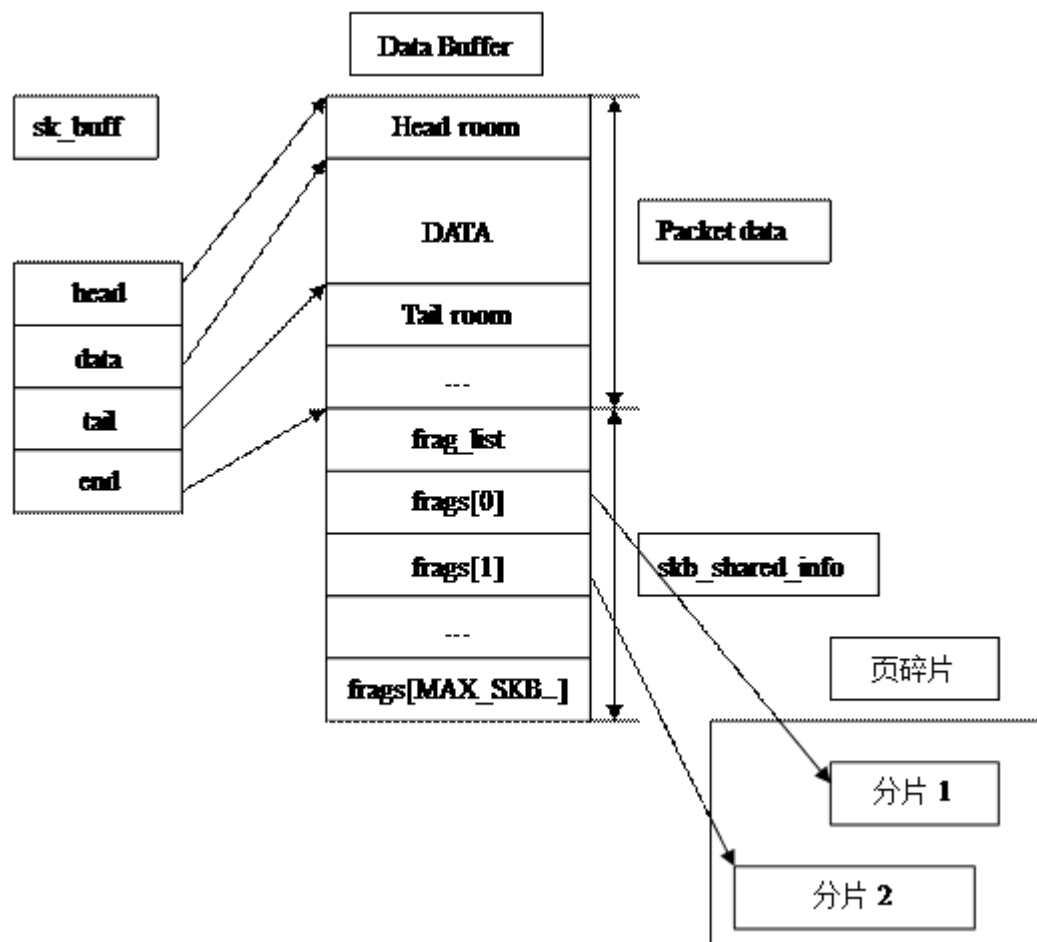sk_buff（socket buffer）结构是 linux 网络代码中重要的数据结构，它管理和控制接收或发送数据包的信息。

sk_buff 组成

Packet data：通过网卡收发的报文，包括链路层、网络层、传输层的协议头和携带的应用数据，包括 head room,data,tail room 三部分。

skb_shared_info 作为 packet data 的补充，用于存储 ip 分片，其中 sk_buff *frag_list 是一系列子 skbuff 链表，而 frag[]是由一组单独的 page 组成的数据缓冲区。

Data buffer：用于存储 packet data 的缓冲区，分为以上两部分。

Sk_buff：缓冲区控制结构 sk_buff。

```c
struct sk_buff *dsr_skb_create(struct dsr_pkt *dp, struct net_device *dev)
{
    struct sk_buff *skb;
    char *buf;
    int ip_len;
    int tot_len;
    int dsr_opts_len = dsr_pkt_opts_len(dp);

    ip_len = dp->nh.iph->ihl << 2;

    tot_len = ip_len + dsr_opts_len + dp->payload_len;

    DEBUG("ip_len=%d dsr_opts_len=%d payload_len=%d tot_len=%d\n",
          ip_len, dsr_opts_len, dp->payload_len, tot_len);
#ifdef KERNEL26
    skb = alloc_skb(tot_len + LL_RESERVED_SPACE(dev), GFP_ATOMIC);
#else
    skb = alloc_skb(dev->hard_header_len + 15 + tot_len, GFP_ATOMIC);
#endif
    if (!skb) {
        DEBUG("alloc_skb failed\n");
        return NULL;
    }

    /* We align to 16 bytes, for ethernet: 2 bytes + 14 bytes header */
#ifdef KERNEL26
    skb_reserve(skb, LL_RESERVED_SPACE(dev));
#else
    skb_reserve(skb, (dev->hard_header_len + 15) & ~15);
#endif
    skb->mac.raw = skb->data - 14;
    skb->nh.raw = skb->data;
    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    /* Copy in all the headers in the right order */
    buf = skb_put(skb, tot_len);

    memcpy(buf, dp->nh.raw, ip_len);

    /* For some reason the checksum has to be recalculated here, at least
     * when there is a record route IP option */
    ip_send_check((struct iphdr *)buf);

    buf += ip_len;

    /* Add DSR header if it exists */
    if (dsr_opts_len) {
        memcpy(buf, dp->dh.raw, dsr_opts_len);
        buf += dsr_opts_len;
    }

    /* Add payload */
    if (dp->payload_len && dp->payload)
        memcpy(buf, dp->payload, dp->payload_len);

    return skb;
} « end dsr_skb_create »
```

添加 packet 内容到 socket_buffer

Dsr_hardware_header_create

*int dsr_hw_header_create(struct dsr_pkt *dp, struct sk_buff *skb)*

{

```c
        struct sockaddr broadcast =
            { AF_UNSPEC, {0xff, 0xff, 0xff, 0xff, 0xff, 0xff} };
        struct neighbor_info neigh_info;

        if (dp->dst.s_addr == DSR_BROADCAST)                    packet 的目的地址为广播
            memcpy(neigh_info.hw_addr.sa_data, broadcast.sa_data, ETH_ALEN);
        else {
            /* Get hardware destination address */
            if (neigh_tbl_query(dp->nxt_hop, &neigh_info) < 0) {
                DEBUG
                    ("Could not get hardware address for next hop %s\n",
                     print_ip(dp->nxt_hop));
                return -1;
            }
        }

        if (skb->dev->hard_header) {
            skb->dev->hard_header(skb, skb->dev, ETH_P_IP,
                        neigh_info.hw_addr.sa_data, 0, skb->len);
        } else {
            DEBUG("Missing hard_header\n");
            return -1;
        }
        return 0;
}
```

网络设备地址事件处理

```c
static int dsr_dev_inetaddr_event(struct notifier_block *this,
                    unsigned long event, void *ptr)
{
    struct in_ifaddr *ifa = (struct in_ifaddr *)ptr;
    struct in_device *indev;

    if (!ifa)
        return NOTIFY_DONE;

    indev = ifa->ifa_dev;

    if (!indev)
        return NOTIFY_DONE;

    switch (event) {
    case NETDEV_UP:
        DEBUG("inetdev UP\n");
```

```
        if (indev->dev == dsr_dev) {
                struct dsr_node *dnode;
                struct in_addr addr, bc;

                dnode = (struct dsr_node *)indev->dev->priv;

                dsr_node_lock(dnode);
                dnode->ifaddr.s_addr = ifa->ifa_address;
                dnode->bcaddr.s_addr = ifa->ifa_broadcast;

                dnode->slave_indev = in_dev_get(dnode->slave_dev);

                        /* Disable rp_filter and enable forwarding */
                        if (dnode->slave_indev) {
                                rp_filter = dnode->slave_indev->cnf.rp_filter;
                                forwarding  =  dnode->slave_indev->cnf.forwarding;
dnode->slave_indev->cnf.rp_filter = 0;
                                dnode->slave_indev->cnf.forwarding = 1;
                        }
                dsr_node_unlock(dnode);

                addr.s_addr = ifa->ifa_address;
                bc.s_addr = ifa->ifa_broadcast;

                DEBUG("New ip=%s broadcast=%s\n",
                        print_ip(addr), print_ip(bc));
        }
        break;
    default:
        break;
    };
    return NOTIFY_DONE;
}
```

Dsr-device-netdevice-event 处理
Dsr-device-netdevice 启动、状态改变、关闭
```
static int dsr_dev_netdev_event(struct notifier_block *this,
                unsigned long event, void *ptr)
{
    struct net_device *dev = (struct net_device *)ptr;
    struct dsr_node *dnode = (struct dsr_node *)dsr_dev->priv;
    int slave_change = 0;
```

```c
    if (!dev)
        return NOTIFY_DONE;

    switch (event) {
    case NETDEV_REGISTER:
        DEBUG("Netdev register %s\n", dev->name);
        if (dnode->slave_dev == NULL &&
            strcmp(dev->name, dnode->slave_ifname) == 0) {    当前无且 name 相同

            DEBUG("Slave dev %s up\n", dev->name);

            dsr_node_lock(dnode);
            dnode->slave_dev = dev;
            dev_hold(dev);
            dsr_node_unlock(dnode);

            /* Reduce the MTU to allow DSR options of 100
             * bytes. If larger, drop or implement
             * fragmentation... ;-) Alternatively find a
             * way to dynamically reduce the data size of
             * packets depending on the size of the DSR
             * header. */
            dsr_dev->mtu = dev->mtu - DSR_OPTS_MAX_SIZE;

            DEBUG("Registering packet type\n");
            dsr_packet_type.func = dsr_dev_llrecv;
            dsr_packet_type.dev = dev;
            dev_add_pack(&dsr_packet_type);

            slave_change = 1;
        }

        if (slave_change)
            DEBUG("New DSR slave interface %s\n", dev->name);
        break;
    case NETDEV_CHANGE:
        DEBUG("Netdev change\n");
        break;
    case NETDEV_UP:
        DEBUG("Netdev up %s\n", dev->name);
        if (ConfVal(PromiscOperation) &&
            dev == dsr_dev && dnode->slave_dev)
            dev_set_promiscuity(dnode->slave_dev, +1);
```

```c
                break;
        case NETDEV_UNREGISTER:
                DEBUG("Netdev unregister %s\n", dev->name);

                dsr_node_lock(dnode);
                if (dev == dnode->slave_dev) {
                        dev_remove_pack(&dsr_packet_type);
                        dsr_packet_type.func = NULL;
                        slave_change = 1;
                        dev_put(dev);
                        dnode->slave_dev = NULL;
                }
                dsr_node_unlock(dnode);

                if (slave_change)
                        DEBUG("DSR slave interface %s unregisterd\n",
                                dev->name);
                break;
        case NETDEV_DOWN:
                DEBUG("Netdev down %s\n", dev->name);
                if (dev == dsr_dev) {
                        if (dnode->slave_dev && ConfVal(PromiscOperation))
                                dev_set_promiscuity(dnode->slave_dev, -1);

                        dsr_node_lock(dnode);
                                if (dnode->slave_indev) {
                                        dnode->slave_indev->cnf.rp_filter = rp_filter;
                                        dnode->slave_indev->cnf.forwarding   =   forwarding;
in_dev_put(dnode->slave_indev);
                                        dnode->slave_indev = NULL;
                                }
                                dsr_node_unlock(dnode);
                } else if (dev == dnode->slave_dev && dnode->slave_indev) {
                        dsr_node_lock(dnode);
                        dnode->slave_indev->cnf.rp_filter = rp_filter;
                        dnode->slave_indev->cnf.forwarding                =             forwarding;
in_dev_put(dnode->slave_indev);
                        dnode->slave_indev = NULL;
                                dsr_node_unlock(dnode);
                }
                break;
        default:
                break;
        };
```

return NOTIFY_DONE;

}

Dsr_dev_start_xmit 发射 dev 开始
Dsr_dev_get_stats 获取状态
Dsr_dev_set_address 设置 dev 的地址
Dsr_dev_accept_fastpath   允许 dev fastpath 可以依据已有状态直接转发的路径   slowpath
需要寻找路由、解析 MAC
以及 open、stop、uninit device

Dsr-io.c
Dsr 接收、开始发送

```c
int NSCLASS dsr_recv(struct dsr_pkt *dp)
{
    int i = 0, action;
    int mask = DSR_PKT_NONE;

    /* Process DSR Options */
    action = dsr_opt_recv(dp);

    /* Add mac address of previous hop to the neighbor table */

    if (dp->flags & PKT_PROMISC_RECV) {
        dsr_pkt_free(dp);
        return 0;
    }
    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
            {
```

Action 为收到的 dsr 报文的选项

添加 MAC 地址到前一跳到相邻节点表中
如果 dp 不为零 收到 则返回
I=0; i<12;switch (action&mask mask=1) 处理报文

```
        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;
            }
            break;
```

主要 case DSR_PKT_FORWARD 2^7
若 ttl<1 则丢弃
否则转发

```
case DSR_PKT_FORWARD_RREQ:
    XMIT(dp);
    return 0;
case DSR_PKT_SEND_RREP:
    /* In dsr-rrep.c */
    break;
case DSR_PKT_SEND_ICMP:
    DEBUG("Send ICMP\n");
    break;
case DSR_PKT_SEND_BUFFERED:
    if (dp->rrep_opt) {
        struct in_addr rrep_srt_dst;
        int i;

        for (i = 0; i < dp->num_rrep_opts; i++) {
            rrep_srt_dst.s_addr = dp->rrep_opt[i]->addrs[DSR_RREP_ADDRS_LEN(dp->rrep_opt[i]) / sizeof(struct in_addr)];

            send_buf_set_verdict(SEND_BUF_SEND, rrep_srt_dst);
        }
    }
    break;
case DSR_PKT_DELIVER:
    DEBUG("Deliver to DSR device\n");
    DELIVER(dp);
    return 0;
case 0:
    break;
default:
    DEBUG("Unknown pkt action\n");
```

DSR-FORWARD-RREQ 转发
DSR-SEND-RREP 路由应答
DSR_PKT_SEND_BUFFERED
发送 buffer 详细在 send-buf.c
DSR-PKT-DELIVER deliver 到 DSR device 函数在 dsr-dev.h

Dsr_start_xmit

```c
void NSCLASS dsr_start_xmit(struct dsr_pkt *dp)
{
    int res;

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

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

    if (dp->srt) {

        if (dsr_srt_add(dp) < 0) {
            DEBUG("Could not add source route\n");
            goto out;
        }
        /* Send packet */

        XMIT(dp);

        return;

    } else {
#ifdef NS2
        res = send_buf_enqueue_packet(dp, &DSRUU::ns_xmit);
#else
        res = send_buf_enqueue_packet(dp, &dsr_dev_xmit);
#endif
```

无 packet DEBUG

dp->srt = dsr_rtc_find(dp->src, dp->dst); 寻找源路由

判断能否添加源路由

发送 dp

发送 buffer 入列的 packet

Res<0 buffer full

Res <0 无 route request table entry   RREQ transmission failed

```c
        if (res < 0) {
            DEBUG("Queueing failed!\n");
            goto out;
        }
        res = dsr_rreq_route_discovery(dp->dst);

        if (res < 0)
            DEBUG("RREQ Transmission failed...");

        return;
    }
    out:
    dsr_pkt_free(dp);
} « end dsr_start_xmit »
```

```c
struct dsr_opt_hdr *dsr_opt_hdr_add(char *buf, unsigned int len,
                    unsigned int protocol)
{
    struct dsr_opt_hdr *opt_hdr;

    if (len < DSR_OPT_HDR_LEN)
        return NULL;

    opt_hdr = (struct dsr_opt_hdr *)buf;

    opt_hdr->nh = protocol;
    opt_hdr->f = 0;
    opt_hdr->res = 0;
    opt_hdr->p_len = htons(len - DSR_OPT_HDR_LEN);

    return opt_hdr;
}
```

添加 dsr-header option

```c
struct iphdr *dsr_build_ip(struct dsr_pkt *dp, struct in_addr src,
            struct in_addr dst, int ip_len, int tot_len,
            int protocol, int ttl)
{
    struct iphdr *iph;

    dp->nh.iph = iph = (struct iphdr *)dp->ip_data;

    if (dp->skb && dp->skb->nh.raw) {
        memcpy(dp->ip_data, dp->skb->nh.raw, ip_len);
    } else {
        iph->version = IPVERSION;
        iph->ihl = 5;
        iph->tos = 0;
        iph->id = 0;
        iph->frag_off = 0;
        iph->ttl = (ttl ? ttl : IPDEFTTL);
        iph->saddr = src.s_addr;
        iph->daddr = dst.s_addr;
    }

    iph->tot_len = htons(tot_len);
    iph->protocol = protocol;

    ip_send_check(iph);

    return iph;
} « end dsr_build_ip »
```

构造 IP 报文

```c
struct dsr_opt *dsr_opt_find_opt(struct dsr_pkt *dp, int type)
{
    int dsr_len, l;
    struct dsr_opt *dopt;

    dsr_len = dsr_pkt_opts_len(dp);

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

    while (l < dsr_len && (dsr_len - l) > 2) {
        if (type == dopt->type)
            return dopt;

        l += dopt->length + 2;
        dopt = DSR_GET_NEXT_OPT(dopt);
    }
    return NULL;
}
```

发现选项 -2 大概因为选项一个字段为 kind 一个字段为 length

```c
int NSCLASS dsr_opt_remove(struct dsr_pkt *dp)
{
    int len, ip_len, prot, ttl;

    if (!dp || !dp->dh.raw)
        return -1;

    prot = dp->dh.opth->nh;
#ifdef NS2
    ip_len = 20;
    ttl = dp->nh.iph->ttl();
#else
    ip_len = (dp->nh.iph->ihl << 2);
    ttl = dp->nh.iph->ttl;
#endif
    dsr_build_ip(dp, dp->src, dp->dst, ip_len,
                 ip_len + dp->payload_len, prot, ttl);

    len = dsr_pkt_free_opts(dp);

    /* Return bytes removed */
    return len;
} « end dsr_opt_remove »
```

移除选项 --通过截断 IP 前 20 字节
Dsr-opt-parse 解析 dsr 的 opt 服务器接收 SYN 包时

```
        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
```

不确定 in one packet reached 是指在这之前已经到达还是这个 packet 包含

Dsr-opt-recv

```
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);
```

Myaddr 由 my_addr()获取 dsr_node 的 ifaddr 即 ipaddr

```c
while (l < dsr_len && (dsr_len - 1) > 2) {
    //DEBUG("dsr_len=%d l=%d\n", dsr_len, l);
    switch (dopt->type) {
    case DSR_OPT_PADN:
        break;
    case DSR_OPT_RREQ:
        if (dp->flags & PKT_PROMISC_RECV)
            break;

        action |= dsr_rreq_opt_recv(dp, (struct dsr_rreq_opt *)dopt);
        break;
    case DSR_OPT_RREP:
        if (dp->flags & PKT_PROMISC_RECV)
            break;

        action |= dsr_rrep_opt_recv(dp, (struct dsr_rrep_opt *)dopt);
        break;
    case DSR_OPT_RERR:
        if (dp->flags & PKT_PROMISC_RECV)
            break;
        if (dp->num_rerr_opts < MAX_RERR_OPTS) {
            action |=
                dsr_rerr_opt_recv(dp, (struct dsr_rerr_opt *)dopt);
        }

        break;
    case DSR_OPT_PREV_HOP:
        break;
    case DSR_OPT_ACK:
        if (dp->flags & PKT_PROMISC_RECV)
            break;
```

判断 action

```
            if (dp->flags & PKT_PROMISC_RECV)
                break;

            if (dp->num_ack_opts < MAX_ACK_OPTS) {
                dp->ack_opt[dp->num_ack_opts++] =
                    (struct dsr_ack_opt *)dopt;
                action |=
                    dsr_ack_opt_recv((struct dsr_ack_opt *)
                            dopt);
            }
            break;
    case DSR_OPT_SRT:
            action |= dsr_srt_opt_recv(dp, (struct dsr_srt_opt *)dopt);
            break;
    case DSR_OPT_TIMEOUT:
            break;
    case DSR_OPT_FLOWID:
            break;
    case DSR_OPT_ACK_REQ:
            action |=
                dsr_ack_req_opt_recv(dp, (struct dsr_ack_req_opt *)
                        dopt);
            break;
    case DSR_OPT_PAD1:
            l++;
            dopt++;
            continue;
    default:
            DEBUG("Unknown DSR option type=%d\n", dopt->type);
    } « end switch dopt->type »
    l += dopt->length + 2;
    dopt = DSR_GET_NEXT_OPT(dopt);
    } « end while l<dsr_len&&(dsr_len-l... »
    return action;
} « end dsr_opt_recv »
```

中间的 while 循环应该是用来读取包内的下一个 opt 结束时减去一个 opt 的大小，进入下一个解析

Dsr-pkt.c
Dsr_pkt_alloc_opts 和 *dsr_pkt_alloc_opts_expand*
添加选项和扩展选项（增加选项）
Dsr_pkt_free_opts
释放/删除选项
Dsr_pkt_alloc （重载）
创建 pkt 填充字段
Dsr_pkt_free
释放 pkt

Dsr-rerr.c
Dsr_rerr_opt_add
填充字段
判断错误类型 不可达、流状态不支持？（猜测不能在链路中传递）选项不支持
Dsr_rerr_send
*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)          不存在包或包的源地址为
当前地址/为当前地址发出的包   (not sure)
        return -1;

  if (!dp_trigg->srt_opt) {                               没有源路由选项 因为源路由选项不为空？
        DEBUG("Could not find source route option\n");
        return -1;
    }

    if (dp_trigg->srt_opt->salv == 0)               重新发送次数为 0
        dst = dp_trigg->src;                         目的地址为 dp 的源地址
    else
        dst.s_addr = dp_trigg->srt_opt->addrs[1];       发送次数不为 1 dst 的地址为 dp_源
路由的第二跳地址   (不确定 addr 数组的规则)

    dp = dsr_pkt_alloc(NULL);                            新分配 pkt

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

    dp->srt = dsr_rtc_find(myaddr, dst);              pkt 的源路由赋值为 routecache？find

    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) +          选项头+源路由选项
        (DSR_RERR_HDR_LEN + 4) +                          +RERR 头+4(自定义 DSR 首部长度)
        DSR_ACK_HDR_LEN * dp_trigg->num_ack_opts;      +ACK 头

    /* Also count in RERR opts in trigger packet */          触发包？
    for (i = 0; i < dp_trigg->num_rerr_opts; i++) {
```

```
        if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAGE_COUNT))    发送次数>最大次数
            break;

        len += (dp_trigg->rerr_opt[i]->length + 2);          下一个 option
}

DEBUG("opt_len=%d SR: %s\n", len, print_srt(dp->srt));
n = dp->srt->laddrs / sizeof(struct in_addr);                 有多少个节点
dp->src = myaddr;
dp->dst = dst;
dp->nxt_hop = dsr_srt_next_hop(dp->srt, n);                下一跳赋值

dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,          构造 ipheader
                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);              分配 pkt option

if (!buf)
    goto out_err;

dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);  构造/添加 optionheader

if (!dp->dh.opth) {
    DEBUG("Could not create DSR options header\n");
    goto out_err;
}

buf += DSR_OPT_HDR_LEN;                     opt-header 放到 buf?
len -= DSR_OPT_HDR_LEN;

dp->srt_opt = dsr_srt_opt_add(buf, len, 0, 0, dp->srt);  添加源路由选项

if (!dp->srt_opt) {
    DEBUG("Could not create Source Route option header\n");
    goto out_err;
}

buf += DSR_SRT_OPT_LEN(dp->srt);
len -= DSR_SRT_OPT_LEN(dp->srt);
```

```
rerr_opt = dsr_rerr_opt_add(buf, len, NODE_UNREACHABLE, dp->src,        rerr 添加 buf
                  dp->dst, unr_addr,
                  dp_trigg->srt_opt->salv);

if (!rerr_opt)
    goto out_err;

buf += (rerr_opt->length + 2);                          下个选项  大概
len -= (rerr_opt->length + 2);

/* Add old RERR options */                         添加旧选项
for (i = 0; i < dp_trigg->num_rerr_opts; i++) {

    if (dp_trigg->rerr_opt[i]->salv > ConfVal(MAX_SALVAGE_COUNT))    重发次数大于
最大次数
            break;

    memcpy(buf, dp_trigg->rerr_opt[i],
            dp_trigg->rerr_opt[i]->length + 2);                 copy    opt

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

/* TODO: Must preserve order of RERR and ACK options from triggering
  * packet */

/* Add old ACK options */                           添加旧 ACK 选项
for (i = 0; i < dp_trigg->num_ack_opts; i++) {
    memcpy(buf, dp_trigg->ack_opt[i],
            dp_trigg->ack_opt[i]->length + 2);

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

err_src.s_addr = rerr_opt->err_src;              err 的源地址
err_dst.s_addr = rerr_opt->err_dst;              err 的目的地址

DEBUG("Send RERR err_src %s err_dst %s unr_dst %s\n",
        print_ip(err_src),
        print_ip(err_dst),
        print_ip(*((struct in_addr *)rerr_opt->info)));
```

XMIT(dp);   *发送 dp*

return 0;

 out_err:

dsr_pkt_free(dp);   *释放 dp*

return -1;

}

Dsr_rerr_opt_recv

```c
int NSCLASS dsr_rerr_opt_recv(struct dsr_pkt *dp, struct dsr_rerr_opt *rerr_opt)
{
    struct in_addr err_src, err_dst, unr_addr;

    if (!rerr_opt)
        return -1;

    dp->rerr_opt[dp->num_rerr_opts++] = rerr_opt;

    switch (rerr_opt->err_type) {
    case NODE_UNREACHABLE:
        err_src.s_addr = rerr_opt->err_src;
        err_dst.s_addr = rerr_opt->err_dst;

        memcpy(&unr_addr, rerr_opt->info, sizeof(struct in_addr));

        DEBUG("NODE_UNREACHABLE err_src=%s err_dst=%s unr=%s\n",
              print_ip(err_src), print_ip(err_dst), print_ip(unr_addr));

        /* For now we drop all unacked packets... should probably
         * salvage */
        maint_buf_del_all(err_dst);

        /* Remove broken link from cache */
        lc_link_del(err_src, unr_addr);

        /* TODO: Check options following the RERR option */
/*      dsr_rtc_del(my_addr(), err_dst); */
        break;
    case FLOW_STATE_NOT_SUPPORTED:
        DEBUG("FLOW_STATE_NOT_SUPPORTED\n");
        break;
    case OPTION_NOT_SUPPORTED:
        DEBUG("OPTION_NOT_SUPPORTED\n");
        break;
    } « end switch rerr_opt->err_type »

    return 0;
} « end dsr_rerr_opt_recv »
```

判断存在
向 rerr_opt 数组添加
判断错误类型
case 不可达 DEBUG 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

Dsr-rrep.c

两个 struct grat_rrep_entry、grat_rrep_query （grant route response xx？）

Crit_query 判断上述两个结构的 src.addr 以及上一跳是否一致 (Criteria Query)条件查询

Crit_time 判断时间

grat_rrep_tbl_timeout 超时

grat_rrep_tbl_add 判断是否在 table 中(读锁) 设定计时器 添加到表头

tbl find、print

grat_rrep_tbl_proc_info 返回数据包长度

```c
static inline int
dsr_rrep_add_srt(struct dsr_rrep_opt *rrep_opt, struct dsr_srt *srt)
{
    int n;

    if (!rrep_opt | !srt)
        return -1;

    n = srt->laddrs / sizeof(struct in_addr);

    memcpy(rrep_opt->addrs, srt->addrs, srt->laddrs);
    rrep_opt->addrs[n] = srt->dst.s_addr;

    return 0;
}

static struct dsr_rrep_opt *dsr_rrep_opt_add(char *buf, int len,
                         struct dsr_srt *srt)
{
    struct dsr_rrep_opt *rrep_opt;

    if (!buf || !srt || (unsigned int)len < DSR_RREP_OPT_LEN(srt))
        return NULL;

    rrep_opt = (struct dsr_rrep_opt *)buf;

    rrep_opt->type = DSR_OPT_RREP;
    rrep_opt->length = srt->laddrs + sizeof(struct in_addr) + 1;
    rrep_opt->l = 0;
    rrep_opt->res = 0;

    /* Add source route to RREP */
    dsr_rrep_add_srt(rrep_opt, srt);

    return rrep_opt;
} « end dsr_rrep_opt_add »
```

Dsr_rrep_add_srt 添加源路由到 RREP

dsr_rrep_send 发送 rrep

*int NSCLASS dsr_rrep_send(struct dsr_srt *srt, struct dsr_srt *srt_to_me)*

{

*struct dsr_pkt *dp = NULL;*

*char *buf;*

int len, ttl, n;

if (!srt || !srt_to_me)

return -1;

```
dp = dsr_pkt_alloc(NULL);

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

dp->src = my_addr();
dp->dst = srt->dst;

if (srt->laddrs == 0)                                    源路由为空
    dp->nxt_hop = dp->dst;                    packet 的下一跳为目的地址
else
    dp->nxt_hop = srt->addrs[0];              packet 的下一跳为源路由第一跳

len = DSR_OPT_HDR_LEN + DSR_SRT_OPT_LEN(srt) +
    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*/
n = srt->laddrs / sizeof(struct in_addr);     跳数

DEBUG("srt: %s\n", print_srt(srt));
DEBUG("srt_to_me: %s\n", print_srt(srt_to_me));
DEBUG("next_hop=%s\n", print_ip(dp->nxt_hop));
DEBUG
    ("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",
     IP_HDR_LEN, DSR_OPT_HDR_LEN, DSR_SRT_OPT_LEN(srt),
     DSR_RREP_OPT_LEN(srt_to_me), DSR_OPT_PAD1_LEN, len);

ttl = n + 1;                                  跳数+1   ttl 为 1 丢弃

DEBUG("TTL=%d, n=%d\n", ttl, n);

buf = dsr_pkt_alloc_opts(dp, len);            分配 buf

if (!buf)
    goto out_err;

dp->nh.iph = dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
            IP_HDR_LEN + len, IPPROTO_DSR, ttl);         构造 IP header

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

```
        goto out_err;
    }

    dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);   构造选项首部

    if (!dp->dh.opth) {
        DEBUG("Could not create DSR options header\n");
        goto out_err;
    }

    buf += DSR_OPT_HDR_LEN;                    加选项首部到 buf
    len -= DSR_OPT_HDR_LEN;

    /* Add the source route option to the packet */
    dp->srt_opt = dsr_srt_opt_add(buf, len, 0, dp->salvage, srt);        源路由选项加到 packet

    if (!dp->srt_opt) {
        DEBUG("Could not create Source Route option header\n");
        goto out_err;
    }

    buf += DSR_SRT_OPT_LEN(srt);
    len -= DSR_SRT_OPT_LEN(srt);

    dp->rrep_opt[dp->num_rrep_opts++] =
        dsr_rrep_opt_add(buf, len, srt_to_me);              把源路由选项添加到 rrep 选项数组

    if (!dp->rrep_opt[dp->num_rrep_opts - 1]) {
        DEBUG("Could not create RREP option header\n");
        goto out_err;
    }

    /* TODO: Should we PAD? The rrep struct is padded and aligned
     * automatically by the compiler... How to fix this? */

/*    buf += DSR_RREP_OPT_LEN(srt_to_me); */
/*    len -= DSR_RREP_OPT_LEN(srt_to_me); */

/*    pad1_opt = (struct dsr_pad1_opt *)buf; */
/*    pad1_opt->type = DSR_OPT_PAD1; */

    /* if (ConfVal(UseNetworkLayerAck)) */
/*        dp->flags |= PKT_REQUEST_ACK; */
```

```
    dp->flags |= PKT_XMIT_JITTER;                    与 0x08 即 1000 做或运算

    XMIT(dp);                                        发送 packet

    return 0;
        out_err:
    if (dp)
        dsr_pkt_free(dp);

    return -1;
}
```

Dsr_路由回答选项 接收

```
int NSCLASS dsr_rrep_opt_recv(struct dsr_pkt *dp, struct dsr_rrep_opt *rrep_opt)
{
    struct in_addr myaddr, srt_dst;
    struct dsr_srt *rrep_opt_srt;

    if (!dp || !rrep_opt || dp->flags & PKT_PROMISC_RECV)
        return DSR_PKT_ERROR;

    if (dp->num_rrep_opts < MAX_RREP_OPTS)
        dp->rrep_opt[dp->num_rrep_opts++] = rrep_opt;
    else
        return DSR_PKT_ERROR;
                                                        错误判断
    myaddr = my_addr();

    srt_dst.s_addr  =  rrep_opt->addrs[DSR_RREP_ADDRS_LEN(rrep_opt)  /  sizeof(struct
in_addr)];                         源路由目的地址赋为路由回答报文选项最后一个的地址

    rrep_opt_srt = dsr_srt_new(dp->dst, srt_dst,
                    DSR_RREP_ADDRS_LEN(rrep_opt),
                    (char *)rrep_opt->addrs);          构造 rrep 选项

    if (!rrep_opt_srt)
        return DSR_PKT_ERROR;

    dsr_rtc_add(rrep_opt_srt, ConfValToUsecs(RouteCacheTimeout), 0);    源路由添加构造
的 rrep 选项源路由

    /* Remove pending RREQs */
    rreq_tbl_route_discovery_cancel(rrep_opt_srt->dst);             移除等待的 RREQs
```

```
    FREE(rrep_opt_srt);

    if (dp->dst.s_addr == myaddr.s_addr) {          目的地址为为当前节点的地址
        /*RREP for this node */

        DEBUG("RREP for me!\n");

        return DSR_PKT_SEND_BUFFERED;
    }

    DEBUG("I am not RREP destination\n");

    /* Forward */
    return DSR_PKT_FORWARD;
}
```
返回值为一系列状态

grat_rrep_tbl_init、grat_rrep_tbl_cleanup   rrep 表初始化和清空

dsr-rreq.c

struct   rreq_tbl_entry、id_entry、rreq_tbl_query 用于 rreq 验证

```
struct rreq_tbl_entry {
    list_t l;
    int state;
    struct in_addr node_addr;
    int ttl;
    DSRUUTimer *timer;
    struct timeval tx_time;
    struct timeval last_used;
    usecs_t timeout;
    unsigned int num_rexmts;
    struct tbl rreq_id_tbl;
};

struct id_entry {
    list_t l;
    struct in_addr trg_addr;
    unsigned short id;
};
struct rreq_tbl_query {
    struct in_addr *initiator;
    struct in_addr *target;
    unsigned int *id;
};
```

```c
static inline int crit_addr(void *pos, void *data)
{
    struct rreq_tbl_entry *e = (struct rreq_tbl_entry *)pos;
    struct in_addr *a = (struct in_addr *)data;

    if (e->node_addr.s_addr == a->s_addr)
        return 1;
    return 0;
}

static inline int crit_duplicate(void *pos, void *data)
{
    struct rreq_tbl_entry *e = (struct rreq_tbl_entry *)pos;
    struct rreq_tbl_query *q = (struct rreq_tbl_query *)data;

    if (e->node_addr.s_addr == q->initiator->s_addr) {
        list_t *p;

        list_for_each(p, &e->rreq_id_tbl.head) {
            struct id_entry *id_e = (struct id_entry *)p;

            if (id_e->trg_addr.s_addr == q->target->s_addr &&
                id_e->id == *(q->id))
                return 1;
        }
    }
    return 0;
}
```

crit_addr 判断 rreq address 是否一致
crit_duplicate 赋 data 值给 id_e

rreq_tbl_set_max_len 设置 rreq 表最大长度
rreq_tbl_print 打印 rreq 表
rreq_tbl_timeout   rreq 超时处理

```c
void NSCLASS rreq_tbl_timeout(unsigned long data)
{
    struct rreq_tbl_entry *e = (struct rreq_tbl_entry *)data;
    struct timeval expires;

    if (!e)
        return;

    tbl_detach(&rreq_tbl, &e->l);

    DEBUG("RREQ Timeout dst=%s timeout=%lu rexmts=%d \n",
          print_ip(e->node_addr), e->timeout, e->num_rexmts);

    if (e->num_rexmts >= ConfVal(MaxRequestRexmt)) {
        DEBUG("MAX RREQs reached for %s\n", print_ip(e->node_addr));

        e->state = STATE_IDLE;

/*      DSR_WRITE_UNLOCK(&rreq_tbl); */
        tbl_add_tail(&rreq_tbl, &e->l);
        return;
    }

    e->num_rexmts++;

    /* if (e->ttl == 1) */
/*      e->timeout = ConfValToUsecs(RequestPeriod);   */
/*  else */
    e->timeout *= 2;    /* Double timeout */

    e->ttl *= 2;        /* Double TTL */

    if (e->ttl > MAXTTL)
        e->ttl = MAXTTL;

    if (e->timeout > ConfValToUsecs(MaxRequestPeriod))
        e->timeout = ConfValToUsecs(MaxRequestPeriod);

    gettime(&e->last_used);

    dsr_rreq_send(e->node_addr, e->ttl);
```

报错目的地址 超时时间 计时器重传次数（没有查到）
重传>=最大请求重传时间？次数
将状态设为闲置
将其加到表尾
加倍超时、加倍 ttl 若超过最大设置为最大
再次发送 rreq
到期时间设置为 e->上一个时间
将这个数据包放在表尾
设置 table 的定时器

```c
    expires = e->last_used;
    timeval_add_usecs(&expires, e->timeout);

    /* Put at end of list */
    tbl_add_tail(&rreq_tbl, &e->l);

    set_timer(e->timer, &expires);
```

__rreq_tbl_entry_creat、__rreq_tbl_add、rreq_tbl_add_id   rreq 表 建表、添加 rreq、添加 id

rreq_tbl_route_discovery_cancel rreq 路由发现取消

```
int NSCLASS rreq_tbl_route_discovery_cancel(struct in_addr dst)
{
    struct rreq_tbl_entry *e;

    e = (struct rreq_tbl_entry *)tbl_find_detach(&rreq_tbl, &dst,
                               crit_addr);

    if (!e) {
        DEBUG("%s not in RREQ table\n", print_ip(dst));
        return -1;
    }

    if (e->state == STATE_IN_ROUTE_DISC)
        del_timer_sync(e->timer);

    e->state = STATE_IDLE;
    gettime(&e->last_used);

    tbl_add_tail(&rreq_tbl, &e->l);

    return 1;
} « end rreq_tbl_route_discovery_cancel »
```

不在 RREQ 表报错
状态为在路由（表）disc 中   (not idle)
删除计时器 并设置状态为闲置
添加到尾部

dsr_rreq_route_discovery 路由发现

int NSCLASS dsr_rreq_route_discovery(struct in_addr target)
{
*   struct rreq_tbl_entry *e;*
*   int ttl, res = 0;*
*   struct timeval expires;*

#define   TTL_START 1

*   DSR_WRITE_LOCK(&rreq_tbl.lock);   写锁*

*   e = (struct rreq_tbl_entry *)__tbl_find(&rreq_tbl, &target, crit_addr);   在路由表中寻找 e*

*   if (!e)*
*   e = __rreq_tbl_add(target);   e 不在表中，添加到 rreq 表*
*   else {*
*   /* Put it last in the table */   e 在表中*
*   __tbl_detach(&rreq_tbl, &e->l);   e 连接断开*
*   __tbl_add_tail(&rreq_tbl, &e->l);   把 e 放在最后*
```

```c
    }

    if (!e) {
        res = -ENOMEM;
        goto out;
    }

    if (e->state == STATE_IN_ROUTE_DISC) {                          e 的状态是在路由 DISC
        DEBUG("Route discovery for %s already in progress\n",
                print_ip(target));
        goto out;
    }
    DEBUG("Route discovery for %s\n", print_ip(target));

    gettime(&e->last_used);
    e->ttl = ttl = TTL_START;                                       设置 ttl 为 1
    /* The draft does not actually specify how these Request Timeout values
      * should be used... ??? I am just guessing here. */

    if (e->ttl == 1)                                                设置超时
        e->timeout = ConfValToUsecs(NonpropRequestTimeout);
    else
        e->timeout = ConfValToUsecs(RequestPeriod);

    e->state = STATE_IN_ROUTE_DISC;
    e->num_rexmts = 0;                                              计时器发送次数

    expires = e->last_used;
    timeval_add_usecs(&expires, e->timeout);

    set_timer(e->timer, &expires);

    DSR_WRITE_UNLOCK(&rreq_tbl.lock);                               解锁

    dsr_rreq_send(target, ttl);                                     发送

    return 1;
      out:
    DSR_WRITE_UNLOCK(&rreq_tbl.lock);

    return res;
}

dsr_rreq_duplicate        复制 rreq
```

dsr_rreq_opt_add   *添加 rreq 选项*

rreq 发送

```
int NSCLASS dsr_rreq_send(struct in_addr target, int ttl)
{
    struct dsr_pkt *dp;
    char *buf;
    int len = DSR_OPT_HDR_LEN + DSR_RREQ_HDR_LEN;                分配长度

    dp = dsr_pkt_alloc(NULL);                                    分配 packet

    if (!dp) {
        DEBUG("Could not allocate DSR packet\n");
        return -1;
    }
    dp->dst.s_addr = DSR_BROADCAST;
    dp->nxt_hop.s_addr = DSR_BROADCAST;
    dp->src = my_addr();                                         赋源地址、设置广播

    buf = dsr_pkt_alloc_opts(dp, len);


    if (!buf)
        goto out_err;

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

    if (!dp->nh.iph)
        goto out_err;

    dp->dh.opth = dsr_opt_hdr_add(buf, len, DSR_NO_NEXT_HDR_TYPE);    构造选项头

    if (!dp->dh.opth) {
        DEBUG("Could not create DSR opt header\n");
        goto out_err;
    }

    buf += DSR_OPT_HDR_LEN;
    len -= DSR_OPT_HDR_LEN;                                      去选项头

    dp->rreq_opt = dsr_rreq_opt_add(buf, len, target, ++rreq_seqno);    添加选项头
```

```
    if (!dp->rreq_opt) {
        DEBUG("Could not create RREQ opt\n");
        goto out_err;
    }
#ifdef NS2
    DEBUG("Sending RREQ src=%s dst=%s target=%s ttl=%d iph->saddr()=%d\n",
            print_ip(dp->src), print_ip(dp->dst), print_ip(target), ttl,
            dp->nh.iph->saddr());
#endif

    dp->flags |= PKT_XMIT_JITTER;                    与 0X08 进行或运算

    XMIT(dp);                                        发送 packet

    return 0;

     out_err:
    dsr_pkt_free(dp);

    return -1;
}
```

接收 RREQ

Dsr_rreq_opt_recv

```
    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

dp->rreq_opt = rreq_opt;

    myaddr = my_addr();

    trg.s_addr = rreq_opt->target;   赋值

```
if (dsr_rreq_duplicate(dp->src, trg, ntohs(rreq_opt->id))) {          判断该请求是否
已在缓存中
      DEBUG("Duplicate RREQ from %s\n", print_ip(dp->src));
      return DSR_PKT_DROP;
}                                                                              存在就丢弃

rreq_tbl_add_id(dp->src, trg, ntohs(rreq_opt->id));        不存在，添加到 rreq 表

dp->srt = dsr_srt_new(dp->src, myaddr, DSR_RREQ_ADDRS_LEN(rreq_opt),
                (char *)rreq_opt->addrs);                  新建源路由

if (!dp->srt) {
      DEBUG("Could not extract source route\n");          无法导入
      return DSR_PKT_ERROR;
}
DEBUG("RREQ target=%s src=%s dst=%s laddrs=%d\n",
        print_ip(trg), print_ip(dp->src),
        print_ip(dp->dst), DSR_RREQ_ADDRS_LEN(rreq_opt));

/* Add reversed source route */
srt_rev = dsr_srt_new_rev(dp->srt);                        源路由反转 以便发送路由回复

if (!srt_rev) {
      DEBUG("Could not reverse source route\n");
      return DSR_PKT_ERROR;
}
DEBUG("srt: %s\n", print_srt(dp->srt));
DEBUG("srt_rev: %s\n", print_srt(srt_rev));

dsr_rtc_add(srt_rev, ConfValToUsecs(RouteCacheTimeout), 0);      添加缓存表

/* Set previous hop */                                     设置反向路由的前一跳
if (srt_rev->laddrs > 0)
      dp->prv_hop = srt_rev->addrs[0];
else
      dp->prv_hop = srt_rev->dst;

neigh_tbl_add(dp->prv_hop, dp->mac.ethh);                  邻居节点表添加信息

/* Send buffered packets */
send_buf_set_verdict(SEND_BUF_SEND, srt_rev->dst);         发送缓存区的包

if (rreq_opt->target == myaddr.s_addr) {       路由请求选项的目的地址为当前地址
```

```c
        DEBUG("RREQ OPT for me - Send RREP\n");

        /* According to the draft, the dest addr in the IP header must
         * be updated with the target address */                    更新 ip 头的目的地址
#ifdef NS2
        dp->nh.iph->daddr() = (nsaddr_t) rreq_opt->target;
#else
        dp->nh.iph->daddr = rreq_opt->target;
#endif
        dsr_rrep_send(srt_rev, dp->srt);                            发送路由回复

        action = DSR_PKT_NONE;
        goto out;
    }                                                               endOfIf

    n = DSR_RREQ_ADDRS_LEN(rreq_opt) / sizeof(struct in_addr);

    if (dp->srt->src.s_addr == myaddr.s_addr)              若源地址为当前地址  丢弃
        return DSR_PKT_DROP;

    for (i = 0; i < n; i++)                                源路由中包含当前地址  丢弃
        if (dp->srt->addrs[i].s_addr == myaddr.s_addr) {
            action = DSR_PKT_DROP;
            goto out;
        }

    /* TODO: Check Blacklist */                            检测列表是否存在
    srt_rc = lc_srt_find(myaddr, trg);

    if (srt_rc) {                    若存在  连接 packet 的源路由和路由缓存后 free 路由缓存
        struct dsr_srt *srt_cat;
        /* Send cached route reply */

        DEBUG("Send cached RREP\n");

        srt_cat = dsr_srt_concatenate(dp->srt, srt_rc);

        FREE(srt_rc);

        if (!srt_cat) {
            DEBUG("Could not concatenate\n");
            goto rreq_forward;
        }
```

```
            DEBUG("srt_cat: %s\n", print_srt(srt_cat));

            if (dsr_srt_check_duplicate(srt_cat) > 0) {    存在重复的源路由  free 拼接的源路由
                DEBUG("Duplicate address in source route!!!\n");
                FREE(srt_cat);
                goto rreq_forward;
            }
#ifdef NS2
            dp->nh.iph->daddr() = (nsaddr_t) rreq_opt->target;
#else
            dp->nh.iph->daddr = rreq_opt->target;
#endif
            DEBUG("Sending cached RREP to %s\n", print_ip(dp->src));
            dsr_rrep_send(srt_rev, srt_cat);                    发送缓存的路由回复

            action = DSR_PKT_NONE;

            FREE(srt_cat);
        } else {

        rreq_forward:
            dsr_pkt_alloc_opts_expand(dp, sizeof(struct in_addr));

            if (!DSR_LAST_OPT(dp, rreq_opt)) {
                char *to, *from;
                to = (char *)rreq_opt + rreq_opt->length + 2 +
                    sizeof(struct in_addr);
                from = (char *)rreq_opt + rreq_opt->length + 2;

                memmove(to, from, sizeof(struct in_addr));
            }
            rreq_opt->addrs[n] = myaddr.s_addr;
            rreq_opt->length += sizeof(struct in_addr);

            dp->dh.opth->p_len = htons(ntohs(dp->dh.opth->p_len) +
                            sizeof(struct in_addr));
#ifdef __KERNEL__
            dsr_build_ip(dp, dp->src, dp->dst, IP_HDR_LEN,
                    ntohs(dp->nh.iph->tot_len) +
                    sizeof(struct in_addr), IPPROTO_DSR,
                    dp->nh.iph->ttl);
#endif
            /* Forward RREQ */
            action = DSR_PKT_FORWARD_RREQ;
```

```
        }
    out:
    FREE(srt_rev);
    return action;
```
rreq_tbl_proc_info 返回 RREQ table 的长度
rreq_tbl_init   RREQ 表初始化
rreq_tbl_cleanup   RREQ 表清空

dsr-rtc-simple.c

```
struct rtc_entry {
    list_t l;
    unsigned long expires;
    unsigned short flags;
    struct dsr_srt srt;
};
```
用于维护路由缓存
__dsr_rtc_set_next_timeout 设置下一个超时
Dsr_rtc_timeout   判断超时

```c
static void dsr_rtc_timeout(unsigned long data)
{
    list_t *pos, *tmp;
    int time = TimeNow;

    DSR_WRITE_LOCK(&rtc_lock);

    DEBUG("srt timeout\n");

    list_for_each_safe(pos, tmp, &rtc_head) {
        struct rtc_entry *e = (struct rtc_entry *)pos;

        if (e->expires > time)
            break;

        list_del(&e->l);
        FREE(e);
        rtc_len--;
    }
    __dsr_rtc_set_next_timeout();
    DSR_WRITE_UNLOCK(&rtc_lock);
} « end dsr_rtc_timeout »
#endif                   /* RTC_TIMER */
```

_dsr_rtc_flush 刷新/清空路由缓存
__dsr_rtc_add 添加到路由缓存

```c
static inline int __dsr_rtc_add(struct rtc_entry *e)
{

    if (rtc_Len >= RTC_MAX_LEN) {
        printk(KERN_WARNING "dsr_rtc: Max list len reached\n");
        return -ENOSPC;
    }

    if (list_empty(&rtc_head)) {
        list_add(&e->l, &rtc_head);
    } else {
        list_t *pos;

        list_for_each(pos, &rtc_head) {
            struct rtc_entry *curr = (struct rtc_entry *)pos;

            if (curr->expires > e->expires)
                break;
        }
        list_add(&e->l, pos->prev);
    }
    return 1;
} « end __dsr_rtc_add »
```

若长度达到缓存表最大长度 return
表为空添加到表头，不为空添加到表尾
__dsr_rtc_del 删除路由表缓存的表项
dsr_rtc_update 更新路由缓存表项
dsr_rtc_print 列表 dsr_rtc_proc_info 获取表长度

dsr-srt.c
dsr_srt_next_hop、dsr_srt_prev_hop 获取上一跳、下一跳 为边界的话为目的地址/源地址
dsr_srt_find_addr 查找当前项
dsr_srt_new 构造源路由
dsr_srt_new_rev 构造反向路由

```
struct dsr_srt *dsr_srt_new_split(struct dsr_srt *srt, struct in_addr addr)
{
    struct dsr_srt *srt_split;
    int i, n;

    if (!srt)
        return NULL;

    n = srt->laddrs / sizeof(struct in_addr);

    if (n == 0)
        return NULL;

    for (i = 0; i < n; i++) {
        if (addr.s_addr == srt->addrs[i].s_addr)
            goto ↓split;
    }
    /* Nothing to split */
    return NULL;
      split:
    srt_split = (struct dsr_srt *)MALLOC(sizeof(struct dsr_srt) +
                        (i * sizeof(struct in_addr)),
                        GFP_ATOMIC);

    if (!srt_split)
        return NULL;

    srt_split->src.s_addr = srt->src.s_addr;
    srt_split->dst.s_addr = srt->addrs[i].s_addr;
    srt_split->laddrs = sizeof(struct in_addr) * i;

    memcpy(srt_split->addrs, srt->addrs, sizeof(struct in_addr) * i);

    return srt_split;
} « end dsr_srt_new_split »
```

若第 i 个源地址与 addr 的源地址相同，则截断 i 之后的
dsr_srt_new_split_rev   split 逆序

dsr_srt_shortcut
struct dsr_srt *dsr_srt_shortcut(struct dsr_srt *srt, struct in_addr a1,
                struct in_addr a2)
{
    struct dsr_srt *srt_cut;
    int i, j, n, n_cut, a1_num, a2_num;

    if (!srt)
        return NULL;

    a1_num = a2_num = -1;

    n = srt->laddrs / sizeof(struct in_addr);   记录总节点数

    if (srt->src.s_addr == a1.s_addr)   a1 为源节点
        a1_num = 0;

    /* Find out how between which node indexes to shortcut */ 记录 a1、a2 下标
    for (i = 0; i < n; i++) {
        if (srt->addrs[i].s_addr == a1.s_addr)

```
            a1_num = i + 1;
        if (srt->addrs[i].s_addr == a2.s_addr)
            a2_num = i + 1;
    }

    if (srt->dst.s_addr == a2.s_addr)                    a2 为目的节点
        a2_num = i + 1;

    n_cut = n - (a2_num - a1_num - 1);                   a1, a2 区间外有多少节点

    srt_cut = (struct dsr_srt *)MALLOC(sizeof(struct dsr_srt) +
                        (n_cut*sizeof(struct in_addr)),
                        GFP_ATOMIC);

    if (!srt_cut)
        return NULL;

    srt_cut->src = srt->src;                             构造削短后的路径
    srt_cut->dst = srt->dst;
    srt_cut->laddrs = n_cut * sizeof(struct in_addr);

    if (srt_cut->laddrs == 0)
        return srt_cut;

    j = 0;

    for (i = 0; i < n; i++) {
        if (i + 1 > a1_num && i + 1 < a2_num)
            continue;
        srt_cut->addrs[j++] = srt->addrs[i];
    }

    return srt_cut;
}
```

dsr_srt_concatenate   连接两个 srt 假设 srt1 的尾是 srt2 的头
dsr_srt_check_duplicate   检测是否有重复
dsr_srt_opt_add   添加源路由选项
dsr_srt_add   添加 packet 到源路由选项头

dsr_srt_opt_recv

```
1. int NSCLASS dsr_srt_opt_recv(struct dsr_pkt *dp, struct dsr_srt_opt *srt_opt
   )
2. {
```

```
3.     struct in_addr next_hop_intended;
4.     struct in_addr myaddr = my_addr();
5.     int n;
6.
7.     if (!dp || !srt_opt)
8.         return DSR_PKT_ERROR;
9.                                                          dp 赋值
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;
104. }
```

Endian.c

It solves the problem of non-existent <endian.h> on some systems by generating it.

Link-cache.c   维护缓存表相关的增删改查

Maint-buf.c

```
struct maint_entry {
    list_t l;
    struct in_addr nxt_hop;
    unsigned int rexmt;
    unsigned short id;
    struct timeval tx_time, expires;
    usecs_t rto;
    int ack_req_sent;
    struct dsr_pkt *dp;
};

struct maint_buf_query {
    struct in_addr *nxt_hop;
    unsigned short *id;
    usecs_t rtt;
};
```

All for buffered packets
crit_addr_id_del   从缓冲根据下一跳和 id 删除 packet
crit_addr_del   根据下一跳删除缓冲区的 packet
crit_addr   判断下一跳地址

crit_expires   判断是否超时

crit_ack_req_sent   判断是否发送

end for buffered packets

maint_buf_set_max_len 设置最大长度

maint_entry_create   构造 maint_entry

maint_buf_salvage   不知道在干什么

maint_buf_timeout   缓冲区超时

重传次数大于最大重传次数 丢弃

设置新传送时间，发送新的 ACK REQ，再次添加到维护缓冲区

maint_buf_set_timeout 设置超时时间

maint_buf_add   判断是否有邻居节点   检查是否添加 ACK REQ

maint_buf_del_all、maint_buf_del_all_id、maint_buf_del_addr 删除相关

maint_buf_print 打印缓冲区

maint_buf_get_info 获取长度

maint_buf_init   初始化缓冲区

maint_buf_cleanup   清空缓冲区

neigh.c

```
struct neighbor {
    list_t l;
    struct in_addr addr;
    struct sockaddr hw_addr;
    unsigned short id;
    struct timeval last_ack_req;
    usecs_t t_srtt, rto, t_rxtcur, t_rttmin, t_rttvar, jitter;  /* RTT in usec */
};

struct neighbor_query {
    struct in_addr *addr;
    struct neighbor_info *info;
};
```

crit_addr 设置队列

crit_addr_id_inc   判断 addr 是否相同，相同则 id+1

set_ack_req_time 设置 ack-请求时间

rto_calc   计算 trip timeout

neigh_tbl_create 、neigh_tbl_add、neigh_tbl_del   建表、增删

neigh_tbl_set_ack_req_time、neigh_tbl_set_rto 设置 ack 请求时间、设置 trip timeout（单次超时）

neigh_tbl_query 添加到邻居表中、neigh_tbl_id_inc 调用 crit_addr_id_inc、neigh_tbl_print 打印、neigh_tbl_proc_info 获取长度

neigh_tbl_init、neigh_tbl_cleanup   初始化邻居表、清空邻居表

send-buf.c

```
struct send_buf_entry {
    list_t l;
    struct dsr_pkt *dp;
    struct timeval qtime;
    xmit_fct_t okfn;
};
```

用于维护 buf

crit_addr 判断 addr 是否相同

crit_garbage 判断是否超时

send_buf_set_max_len 设置发送 buf 最大长度

send_buf_timeout   设置 buf 超时

send_buf_entry_create 构造 packet

send_buf_enqueue_packet 判断 buffer 是否满，排在队尾，若 buffer 满则移除第一个

send_buf_set_verdict   根据状态确定发送 buffer 内的还是丢弃 buffer 内的数据

send_buf_flush 刷新发送缓冲区、send_buf_print 打印发送缓冲区 、send_buf_get_info 获取缓冲区长度、send_buf_init 初始化缓冲区、   send_buf_cleanup 清空缓冲区