大连理工大学

OLSR 路由协议代码分析

学号	姓名	班级	负责模块	成绩
201692129	王诗玮	软网 1603	1.OLSR 协议数据结构 2.邻居表操作 3.MPR 算法 4.路由计算 5.文献搜集 6.文档&PPT 制作	
201692040	孙丽爱	软网 1604	1.OLSR 协议简介 2.OLSR 协议框架 3.各种消息的处理 4.文献搜集 5.文档&PPT 制作&排版	

目 录

1 OLSR 协议简介 1
1.1 OLSR 协议特点 1
1.2 OLSR 协议的 MPR 机制 1
1.3 OLSR 协议路由计算 3
2 OLSR 协议框架 4
2. 1 OLSR 协议框架 4
3 OLSR 协议数据结构 5
3.1 OLSR 协议 Packet 格式 5
3.1.1 OLSR 协议分组的基本格式 6
3.1.2 HELLO 消息格式 7
3.1.2 TC 消息格式 8
3.1.3 MID 消息格式 8
3.2 OLSR 协议存储结构 9
3.2.1 多接口相关信息库 9
3.2.2 本地链路信息表 9
3.2.3 邻居信息库 9
3.2.4 拓扑表 10
3.2.5 路由表 11
4 OLSR 中各种消息的处理 12
4.1 HELLO 消息 12
4.1.1 HELLO 消息格式 12
4.1.2 HELLO 消息输入处理 12
4.2 MID 消息 14
4.2.1 MID 消息格式 14
4.2.2 MID 消息接收处理 15
4.3 TC 消息 16
4.3.1 TC 消息结构 16
4.3.2 TC 消息产生 16
4.3.3 TC 消息接收处理 17
5 邻居表操作 19
5.1 一跳邻居表集合的增删改查操作 19
5.1.1 olsr_init_neighbor_table 函数 19
5.1.2 olsr_delete_neighbor_table 函数 19
5.1.3 olsr_insert_neighbor_table 函数 20
5.1.5 olsr_look_up_neighbor_table_alias 函数 20
5.1.5 update_neighbor_status 函数 21
5.2 邻居表的其他操作 21
5.2.1 get_neighbor_status 函数 21
5.2.2 update_link_entry 函数 22
5.2.3 check_link_status 函数 23
6 MPR 算法 24
6.1 MPR 计算过程 25

6.2 add_will_always_nodes 函数.....25
6.3 olsr_find_2_hop_neighbors_with_1_link 函数.....26
6.4 olsr_chosen_mpr 函数.....27
6.5 olsr_find_maximum_covered 函数.....28
6.6 olsr_check_mpr_changes 函数.....28
6.7 olsr_optimize_mpr_set 函数.....29
7 路由计算.....30
7.1 全局变量：.....30
7.2 路由表基本操作.....30
7.2.1 olsr_init_routing_table 函数.....30
7.2.2 olsr_alloc_tc_entry 函数.....31
7.2.3 olsr_alloc_rt_path 函数.....31
7.2.4 olsr_cmp_rtp 函数.....32
7.3 路由表计算.....32
8 总结.....34

1 OLSR 协议简介

OLSR 是 Optimized Link State Routing 的简称，主要用于 MANET 网络 (Mobile Ad hoc network) 的路由协议。

1.1 OLSR 协议特点

OLSR 路由协议是一种先验式的链路状态路由协议，它是为了适应 Ad hoc 网络的需求，对标准链路状态路由协议进行优化而形成的。使用 OLSR 协议的节点在进行数据传输时路由表中就已经存在到达目标节点的路径信息，这使其具有路径选择等待时延小的优点。OLSR 协议对标准链路状态路由协议的优化包括：

(1) 只有被选为多点中继 (MPR：Multipoint Relay) 的节点才产生并周期性洪泛拓扑控制信息，这样可以显著地减少网络中广播的控制分组数量。

(2) 仅选择部分节点作为控制分组的中继节点，以减少路由控制信息的开销。任何节点仅选择部分邻居节点作为它的中继节点，全网范围内都只有选定的中继节点才转发控制分组，其它邻居节点收到该节点发送的控制分组时，只进行处理而不转发。这样就显著地减少了网络中广播的控制分组数量。这类节点被称为多点中继节点。

(3) 缩减了控制分组的大小。节点并不发布与所有邻居节点相连的链路信息，而只发布它与部分邻居的链路子集，但至少发布它与 MPR Selectors 间的链路状态。

1.2 OLSR 协议的 MPR 机制

多点中继的思想是通过减少同一区域内部，相同控制分组的重复转发次数来减少网络中控制分组的数量。

每个节点从其一跳邻居中选择自己的 MPR 集，该节点通过该集合转发的分组能够覆盖该节点所有的二跳邻居节点。节点 A 的多点中继集 MPR(A) 是节点 A 的一跳邻居节点的子集，它满足条件：A 的每个二跳邻居节点都必须有到达 MPR(A) 的双向链路，且节点 A 与 MPR(A) 间的链路也是双向的。而该节点 A 就称为这些 MPR 的多点中继选择者 (MPR Selector)。图 1.1 给出了节点 A 的多点中继集。

被选为 MPR 的节点通过发送控制消息周期性地向全网声明通过自己可以

到达自己的 MPR Selector。在路由计算的过程中，通过 MPR 形成从一个节点到网络中其他节点的路由。每个节点从自己的一跳邻居节点中选择 MPR 时，必须选择和自己之间存在双向对称链路的节点，所以采用这种策略所形成的路由自然能够避开单向链路的问题。

节点 A 的一跳邻居节点被分为两类：MPR 和非 MPR。节点 A 洪泛的拓扑控制分组通过 MPR 转发能到达节点 A 的所有二跳邻居。对于节点 A 的非 MPR，收到来自节点 A 的拓扑控制分组，只进行接收和处理，而不进行转发。OLSR 利用 MPR 节点进行选择性洪泛，有效地减少了控制信息洪泛的规模。

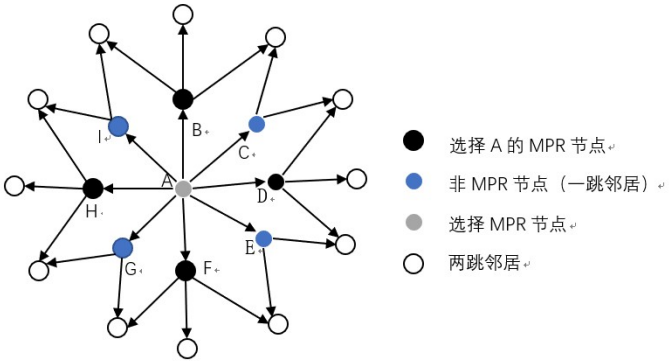

图 1.1 节点 A 的所有两跳邻居、一跳邻居和 MPR 节点

如图 1.1，节点 A 周期性地向网络中其他节点发送 TC 消息，其中至少包含了将其选为 MPR 的邻居节点（即其 MPR selectors）的地址。当节点 B、D、F、H 收到该消息时，判断并发现自己为 A 的 MPR 节点，于是根据消息中的序列号判断其是否最新，若是则转发，若否则丢弃；其他一跳邻居节点通过判断发现自己不属于 A 的 MPR 节点，于是不作转发。OLSR 就是通过这种 MPR 机制来控制 TC 消息在网络中洪泛的规模，减少控制消息给网络带来的负荷。

进一步地，为了压缩 TC 消息的长度，通常在 TC 消息中并不包含源节点(节点 A)所有邻居节点的地址，而仅包含其 MPR selector 的地址，这些消息足以让网络中的各个节点建立整个网络拓扑结构图，进而根据 Dijkstra 最短路径算法独立计算路由表。

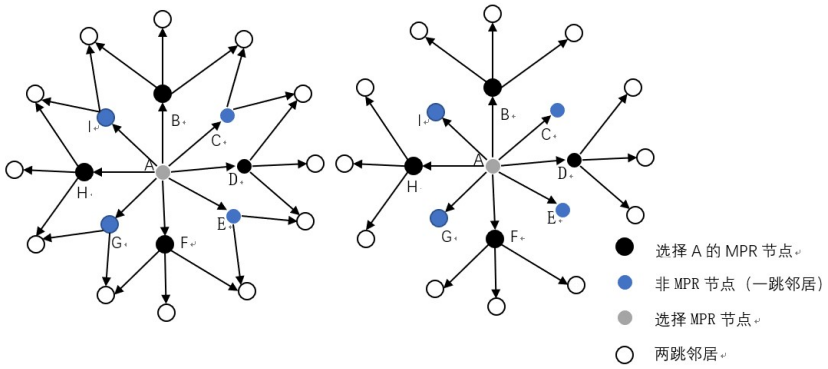

图 1.2 经典的洪泛和 MPR 优化后的洪泛

如图 1.2 所示，OLSR 采用 MPR 机制对路由控制消息进行选择性的洪泛,可以有效地减少整个网络范围内的路由控制消息数量。在节点密度大，数量多的大规模网络中采用 MPR 机制其优势会更加明显。网络中的每个节点都要选择一部分自己的对称邻居节点作为通信的中继节点，即 MPR 节点，而该节点自身则成为 MS 节点。剩下的那些非 MPR 的节点也会接收和处理广播消息，但它们不会转发任何收到的控制消息。

1.3 OLSR 协议路由计算

每个节点都有一张路由表，通过路由表寻找路径信息。一旦网络发生变化，例如节点增减等都会导致路由表的更新变化。计算路径采用 Dijkstra 的最短路径优先算法，跳数，链路带宽，时延，队列长度等都可以作为路径长度的判据。OLSR 路由协议要完成它的路由功能需要这三方面的相互配合：

1. 节点之间要具有完备的控制信息的交互机制，以完成 MPR 节点选择及获取网络拓扑信息
2. 节点内部需要对掌握的邻居信息、链路状态、网络拓扑以及路由信息进行存储
3. 具备可靠的算法来保证节点间控制信息的获取和处理。

OLSR 不依赖于任何中心控制，以一种全分布的方式运转。在无线通信条件下，因为通信冲突或者节点故障等原因，数据包的丢失是在所难免。OLSR 不要求下层提供可靠的数据分组传输，每个节点都周期性的发送控制分组，在一定程度上可以缓解因某一个控制分组丢失所造成的影响。OLSR 也不要求控制消息按序收发。控制消息都带有自己的序列号，每发一个消息，序列号便自增 1；接收到分组的节点通过判断序列号的大小就可以知道该控制消息的新旧程度。

2 OLSR 协议框架

2．1 OLSR 协议框架

OLSR 采用三种控制消息，HELLO 消息、TC(Topology Control) 消息和 MID(Multiple Interface Declaration) 消息，下面分别对这些消息的功能进行说明：

(1) 周期性向邻居节点发送 HELLO 消息，以实现以下功能：

- 邻居感知(Neighbor sensing)
- 确定节点间链路的状态：单向链路、双向链路或者未确定。
- MPR 推选(Multipoint relay selection)
- MPR Selector 计算

(2) 周期性地全网洪泛 TC 消息，节点根据收到的 TC 消息来感知全网的拓扑，根据拓结构图，使用 Dijkstra 算法来计算(更新)路由表。

(3) 周期性地全网洪泛 MID 消息，节点根据收到的 MID 消息来感知其它节点的多接口情况。

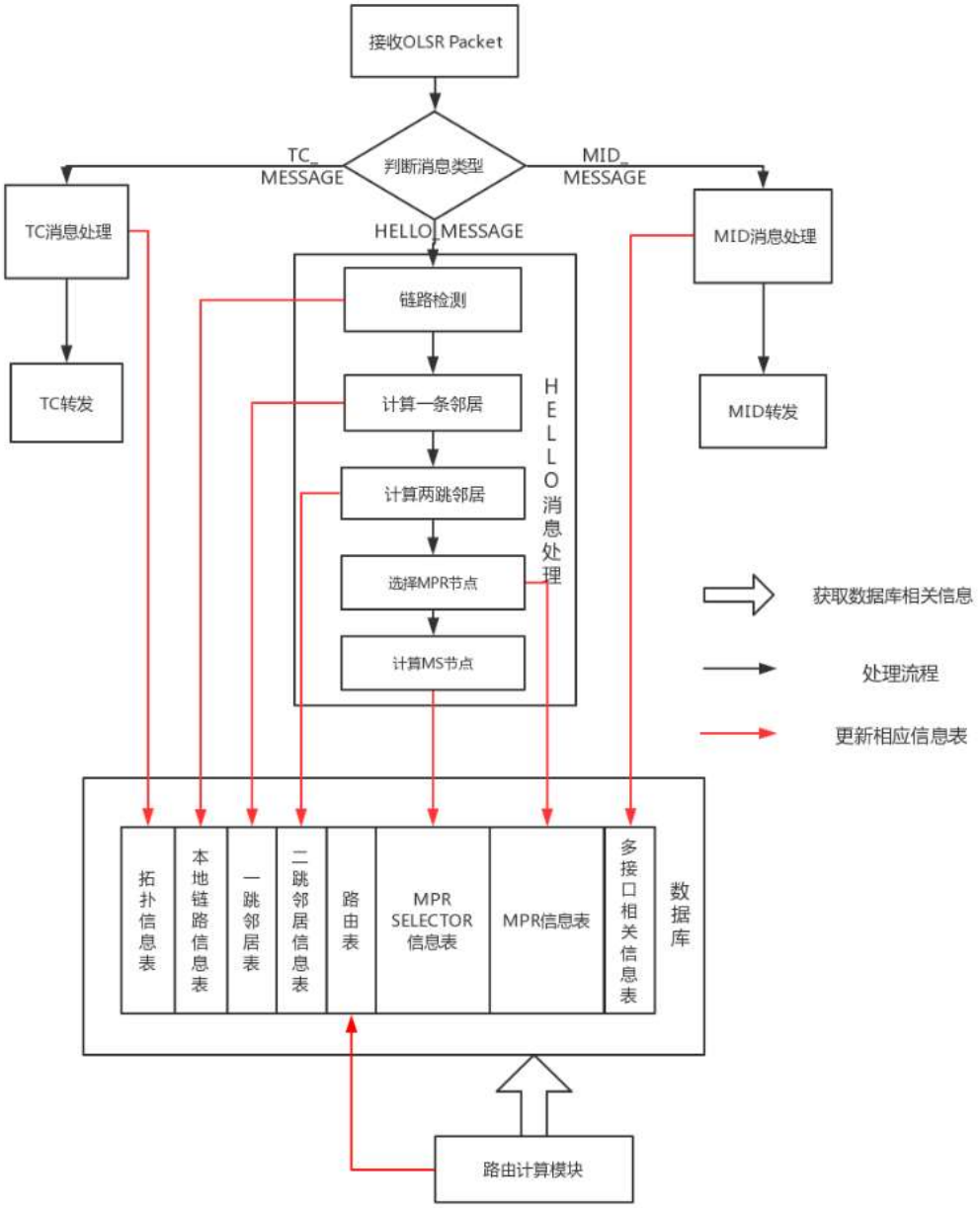

图 2．1　OLSR 框架图

3　OLSR 协议数据结构

3.1　OLSR 协议 Packet 格式

OLSR 路由协议对于所有相关数据都采用统一的 Packet 格式进行通信（如图 3．1），这样做的目的是使协议不会破坏向后的兼容性，便于扩展。这同时也使得属于不同种类的路由信息可以在同一个分组里一起发送。

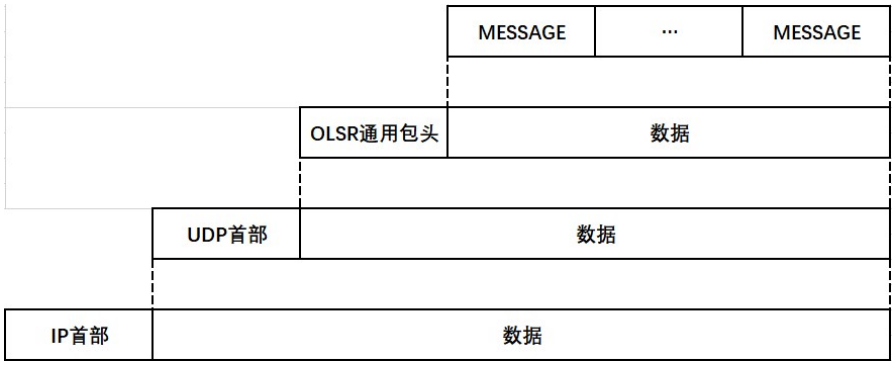

图 3.1 路由协议 Packet 的封装格式

3.1.1 OLSR 协议分组的基本格式

OLSR 协议所使用的相关数据包使用了统一的格式。这样做的目的是促进协议的扩展性且不损害其反向兼容性同时也提供了一种把背负不同种类信息的传输到单一的传输的简便方式。每个分组封装了一个或多个消息这些消息共用同一个头格式，OLSR 协议分组的基本格式如图 3.2 所示：

Packet Length (16bits)		Packet Sequence Number (16bits)
Message Type (8bits)	Vtime (8bits)	Message Size (16bits)
Originator Address (32bits)		
Time To Live (8bits)	Hop Count (8bits)	Message Sequence Number (16bits)
Message		

图 3.2 OLSR 协议分组的基本格式

Packet Length：分组的总长度(以 bytes 为单位)

Packet Sequence Number：分组的序列号，每产生一个新的分组，该序列号加 1

Message Type：该域用于说明分组中 MESSAGE 部分的类型，目前预留 0~127 的消息类型，如 HELLO(HELLO_MESSAGE=1)，TC(TC_MESSAGE=2) 或 MID (MID_MESSAGE=3) 消息类型。

Vtime：用于说明分组中所携带的信息的有效期

Message Size：说明消息的长度，以 bytes 为单位，从 Message Type 域开始到下一个域 Message Type 的开始处，如果没有下一个 MESSAGE，则到分组结束处。

Originator Address：发送该消息源节点的主地址（非中继节点的接口地址），在消息的传输过程中这个域必须保持不变

Time To Live：表示该消息能被传送的最大跳数，每转发一次，该值减小 1，当节点收到一个 TTL 值为 0 或 1 的 MESSAGE 后，不再对其进行转发，通过设置 TTL 的值，节点可以控制分组洪泛的范围。

而具体的 HELLO 消息，TC 消息都作为基本消息格式中的 MESSAGE 部分，通过 Message Type 域来区分，下面将具体地介绍这几种消息格式。

3.1.2 HELLO 消息格式

HELLO 消息格式是在通用消息格式基础上将 Message Type 设置为 HELLO_MESSAGE；Time To Live 设置 1；Vtime 设置为 NEIGHBOR_HOLD_TIME。其结构如图 3.3 所示：

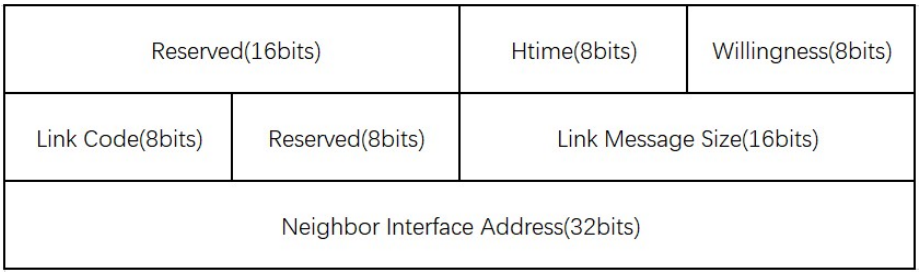

图 3.3 HELLO 消息格式

Reserved：预留域必须设置为全 0。

Htime：描述此接口的 HELLO 消息发送时间间隔。

Willingness：表示节点为其他节点转发消息的愿意程度。

一个具有 WILL_NEVER 的节点永远不会被任何节点选为 MPR。

一个具有 WILL_ALWAYS 的节点永远会被选为 MPR。

缺省情况下，一个节点的 willingness 应设为 WILL_DEFAULT。

Link Code：说明了节点与其邻居节点之间的链路状态信息，其比特位如图 3.4 所示：

7	6	5	4	3	2	1	0
0	0	0	0	Neighbor Type		Link Type	

图 3.4 Link Code 域

Link Type：该域指出链路的类型，总共有 4 种类型：

(1) ASYM-LINK：表示链路是非对称的（即单向的）

(2) SYM_LINK：表示该链路是对称的（即双向的）

(3) UNSPEC_LINK：表示没有指定链路的信息。

(4) LOST_LINK：表示链路断开。

Neighbor Type:该域指出邻居的类型，总共有 3 种类型：

(1) SYM_NEIGH：表示该节点和邻居节点之间至少有一个对称（即双向）链路。

(2) MPR_NEIGH：表示该节点和邻居节点至少有一个对称（即双向）链路并且该邻居节点被本节点选择为 MPR。

(3) NOT_NEIGH：表示该邻居不是对称邻居

Link Message Size：本链路消息的大小。从“链路类型”字段开始直到下一个“链路类型”字段之前（若无下一个“链路类型”字段，则到分组结尾）。

Neighbor Interface Address：邻居地址列表，每一种链路类型之后都紧随一个邻居列表地址，表示发送该 HELLO 分组的节点到这个邻居列表中的所有节点的链路类型是相同的，都为前面给出的链路类型。

3.1.2 TC 消息格式

TC 消息格式是在通用消息格式基础上将 Message Type 设置为 TC_MESSAGE；Vtime 设置为 TOP_HOLD_TIME。其结构如图 3.5 所示：

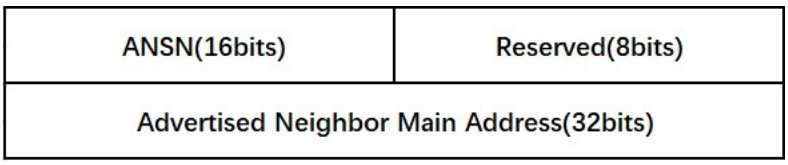

图 3.5 TC 消息格式

ANSN(Advertised Neighbor Sequence Number)：当节点发现其邻居节点集发生变化时，将 TC 消息中的 ANSN 加 1，其他节点收到一个 TC 消息时，可以通过对 ANSN 的比较来确定该消息是否是较新的消息。

Reserved：这个域作为保留域，全填充为 0。

Advertised Neighbor Main Address：MPR selector 节点的主地址，发送 TC 消息的节点至少将所有的 MPR selector 主地址封装在 TC 消息中。如果需要提供一定的冗余度（以提高健壮性），可以把其他非 MPR Selector 邻居的主地址包括进去。

3.1.3 MID 消息格式

MID 消息格式是在通用消息格式基础上将 Message Type 设置为 MID_MESSAGE；Time To Live 设置为最大值 255，以洪泛至整个网络；Vtime 设置为 MID_HOLD_TIME。其结构如图 3.5 所示：

图 3.6 MID 消息格式

OLSR Interface Address 为节点除了主地址之外的所有接口地址，节点的主地址

设置于 Originator address 中。

3.2 OLSR 协议存储结构

通过交换 OLSR 的控制信息，每个节点都积累了网络的信息，这些信息是以表的形式存储中数据库中，用于计算或更新路由，

3.2.1 多接口相关信息库

多接口信息库中存在多接口信息表，该表存储了节点接口的相关信息，并以如下方式存储：

(I_iface_addr, I_main_addr, I_time)

- I_iface_addr 是节点的接口地址
- I_main_addr 节点的主地址
- I_time 指示这个记录的有效时间，超时时删除该记录

3.2.2 本地链路信息表

本地链路信息库存在本地链路信息表，该表存储了该节点和邻居节点的链路信息，以如下的方式存储

(L_local_iface_addr, L_neighbor_iface_addr, L_SYM_time, L_ASYM_time, L_time)

- L_local_iface_addr：本节点的接口地址
- L_neighbor_iface_addr：邻居节点的接口地址
- L_SYM_time：链路对称时刻，在该时刻之前，链路是对称的双向链路
- L_ASYM_time：链路非对称时刻，在 L_SYM_time 之后而 L_ASYM_time 之前，链路是非对称的单向链路。
- L_time：链路维持时刻，链路在该时刻失效，必须被删除，当链路对称时刻和链路非对称时刻都到达时，链路丢失。

3.2.3 邻居信息库

邻居信息库存储了邻居节点信息，二跳邻居节点信息，MPR 节点信息，MPR Selectors 节点信息。

(1)一跳邻居信息表格式：

(N_neighbor_main_addr, N_status, N_willingness)

- N_neighbor_main_addr：邻居节点的主地址（节点在所有的接口地址中选择一个作为其主地址，用于在网络中表示该节点）。
- N_status:标识与邻居节点的状态关系，如果本节点与邻居节点之间存在对称链路，N_status 为 SYM，否则为 NOT_SYM。
- N_willingness：邻居节点为其他节点转发分组的愿意度，可以设置 0~7 中的任意一个整数。WILL_NEVER（0）表示不愿意为其他节点转发数据，而 WILL_ALWAYS（7）表示必须被选择为其他节点转发数据。其愿意度从 7 到 0 逐渐递减。

⑵两跳邻居信息表格式：

(N_neighbor_main_addr, N_2hop_addr, N_time)

- N_neighbor_main_addr：对称邻居节点的主地址。
- N_2hop_addr：二跳邻居节点主地址，二跳邻居节点与邻居节点之间以对称链路连接。
- N_time：指示了这个二跳邻居节点信息的有效时间，超时时删除该记录。

⑶MPR 节点信息表

每个节点都记录邻居节点中作为本节点 MPR 的节点的主地址。

⑷MS 节点信息表格式：

(MS_main_addr, MS_time)

- MS_main_addr 表示 MPR selectors 的主地址。
- MS_time 表示了这条记录的有效时间，超时时删除该记录。

3.2.4 拓扑表

OLSR 路由协议中每个节点维持一张描述网络拓扑信息的网络拓扑信息表，该网络拓扑信息表是从各节点的 TC 消息获取，用于计算路由。其包含四个部分：目的地址，到达目的地址的最后一跳地址，表项序列号，表项有效时间，其结构如下：

(T_dest_addr, T_last_addr, T_seq, T_time)

- T_dest_addr 是一个节点的主地址，该节点可以通过拥有主地址为 T_last_addr 节点一跳到达。

- T_last_addr 通常为 T_dest_addr 的 MPR。
- T_seq 是一个表项序列号，该表项序列号用于记录本节点收到的最后一个 TC 消息的序号，当收到一个新的 TC 消息时，将新的 TC 消息的序列号与表项序列号相比较来决定接收还是丢弃该消息。
- T_time 表项有效时间用于表示该表项生存时间，超过生存时间的表项不能用于路由计算，必须删除。

3.2.5 路由表

在路由表结构上，OLSR 路由协议同大多数基于表驱动工作方式的路由协议一样，包含四个部分：目的地址，下一跳地址，到目的地的距离，可以到达 R_next_addr 邻居的本地接口地址。格式如下：

(R_dest_addr ，R_next_addr，R_dist，R_iface_addr)

- R_dest_addr：路由表中目的节点 IP 地址域，保存所要到达的目的节点的 IP 地址，每个表项都具有不同的目的地址，以此作为区别和查找路由的关键字。
- R_next_addr：下一跳节点 IP 地址域，保存到达目的节点的路径上的下一跳节点的地址。
- R_dist：跳数值域保存了从本节点到达目的路由器所需要的跳数。

4 OLSR 中各种消息的处理

4.1 HELLO 消息

4.1.1 HELLO 消息格式

在 OLSR 中传播的 HELLO 消息用来进行邻居感知，确定链路状态等操作。在 OLSR 中定义的 HELLO 消息声明如下。

packter.h

```
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;
66     struct hello_neighbor *neighbors;
67 };
```

packter.h

60-67：该 HELLO 消息的有效时间 *vtime*，为消息发送间隔 *htime*，该节点为其他节点转播数据包的意愿 *willingness*，和该节点的邻居节点描述信息，描述信息包括该邻居节点的地址和到该邻居节点的联络状态、花费等。

4.1.2 HELLO 消息输入处理

olsr_input_hello 函数

OLSR 使用 olsr_input_hello 函数对消息进行处理，该函数返回值永远是 false 表示 HELLO 消息永远不会被转发。

process_package.h

```
370 bool
371 olsr_input_hello(union olsr_message * ser, struct interface * inif, union olsr_ip_addr * from)
372 {
373     struct hello_message hello;

374     if (ser == NULL) {
375         return false;
376     }
377     if (deserialize_hello(&hello, ser) != 0) {
378         return false;
379     }
380     olsr_hello_tap(&hello, inif, from);
381
382     return false;
383 }
```

process_package.h

370-383：在 OLSR 中 HELLO 消息的作用是向邻居节点确认链路状态，从而进行 MPR 的选择，因此该函数的返回值总是 false，即该消息永远不会被转发。节点首先调用 deserialize_hello 函数从收到的通用消息 *ser* 中反向生成一个 HELLO 消息。若成功提取 HELLO 消息，则调用 olsr_hello_tap 函数对该消息进行处理。

deserialize_hello 函数

OLSR 通过调用 deserialize_hello 函数从接收到的消息中重构 HELLO 消息，如果构造成功函数返回 0，否则返回一个非 0 值。参数 *hello* 为一个传出参数，其内容为从消息中重构出的 HELLO 消息，参数 *ser* 为一个传入参数，指出重构 *hello* 消息的来源。

(1)边界检查

在重构 HELLO 消息时，首先对传入的消息进行类型、ttl 等合法性检查。如:

process_package.h

```
323    curr = ser;
324    pkt_get_u8(&curr, &type);
325    if (type != HELLO_MESSAGE && type != LQ_HELLO_MESSAGE) {
326      return 1;
327    }
```

process_package.h

323-327：根据消息基础格式从中获取类型，若消息的类型不是 HELLO 消息 或者封装后的 HELLO 消息，则中止函数的执行，返回重构失败。

(2)对 HELLO 消息传递的有效信息进行重构

process_package.h

```
343    limit = ((const unsigned char *)ser) + size;
344    while (curr < limit) {
345       const unsigned char *limit2 = curr;
346       uint8_t link_code;
347       uint16_t size2;

348       pkt_get_u8(&curr, &link_code);
349       pkt_ignore_u8(&curr);
350       pkt_get_u16(&curr, &size2);

351       limit2 += size2;
352       while (curr < limit2) {
353          struct hello_neighbor *neigh = olsr_malloc_hello_neighbor("HELLO deserialization");
354          pkt_get_ipaddress(&curr, &neigh->address);
355          if (type == LQ_HELLO_MESSAGE) {
356             olsr_deserialize_hello_lq_pair(&curr, neigh);
357          }
358          neigh->link = EXTRACT_LINK(link_code);
359          neigh->status = EXTRACT_STATUS(link_code);

360          neigh->next = hello->neighbors;
361          hello->neighbors = neigh;
362    }
```

process_package.h

348-350：确定该消息为 HELLO 消息后从该消息的邻居列表中逐一取出到该节点的链路状态 *link_code* 并移动 *curr* 指针到描述节点信息的地址。

353-355：对消息携带的邻居信息逐步重构出 HELLO 消息。首先调用 olsr_malloc_hello_neighbor 函数为该邻居条目分配一块内存空间。随后从消息中取出该邻居节点的地址。最后从 *link_code* 的解析出该节点的链路类型和邻居类型(*link_code* 比特位 0，1 表示链路类型，2，3 表示邻居类型）赋给该邻居条目并加入到 HELLO 消息的邻居链表中。

olsr_hello_tap 函数

在重建 HELLO 消息后对 HELLO 消息承载的内容进行处理。

process_package.h

```
412 struct link_entry *lnk = update_link_entry(&in_if->ip_addr, from_addr, message, in_if);
```

```
413 if (olsr_validate_address(from_addr)) {
414     union olsr_ip_addr * main_addr = mid_lookup_main_addr(from_addr);
415     if ((main_addr==NULL)||(ipequal(&message->source_addr, main_addr))){
416         insert_mid_alias(&message->source_addr, from_addr, message->vtime);
417       }
418           else {
      /* log code*/
423           }
424       }
```

process_package.h

412：函数首先根据传来的 HELLO 消息更新从接受接口到该包发送接口的链路状态

413-424：函数检查该 HELLO 消息的源 IP 地址是否通过验证，若合法则更新多路接口信息。否则在日志中记录。

414-417：函数首先在 MID 集合中查找有无该接口的记录。若找不到对应记录则插入一条记录来记录该多接口信息。

process_package.h

```
465 neighbor = lnk->neighbor;
466   if (lookup_mpr_status(message, in_if))
467       olsr_update_mprs_set(&message->source_addr, message->vtime);
468   if (neighbor->willingness != message->willingness) {
469       struct ipaddr_str buf;
470       OLSR_PRINTF(1, "Willingness for %s changed from %d to %d - UPDATING\n", olsr_ip_to_string(&buf,
&neighbor->neighbor_main_addr),
471                               neighbor->willingness, message->willingness);
472       neighbor->willingness = message->willingness;
473       changes_neighborhood = true;
474       changes_topology = true;
475   }
476   if (neighbor->willingness != WILL_NEVER)
477       process_message_neighbors(neighbor, message);
478   olsr_process_changes();
479   olsr_free_hello_packet(message);
480   return;
```

process_package.h

466-467：函数检查节点是否被选为 MPR，如果是，更新 MPR 集合

468-474：对比链路集合中存储的对侧节点转发意愿，如果不一致，则说明对侧节点发生了改变，更新转发意愿值。

476-477：如果对侧节点有意愿转发数据，则将该 HELLO 消息中的 neighbors 字段所携带的信息记录下来。

478-479：更新该 HELLO 消息所带来的改变最终释放该 HELLO 消息。

4.2 MID 消息

4.2.1 MID 消息格式

在 OLSR 中，节点通过洪泛 MID 消息通知本节点的多接口情况。MID 消息声明如下：

packet.h

```
97 struct mid_alias {
98     union olsr_ip_addr alias_addr;
99     struct mid_alias *next;
100 };

101 truct mid_message {
102     olsr_reltime vtime;
```

```
103     union olsr_ip_addr mid_origaddr;        /* originator's address */
104     uint8_t mid_hopcnt;                       /* number of hops to destination */
105     uint8_t mid_ttl;                         /* ttl */
106     uint16_t mid_seqno;                        /* sequence number */
107     union olsr_ip_addr addr;                 /* main address */
108     struct mid_alias *mid_addr;              /* variable length */
109 };
```
packet.h

107-108：在 MID 消息中除了 OLSR 基本包的通用字段外，声明了本节点的主 IP 地址和持有的其他 IP 地址。这些地址以链表的形式存储在 mid_alias* 类型的变量 *mid_addr* 中。

4.2.2 MID 消息接收处理

OLSR 调用 olsr_input_mid 函数处理 MID 消息，若函数返回 true 则转发该数据包，否则不进行转发。

```
562   bool
563   olsr_input_mid(union olsr_message *m, struct interface *in_if __attribute__ ((unused)),
                          union olsr_ip_addr *from_addr)
564   {
565   struct ipaddr_str buf;
567   struct mid_alias *tmp_adr;
568   struct mid_message message;
569   mid_chgestruct(&message, m);
570   if (!olsr_validate_address(&message.mid_origaddr)) {
571                   olsr_free_mid_packet(&message);
572       return false;
573    }

574   tmp_adr = message.mid_addr;
575   if (check_neighbor_link(from_addr) != SYM_LINK) {
576   OLSR_PRINTF(2, "Received MID from NON SYM neighbor %s\n", olsr_ip_to_string(&buf, from_addr));
577      olsr_free_mid_packet(&message);
578      return false;
579   }
580   olsr_update_mid_table(&message.mid_origaddr, message.vtime);
581   for (;tmp_adr; tmp_adr = tmp_adr->next) {
      /*Robust code*/
595   if (!mid_lookup_main_addr(&tmp_adr->alias_addr)) {
      /*Log code*/
598           insert_mid_alias(&message.mid_origaddr, &tmp_adr->alias_addr, message.vtime);
599   } else {
600          olsr_insert_routing_table(&tmp_adr->alias_addr, olsr_cnf->maxplen, &message.mid_origaddr,
601     OLSR_RT_ORIGIN_MID);
602   }
603 }
604 olsr_prune_aliases(&message);
605 olsr_free_mid_packet(&message);
606 return true;
607 }
```
mid_set.c

569-570：将接收到的 OLSR 通用消息转变为 MID 消息，并检查该消息中的生成地址是否有效，若为无效地址，则释放该消息不再转发。

575-579：判断该消息是否是从对称邻居发送的，如果不是则丢弃该数据包，不再转发。

580：更新该节点主地址的 MID 记录有效期。

595-603：在 MID 集合中寻找该 MID 消息相关 IP 地址的记录并插入地址别名，如果找不到记录，则在路由表中插入相应记录。

604：移除 MID 消息中没有记录的别名

605-606：处理结束，释放该内存空间并转发该消息。

4.3 TC 消息

4.3.1 TC 消息结构

在 OLSR 中声明的 TC 消息结构声明如下：

packet.h
```
71 struct tc_mpr_addr {
72    union olsr_ip_addr address;
73    struct tc_mpr_addr *next;
74    uint32_t linkquality[0];
75};
76 struct tc_message {
77     olsr_reltime vtime;
78     union olsr_ip_addr source_addr;
79     union olsr_ip_addr originator;
80     uint16_t packet_seq_number;
81     uint8_t hop_count;
82     uint8_t ttl;
83     uint16_t ansn;
84     struct tc_mpr_addr *multipoint_relay_selector_address;
85};
```
packet.h

71-84：在 OLSR 中传输的 TC 消息的作用是帮助节点感知全网拓扑信息。因此一个 TC 分组的内容包括 TC 消息的产生节点 *originator*，TC 消息序列号 *ansn*，和所有与该节点有关的 MPR 信息。在 TC 消息数据包中，MPR 信息通过一个单项链表进行存储在 *multipoint_relay_selector_address* 中，其每一个节点均为 tc_mpr_addr 类型变量。每个节点均包含将其中一个 MPR Selector 的地址 *address*，和到该节点的链路状态质量 *linkquality*。

4.3.2 TC 消息产生

OLSR 通过 generate_tc 函数和 olsr_build_tc_packet 函数来产生并填充 TC 数据包。

generate_tc 函数为生成 TC 消息的一级调用，该函数负责构建 TC 消息并将其加入到 TC 消息队列。

olsr_build_tc_packet 函数负责根据配置文件等填充数据包中的内容。

generate_tc 函数

generate_msg.c
```
82 void
83 generate_tc(void *p)
84 {
85     struct tc_message tcpacket;
86     struct interface *ifn = (struct interface *)p;
87     olsr_build_tc_packet(&tcpacket);
88     if (queue_tc(&tcpacket, ifn) && TIMED_OUT(ifn->fwdtimer)) {
89         set_buffer_timer(ifn);
90     }
91     olsr_free_tc_packet(&tcpacket);
92 }
```
generate_msg.c

85-91：首先声明一个变量并调用 olsr_build_tc_packet 函数填充该数据包的内容并将该数据包放入消息队列中，同时如果设置的消息发送时间到时，即通过该接口发送消息并通过 set_buffer_time 函数重新设置定时器，最后调用

olsr_free_tc_packet 释放该消息。

olsr_build_tc_packet 函数

packet.c
```
316 int
317 olsr_build_tc_packet(struct tc_message *message)
318 {
...
323     message->multipoint_relay_selector_address = NULL;
324     message->packet_seq_number = 0;
325     message->hop_count = 0;
326     message->ttl = MAX_TTL;
327     message->ansn = get_local_ansn();
328     message->originator = olsr_cnf->main_addr;
329     message->source_addr = olsr_cnf->main_addr;

330     OLSR_FOR_ALL_NBR_ENTRIES(entry) {
331       if (entry->status != SYM) {
332         continue;
333       }

334       switch (olsr_cnf->tc_redundancy) {
335       case (2):
336         {
337           message_mpr = olsr_malloc_tc_mpr_addr("Build TC");
338           message_mpr->address = entry->neighbor_main_addr;
339           message_mpr->next = message->multipoint_relay_selector_address;
340           message->multipoint_relay_selector_address = message_mpr;
341           entry_added = true;
342           break;
343         }
...
375     } OLSR_FOR_ALL_NBR_ENTRIES_END(entry);
}
```
packet.c

323-329：节点将自身的信息填入 TC 消息的基本信息字段。其中，将无法立即指定的字段如数据包的序列号 *packet_seq_number* 置零。*Ansn* 号由 get_local_ansn 函数公共管理。消息的生成者 *originator* 和消息的源地址 *source_addr* 置为本节点的主地址。

334-383：对所有邻居节点进行遍历，对于所有正常工作的节点，根据公共配置中 TC 消息冗余性的不同要求将不同范围的邻居节点主地址包含进来（默认配置为只将 MPR selector 的主地址包含进来，可通过修改配置将范围改为所有邻居节点（配置值为 2），所有 MRP selector&选择的 MPR 节点（配置值为 1））。对于不同的范围执行的操作相同，故略去了重复代码。一旦发现满条件的邻居节点，则通过 olsr_malloc_tc_mpr_addr 函数申请一块内存空间，将该邻居节点的主地址填入，并以头插入的方式加入到 TC 消息的链表当中。

4.3.3 TC 消息接收处理

TC 消息在 OLSR 网络中以洪泛的方式进行传播，因此节点处理 TC 消息的任务之一就是判断该消息是否应该继续转发。OLSR 调用 olsr_input_tc 函数处理 TC 消息，若函数返回 true 则继续转发该消息，函数返回值为 false 则该消息被丢弃不再转发。

tc_set.c
```
810   pkt_get_u8(&curr, &type);
811   if ((type != LQ_TC_MESSAGE) && (type != TC_MESSAGE)) {
812     return false;
813   }
```

```
814   /*
815    * If the sender interface (NB: not originator) of this message
816    * is not in the symmetric 1-hop neighborhood of this node, the
817    * message MUST be discarded.
818    */
819   if (check_neighbor_link(from_addr) != SYM_LINK) {
820       OLSR_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr_ip_to_string(&buf, from_addr));
821       return false;
822   }
```
tc_set.c

当节点收到一个 TC 消息时，会首先对该数据包进行检查。

811-813：节点判断该数据包是不是 TC 数据包，如果不是则不进行处理。

819-820：判断该数据包的发送者是不是该节点的对称邻居，如果是则不进行处理。

tc_set.c
```
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)) {
...
852           if ((tc->msg_seq == msg_seq) || (tc->ignored++ < 32)) {
853               return false;
854           }
```
tc_set.c

847-854：节点随后处理数据包的 seq 和 ANSN 序列号，首先查询存储集合中所储存的序列号，随后查询 *ignored* 变量，如果序列号已经存在或 *ignored* 小于 32，则该数据包为冗余数据包，不对该包进行处理。

tc_set.c
```
880       tc = olsr_add_tc_entry(&originator);
...
898     limit = (unsigned char *)msg + size;
899     borderSet = 0;
900     emptyTC = curr >= limit;
901     while (curr < limit) {
902         if (olsr_tc_update_edge(tc, ansn, &curr, &upper_border_ip)) {
903             changes_topology = true;
904     }
```
tc_set.c

880：节点根据 tc_message 中的 originator 字段在其所维护的 TC 集合中开辟一个新的条目。

898-900：节点通过比较数据包大小与 tc_message 头部大小判断该消息是否携带信息。

901-904：遍历节点所携带的信息，更新拓扑集合。

tc_set.c
```
927   if (emptyTC && lower_border == 0xff && upper_border == 0xff) {
929       memset(&lower_border_ip, 0x00, sizeof(lower_border_ip));
930       memset(&upper_border_ip, 0xff, sizeof(upper_border_ip));
931       borderSet = 1;
932   }

948   if (emptyTC && borderSet) {
      ◦◦◦
954   olsr_cleanup_mid(&originator);
955   olsr_cleanup_hna(&originator);
956           olsr_cleanup_duplicates(&originator);

957   olsr_delete_tc_entry(tc);
958   }
```
tc_set.c

最终对不传递信息的 TC 消息进行处理。

927-931：如果该包被判定为空，则将该边信息赋值为特殊标志值，并置边界集变量为 1

954-957：删除所有被影响的 MID，HNA 集合并释放掉该 TC 数据包。

5 邻居表操作

在 OLSR 中每一个节点维护一个邻居表集合来储存与其相连的节点信息。这些集合主要根据 HELLO 消息进行修改。由于一跳邻居表和二跳邻居表的相关操作比较类似，因此这里只介绍一跳邻居表的相关操作

5.1 一跳邻居表集合的增删改查操作

5.1.1 olsr_init_neighbor_table 函数

函数功能：对节点存储的一跳邻居集合进行初始化。

neighbor_table.c

```
56 void olsr_init_neighbor_table(void){
58    int i;
59    for(i=0; i < HASHSIZE; i++){
60          neighbortable[i].next=&neighbortable[i];
61          neighbortable[i].prev=&neighbortalbe[i];
62    }
63 }
```

neighbor_table.c

60-62：在 OLSR 中，邻居集合以双向循环列表进行存储，在节点的初始化阶段，将链表的头节点的前驱节点和后继节点均指向自身，以防止内存非法访问的操作。

5.1.2 olsr_delete_neighbor_table 函数

函数功能：根据参数的地址从邻居集合中删除某节点及其相连的二跳节点，若成功删除返回 1，否则返回 0

neighbor_table.c

```
160 int olsr_delete_neighbor_table(const union olsr_ip_addr *neighbor_addr){
161     struct neighbor_2_list_entry *two_hop_list, *two_hop_to_delete;
162     uint32_t hash;
163     struct neighbor_entry *entry;
164     hash = olsr_ip_hashing(neighbor_addr);
165     entry = neighbortable[hash].next;
166     while (entry != &neighbortable[hash]) {
167        if (ipequal(&entry->neighbor_main_addr, neighbor_addr))
168           break;
169        entry = entry->next;
170     }
171     if (entry == &neighbortable[hash])
172        return 0;
173     two_hop_list = entry->neighbor_2_list.next;
174     while (two_hop_list != &entry->neighbor_2_list) {
175       two_hop_to_delete = two_hop_list;
176       two_hop_list = two_hop_list->next;
177       two_hop_to_delete->neighbor_2->neighbor_2_pointer--;
178       olsr_delete_neighbor_pointer(two_hop_to_delete->neighbor_2, entry);
179        olsr_del_nbr2_list(two_hop_to_delete);
180 }
181     DEQUEUE_ELEM(entry);
182     free(entry);
183     changes_neighborhood = true;
184     return 1;
185 }
```

neighbor_table.c

164：OLSR 在存储时对 IP 地址进行了哈希散列以减少占用的连续内容大小。

因此函数处理中首先要对传入的 IP 地址进行哈希散列。
166-172：在邻居双向循环链表组中遍历查找该节点。如果没有找到与该 IP 地址对应的几点，则结束处理，返回删除失败，如果找到该节点则进行下一步处理。
173-179：由于删除了一跳邻居，故也要对通过该节点可达的二跳邻居进行处理。遍历其所覆盖的二跳邻居节点，减少指向计数器并删除该节点。
181-182：最后从双向链表中删除该节点，并释放掉该节点所占用的内存空间。

5.1.3 olsr_insert_neighbor_table 函数

函数功能：在一跳邻居集合中新增一个条目，如果成功，返回增加的内容。

neighbor_table.c
```
215 struct neighbor_entry *
216 olsr_insert_neighbor_table(const union olsr_ip_addr *main_addr){
217     uint32_t hash;
218     struct neighbor_entry *new_neigh;
219     hash = olsr_ip_hashing(main_addr);
220     for (new_neigh = neighbortable[hash].next; new_neigh != &neighbortable[hash]; new_neigh = new_neigh->next) {
221         if (ipequal(&new_neigh->neighbor_main_addr, main_addr))
222             return new_neigh;
223     }
224     new_neigh = olsr_malloc(sizeof(struct neighbor_entry), "New neighbor entry");
225     new_neigh->neighbor_main_addr = *main_addr;
226     new_neigh->willingness = WILL_NEVER;
227     new_neigh->status = NOT_SYM;
228     new_neigh->neighbor_2_list.next = &new_neigh->neighbor_2_list;
229     new_neigh->neighbor_2_list.prev = &new_neigh->neighbor_2_list;
230     new_neigh->linkcount = 0;
231     new_neigh->is_mpr = false;
232     new_neigh->was_mpr = false;
233     QUEUE_ELEM(neighbortable[hash], new_neigh);
234     return new_neigh;
235 }
```
neighbor_table.c

219：计算该地址的哈希值，以正确的插入到邻居表中。
220-222：函数首先在已经存在的邻居集合中查找该地址，如果找到有地址匹配的节点，则说明该节点已存在，无需进行插入操作。
224-233：为该邻居节点申请一块内存空间，填充缺省内容并合并到邻居链表中。

5.1.5 olsr_look_up_neighbor_table_alias 函数

函数功能：根据 IP 地址在邻居表中寻找节点，如果成功则返回指向该节点条目的指针。

neighbor_table.c
```
280 struct neighbor_entry * olsr_lookup_neighbor_table_alias(const union olsr_ip_addr *dst) {
281     struct neighbor_entry *entry;
282     uint32_t hash = olsr_ip_hashing(dst);
283     for (entry = neighbortable[hash].next; entry != &neighbortable[hash]; entry = entry->next) {
284         if (ipequal(&entry->neighbor_main_addr, dst))
285           return entry;
286     }
287     return NULL;
288 }
```
neighbor_table.c

282-287：函数首先计算该 IP 的哈希值，并在对应的邻居链表中查询是否有节

点的 IP 与参数相同，若有则将该节点返回。若遍历完整个链表仍然没有找到满足条件的节点，则返回空值。

5.1.5 update_neighbor_status 函数

函数功能：修改指定邻居节点的状态并返回修改后的状态

neighbor_table.c

```
299 int update_neighbor_status(struct neighbor_entry *entry, int lnk){
231         if (lnk == SYM_LINK) {
232                 if (entry->status == NOT_SYM) {
233                         struct neighbor_2_entry *two_hop_neighbor;
234                         if ((two_hop_neighbor =
235                         olsr_lookup_two_hop_neighbor_table(&entry->neighbor_main_addr)) != NULL) {
236                                 olsr_delete_two_hop_neighbor_table(two_hop_neighbor);
237                         }
238                         changes_neighborhood = true;
239                         changes_topology = true;
240                         if (olsr_cnf->tc_redundancy > 1)
241                                 signal_link_changes(true);
242                 }
243                 entry->status = SYM;
244         } else {
245                 if (entry->status == SYM) {
246                         changes_neighborhood = true;
247                         changes_topology = true;
248                         if (olsr_cnf->tc_redundancy > 1)
249                                 signal_link_changes(true);
250                 }
251           entry->status = NOT_SYM;
252         }
253         return entry->status;
254  }
```

neighbor_table.c

231-243：如果欲将邻居节点的状态由非对称节点变更为对称节点，则删除它所连接到的二跳邻居，如果全局配置中 TC 包存在冗余度且大于 1，则通过通知 OLSR 当前节点状态发生改变，重新进行 MPR 选举等操作。

233-250：如果欲将邻居节点的状态由对称节点变更为非对称节点，则根据上述条件进行通知 OLSR 进行路由更新。

5.2 邻居表的其他操作

5.2.1 get_neighbor_status 函数

函数功能：查找到该邻居节点的最佳链路状态。若不存在链路返回 0。

link_set.c

```
212 static int get_neighbor_status(const union olsr_ip_addr *address){
213     const union olsr_ip_addr *main_addr;
214     struct interface *ifs;
215     if (!(main_addr = mid_lookup_main_addr(address)))
216         main_addr = address;
217     for (ifs = ifnet; ifs != NULL; ifs = ifs->int_next) {
218         struct mid_address *aliases;
219         struct link_entry *lnk = lookup_link_entry(main_addr, NULL, ifs);
220         if (lnk != NULL) {
221             if (lookup_link_status(lnk) == SYM_LINK)
222                 return SYM_LINK;
223         }
224         for (aliases = mid_lookup_aliases(main_addr); aliases != NULL; aliases = aliases->next_alias) {
225             lnk = lookup_link_entry(&aliases->alias, NULL, ifs);
226             if (lnk && (lookup_link_status(lnk) == SYM_LINK)) {
227                 return SYM_LINK;
228             }
```

```
229         }
230     }
231     return 0;
232 }
```
link_set.c

215-216：函数首先在 MID 集合中找到持有目标 IP 的节点的主地址。
217-231：函数对节点的所有接口和目标节点的所有接口之间的链路进行遍历，查找是否存在对称链路。
219-223：首先查看当前接口与对方主地址之间的链路，若该链路为对称链路，则节点与目标节点之间是对称连接，中断查找。
224-229：若当前接口与目标节点的主地址之间不是对称链路，则查找 mid 记录，获取该节点的其他所有接口，逐一检查是否是对称链路。

5.2.2 update_link_entry 函数

函数功能：OLSR 在收到一个 HELLO 消息时调用该函数更新节点到 HELLO 消息发送节点之间的链路状态，返回更新或创建的链路条目。

link_set.c
```
686 struct link_entry *
687 update_link_entry(const union olsr_ip_addr *local, const union olsr_ip_addr *remote, const struct
hello_message *message, const struct interface *in_if){
689     struct link_entry *entry;
690     entry =
691       add_link_entry(local, remote, &message->source_addr, message->vtime, message->htime, in_if);
692     entry->vtime = message->vtime;
693     entry->ASYM_time = GET_TIMESTAMP(message->vtime);
694     entry->prev_status = check_link_status(message, in_if);
695     switch (entry->prev_status) {
696         case (LOST_LINK):
697             olsr_stop_timer(entry->link_sym_timer);
698             entry->link_sym_timer = NULL;
699             break;
700         case (SYM_LINK):
701         case (ASYM_LINK):
702             olsr_set_timer(&entry->link_sym_timer, message->vtime, OLSR_LINK_SYM_JITTER,
                                OLSR_TIMER_ONESHOT, &olsr_expire_link_sym_timer, entry, 0);
703             olsr_set_link_timer(entry, message->vtime + NEIGHB_HOLD_TIME * MSEC_PER_SEC);
704             break;
705         default:;
706     }
707     if (entry->link_timer && (entry->link_timer->timer_clock < entry->ASYM_time)) {
708         olsr_set_link_timer(entry, TIME_DUE(entry->ASYM_time));
709     if (olsr_cnf->use_hysteresis)
710         olsr_process_hysteresis(entry);
711     update_neighbor_status(entry->neighbor, get_neighbor_status(remote));
712     return entry;
713 }
```
link_set.c

690-693：函数根据 HELLO 消息中的地址字段在链接集合中新建从本节点到该地址的条目并设置该条目的有效时间。
694：根据 HELLO 消息解析当前链路的状态，
696-699：如果当前链接状态变更为失去链接，则取消该链接条目的计时器
700-704：如果当前链路状态变更为有效链接(对称链接或非对称链接),则为该消息条目根据消息的有效时间设置定时器。
711：根据以上进行的改变更新连接表中对应的信息。

5.2.3 check_link_status 函数

通过查看接收到的 HELLO 消息检查到邻居的链接状态。

link_set.c

```
773 static int check_link_status(const struct hello_message *message, const struct interface *in_if){
774     int ret = UNSPEC_LINK;
775     struct hello_neighbor *neighbors;
776     neighbors = message->neighbors;
777     while (neighbors) {
778         if (ipequal(&neighbors->address, &in_if->ip_addr) && neighbors->link != UNSPEC_LINK) {
779             ret = neighbors->link;
780             if (SYM_LINK == ret) {
781                 break;
782             }
783         }
784         neighbors = neighbors->next;
785     }
786     return ret;
787 }
```

link_set.c

775-776：函数只取 HELLO 消息的 neighbors 部分，从中找出满足条件的信息。

777-784：对 neighbors 部分数据进行遍历。如果某一邻居条目的地址等于收到该 HELLO 消息的接口地址，说明该条目描述的是接收接口到 HELLO 消息产生节点的链接。取出该链接的链接状态并判断该链接是不是被当作对称链接，如果是则不可能有更好的链接状态，将该链接状态返回，否则继续向下一个条目寻找，直至找到一个到接收接口的对称记录或遍历完整个 neighbors 部分。

6 MPR 算法

OLSR 路由协议与传统的链路状态路由算法最显著的区别在于 MPR 的引入，一个节点通过 HELLO 等消息获取周围邻居节点信息后首先选出一定被称作 MPR 的邻居节点，当节点转发消息时，只向这些节点之一进行转发：

MPR 算法的大致流程如下：

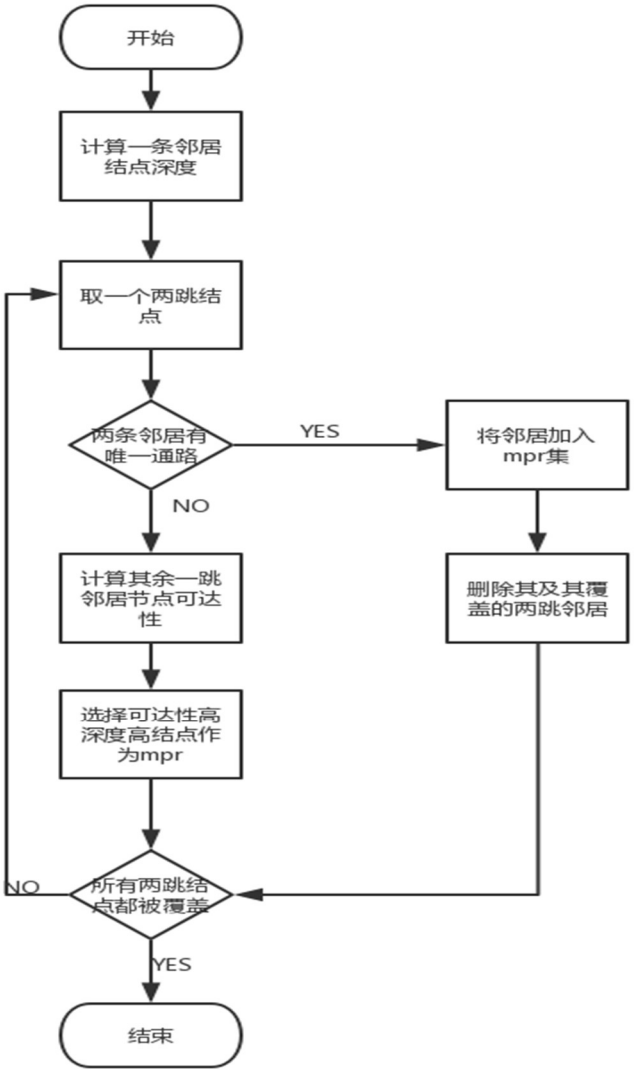

图 6.1 MPR 算法大致流程

6.1 MPR 计算过程

mpr.c
```
384 void
385 olsr_calculate_mpr(void){
386     uint16_t two_hop_covered_count;
387     uint16_t two_hop_count;
388     int i;
389     olsr_clear_mprs();
390     two_hop_count = olsr_calculate_two_hop_neighbors();
391     two_hop_covered_count = add_will_always_nodes();
392     for (i = WILL_ALWAYS - 1; i > WILL_NEVER; i--) {
393         struct neighbor_entry *mprs;
394         struct neighbor_2_list_entry *two_hop_list = olsr_find_2_hop_neighbors_with_1_link(i);
395         while (two_hop_list != NULL) {
396             struct neighbor_2_list_entry *tmp;
397             if (!two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor->is_mpr)
398                 olsr_chosen_mpr(two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor,
399                                 &two_hop_covered_count);
400             tmp = two_hop_list;
401             two_hop_list = two_hop_list->next;;
402             free(tmp);
403         }
404         if (two_hop_covered_count >= two_hop_count) {
405             i = WILL_NEVER;
406             break;
407         }
408         while ((mprs = olsr_find_maximum_covered(i)) != NULL) {
409             olsr_chosen_mpr(mprs, &two_hop_covered_count);
410             if (two_hop_covered_count >= two_hop_count) {
411                 i= WILL_NEVER;
412                 break;
413             }
414         }
415     }
416     olsr_optimize_mpr_set();
417     if (olsr_check_mpr_changes()) {
418         if (olsr_cnf->tc_redundancy > 0)
419         signal_link_changes(true);
420     }
421 }
```
mpr.c

389-391：计算 MPR 集合的准备工作，清空原本的 MPR 集合，计算所有两跳节点个数，并将转发意愿为 WILL_ALWAYS 的节点选作 MPR，随后根据转发意愿逐级计算选择 MPR。

394-402：找一二跳邻居节点，判断该二跳邻居节点到本节点之间是否有为题通路，若有，则将该通路上的一跳邻居节点选为 MPR 并从待选集合中删除被选中的节点。

404-406：判断是否所有的二跳节点都被覆盖，若是则结束计算。

408-414：从一跳邻居节点集合中选择覆盖最多二跳邻居节点的节点，将其选作 MPR，判断是否全覆盖，从而决定是否结束算法。

416：选取完所有 MPR 之后进行 MPR 集合的优化。

417-420：检查 MPR 集合是否发生变化，若发生变化且全局配置文件中为 TC 包设置了一定的冗余度（为了提高协议的健壮性），则产生一个信号通知网络处理发生的改变。

6.2 add_will_always_nodes 函数

在 OLSR 网络构造阶段中传播的 HELLO 消息存在 willness 字段，当这个字段被设置为 WILL_ALWAYS 时，该节点一定会被其邻居节点选作 MPR。

因此，在 MPR 算法的初期，首先要遍历邻居节点集合，找出 willness 字段为 WILL_ALWAYS 的节点，将其加入 MPR 集合。

mpr.c
```
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_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
```
mpr.c

367-374 对节点的所有邻居节点集合进行遍历，找出其中正常工作的并且 willness 为 WILL_ALWAYSd 的节点，将其选作 MPR 并更新覆盖节点的数量。

6.3 olsr_find_2_hop_neighbors_with_1_link 函数

函数功能： 根据给定的 willingness 值找到所有仅有唯一通路的二跳邻居

mpr.c
```
86 static struct neighbor_2_list_entry *
87 olsr_find_2_hop_neighbors_with_1_link(int willingness)
88 {
89   uint8_t idx;
90   struct neighbor_2_list_entry *two_hop_list_tmp = NULL;
91   struct neighbor_2_list_entry *two_hop_list = NULL;
92   struct neighbor_entry *dup_neighbor;
93   struct neighbor_2_entry *two_hop_neighbor = NULL;
94   for (idx = 0; idx < HASHSIZE; idx++) {
95    for (  two_hop_neighbr = two_hop_neighbortable[idx].next;
96           two_hop_neighbor != &two_hop_neighbortable[idx];
97           two_hop_neighbor = two_hop_neighbor->next) {
98         dup_neighbor = olsr_lookup_neighbor_table(&two_hop_neighbor->neighbor_2_addr);
99         if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
100            continue;
101          }
102          if (two_hop_neighbor->neighbor_2_pointer == 1) {
103            if ((two_hop_neighbor->neighbor_2_nblist.next->neighbor->willingness == willingness)
104                  && (two_hop_neighbor->neighbor_2_nblist.next->neighbor->status == SYM)) {
105              two_hop_list_tmp = olsr_malloc(sizeof(struct neighbor_2_list_entry), "MPR two hop list");
106              two_hop_list_tmp->neighbor_2 = two_hop_neighbor;
107              two_hop_list_tmp->next = two_hop_list;
108              two_hop_list = two_hop_list_tmp;
109            }
110          }
111     }
112   }
113   return (two_hop_list_tmp);
114 }
```
mpr.c

89-93：函数局部变量的声明，其中最终返回的结果以单向链表的形式存储在变量 two_hop_list_tmp 中。

94-111：节点遍历所有二跳节点集合从中挑选出满足条件的节点加入结果链表中。

98-101：判断二跳节点是否与节点具有唯一通路的方法是在节点的邻居表中寻找该二跳节点的邻居，如果存在该二跳节点的邻居且该邻居为对称节点，则说明还有其他通路，不满足结果条件。

102-108：继续判断该节点的 willingness 是否满足条件。若满足条件则通过 olsr_malloc 函数为存储该节点的信息分配一块内容，并将该内存加入到结果集合中。

6.4 olsr_chosen_mpr 函数

当一个节点被选作 MPR 时，OLSR 调用 olsr_chosen_mpr 函数更新邻居表中该节点的标志信息，并更新中已经覆盖的二跳节点数量。其中参数 *one_hop_neighbor* 为被选作 MPR 的节点条目，two_hop_covered_count 为当前覆盖的二跳节点计数。

mpr.c

```
131 static int
132 olsr_chosen_mpr(struct neighbor_entry *one_hop_neighbor, uint16_t * two_hop_covered_count)
133 {
134     struct neighbor_list_entry *the_one_hop_list;
135     struct neighbor_2_list_entry *second_hop_entries;
136     struct neighbor_entry *dup_neighbor;
137     uint16_t count;
138     struct ipaddr_str buf;
139     count = *two_hop_covered_count;
140     one_hop_neighbor->is_mpr = true;
141     for (second_hop_entries = one_hop_neighbor->neighbor_2_list.next;
          second_hop_entries != &one_hop_neighbor->neighbor_2_list;
          second_hop_entries = second_hop_entries->next) {

142               dup_neighbor =
                        olsr_lookup_neighbor_table(&second_hop_entries->neighbor_2->neighbor_2_addr);
143               if ((dup_neighbor != NULL) && (dup_neighbor->status == SYM)) {
144                     continue;
145               }
146               second_hop_entries->neighbor_2->mpr_covered_count++;
147               the_one_hop_list = second_hop_entries->neighbor_2->neighbor_2_nblist.next;
148               if (second_hop_entries->neighbor_2->mpr_covered_count >= olsr_cnf->mpr_coverage)
149                     count++;
150         while (the_one_hop_list != &second_hop_entries->neighbor_2->neighbor_2_nblist) {
151               if ((the_one_hop_list->neighbor->status == SYM)) {
152                     if (second_hop_entries->neighbor_2->mpr_covered_count >=
                              olsr_cnf->mpr_coverage) {
153                           the_one_hop_list->neighbor->neighbor_2_nocov--;
154               }
155         }
156         the_one_hop_list = the_one_hop_list->next;
157     }
158     *two_hop_covered_count = count;
159     return count;
160     }
```

mpr.c

140：将被选作 MPR 节点的标志变量 *is_mpr* 置 true 表示该节点已经被选为 MPR。

141-157：遍历该节点连接的所有二跳邻居节点

142-144：在路由表中查找该二跳邻居节点，如果路由表中已存在与该二跳邻居节点有关的条目，说明该节点已经被处理过，忽略该节点，开始对下一个二跳邻居节点的处理。

146：由于选择的 MPR 能到某个二跳邻居节点，故将该二跳邻居节点中可达的 MPR 计数增 1。

148-149：如果到达可该节点的 MPR 数高于全局配置中的数量，则认为该节点已经被覆盖，增加覆盖计数器。

150-155：由于我们已经选择了一个 MPR 节点，故应该在 OLSR 网络中将与该节点有关信息删除。即对该节点除当前被选作 MPR 节点的邻居进行遍历，查询其可达的二跳邻居节点，将相关可达计数减 1

6.5 olsr_find_maximum_covered 函数

函数功能：在邻居表中根据传播意愿找到可以覆盖最多二跳邻居的节点

mpr.c

```
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   maximum = 0;
213   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
214       if ((!a_neighbor->is_mpr) && (a_neighbor->willingness == willingness)
                && (maximum < a_neighbor->neighbor_2_nocov)) {
215                   maximum = a_neighbor->neighbor_2_nocov;
216                 mpr_candidate = a_neighbor;
217       }
218   }
219   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
220   return mpr_candidate;
221 }
```

mpr.c

209-221：对所有邻居节点进行遍历，寻找能连通最多二跳邻居的节点。邻居所能连通的二跳节点数量储存在邻居节点的 *neighbor_2_nocov* 中。一旦发现某个节点的连通数量比当前记录值还高，就将该节点值赋给最终返回的结果变量，并更新最高连通数量，典型的在集合中寻找最大值的算法。

6.6 olsr_check_mpr_changes 函数

函数功能: 检查当前节点的 MPR 集合是否发生过变化，若变化过返回 1，不变返回 0

mpr.c

```
260   static int
261   olsr_check_mpr_changes(void)
262   {
263   struct neighbor_entry *a_neighbor;
264   int retval;
265   retval = 0;
266   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
267         if (a_neighbor->was_mpr) {
268             a_neighbor->was_mpr = false;
269             if (!a_neighbor->is_mpr) {
270                   retval = 1;
271             }
272         }
273       }
274   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
275   return retval;
276   }
```

mpr.c

266-274：遍历所有邻居阶段，对于邻居节点 *a_neighbor* 如果它曾经是 MPR，但是现在不是 MPR，即 *was_mpr* 被置 1, is_mpr 置 0，则它的 MPR 状态发生变化，将 *retval* 置为 1，否则返回 retval 默认值 0。

6.7 olsr_optimize_mpr_set 函数

当计算所有 MPR 之后，如果 MPR 有冗余，即删去某些 MPR 之后仍然可以覆盖所有二跳邻居节点，可以调用 olsr_optimize_mpr_set 函数进行优化处理。

mpr.c

```
458 static void
459 olsr_optimize_mpr_set(void)
460 {
461     struct neighbor_entry *a_neighbor, *dup_neighbor;
462     struct neighbor_2_list_entry *two_hop_list;
463     int i, removeit;
464     for (i = WILL_NEVER + 1; i < WILL_ALWAYS; i++) {
465         OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
466         if (a_neighbor->willingness != i) {
467                 continue;
468         }
469         if (a_neighbor->is_mpr) {
470             removeit = 1;

471             for (two_hop_list = a_neighbor->neighbor_2_list.next;
472             two_hop_list !=&a_neighbor->neighbor_2_list;
473             two_hop_list = two_hop_list->next) {
474             dup_neighbor =
                olsr_lookup_neighbor_table(&two_hop_list->neighbor_2->neighbor_2_addr);
475             if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
476                     continue;
477             }
478             if (two_hop_list->neighbor_2->mpr_covered_count <= olsr_cnf->mpr_coverage) {
479                         removeit = 0;
480             }
481         }
482         if (removeit) {
483               a_neighbor->is_mpr = false;
484           }
485         }
486       } OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
487   }
488 }
```

mpr.c

464-487：节点对所有邻居节点按照转发意愿逐级遍历，删除冗余节点。

469-486：首先找到一个被设置为 MPR 的节点。并遍历该经由该节点的二跳邻居，直指找到路由表项上存在的二跳节点，如果该二跳邻居节点已经有足够的 MPR 覆盖，则从 MPR 集合中删除该节点。

7 路由计算

节点根据 HELLO 消息和 MID 消息探知周围的邻居节点，随后通过 TC 消息洪范探知整个网络的拓扑，最终独立地根据 Dijkstra 算法计算出路由表，不对除路由表之外的信息作任何修改。当这些消息发生变化时，节点会再次进行路由计算。

7.1 全局变量：

```
struct olsr_cookie_info *rt_mem_cookie
struct olsr_cookie_info *rtp_mem_cookie
static struct rt_path *current_inetgw
struct avl_tree routing_tree
unsigned int routingtree_version
```

其中 *rt_mem_cookie* 和 *rtp_mem_cookie* 分别是 OLSR 为路由节点和路径分配的 cookie。

current_inetgw 用来追踪当前的网关。网关的切换会导致所有 NAT 连接中 TCP 和 UDP 会话信息全部丢失，因此我们在切换网关的优势足以弥补丢失所有会话信息的损失时才会进行切换。

routing_tree 是当前路由 *avl* 树的根节点。

routingtree_version 用来检测路由树中的信息是否过时。

7.2 路由表基本操作

7.2.1 olsr_init_routing_table 函数

函数功能：对路由相关变量进行初始化

routing_table.c

```
167 void olsr_init_routing_table(void){
168     avl_init(&routingtree, avl_comp_prefix_default);
169     routingtree_version = 0;
170     rt_mem_cookie = olsr_alloc_cookie("rt_entry", OLSR_COOKIE_TYPE_MEMORY);
171     olsr_cookie_set_memory_size(rt_mem_cookie, sizeof(struct rt_entry));
172     rtp_mem_cookie = olsr_alloc_cookie("rt_path", OLSR_COOKIE_TYPE_MEMORY);
173     olsr_cookie_set_memory_size(rtp_mem_cookie, sizeof(struct rt_path));
174 }
```

routing_table.c

168：对路由树进行初始化，将节点比较函数置为 avl_comp_preifx_default，即根据节点中 IP 地址的前缀对节点进行排序。

170-173：在内存中为两个 cookie 分配大小并指明单个 cookie 的大小。

7.2.2 olsr_alloc_tc_entry 函数

函数功能：为通往目标地址的路由分配一个条目，插入到全局的路由记录中，并返回新创建的条目。

routing_table.c

```
231 static struct rt_entry *
232 olsr_alloc_rt_entry(struct olsr_ip_prefix *prefix){
233     struct rt_entry *rt = olsr_cookie_malloc(rt_mem_cookie);
234     if (!rt) {
235         return NULL;
236     }
237     memset(rt, 0, sizeof(*rt));
238     rt->rt_nexthop.iif_index = -1;
239     rt->rt_dst = *prefix;
240     rt->rt_tree_node.key = &rt->rt_dst;
241     avl_insert(&routingtree, &rt->rt_tree_node, AVL_DUP_NO);
242     avl_init(&rt->rt_path_tree, avl_comp_default);
243     return rt;
244 }
```

routing_table.c

233-234：根据对应 cookie 申请一块内存空间，如果没有足够的空间则中止函数处理。

237：将申请到的内存空间清零。

238-240：标识该条目为新建立的条目并将条目的目的地址设置为参数传入地址。

241-242：将该条目挂载到全局的路由树中并对该条目的路由子树的比较函数进行初始化。

7.2.3 olsr_alloc_rt_path 函数

函数功能：为一个路由路径记录分配内存并挂载到 TC 条目上。

routing_table.c

```
249 static struct rt_path *
250 olsr_alloc_rt_path(struct tc_entry *tc, struct olsr_ip_prefix *prefix, uint8_t origin){
251     struct rt_path *rtp = olsr_cookie_malloc(rtp_mem_cookie);
252     if (!rtp) {
253         return NULL;
254     }
255     memset(rtp, 0, sizeof(*rtp));
256     rtp->rtp_dst = *prefix;
257     rtp->rtp_prefix_tree_node.key = &rtp->rtp_dst;
258     avl_insert(&tc->prefix_tree, &rtp->rtp_prefix_tree_node, AVL_DUP_NO);
259     olsr_lock_tc_entry(tc);
260     rtp->rtp_tc = tc;
261     rtp->rtp_origin = origin;
262     return rtp;
263 }
```

routing_table.c

251-255：函数尝试在 *cookie* 中为新建的 *rtp* 条目分配内存空间，并将该块内存置 0。

256-257：将 *rtp* 的目的地址和地址前缀树的 key 值置为参数传入的地址，并将 rtp 中的地址前缀树挂载到 TC 条目的地址树中。

259：新增了与该 TC 条目有关的一条路径，故调用 olsr_lock_tc_entry 函数将该 TC 条目的引用数增 1。

260：建立起该路径与条目的反向链接，便于通过路径查找到对应的 TC 条目。

7.2.4 olsr_cmp_rtp 函数

函数功能：比较两路径哪一条更优，若 *rtp1* 优于 *rtp2* 则返回 true，反之返回 false。

routing_table.c

```
412 static bool
413 olsr_cmp_rtp(const struct rt_path *rtp1, const struct rt_path *rtp2, const struct rt_path *inetgw){
414     olsr_linkcost etx1 = rtp1->rtp_metric.cost;
415     olsr_linkcost etx2 = rtp2->rtp_metric.cost;
416     if (inetgw == rtp1)
417         etx1 *= olsr_cnf->lq_nat_thresh;
418     if (inetgw == rtp2)
419         etx2 *= olsr_cnf->lq_nat_thresh;
420     if (etx1 < etx2)
421         return true;
422     if (etx1 > etx2)
423         return false;
424     if (rtp1->rtp_metric.hops < rtp2->rtp_metric.hops)
425         return true;
426     if (rtp1->rtp_metric.hops > rtp2->rtp_metric.hops)
427         return false;
428     if (memcmp(&rtp1->rtp_originator, &rtp2->rtp_originator, olsr_cnf->ipsize) < 0)
429         return true;
430     return false;
431 }
```

routing_table.c

414-427：函数首先比较该路径的花销，如果两路径花销不同则返回花销较小的路径更优，否则比较两条路径的跳数，跳数较少的路径更优，若二者都相同，则进行下一步比较。

428-430：最终比较两路径的源 IP 地址，地址较小的路径较优，由于 IP 地址在网络中唯一，因此一定能返回比较结果。

7.3 路由表计算

OLSR 通过调用 olsr_calculate_routing_table 函数计算路由表

(1) 变量信息

olsr_spf.c

```
311   struct avl_tree cand_tree;
312   struct avl_node *rtp_tree_node;
313   struct list_node path_list;              /* head of the path_list */
314   struct tc_entry *tc;
315   struct rt_path *rtp;
316   struct tc_edge_entry *tc_edge;
317   struct neighbor_entry *neigh;
318   struct link_entry *link;
319   int path_count = 0;
```

olsr_spf.c

311-312：函数为候选节点和已经选定的路由表分别分配一个 avl 树进行存储

313-318：计算路由时用到的各种描述路径、邻居节点的变量声明

319：路由路径统计量

(2) 当前路由表合法性状态判断

olsr_spf.c

```
322   if (!force) {
323       if (spf_backoff_timer) {
324           return;
325       }
326       spf_backoff_timer = olsr_start_timer(1000, 5, OLSR_TIMER_ONESHOT, &olsr_expire_spf_backoff,
NULL, 0);
327   }
```

olsr_spf.c

322-327：函数查看当前路由表的有效时间计时器是否正在工作，若是，则该路由表的信息仍在有效期内，无需计算。否则为该路由表设置一个新的计时器并开始计算路由。

(3) 计算过程

olsr_spf.c

```
349 olsr_change_myself_tc();
350   if (!tc_myself) {
351         olsr_update_rib_routes();
352         olsr_update_kernel_routes();
353         return;
354  }
355 tc_myself->path_cost = ZERO_ROUTE_COST;
356 olsr_spf_add_cand_tree(&cand_tree, tc_myself);
```

olsr_spf.c

349-354：检查自身节点的 IP 地址是否发生改变，如果没有配置主 IP 地址，则无法计算路由，更新路由表并中断函数。

355-356：将当前节点加入到候选节点树中

olsr_spf.c

```
358 OLSR_FOR_ALL_NBR_ENTRIES(neigh) {
359      if (neigh->status != SYM) {
360         tc_edge = olsr_lookup_tc_edge(tc_myself, &neigh->neighbor_main_addr);
361         if (tc_edge) {
362            olsr_delete_tc_edge_entry(tc_edge);
363         }
364      }
365      else {
366         tc_edge = olsr_lookup_tc_edge(tc_myself, &neigh->neighbor_main_addr);
367         link = get_best_link_to_neighbor(&neigh->neighbor_main_addr);
368         if (!link || lookup_link_status(link) == LOST_LINK) {
369            if (tc_edge) {
370               olsr_delete_tc_edge_entry(tc_edge);
371            }
372            continue;
373         }

374         if (link->if_name) {
375            link->inter = if_ifwithname(link->if_name);
376         } else {
377            link->inter = if_ifwithaddr(&link->local_iface_addr);
378         }

379         if (!tc_edge) {
380            tc_edge = olsr_add_tc_edge_entry(tc_myself, &neigh->neighbor_main_addr, 0);
381         } else {
382            olsr_copylq_link_entry_2_tc_edge_entry(tc_edge, link);
383            olsr_calc_tc_edge_entry_etx(tc_edge);
384         }

385         if (tc_edge->edge_inv) {
386            tc_edge->edge_inv->tc->next_hop = link;
387         }
388      }
389   }
390   OLSR_FOR_ALL_NBR_ENT RIES_END(neigh);
```

olsr_spf.c

将所有与邻居节点相关联的边加入计算

359-364：若当前邻居节点的状态为非对称节点，则删除与该节点相关的 TC 边记录（OLSR 只考虑对称节点）。

366-367：若当前邻居节点为对称节点，则查找当前节点与该邻居节点连接的边记录，提取链接状态。

368-372：如果当前链接损坏，则在 TC 边记录集合中将该边删除，继续寻找下一个邻居节点。

374-378：寻找该链接的接口信息。

379-384：将该链接记录加入到记录集合，并指定到邻居节点的路由的下一跳地

址。

olsr_spf.c

```
406    olsr_spf_run_full(&cand_tree, &path_list, &path_count);
...
417    for (; !list_is_empty(&path_list); list_remove(path_list.next)) {
418          tc = pathlist2tc(path_list.next);
419          link = tc->next_hop;
420          if (!link) {
421                continue;
422          }
423          for (rtp_tree_node = avl_walk_first(&tc->prefix_tree); rtp_tree_node; rtp_tree_node =
             avl_walk_next(rtp_tree_node)) {
424                rtp = rtp_prefix_tree2rtp(rtp_tree_node);
425                if (rtp->rtp_rt) {
426                      olsr_update_rt_path(rtp, tc, link);
427                } else {
428                      olsr_insert_rt_path(rtp, tc, link);
429                }
430          }
431    }
432    olsr_update_rib_routes( );
433    olsr_update_kernel_routes( );
```

olsr_spf.c

406：根据 Dijkstra 算法算出到每个节点的最短路径，储存在 path_list 中
417-431：计算完到网络拓扑中各节点的路径后更新路由表。首先将该路径转换为一条 TC 记录，并取出下一跳的链接，如果该链接为空，则该节点不可达，不对该项路由进行修改。若存在该链接则在路由表中加入或更新该项路由。
432-433：将路由表更新至内核中。

8 总结

本文首先简要介绍了 OLSR 的特点及其对标准链路状态路由协议的优化情况。通过整体框架图可以整体了解 OLSR 协议的整个的工作流程。

我们小组对 OLSR 协议的数据结构，其中包括消息结构和存储结构进行了简要的分析。我们小组同时对 HELLO 消息，TC 消息，MID 消息的结构、消息的产生、消息的接收、消息的处理部分的代码以及邻居表操作、MPR 算法，以及路由计算部分的代码进行了分析。

在分析过程中我们发现了 OLSR 协议确实对标准链路状态路由协议进行了优化：OLSR 协议通过拓扑控制信息的产生和洪泛，显著地减少网络中广播的控制分组数量，OLSR 很大程度上减少了转发的信息。在 OLSR 协议中，链路状态信息都是由被挑选为 MPRs 的节点产生的，这样减少了在网络中洪泛的控制信息。并且 MPR 节点只选择在 MPR 或者 MPR Selector 之间传递链接状态信息。

当然，在代码分析过程中，我们也发现了 OLSR 协议中存在的部分问题：OLSR 路由协议的链路状态更新只采用周期性更新，这与在链路状态变化时采用的触发式更新相比，路由的收敛性与强壮性显得逊色。OLSR 路由协议，所有不同移动速度及不同距离的节点都同等对待，这明显存在不合理性。

但是我们可以看到，身处当代，随着人类的不断发展，技术的不断创新，无线网络技术的运用越来越广泛。而我们作为网络专业的学生，这正是我们应该研究的问题，同时我们也希望 OLSR 能继续不断发展，在今后的网络中发挥越来越重要的作用。