数据结构部分

#### OLSR包格式

OLSR使用统一的数据包格式与所有与协议相关的数据进行通信。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。这还提供了将不同“类型”信息装配到到一条消息中的简单方式，因此优化了传送效率最大化利用由网络提供的最大帧大小。这些数据包在网络传输中嵌入在UDP数据报中。在本文分析中的网络地址为IPv4格式。

每个数据包可以封装一个或多个消息。这些消息共享一个通用的头格式，使收到包的节点能够正确接受和重传未知类型的消息。

  消息可以泛洪到整个网络上，或者洪泛可以限制在消息发送结点的特定范围（就跳数而言）内的节点中传播。因此，将消息发送到节点的邻域只是泛洪的特殊情况。当泛洪任何控制消息时，会暂时的取消重传机制（即，每个节点维护一个重复集以防止发送两次相同的OLSR控制消息）并且通过使用MPR机制在整个网络中保持泛洪最小化性能最优化，MPR选择机制在后面部分中所述。

下面将介绍OLSR包的格式

1 2 3 4

|  |  |  |
| --- | --- | --- |
| Packet Length | | Packet Sequence Number |
| Message Type | Vtime | Message Size |
| Originator Address | | |
| Time to Live | Hop Count | Message Sequence Number |
| MESSAGE | | |
| Message Type | Vtime | Message Size |
| Originator Address | | |
| Time to Live | Hop Count | Message Sequence Number |
| MESSAGE | | |

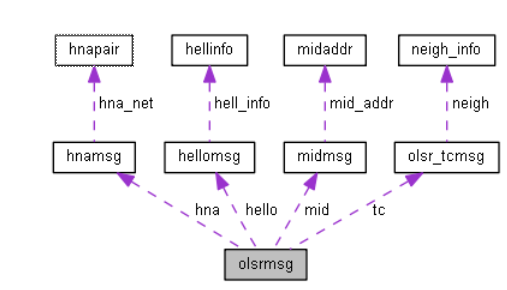
各项介绍:

* Packet Length：包的长度（字节）
* PacketSequenceNumber:（PSN）包的序列号 PSN在每生成一个新的包后会自动增加。该数值由每个结点的接口维护。需要说明的是，当一个结点收到一个大小小于等于OLSR包首部的消息，该消息会被结点认为是一个无效的消息将其抛弃，对于IPv4来说，当PacketLength小于16 会被抛弃。
* MessageType：该项表明在Message 部分中是什么样的信息，该值取值范围为0~127。
* Vtime 在一个结点收到该消息后，认为是一个合法消息的持续时间，由结点根据其值进行不同的计算方法得出时间。
* Message Size：消息的大小
* TTL：Time to live
* Hop Count：当前消息的跳数
* Originator Address： 该项包含了生成当前消息的源节点的地址
* Message Sequence Number：当生成一条消息时，源节点会为每条消息指定一个特定唯一的数。

1. /\*
2. \* OLSR message (several can exist in one OLSR packet)
3. \*/
5. **struct** olsrmsg {
6. uint8\_t olsr\_msgtype;
7. uint8\_t olsr\_vtime;
8. uint16\_t olsr\_msgsize;
9. uint32\_t originator;
10. uint8\_t ttl;
11. uint8\_t hopcnt;
12. uint16\_t seqno;
14. **union** {
15. **struct** hellomsg hello;
16. **struct** olsr\_tcmsg tc;
17. **struct** hnamsg hna;
18. **struct** midmsg mid;
19. } message;
21. } \_\_attribute\_\_ ((packed));

olsr\_protocol.h

上述是OLSR定义的OLSR包结构体变量，这里各项代表的含义和上述一样，就不在做表述。画出组织图为：



## Hello消息格式

在路由器上激活了OLSR协议之后，只有通过路由协议消息“探索”到了邻居路由器，才能与其建立邻接关系。运行OLSR协议的路由器会同时发送并侦听Hello消息。Hello协议消息所起的作用如下所列：

链路感知，邻居探测和MPR集的选择。

下图为OLSR Hello消息的格式：

|  |  |  |  |
| --- | --- | --- | --- |
| Reserved | | Htime | Willingness |
| Link Code | Reserved | Link Message Size | |
| Neighbor Interface Address | | | |
| Neighbor Interface Address | | | |

需要说明的是，在hello消息中消息首部中的数据已经确定好了：MessageType设为：HELLO\_MESSAGE, TTL：被设置为1。

* Reserved: 永远被设置为“0000000000000”
* Htime： Hello消息发送的间隔时间，根据首部的Vtime得到，其计算公式为：emission interval=C\*(1+a/16)\*2^b，在这里a是OLSR首部的Vtime的高四位，b是Vtime的低四位，C是一个特殊的数字
* Willgnes： 表明结点的成为MPR结点的程度

WILL\_NEVER 不会被选择为MPR结点

WILL\_ALWAYS 总是会被选择为MPR结点

* Link\_code: 该项是一个8比特的数据项，其值永远<15,当一个消息中的Link\_code<15时，代表其具有2个域：0~1位代表链路类型，2~3位代表邻居类型。OLSR中的链路类型有4种：不确定链路UNSPEC\_LINK（0），非对称链路ASYM\_LINK（1），对称链路SYM\_LINK（2），丢失链路LOST\_LINK（3）。3个邻居类型：非邻居NOT\_NEIGH（0），对称邻居SYM\_NEIGH（1），MPR邻居MPR\_NEIGH（2）。
* Neighbor Interface Address: 源节点的邻居结点的地址

关于Link\_code有一点需要说明的是，邻居类型和链路类型不是随意搭配的，需要符合一定的逻辑，比如在link\_code中，链路类型为SYM\_LINK而邻居类型为NOT\_NEIGH就是不合法的，在收到后会直接被丢弃。

在代码中，Hello消息实现如下：

1. **struct** hellinfo {
2. uint8\_t link\_code;
3. uint8\_t reserved;
4. uint16\_t size;
5. uint32\_t neigh\_addr[1];              /\* neighbor IP address(es) \*/
6. } \_\_attribute\_\_ ((packed));
8. **struct** hellomsg {
9. uint16\_t reserved;
10. uint8\_t htime;
11. uint8\_t willingness;
12. **struct** hellinfo hell\_info[1];
13. } \_\_attribute\_\_ ((packed));

olsr\_protocol.h

在packet.h中的hello消息的组织图如下：

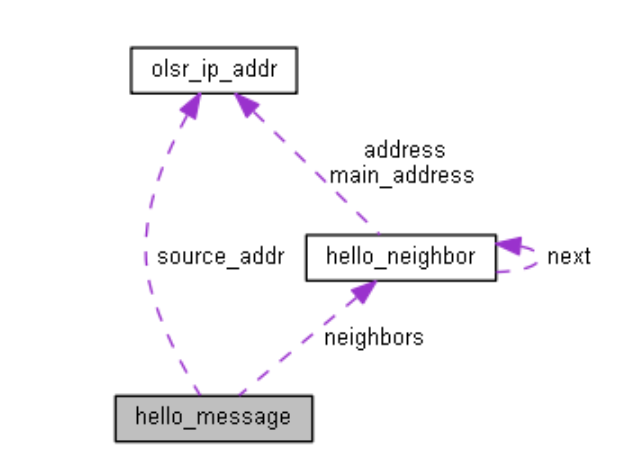


图. Hello消息与其他结构的关系

## 结点保存的信息表

链路信息结构：

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

* L\_local\_iface\_addr ：本地节点的接口地址。
* L\_neighbor\_iface\_addr邻节点的接口地址。
* L\_SYM\_time：直到此时刻前，链路被认为是对称的。
* L\_ASYM\_time：直到此时刻前，链路被认为是单向的。
* L\_time：链路维护时刻，链路在该时刻失效，必须被删除，当 L\_SYM\_time和 L\_ASYM\_time 都过期，链路被声明为丢失。

## 邻居集信息表

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

该信息表表示该结点的一条一跳邻居信息。

* + N\_neighbor\_main\_addrL: 表示一跳邻居的主地址
  + N\_status:表示邻居的类型，在OLSR里取值有三种分别是MPR\_NEIGH,SYM\_NEIGH,ASYM\_NEIGH
  + N\_willingnes:表示邻居结点愿意为邻居结点转发分组的愿意成都，如果为WILL\_ALWAYS,那么该结点一定会被其他结点选择为MPR结点

## 二跳邻居表

结点存储一个二跳邻居表，二跳邻居指的是与本结点的对称邻居有一条对称链路的结点（除本结点），表结构如下：

|  |  |  |
| --- | --- | --- |
| N\_neighbor\_main\_addr | N \_2hop \_addr | N \_time |

* N\_neighbor\_main\_addr：表示邻节点的地址。
* N\_2hop\_addr ：表示与 N\_neighbor\_main\_addr 有对称链路的两跳邻节点的地址。
* N\_time ：表示表项到期必须被移除的时间。

## 拓扑信息表

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

每个结点都会维护一张拓扑表，表中记录了从TC消息中获得网络拓扑信息。

T\_dest\_addr: MPR选择结点的地址，表示该结点已经选择结点T\_last作为MPR

T\_last: 被T\_dest选为MPR结点的结点地址

T\_seq：表示T\_last已经发布了它保存的序列号为T\_seq的MPR Selector集合的控制信息。

T\_time: 表项的保持时间，过期后就失效

## Tc消息

拓扑控制（TC）消息的洪泛具有十分重要的意义。在使用 OLSR 协议的网络中，每个节点都周期性发送TC 分组。当一个节点接收到TC 消息时，就进入拓扑信息维护模块。收到TC 消息必要时进行转发（实现拓扑泛洪）；如果得到网络中一条有效的链路（拓扑），则将其添加到拓扑表中，用以计算路由。当检测到拓扑表发生变化时，就要转到路由建立与维护模块，重新计算路由。当收到数据包时，对其进行转发。



图 结点和它的MPR结点

如图所示，节点 B、C、D和 E 是节点 A的邻居节点，其中节点 B、C和D又是节点A的 MPR节点。当节点 B、C、D和E 收到节点A发送来的TC 控制消息时首先，它们都需要判断自己是不是节点 A的 MPR节点。节点 B、C和D 发现自己是节点A的 MPR 节点，它们再判断TC控制消息是否是最新的，如果是，则转发该TC 控制消息，否则丢弃该TC控制消息；而节点E发现自己不是节点A的MPR节点，则不转发该TC控制消息。

1 2 3 4

|  |  |
| --- | --- |
| ANSN | Reserved |
| Advertised Neighbor Main Address | |
| Advertised Neighbor Main Address | |
| … | |

上面是TC消息格式，在消息首部MessageType中为TC\_MESSAGE，消息首部的Time应该设为255（最大值）目的是为了让消息能够在网络中更充分的传播。

* Advertised Neighbor Sequence Number (ANSN)：序列号。与多点中继 MPR Selector 集相对应的序列号，每当节点检测到 MPR Selector集发生变化时，就增加该序列号的值。节点收到TC 分组时，根据 ASSN ，决定有关发送者的 MPR Selector 的信息是否比已有的要新。
* Advertised Neighbor Main Address：这个域里包含邻居结点的主地址。所有源结点的邻居的主地址都会被放到TC消息的这个域中。如果当前的消息已经超过了消息的最大长度而仍然有地址未被放入消息中，那么会生成另外一个TC消息来装载剩余的地址。
* Reserved：在这里是一个固定的数字，“0000000000000000”

下面我们看看代码中的实现：

1. **struct** tc\_message {
2. olsr\_reltime vtime;
3. **union** olsr\_ip\_addr source\_addr;
4. **union** olsr\_ip\_addr originator;
5. uint16\_t packet\_seq\_number;
6. uint8\_t hop\_count;
7. uint8\_t ttl;
8. uint16\_t ansn;
9. **struct** tc\_mpr\_addr \*multipoint\_relay\_selector\_address;
10. };

Packet.h

上述代码是一个完整的TC消息，包含了消息类型和Vtime。

下面是TC消息与其他的结构的关系图：

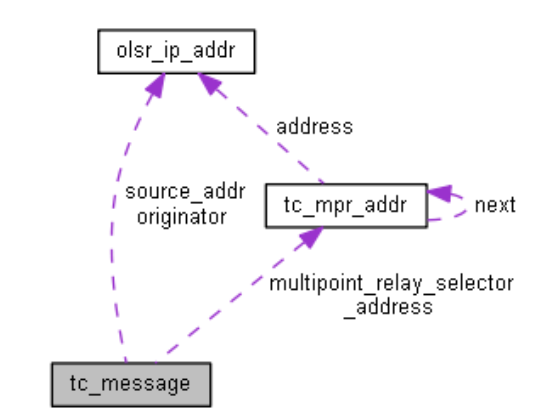


图. TC消息的关系图

## 链路感知：

在介绍链路感知之前首先我们需要明确一下一条链路应当包含哪些信息，下面是结点存储的关于链路的信息。

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

链路感知过程就是完善一个结点链路信息的过程。链路感知的原理是Hello消息的周期交换。通过hello信息的周期性交换，对每个结点的链路信息集进行更新。

我们该怎样确定一条链路的具体状态呢：具体规则如下：

* 如果Link\_SYM\_time >= current Time(not expired)，那么就设定该链路为SYM\_LINK状态
* 如果上述情况不成立，如果 L\_ASYM\_time >= current time (not expired) 而且L\_SYM\_time < current time (expired)那么设定链路状态为：ASYM\_LINK状态
* 如果上述情况不成立，那么如果L\_ASYM\_time < current time (expired) 而且 L\_SYM\_time < current time (expired)，那么设定链路状态为 Link Type = LOST\_LINK

链路感知的目的是使链路有一个对应的状态：对称或者非对称。对称意味着该从该接口到邻居接口的链路已经被核实为是双向的，以为它们可以在两个方向上传递信息。非对称的链路表示hello消息只能在一个方向上被听到，意思就是从邻居接口到本接口传递消息是可能的，而从本结点到邻居结点却不一定可行。

为了探测到从接口到邻居接口的链路，一个结点路由器节点应该对它的所有接口都进行链路感知。而且，一个结点必须在所有接口内广播他的所有一跳范围内是对称状态邻居，目的也是为了探测邻居。因此，一条hello消息应该包含该接口的链路列表（包含链路类型）和整个邻居列表（包含邻居类型），

链路感知部分主要由四个部分的代码完成，如下图所示：

check\_link\_status()

add\_link\_entry()

lookup\_link\_entry()

update\_link\_entry()

Check\_link\_status()函数通过查找收到的Hello消息查看链路的状态，add\_link\_entry()函数为链路集中添加一条新的链路信息，lookup\_link\_entry()根据指定的IP地址在链路集中查找一条链路信息，update\_link\_entry()根据收到的hello消息更新新的链路信息。下面我们来看看链路感知机制是怎样的：

1. **struct** link\_entry \*
2. update\_link\_entry(**const** **union** olsr\_ip\_addr \*local, **const** **union** olsr\_ip\_addr \*remote, **const** **struct** hello\_message \*message,
3. **const** **struct** interface \*in\_if)
4. {
5. **struct** link\_entry \*entry;
6. /\* Add if not registered \*/
7. entry = add\_link\_entry(local, remote, &message->source\_addr, message->vtime, message->htime, in\_if);
8. /\* Update ASYM\_time \*/
9. entry->vtime = message->vtime;
10. entry->ASYM\_time = GET\_TIMESTAMP(message->vtime);
11. entry->prev\_status = check\_link\_status(message, in\_if);
12. **switch** (entry->prev\_status) {
13. **case** (LOST\_LINK):
14. olsr\_stop\_timer(entry->link\_sym\_timer);
15. entry->link\_sym\_timer = NULL;
16. **break**;
17. **case** (SYM\_LINK):
18. **case** (ASYM\_LINK):
19. /\* L\_SYM\_time = current time + validity time \*/
20. olsr\_set\_timer(&entry->link\_sym\_timer, message->vtime, OLSR\_LINK\_SYM\_JITTER, OLSR\_TIMER\_ONESHOT, &olsr\_expire\_link\_sym\_timer,
21. entry, 0);
22. /\* L\_time = L\_SYM\_time + NEIGHB\_HOLD\_TIME \*/
23. olsr\_set\_link\_timer(entry, message->vtime + NEIGHB\_HOLD\_TIME \* MSEC\_PER\_SEC);
24. **break**;
25. **default**:;
26. }
27. /\* L\_time = max(L\_time, L\_ASYM\_time) \*/
28. **if** (entry->link\_timer && (entry->link\_timer->timer\_clock < entry->ASYM\_time)) {
29. olsr\_set\_link\_timer(entry, TIME\_DUE(entry->ASYM\_time));
30. }
31. /\* Update hysteresis values \*/
32. **if** (olsr\_cnf->use\_hysteresis)
33. olsr\_process\_hysteresis(entry);
34. /\* Update neighbor \*/
35. update\_neighbor\_status(entry->neighbor, get\_neighbor\_status(remote));
36. **return** entry;
37. }

link\_set.c

7：通过给定的信息查找链路，如果存在返回该链路；如果不存在则新建立一

条链路并返回

9-11：用刚刚收到的hello消息更新之前返回的链路的信息，如果找到一条匹配的链路信息那就把这条链路信息修改为：

L\_ASYM\_time = current time + validity time;

12-26：如果该结点发现收到Hello消息的接口地址在hello消息的地址列表内，那么做出如下的修改：

如果链路类型是LOST\_LINK类型那么

L\_SYM\_time = current time - 1 (i.e., expired)

或者链路类型为：SYM\_LINK 或者ASYM\_LINK类型那么

L\_SYM\_time = current time + validity time,

L\_time = L\_SYM\_time + NEIGHB\_HOLD\_TIME

1. **static** **struct** link\_entry \*
2. add\_link\_entry(**const** **union** olsr\_ip\_addr \*local, **const** **union** olsr\_ip\_addr \*remote, **const** **union** olsr\_ip\_addr \*remote\_main,
3. olsr\_reltime vtime, olsr\_reltime htime, **const** **struct** interface \*local\_if)
4. {
5. **struct** link\_entry \*new\_link;
6. **struct** neighbor\_entry \*neighbor;
7. **struct** link\_entry \*tmp\_link\_set;
9. tmp\_link\_set = lookup\_link\_entry(remote, remote\_main, local\_if);
10. **if** (tmp\_link\_set) {
11. **return** tmp\_link\_set;
12. }
13. #ifdef DEBUG
14. {
15. **struct** ipaddr\_str localbuf, rembuf;
16. OLSR\_PRINTF(1, "Adding %s=>%s to link set\n", olsr\_ip\_to\_string(&localbuf, local), olsr\_ip\_to\_string(&rembuf, remote));
17. }
18. #endif
19. /\* a new tuple is created with... \*/
20. new\_link = olsr\_malloc\_link\_entry("new link entry");
21. /\* copy if\_name, if it is defined \*/
22. **if** (local\_if->int\_name) {
23. **size\_t** name\_size = strlen(local\_if->int\_name) + 1;
24. new\_link->if\_name = olsr\_malloc(name\_size, "target of if\_name in new link entry");
25. strscpy(new\_link->if\_name, local\_if->int\_name, name\_size);
26. } **else**
27. new\_link->if\_name = NULL;
29. /\* shortcut to interface. XXX refcount \*/
30. new\_link->inter = local\_if;
31. /\*
32. \* L\_local\_iface\_addr = Address of the interface
33. \* which received the HELLO message
34. \*/
35. new\_link->local\_iface\_addr = \*local;
36. /\* L\_neighbor\_iface\_addr = Source Address \*/
37. new\_link->neighbor\_iface\_addr = \*remote;
38. /\* L\_time = current time + validity time \*/
39. olsr\_set\_link\_timer(new\_link, vtime);
40. new\_link->prev\_status = ASYM\_LINK;

link\_set.c

10-11 如果在链路集合中找到链路，则返回

29-40 如果没有一条链路信息为L\_neighbor\_iface\_addr = Source Address

那么就建立一条新的链路信息

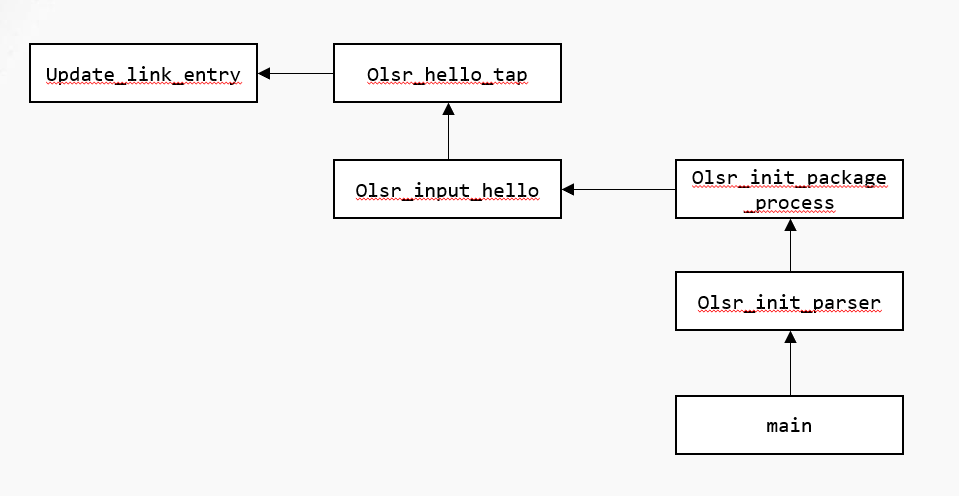
L\_neighbor\_iface\_addr = Source Address

L\_local\_iface\_addr = Address of the interface which received the HELLO message

L\_SYM\_time = current time - 1 (expired)

L\_time = current time + validity time

下面我们再看看关于update函数的调用模型：



我们可以看到在链路感知的发生过程：当收到hello消息后，首先，实现计算该Hello消息的间隔时间，通过OLSR消息首部的VTIME域计算出来之后。包调用olsr\_input\_hello函数来处理hello消息，然后调用Update\_link\_entry函数来更新链路中的信息，也就是说Update\_link\_entry是链路感知的主入口，通过收到邻居结点周期性发来的Hello消息，首先先调用add\_link\_entry来查找链路集中是否包含包含特定信息的链路，如果存在则返回该链路，如果不存在就创建一条新的链路。然后根据hello消息中的信息以及链路先前的状态，更新链路的特定数据。通过周期性的Hello信息交互，结点的链路信息也就得到了更新。

## 邻居探测

下面是结点存储的关于邻居集的信息。下面是一条存储的邻居信息，各项含义都已经在上述数据机构部分已经描述过了

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

邻居探测的过程就是一个填充邻居集信息的过程，其机制也是周期性的交换Hello消息。结点会根据它的链路集信息维护一个邻居集信息。邻居集根据链路集中的发生的变化进行信息更新。链路集保存链路的相关信息，而邻居集保存着关于邻居的信息。这两个集合之间有着至关重要的联系，那是因为一个结点是另一个结点的邻居当且仅当在这两个结点之间有至少一条链路。

在任何情况下，链路与邻居之间的对应关系如下定义：

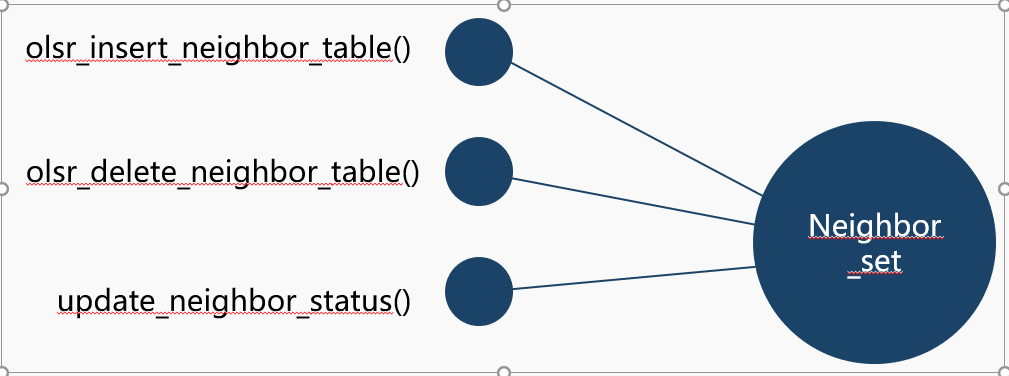
一条链路集合的有关邻居需要满足的是

N\_neighbor\_main\_addr == main address of L\_neighbor\_iface\_addr

一个邻居的相关链路需要满足：

N\_neighbor\_main\_addr == main address of L\_neighbor\_iface\_addr

邻居检测主要由三个部分组成：创建邻居，更新邻居，移除邻居。其对应的函数如下



其中olsr\_insert\_neighbor\_table()是负责建立邻居的函数，olsr\_delete\_neighbor\_table()是负责删除已经没有之间没有链路的邻居的函数，update\_neighbor\_status()负责更新邻居的状态

1. **struct** neighbor\_entry \*
2. olsr\_insert\_neighbor\_table(**const** **union** olsr\_ip\_addr \*main\_addr)
3. {
4. uint32\_t hash;
5. **struct** neighbor\_entry \*new\_neigh;
7. hash = olsr\_ip\_hashing(main\_addr);
9. /\* Check if entry exists \*/
11. **for** (new\_neigh = neighbortable[hash].next; new\_neigh != &neighbortable[hash]; new\_neigh = new\_neigh->next) {
12. **if** (ipequal(&new\_neigh->neighbor\_main\_addr, main\_addr))
13. **return** new\_neigh;
14. }
16. //printf("inserting neighbor\n");
18. new\_neigh = olsr\_malloc(**sizeof**(**struct** neighbor\_entry), "New neighbor entry");
20. /\* Set address, willingness and status \*/
21. new\_neigh->neighbor\_main\_addr = \*main\_addr;
22. new\_neigh->willingness = WILL\_NEVER;
23. new\_neigh->status = NOT\_SYM;
25. new\_neigh->neighbor\_2\_list.next = &new\_neigh->neighbor\_2\_list;
26. new\_neigh->neighbor\_2\_list.prev = &new\_neigh->neighbor\_2\_list;
28. new\_neigh->linkcount = 0;
29. new\_neigh->is\_mpr = **false**;
30. new\_neigh->was\_mpr = **false**;
32. /\* Queue \*/
33. QUEUE\_ELEM(neighbortable[hash], new\_neigh);
35. **return** new\_neigh;
36. }

Neighbor\_table.c

11-13 在邻居集中查找是否有满足下列条件的邻居：

N\_neighbor\_main\_addr = main address of L\_neighbor\_iface\_addr

如果有就直接返回，如果没有的话就会创建出一个新的邻居元组

21-30 建立一个新的邻居元组，并对其各项进行赋予初值

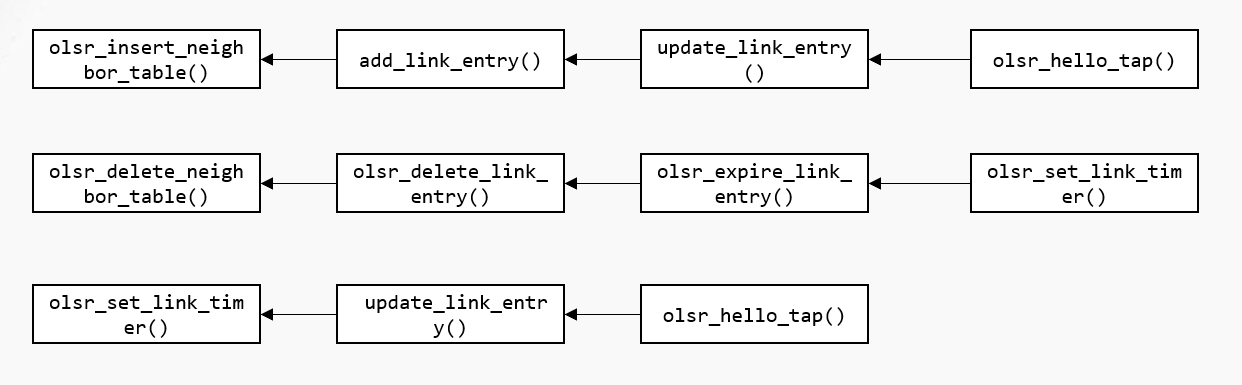
1. olsr\_delete\_link\_entry(**struct** link\_entry \*link)
2. {
3. **struct** tc\_edge\_entry \*tc\_edge;
5. /\* delete tc edges we made for SPF \*/
6. tc\_edge = olsr\_lookup\_tc\_edge(tc\_myself, &link->neighbor\_iface\_addr);
7. **if** (tc\_edge != NULL) {
8. olsr\_delete\_tc\_edge\_entry(tc\_edge);
9. }

12. /\* Delete neighbor entry \*/
13. **if** (link->neighbor->linkcount == 1) {
14. olsr\_delete\_neighbor\_table(&link->neighbor->neighbor\_main\_addr);
15. } **else** {
16. link->neighbor->linkcount--;
17. }

Link\_set.c

13-16 计算待删除邻居的链路数量，如果链路数量等于1那么调用 olsr\_delete\_neighbor\_table函数从邻居集中将该邻居信息删除，如果不等于1那么，将邻居的链路数减1

关于邻居探测涉及到的函数调用如下所示



我们可以看到邻居探测机制与链路感知机制是同时进行的，在链路感知进行链路信息的更新后，OLSR会根据链路更新后的信息更新结点的邻居集信息，这也从侧面说明了链路集与邻居集的关系非常密切。

下面简单介绍一下关于二跳邻居表的更新过程，更新机制与一跳邻居机制相似。

二跳邻居表的信息结构如下：

|  |  |  |
| --- | --- | --- |
| N\_neighbor\_main\_addr | N \_2hop \_addr | N \_time |

2跳邻居集的维护机制也是周期交换hello消息

当结点从一个对称状态的邻居收到hello消息后，结点应该立即更新它的二跳邻居集。

对于hello消息中的每个类型为SYM\_NEIGH和MPR\_NEIGH的地址：

1.如果Hello消息中的地址为收到消息的主地址，那么跳过该地址

2.如果上述条件不成立，那么建立一个二跳邻居元组

N\_neighbor\_main\_addr = Originator Address;

N\_2hop\_addr = main address of the 2-hop neighbor;

N\_time = current time + validity time

3如果hello消息中的结点是非邻居类型，那么在本结点的2跳邻居集里删除下列元组：

N\_neighbor\_main\_addr == Originator Address AND

N\_2hop\_addr == main address of the 2-hop neighbor

## TC控制

结点在网络中传送一条TC消息的木得是为了向外广播一个链路集合，这个集合称作广播链路集合。这些链路必须包含到该结点选择的 MPR集合的所有链路。与MPR集相关的序列号（ANSN）也要随着TC消息一起发送。每当一条链路从MPR集中移除或添加的时候，ANSN数必须要增加。被选择为MPR结点的结点为了建立整个网络的拓扑信息，必须向外广播TC消息。TC消息会在整个网络中泛洪到各个结点。

TC消息机制分为两个部分：生成和处理。

1. **void**
2. generate\_tc(**void** \*p)
3. {
4. **struct** tc\_message tcpacket;
5. **struct** interface \*ifn = (**struct** interface \*)p;
7. olsr\_build\_tc\_packet(&tcpacket);
9. **if** (queue\_tc(&tcpacket, ifn) && TIMED\_OUT(ifn->fwdtimer)) {
10. set\_buffer\_timer(ifn);
11. }
13. olsr\_free\_tc\_packet(&tcpacket);
14. }

Generate\_msg.c

4-13: MID消息通过调用olsr\_build\_tc\_packet()函数生成之后放在MID队列中。当时间戳期满的时候，调用set\_buffer\_timer()设置定时器，最后从给定接口ifn释放消息，同时调用olsr\_free\_tc\_packet()释放内存。

为了构建拓扑信息库，每个被选择为MPR的节点必须广播拓扑控制TC消息，这些通过TC 消息扩散到网络中的信息将有所帮助每个节点计算其路由表。并且TC 消息必须是根据“默认转发算法” 进行转发的。MPR使得拓扑信息的分布具有更好的可扩展性。

当节点的通告链路集变为空时，该节点应当在等于其先前发送的TC消息的

“有效时间” 的持续时间期间仍然发送（空） TC 消息，以便使先前的TC 消息无效， 直到有节点加入到通告链路集。

节点可以传送附加的TC消息以增加其链接故障的反应性。当检测到对 MPR选择器集合发生改变并且这种改变可以归因于链路故障时，TC消息应当在短于TC\_INTERVAL的时间间隔内被发送。

接下来要介绍的是关于TC消息的处理过程：当一个结点收到了一条，首先根据消息首部的Vtime域计算其有效时间。然后按照下列的代码步骤更新

1. **if** (check\_neighbor\_link(from\_addr) != SYM\_LINK) {
2. OLSR\_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr\_ip\_to\_string(&buf, from\_addr));
3. **return** **false**;
4. }

Tc\_set.c

820-824：如果这条消息的发送接口地址不是收到消息的对称一跳邻居，那么丢弃这条消息

1. **if** (tc && 0 != tc->edge\_tree.count) {
2. **if** (olsr\_seq\_inrange\_high((**int**)tc->msg\_seq - TC\_SEQNO\_WINDOW, tc->msg\_seq, msg\_seq)
3. && olsr\_seq\_inrange\_high((**int**)tc->ansn - TC\_ANSN\_WINDOW, tc->ansn, ansn)) {
5. /\*
6. \* Ignore already seen seq/ansn values (small window for mesh memory)
7. \*/
8. **if** ((tc->msg\_seq == msg\_seq) || (tc->ignored++ < 32)) {
9. **return** **false**;
10. }
12. OLSR\_PRINTF(1, "Ignored to much LQTC's for %s, restarting\n", olsr\_ip\_to\_string(&buf, &originator));
14. } **else** **if** (!olsr\_seq\_inrange\_high(tc->msg\_seq, (**int**)tc->msg\_seq + TC\_SEQNO\_WINDOW \* TC\_SEQNO\_WINDOW\_MULT, msg\_seq)
15. || !olsr\_seq\_inrange\_low(tc->ansn, (**int**)tc->ansn + TC\_ANSN\_WINDOW \* TC\_ANSN\_WINDOW\_MULT, ansn)) {
17. /\*
18. \* Only accept future seq/ansn values (large window for node reconnects).
19. \* Restart in all other cases. Ignore a single stray message.
20. \*/
21. **if** (!tc->err\_seq\_valid) {
22. tc->err\_seq = msg\_seq;
23. tc->err\_seq\_valid = **true**;
24. }
25. **if** (tc->err\_seq == msg\_seq) {
26. **return** **false**;
27. }
28. OLSR\_PRINTF(2, "Detected node restart for %s\n", olsr\_ip\_to\_string(&buf, &originator));
29. }
30. }

Tc\_set.c

853-855: 对于在拓扑集中存在满足:

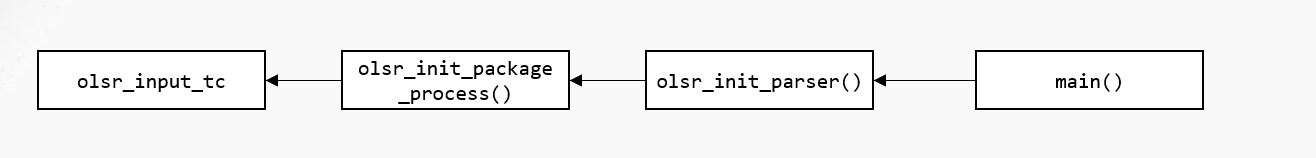
T\_last\_addr == originator address AND

T\_seq > ANSN,

直接丢弃该TC消息，因为消息顺序发生了错误

859-875 只接受那些正确序列号的消息

下面是关于tc消息处理过程的调用



我们可以看到在tc消息的处理过程在包处理函数中被调用。