[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-n257)  
 [2.3 邻居检测](#header-n258)  
 [2.4 MPR](#header-n259)  
 [2.5 拓扑发现](#header-n260)  
 [2.5.1 TC分组生成](#header-n262)  
 [2.5.2 拓扑信息集的初始化](#header-n267)  
 [2.5.3 TC分组处理](#header-n274)  
 [2.5.4 拓扑信息集的删除](#header-n292)  
 [2.6 路由表计算](#header-n301)  
 [2.6.1 相关结构体](#header-n302)  
 [2.6.2 路由表计算](#header-n334)

# 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

### 2.2 数据结构

### 2.3 邻居检测

### 2.4 MPR

### 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

void  
generate\_tc(void \*p)  
{  
 struct tc\_message tcpacket;  
 struct interface \*ifn = (struct interface \*)p;  
  
 olsr\_build\_tc\_packet(&tcpacket);  
  
 if (queue\_tc(&tcpacket, ifn) && TIMED\_OUT(ifn->fwdtimer)) {  
 set\_buffer\_timer(ifn);  
 }  
  
 olsr\_free\_tc\_packet(&tcpacket);  
}

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

void  
olsr\_init\_tc(void)  
{  
 OLSR\_PRINTF(5, "TC: init topo\n");  
  
 avl\_init(&tc\_tree, avl\_comp\_default);  
  
 /\*  
 \* Get some cookies for getting stats to ease troubleshooting.  
 \*/  
 tc\_edge\_gc\_timer\_cookie = olsr\_alloc\_cookie("TC edge GC", OLSR\_COOKIE\_TYPE\_TIMER);  
 tc\_validity\_timer\_cookie = olsr\_alloc\_cookie("TC validity", OLSR\_COOKIE\_TYPE\_TIMER);  
  
 tc\_edge\_mem\_cookie = olsr\_alloc\_cookie("tc\_edge\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
 olsr\_cookie\_set\_memory\_size(tc\_edge\_mem\_cookie, sizeof(struct tc\_edge\_entry) + active\_lq\_handler->tc\_lq\_size);  
  
 tc\_mem\_cookie = olsr\_alloc\_cookie("tc\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
 olsr\_cookie\_set\_memory\_size(tc\_mem\_cookie, sizeof(struct tc\_entry));  
  
 /\*  
 \* Add a TC entry for ourselves.  
 \*/  
 tc\_myself = olsr\_add\_tc\_entry(&olsr\_cnf->main\_addr);  
}

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

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

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。

if (olsr\_seq\_inrange\_high((int)tc->msg\_seq - TC\_SEQNO\_WINDOW, tc->msg\_seq, msg\_seq)  
 && olsr\_seq\_inrange\_high((int)tc->ansn - TC\_ANSN\_WINDOW, tc->ansn, ansn)) {  
  
 /\*  
 \* Ignore already seen seq/ansn values (small window for mesh memory)  
 \*/  
 if ((tc->msg\_seq == msg\_seq) || (tc->ignored++ < 32)) {  
 return false;  
 }  
  
 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。

/\*  
 \* Generate a new tc\_entry in the lsdb and store the sequence number.  
 \*/  
 if (!tc) {  
 tc = olsr\_add\_tc\_entry(&originator);  
 }  
  
 /\*  
 \* Update the tc entry.  
 \*/  
 tc->msg\_hops = msg\_hops;  
 tc->msg\_seq = msg\_seq;  
 tc->ansn = ansn;  
 tc->ignored = 0;  
 tc->err\_seq\_valid = false;  
  
 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。

/\*  
 \* Calculate real border IPs.  
 \*/  
 if (borderSet) {  
 borderSet = olsr\_calculate\_tc\_border(lower\_border, &lower\_border\_ip, upper\_border, &upper\_border\_ip);  
 }  
  
 /\*  
 \* Set or change the expiration timer accordingly.  
 \*/  
 olsr\_set\_timer(&tc->validity\_timer, vtime, OLSR\_TC\_VTIME\_JITTER, OLSR\_TIMER\_ONESHOT, &olsr\_expire\_tc\_entry, tc,  
 tc\_validity\_timer\_cookie);  
  
 if (emptyTC && lower\_border == 0xff && upper\_border == 0xff) {  
 /\* handle empty TC with border flags 0xff \*/  
 memset(&lower\_border\_ip, 0x00, sizeof(lower\_border\_ip));  
 memset(&upper\_border\_ip, 0xff, sizeof(upper\_border\_ip));  
 borderSet = 1;  
 }

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

void  
olsr\_delete\_tc\_entry(struct tc\_entry \*tc)  
{  
 struct tc\_edge\_entry \*tc\_edge;  
 struct rt\_path \*rtp;  
#if 0  
 struct ipaddr\_str buf;  
 OLSR\_PRINTF(1, "TC: del entry %s\n", olsr\_ip\_to\_string(&buf, &tc->addr));  
#endif  
  
 /\* delete gateway if available \*/  
#ifdef LINUX\_NETLINK\_ROUTING  
 olsr\_delete\_gateway\_entry(&tc->addr, FORCE\_DELETE\_GW\_ENTRY);  
#endif  
 /\*  
 \* Delete the rt\_path for ourselves.  
 \*/  
 olsr\_delete\_routing\_table(&tc->addr, olsr\_cnf->maxplen, &tc->addr);  
  
 /\* The edgetree and prefix tree must be empty before \*/  
 OLSR\_FOR\_ALL\_TC\_EDGE\_ENTRIES(tc, tc\_edge) {  
 olsr\_delete\_tc\_edge\_entry(tc\_edge);  
 } OLSR\_FOR\_ALL\_TC\_EDGE\_ENTRIES\_END(tc, tc\_edge);  
  
 OLSR\_FOR\_ALL\_PREFIX\_ENTRIES(tc, rtp) {  
 olsr\_delete\_rt\_path(rtp);  
 } OLSR\_FOR\_ALL\_PREFIX\_ENTRIES\_END(tc, rtp);  
  
 /\* Stop running timers \*/  
 olsr\_stop\_timer(tc->edge\_gc\_timer);  
 tc->edge\_gc\_timer = NULL;  
 olsr\_stop\_timer(tc->validity\_timer);  
 tc->validity\_timer = NULL;  
  
 avl\_delete(&tc\_tree, &tc->vertex\_node);  
 olsr\_unlock\_tc\_entry(tc);  
}

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

/\* a composite metric is used for path selection \*/  
struct rt\_metric {  
 olsr\_linkcost cost;  
 uint32\_t hops;  
};  
  
/\* a nexthop is a pointer to a gateway router plus an interface \*/  
struct rt\_nexthop {  
 union olsr\_ip\_addr gateway; /\* gateway router \*/  
 int iif\_index; /\* outgoing interface index \*/  
};

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。

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

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。

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

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。

union olsr\_kernel\_route {  
 struct {  
 struct sockaddr rt\_dst;  
 struct sockaddr rt\_gateway;  
 uint32\_t metric;  
 } v4;  
 struct {  
 struct in6\_addr rtmsg\_dst;  
 struct in6\_addr rtmsg\_gateway;  
 uint32\_t rtmsg\_metric;  
 } v6;  
};

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。

enum olsr\_rt\_origin {  
 OLSR\_RT\_ORIGIN\_MIN,  
 OLSR\_RT\_ORIGIN\_INT,  
 OLSR\_RT\_ORIGIN\_MID,  
 OLSR\_RT\_ORIGIN\_HNA,  
 OLSR\_RT\_ORIGIN\_MAX  
};

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

void  
olsr\_init\_routing\_table(void)  
{  
 OLSR\_PRINTF(5, "RIB: init routing tree\n");  
 /\* the routing tree \*/  
 avl\_init(&routingtree, avl\_comp\_prefix\_default);  
 routingtree\_version = 0;  
  
 /\*  
 \* Get some cookies for memory stats and memory recycling.  
 \*/  
 rt\_mem\_cookie = olsr\_alloc\_cookie("rt\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);  
 olsr\_cookie\_set\_memory\_size(rt\_mem\_cookie, sizeof(struct rt\_entry));  
 rtp\_mem\_cookie = olsr\_alloc\_cookie("rt\_path", OLSR\_COOKIE\_TYPE\_MEMORY);  
 olsr\_cookie\_set\_memory\_size(rtp\_mem\_cookie, sizeof(struct rt\_path));  
}

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。

static struct rt\_entry \*  
olsr\_alloc\_rt\_entry(struct olsr\_ip\_prefix \*prefix)  
{  
 struct rt\_entry \*rt = olsr\_cookie\_malloc(rt\_mem\_cookie);  
 if (!rt) {  
 return NULL;  
 }  
 memset(rt, 0, sizeof(\*rt));  
  
 /\* Mark this entry as fresh (see process\_routes.c:512) \*/  
 rt->rt\_nexthop.iif\_index = -1;  
 /\* set key and backpointer prior to tree insertion \*/ rt->rt\_dst = \*prefix; rt->rt\_tree\_node.key = &rt->rt\_dst;  
 avl\_insert(&routingtree, &rt->rt\_tree\_node, AVL\_DUP\_NO);  
 /\* init the originator subtree \*/ avl\_init(&rt->rt\_path\_tree, avl\_comp\_default);  
 return rt;  
}

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。

void  
olsr\_insert\_rt\_path(struct rt\_path \*rtp, struct tc\_entry \*tc, struct link\_entry \*link)  
{  
 struct rt\_entry \*rt;  
 struct avl\_node \*node;  
 /\*  
 \* no unreachable routes please.  
 \*/  
 if (tc->path\_cost == ROUTE\_COST\_BROKEN) {  
 return;  
 } /\*  
 \* No bogus prefix lengths.  
 \*/  
 if (rtp->rtp\_dst.prefix\_len > olsr\_cnf->maxplen) {  
 return;  
 }  
 /\*  
 \* first check if there is a route\_entry for the prefix.  
 \*/  
 node = avl\_find(&routingtree, &rtp->rtp\_dst);  
 if (!node) {  
 /\* no route entry yet \*/  
 rt = olsr\_alloc\_rt\_entry(&rtp->rtp\_dst);  
 if (!rt) {  
 return;  
 }  
 } else {  
 rt = rt\_tree2rt(node);  
 }  
 /\* Now insert the rt\_path to the owning rt\_entry tree \*/  
 rtp->rtp\_originator = tc->addr;  
 /\* set key and backpointer prior to tree insertion \*/  
 rtp->rtp\_tree\_node.key = &rtp->rtp\_originator;  
 /\* insert to the route entry originator tree \*/  
 avl\_insert(&rt->rt\_path\_tree, &rtp->rtp\_tree\_node, AVL\_DUP\_NO);  
 /\* backlink to the owning route entry \*/  
 rtp->rtp\_rt = rt;  
 /\* update the version field and relevant parameters \*/  
 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。

void  
olsr\_update\_rt\_path(struct rt\_path \*rtp, struct tc\_entry \*tc, struct link\_entry \*link)  
{  
 rtp->rtp\_version = routingtree\_version;  
 /\* gateway \*/  
 rtp->rtp\_nexthop.gateway = link->neighbor\_iface\_addr;  
 /\* interface \*/  
 rtp->rtp\_nexthop.iif\_index = link->inter->if\_index;  
  
 /\* metric/etx \*/  
 rtp->rtp\_metric.hops = tc->hops;  
 rtp->rtp\_metric.cost = tc->path\_cost;  
}

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。

void  
olsr\_delete\_rt\_path(struct rt\_path \*rtp)  
{  
 /\* remove from the originator tree \*/  
 if (rtp->rtp\_rt) {  
 avl\_delete(&rtp->rtp\_rt->rt\_path\_tree, &rtp->rtp\_tree\_node);  
 rtp->rtp\_rt = NULL;  
 }  
 /\* remove from the tc prefix tree \*/  
 if (rtp->rtp\_tc) {  
 avl\_delete(&rtp->rtp\_tc->prefix\_tree, &rtp->rtp\_prefix\_tree\_node);  
 olsr\_unlock\_tc\_entry(rtp->rtp\_tc);  
 rtp->rtp\_tc = NULL;  
 }  
 /\* no current inet gw if the rt\_path is removed \*/  
 if (current\_inetgw == rtp) {  
 current\_inetgw = NULL;  
 }  
 olsr\_cookie\_free(rtp\_mem\_cookie, rtp);  
}

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。

static bool  
olsr\_cmp\_rtp(const struct rt\_path \*rtp1, const struct rt\_path \*rtp2, const struct rt\_path \*inetgw)  
{  
 olsr\_linkcost etx1 = rtp1->rtp\_metric.cost;  
 olsr\_linkcost etx2 = rtp2->rtp\_metric.cost;  
 if (inetgw == rtp1)  
 etx1 \*= olsr\_cnf->lq\_nat\_thresh;  
 if (inetgw == rtp2)  
 etx2 \*= olsr\_cnf->lq\_nat\_thresh;  
 /\* etx comes first \*/  
 if (etx1 < etx2) {  
 return true;  
 }  
 if (etx1 > etx2) {  
 return false;  
 }  
 /\* hopcount is next tie breaker \*/  
 if (rtp1->rtp\_metric.hops < rtp2->rtp\_metric.hops) {  
 return true;  
 }  
 if (rtp1->rtp\_metric.hops > rtp2->rtp\_metric.hops) {  
 return false;  
 }  
 /\* originator (which is guaranteed to be unique) is final tie breaker \*/  
 if (memcmp(&rtp1->rtp\_originator, &rtp2->rtp\_originator, olsr\_cnf->ipsize) < 0) {  
 return true;  
 }  
 return false;  
}

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。

void  
olsr\_rt\_best(struct rt\_entry \*rt)  
{  
 /\* grab the first entry \*/  
 struct avl\_node \*node = avl\_walk\_first(&rt->rt\_path\_tree);  
 assert(node != 0); /\* should not happen \*/  
 rt->rt\_best = rtp\_tree2rtp(node);  
 /\* walk all remaining originator entries \*/  
 while ((node = avl\_walk\_next(node))) {  
 struct rt\_path \*rtp = rtp\_tree2rtp(node);  
 if (olsr\_cmp\_rtp(rtp, rt->rt\_best, current\_inetgw)) {  
 rt->rt\_best = rtp;  
 }  
 }  
 if (0 == rt->rt\_dst.prefix\_len) {  
 current\_inetgw = rt->rt\_best;  
 }  
}

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。

struct rt\_path \*  
olsr\_insert\_routing\_table(union olsr\_ip\_addr \*dst, int plen, union olsr\_ip\_addr \*originator, int origin)  
{  
#ifdef DEBUG  
 struct ipaddr\_str dstbuf, origbuf;  
#endif  
 struct tc\_entry \*tc;  
 struct rt\_path \*rtp;  
 struct avl\_node \*node;  
 struct olsr\_ip\_prefix prefix;  
 /\*  
 \* No bogus prefix lengths.  
 \*/  
 if (plen > olsr\_cnf->maxplen) {  
 return NULL;  
 }  
 /\*  
 \* For all routes we use the tc\_entry as an hookup point.  
 \* If the tc\_entry is disconnected, i.e. has no edges it will not  
 \* be explored during SPF run.  
 \*/  
 tc = olsr\_locate\_tc\_entry(originator);  
 /\*  
 \* first check if there is a rt\_path for the prefix.  
 \*/  
 prefix.prefix = \*dst;  
 prefix.prefix\_len = plen;  
 node = avl\_find(&tc->prefix\_tree, &prefix);  
 if (!node) {  
 /\* no rt\_path for this prefix yet \*/  
 rtp = olsr\_alloc\_rt\_path(tc, &prefix, origin);  
 if (!rtp) {  
 return NULL;  
 }  
#ifdef DEBUG  
 OLSR\_PRINTF(1, "RIB: add prefix %s/%u from %s\n", olsr\_ip\_to\_string(&dstbuf, dst), plen,  
 olsr\_ip\_to\_string(&origbuf, originator));  
#endif  
 /\* overload the hna change bit for flagging a prefix change \*/  
 changes\_hna = true;  
 } else { rtp = rtp\_prefix\_tree2rtp(node);  
 } 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。