**网络协议栈分析与设计课程大作业**

|  |
| --- |
| **DSR路由协议代码分析** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 学号 | 姓名 | 班级 | 负责模块 | 成绩 | | 201692258 | 高子彭 | 软网1603 | 路由发现  通用包处理  列表操作  文档审核 |  | | 201692064 | 贺湧钧 | 软网1603 | 引言  代码介绍  数据结构 |  | | 201692155 | 张晟瑀 | 软网1603 | 路由维护  总结  整合 |  | |

目录

[第一章 引言 3](#_Toc534199650)

[第二章 代码介绍 4](#_Toc534199651)

[2.1 符号与缩写 4](#_Toc534199652)

[2.2 文件介绍 4](#_Toc534199653)

[2.3 全局变量 5](#_Toc534199654)

[第三章　数据结构 6](#_Toc534199655)

[3.1 DSR协议相关存储结构 6](#_Toc534199656)

[3.1.1 路由缓存(Route Cache) 6](#_Toc534199657)

[3.1.2 路由请求表(Route Request Table) 7](#_Toc534199658)

[3.1.3 发送缓冲区(Send Buffer) 8](#_Toc534199659)

[3.2 DSR分组结构 8](#_Toc534199660)

[3.2.1 DSR头(固定部分) 9](#_Toc534199661)

[3.2.2 路由请求选项(Route Request Option) 10](#_Toc534199662)

[3.2.3 路由应答选项(Route Reply Option) 11](#_Toc534199663)

[3.2.4 路由错误选项(Route Error Option) 12](#_Toc534199664)

[3.2.5 确认请求选项(Acknowledgment Request Option) 13](#_Toc534199665)

[3.2.6 确认选项(Acknowledgment Option) 13](#_Toc534199666)

[3.2.7 源路由选项(Source Route Option) 14](#_Toc534199667)

[第四章 DSR路由协议的实现 16](#_Toc534199668)

[4.1 分组处理 16](#_Toc534199669)

[4.1.1 概述 16](#_Toc534199670)

[4.1.2 dsr\_opt\_hdr\_add 函数 16](#_Toc534199671)

[4.1.3 建立IP结构dsr\_build\_ip 函数 17](#_Toc534199672)

[4.2接收处理 18](#_Toc534199673)

[4.2.1 处理接收到的DSR数据包dsr\_opt\_recv函数 18](#_Toc534199674)

[4.2.2 dsr\_rreq\_opt\_recv函数 20](#_Toc534199675)

[4.2.3发送RREP数据包 25](#_Toc534199676)

[4.3 列表操作函数 27](#_Toc534199677)

[4.3.1 判断是否是空表 27](#_Toc534199678)

[4.3.2 向表头添加表项 28](#_Toc534199679)

[4.3.3 向表尾添加表项 29](#_Toc534199680)

[4.3.4 根据表项ID查找 29](#_Toc534199681)

[4.3.5 删除t表中的l表项 30](#_Toc534199682)

[4.3.6 查找并执行操作 30](#_Toc534199683)

[4.3.7 对表中每个表项执行func操作 30](#_Toc534199684)

[4.3.8 查询表项并删除 31](#_Toc534199685)

[4.4路由维护处理 32](#_Toc534199686)

[4.4.1 概述 32](#_Toc534199687)

[4.4.2 maint\_buf\_salvage 函数 32](#_Toc534199688)

[4.4.3 dsr\_rerr\_send 函数 36](#_Toc534199689)

[第五章　总结 41](#_Toc534199690)

# 引言

动态源路由协议（DSR）是一种简单有效的路由协议，专门设计用于移动节点的多跳无线自组网。使用DSR，网络完全是自组织和自配置的，不需要现有的网络基础设施或管理。网络节点相互协作转发数据包，以允许不直接在彼此无线传输范围内的节点之间通过多个“跃点”进行通信。当网络中的节点移动、加入或离开网络，以及无线传输条件（如干扰源）发生变化时，所有路由都由DSR路由协议自动确定和维护。由于到达任何目的地所需的中间跳的数量或顺序可能随时发生变化，因此所产生的网络拓扑结构可能相当丰富且变化迅速。

DSR协议提供高响应性服务，以帮助确保在节点移动或网络条件发生其他变化的情况下成功传输数据包。

DSR协议由两个主要机制组成，它们共同工作，允许在即席网络中发现和维护源路由：

-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

## 2.1 符号与缩写

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 说明 | 符号 | 说明 |
| DEBUG\_BUFLEN  DBG\_LOG\_BUF\_LEN  DSR\_ACK\_REQ\_OPT\_LEN  DSR\_ACK\_OPT\_LEN  MAX\_MACKILL  NODE\_UNREACHABLE  STATE\_IDLE  RTC\_MAX\_LEN  LC\_NODES\_MAX  LC\_LINKS\_MAX | 测试时的缓冲长度  测试时记录的缓冲长度  ack请求选择部长度  ack选择部长度  MACKILL模块最大数量  连接不上的节点  空闲状态  rtc最大长度  路由缓存节点最大值  路由缓存链路最大值 | DBG\_LOG\_BUF\_MASK DSR\_ACK\_REQ\_HDR\_LEN  DSR\_ACK\_HDR\_LEN  CONFIG\_PROC\_NAME  DSR\_FIXED\_HDR\_LEN  DSR\_RREP\_HDR\_LEN  MAXTTL  IP\_HDR\_LEN  LC\_LINKS\_MAX  NEIGH\_TBL\_MAX\_LEN | 位置序号掩码  ack请求头部长度  ack头部长度  配置名称  fixed头部长度  应答头部长度  最大存活时间  IP头部长度  路由缓存链路最大值  相邻节点路由表最大长度 |

表2.1 DSR协议符号缩写说明

## 2.2 文件介绍

DSR协议是由C语言实现的，头文件加上源文件一共40个，下表给出了主要的文件并进行了说明：

|  |  |
| --- | --- |
| 文件名 | 说明 |
| debug.[ch]  dsr-ack.[ch]  dsr-io.[ch]  dsr-module.c  dsr-opt.[ch]  dsr-pkt.[ch]  dsr-rerr.[ch]  dsr-rrep.[ch]  dsr-rreq.[ch]  dsr-rtc(-simple).[ch]  dsr-srt.[ch]  dsr.h  endian.c  link-cache.[ch]  link.h  maint-buf.[ch]  neigh.[ch]  send-buf.[ch]  tbl.h  timer.h | 测试文件  命令确认字符ack  dsr中输入输出操作  dsr中模块的创建与定义  dsr中选择部分  dsr各项包  dsr操作中的错误  dsr应答操作  dsr请求操作  硬件时钟  dsr源路由选项  动态路由协议  用于解决某些系统中不存在的问题  路由缓存  链表操作  路由维护  相邻节点操作  发送缓存  路由表简单操作  时间器 |

表2.2 DSR 路由协议主要文件

## 2.3 全局变量

|  |  |  |
| --- | --- | --- |
| 变量 | 数据类型 | 说明 |
| dbg\_logbuf\_lock  \_\_dbg\_log\_buf[]  dbg\_log\_start  dbg\_log\_end  logged\_chars  \*dsr\_dev  \*dsr\_node  rp\_filter  forwarding  \*ifname  \*mackill  mackill\_list[][]  mackill\_len  rreq\_seqno  rtc\_len  rtc\_lock  rtc\_timer  LC  ack\_timer  send\_buf\_timer | spinlock\_t  char  unsigned long  unsigned long  unsigned long  struct net\_device  struct dsr\_node  int  int  char  char  unsigned char  int  unsigned int  int  rwlock\_t  DSRUUTimer  struct lc\_graph  DSRUUTimer  DSRUUTimer | 初始化自旋锁  初始化测试时记录的缓冲  dbg\_log\_buf中的索引  dbg\_log\_buf中的索引  自上次读取后生成的字符数+清除操作  初始化我们自己的dsr设备  初始化DSR\_节点  过滤器标记1或0  前进标记1或0  模块ifname名  模块mackill名  创建MACKILL表项  mackill长度  序列号  rtc长度  读写锁  rtc时间器  初始化路由缓存表  命令确认字符时间器  发送缓存时间器 |

表2.3 DSR项目中的全局变量

# 第三章　数据结构

## DSR协议相关存储结构

### 3.1.1 路由缓存(Route Cache)

网络中每个节点发送数据所需的路由信息部保存在节点的路由缓存中，当节点发现一条有效的路由信息时，就将其加入到自己的路由缓存中各用。路由缓存中可以保存多于一条的到同一目的地址的路由信息，这样，当一条路由失效后可以选用另一条来继续发送数据，而不需立刻就启动一次路由发现过程。

路由缓存的结构可以有两种选择即路式缓存(Path Cache)和链式缓存(Link Cache)，路式缓存中存放从该节点到达目的节点的完整路径，而链式缓存中存放网络中各个节点之间的链接关系。链式路由，利用它可以构造出潜在的其它路由信息，所以基于上述考虑，选择使用链式缓存，链式缓存结构定义如下

link-cache.c

1. **struct** lc\_node {
2. list\_t l;
3. **struct** in\_addr addr;
4. unsigned **int** links;
5. unsigned **int** cost;  /\* Cost estimate from source when running Dijkstra \*/
6. unsigned **int** hops;  /\* Number of hops from source. Used to get the
7. \* length of the source route to allocate. Same as
8. \* cost if cost is hops. \*/
9. **struct** lc\_node \*pred;   /\* predecessor \*/
10. };
12. **struct** lc\_link {
13. list\_t l;
14. **struct** lc\_node \*src, \*dst;
15. **int** status;
16. unsigned **int** cost;
17. **struct** timeval expires;
18. };
20. **struct** link\_query {
21. **struct** in\_addr src, dst;
22. };
24. **struct** cheapest\_node {
25. **struct** lc\_node \*n;
26. };

link-cache.c

图3.1 路有缓存结构定义

当节点获悉了一个新的链接时将该路由信息添加到它的路由缓存，首先需要检查routetable内是否已经包含该条路由（包括在cache中的路由信息）

当节点获悉到自组织网络中的现有链接已断开时，节点从其路由缓存中删除该条路由信息。通过接收Route Error Packet来判断链接是否断开。

### 3.1.2 路由请求表(Route Request Table)

路由请求表中记录着最近由本节点发起的路由请求或经本节点转发的路由请求信息。每当本节点发送一条路由请求或者从其它节点接收到的数据包中获得一条路由请求时，如果这条路由请求的ID在该节点的路由请求表中没有记录过，那么就将该路由请求添加到路由请求表中，然后做相应的处理。其结构定义如下：

dsr-rreq.c

1. **struct** rreq\_tbl\_entry {
2. list\_t l;
3. **int** state;
4. **struct** in\_addr node\_addr;
5. **int** ttl;//Time to Live
6. DSRUUTimer \*timer;
7. **struct** timeval tx\_time;
8. **struct** timeval last\_used;//直到最后使用的时间
9. usecs\_t timeout;
10. unsigned **int** num\_rexmts;//寻找的次数
11. **struct** tbl rreq\_id\_tbl;
12. };
14. **struct** id\_entry {
15. list\_t l;
16. **struct** in\_addr trg\_addr;
17. unsigned **short** id;
18. };
19. **struct** rreq\_tbl\_query {
20. **struct** in\_addr \*initiator;
21. **struct** in\_addr \*target;
22. unsigned **int** \*id;
23. };

dsr-rreq.c

图3.2 路由请求表结构定义

id\_entry 用来存储经本机转发的路由请求信息. trg\_addr 是产生这条路由请求的节点的IP地址。

rreq\_tbl\_entry用来存储本机产生的路由请求信息.node\_addr是这条路由请求所要查询的目标节点的IP地址:timer记录了这条路由请求最近一次的发送时间,在timcout时间内如果本节点没有收到对应该请求的路由应答,如果此时num\_rexmts记录的重传次数还没有到达最大重传次数的限制，那么就重发这条路由请求。

### 3.1.3 发送缓冲区(Send Buffer)

当一个数据包由于目前没有合适的路由而无法立即发送时,节点为它启动一次路由发现过程并将它放入路由缓冲区中. 当路由发现过程返回合适的路由信息后节点再将该数据包从发送缓冲区中取出并发送. 发送缓冲区结构定义如下:

send-buf.c

1. **struct** send\_buf\_entry {
2. list\_t l;
3. **struct** dsr\_pkt \*dp;
4. **struct** timeval qtime;
5. xmit\_fct\_t okfn;
6. };

send-buf.c

图3.3 发送缓存区结构定义

发送缓冲区是一个暂存不能立即发送的数据包的缓冲区。我们定义了一个 send\_buf\_entry 结构用来记录缓冲区中每一个数据包的发送信息:l是记录该数据包的链表;dp 是一个函数指针, 它指向用来发送该数据包的网络设备: qtime 记录了该数据包被放入发送缓冲区的时间、失效时间 timeout 以及失效后的处理函数等信息,在timeout 时间后, 如果还没有合适的路出能发送该数据包的话, 就丢弃这个数据包.

## DSR分组结构

DSR协议使用一个结构特殊的DSR头来携带控制信息, 这个DSR头可以被任何已经存在的IP分组携带.DSR头的开始部分是一个固定尺寸的结构体,其后跟随若干个携带DSR选项信息的DSR选项结构体。整个DSR头的结束由包含在开始部分固定结构体中的头部长度来指示。DSR头应该被插入到现存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3.4所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4号版本号 | 4号首部长度 | | 8位服务类型(TOS) | 16位总长度(字节数) | |
| 16位标识 | | | | 3位标志 | 13位片偏移 |
| 8位生存时间(TTL) | | | 8位协议 | 16位首部校验和 | |
| 32位源IP地址 | | | | | |
| 32位目的IP地址 | | | | | |
| 选项(如果有) | | | | | |
| 8位DSR头固定部分 | | …… DSR option部分 …… | | | |
| 传输层协议头部信息 | | | | | |

图3.4 DSR分组结构

插入DSR头以后,IP头的结构无需改动但IP头中的若干个域需要修改以区别DSR包与普通IP包：IP头中的protocol域需要修改为DSR协议标识;由于DSR信息的播入,整个数据包的长度即头中的总长度域也要被修改: IP头的目的地址应该枝修改为DSR源路由中的下一跳地址, 如果是DSR路由请求包则应该将其置为本地广播地址:IP头中任意其它域的改动都会引起头部校验和 (Checksum) 的变化。

### 3.2.1 DSR头(固定部分)

DSR头携带了每个DSR数据包中所必有的一些信息。他的格式如下图3.5所示。

|  |  |  |
| --- | --- | --- |
| next header | reserved | Payload Length |
| Option | | |

图3.5 DSR头部结构

其中next header域指示紧跟在DSR头后面的是哪个协议头.。Payload Length域指示DSR头中整个option域的长度。option域是一个变长的数据区,各种DSR选项在需要时都被插入到这个option域中传输。在我们的程序中DSR头是如下定义的:

dsr-opt.h

1. /\* The DSR options header (always comes first) \*/
2. **struct** dsr\_opt\_hdr {
3. u\_int8\_t nh;
4. #if defined(\_\_LITTLE\_ENDIAN\_BITFIELD)
6. u\_int8\_t res:7;
7. u\_int8\_t f:1;
8. #elif defined (\_\_BIG\_ENDIAN\_BITFIELD)
9. 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\_;
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. }
25. **int** size() {
26. **return** ntohs(p\_len) + **sizeof**(**struct** dsr\_opt\_hdr);
27. }
28. #endif              /\* NS2 \*/
29. **struct** dsr\_opt option[0];
30. };

dsr-opt.h

图3.6 DSR头结构定义

### 3.2.2 路由请求选项(Route Request Option)

路由请求选项的格式如下图3.7所示。

当DSR头中包含路由请求选项时, 承载这个DSR包的IP数据包的源地址应该被置为发送这个路由请求的源节点地址, 在这个数据包的传播过程中, 源地址域不能被改动:目的地址应该被置为广播地址。选项类型域为2, Opt Data Len为除过选项类型和长度域外, 路由请求选项中其它内容的长度。标志符域是一个由发出这个路由请求的源节点产生的一个区别于其它的唯一值, 用来区别该路由请求和本节点发出的其它路由请求。目的节点地址城记录着该路由请求所要查询的目的节点地址, 后面紧跟着的是该路由请求从源节点到目的节点前所经过的中间节点的地址列表。

|  |  |  |
| --- | --- | --- |
| Option type | Opt data len | Identification |
| Target address | | |
| Address[1] | | |
| Address[2] | | |
| …… | | |
| Address[n] | | |

图3.7 路由请求选项结构

在我们的程序中路由请求选项是如下定义的：

dsr-rreq.h

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. };

dsr-rreq.h

图3.8 路由请求选项结构定义

### 3.2.3 路由应答选项(Route Reply Option)

路由应答选项的格式如图3.9所示。

|  |  |  |  |
| --- | --- | --- | --- |
| Opt  Date len | L | Reserved | identification |
| Address[1] | | | |
| Address[2] | | | |
| …… | | | |
| Address[n] | | | |

图3.9 路由应答选项结构

当DSR头中包含路由应答选项时, 承载这个DSR包的IP数据包的源地址应该被置为发送这个路由应答的源节点地址:目的地址和收到的承载路由请求的IP包的源地址相同。 选项类型 (Option Type) 域为3, Opt Data Ien为除过选项类型和长度域外, 路由应答选项中其它内容的长度. 外部最后一跳 (Last Hop External) 指示由该路由应答所返回的路由的最后一跳的节点是一个DSR子网外部的节点。 标志符 (Idcntificaton) 域的值和引起该路由应答的路由请求的标志符域的值相同, 后面紧跟着的是从以承载该路由应答的IP包的目的地址域的地址所指示的节点到它所查询的目标节点的源路出列表, Address[n]即为它所查询的目表节点的地址.在我们的程序中路由应答选项是如下定义的:

dsr-rrep.h

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. };

dsr-rrep.h

图3.10 路由应答选项结构定义

### 3.2.4 路由错误选项(Route Error Option)

路由错误选项的格式如图3.11所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Option type | Opt data Len | Error Type | Reserved | Slavage |
| Error Source Address | | | | |
| Error Destination Address | | | | |
| Type-Specific Information | | | | |

图3.11 路由错误选项结构

路由错误的选项类型域为4, 错误类型域指示出错类型, 根据IETF的DSR草案, 目前仅定义了"1 = NODE\_UNREACHABLE"这一种类型的错误, 错误源地址域中的地址即为产生这条路由错误的节点的地址, 错误目的地址域中的地址是该路由错误信息必须通知到的节点, 也就是那个因为发送的数据包在到达目的节点之前遇到了链路失效的节点. 特定的类型信息中填写的是关于这条路由错误类型的具体信息, 目前在这个数据域内仅仅是写入那个目前不可达的节点的地址. 在我们的程序中路由应答选项是如下定义的:

dsr-rerr.h

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. };

dsr-rerr.h

图3.12 路由错误选项结构定义

### 3.2.5 确认请求选项(Acknowledgment Request Option)

确认请求选项的格式如图3.13所示。

|  |  |  |
| --- | --- | --- |
| Option type | Opt Data Len | identification |
| ACK Request Source Address | | |

图3.13 确认请求选项结构

确认请求的选项类型值为5，标志符域由接收到DSR数据包的节点置为一个非零的数值用于区别其它的确认请求信息，确认请求源地址中保存的是发出这条确认请求的源节点的IP地址。在我们的程序中路由应答选项是如下定义的：

dsr-ack.h

1. **struct** dsr\_ack\_req\_opt {
2. u\_int8\_t type;
3. u\_int8\_t length;
4. u\_int16\_t id;
5. };

dsr-ack.h

图3.14 确认请求选项结构定义

### 3.2.6 确认选项(Acknowledgment Option)

确认选项的格式如图3.15所示。

|  |  |  |
| --- | --- | --- |
| Option type | Opt Data Len | identification |
| ACK Source Address | | |
| ACK Destination Address | | |

图3.15 确认选项结构

确认选项的类型值为6，标志符域的值和引发该确认的请求选项的标志符域的值相同，在我们的程序中路由应答选项是如下定义的：

dsr-ack.h

1. **struct** dsr\_ack\_opt {
2. u\_int8\_t type;
3. u\_int8\_t length;
4. u\_int16\_t id;
5. u\_int32\_t src;
6. u\_int32\_t dst;
7. };

dsr-ack.h

图3.16 确认选项结构定义

### 3.2.7 源路由选项(Source Route Option)

源路由选项的格式如图3.17所示。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Option Type | Opt Data Len | F | L | Reserved | Slavage | Segs Left |
| Address[1] | | | | | | |
| Address[2] | | | | | | |
| …… | | | | | | |
| Address[n] | | | | | | |

图3.17 源路由选项结构

源路由选项的选项类型值为7，Segs Left域中记录的是根据Address[]数组记录的源路由列表，从当前节点到目标节点所要经历的跳数。源路由选项结构定义如下：

dsr-srt.h

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. };

dsr-srt.h

图3.18 源路由选项结构定义

# 第四章 DSR路由协议的实现

## 分组处理

### 概述

当打包一个使用DSR协议的数据包的时候，必须遵守以下这几步规则：

1. 在路由缓存中搜索在这个数据包头部中的目的地址。
2. 如果没有找到到达目的地址的路由，就启动路由寻找机制来寻找到达目的地址的路由（下一节将会介绍），为此目标节点地址启动路由发现会导致节点在此现有数据包的DSR选项标头中添加路由请求选项，或者将此现有数据包保存到其发送缓冲区，并通过发送包含路由请求选项的单独数据包来启动路由发现。如果节点选择通过向现有数据包添加路由请求选项来启动路由发现，那么它将使用IP“有限广播”地址（255.255.255.255）替换IP目标地址字段，并且复制原始IP目标 添加到新数据包的Route Request选项的Target Address字段的地址。
3. 如果数据包现在不包含路由请求选项，则此节点必须具有到数据包的目标地址的路由; 如果节点有多个到此目标地址的路由，则该节点选择一个用于传送此数据包。如果此路由的长度大于1跳，或者节点确定从该路由中的第一跳节点请求DSR网络层确认，则将DSR选项头插入数据包，如第8.1.2节所述，并插入DSR源路由选项。数据包中的源路由从所选路由初始化为数据包的目标地址。
4. 将数据包传输到选定源路由中给出的第一跳节点地址，并且使用路由维护确定下一跳的可达性。

### 4.1.2 dsr\_opt\_hdr\_add 函数

发送DSR分组的节点将DSR选项添加到分组头部，以携带路由协议所需的信息。 数据包绝不能包含多个DSR选项头。

dsr-opt.c

1. **struct** dsr\_opt\_hdr \*dsr\_opt\_hdr\_add(**char** \*buf, unsigned **int** len,
2. unsigned **int** protocol)
3. {
4. **struct** dsr\_opt\_hdr \*opt\_hdr;
6. **if** (len < DSR\_OPT\_HDR\_LEN)
7. **return** NULL;
9. opt\_hdr = (**struct** dsr\_opt\_hdr \*)buf;
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);
16. **return** opt\_hdr;
17. }

dsr-opt.c

11 将八位无符号整型opt\_hdr->nh 的值置为protocol

12 将Flow State Header（F）的值置为0用于辨识是否是Flow State Header

13 发送时将七位的保留位（Reserved）置为0，并且在接收时忽略该选项

14-16 将有效载荷长度（p\_len）置为数据包的长度减去头部的长度，返回一个数据包首部

### 4.1.3 建立IP结构dsr\_build\_ip 函数

此函数用于生成IP首部

dsr-opt.c

1. **struct** iphdr \*dsr\_build\_ip(**struct** dsr\_pkt \*dp, **struct** in\_addr src,
2. **struct** in\_addr dst, **int** ip\_len, **int** tot\_len,
3. **int** protocol, **int** ttl)
4. {
5. **struct** iphdr \*iph;
7. dp->nh.iph = iph = (**struct** iphdr \*)dp->ip\_data;
9. **if** (dp->skb && dp->skb->nh.raw) {
10. memcpy(dp->ip\_data, dp->skb->nh.raw, ip\_len);
11. } **else** {
12. iph->version = IPVERSION;
13. iph->ihl = 5;
14. iph->tos = 0;
15. iph->id = 0;
16. iph->frag\_off = 0;
17. iph->ttl = (ttl ? ttl : IPDEFTTL);
18. iph->saddr = src.s\_addr;
19. iph->daddr = dst.s\_addr;
20. }
22. iph->tot\_len = htons(tot\_len);
23. iph->protocol = protocol;
24. ip\_send\_check(iph);
26. **return** iph;
27. }

dsr-opt.c

9-20 当dp->skb和dp->skb->nh.raw都不为空时，将dp->skb->nh.raw中的数据的前ip\_len字节的数据拷贝置dp-skb中，否则将对IP首部的各个字节进行分别复制

22-23 修改IP首部的tot\_len选项为传入的参数值

## 4.2接收处理

### 4.2.1 处理接收到的DSR数据包dsr\_opt\_recv函数

dsr-opt.c

1. **int** NSCLASS dsr\_opt\_recv(**struct** dsr\_pkt \*dp)  //收到一个含option的dsr包
2. {
3. **int** dsr\_len, l;
4. **int** action = 0;
5. **struct** dsr\_opt \*dopt;
6. **struct** in\_addr myaddr;
8. **if** (!dp)
9. **return** DSR\_PKT\_ERROR;
11. myaddr = my\_addr();
13. /\* Packet for us ? \*/
14. #ifdef NS2
15. //DEBUG("Next header=%s\n", packet\_info.name((packet\_t)dp->dh.opth->nh));
17. **if** (dp->dst.s\_addr == myaddr.s\_addr &&
18. (DATA\_PACKET(dp->dh.opth->nh) || dp->dh.opth->nh == PT\_PING))
19. action |= DSR\_PKT\_DELIVER; //向上层递交
20. #else
21. **if** (dp->dst.s\_addr == myaddr.s\_addr && dp->payload\_len != 0)
22. action |= DSR\_PKT\_DELIVER; //向上层递交
23. #endif
24. dsr\_len = dsr\_pkt\_opts\_len(dp);
26. l = DSR\_OPT\_HDR\_LEN;
27. dopt = DSR\_GET\_OPT(dp->dh.opth);
29. //DEBUG("Parsing DSR packet l=%d dsr\_len=%d\n", l, dsr\_len);
31. **while** (l < dsr\_len && (dsr\_len - l) > 2) {
32. //DEBUG("dsr\_len=%d l=%d\n", dsr\_len, l);
33. **switch** (dopt->type) {
34. **case** DSR\_OPT\_PADN:
35. **break**;
36. **case** DSR\_OPT\_RREQ: //如果是收到一个RREQ包
37. **if** (dp->flags & PKT\_PROMISC\_RECV)
38. **break**;
40. action |= dsr\_rreq\_opt\_recv(dp, (**struct** dsr\_rreq\_opt \*)dopt); //action 与dsr\_rreq\_opt\_recv(,)按位或运算后赋值给action
41. **break**;
42. **case** DSR\_OPT\_RREP: //如果是收到一个RREP包
43. **if** (dp->flags & PKT\_PROMISC\_RECV)
44. **break**;
46. action |= dsr\_rrep\_opt\_recv(dp, (**struct** dsr\_rrep\_opt \*)dopt);  //同上
47. **break**;
48. **case** DSR\_OPT\_RERR:  //如果收到的是RERR包
49. **if** (dp->flags & PKT\_PROMISC\_RECV)
50. **break**;
51. **if** (dp->num\_rerr\_opts < MAX\_RERR\_OPTS) {
52. action |=
53. dsr\_rerr\_opt\_recv(dp, (**struct** dsr\_rerr\_opt \*)dopt);  //同上
54. }
56. **break**;
57. **case** DSR\_OPT\_PREV\_HOP:  //
58. **break**;
59. **case** DSR\_OPT\_ACK:
60. **if** (dp->flags & PKT\_PROMISC\_RECV)
61. **break**;
63. **if** (dp->num\_ack\_opts < MAX\_ACK\_OPTS) {
64. dp->ack\_opt[dp->num\_ack\_opts++] =
65. (**struct** dsr\_ack\_opt \*)dopt;
66. action |=
67. dsr\_ack\_opt\_recv((**struct** dsr\_ack\_opt \*)
68. dopt);
69. }
70. **break**;
71. **case** DSR\_OPT\_SRT:
72. action |= dsr\_srt\_opt\_recv(dp, (**struct** dsr\_srt\_opt \*)dopt);
73. **break**;
74. **case** DSR\_OPT\_TIMEOUT:
75. **break**;
76. **case** DSR\_OPT\_FLOWID:
77. **break**;
78. **case** DSR\_OPT\_ACK\_REQ:
79. action |=
80. dsr\_ack\_req\_opt\_recv(dp, (**struct** dsr\_ack\_req\_opt \*)
81. dopt);
82. **break**;
83. **case** DSR\_OPT\_PAD1:
84. l++;
85. dopt++;
86. **continue**;
87. **default**:
88. DEBUG("Unknown DSR option type=%d\n", dopt->type);
89. }
90. l += dopt->length + 2;
91. dopt = DSR\_GET\_NEXT\_OPT(dopt);
92. }
93. **return** action;
94. }

dsr-rrep.c

36-40 如果是收到一个RREQ包且该数据包的flag与PKT\_PROMISC\_RECV进行按位与操作后为false则调用dsr\_rreq\_opt\_recv函数用于处理RREQ消息

42-46 如果是收到一个RREP包且该数据包的flag与PKT\_PROMISC\_RECV进行按位与操作后为false则调用dsr\_rrep\_opt\_recv函数用于处理RREP消息

48-53 如果是收到一个RERR包且该数据包的flag与PKT\_PROMISC\_RECV进行按位与操作后为false则调用dsr\_rerr\_opt\_recv函数用于处理RERR消息

59-69 如果是收到一个ACK包且该数据包的flag与PKT\_PROMISC\_RECV进行按位与操作后为false则调用dsr\_ack\_opt\_recv函数用于处理ACK消息

71-72 如果是收到一个SRT包则调用dsr\_srt\_opt\_recv函数用于处理SRT消息

78-81 如果是收到一个ACK\_REQ包则调用dsr\_ack\_req\_opt\_recv函数用于处理SRT消息

87-88 如果是无法辨识的消息类型则返回错误消息

### 4.2.2 dsr\_rreq\_opt\_recv函数

当一个节点收到路由请求包时，此节点必须对该路由请求包进行处理，具体的处理过程如下：

dsr-rreq.c

1. **int** NSCLASS dsr\_rreq\_opt\_recv(**struct** dsr\_pkt \*dp, **struct** dsr\_rreq\_opt \*rreq\_opt)
2. {
3. **struct** in\_addr myaddr;
4. **struct** in\_addr trg;
5. **struct** dsr\_srt \*srt\_rev, \*srt\_rc;
6. **int** action = DSR\_PKT\_NONE;
7. **int** i, n;
9. **if** (!dp || !rreq\_opt || dp->flags & PKT\_PROMISC\_RECV)
10. **return** DSR\_PKT\_DROP;
12. dp->num\_rreq\_opts++;
14. **if** (dp->num\_rreq\_opts > 1) {
15. DEBUG("More than one RREQ opt!!! - Ignoring\n");
16. **return** DSR\_PKT\_ERROR;
17. }
19. dp->rreq\_opt = rreq\_opt;
21. myaddr = my\_addr();
23. trg.s\_addr = rreq\_opt->target;
25. **if** (dsr\_rreq\_duplicate(dp->src, trg, ntohs(rreq\_opt->id))) {
26. DEBUG("Duplicate RREQ from %s\n", print\_ip(dp->src));
27. **return** DSR\_PKT\_DROP;
28. }
30. rreq\_tbl\_add\_id(dp->src, trg, ntohs(rreq\_opt->id));
32. dp->srt = dsr\_srt\_new(dp->src, myaddr, DSR\_RREQ\_ADDRS\_LEN(rreq\_opt),
33. (**char** \*)rreq\_opt->addrs);
35. **if** (!dp->srt) {
36. DEBUG("Could not extract source route\n");
37. **return** DSR\_PKT\_ERROR;
38. }
39. DEBUG("RREQ target=%s src=%s dst=%s laddrs=%d\n",
40. print\_ip(trg), print\_ip(dp->src),
41. print\_ip(dp->dst), DSR\_RREQ\_ADDRS\_LEN(rreq\_opt));
43. /\* Add reversed source route \*/
44. srt\_rev = dsr\_srt\_new\_rev(dp->srt);
46. **if** (!srt\_rev) {
47. DEBUG("Could not reverse source route\n");
48. **return** DSR\_PKT\_ERROR;
49. }
50. DEBUG("srt: %s\n", print\_srt(dp->srt));
51. DEBUG("srt\_rev: %s\n", print\_srt(srt\_rev));
53. dsr\_rtc\_add(srt\_rev, ConfValToUsecs(RouteCacheTimeout), 0);
55. /\* Set previous hop \*/
56. **if** (srt\_rev->laddrs > 0)
57. dp->prv\_hop = srt\_rev->addrs[0];
58. **else**
59. dp->prv\_hop = srt\_rev->dst;
61. neigh\_tbl\_add(dp->prv\_hop, dp->mac.ethh);
63. /\* Send buffered packets \*/
64. send\_buf\_set\_verdict(SEND\_BUF\_SEND, srt\_rev->dst);
66. **if** (rreq\_opt->target == myaddr.s\_addr) {
68. DEBUG("RREQ OPT for me - Send RREP\n");
70. /\* According to the draft, the dest addr in the IP header must
71. \* be updated with the target address \*/
72. #ifdef NS2
73. dp->nh.iph->daddr() = (nsaddr\_t) rreq\_opt->target;
74. #else
75. dp->nh.iph->daddr = rreq\_opt->target;
76. #endif
77. dsr\_rrep\_send(srt\_rev, dp->srt);
79. action = DSR\_PKT\_NONE;
80. **goto** out;
81. }
83. n = DSR\_RREQ\_ADDRS\_LEN(rreq\_opt) / **sizeof**(**struct** in\_addr);
85. **if** (dp->srt->src.s\_addr == myaddr.s\_addr)
86. **return** DSR\_PKT\_DROP;
88. **for** (i = 0; i < n; i++)
89. **if** (dp->srt->addrs[i].s\_addr == myaddr.s\_addr) {
90. action = DSR\_PKT\_DROP;
91. **goto** out;
92. }
94. /\* TODO: Check Blacklist \*/
95. srt\_rc = lc\_srt\_find(myaddr, trg);
97. **if** (srt\_rc) {
98. **struct** dsr\_srt \*srt\_cat;
99. /\* Send cached route reply \*/
101. DEBUG("Send cached RREP\n");
103. srt\_cat = dsr\_srt\_concatenate(dp->srt, srt\_rc);
105. FREE(srt\_rc);
107. **if** (!srt\_cat) {
108. DEBUG("Could not concatenate\n");
109. **goto** rreq\_forward;
110. }
112. DEBUG("srt\_cat: %s\n", print\_srt(srt\_cat));
114. **if** (dsr\_srt\_check\_duplicate(srt\_cat) > 0) {
115. DEBUG("Duplicate address in source route!!!\n");
116. FREE(srt\_cat);
117. **goto** rreq\_forward;
118. }
119. #ifdef NS2
120. dp->nh.iph->daddr() = (nsaddr\_t) rreq\_opt->target;
121. #else
122. dp->nh.iph->daddr = rreq\_opt->target;
123. #endif
124. DEBUG("Sending cached RREP to %s\n", print\_ip(dp->src));
125. dsr\_rrep\_send(srt\_rev, srt\_cat);
127. action = DSR\_PKT\_NONE;
129. FREE(srt\_cat);
130. } **else** {
132. rreq\_forward:
133. dsr\_pkt\_alloc\_opts\_expand(dp, **sizeof**(**struct** in\_addr));
135. **if** (!DSR\_LAST\_OPT(dp, rreq\_opt)) {
136. **char** \*to, \*from;
137. to = (**char** \*)rreq\_opt + rreq\_opt->length + 2 +
138. **sizeof**(**struct** in\_addr);
139. from = (**char** \*)rreq\_opt + rreq\_opt->length + 2;
141. memmove(to, from, **sizeof**(**struct** in\_addr));
142. }
143. rreq\_opt->addrs[n] = myaddr.s\_addr;
144. rreq\_opt->length += **sizeof**(**struct** in\_addr);
146. dp->dh.opth->p\_len = htons(ntohs(dp->dh.opth->p\_len) +
147. **sizeof**(**struct** in\_addr));
148. #ifdef \_\_KERNEL\_\_
149. dsr\_build\_ip(dp, dp->src, dp->dst, IP\_HDR\_LEN,
150. ntohs(dp->nh.iph->tot\_len) +
151. **sizeof**(**struct** in\_addr), IPPROTO\_DSR,
152. dp->nh.iph->ttl);
153. #endif
154. /\* Forward RREQ \*/
155. action = DSR\_PKT\_FORWARD\_RREQ;
156. }
157. out:
158. FREE(srt\_rev);
159. **return** action;
160. }

dsr-rreq.c

66-81 如果该节点为是路由请求包中的目的节点，那么该节点通过调用dsr\_rrep\_send函数回复一个路由应答包给请求方，并且停止转发

85-86 如果该节点的IP地址与DSR数据包中的源IP地址相同，则返回DSR\_PKT\_DROP宏。

88-92 遍历DSR数据包的首部IP地址列表，检查该数据包所经过的节点IP以确定该数据包是否是自己发出的数据包，或者是自己已经转发过的数据包。如果是的话，该节点将会丢弃该数据包，并返回DSR\_PKT\_DROP宏。

95-130如果该数据包是通过需要双向链路进行单播传输的网络接口所接收到。则该节点需要检查该路由请求是否由其黑名单上的节点所发送，如果是由其黑名单上的节点所发送的，

则丢弃该数据包。

130-156如果该数据包不是由黑名单上的IP地址所发送，则该节点需要再进一步处理该条路由请求：

A．在此最近收到的路由请求的（标识，目标地址）值的缓存中添加此路径请求的条目。

B．创建整个数据包的副本，并在数据包副本上执行以下步骤。

C． 将此节点自己的IP地址附加到数据包中的Address [i]值列表，并将Route Request中的Opt Data Len字段的值增加4（IP地址的大小）。 但是，如果该节点有多个网络接口，则必须通过sec：multiple中指定的特殊处理修改此步骤。

D．该节点应该在其自己的路由缓存中搜索到该路由请求的目标的路由（从其自身，就好像它是数据包的源）。 如果在路由缓存中找到这样的路由，则该节点应该按照8.2.3节中概述的过程向该路由请求的发起者返回“缓存的路由回复”，如果在那里指定的限制允许的话。

### 4.2.3发送RREP数据包

Route Reply选项必须要包含在返回给发起者的数据包的DSR Options头中。 要初始化Route Reply选项，节点将执行以下步骤：

dsr-rrep.c

1. **int** NSCLASS dsr\_rrep\_send(**struct** dsr\_srt \*srt, **struct** dsr\_srt \*srt\_to\_me)
2. {
3. **struct** dsr\_pkt \*dp = NULL;
4. **char** \*buf;
5. **int** len, ttl, n;
7. **if** (!srt || !srt\_to\_me)
8. **return** -1;
10. dp = dsr\_pkt\_alloc(NULL);
12. **if** (!dp) {
13. DEBUG("Could not allocate DSR packet\n");
14. **return** -1;
15. }
17. dp->src = my\_addr();
18. dp->dst = srt->dst;
20. **if** (srt->laddrs == 0)
21. dp->nxt\_hop = dp->dst;
22. **else**
23. dp->nxt\_hop = srt->addrs[0];
25. len = DSR\_OPT\_HDR\_LEN + DSR\_SRT\_OPT\_LEN(srt) +
26. DSR\_RREP\_OPT\_LEN(srt\_to\_me)/\*  + DSR\_OPT\_PAD1\_LEN \*/;
28. n = srt->laddrs / **sizeof**(**struct** in\_addr);
30. DEBUG("srt: %s\n", print\_srt(srt));
31. DEBUG("srt\_to\_me: %s\n", print\_srt(srt\_to\_me));
32. DEBUG("next\_hop=%s\n", print\_ip(dp->nxt\_hop));
33. DEBUG
34. ("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",
35. IP\_HDR\_LEN, DSR\_OPT\_HDR\_LEN, DSR\_SRT\_OPT\_LEN(srt),
36. DSR\_RREP\_OPT\_LEN(srt\_to\_me), DSR\_OPT\_PAD1\_LEN, len);
38. ttl = n + 1;
40. DEBUG("TTL=%d, n=%d\n", ttl, n);
42. buf = dsr\_pkt\_alloc\_opts(dp, len);
44. **if** (!buf)
45. **goto** out\_err;
47. dp->nh.iph = dsr\_build\_ip(dp, dp->src, dp->dst, IP\_HDR\_LEN,
48. IP\_HDR\_LEN + len, IPPROTO\_DSR, ttl);
50. **if** (!dp->nh.iph) {
51. DEBUG("Could not create IP header\n");
52. **goto** out\_err;
53. }
55. dp->dh.opth = dsr\_opt\_hdr\_add(buf, len, DSR\_NO\_NEXT\_HDR\_TYPE);
57. **if** (!dp->dh.opth) {
58. DEBUG("Could not create DSR options header\n");
59. **goto** out\_err;
60. }
62. buf += DSR\_OPT\_HDR\_LEN;
63. len -= DSR\_OPT\_HDR\_LEN;
65. /\* Add the source route option to the packet \*/
66. dp->srt\_opt = dsr\_srt\_opt\_add(buf, len, 0, dp->salvage, srt);
68. **if** (!dp->srt\_opt) {
69. DEBUG("Could not create Source Route option header\n");
70. **goto** out\_err;
71. }
73. buf += DSR\_SRT\_OPT\_LEN(srt);
74. len -= DSR\_SRT\_OPT\_LEN(srt);
76. dp->rrep\_opt[dp->num\_rrep\_opts++] =
77. dsr\_rrep\_opt\_add(buf, len, srt\_to\_me);
79. **if** (!dp->rrep\_opt[dp->num\_rrep\_opts - 1]) {
80. DEBUG("Could not create RREP option header\n");
81. **goto** out\_err;
82. }
84. /\* TODO: Should we PAD? The rrep struct is padded and aligned
85. \* automatically by the compiler... How to fix this? \*/
86. dp->flags |= PKT\_XMIT\_JITTER;
88. XMIT(dp);
90. **return** 0;
91. out\_err:
92. **if** (dp)
93. dsr\_pkt\_free(dp);
95. **return** -1;
96. }

dsr-rrep.c

17-18 将DSR数据包的源地址置为该节点自己的IP，将目的地址置为发起路由寻找的地址

20-23 如果地址长度为0的话，则将下一跳地址置为目的地址，否则将下一跳地址置为address[0]的地址

47-48 调用dsr\_build\_ip函数进行IP包装

73-74 更新buf和len

## 4.3 列表操作函数

### 4.3.1 判断是否是空表

tbl.h

1. **static** **inline** **int** tbl\_empty(**struct** tbl \*t) /\*判断是否是空表\*/
2. {
3. **int** res = 0;
4. DSR\_READ\_LOCK(&t->lock);
6. **if** (TBL\_FIRST(t) == &(t)->head)
7. res = 1;
9. DSR\_READ\_UNLOCK(&t->lock);
10. **return** res;
11. }

tbl.h

### 4.3.2 向表头添加表项

tbl.h

1. **static** **inline** **int** \_\_tbl\_add(**struct** tbl \*t, list\_t \* l, criteria\_t crit) //向前添加表项
2. {
3. **int** len;
5. **if** (t->len >= t->max\_len) {
6. //printk(KERN\_WARNING "Max list len reached\n");
7. **return** -ENOSPC;
8. }
10. **if** (list\_empty(&t->head)) {
11. list\_add(l, &t->head);
12. } **else** {
13. list\_t \*pos;
15. list\_for\_each(pos, &t->head) {
17. **if** (crit(pos, l))
18. **break**;
19. }
20. list\_add(l, pos->prev);
21. }
23. len = ++t->len;
25. **return** len;
26. }

tbl.h

### 4.3.3 向表尾添加表项

tbl.h

1. **static** **inline** **int** \_\_tbl\_add\_tail(**struct** tbl \*t, list\_t \* l) //向尾部添加表项
2. {
3. **int** len;
5. **if** (t->len >= t->max\_len) {
6. //printk(KERN\_WARNING "Max list len reached\n");
7. **return** -ENOSPC;
8. }
10. list\_add\_tail(l, &t->head);
12. len = ++t->len;
14. **return** len;
15. }

tbl.h

### 4.3.4 根据表项ID查找

tbl.h

1. **static** **inline** **void** \*\_\_tbl\_find(**struct** tbl \*t, **void** \*id, criteria\_t crit) //查找节点
2. {
3. list\_t \*pos;
5. list\_for\_each(pos, &t->head) {
6. **if** (crit(pos, id))
7. **return** pos;
8. }
9. **return** NULL;
10. }

tbl.h

### 4.3.5 删除t表中的l表项

tbl.h

1. **static** **inline** **void** \*\_\_tbl\_detach(**struct** tbl \*t, list\_t \* l) //删除表
2. {
3. **int** len;
5. **if** (TBL\_EMPTY(t))
6. **return** NULL;
8. list\_del(l);
10. len = --t->len;
12. **return** l;
13. }

tbl.h

### 4.3.6 查找并执行操作

tbl.h

1. **static** **inline** **int** \_\_tbl\_find\_do(**struct** tbl \*t, **void** \*data, do\_t func)
2. {
3. list\_t \*pos, \*tmp;
5. list\_for\_each\_safe(pos, tmp, &t->head)
6. **if** (func(pos, data))
7. **return** 1;
9. **return** 0;
10. }

tbl.h

### 4.3.7 对表中每个表项执行func操作

tbl.h

1. **static** **inline** **int** \_\_tbl\_do\_for\_each(**struct** tbl \*t, **void** \*data, do\_t func)
2. {
3. list\_t \*pos;
4. **int** res = 0;
6. list\_for\_each(pos, &t->head)
7. res += func(pos, data);
9. **return** res;
10. }

tbl.h

### 4.3.8 查询表项并删除

tbl.h

1. **static** **inline** **void** \*tbl\_find\_detach(**struct** tbl \*t, **void** \*id, criteria\_t crit)
2. {
3. list\_t \*e;
5. DSR\_WRITE\_LOCK(&t->lock);
7. e = (list\_t \*) \_\_tbl\_find(t, id, crit);
9. **if** (!e) {
10. DSR\_WRITE\_UNLOCK(&t->lock);
11. **return** NULL;
12. }
13. list\_del(e);
14. t->len--;
16. DSR\_WRITE\_UNLOCK(&t->lock);
18. **return** e;
19. }

tbl.h

## 4.4路由维护处理

### 4.4.1 概述

路由维护是这样一种机制: 当源节点S使用到某个目标节点D的源路由时，如果网络拓扑发生了变化，由于路由上的链接不再工作，它就不能再使用到D的路由，则源节点S可以通过这种机制进行检测。当路由维护表明源路由不再工作, S可以尝试使用任何其他恰好知道D的路由, 或可以调用路由发现再次找到一个新的后续数据包路由到D。路由维护只有当S发送数据包到D时被使用。

在转发包时，节点必须尝试确认下一跳节点的可达性，除非在上一个MaintHoldoffTime中已经接收到这种确认。如果在MaxMaintRexmt确认请求重传之后，在数据包初始传输之后，并且在概念上包括MAC层提供的所有重传之后，没有收到确认，则节点确定源路由下一跳节点的链接“断开”。来自下一跳节点的路由维护确认可以使用链路层确认、被动确认或网络层确认

### 4.4.2 maint\_buf\_salvage 函数

维护缓冲区为路由维护提供了完成操作的基础. 当一个节点收到维护缓冲区中任何包的确认时,该节点会从维护缓冲区中删除该包以及维护缓冲区中具有相同下一跳目的地的任何其他包.

maint\_buf\_salvage 函数便实现了上述操作.

maint-buf.c

1. **int** NSCLASS maint\_buf\_salvage(**struct** dsr\_pkt \*dp)
2. {
3. **struct** dsr\_srt \*alt\_srt, \*old\_srt, \*srt;
4. **int** old\_srt\_opt\_len, new\_srt\_opt\_len, sleft, salv;
6. **if** (!dp)
7. **return** -1;
9. **if** (dp->srt) {
10. DEBUG("old internal source route exists\n");
11. FREE(dp->srt);
12. }
14. alt\_srt = dsr\_rtc\_find(my\_addr(), dp->dst);
16. **if** (!alt\_srt) {
17. DEBUG("No alt. source route - cannot salvage packet\n");
18. **return** -1;
19. }
21. **if** (!dp->srt\_opt) {
22. DEBUG("No old source route\n");
23. FREE(alt\_srt);
24. **return** -1;
25. }
27. old\_srt = dsr\_srt\_new(dp->src, dp->dst, dp->srt\_opt->length - 2,
28. (**char** \*)dp->srt\_opt->addrs);
30. **if** (!old\_srt) {
31. FREE(alt\_srt);
32. **return** -1;
33. }
35. DEBUG("opt\_len old srt: %s\n", print\_srt(old\_srt));

maint-buf.c

191 - 192声明了几个源路由结构的指针, 包括alt\_srt为要更改的源路由, old\_srt为之前的源路由, 以及源路由选项的长度.

194 – 214 排除异常情况,包括目标包本身不存在, 包源路由选项不存在等异常情况.

215 - 221创建一个以dp中的源节点作为源节点, dp中目的节点作为目的节点的源路由作为old\_srt, 以便进行后续的更改.

218 - 223排除不存在上述源路由的异常情况的主机名

maint-buf.c

1. **if** (old\_srt->addrs[0].s\_addr == dp->nxt\_hop.s\_addr) {
2. srt = alt\_srt;
3. sleft = (srt->laddrs) / 4;
4. } **else** {
5. **struct** dsr\_srt \*srt\_to\_me;
7. srt\_to\_me = dsr\_srt\_new\_split(old\_srt, my\_addr());
9. **if** (!srt\_to\_me) {
10. FREE(alt\_srt);
11. FREE(old\_srt);
12. **return** -1;
13. }
14. srt = dsr\_srt\_concatenate(srt\_to\_me, alt\_srt);
16. sleft = (srt->laddrs) / 4 - (srt\_to\_me->laddrs / 4) - 1;
18. DEBUG("old\_srt: %s\n", print\_srt(old\_srt));
19. DEBUG("alt\_srt: %s\n", print\_srt(alt\_srt));
21. FREE(alt\_srt);
22. FREE(srt\_to\_me);
23. }
25. FREE(old\_srt);
27. **if** (!srt)
28. **return** -1;
30. DEBUG("Salvage packet sleft=%d srt: %s\n", sleft, print\_srt(srt));
32. **if** (dsr\_srt\_check\_duplicate(srt)) {
33. DEBUG("Duplicate address in new source route, aborting salvage\n");
34. FREE(srt);
35. **return** -1;
36. }
38. old\_srt\_opt\_len = dp->srt\_opt->length + 2;
39. new\_srt\_opt\_len = DSR\_SRT\_OPT\_LEN(srt);
40. salv = dp->srt\_opt->salv;
42. DEBUG("Salvage - source route length new=%d old=%d\n",
43. new\_srt\_opt\_len, old\_srt\_opt\_len);

maint-buf.c

225 - 227 实际上这些代码是在以判断old\_srt的首个中间地址是否与dp下一跳的地址是否相同的方式来判断old\_srt是否还有其源节点, 如果它本身就是一个源节点,那么便会满足判断条件的要求, 此时直接将要更改的alt\_srt赋值给srt

228 如果old\_srt不符合上述的判断条件,那么证明它还有另外的源节点.

229 – 231 当old\_srt还有另外的源节点时, 再次使用dsr\_srt\_new\_split函数获得其源节点, 赋值给srt\_to\_me.

233 – 237 排除上述源节点不存在的异常情况

238 – 246 使用dsr\_srt\_concatenate函数将两源路由连接, 结果便是被赋值后的源路由.

251 – 260 排除srt不存在和重复等异常情况.

262 – 264 更新路由选项的长度.

maint-buf.c

1. **if** (old\_srt\_opt\_len == new\_srt\_opt\_len) {
2. DEBUG("new and old srt of same length\n");
4. dp->srt\_opt = dsr\_srt\_opt\_add((**char** \*)dp->srt\_opt,
5. new\_srt\_opt\_len, 0,
6. salv + 1, srt);
7. } **else** {
8. **int** old\_opt\_len, new\_opt\_len;
9. **char** \*old\_opt = dp->dh.raw;
10. **char** \*old\_srt\_opt = (**char** \*)dp->srt\_opt;
11. **char** \*buf;
13. DEBUG("Creating new options header\n");
15. old\_opt\_len = dsr\_pkt\_opts\_len(dp);
16. new\_opt\_len = old\_opt\_len - old\_srt\_opt\_len + new\_srt\_opt\_len;
18. DEBUG("opt\_len old=%d new=%d srt: %s\n",
19. old\_opt\_len, new\_opt\_len, print\_srt(srt));
21. buf = dsr\_pkt\_alloc\_opts(dp, new\_opt\_len);
23. **if** (!buf) {
24. FREE(srt);
25. **return** -1;
26. }
28. memcpy(buf, old\_opt, old\_srt\_opt - old\_opt);
30. buf += (old\_srt\_opt - old\_opt);
32. dp->srt\_opt = dsr\_srt\_opt\_add(buf, new\_srt\_opt\_len, 0,
33. salv + 1, srt);
35. buf += new\_srt\_opt\_len;
37. memcpy(buf, old\_srt\_opt + old\_srt\_opt\_len,
38. old\_opt + old\_opt\_len -
39. (old\_srt\_opt + old\_srt\_opt\_len));
41. FREE(old\_opt);
43. dp->dh.opth->p\_len = htons(new\_opt\_len - DSR\_OPT\_HDR\_LEN);
44. }
46. dp->srt\_opt->sleft = sleft;
48. dp->nxt\_hop = dsr\_srt\_next\_hop(srt, dp->srt\_opt->sleft);
50. DEBUG("Next hop=%s p\_len=%d\n", print\_ip(dp->nxt\_hop), ntohs(dp->dh.opth->p\_len));
52. dp->srt = srt;
54. XMIT(dp);
56. **return** 0;
57. }

maint-buf.c

269 – 274 如果新旧源路由选项的长度相同，那么只需要将salv回收次数加1，将新的路由选项添加到路由选项中。

275 – 279 如果新旧源路由选项的长度不同，那么添加的路由选项的长度则需要修改。这时创建几个变量，包括新旧选项长度等。

283 – 284 使用dsr\_pkt\_opts\_len 函数获取dp的包选项长度，作为包旧长度，然后加上新旧源路由选项长度作为包新长度。

289 以新长度申请新选项的空间。

296 – 298 将所有的源路由选项内容复制到申请的空间buf中。

300 – 307 添加新的源路由选项。

1. – 320 更新源路由选项，下一跳等信息。

### 4.4.3 dsr\_rerr\_send 函数

当节点达到最大重传次数后无法验证下一跳节点的可达性时，节点应将路由错误发送到包的IP源地址。

如图4.x，当节点J和D之间的连接出现问题时，J会向源节点发送RERR包报告错误。源节点收到RERR包，将J和D之间的连接信息移出路由缓存，重新使用路由发现建立到节点D的路由。

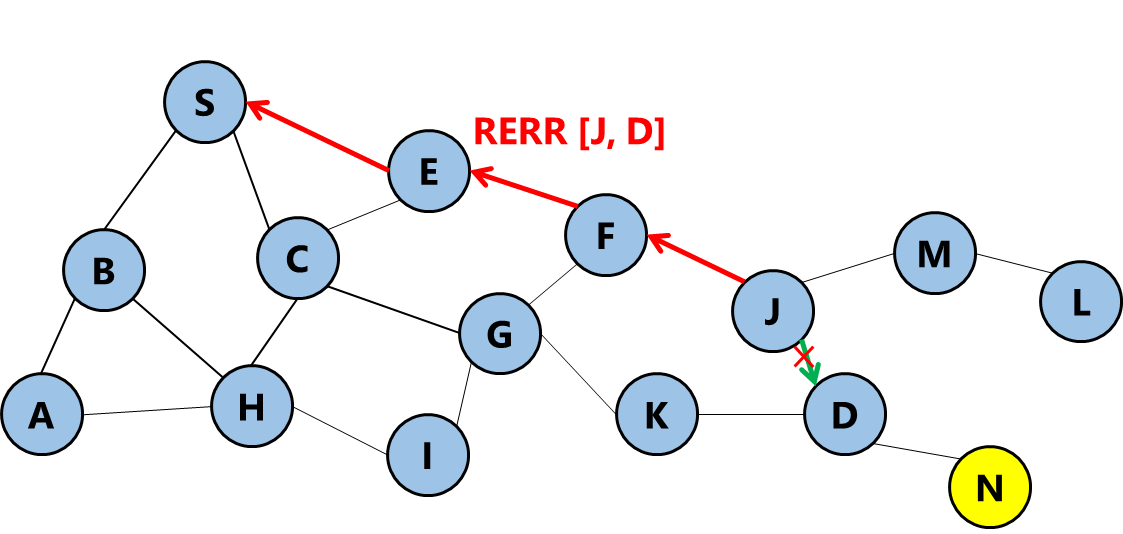


图4.1 节点发送rerr包

而dsr\_err\_send 函数便实现了发送rerr包的功能。

Dsr-rerr.c

1. **int** NSCLASS dsr\_rerr\_send(**struct** dsr\_pkt \*dp\_trigg, **struct** in\_addr unr\_addr)
2. {
3. **struct** dsr\_pkt \*dp;
4. **struct** dsr\_rerr\_opt \*rerr\_opt;
5. **struct** in\_addr dst, err\_src, err\_dst, myaddr;
6. **char** \*buf;
7. **int** n, len, i;
9. myaddr = my\_addr();
11. **if** (!dp\_trigg || dp\_trigg->src.s\_addr == myaddr.s\_addr)
12. **return** -1;
14. **if** (!dp\_trigg->srt\_opt) {
15. DEBUG("Could not find source route option\n");
16. **return** -1;
17. }
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];
24. dp = dsr\_pkt\_alloc(NULL);
26. **if** (!dp) {
27. DEBUG("Could not allocate DSR packet\n");
28. **return** -1;
29. }
31. dp->srt = dsr\_rtc\_find(myaddr, dst);
33. **if** (!dp->srt) {
34. DEBUG("No source route to %s\n", print\_ip(dst));
35. **return** -1;
36. }
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;
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\_SALVAGE\_COUNT))
45. **break**;
47. len += (dp\_trigg->rerr\_opt[i]->length + 2);
48. }
50. DEBUG("opt\_len=%d SR: %s\n", len, print\_srt(dp->srt));

Dsr-rerr.c

65 – 69 声明了一些局部变量，包括DSR包dp，rerr选项，rerr包的源、目的地址等。

71 – 79 使用my\_addr 函数获取了dsr节点的地址，并排除了dp\_trigg不存在或没有对应源路由选项的异常情况。

81 – 84 确定rerr包的目的节点。如果触发异常的节点的源路由选项包没有被回收，那么目的节点就是其源节点；否则通过包的中间地址确定目的地址。

86 – 91 为dp 包分配空间，并排除其不存在的异常情况。

93 – 102 通过当前dsr 地址确定dp的源路由，并更新首部长度。

105 – 112 对所有的路由错误包选项进行检查操作，判断其是否超过最大回收数，对符合条件的继续增加首部长度

Dsr-rerr.c

1. n = dp->srt->laddrs / **sizeof**(**struct** in\_addr);
2. dp->src = myaddr;
3. dp->dst = dst;
4. dp->nxt\_hop = dsr\_srt\_next\_hop(dp->srt, n);
6. dp->nh.iph = dsr\_build\_ip(dp, dp->src, dp->dst, IP\_HDR\_LEN,
7. IP\_HDR\_LEN + len, IPPROTO\_DSR, IPDEFTTL);
9. **if** (!dp->nh.iph) {
10. DEBUG("Could not create IP header\n");
11. **goto** out\_err;
12. }
14. buf = dsr\_pkt\_alloc\_opts(dp, len);
16. **if** (!buf)
17. **goto** out\_err;
19. dp->dh.opth = dsr\_opt\_hdr\_add(buf, len, DSR\_NO\_NEXT\_HDR\_TYPE);
21. **if** (!dp->dh.opth) {
22. DEBUG("Could not create DSR options header\n");
23. **goto** out\_err;
24. }
26. buf += DSR\_OPT\_HDR\_LEN;
27. len -= DSR\_OPT\_HDR\_LEN;
29. dp->srt\_opt = dsr\_srt\_opt\_add(buf, len, 0, 0, dp->srt);
31. **if** (!dp->srt\_opt) {
32. DEBUG("Could not create Source Route option header\n");
33. **goto** out\_err;
34. }
36. buf += DSR\_SRT\_OPT\_LEN(dp->srt);
37. len -= DSR\_SRT\_OPT\_LEN(dp->srt);
39. rerr\_opt = dsr\_rerr\_opt\_add(buf, len, NODE\_UNREACHABLE, dp->src,
40. dp->dst, unr\_addr,
41. dp\_trigg->srt\_opt->salv);
43. **if** (!rerr\_opt)
44. **goto** out\_err;
46. buf += (rerr\_opt->length + 2);
47. len -= (rerr\_opt->length + 2);
49. /\* Add old RERR options \*/
50. **for** (i = 0; i < dp\_trigg->num\_rerr\_opts; i++) {
52. **if** (dp\_trigg->rerr\_opt[i]->salv > ConfVal(MAX\_SALVAGE\_COUNT))
53. **break**;
55. memcpy(buf, dp\_trigg->rerr\_opt[i],
56. dp\_trigg->rerr\_opt[i]->length + 2);
58. len -= (dp\_trigg->rerr\_opt[i]->length + 2);
59. buf += (dp\_trigg->rerr\_opt[i]->length + 2);
60. }

Dsr-rerr.c

113 – 124 更新dp节点的信息，包括源、目的地址，下一跳。以及创建ip首部信息并排除异常情况。

126 – 129 为dp节点选项分配空间，暂时储存在buf中，并排除异常情况。

131 – 136 创建dp节点的操作首部并排除异常。

138 – 139 上一步操作结束之后，增加buf也即dp节点选项的空间，并减小len首部长度。

141 – 146 创建源路由的选项，并排除异常情况。

148 – 149 与上一步相似，增加buf也即dp节点选项的空间，并减小len首部长度。

之后对rerr选项也进行了相似的操作。

# 第五章　总结

在无线网络的应用越来越广泛的今天，DSR路由协议为我们提供了一种效率高、开销低的，便于管理的路由协议。由于其能够再网络拓扑或传输条件产生变化时，自动确定和维护路由，因此该路由协议对网络条件不断变化的实际情况适用性较强。

相比于其他的路由协议，DSR的路由发现可以发现多条到目的节点的路由，使用路由缓存也可以进一步减少路由发现的开销；路由维护时只有节点之间需要沟通，以此减少路由维护开销。

同时我们也认识到，DSR路由协议也有一些问题存在，比如路由的包的首部大小会随着路由长度的增加而增加，以及路由请求中会产生相邻节点之间的碰撞以及洪泛滥等问题。这些问题都需要我们留意去规避。

本次大作业我们的主要成果在于：

* 阅读了DSR动态源路由算法的相关文档，了解了DSR算法的工作原理。
* 对DSR算法的数据结构进行了深入解析，对DSR算法在实际运行时应用的数据及其发生的变化有了深入的了解。
* 对DSR算法实现主要功能的函数进行了细致的分析，对DSR算法的工作过程有了深入的理解。

作为网络工程专业的学生，我们应该对更多的路由算法有着更深入的研究。希望DSR及其他网络能够不断发展，让网络技术更好地被人类所应用。