**网络协议栈分析与设计课程大作业**

|  |
| --- |
| **OLSR路由协议代码分析** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 学号 | 姓名 | 班级 | 负责模块 | 成绩 | |  |  |  |  |  | |  |  |  |  |  | | 201692231 | 陈雨涵 | 软网1604 | 1.引言：1.1、1.2、1.3路由表计算  2.数据结构：HNA消息、路由表  3.协议概述：路由计算、HNA消息处理 |  | |

目录

[第一章 引言 2](#_Toc534207194)

[1.1 适用性 3](#_Toc534207195)

[1.2 基本原理 3](#_Toc534207196)

[1.3 算法描述 3](#_Toc534207197)

[1.3.1路由表计算 3](#_Toc534207198)

[第二章 代码介绍 4](#_Toc534207199)

[第三章　数据结构 4](#_Toc534207200)

[3.1 HNA消息 4](#_Toc534207201)

[3.2 路由表 5](#_Toc534207202)

[第四章 OLSR路由协议概述 6](#_Toc534207203)

[4.6 路由计算 6](#_Toc534207204)

[4.6.1 相关数据结构 6](#_Toc534207205)

[4.6.2 路由表计算 8](#_Toc534207206)

[4.7 HNA消息处理 17](#_Toc534207207)

[4.7.1 HNA包的生成 17](#_Toc534207208)

[4.7.2 HNA包的发送 18](#_Toc534207209)

[4.7.3 HNA包的处理 19](#_Toc534207210)

[第五章 总结 20](#_Toc534207211)

# 引言

最佳链路状态路由协议（The Optimized Link State Routing Protocol, OLSR）是专门为移动自组网络提出来的一种标准化的先验式的优化链路状态路由协议。它与网络中的其它节点有规律地交换拓扑信息。每个节点选择一系列它们的邻居节点作为它们的多点中继（Multipoint Relay, MPR）节点。在OLSR协议中，只有那些被选为多点中继节点的节点才转发该节点发送的控制分组。多点中继机制提供了一个高效的流量控制方式。

那些被选为多点中继的节点，当在网络中广播链路状态消息时，也有一个特殊的职责。事实上，最佳链路状态路由协议在提供到各目的的最短路径的唯一需求就是多点中继节点向选他们为多点中继节点的节点宣布链路状态信息。此外，剩余的链路状态信息可能已经被利用。

那些被选为MPR节点的节点定期在他们的控制消息中宣布这个信息。同时，一个节点向网络宣布说它对那些被选为MPR节点的节点具有可达性。在路由计算中，多点中继节点被用为形成从给定节点到网络中任意其它节点的路由。此外，协议采用多点中继机制促进在网络中高效地洪泛控制信息。

## 适用性

最佳链路状态路由协议是用于移动自组网络的先验式路由协议。它十分适用于大且密集的移动网络，至于优化的实现，中继节点机制能够很好地适用于移动网络。越大越密集的网络，和传统的链路状态算法相比，优化能达到更大。OLSR采用逐段的路由，比如：每个节点都使用自己的本地信息去路由发包。

OLSR十分适用于那些流量在较大的节点集之间是随机的和零星的，而不是几乎仅在一小组特定的节点之间的网络。作为一个先验式的协议。OLSR也适用于那些通信随时间变化的场景：没有额外的流量控制产生，因为路由是由已知的所有节点时刻维护的。

## 基本原理

## 算法描述

### 1.3.1路由表计算

每个节点维护一张保存到每个目的节点的路由表。路由表是基于本地链路信息库中所包含的信息和拓扑集。因此，任何这些数据集发生了变动，路由表就需要重新计算以更新到网络中各目的节点的路由。路由条目在路由表中按照如下格式被记录：

1. R\_dest\_addr R\_next\_addr R\_dist R\_iface\_addr

2. R\_dest\_addr R\_next\_addr R\_dist R\_iface\_addr

3. ,, ,, ,, ,,

准确地说，在以下这些数据集或库被检测到的情况下下，路由表需要更新：

— 链路集

— 邻居集

— 2跳邻居集

— 拓扑集

— 多接口关联信息库

为了找到从 给定源节点到较远节点R的路径，必须找到连接对（X , R ），然后是找到连接对（Y , X ），以此类推，直到发现节点Y就是源节点的邻居节点时结束连接对的查找。

为了使得到的路径是最优的，转发节点仅选择最小路径上的连接对。路径选择算法在拓扑图的基础上采用了多重Dijkstra算法。

在 OLSR 标准协议中，协议根据最小跳数建立每个节点的路由表。任意一个节点路由表的添加过程可分为三部分：首先，添加自己的邻节点进入路由表，即跳数h=1；其次，添加自己的两跳邻节点进入路由表，即h=2；最后，循环添加跳数等于h+1（h=2开始）的节点进入路由表。不考虑多接口时，具体过程描述如下：

1. 首先清除现有路由表中的所有表项。

2. 将具有对称链路的邻居作为目的地址，一个如下的新的表项加入路由表：

R\_dest\_addr ==邻节点地址

R\_next\_addr ==邻节点地址

R\_dist==1

R\_iface\_addr ==本地地址

3. 对于任意一个两跳邻居节点，其两跳邻居集中必然存在着这样一个表项，其 N\_neighbor\_main\_addr 等于本地的一个一跳邻居且其willingness不为WILL\_NEVER 。则将每个两跳邻居加入路由表如下：

R\_dest\_addr ==两跳邻居地址

R\_next\_addr ==已有的具有如下特征的路由表表项的R\_next\_addr：其R\_dest\_addr为该两跳邻居表项的N\_neighbor\_main\_addr

R\_dist==2

R\_iface\_addr ==已有的具有如下特征的路由表表项的R\_iface\_addr：其R\_dest\_addr 为该两跳邻居表项的N\_neighbor\_main\_addr

4. 将目的地为 h+1 跳的路由表项记录在路由表中，对于h的每个值，下面的程序必须被执行，h从2开始，每次加1，在迭代中没有新表项被记录时，执行停止。对于拓扑集中的每个拓扑表项，如果其T\_dest\_addr不符合于路由表中任何表项的R\_dest\_addr且其T\_dest\_addr符合于一个R\_dist为h的路由表项的R\_dest\_addr，则一个新的路由表项被记录如下：

R\_dest\_addr==T\_dest\_addr

R\_next\_addr==已有的具有如下特征的路由表表项的R\_next\_addr：其 R\_dest\_addr==T\_last\_addr

R\_dist==h+1

R\_iface\_addr==已有的具有如下特征的路由表表项的R\_iface\_addr：其 R\_dest\_addr==T\_last\_addr

计算路由表之后，为节约内存空间，要删除没用于计算的拓扑表中的表项。否则，这些表项将会提供到同一目的节点的多条路由。

# 第二章 代码介绍

# 第三章　数据结构

## HNA消息

包含主机的网络节点需要定期生成HNA消息，并向网络中其它节点发送该消息，说明该网络节点下的主机能否接入到其它网络中。

HNA消息的结构如下图所示：

|  |
| --- |
| Network Address |
| Netmask |
| Network Address |
| Netmask |
| ··· |

每个节点都维护一个信息：哪些节点可能是关联主机和网络的“网关”，并将他们记录为“关联组”。

在HNA消息机制中，通过网络向一个地址序列宣布可达性，显然比向一个个主机地址宣布要来的高效。

TC消息和HNA消息之间的一个重要的区别就是，TC消息可以在之前的信息中含有取消指令，而HNA消息只会在消息失效才被删除。

hna\_set.c

62 int

63 olsr\_init\_hna\_set(void)

64 {

65 int idx;

66

67 for (idx = 0; idx < HASHSIZE; idx++) {

68 hna\_set[idx].next = &hna\_set[idx];

69 hna\_set[idx].prev = &hna\_set[idx];

70 }

71

72 hna\_net\_timer\_cookie = olsr\_alloc\_cookie("HNA Network", OLSR\_COOKIE\_TYPE\_TIMER);

73

74 hna\_net\_mem\_cookie = olsr\_alloc\_cookie("hna\_net", OLSR\_COOKIE\_TYPE\_MEMORY);

75 olsr\_cookie\_set\_memory\_size(hna\_net\_mem\_cookie, sizeof(struct hna\_net));

76

77 hna\_entry\_mem\_cookie = olsr\_alloc\_cookie("hna\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);

78 olsr\_cookie\_set\_memory\_size(hna\_entry\_mem\_cookie, sizeof(struct hna\_entry));

79

80 return 1;

81 }

函数功能：初始化HNA数据集。

72-74:hna\_net\_timer\_cookie记录的是数据包头部的Vtime值，类似的，下面的cookie都是对包内信息的记录。

75-78: olsr\_cookie\_set\_memory\_size记录的是发消息的表，根据HNA消息的接收消息的规则，对于通信过的节点和未通信过的节点通信方式不同。

## 路由表

网络中每个节点维护一个路由表，表中保存了节点到网路中所有可达目的节点的路由，对于路由已知的网络中的每一个目的地，表项被存储在路由表中，所有路由未到达或部分已知的表项不被记录在表中。表项格式如下图所示：

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

R\_dest\_addr：路由目的节点地址

R\_next\_addr：路由的下一跳节点地址

R\_dist：本节点到目的节点的距离

R\_iface\_addr：表示下一跳节点通过本地接口R\_iface\_addr到达

# 第四章 OLSR路由协议概述

## 路由计算

节点通过TC消息的扩散获得全网拓扑图，再根据邻居表、两跳邻居表和拓扑表，独立地按照Dijkstra算法计算出路由表。每个节点都有一张路由表，通过路由表寻找路径信息。对于路由已知的网络中的每个目的地，将路由信息记录在路由表中。所有路由被破坏或仅部分已知的目的地的路由信息不被记录在表中。

每一条路由信息都包含信息目的地址、下一跳地址、总跳数、下一跳接口地址，格式如下：

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

如果该节点所维护的这些信息表中的任何信息发生改变，则重新计算路由表以更新关于网络中的每个目的地的路由信息。即更新路由表的条件为：

(1) 邻居表变化，需要重新选择路由，根据路由计算结果更新路由表。

(2) 两跳邻居表变化，需要重新选择路由，根据路由计算结果更新路由表。

(3) 拓扑表发生变化，需要重新选择路由，根据路由计算结果更新路由表。

路由表的更新既不在网络中，也不在一跳邻居域中生成或触发任何消息。

操作系统的路由体系结构按功能可以分成两个部分。一部分是负责与其它节点交换信息，计算到其它节点的正确路由，称之为“路由功能模块”；一部分则是根据内核路由表，将需要发送到网络中的数据分组，通过正确的网络接口发送到下一跳节点，称为“转发功能模块”。这样，操作系统就可以在“转发功能模块”保持不变的情况下，通过修改“路由功能模块”，从而实现不同的路由协议。

OLSR 协议的实现通过端口号为698的UDP端口收发路由控制分组，然后维护邻居表，进行逻辑计算，最后生成路由表并反映到内核路由表中。数据分组和协议控制分组按照内核路由表中的最佳匹配表项进行发送和转发。当网络中有分组到达本节点时，内核将判断该分组的目的地是否是自己，如果不是，则“转发功能模块”根据内核路由表转发该分组；如果是，则根据分组的不同交给相应的模块进行处理，当收到OLSR协议控制分组时，转由OLSR 路由协议模块处理。

### 相关数据结构

routing\_table.h

66 /\* a composite metric is used for path selection \*/

67 struct rt\_metric {

68 olsr\_linkcost cost;

69 uint32\_t hops;

70 };

71

72 /\* a nexthop is a pointer to a gateway router plus an interface \*/

73 struct rt\_nexthop {

74 union olsr\_ip\_addr gateway; /\* gateway router \*/

75 int iif\_index; /\* outgoing interface index \*/

76 };

67:路由选择时采用复合矩阵。

68:两个路由节点之间的花销。

69:两个路由节点之间的跳数。

74:下一跳网关。

75:接口索引。

routing\_table.h

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 };

85-93: 每一个RIB节点都会有一个路由的接口，这个接口很重要，里面包含了最佳路径的下一个网关信息，而且是rt\_path\_tree的根，同样也包含了一个在所有路由信息里的一个最好的路径。rt\_dst包含了该信息的路由地址与前缀长度。rt\_tree\_node包含了val\_tree的一些信息。该结构体里面也包含了父节点、子节点、左右节点、表示一棵树的信息值。

routing\_table.h

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 HNA \*/

117 };

106-117: 该结构体包含了rt\_path的信息，接收到的rt\_path会被加入到路由信息表中，然后用于计算best\_path。

routing\_table.h

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 };

182-194: 核心路由表的结构。核心路由表表项的添加和删除主要受目的地、网关（下一跳）和标志设置的影响。标志位的设置非常重要，决定着传输数据时，添加的路由表项是否发挥作用。

routing\_table.h

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协议中有三种路由类型，INT（内部路由信息）是由接收TC\_Message生成的，而MID类型的路由信息是由接收MID\_Message生成的，HNA路由信息是来自HNA公告。

### 路由表计算

routing\_table.c

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 }

函数功能：初始化路由表（初始化平衡二叉树）。

173:调用avl\_init( )初始化一个平衡二叉树。

174:用以检测每一个rt\_entry和rt\_path子树中过时的信息。

179-183: 为rt\_entry和rt\_path分配内存，创建相应的cookie。

routing\_table.c

194 struct rt\_entry \*

195 olsr\_lookup\_routing\_table(const union olsr\_ip\_addr \*dst)

196 {

197 struct avl\_node \*rt\_tree\_node;

198 struct olsr\_ip\_prefix prefix;

199

200 prefix.prefix = \*dst;

201 prefix.prefix\_len = olsr\_cnf->maxplen;

202

203 rt\_tree\_node = avl\_find(&routingtree, &prefix);

204

205 return rt\_tree\_node ? rt\_tree2rt(rt\_tree\_node) : NULL;

206 }

函数功能：从avl树里面找到一个地址的路由表条目，根据参数地址并配上设置里的最长前缀长度。在avl树里调用avl\_find函数从routingtree里面找到该地址的rt\_entry。

routing\_table.c

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;

577 }

546:先调用olsr\_locate\_tc\_entry函数根据源地址判断该tc\_entry是否可连接，因为tc\_entry作为所有路由的连接点，如果它是不可连接的，则在最短路径优先计算时它不会被考虑。

551-563: 首先检查该前缀是否已经存在一条路径。如果没有的话，则调用olsr\_alloc\_rt\_path函数创建一条该前缀和源地址的路径。

routing\_table.c

582 void

583 olsr\_delete\_routing\_table(union olsr\_ip\_addr \*dst, int plen, union olsr\_ip\_addr \*originator)

584 {

585 #ifdef DEBUG

586 struct ipaddr\_str dstbuf, origbuf;

587 #endif

588

589 struct tc\_entry \*tc;

590 struct rt\_path \*rtp;

591 struct avl\_node \*node;

592 struct olsr\_ip\_prefix prefix;

593

594 /\*

595 \* No bogus prefix lengths.

596 \*/

597 if (plen > olsr\_cnf->maxplen) {

598 return;

599 }

600

601 tc = olsr\_lookup\_tc\_entry(originator);

602 if (!tc) {

603 return;

604 }

605

606 /\*

607 \* Grab the rt\_path for the prefix.

608 \*/

609 prefix.prefix = \*dst;

610 prefix.prefix\_len = plen;

611

612 node = avl\_find(&tc->prefix\_tree, &prefix);

613

614 if (node) {

615 rtp = rtp\_prefix\_tree2rtp(node);

616 olsr\_delete\_rt\_path(rtp);

617

618 #ifdef DEBUG

619 OLSR\_PRINTF(1, "RIB: del prefix %s/%u from %s\n", olsr\_ip\_to\_string(&dstbuf, dst), plen,

620 olsr\_ip\_to\_string(&origbuf, originator));

621 #endif

622

623 /\* overload the hna change bit for flagging a prefix change \*/

624 changes\_hna = true;

625 }

626 }

函数功能：从字典树中删除前缀，并重载HNA消息。

routing\_table.c

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 }

函数功能：此函数用于为一条给定的rt\_path创建一个route entry，并将其插入到RIB树中和计算路由表。

215:每条路径都是周期性更新的，修改所维护的routingtree\_version值。

217-225:将相应的网关地址、接口地址、跳数和路径花销置为新接收到的值。

routing\_table.c

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 }

函数功能：创建一个可用的路由条目，对提供的参数ip前缀分配一个路由条目空间，并做一些相应的初始化，并把该条目插入到avl树里。

234-237:申请内存空间并把空间内清零。

241-245:表示该入口为新分配入口，并把该入口的目的地址设置成为参数提供的入口地址。

247-251:把入口的树节点插入到整个路由表中，并初始化avl树。

routing\_table.c

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);

344 }

函数功能：对给定的rt\_path创建一个路由条目，并将它插入到全局RIB树中。

301-310:如果传入的tc\_entry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，则直接返回。

315-328:调用avl\_find( )函数检查传入的rtp的节点是否在路由表中，如果节点不在路由表的avl树中则重新分配一个节点，如果在就把节点类型从avl\_node类型转化成rt\_entry类型。

330-343:把新路由条目添加进avl树里，改变相应的参数，更新路径。

routing\_table.c

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 }

函数功能：从路由表中删除rtp树，并把它从tc树里删除，然后改变路由表版本。

354-357:把rtp所指向的树节点从所在的树里删除，并把指向的树的根置空。

360-364:将其从字典树（前缀树）中移除，并解锁相应的tc\_entry。

367-372:把rtp所指向的树节点再从tc树里删除，释放cookie所占用的内存。

routing\_table.c

435 static bool

436 olsr\_cmp\_rtp(const struct rt\_path \*rtp1, const struct rt\_path \*rtp2, const struct rt\_path \*i netgw)

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 }

函数功能：比较两条路由路径，如果第一条更优，则返回true，否则返回false。

445-451:比较两个路由路径的花销，花销小的路径更优。

453-459:如果两个路由路径花销相同，则比较两条路径的跳数，跳数少的更优。

461-464:如果两个路由路径的花销和跳数都相同，则比较两条路由的源地址，地址更小的更优。

routing\_table.c

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 }

函数功能：运行最优路径，首先得到第一条目，然后遍历所有条目，找到一条最优路径并修改当前网关路径到最佳路径。

488-493: 通过调用avl\_walk\_first函数从rt-> rt\_path\_tress里找到树里的第一个条目并保证不为零，然后把节点类型转化为rt\_entry类型。

495-506: 历整棵树并比较当前路径与最佳路径，如果当前路径比最佳路径还要好则把当前路径记录为最佳路径。

路由计算过程：路由节点启动后，首先执行对相邻节点的发现工作，并获取它们的地址。然后路由节点主动测试到每一个相邻路由器的链路时延或成本，并根据测试结果设置相关链路的状态。然后路由节点构造自己的链路状态信息包，链路状态信息的内容包括本路由器的标号、本路由器的邻居路由器列表、本路由器到各邻居路由器的链路状态（时延或成本）、该链路状态信息包的序号和生存时间等。然后该路由节点向所有参与链路状态交互的路由节点广播其链路状态信息，可以是周期性地发送，也可在链路状态发生变化时发送。每个路由节点在收到所有的链路状态信息后，可以构造或更新表示整个网络拓扑结构的图G（V, E），顶点V表示路由器，边E表示连接路由器的链路。这时路由节点就可以用 Dijkstra 算法从图G中计算出到所有目的路由器的最短路径，也就是构造以自己为根节点的最短路径优先树。

## HNA消息处理

### HNA包的生成

在OLSR网络中，应用HNA消息的节点需周期性的生成HNA消息，该消息内容应包括该节点的IP地址及子网掩码，而发包时周期则根据该节点的Vtime来确定。

从该函数中可以看出，节点生成的HNA消息包被存放进一个队列中，并打上时间戳，对于未过期的HNA消息，将其置入缓冲区中，等待发送。

generate\_msg.c

107 void

108 generate\_hna(void \*p)

109 {

110 struct interface \*ifn = (struct interface \*)p;

111

112 if (queue\_hna(ifn) && TIMED\_OUT(ifn->fwdtimer)) {

113 set\_buffer\_timer(ifn);

114 }

115 }

112-114: 生成HNA消息的节点将HNA消息放在队列中，当读取到该消息时，判断该消息是否过期，如未过期，则读取消息内容。

### HNA包的发送

一般情况下，HNA消息生成的包为OLSR标准包，所以发送HNA运用的是OLSR协议的洪泛发包方式。下图所示为HNA消息的发送函数，从函数中可以看到，HNA消息发送时在消息中记录了初始化过程中的vtime，msgtype，msgsize等数据，而hop\_count则用于接收方进行判断消息跳数，接收方借此来判断如何处理这条HNA消息。

hna\_set.c

367 bool

368 olsr\_input\_hna(union olsr\_message \*m, struct interface \*in\_if \_\_attribute\_\_ ((unused)), unio n olsr\_ip\_addr \*from\_addr)

369 {

370

371 uint8\_t olsr\_msgtype;

372 olsr\_reltime vtime;

373 uint16\_t olsr\_msgsize;

374 union olsr\_ip\_addr originator;

375 uint8\_t hop\_count;

376 uint16\_t msg\_seq\_number;

函数中的一些判断是在发消息之前确保该节点所发的节点为标准的HNA消息包，确保该消息符合OLSR协议的规范。

hna\_set.c

386 /\* Check if everyting is ok \*/

387 if (!m) {

388 return false;

389 }

390 curr = (const uint8\_t \*)m;

391

392 /\* olsr\_msgtype \*/

393 pkt\_get\_u8(&curr, &olsr\_msgtype);

394 if (olsr\_msgtype != HNA\_MESSAGE) {

395 OLSR\_PRINTF(1, "not a HNA message!\n");

396 return false;

397 }

398 /\* Get vtime \*/

399 pkt\_get\_reltime(&curr, &vtime);

400

401 /\* olsr\_msgsize \*/

402 pkt\_get\_u16(&curr, &olsr\_msgsize);

403

404 hnasize = olsr\_msgsize - 8 - olsr\_cnf->ipsize;

405 curr\_end = (const uint8\_t \*)m + olsr\_msgsize;

406

407 /\* validate originator \*/

408 pkt\_get\_ipaddress(&curr, &originator);

409 /\*printf("HNA from %s\n\n", olsr\_ip\_to\_string(&buf, &originator)); \*/

410

411 /\* ttl \*/

412 pkt\_ignore\_u8(&curr);

413

414 /\* hopcnt \*/

415 pkt\_get\_u8(&curr, &hop\_count);

416

417 /\* seqno \*/

418 pkt\_get\_u16(&curr, &msg\_seq\_number);

392-397: 检测该HNA消息包是否符合OLSR消息包的规范，如规范则可继续发送，如不符合规范，则返回false，不发送包。

398-399:调用pkt\_get\_reltime( )函数，获取vtime。

407-409:验证发包源地址的有效性。

### HNA包的处理

位于OLSR协议中的每个节点持续的对所接收的消息头进行判断，当检测到所需的消息是，就读取该消息。每个节点都会多次接收到同一消息，为了避免因读取重复消息而造成资源浪费，每个节点都维持了一个防重读表。在该表中，节点自动记录近期接收到的消息，并将新接受的消息与该表中的信息进行对比，如果新消息不在改防重读表中，才会读取新消息的具体内容。

每一个包含HNA包的消息的有效时间都是根据包含在该包头部的Vtime数据来计算的，当一个节点读取一个HNA消息包时，所经历的具体步骤如下：

1 若发送HNA包的根节点与接收该包的节点的距离超过一跳，则丢弃这个包；

2 所发送HNA包的根节点与接收该包的节点的距离不超过一跳，读取这个包的源IP与子网掩码；

3 若之前曾与该节点通信，则将该元组的保存时间设为现有时间与有效时间之和；

4 若之前不曾与该节点通信，则需初始化该元组的数据，初始化内容如下：

A\_gateway\_addr = originator address

A\_network\_addr = network address

A\_netmask = netmask

A\_time = current time + validity time

# 第五章 总结