[1. 概述](#header-n3)  
 [1.1 协议概述](#header-n4)  
 [1.2 主要算法描述](#header-n7)  
 [1.2.1 部分元组结构](#header-n8)  
 [1.2.2链路感知](#header-n14)  
 [1.2.3 邻居检测](#header-n36)  
 [1. 扩充邻居集](#header-n38)  
 [2. 扩充二跳邻居集](#header-n72)  
 [1.2.4 MPR](#header-n91)  
 [1. MPR计算](#header-n96)  
 [2. MPR selector扩充计算](#header-n114)  
 [3. 邻居和二跳邻居变化](#header-n132)  
 [1.2.5 拓扑发现](#header-n149)  
 [1. 广播邻居集](#header-n152)  
 [2. TC分组生成](#header-n155)  
 [3. TC分组转发](#header-n160)  
 [4. TC分组处理](#header-n162)  
 [1.2.6 路由表计算](#header-n193)  
 [1.3 OLSR的优点和局限性](#header-n245)  
[2.代码分析](#header-n255)  
 [2.1 文件介绍](#header-n256)  
 [2.2 数据结构](#header-n259)  
 [2.2.1 OLSR头部(省略IP和UDP报头)](#header-n260)  
 [2.2.2 HELLO分组](#header-n278)  
 [2.2.3 TC分组](#header-n312)  
 [2.2.4 节点存储表项](#header-n322)  
 [2.3 链路感知和邻居检测](#header-n377)  
 [2.3.1 HELLO分组生成](#header-n379)  
 [2.3.2 节点操作](#header-n403)  
 [2.3.3 操作邻居表](#header-n413)  
 [2.4 MPR](#header-n442)  
 [2.4.1 MPR生成](#header-n444)  
 [2.4.2 MPR选择](#header-n585)  
 [2.5 拓扑发现](#header-n465)  
 [2.5.1 TC分组生成](#header-n467)  
 [2.5.2 拓扑信息集的初始化](#header-n472)  
 [2.5.3 TC分组处理](#header-n479)  
 [2.5.4 拓扑信息集的删除](#header-n497)  
 [2.6 路由表计算](#header-n506)  
 [2.6.1 相关结构体](#header-n507)  
 [2.6.2 路由表计算](#header-n529)

# 1. 概述

## 1.1 协议概述

## 1.2 主要算法描述

### 1.2.1 部分元组结构

**链接元组 *(L*local*iface*addr, L*neighbor*iface*addr, L*SYM*time, L*ASYM*time, Ltime)***

**邻居元组 *(N*neighbor*main*addr, N*2hop*addr,N*time)***

**MPR选择元组 *(MS*main*addr, MS*time)\_**

**拓扑元组 *{T*test*addr,T*lase*addr, T*seq, T*time}***

**路由表项元组{R*dest*addr, R*next*addr, R*dist, R*iface\_addr}**

### 1.2.2链路感知

L*neighbor*iface\_addr == 该分组的Originator Address

必须加入在链接信息库中新加入：

L*beighbor*iface\_addr = 该分组的Originator Address>

L*local*iface\_addr = 该分组的接口地址

L*SYM*time = Current time - 1

L\_time =current time + validity time

* 如果该元组已经存在,则按照如下规则修改链路信息集：

L*ASYM*time = current time + validity time

如果接口地址在HELLO分组链路信息列表中，则如下更新：

\* 如果链接状态是LOST\_LINK,则：

L*SYM*time = current time + validity time

* 如果链接状态是SYM*LINK或者ASYM*LINK,则：

L*SYM*time = current time + validity time

L*time = L*SYM*time+NEIGHB*HELLO\_TIME

L*time = max(L*time, L*ASYM*time)

### 1.2.3 邻居检测

#### 1. 扩充邻居集

* 如果一个链接元组中的关联邻居元组存在，邻居元组为：
* N*neighbor*main*addr ==L*neightbor\_iface \_addr的主地址
* 如果邻居元组的关联链接元组都是链接元组，并且：
* N*neighbor*main*addr == L*neighbot\_iface \_addr的主地址
* 则邻居集必须通过维护链接元组和关联邻居元组之间对应关系来填充，填充规则如下：
  + 创建
  + 每次出现一个链接时，必须创建关联的邻居元组，如果邻居元组不存在，则：
  + N*neighbor*main*addr = Lneighbor*iface\_addr 的主地址
  + 更新
  + 每次链接信息更改时，节点必须确保相关邻居元组的N\_status遵循：
  + 如果邻居有任何关联元组代表一个对称链接，则：
  + N\_status = SYM
  + 否则：
  + N*status = NOT*SYM
  + 删除
  + 每次删除一个链接时，相关的链接元组必须被删除，相关的邻居元组也必须被删除。
* 这些规则都是为了确保存在一个唯一的相关邻居元组对应一个链接元组，并且每个邻居元组至少对应一个关联元组。
* HELLO分组处理
* 如果发出者地址是邻居集中邻居元组中的N*neighbor*main\_addr，则邻居元组应该更新：
* N\_willingness = HELLO分组中的willingness

#### 2. 扩充二跳邻居集

* 对于每一个在HELLO中分组中且邻居类型为SYM*NEIGH或MPR*NEIGH的地址：
  + 如果一个二跳邻居节点的主地址不等于接受节点的主地址，则丢弃该二跳邻居。
  + 否则，新创建一个二跳邻居元组：
  + N*neighbor*main\_addr = 发出者地址
  + N*2hop*addr = 二跳邻居的主地址
  + N\_time = current time+validity time
  + 这个新分组用于替换和其具有相同N*neighbor*main*addr 和N*2hop\_addr值的元组。
* 对于每一个在HELLO中分组中且邻居类型为NOT\_NEIGH的地址，所有的二跳邻居应该是：
* N*neighbor*main\_addr == 发出者地址 and
* N*2hop*addr == 二跳邻居的主地址

### 1.2.4 MPR

#### 1. MPR计算

* 从一个节点的子集开始，当N*willness等于WILL*ALWAYS该子集中成员属于一个MPR集。
* 计算该子集中所有节点的一跳邻居集。
* 添加节点，当且仅当该节点对另一子集中节点可达时。例如另一个子集中的b节点通过对称链路可访问到节点该子集中的a节点时，向MPR集中添加a节点，同时去除另一子集中的节点。
* 当另一子集中存在未被当前MPR中节点覆盖节点时:
  + 计算当前子集中的每一个节点的可达性。即另一子集中尚未被当前子集中的节点覆盖，并且通过一跳邻居能到达的节点。
  + 在当前子集中选择一个拥有最大N\_willingness，且可达节点不为0的节点。在这种情况下，选择的几点应该具有对另一子集节点的最大可达性。(即拥有对另一子集可达节点最多)在这种当多个节点拥有相同最大可达性的情况下，选择属于MPR集，且拥有最大一跳邻居集的节点。然后删除被该节点覆盖掉的另一子集中的节点。
* 从已有的MPR集中生成每个节点的MPR集。
* 对于上述算法，还可以有部分优化。

#### 2. MPR selector扩充计算

节点的MPR selector集通过其他将同一子集选为初始MPR集的节点的主地址来扩充。MPR选择过程通过HELLO消息的交互来实现。

* HELLO消息处理
* 一个节点接受一个HELLO分组后，如果发现本节点的其中一个接口地址在邻居类型等于MPR\_NEIGH的列表中时，该HELLO分组携带的信息必须被重新记录在MPR Selector集中。
* Validity time必须通过HELLO分组的Vtime字段重新计算。MPR selector集更新规则如下：
  1. 如果不存在具有以下内容MPR selector元组：
  + MS*main*addr == 发出者地址
  + 则创建一个新元组：
  + MS*main*addr = 发出者地址
  1. 如果元组：
  + MS*main*addr == 发出者地址
  + 则修改MS\_time字段:
  + MS\_time = current time + validity time

#### 3. 邻居和二跳邻居变化

邻居节点的变化情况将在以下情况下被检测到:

* 如果链接元组的L*SYM*Time字段到期。
* 在链接集中插入一个新的链接元组，其中L*SYM*Time字段未过期，或者L*SYM*Time被修改避免过期。

当邻居集或者二跳邻居集被检测到发生变化，则执行以下流程：

* 如果有邻居丢失，则所有N*neighbor*main\_addr==Main Address的二跳元组必须被删除。
* 如果有邻居丢失，所有MS*main*addr == Main Address的MPR selector元组必须被删除。
* 当有邻居或者二跳邻居改变，丢失，或者删除时，MPR集必须重新计算。
* 当MPR集发生改变时，可能发出一个额外的HELLO分组。

### 1.2.5 拓扑发现

#### 1. 广播邻居集

#### 2. TC分组生成

#### 3. TC分组转发

#### 4. TC分组处理

* 当接受到一个TC分组后，必须先通过其头部的Vtime字段计算其有效时间然后，拓扑集根据以下规则更新：
* 如果此分组的发送者接口不在本节点的一跳对称邻居中，则丢弃该分组。
* 如果拓扑信息集中存在一些元组：
* T*last*addr == 发出者地址 AND
* T\_seq > ANSN
* 则不对该分组做任何进一步处理，并且丢弃该分组。
* 如果拓扑集中所有的元组都是:
* T*last*addr == 发出者地址 AND
* T\_seq < ANSN
* 则必须从拓扑集中删除。
* 对于每一个在TC分组中受到的被广播的邻居的主地址：
  + 如果在拓扑集中存在一些元组：
  + T*test*addr == 广播邻居的主地址 AND
  + T*last*addr == 发出者地址
  + 则该元组的有效时间必须被设为：
  + T\_time = current time + validity time
  + 否则，必须在拓扑集中记录一条新的原则记录:
  + T*dest*addr = 被广播邻居主地址
  + T*last*addr = 发出者地址
  + T\_seq = ANSN
  + T\_time = current time + validity time

### 1.2.6 路由表计算

* 当以下任何一个有变化时，都需要更新路由表:
  + 链路集
  + 邻居集
  + 拓扑集
  + 二条邻居集
  + 多接口关联信息集
* 路由表的更新既不在网络中，也不在一跳邻居中生成或触发任何消息。

路由表计算方法如下：

* 删除路由表中所有条目
* 新增加路由表项从以对称邻居作为目的节点开始。因此，对于邻居集中的每个邻居元组，都有:
* N\_status = SYM
* 并且，对于邻居节点的每个关联链接元组，都有L\_time >= current time,路由表中新加的表项为:
* R*dest*addr = 关联链接元组的L*neighbor*iface\_addr
* R*next*addr = 关联链接元组的L*neighbor*iface\_addr
* R\_dist = 1
* R*iface*addr = 关联链接元组的L*local*iface\_addr
* 如果按照上述情况，没有R*dest*addr等于邻居节点的主地址，则必须添加新的路由表项:
* R*dest*addr = 邻居的Main address
* R*next*addr = L*time>0的关联链接元组的L*neighbor*iface*addr
* R\_dist = 1
* R*iface*addr = 关联链接元组的L*local*iface
* 对于严格二条邻居节点，如果二跳邻居集中至少存在一项记录中N*neighbor*main*addr对于一个willingness不为WILL*NEVER的节点，选择一个二跳邻居节点，并在路由表中添加新表项：
* R*dest*addr = 二跳邻居的Main Address
* R*next*addr = 路由表项中R*dest*addr等于二跳邻居N*neighbor*main*addr的记录的R*next\_addr
* R\_dist = 2
* R*ifacce*addr = 路由表中R*dest*addr等于二跳元组中N*neighbor*main-addr的记录的R*iface*addr
* 对于h+1跳的节点，按照Dijkstra算法加入路由表中
* 对于多接口关联信息集中的每个实体，如果不存在一个路由表项：
* R*dest*addr == I*main*addr
* 并且，也没有表项：
* R*dest*addr == I*iface*addr
* 则添加一个新表项：
* R*dext*addr = I*iface*addr
* R*dext*addr = R*next*addr
* R*dist = R*dist
* R*iface*addr = R*iface*addr

## 1.3 OLSR的优点和局限性

* OLSR协议是一种先应式路由协议，具有查找路由延时小的优点。
* OLSR用于移动自组织网络，MPR的实现极大的减小了网络中的控制流量，特别适用大型密集型移动网络，而且网络越大，效果越好。
* OLSR最初就是为以完全分布式方式工作，不依赖任何中心实体。
* OLSR不需要对IP数据包做任何修改。

# 2.代码分析

### 2.1 文件介绍

OLSR协议中共有123个源文件，以下对部分重要文件进行罗列。



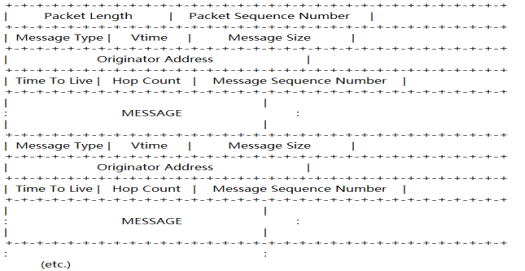
### 2.2 数据结构

#### 2.2.1 OLSR头部(省略IP和UDP报头)

55 struct olsr\_common {  
56 uint8\_t type;  
57 olsr\_reltime vtime;  
58 uint16\_t size;  
59 union olsr\_ip\_addr orig;  
60 uint8\_t ttl;  
61 uint8\_t hops;  
62 uint16\_t seqno;  
63 };

53-63：olsr\_common是OLSR协议的基本数据包首部。

对于与协议相关的所有数据，OLSR使用统一的数据包格式进行通信，这样做的目的是在不破坏向后兼容性的情况下促进协议的可扩展性。这也提供了一种简单的方法，将不同“类型”的信息汇集到一个单一的传输中。这些数据包嵌入在UDP数据报中，使用UDP通信，IANA将端口698分配给OLSR协议专用。每个分组封装一个或多个消息，这些消息共享一个通用的报头格式，使节点能够接受和重传未知类型的消息。OLSR协议分组的基本格式如图所示：



**a.**数据包首部

**Packet** **Length：**数据包的长度（单位：字节）

**Packet** **Sequence** **Number：**数据包序列号，每当传送一个新的OLSR分组时，序列号加一。为每个接口维护一个单独的分组序列号，以便对通过接口发送的分组进行顺序枚举。

**b.**消息首部

**Message** **Type** ：消息类型，在0-127范围内的消息类型保留给本协议中的所有消息和以后可能的扩展。

**Vtime** ：该字段表明接收节点后多长时间必须将消息中包含的信息视为有效，除非接收到对信息的最新更新。有效时间由它的尾数a（Vtime字段的四个最高位）和它的指数b（Vtime字段的四个最低位）表示。有效期=C*（1a/16）*2^b[秒]，C为一系数。

**Message Size**：消息的大小，以字节为单位计算。并从“消息类型”字段的开头到下一个“消息类型”字段的开头（如果没有下一个“消息类型”，则到该消息的结尾为止)。

**Originator Address**：此字段包含最初生成此消息的节点的主地址，重传过程中地址不发生改变。此字段不应与IP标头中的源地址混淆，后者每次都更改为重传该消息的中间接口的地址。

**Time To Live**：此字段包含将发送消息的最大跳数。消息每次重传前TTL减1，当TTL为0或1时，则不再重传。通过此方式可以限制一个消息的洪泛范围。

**Hop Count**：一个消息获得的跳数的数量。每次重传消息前，跳数加一。初始值为0。

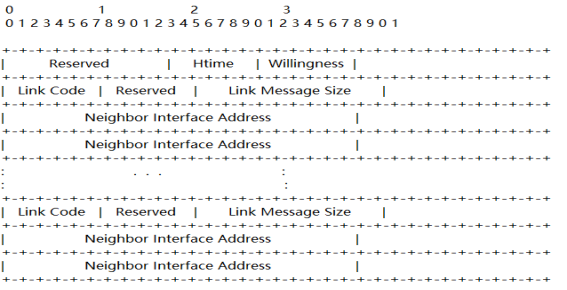
**Message Sequence Number**：由源节点产生的一个消息的唯一标识，用来保证一个消息不会被任何节点重传一次以上。

#### 2.2.2 HELLO分组

OLSR协议采用一种通用的机制来填充本地链路信息库和邻居信息库，即周期性地交换HELLO消息。

106 /\* serialized LQ\_HELLO \*/  
107 struct lq\_hello\_info\_header {  
108 uint8\_t link\_code;  
109 uint8\_t reserved;  
110 uint16\_t size;  
111 };  
112  
113 struct lq\_hello\_header {  
114 uint16\_t reserved;  
115 uint8\_t htime;  
116 uint8\_t will;  
117 };

106-117：结构体lq*hello*info*header和lq*hello\_header共同组成HELLO消息数据包的首部。格式如下图：

 这一部分是将OLSR基本数据包首部的"Message Type"设为HELLO*MESSAGE，TTL设为1，Vtime设为NEIGHB*HOLD\_TIME。

**Reserved**:保留字段,必须设置设为“0000000000000

**Htime**:指定节点在特定接口上的HELLO发射间隔，即下一个HELLO传输前的时间。Hello发射间隔用尾数(Htime的四个最高位)和指数(Htime的四个最低位)来表示。即HELLO发射间隔=C*（1+a/16）*2^b[秒]。其中a是Htime字段的四个最高位表示的整数，b是Htime字段的四个最低位表示的整数。

**willingness**:此字段指定节点为其他节点承载和转发流量的意愿。

Willingness有三种取值：

1. **WILL\_NEVER**：永远不会选择意愿为WILL\_NEVER的节点作为MPR。
2. **WILL\_ALWAYS**：具有WILL\_ALWAYS意愿的节点将始终被选择为MPR。
3. **WILL\_DEFAULT**：默认情况下，节点意愿为WILL\_DEFAULT。

**Link Code**:此字段指定发送方的接口和邻居列表中邻居接口之间的链路类型。它还指定有关邻居状态的信息。链路类型不为节点所知的邻居信息被静默丢弃:

链路类型有以下三种：

1. **ASYM\_LINK**:发送HELLO分组的节点与邻居列表中的节点间的链路是非对称的。表示可以收到邻居节点的消息,但不确定邻居节点能否收到本节点的消息。
2. **SYM\_LINK**:发送HELLO分组的节点与列表中的邻节点间的链路是对称的。表示链路已经被验证为双向的。
3. **MPR\_LINK**:表示列表中的节点已被发送该HELLO分组的节点选择为MPR。

**Link Message Size**：本链路消息的大小，以字节为单位，从Link Code字段开始到下一个Link Code字段之前（如果没有下一个Link Code，则到该消息尾部）。

**Neighbor Interface Address**：邻居节点的接口地址。每一种链路类型之后都紧跟一组邻居节点接口地址，表明该节点与这组邻居节点中的每一个节点链路类型都相同，都为前面给出的链路类型。

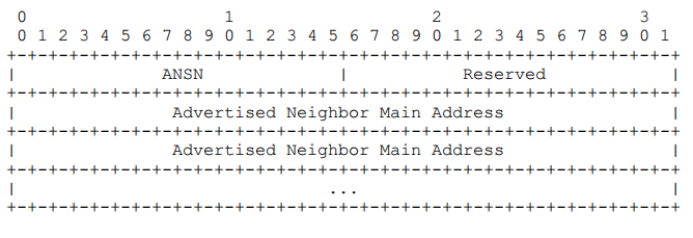
49 struct hello\_neighbor {  
50 uint8\_t status;  
51 uint8\_t link;  
52 union olsr\_ip\_addr main\_address;  
53 union olsr\_ip\_addr address;  
54 struct hello\_neighbor \*next;  
55 olsr\_linkcost cost;  
56 uint32\_t linkquality[0];  
57 };  
58  
59 struct hello\_message {  
60 olsr\_reltime vtime;  
61 olsr\_reltime htime;  
62 union olsr\_ip\_addr source\_addr;  
63 uint16\_t packet\_seq\_number;  
64 uint8\_t hop\_count;  
65 uint8\_t ttl;  
66 uint8\_t willingness;  
67 struct hello\_neighbor \*neighbors;  
68  
69 };

49-57：结构体hello*neighbor是HELLO消息邻居节点集。Status，记录邻居的状态；link指明链路类型；main*address是邻居的主地址；address是邻居的其他地址；cost，链路代价；linkquality，链路质量。

59-69：hello*message是消息数据包。vtime，消息有效时间；htime，HELLO消息的发射间隔；source*addr，发送消息的原地址；packet*seq*number，数据包的序列号；hop\_count，消息已经经历的跳数；ttl，数据包生命周期；willingness，节点进行转发的意愿；neighbors，下一个传递的邻居节点。

#### 2.2.3 TC分组

该部分是将OLSR首部中“MessageType”设置为TC*Message。TTL设置为255(最大值)，以便将消息传播到网络中Vtime相应地设置为Top*hold\_time的值。



**ANSN**：序列号与公布的邻居集相关。每当节点检测到其邻居集中的更改时，它都会递增这个序列号。当节点接收到TC消息时，它可以根据该消息来判定所接收到的信息是否比现有消息更新。

**Advertised Neighbor Main Address**：此字段包含邻居节点（此处的邻居节点仅为该节点的MPR Selector集中的节点）的主地址。

**Reserved**：此字段是保留的，必须设置为“0000000000000000”。

77 struct tc\_message {  
78 olsr\_reltime vtime;  
79 union olsr\_ip\_addr source\_addr;  
80 union olsr\_ip\_addr originator;  
81 uint16\_t packet\_seq\_number;  
82 uint8\_t hop\_count;  
83 uint8\_t ttl;  
84 uint16\_t ansn;  
85 struct tc\_mpr\_addr \*multipoint\_relay\_selector\_address;  
86 };

77-86：tc\_message是TC消息数据包格式。OLSR协议利用TC拓扑表记录接收到的TC消息内容。TC拓扑表的介绍见“节点存储的表项”的“TC拓扑表”。

#### 2.2.4 节点存储表项

1. 接口关联集

* 网络中的每一个目的地节点存储了接口关联多元组。

|  |  |  |
| --- | --- | --- |
| I\_iface\_addr | I\_main\_addr | I\_time |

***I\_iface\_*addr**是节点的接口地址；

* ***I\_main\_*addr**是该节点的主地址；
* **I\_time**指定此元组过期的时间，以及必须删除的时间。

1. 本地链路信息表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| L\_local\_iface\_addr | L\_neighbor\_iface\_addr | L\_SYM\_time | L\_ASYM\_time | L\_tiem |

**L\_*local*iface\_addr**是本地节点(即链路的一个端点)的接口地址；

**L\_*neighbor*iface\_addr**是相邻节点的接口地址；

**L\_*SYM\_*time**是链路被视为对称的时间；

**L\_*ASYM\_*time**是被认为听到邻居接口的时间；

**L\_Time**指定了此记录过期的时间，并且必须被删除。当L*SYM*TIME和L*ASYM*TIME过期时，该链接被视为丢失。

1. 一跳邻居表

|  |  |  |
| --- | --- | --- |
| N\_neighbor\_main\_addr | N\_status | N\_willingness |

**N\_*neighbor\_*main\_addr**：邻居的主地址；

**N\_status**：指定节点是NOT\_SYM还是SYM；

**N\_willingness**：指定节点的携带意愿，取值为0到7之间的整数。

1. 二跳邻居表

|  |  |  |
| --- | --- | --- |
| N\_neighbor\_main\_addr | N2\_hop\_addr | N\_time |

**N\_*neighbor\_*main\_addr**是邻居的主地址；

**N*2\_hop\_*addr**是具有与N*neighbor*main*addr对称链接的2跳邻居的主地址，也就是说节点通过邻居节点N*neighbor*main*addr到达其二跳邻居节点N*2hop*addr；

**N\_time**指定元组过期和必须删除的时间。

1. MPR表

* 一个节点维护着其被选为MPR的邻居节点的集合，这些被选为MPR的邻居节点的主地址存放在MPR集中。

1. MPR selector表

|  |  |
| --- | --- |
| MS\_main\_addr | MS\_time |

* **MS*main*addr**是节点的主地址，该节点选择本节点为MPR；
* **MS\_time**指定元组过期的时间和必须删除的时间。

1. TC拓扑表

* 网络中的每个节点维护有关网络的拓扑信息。此信息从TC消息中获取，并用于路由表计算。

|  |  |  |  |
| --- | --- | --- | --- |
| T\_dest\_addr | T\_lase\_addr | T\_seq | T\_time |

* **T\_*dest\_*addr**目的地节点的主地址，它可以从地址为T*last*addr的节点经过一跳到达；
* **T\_*last\_*addr**是T*dest*addr的MPR；
* **T\_seq**是一个序列号；
* **T\_time**指定此元组过期的时间，以及必须删除的时间。

1. 路由表

|  |  |  |  |
| --- | --- | --- | --- |
| R\_dest\_addr | R\_next\_addr | R\_dist | R\_iface\_addr |

* **R\_*dest\_*addr**目的地节点地址
* **R\_*next\_*addr**下一跳节点地址
* **R\_dist**本节点到目的节点的距离
* **R\_*iface\_*addr**该节点转发路由信息的接口

### 2.3 链路感知和邻居检测

链路感知的机制是HELLO消息的周期性交换。节点必须在每个接口上执行链路感知，以便检测接口和邻居接口之间的链路。因此，对于给定的接口，HELLO消息将包含该接口上的链路列表，以及整个邻居的列表。

#### 2.3.1 HELLO分组生成

原则上，HELLO消息服务于三个独立的任务：链路感知，邻居检测和MPR选择。

66 void  
67 generate\_hello(void \*p)  
68 {  
69 struct hello\_message hellopacket;  
70 struct interface \*ifn = (struct interface \*)p;  
71  
72 olsr\_build\_hello\_packet(&hellopacket, ifn);  
73  
74 if (queue\_hello(&hellopacket, ifn))  
75 net\_output(ifn);  
76  
77 olsr\_free\_hello\_packet(&hellopacket);  
78  
79 }

66-79：generate*hello()函数用来产生hello消息包。调用olsr*build*hello*packet()函数为指定的接口生成要发送的HELLO数据包。如果创建成功，则通过net\_output()将该HELLO数据包通过指定的接口发送出去。最后释放该HELLO消息。

88 int  
89 olsr\_build\_hello\_packet(struct hello\_message \*message, struct interface \*outif)  
90 {  
91 struct hello\_neighbor \*message\_neighbor, \*tmp\_neigh;  
92 struct link\_entry \*links;  
93 struct neighbor\_entry \*neighbor;  
94   
95 #ifdef DEBUG  
96 OLSR\_PRINTF(3, "\tBuilding HELLO on interface \"%s\"\n", outif->int\_name ? outif->int\_name : "<null>");  
97 #endif  
98   
99 message->neighbors = NULL;  
100 message->packet\_seq\_number = 0;  
101  
102 //message->mpr\_seq\_number=neighbortable.neighbor\_mpr\_seq;  
103  
104 /\* Set willingness \*/  
105  
106 message->willingness = olsr\_cnf->willingness;  
107 #ifdef DEBUG  
108 OLSR\_PRINTF(3, "Willingness: %d\n", olsr\_cnf->willingness);  
109 #endif  
110  
111 /\* Set TTL \*/  
112  
113 message->ttl = 1;  
114 message->source\_addr = olsr\_cnf->main\_addr;  
115  
116 #ifdef DEBUG  
117 OLSR\_PRINTF(5, "On link:\n");  
118 #endif  
119  
120 /\* Walk all links of this interface \*/  
121 OLSR\_FOR\_ALL\_LINK\_ENTRIES(links) {  
122 #ifdef DEBUG  
123 struct ipaddr\_str buf;  
124 #endif  
125 int lnk = lookup\_link\_status(links);  
126 /\* Update the status \*/  
127  
128 /\* Check if this link tuple is registered on the outgoing interface \*/  
129 if (!ipequal(&links->local\_iface\_addr, &outif->ip\_addr)) {  
130 continue;  
131 }  
132  
133 message\_neighbor = olsr\_malloc\_hello\_neighbor("Build HELLO");  
134  
135 /\* Find the link status \*/  
136 message\_neighbor->link = lnk;  
137  
138 /\*  
139 \* Calculate neighbor status  
140 \*/  
141 /\*  
142 \* 2.1 If the main address, corresponding to  
143 \* L\_neighbor\_iface\_addr, is included in the MPR set:  
144 \*  
145 \* Neighbor Type = MPR\_NEIGH  
146 \*/  
147 if (links->neighbor->is\_mpr) {  
148 message\_neighbor->status = MPR\_NEIGH;  
149 }  
150 /\*  
151 \* 2.2 Otherwise, if the main address, corresponding to  
152 \* L\_neighbor\_iface\_addr, is included in the neighbor set:  
153 \*/  
154  
155 /\* NOTE:  
156 \* It is garanteed to be included when come this far  
157 \* due to the extentions made in the link sensing  
158 \* regarding main addresses.  
159 \*/  
160 else {  
161  
162 /\*  
163 \* 2.2.1  
164 \* if N\_status == SYM  
165 \*  
166 \* Neighbor Type = SYM\_NEIGH  
167 \*/  
168 if (links->neighbor->status == SYM) {  
169 message\_neighbor->status = SYM\_NEIGH;  
170 }  
171  
172 /\*  
173 \* 2.2.2  
174 \* Otherwise, if N\_status == NOT\_SYM  
175 \* Neighbor Type = NOT\_NEIGH  
176 \*/  
177 else if (links->neighbor->status == NOT\_SYM) {  
178 message\_neighbor->status = NOT\_NEIGH;  
179 }  
180 }  
181  
182 /\* Set the remote interface address \*/  
183 message\_neighbor->address = links->neighbor\_iface\_addr;  
184  
185 /\* Set the main address \*/  
186 message\_neighbor->main\_address = links->neighbor->neighbor\_main\_addr;  
187 #ifdef DEBUG  
188 OLSR\_PRINTF(5, "Added: %s - status %d\n", olsr\_ip\_to\_string(&buf, &message\_neighbor->address), message\_neighbor->status);  
189 #endif  
190 message\_neighbor->next = message->neighbors;  
191 message->neighbors = message\_neighbor;  
192  
193 }  
194 OLSR\_FOR\_ALL\_LINK\_ENTRIES\_END(links);  
195  
196 /\* Add the links \*/  
197  
198 #ifdef DEBUG  
199 OLSR\_PRINTF(5, "Not on link:\n");  
200 #endif  
201  
202 /\* Add the rest of the neighbors if running on multiple interfaces \*/  
203  
204 if (ifnet != NULL && ifnet->int\_next != NULL)  
205 OLSR\_FOR\_ALL\_NBR\_ENTRIES(neighbor) {  
206  
207 #ifdef DEBUG  
208 struct ipaddr\_str buf;  
209 #endif  
210 /\* Check that the neighbor is not added yet \*/  
211 tmp\_neigh = message->neighbors;  
212 //printf("Checking that the neighbor is not yet added\n");  
213 while (tmp\_neigh) {  
214 if (ipequal(&tmp\_neigh->main\_address, &neighbor->neighbor\_main\_addr)) {  
215 //printf("Not adding duplicate neighbor %s\n", olsr\_ip\_to\_string(&neighbor->neighbor\_main\_addr));  
216 break;  
217 }  
218 tmp\_neigh = tmp\_neigh->next;  
219 }  
220  
221 if (tmp\_neigh) {  
222 continue;  
223 }  
224  
225 message\_neighbor = olsr\_malloc\_hello\_neighbor("Build HELLO 2");  
226  
227 message\_neighbor->link = UNSPEC\_LINK;  
228  
229 /\*  
230 \* Calculate neighbor status  
231 \*/  
232 /\*  
233 \* 2.1 If the main address, corresponding to  
234 \* L\_neighbor\_iface\_addr, is included in the MPR set:  
235 \*  
236 \* Neighbor Type = MPR\_NEIGH  
237 \*/  
238 if (neighbor->is\_mpr) {  
239 message\_neighbor->status = MPR\_NEIGH;  
240 }  
241 /\*  
242 \* 2.2 Otherwise, if the main address, corresponding to  
243 \* L\_neighbor\_iface\_addr, is included in the neighbor set:  
244 \*/  
245  
246 /\* NOTE:  
247 \* It is garanteed to be included when come this far  
248 \* due to the extentions made in the link sensing  
249 \* regarding main addresses.  
250 \*/  
251 else {  
252  
253 /\*  
254 \* 2.2.1  
255 \* if N\_status == SYM  
256 \*  
257 \* Neighbor Type = SYM\_NEIGH  
258 \*/  
259 if (neighbor->status == SYM) {  
260 message\_neighbor->status = SYM\_NEIGH;  
261 }  
262  
263 /\*  
264 \* 2.2.2  
265 \* Otherwise, if N\_status == NOT\_SYM  
266 \* Neighbor Type = NOT\_NEIGH  
267 \*/  
268 else if (neighbor->status == NOT\_SYM) {  
269 message\_neighbor->status = NOT\_NEIGH;  
270 }  
271 }  
272  
273 message\_neighbor->address = neighbor->neighbor\_main\_addr;  
274 message\_neighbor->main\_address = neighbor->neighbor\_main\_addr;  
275 #ifdef DEBUG  
276 OLSR\_PRINTF(5, "Added: %s - status %d\n", olsr\_ip\_to\_string(&buf, &message\_neighbor->address), message\_neighbor->status);  
277 #endif  
278 message\_neighbor->next = message->neighbors;  
279 message->neighbors = message\_neighbor;  
280  
281 }  
282 OLSR\_FOR\_ALL\_NBR\_ENTRIES\_END(neighbor);  
283  
284 return 0;  
285 }

106和113：设置willingness和ttl。HELLO消息只能在一跳范围内传播，所以ttl值设为1.

120-180：遍历该接口的所有链路得到链路状态，用以计算邻居的状态。如果邻居的主地址位于该节点的MPR集合，则将邻居的状态更新为MPR\_NEIGH；如果邻居的主地址位于该节点的邻居列表（即不为MPR），则判断之间的链路状态是否对称并更新邻居的状态域。

182-186：设置邻居节点的接口地址以及主地址。

58 struct link\_entry {  
59 union olsr\_ip\_addr local\_iface\_addr;  
60 union olsr\_ip\_addr neighbor\_iface\_addr;  
61 const struct interface \*inter;  
62 char \*if\_name;  
63 struct timer\_entry \*link\_timer;  
64 struct timer\_entry \*link\_sym\_timer;  
65 uint32\_t ASYM\_time;  
66 olsr\_reltime vtime;  
67 struct neighbor\_entry \*neighbor;  
68 uint8\_t prev\_status;  
69  
70 /\*  
71 \* Hysteresis  
72 \*/  
73 float L\_link\_quality;  
74 int L\_link\_pending;  
75 uint32\_t L\_LOST\_LINK\_time;  
76 struct timer\_entry \*link\_hello\_timer; /\* When we should receive a new HELLO \*/  
77 olsr\_reltime last\_htime;  
78 bool olsr\_seqno\_valid;  
79 uint16\_t olsr\_seqno;  
80  
81 /\*  
82 \* packet loss  
83 \*/  
84 olsr\_reltime loss\_helloint;  
85 struct timer\_entry \*link\_loss\_timer;  
86  
87 /\* user defined multiplies for link quality, multiplied with 65536 \*/  
88 uint32\_t loss\_link\_multiplier;  
89  
90 /\* cost of this link \*/  
91 olsr\_linkcost linkcost;  
92  
93 struct list\_node link\_list; /\* double linked list of all link entries \*/  
94 uint32\_t linkquality[0];

58-62：local*iface*addr存储了节点接口的IP地址，neighbor*iface*addr存储邻居节点的接口IP地址。

63-68：各种时间的存储结构。link*timer定时器，link*sys*timer定时器。Vtime表示接受该消息之后多少时间之内视其为有效。Neighbor以链表的形式存储邻居节点信息，pre*status记录上一个节点的状态。

58 struct neighbor\_entry {  
59 union olsr\_ip\_addr neighbor\_main\_addr;  
60 uint8\_t status;  
61 uint8\_t willingness;  
62 bool is\_mpr;  
63 bool was\_mpr; /\* Used to detect changes in MPR \*/  
64 bool skip;  
65 int neighbor\_2\_nocov;  
66 int linkcount;  
67 struct neighbor\_2\_list\_entry neighbor\_2\_list;  
68 struct neighbor\_entry \*next;  
69 struct neighbor\_entry \*prev;  
70 };

58-69：结构体neighbor\_entry用来存储邻居节点的信息。分别存储了邻居的主地址、状态、转发意愿、覆盖两跳邻居的数量、节点连接的链路数量。其中还存储两个布尔类型的值用来判断该节点是否是MPR以及该MPR是否发生过改变。最后包含了指向两跳邻居列表的指针。

49 struct neighbor\_2\_list\_entry {  
50 struct neighbor\_entry \*nbr2\_nbr; /\* backpointer to owning nbr entry \*/  
51 struct neighbor\_2\_entry \*neighbor\_2;  
52 struct timer\_entry \*nbr2\_list\_timer;  
53 struct neighbor\_2\_list\_entry \*next;  
54 struct neighbor\_2\_list\_entry \*prev;  
55 };

49-54：结构体neighbor*2*list\_entry用来存储两跳节点列表中的信息。均采用指针的方式记录节点信息及结点的两跳邻居节点的信息以及列表的有效时间等。

#### 2.3.2 节点操作

119 struct link\_entry \*  
120 lookup\_link\_entry(const union olsr\_ip\_addr \*remote, const union olsr\_ip\_addr \*remote\_main, const struct interface \*local)  
121 {  
122 struct link\_entry \*link;  
123  
124 OLSR\_FOR\_ALL\_LINK\_ENTRIES(link) {  
125 if (ipequal(remote, &link->neighbor\_iface\_addr)  
126 && (link->if\_name ? !strcmp(link->if\_name, local->int\_name) : ipequal(&local->ip\_addr, &link->local\_iface\_addr))) {  
127 /\* check the remote-main address only if there is one given \*/  
128 if (NULL != remote\_main && !ipequal(remote\_main, &link->neighbor->neighbor\_main\_addr)) {  
129 /\* Neighbor has changed it's main\_addr, update \*/  
130 struct ipaddr\_str oldbuf, newbuf;  
131 OLSR\_PRINTF(1, "Neighbor changed main\_ip, updating %s -> %s\n",  
132 olsr\_ip\_to\_string(&oldbuf, &link->neighbor->neighbor\_main\_addr), olsr\_ip\_to\_string(&newbuf, remote\_main));  
133 link->neighbor->neighbor\_main\_addr = \*remote\_main;  
134 }  
135 return link;  
136 }  
137 }  
138 OLSR\_FOR\_ALL\_LINK\_ENTRIES\_END(link);  
139  
140 return NULL;

160-168：该函数的功能是获取链路的状态，状态基于链路条目中的不同超时，根据不同的定时器值将返回不同的链路状态信息，如link*sys*timer对应链路状态为对称链路。

179 static int  
180 get\_neighbor\_status(const union olsr\_ip\_addr \*address)  
181 {  
182 const union olsr\_ip\_addr \*main\_addr;  
183 struct interface \*ifs;  
184  
185 /\* Find main address \*/  
186 if (!(main\_addr = mid\_lookup\_main\_addr(address)))  
187 main\_addr = address;  
188  
189 /\* Loop trough local interfaces to check all possebilities \*/  
190 for (ifs = ifnet; ifs != NULL; ifs = ifs->int\_next) {  
191 struct mid\_address \*aliases;  
192 struct link\_entry \*lnk = lookup\_link\_entry(main\_addr, NULL, ifs);  
193  
194 if (lnk != NULL) {  
195 if (lookup\_link\_status(lnk) == SYM\_LINK)  
196 return SYM\_LINK;  
197 }  
198  
199 /\* Get aliases \*/  
200 for (aliases = mid\_lookup\_aliases(main\_addr); aliases != NULL; aliases = aliases->next\_alias) {  
201  
202 lnk = lookup\_link\_entry(&aliases->alias, NULL, ifs);  
203 if (lnk && (lookup\_link\_status(lnk) == SYM\_LINK)) {  
204 return SYM\_LINK;  
205 }  
206 }  
207 }  
208  
209 return 0;  
210 }  
211

182-197：通过查找main*addr找到节点，然后通过look*link\_status找到节点链路状态，并判断是否对称。

200-206：查找主地址并找出节点其他端口的IP，判断该节点其他端口的链路状态。并判断该IP所在的链路状态是否对称，并返回对称链路的信息。

#### 2.3.3 操作邻居表

void  
olsr\_init\_neighbor\_table(void)  
{  
 int i;  
  
 for (i = 0; i < HASHSIZE; i++) {  
 neighbortable[i].next = &neighbortable[i];  
 neighbortable[i].prev = &neighbortable[i];  
 }  
}

59-63：初始化邻居表。将每一个邻居表neighbortable[i]初始化为指向自身的仅有一个节点的链表。

139 struct neighbor\_2\_list\_entry \*  
140 olsr\_lookup\_my\_neighbors(const struct neighbor\_entry \*neighbor, const union olsr\_ip\_addr \*neighbor\_main\_address)  
141 {  
142 struct neighbor\_2\_list\_entry \*entry;  
143  
144 for (entry = neighbor->neighbor\_2\_list.next; entry != &neighbor->neighbor\_2\_list; entry = entry->next) {  
145  
146 if (ipequal(&entry->neighbor\_2->neighbor\_2\_addr, neighbor\_main\_address))  
147 return entry;  
148  
149 }  
150 return NULL;  
151 }

142-150：该函数的功能是检查两跳邻居是否可以通过给定的邻居访问。遍历邻居节点的两跳邻居节点列表。如果找到列表中存在IP地址与给定的地址相匹配的两跳邻居节点，则返回该两跳邻居节点列表结构体。

260 struct neighbor\_entry \*  
261 olsr\_lookup\_neighbor\_table(const union olsr\_ip\_addr \*dst)  
262 {  
263 /\*  
264 \*Find main address of node  
265 \*/  
266 union olsr\_ip\_addr \*tmp\_ip = mid\_lookup\_main\_addr(dst);  
267 if (tmp\_ip != NULL)  
268 dst = tmp\_ip;  
269 return olsr\_lookup\_neighbor\_table\_alias(dst);  
270 }

271-274：该函数的功能是基于给定的地址在邻居表中查找邻居项。

299 int  
300 update\_neighbor\_status(struct neighbor\_entry \*entry, int lnk)  
301 {  
302 /\*  
303 \* Update neighbor entry  
304 \*/  
305  
306 if (lnk == SYM\_LINK) {  
307 /\* N\_status is set to SYM \*/  
308 if (entry->status == NOT\_SYM) {  
309 struct neighbor\_2\_entry \*two\_hop\_neighbor;  
310  
311 /\* Delete posible 2 hop entry on this neighbor \*/  
312 if ((two\_hop\_neighbor = olsr\_lookup\_two\_hop\_neighbor\_table(&entry->neighbor\_main\_addr)) != NULL) {  
313 olsr\_delete\_two\_hop\_neighbor\_table(two\_hop\_neighbor);  
314 }  
315  
316 changes\_neighborhood = true;  
317 changes\_topology = true;  
318 if (olsr\_cnf->tc\_redundancy > 1)  
319 signal\_link\_changes(true);  
320 }  
321 entry->status = SYM;  
322 } else {  
323 if (entry->status == SYM) {  
324 changes\_neighborhood = true;  
325 changes\_topology = true;  
326 if (olsr\_cnf->tc\_redundancy > 1)  
327 signal\_link\_changes(true);  
328 }  
329 /\* else N\_status is set to NOT\_SYM \*/  
330 entry->status = NOT\_SYM;  
331 /\* remove neighbor from routing list \*/  
332 }  
333  
334 return entry->status;  
335 }

该函数的功能为更新邻居的状态。

313-328：如果链路状态更新为SYM\_LINK时，原来非对称的链路通知网络重新选举MPR和更新路由表并删除通过该邻居节点到达的两跳邻居节点。

329-339：如果链路状态原先为SYS\_LINK，通知网络重新进行MPR的选取和路由表的更新。

215 struct neighbor\_entry \*  
216 olsr\_insert\_neighbor\_table(const union olsr\_ip\_addr \*main\_addr)  
217 {  
218 uint32\_t hash;  
219 struct neighbor\_entry \*new\_neigh;  
220  
221 hash = olsr\_ip\_hashing(main\_addr);  
222  
223 /\* Check if entry exists \*/  
224  
225 for (new\_neigh = neighbortable[hash].next; new\_neigh != &neighbortable[hash]; new\_neigh = new\_neigh->next) {  
226 if (ipequal(&new\_neigh->neighbor\_main\_addr, main\_addr))  
227 return new\_neigh;  
228 }  
229  
230 //printf("inserting neighbor\n");  
231  
232 new\_neigh = olsr\_malloc(sizeof(struct neighbor\_entry), "New neighbor entry");  
233  
234 /\* Set address, willingness and status \*/  
235 new\_neigh->neighbor\_main\_addr = \*main\_addr;  
236 new\_neigh->willingness = WILL\_NEVER;  
237 new\_neigh->status = NOT\_SYM;  
238  
239 new\_neigh->neighbor\_2\_list.next = &new\_neigh->neighbor\_2\_list;  
240 new\_neigh->neighbor\_2\_list.prev = &new\_neigh->neighbor\_2\_list;  
241  
242 new\_neigh->linkcount = 0;  
243 new\_neigh->is\_mpr = false;  
244 new\_neigh->was\_mpr = false;  
245  
246 /\* Queue \*/  
247 QUEUE\_ELEM(neighbortable[hash], new\_neigh);  
248  
249 return new\_neigh;  
250 }

该函数的功能是在邻居节点表中插入一条邻居节点信息。如果已存在该信息返回插入成功则返回1。

230-233：检查表项是否存在。

237-249：添加邻居节点信息，将地址、意愿、状态等值初始化。

160 int  
161 olsr\_delete\_neighbor\_table(const union olsr\_ip\_addr \*neighbor\_addr)  
162 {  
163 struct neighbor\_2\_list\_entry \*two\_hop\_list, \*two\_hop\_to\_delete;  
164 uint32\_t hash;  
165 struct neighbor\_entry \*entry;  
166  
167 //printf("inserting neighbor\n");  
168  
169 hash = olsr\_ip\_hashing(neighbor\_addr);  
170  
171 entry = neighbortable[hash].next;  
172  
173 /\*  
174 \* Find neighbor entry  
175 \*/  
176 while (entry != &neighbortable[hash]) {  
177 if (ipequal(&entry->neighbor\_main\_addr, neighbor\_addr))  
178 break;  
179  
180 entry = entry->next;  
181 }  
182  
183 if (entry == &neighbortable[hash])  
184 return 0;  
185  
186 two\_hop\_list = entry->neighbor\_2\_list.next;  
187  
188 while (two\_hop\_list != &entry->neighbor\_2\_list) {  
189 two\_hop\_to\_delete = two\_hop\_list;  
190 two\_hop\_list = two\_hop\_list->next;  
191  
192 two\_hop\_to\_delete->neighbor\_2->neighbor\_2\_pointer--;  
193 olsr\_delete\_neighbor\_pointer(two\_hop\_to\_delete->neighbor\_2, entry);  
194  
195 olsr\_del\_nbr2\_list(two\_hop\_to\_delete);  
196 }  
197  
198 /\* Dequeue \*/  
199 DEQUEUE\_ELEM(entry);  
200  
201 free(entry);  
202  
203 changes\_neighborhood = true;  
204 return 1;  
205  
206 }

165-211：该函数的功能是删除邻居表项，删除邻居表项的同时会删除掉该节点存储的两跳邻居列表。

### 2.4 MPR

OLSR采用MPR机制对路由信息进行选择性的洪泛。MPR作为OLSR协议的核心部分，算法描述已经在前面描述过了，此处不再赘述。

#### 2.4.1 MPR生成

357 static uint16\_t  
358 add\_will\_always\_nodes(void)  
359 {  
360 struct neighbor\_entry \*a\_neighbor;  
361 uint16\_t count = 0;  
362  
363 #if 0  
364 printf("\nAdding WILL ALWAYS nodes....\n");  
365 #endif  
366  
367 OLSR\_FOR\_ALL\_NBR\_ENTRIES(a\_neighbor) {  
368 struct ipaddr\_str buf;  
369 if ((a\_neighbor->status == NOT\_SYM) || (a\_neighbor->willingness != WILL\_ALWAYS)) {  
370 continue;  
371 }  
372 olsr\_chosen\_mpr(a\_neighbor, &count);  
373  
374 OLSR\_PRINTF(3, "Adding WILL\_ALWAYS: %s\n", olsr\_ip\_to\_string(&buf, &a\_neighbor->neighbor\_main\_addr));  
375  
376 }  
377 OLSR\_FOR\_ALL\_NBR\_ENTRIES\_END(a\_neighbor);  
378  
379 #if 0  
380 OLSR\_PRINTF(1, "Count: %d\n", count);  
381 #endif  
382 return count;  
383 }

该函数的作用是添加willingness为WILLALWAYS的邻居节点到MPR集合中。

368-383：对于非对称以及转发意愿为WILLNEVER的邻居节点不进行处理，其余节点添加到MPR集合中，并返回添加的节点的数量。

236 static void  
237 olsr\_clear\_mprs(void)  
238 {  
239 struct neighbor\_entry \*a\_neighbor;  
240 struct neighbor\_2\_list\_entry \*two\_hop\_list;  
241  
242 OLSR\_FOR\_ALL\_NBR\_ENTRIES(a\_neighbor) {  
243  
244 /\* Clear MPR selection. \*/  
245 if (a\_neighbor->is\_mpr) {  
246 a\_neighbor->was\_mpr = true;  
247 a\_neighbor->is\_mpr = false;  
248 }  
249  
250 /\* Clear two hop neighbors coverage count/ \*/  
251 for (two\_hop\_list = a\_neighbor->neighbor\_2\_list.next; two\_hop\_list != &a\_neighbor->neighbor\_2\_list;  
252 two\_hop\_list = two\_hop\_list->next) {  
253 two\_hop\_list->neighbor\_2->mpr\_covered\_count = 0;  
254 }  
255  
256 }  
257 OLSR\_FOR\_ALL\_NBR\_ENTRIES\_END(a\_neighbor);

该函数的功能是清除被选为MPR的节点。

246-249：如果节点目前是MPR节点，那就将is*mpr项设为false，was*mpr设为true，表明该节点过去是MPR节点，现在不是了。

252-255：当删除一个MPR节点时，该节点存储的两跳邻居列表也应该清除。所以将邻居节点覆盖的两跳邻居节点的数量置0。

140 static int  
141 olsr\_chosen\_mpr(struct neighbor\_entry \*one\_hop\_neighbor, uint16\_t \* two\_hop\_covered\_count)  
142 {  
143 struct neighbor\_list\_entry \*the\_one\_hop\_list;  
144 struct neighbor\_2\_list\_entry \*second\_hop\_entries;  
145 struct neighbor\_entry \*dup\_neighbor;  
146 uint16\_t count;  
147 struct ipaddr\_str buf;  
148 count = \*two\_hop\_covered\_count;  
149  
150 OLSR\_PRINTF(1, "Setting %s as MPR\n", olsr\_ip\_to\_string(&buf, &one\_hop\_neighbor->neighbor\_main\_addr));  
151  
152 //printf("PRE COUNT: %d\n\n", count);  
153  
154 one\_hop\_neighbor->is\_mpr = true; //NBS\_MPR;  
155  
156 for (second\_hop\_entries = one\_hop\_neighbor->neighbor\_2\_list.next; second\_hop\_entries != &one\_hop\_neighbor->neighbor\_2\_list;  
157 second\_hop\_entries = second\_hop\_entries->next) {  
158 dup\_neighbor = olsr\_lookup\_neighbor\_table(&second\_hop\_entries->neighbor\_2->neighbor\_2\_addr);  
159  
160 if ((dup\_neighbor != NULL) && (dup\_neighbor->status == SYM)) {  
161 //OLSR\_PRINTF(7, "(2)Skipping 2h neighbor %s - already 1hop\n", olsr\_ip\_to\_string(&buf, &second\_hop\_entries->neighbor\_2->neighbor\_2\_addr));  
162 continue;  
163 }  
164 // if(!second\_hop\_entries->neighbor\_2->neighbor\_2\_state)  
165 //if(second\_hop\_entries->neighbor\_2->mpr\_covered\_count < olsr\_cnf->mpr\_coverage)  
166 //{  
167 /\*  
168 Now the neighbor is covered by this mpr  
169 \*/  
170 second\_hop\_entries->neighbor\_2->mpr\_covered\_count++;  
171 the\_one\_hop\_list = second\_hop\_entries->neighbor\_2->neighbor\_2\_nblist.next;  
172  
173 //OLSR\_PRINTF(1, "[%s](%x) has coverage %d\n", olsr\_ip\_to\_string(&buf, &second\_hop\_entries->neighbor\_2->neighbor\_2\_addr), second\_hop\_entries->neighbor\_2, second\_hop\_entries->neighbor\_2->mpr\_covered\_count);  
174  
175 if (second\_hop\_entries->neighbor\_2->mpr\_covered\_count >= olsr\_cnf->mpr\_coverage)  
176 count++;  
177  
178 while (the\_one\_hop\_list != &second\_hop\_entries->neighbor\_2->neighbor\_2\_nblist) {  
179  
180 if ((the\_one\_hop\_list->neighbor->status == SYM)) {  
181 if (second\_hop\_entries->neighbor\_2->mpr\_covered\_count >= olsr\_cnf->mpr\_coverage) {  
182 the\_one\_hop\_list->neighbor->neighbor\_2\_nocov--;  
183 }  
184 }  
185 the\_one\_hop\_list = the\_one\_hop\_list->next;  
186 }  
187  
188 //}  
189 }  
190  
191 //printf("POST COUNT %d\n\n", count);  
192  
193 \*two\_hop\_covered\_count = count;  
194 return count;  
195  
196 }

该函数的作用是用来处理已经选定的MPR节点，函数的返回值为一跳邻居节点中MPR的数量。

159-165：对second*hop*entries进行遍历

173：该邻居节点被MPR覆盖，将两跳邻居节点被MPR覆盖的数量+1。

181-188：如果两跳节点被MPR覆盖的数量大于全局变量mpr\_coverage，将count递减。

265 static int  
266 olsr\_check\_mpr\_changes(void)  
267 {  
268 struct neighbor\_entry \*a\_neighbor;  
269 int retval;  
270  
271 retval = 0;  
272  
273 OLSR\_FOR\_ALL\_NBR\_ENTRIES(a\_neighbor) {  
274  
275 if (a\_neighbor->was\_mpr) {  
276 a\_neighbor->was\_mpr = false;  
277  
278 if (!a\_neighbor->is\_mpr) {  
279 retval = 1;  
280 }  
281 }  
282 }  
283 OLSR\_FOR\_ALL\_NBR\_ENTRIES\_END(a\_neighbor);  
284  
285 return retval;  
286 }

该函数的作用是遍历所有的MPR节点判断其状态是否发生变化，若发生变化返回值为1，否则为0。

282-292：对于结点如果其was*mpr值为true，表明它过去是MPR，is*mpr值为false，表示现在不是MPR，所以它的状态发生了变化，将retval置为1

#### 2.4.2 MPR选择

MPR的选择原则是MPR为对称邻居节点，通过MPR可以到达所有的两跳邻居节点，并且MPR的数量要尽可能少。接下来介绍有关两跳邻居节点的函数以及如何周到能够覆盖最多两跳节点的MPR。

86 static struct neighbor\_2\_list\_entry \*  
87 olsr\_find\_2\_hop\_neighbors\_with\_1\_link(int willingness)  
88 {  
89   
90 uint8\_t idx;  
91 struct neighbor\_2\_list\_entry \*two\_hop\_list\_tmp = NULL;  
92 struct neighbor\_2\_list\_entry \*two\_hop\_list = NULL;  
93 struct neighbor\_entry \*dup\_neighbor;  
94 struct neighbor\_2\_entry \*two\_hop\_neighbor = NULL;  
95   
96 for (idx = 0; idx < HASHSIZE; idx++) {  
97   
98 for (two\_hop\_neighbor = two\_hop\_neighbortable[idx].next; two\_hop\_neighbor != &two\_hop\_neighbortable[idx];  
99 two\_hop\_neighbor = two\_hop\_neighbor->next) {  
100  
101 //two\_hop\_neighbor->neighbor\_2\_state=0;  
102 //two\_hop\_neighbor->mpr\_covered\_count = 0;  
103  
104 dup\_neighbor = olsr\_lookup\_neighbor\_table(&two\_hop\_neighbor->neighbor\_2\_addr);  
105  
106 if ((dup\_neighbor != NULL) && (dup\_neighbor->status != NOT\_SYM)) {  
107  
108 //OLSR\_PRINTF(1, "(1)Skipping 2h neighbor %s - already 1hop\n", olsr\_ip\_to\_string(&buf, &two\_hop\_neighbor->neighbor\_2\_addr));  
109  
110 continue;  
111 }  
112  
113 if (two\_hop\_neighbor->neighbor\_2\_pointer == 1) {  
114 if ((two\_hop\_neighbor->neighbor\_2\_nblist.next->neighbor->willingness == willingness)  
115 && (two\_hop\_neighbor->neighbor\_2\_nblist.next->neighbor->status == SYM)) {  
116 two\_hop\_list\_tmp = olsr\_malloc(sizeof(struct neighbor\_2\_list\_entry), "MPR two hop list");  
117  
118 //OLSR\_PRINTF(1, "ONE LINK ADDING %s\n", olsr\_ip\_to\_string(&buf, &two\_hop\_neighbor->neighbor\_2\_addr));  
119  
120 /\* Only queue one way here \*/  
121 two\_hop\_list\_tmp->neighbor\_2 = two\_hop\_neighbor;  
122  
123 two\_hop\_list\_tmp->next = two\_hop\_list;  
124  
125 two\_hop\_list = two\_hop\_list\_tmp;  
126 }  
127 }  
128  
129 }  
130  
131 }  
132  
133 return (two\_hop\_list\_tmp);  
134 }  
135

该函数的作用是查找一条连接两跳邻居节点的链表。

92-95：定义了两个两跳邻居节点链表（two*hop*temp，two*hop*list），前者是函数的返回值。

Dup*neighor记录邻居节点集合中已经存在的节点，two*hop\_neighbor记录一个两跳邻居节点的局部变量。

97-112：遍历两跳邻居列表two*hop*neighbortable。

114-117：寻找邻居表中已经存在的邻居地址neighbor*2*addr，忽略与本节点不对称的邻居节点。

122-126：如果该两跳邻居节点不在两跳邻居链表中，且只有一个邻居节点，则将该两跳邻居节点加入链表。

206 static struct neighbor\_entry \*  
207 olsr\_find\_maximum\_covered(int willingness)  
208 {  
209 uint16\_t maximum;  
210 struct neighbor\_entry \*a\_neighbor;  
211 struct neighbor\_entry \*mpr\_candidate = NULL;  
212  
213 maximum = 0;  
214  
215 OLSR\_FOR\_ALL\_NBR\_ENTRIES(a\_neighbor) {  
216  
217 #if 0  
218 printf("[%s] nocov: %d mpr: %d will: %d max: %d\n\n", olsr\_ip\_to\_string(&buf, &a\_neighbor->neighbor\_main\_addr),  
219 a\_neighbor->neighbor\_2\_nocov, a\_neighbor->is\_mpr, a\_neighbor->willingness, maximum);  
220 #endif  
221  
222 if ((!a\_neighbor->is\_mpr) && (a\_neighbor->willingness == willingness) && (maximum < a\_neighbor->neighbor\_2\_nocov)) {  
223  
224 maximum = a\_neighbor->neighbor\_2\_nocov;  
225 mpr\_candidate = a\_neighbor;  
226 }  
227 }  
228 OLSR\_FOR\_ALL\_NBR\_ENTRIES\_END(a\_neighbor);  
229  
230 return mpr\_candidate;  
231 }

该函数的作用是找到能够覆盖最多两跳节点的MPR。

225-233：如果该一跳邻居节点不是MPR，而且该节点覆盖的两跳邻居节点的数量比maximum大，则更新maximum的值并将该一跳邻居节点作为候选MPR节点。

### 2.5 拓扑发现

网络中的MPR节点每隔一段时间会广播TC分组，用以维护网络的拓扑信息。在算法描述中已经说明，对于相同的TC分组,节点只有在第一次收到且其选择为MPR的情况下才转发，这样减少了网络中的广播包的数量，尽可能的避免了网络风暴。

#### 2.5.1 TC分组生成

81 void  
82 generate\_tc(void \*p)  
83 {  
84 struct tc\_message tcpacket;  
85 struct interface \*ifn = (struct interface \*)p;  
86  
87 olsr\_build\_tc\_packet(&tcpacket);  
88  
89 if (queue\_tc(&tcpacket, ifn) && TIMED\_OUT(ifn->fwdtimer)) {  
90 set\_buffer\_timer(ifn);  
91 }  
92  
93 olsr\_free\_tc\_packet(&tcpacket);  
94 }

首先构建一个TC分组的结构体，然后使用olsr*build*tc*packet（）函数对该结构体进行一些初始化和赋值操作，然后queue*tc（）函数将该分组加入MID队列中，同时TIMED*OUT()检测接口的时间的时间戳是否已满，调用set*buffer*timer()设置定时器。最后由接口ifn释放该分组，然后调用olsr*free*tc*packet()函数释放内存。

#### 2.5.2 拓扑信息集的初始化

185 void  
186 olsr\_init\_tc(void)  
187 {  
188 OLSR\_PRINTF(5, "TC: init topo\n");  
189  
190 avl\_init(&tc\_tree, avl\_comp\_default);  
191  
192 /\*  
193 \* Get some cookies for getting stats to ease troubleshooting.  
194 \*/  
195 tc\_edge\_gc\_timer\_cookie = olsr\_alloc\_cookie("TC edge GC", OLSR\_COOKIE\_TYPE\_TIMER);  
196 tc\_validity\_timer\_cookie = olsr\_alloc\_cookie("TC validity", OLSR\_COOKIE\_TYPE\_TIMER);  
197  
198 tc\_edge\_mem\_cookie = olsr\_alloc\_cookie("tc\_edge\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
199 olsr\_cookie\_set\_memory\_size(tc\_edge\_mem\_cookie, sizeof(struct tc\_edge\_entry) + active\_lq\_handler->tc\_lq\_size);  
200  
201 tc\_mem\_cookie = olsr\_alloc\_cookie("tc\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
202 olsr\_cookie\_set\_memory\_size(tc\_mem\_cookie, sizeof(struct tc\_entry));  
203  
204 /\*  
205 \* Add a TC entry for ourselves.  
206 \*/  
207 tc\_myself = olsr\_add\_tc\_entry(&olsr\_cnf->main\_addr);  
208 }

6:avl\_init()初始化拓扑信息集为avl树。

11-18: 对拓扑表集合进行初始化，主要通过从cookie中获取的值为拓扑表集合的属性做初始化赋值。

23:调用olsr*add*tc\_entry（）配置一个entry,并将该entry加入avl树中。

#### 2.5.3 TC分组处理

809 /\* We are only interested in TC message types. \*/  
810 pkt\_get\_u8(&curr, &type);  
811 if ((type != LQ\_TC\_MESSAGE) && (type != TC\_MESSAGE)) {  
812 return false;  
813 }  
814  
815 /\*  
816 \* If the sender interface (NB: not originator) of this message  
817 \* is not in the symmetric 1-hop neighborhood of this node, the  
818 \* message MUST be discarded.  
819 \*/  
820 if (check\_neighbor\_link(from\_addr) != SYM\_LINK) {  
821 OLSR\_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr\_ip\_to\_string(&buf, from\_addr));  
822 return false;  
823 }

当节点接受到一个TC分组后，只考虑其消息类型。如果类型不等于LQ*TC*MESSAGE或者TC\_MESSAGE，则直接丢弃。

如果检测到该消息的发送者接口不是本节点的对称一跳邻居，则丢弃该分组。

847 if (olsr\_seq\_inrange\_high((int)tc->msg\_seq - TC\_SEQNO\_WINDOW, tc->msg\_seq, msg\_seq)  
848 && olsr\_seq\_inrange\_high((int)tc->ansn - TC\_ANSN\_WINDOW, tc->ansn, ansn)) {  
849  
850 /\*  
851 \* Ignore already seen seq/ansn values (small window for mesh memory)  
852 \*/  
853 if ((tc->msg\_seq == msg\_seq) || (tc->ignored++ < 32)) {  
854 return false;  
855 }  
856  
857 OLSR\_PRINTF(1, "Ignored to much LQTC's for %s, restarting\n", olsr\_ip\_to\_string(&buf, &originator));

如果分组中的msg*seq和外部变量msg*seq相等，且ignored小于32，说明该分组已经处理过，所以丢弃该分组，返回false。

878 /\*  
879 \* Generate a new tc\_entry in the lsdb and store the sequence number.  
880 \*/  
881 if (!tc) {  
882 tc = olsr\_add\_tc\_entry(&originator);  
883 }  
884  
885 /\*  
886 \* Update the tc entry.  
887 \*/  
888 tc->msg\_hops = msg\_hops;  
889 tc->msg\_seq = msg\_seq;  
890 tc->ansn = ansn;  
891 tc->ignored = 0;  
892 tc->err\_seq\_valid = false;  
893  
894 OLSR\_PRINTF(1, "Processing TC from %s, seq 0x%04x\n", olsr\_ip\_to\_string(&buf, &originator), tc->msg\_seq);

如果现有的拓扑集中没有当前收到的TC分组中的Main Address，则添加一条新纪录，用TC分组中获取到的信息为该记录赋值。

915 /\*  
916 \* Calculate real border IPs.  
917 \*/  
918 if (borderSet) {  
919 borderSet = olsr\_calculate\_tc\_border(lower\_border, &lower\_border\_ip, upper\_border, &upper\_border\_ip);  
920 }  
921  
922 /\*  
923 \* Set or change the expiration timer accordingly.  
924 \*/  
925 olsr\_set\_timer(&tc->validity\_timer, vtime, OLSR\_TC\_VTIME\_JITTER, OLSR\_TIMER\_ONESHOT, &olsr\_expire\_tc\_entry, tc,  
926 tc\_validity\_timer\_cookie);  
927  
928 if (emptyTC && lower\_border == 0xff && upper\_border == 0xff) {  
929 /\* handle empty TC with border flags 0xff \*/  
930 memset(&lower\_border\_ip, 0x00, sizeof(lower\_border\_ip));  
931 memset(&upper\_border\_ip, 0xff, sizeof(upper\_border\_ip));  
932 borderSet = 1;  
933 }

调用olsr*calculate*tc\_border()计算borderset的值，并且重置相关的定时器。

#### 2.5.4 拓扑信息集的删除

278 void  
279 olsr\_delete\_tc\_entry(struct tc\_entry \*tc)  
280 {  
281 struct tc\_edge\_entry \*tc\_edge;  
282 struct rt\_path \*rtp;  
283 #if 0  
284 struct ipaddr\_str buf;  
285 OLSR\_PRINTF(1, "TC: del entry %s\n", olsr\_ip\_to\_string(&buf, &tc->addr));  
286 #endif  
287  
288 /\* delete gateway if available \*/  
289 #ifdef LINUX\_NETLINK\_ROUTING  
290 olsr\_delete\_gateway\_entry(&tc->addr, FORCE\_DELETE\_GW\_ENTRY);  
291 #endif  
292 /\*  
293 \* Delete the rt\_path for ourselves.  
294 \*/  
295 olsr\_delete\_routing\_table(&tc->addr, olsr\_cnf->maxplen, &tc->addr);  
296  
297 /\* The edgetree and prefix tree must be empty before \*/  
298 OLSR\_FOR\_ALL\_TC\_EDGE\_ENTRIES(tc, tc\_edge) {  
299 olsr\_delete\_tc\_edge\_entry(tc\_edge);  
300 } OLSR\_FOR\_ALL\_TC\_EDGE\_ENTRIES\_END(tc, tc\_edge);  
301  
302 OLSR\_FOR\_ALL\_PREFIX\_ENTRIES(tc, rtp) {  
303 olsr\_delete\_rt\_path(rtp);  
304 } OLSR\_FOR\_ALL\_PREFIX\_ENTRIES\_END(tc, rtp);  
305  
306 /\* Stop running timers \*/  
307 olsr\_stop\_timer(tc->edge\_gc\_timer);  
308 tc->edge\_gc\_timer = NULL;  
309 olsr\_stop\_timer(tc->validity\_timer);  
310 tc->validity\_timer = NULL;  
311  
312 avl\_delete(&tc\_tree, &tc->vertex\_node);  
313 olsr\_unlock\_tc\_entry(tc);  
314 }

该函数的功能是删除一个tc\_entry.

如果预定义了 LINUX*NETLINK*ROUTING(即在linux系统上运行)，则删除网关信息。删除时对网关的时间信息，网关协议等等先进行判断，判断这些信息是否为空后再进行删除。

首先删除本地路由表中的rt\_path；

清空所有的边，停止相应的计时器，将edge*gc*timer和validity\_time属性都置为空。

最后在avl树中删除相应节点。

### 2.6 路由表计算

#### 2.6.1 相关结构体

85 /\* a composite metric is used for path selection \*/  
86 struct rt\_metric {  
87 olsr\_linkcost cost;  
88 uint32\_t hops;  
89 };  
90  
91 /\* a nexthop is a pointer to a gateway router plus an interface \*/  
92 struct rt\_nexthop {  
93 union olsr\_ip\_addr gateway; /\* gateway router \*/  
94 int iif\_index; /\* outgoing interface index \*/  
95 };

rt\_metric:在路径选择时使用符合矩阵，矩阵中包括两个节点间的路径花销和跳数。

rt\_nexthop:该结构体表示吓一跳的网关和接口索引。

78 /\*  
79 \* Every prefix in our RIB needs a route entry that contains  
80 \* the nexthop of the best path as installed in the kernel FIB.  
81 \* The route entry is the root of a rt\_path tree of equal prefixes  
82 \* originated by different routers. It also contains a shortcut  
83 \* for accessing the best route among all contributing routes.  
84 \*/  
85 struct rt\_entry {  
86 struct olsr\_ip\_prefix rt\_dst;  
87 struct avl\_node rt\_tree\_node;  
88 struct rt\_path \*rt\_best; /\* shortcut to the best path \*/  
89 struct rt\_nexthop rt\_nexthop; /\* nexthop of FIB route \*/  
90 struct rt\_metric rt\_metric; /\* metric of FIB route \*/  
91 struct avl\_tree rt\_path\_tree;  
92 struct list\_node rt\_change\_node; /\* queue for kernel FIB add/chg/del \*/  
93 };

每一个RIB需要一个路由接口，这个接口中包含最佳路径的下一跳网关信息，同时该接口时rt*path*tree的根节点。同样也包含了所有路由信息中最佳的一个路径。rt*dst包含该信息的路由地址和前缀长度。rt*path*tree是一个avl树，rt*tree\_node表示最短路径的引用。

106 struct rt\_path {  
107 struct rt\_entry \*rtp\_rt; /\* backpointer to owning route head \*/  
108 struct tc\_entry \*rtp\_tc; /\* backpointer to owning tc entry \*/  
109 struct rt\_nexthop rtp\_nexthop;  
110 struct rt\_metric rtp\_metric;  
111 struct avl\_node rtp\_tree\_node; /\* global rtp node \*/  
112 union olsr\_ip\_addr rtp\_originator; /\* originator of the route \*/  
113 struct avl\_node rtp\_prefix\_tree\_node; /\* tc entry rtp node \*/  
114 struct olsr\_ip\_prefix rtp\_dst; /\* the prefix \*/  
115 uint32\_t rtp\_version; /\* for detection of outdated rt\_paths \*/  
116 uint8\_t rtp\_origin; /\* internal, MID or  
117 HNA \*/  
118 };

这个结构体主要描述了rt*path的成员，每接收到一个rt*path就将其加入RIB。根据Dijkstra算法计算出的结果可以得到最优路径，同时可以得到一个最小的矩阵。rt*path首先被加入到tc*entry树中，如果根据Dijkstra计算可得当前tc\_entry是可达到，则在全局RIB树中加入下一跳地址。

182 union olsr\_kernel\_route {  
183 struct {  
184 struct sockaddr rt\_dst;  
185 struct sockaddr rt\_gateway;  
186 uint32\_t metric;  
187 } v4;  
188  
189 struct {  
190 struct in6\_addr rtmsg\_dst;  
191 struct in6\_addr rtmsg\_gateway;  
192 uint32\_t rtmsg\_metric;  
193 } v6;  
194 };

该联合体分别定义了IPV4和IPV6的olsr核心路由表结构。路由表项中主要包含目的地和网关。

129 enum olsr\_rt\_origin {  
130 OLSR\_RT\_ORIGIN\_MIN,  
131 OLSR\_RT\_ORIGIN\_INT,  
132 OLSR\_RT\_ORIGIN\_MID,  
133 OLSR\_RT\_ORIGIN\_HNA,  
134 OLSR\_RT\_ORIGIN\_MAX  
135 };

OLSR中有三种不同的路由类型，INT（internal route）有简单的TC分组接收生成，MID由MID分组生成并且HNA路由由HNA公告生成。

#### 2.6.2 路由表计算

167 void  
168 olsr\_init\_routing\_table(void)  
169 {  
170 OLSR\_PRINTF(5, "RIB: init routing tree\n");  
171  
172 /\* the routing tree \*/  
173 avl\_init(&routingtree, avl\_comp\_prefix\_default);  
174 routingtree\_version = 0;  
175  
176 /\*  
177 \* Get some cookies for memory stats and memory recycling.  
178 \*/  
179 rt\_mem\_cookie = olsr\_alloc\_cookie("rt\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
180 olsr\_cookie\_set\_memory\_size(rt\_mem\_cookie, sizeof(struct rt\_entry));  
181  
182 rtp\_mem\_cookie = olsr\_alloc\_cookie("rt\_path", OLSR\_COOKIE\_TYPE\_MEMORY);  
183 olsr\_cookie\_set\_memory\_size(rtp\_mem\_cookie, sizeof(struct rt\_path));  
184 }

该函数的主要功能是初始化路由表。

首先调用avl*init()将路由表初始化为avl树，然后维护一个版本号routingtree*version用以检测每一个每一个rt*entry和rt*path，检查其是否过期，并将版本号初始化为0。

然后是为rt*entry和rt*path分配内存，并创建相应的cookie。

231 static struct rt\_entry \*  
232 olsr\_alloc\_rt\_entry(struct olsr\_ip\_prefix \*prefix)  
233 {  
234 struct rt\_entry \*rt = olsr\_cookie\_malloc(rt\_mem\_cookie);  
235 if (!rt) {  
236 return NULL;  
237 }  
238  
239 memset(rt, 0, sizeof(\*rt));  
240  
241 /\* Mark this entry as fresh (see process\_routes.c:512) \*/  
242 rt->rt\_nexthop.iif\_index = -1;  
243  
244 /\* set key and backpointer prior to tree insertion \*/  
245 rt->rt\_dst = \*prefix;  
246  
247 rt->rt\_tree\_node.key = &rt->rt\_dst;  
248 avl\_insert(&routingtree, &rt->rt\_tree\_node, AVL\_DUP\_NO);  
249  
250 /\* init the originator subtree \*/  
251 avl\_init(&rt->rt\_path\_tree, avl\_comp\_default);  
252  
253 return rt;  
254 }

该函数的功能是创建一个路由表项，并作一些初始化操作后将其插入avl树中。

首先是为新的路由表项申请内存空间，并且将空间清零。

标识该表项为新建表项，并将目的地址设置为参数提供的入口地址。

把该表项的树节点插入avl树，并初始化该树。

292 void  
293 olsr\_insert\_rt\_path(struct rt\_path \*rtp, struct tc\_entry \*tc, struct link\_entry \*link)  
294 {  
295 struct rt\_entry \*rt;  
296 struct avl\_node \*node;  
297  
298 /\*  
299 \* no unreachable routes please.  
300 \*/  
301 if (tc->path\_cost == ROUTE\_COST\_BROKEN) {  
302 return;  
303 }  
304  
305 /\*  
306 \* No bogus prefix lengths.  
307 \*/  
308 if (rtp->rtp\_dst.prefix\_len > olsr\_cnf->maxplen) {  
309 return;  
310 }  
311  
312 /\*  
313 \* first check if there is a route\_entry for the prefix.  
314 \*/  
315 node = avl\_find(&routingtree, &rtp->rtp\_dst);  
316  
317 if (!node) {  
318  
319 /\* no route entry yet \*/  
320 rt = olsr\_alloc\_rt\_entry(&rtp->rtp\_dst);  
321  
322 if (!rt) {  
323 return;  
324 }  
325  
326 } else {  
327 rt = rt\_tree2rt(node);  
328 }  
329  
330 /\* Now insert the rt\_path to the owning rt\_entry tree \*/  
331 rtp->rtp\_originator = tc->addr;  
332  
333 /\* set key and backpointer prior to tree insertion \*/  
334 rtp->rtp\_tree\_node.key = &rtp->rtp\_originator;  
335  
336 /\* insert to the route entry originator tree \*/  
337 avl\_insert(&rt->rt\_path\_tree, &rtp->rtp\_tree\_node, AVL\_DUP\_NO);  
338  
339 /\* backlink to the owning route entry \*/  
340 rtp->rtp\_rt = rt;  
341  
342 /\* update the version field and relevant parameters \*/  
343 olsr\_update\_rt\_path(rtp, tc, link);

该函数功能是对于每个rt\_path创建一个路由表项并将其加入全局的RIB树中。

首先检查传入的参数是否合法，如果传入的tc*entry的path*cost为ROUTE*COST*BROKEN，或者传入的rtp的目的地址长度大于所设置的最大地址长度，直接返回NULL。

然后调用avl*find()函数检查传入的rtp节点是否在路由表中，如果节点不在路由表中，将其加入avl树中。如果在，则将其从avl*node类型转为rt\_entry类型。

把新节点加入avl树，然后改变相应的参数，更新路由表。

211 void  
212 olsr\_update\_rt\_path(struct rt\_path \*rtp, struct tc\_entry \*tc, struct link\_entry \*link)  
213 {  
214  
215 rtp->rtp\_version = routingtree\_version;  
216  
217 /\* gateway \*/  
218 rtp->rtp\_nexthop.gateway = link->neighbor\_iface\_addr;  
219  
220 /\* interface \*/  
221 rtp->rtp\_nexthop.iif\_index = link->inter->if\_index;  
222  
223 /\* metric/etx \*/  
224 rtp->rtp\_metric.hops = tc->hops;  
225 rtp->rtp\_metric.cost = tc->path\_cost;  
226 }

该函数的主要功能是更新路由路径的网关，接口和路由跳数和路径的版本信息。

349 void  
350 olsr\_delete\_rt\_path(struct rt\_path \*rtp)  
351 {  
352  
353 /\* remove from the originator tree \*/  
354 if (rtp->rtp\_rt) {  
355 avl\_delete(&rtp->rtp\_rt->rt\_path\_tree, &rtp->rtp\_tree\_node);  
356 rtp->rtp\_rt = NULL;  
357 }  
358  
359 /\* remove from the tc prefix tree \*/  
360 if (rtp->rtp\_tc) {  
361 avl\_delete(&rtp->rtp\_tc->prefix\_tree, &rtp->rtp\_prefix\_tree\_node);  
362 olsr\_unlock\_tc\_entry(rtp->rtp\_tc);  
363 rtp->rtp\_tc = NULL;  
364 }  
365  
366 /\* no current inet gw if the rt\_path is removed \*/  
367 if (current\_inetgw == rtp) {  
368 current\_inetgw = NULL;  
369 }  
370  
371 olsr\_cookie\_free(rtp\_mem\_cookie, rtp);  
372 }

该函数功能是删除并释放一条路由路径。

首先将rtp所指的树节点从所在的树里面删除供将指向该树根节点的指针置空。

然后将其从前缀树中删除，并解锁相应的tc\_entry。

最后把rtp所指向的树节点从rtp树里面删除，释放cookie所占用的内存。

435 static bool  
436 olsr\_cmp\_rtp(const struct rt\_path \*rtp1, const struct rt\_path \*rtp2, const struct rt\_path \*inetgw)  
437 {  
438 olsr\_linkcost etx1 = rtp1->rtp\_metric.cost;  
439 olsr\_linkcost etx2 = rtp2->rtp\_metric.cost;  
440 if (inetgw == rtp1)  
441 etx1 \*= olsr\_cnf->lq\_nat\_thresh;  
442 if (inetgw == rtp2)  
443 etx2 \*= olsr\_cnf->lq\_nat\_thresh;  
444  
445 /\* etx comes first \*/  
446 if (etx1 < etx2) {  
447 return true;  
448 }  
449 if (etx1 > etx2) {  
450 return false;  
451 }  
452  
453 /\* hopcount is next tie breaker \*/  
454 if (rtp1->rtp\_metric.hops < rtp2->rtp\_metric.hops) {  
455 return true;  
456 }  
457 if (rtp1->rtp\_metric.hops > rtp2->rtp\_metric.hops) {  
458 return false;  
459 }  
460  
461 /\* originator (which is guaranteed to be unique) is final tie breaker \*/  
462 if (memcmp(&rtp1->rtp\_originator, &rtp2->rtp\_originator, olsr\_cnf->ipsize) < 0) {  
463 return true;  
464 }  
465  
466 return false;  
467 }

该函数的功能是比较两个路由路径，如过第一个路径更好，返回TRUE，否则，返回False。

首先比较路径花销，花销小的路径更优。

如果路径花销相同，则比较路径跳数，跳数更小的路径更优。

如果前两项比较结果相同，则比较源地址，源地址小的更优。

485 void  
486 olsr\_rt\_best(struct rt\_entry \*rt)  
487 {  
488 /\* grab the first entry \*/  
489 struct avl\_node \*node = avl\_walk\_first(&rt->rt\_path\_tree);  
490  
491 assert(node != 0); /\* should not happen \*/  
492  
493 rt->rt\_best = rtp\_tree2rtp(node);  
494  
495 /\* walk all remaining originator entries \*/  
496 while ((node = avl\_walk\_next(node))) {  
497 struct rt\_path \*rtp = rtp\_tree2rtp(node);  
498  
499 if (olsr\_cmp\_rtp(rtp, rt->rt\_best, current\_inetgw)) {  
500 rt->rt\_best = rtp;  
501 }  
502 }  
503  
504 if (0 == rt->rt\_dst.prefix\_len) {  
505 current\_inetgw = rt->rt\_best;  
506 }  
507 }

运行最优路径，首先得到第一个条表项后遍历所有表项，找到一条最有路径并修改当前网关路径到最优路径。

首先调用avl*walk*first()函数从rt.rt*path*tree中的到第一个条目并坚持其是否为0，然后把节点转为rt\_entry类型。

然后遍历整棵avl树，并比较当前路径和当前最优路径，获取最优路径。

523 struct rt\_path \*  
524 olsr\_insert\_routing\_table(union olsr\_ip\_addr \*dst, int plen, union olsr\_ip\_addr \*originator, int origin)  
525 {  
526 #ifdef DEBUG  
527 struct ipaddr\_str dstbuf, origbuf;  
528 #endif  
529 struct tc\_entry \*tc;  
530 struct rt\_path \*rtp;  
531 struct avl\_node \*node;  
532 struct olsr\_ip\_prefix prefix;  
533  
534 /\*  
535 \* No bogus prefix lengths.  
536 \*/  
537 if (plen > olsr\_cnf->maxplen) {  
538 return NULL;  
539 }  
540  
541 /\*  
542 \* For all routes we use the tc\_entry as an hookup point.  
543 \* If the tc\_entry is disconnected, i.e. has no edges it will not  
544 \* be explored during SPF run.  
545 \*/  
546 tc = olsr\_locate\_tc\_entry(originator);  
547  
548 /\*  
549 \* first check if there is a rt\_path for the prefix.  
550 \*/  
551 prefix.prefix = \*dst;  
552 prefix.prefix\_len = plen;  
553  
554 node = avl\_find(&tc->prefix\_tree, &prefix);  
555  
556 if (!node) {  
557  
558 /\* no rt\_path for this prefix yet \*/  
559 rtp = olsr\_alloc\_rt\_path(tc, &prefix, origin);  
560  
561 if (!rtp) {  
562 return NULL;  
563 }  
564 #ifdef DEBUG  
565 OLSR\_PRINTF(1, "RIB: add prefix %s/%u from %s\n", olsr\_ip\_to\_string(&dstbuf, dst), plen,  
566 olsr\_ip\_to\_string(&origbuf, originator));  
567 #endif  
568  
569 /\* overload the hna change bit for flagging a prefix change \*/  
570 changes\_hna = true;  
571  
572 } else {  
573 rtp = rtp\_prefix\_tree2rtp(node);  
574 }  
575  
576 return rtp;

数功能是将一个前缀节点插入一颗前缀树。首先检查是否该rt\_path是已知的，如果不是，则创建，如果根据Dijkstra算法得到节点不可达，最后计算最短路径是不考虑。