网络协议栈分析及设计大作业

OLSR 协议分析

27 组小组成员及分工			
姓名	班级	学号	分工
王子豪	软网 1703	201792232	RFC 分析：0-2，6-8
			代码分析及论文撰写：1，3，5.3，5.5
李赞	软网 1703	201792355	RFC 分析：3-5
			代码分析及论文撰写：2，4.2，5.2，5.4
王宇	软网 1703	201792301	RFC 分析：9-11
			代码分析及论文撰写：4.1，5.1，6

大连理工大学

Dalian University of Technology

目　　录

1 OLSR 协议简介

1.1 OLSR 基本概念

OLSR 是 Optimized Link State Routing 的简称，是一种优化的链路状态路由协议，主要用于 MANET 网络（移动自组织网络）的路由协议。它是一种标准化的表驱动式路由协议，为了适应无线自组网的需求，对经典链路状态算法进行化而形成的。协议用到的核心概念是多点中继机制，通过此机制，协议显著的减少了广播消息的开支。总体来讲，协议对传统链路状态算法所做的优化主要有:通过采用多点中继机制,有效减小了控制分组的泛范围;节点在其所有的邻节点中，选择部分节点作为其多点中继节点 MPR。对比经典的洪泛机制中每节点当其第一次收到每一个控制消息时转发每一个消息，OLSR 协议中只有 MPR 节点才转发该节点发送的控制分组,而其他非 MPR 节点只进行处理不进行转发。这样就显著的减少了网络中广播的控制分的数量，避免了广播风暴。其次是缩减了控制分组的大小。节点并不发布与所有邻邻居节点相连的链路信息而只发布与其 MRP 节点间的链路。OLSR 协议中，每个控制分组都携带有序列号，可以用来区分旧信息，所以协议不要求按序传输控制分组。协议通过节点周期性的发送 TC 分组来发布 MPRSelector 消息，以帮助其他节点建立网络拓扑结构来形成到它的路由，并通过周期性的交换信息来维护网络拓扑。网络中的每个节点都存有到网络中所有可达目节点的路由，因此 OLSR 协议特别适用于网络规模大、节点分布密集的网络。

1.2 MPR 机制

OLSR 协议的改进之处就是引入了 MPR 机制，目的是为了防止网络洪泛以及减少开销，每个节点会在自己的一跳邻居中选择一部分节点作为自己的 MPR 节点，此节点发送的广播消息只有 MPR 节点可以转发，其他节点只能接收和处理，MPR 节点越少，网络的性能越高，同时，被选为 MPR 的节点维护者一个 MPR Selector 集合，这个集合记载了有那些节点将自己选为 MPR，这个集合会随 TC 消息，由 MPR 节点广播到网络中来帮助所有节点形成网络的拓扑结构来建立路由。

引入 MPR 的洪泛机制如下图所示：

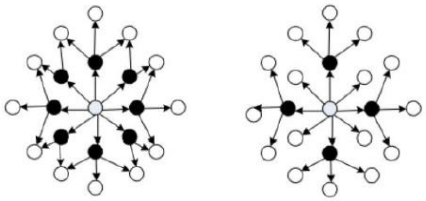


图 无 MPR 广播和使用 MPR 机制的广播

1.3 OLSRd

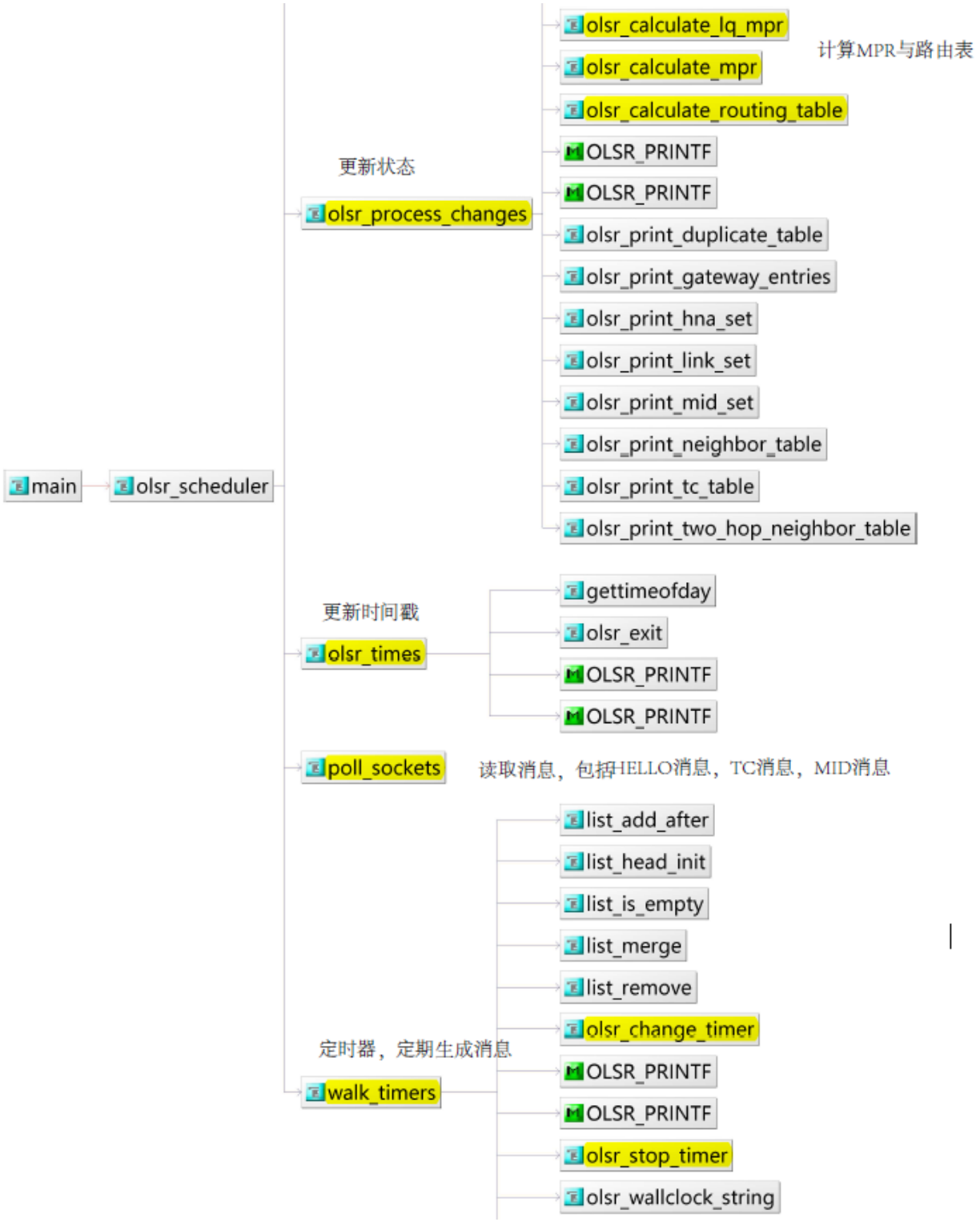
传统的 OLSR 默认每两个邻居之间的距离都是 1，但是有时候一个跳数更多的路径可能有更好的链路质量，这样使用 OLSR 可能就会做出错误的 MPR 选择和路由建立，但是 OLSRd 机制引入了链路质量（Link Quility）这个要素，现在，不同的邻居之间的距离（distance）变得不一样了，可以更好的进行 MPR 及路由选择。具体算法会在 5.3.2 中阐述。

2 协议示意图

2.1 main 函数的函数调用图

我们利用 Source Insight，对 main 函数生成关系调用图，高亮并重点研究调用的与协议工作流程相关的函数。其中，olsr_scheduler 函数体现了协议的工作过程，因此将会着重介绍它的流程。

2.2 olsr_scheduler 函数调用图



在 olsr_scheduler 中，执行顺序为 olsr_times — poll_sockets — walk_timers — olsr_process_changes.每次读取消息或生成消息前，协议更新时间戳，接收到读取的消息处理它，更新信息表。

3 文件及变量

3.1 文件

文件名	描述
link_set.c/h	链路表相关操作
packet.c/h	消息包的数据结构定义
mpr.c	MPR 相关的一些操作
mpr_selector_set.c	对 MPR Selector 集的定义及操作
neighbor _table.h	邻居表的定义
neighbor _table.c	对一跳邻居及二跳邻居的处理
olsr_cfg.h	定义许多全局变量
olsr.c	定义许多全局函数
olsr_spf.c	构造 spf 数来利用迪杰斯特拉算法建立路由表
routing_table.c	路由表的处理操作
TC_set.c	TC 表相关操作

3.2 全局变量

变量名	类型	描述
olsrport	uint16_t	OLSR 消息发送端口
rt_proto	uint8_t	采用的路由协议
min_tc_vtime	float	TC 消息 vtime 最小值
max_tc_vtime	float	TC 消息 vtime 最大值
changes_topology	bool	拓扑控制消息是否变化

变量名	类型	描述
changes_neighborhood	bool	邻居表信息是否变化
use_hysteresis	bool	判断消息是否延迟

3.3 配置变量

变量名	缺省值	描述
DEF_IP_VERSION	AF_INET	缺省 IP 协议域
DEF_USE_HYST	false	缺省消息延迟
DEF_LQ_LEVEL	2	缺省链路质量等级
DEF_OLSRPORT	698	缺省 olsr 端口号
DEF_MIN_TC_VTIME	0.0	TC 消息 vtime 最小取值
DEF_GW_TYPE	GW_UPLINK_IPV46	缺省网关类型
MAXMESSAGESIZE	1500kb	广播数据包最大大小
MAX_TTL	0.165ms	TTL 最大值
MAX_LQ_LEVEL	2	链路质量最高等级

4 OLSR 数据结构

4.1 消息结构

消息结构定义在 lq_packet.h 文件中：

4.1.1 消息通用格式

OLSR 有 HELLO、MID、TC 几种消息。为了促进可扩展性的同时，保证兼容性，OLSR 采取统一的数据包格式。即，消息共享一个通用的头格式，让节点可以正确接受和重传未知类型的消息。

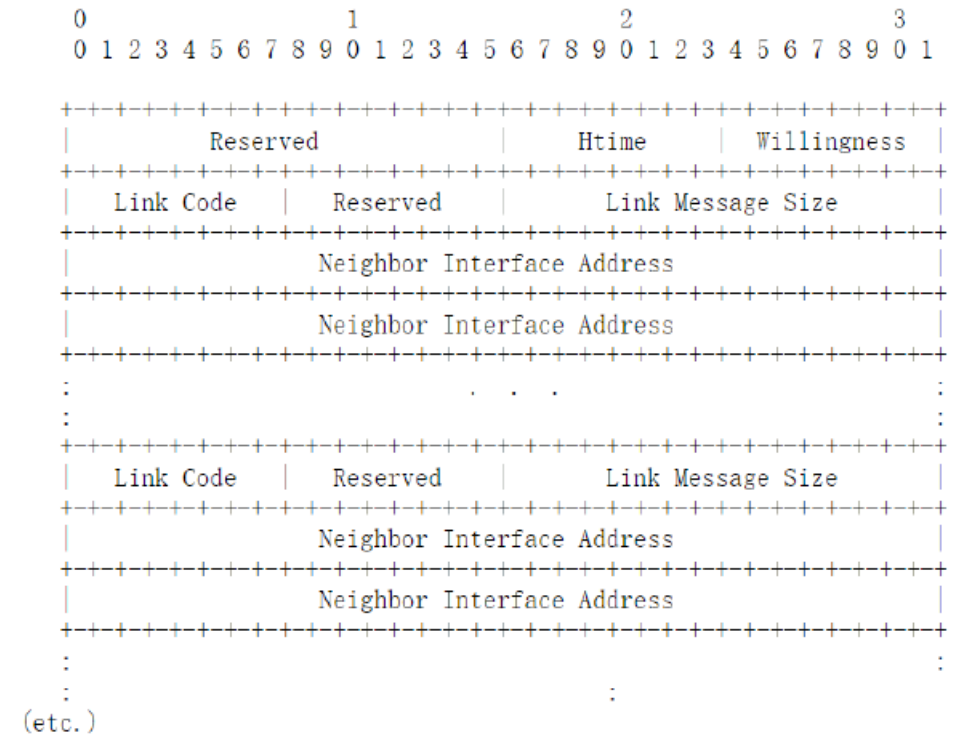
```
1.  struct olsr_common {
2.    uint8_t type;
3.    olsr_reltime vtime;
4.    uint16_t size;
5.    union olsr_ip_addr orig;
```

```
6.    uint8_t ttl;
7.    uint8_t hops;
8.    uint16_t seqno;
9.  };
```

规定了消息类型、有效时间、消息大小、源地址、TTL、跳数、序列号等公共数学。另外，OLSR 对 IPv4 与 IPv6，序列化了不同的头部。

4.1.2 HELLO 消息

HELLO 消息结构如图：



除了公共的头部属性，它还需要的属性如下：

```
1.  struct lq_hello_info_header {
2.    uint8_t link_code;
3.    uint8_t reserved;
4.    uint16_t size;
5.  };
6.  struct lq_hello_header {
7.    uint16_t reserved;
8.    uint8_t htime;
9.    uint8_t will;
10. };
```

- link_code:包括链路类型和邻居类型：
  - OLSR 规定的链路类型（link type）有以下四种

- UNSPEC_LINK ： 表示没有给出关于连接的特定信息（unspecified）
- ASYM_LINK ： 表示链路是非对称的
- SYM_LINK： 表示链路和接口是对称的
- LOST_LINK： 连接丢失

- OLSR 规定的邻居类型（Neighbor Type）有以下三种
  - SYM_NEIGH： 表示邻居至少有一个关于此节点的对称链接。
  - MPR_NEIGH： 表示邻居具有至少一个对称连接并且已被发送方选择为 MPR。
  - NOT_NEIGH： 无对称邻居

- reserved:保留字段
- size:消息大小
- htime:HELLO 消息发送的时间间隔
- will:意愿值，节点是否被愿意、有多愿意被选为 MPR

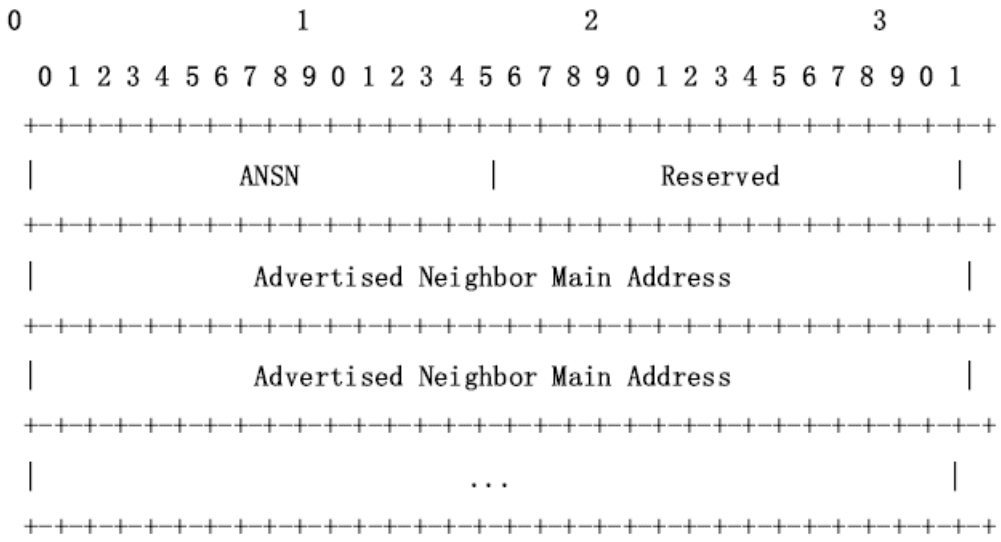
头部之后，HELLO 消息由邻居接口的地址构成，广播 HELLO 消息的同时完成地址交换，从而实现链路检测、邻居发现及 MPR 选择。
值得一提的是，HELLO 消息不可被转发，只能被发给邻居。

4.1.3 MID 消息

MID 消息，即多接口消息。它没有特殊的头定义，只需要把通用消息头里的消息类型设置为 MID 消息就可以。它包括除该节点的主地址外任何其他接口的地址，因为主地址已经在共通格式的源地址字段声明了。根据它的内容，我们可以知道，MID 消息被用来声明一个节点的多个接口，所以它的 TTL 应设置为 255（最大值）以将消息传播到整个网络。

4.1.4 TC 消息

TC 消息结构如图：



头部附加定义如下：

```
1.  struct lq_tc_message {
2.    struct olsr_common comm;
3.    union olsr_ip_addr from;
4.    uint16_t ansn;
5.    struct tc_mpr_addr *neigh;
6.  };
```

ansn: 本节点收到的最近一个 TC 分组的序列号，用于防重。

TC 消息的主要内容是邻居节点的主地址，因此广播 TC 消息可以实现拓扑表的填充。

4.2 表结构

每个节点都维护着一些表项，这些表中记录着有关其他节点以及网络状态的一些关键信息，这些表包括：

- 链路信息表：该节点与邻居节点的链路信息
- 一跳邻居表：该节点的一跳邻居信息
- 二跳邻居表：该节点的二跳邻居信息
- MPR Selector 表：选择该节点为 MPR 节点的节点的信息
- 拓扑控制信息表：包含该节点已知的 MPR 选择信息
- 路由表：描述通过该节点的数据包该往何处转发

接下来，我们将通过代码分析每个表的内容及作用。

4.2.1 链路信息表

本地链路信息表存储了该节点和邻居节点的链路信息。

```
1.  struct link_entry {
2.    union olsr_ip_addr local_iface_addr;
3.    union olsr_ip_addr neighbor_iface_addr;
4.    const struct interface *inter;
5.    char *if_name;
6.    struct timer_entry *link_timer;
7.    struct timer_entry *link_sym_timer;
8.    uint32_t ASYM_time;
9.    olsr_reltime vtime;
10.   struct neighbor_entry *neighbor;
11.   uint8_t prev_status;
12.
13.   /*
14.    * Hysteresis
15.    */
16.   float L_link_quality;
17.   int L_link_pending;
18.   uint32_t L_LOST_LINK_time;
19.   struct timer_entry *link_hello_timer; /* When we should receive a new HELLO */
20.   olsr_reltime last_htime;
21.   bool olsr_seqno_valid;
22.   uint16_t olsr_seqno;
23.
24.   /*
25.    * packet loss
26.    */
27.   olsr_reltime loss_helloint;
28.   struct timer_entry *link_loss_timer;
29.
30.   /* user defined multiplies for link quality, multiplied with 65536 */
31.   uint32_t loss_link_multiplier;
32.
33.   /* cost of this link */
34.   olsr_linkcost linkcost;
35.
36.   struct list_node link_list;          /* double linked list of all link entries */
37.   uint32_t linkquality[0];
38. };
```

L_local_iface_addr	L_neighbor_iface_addr	L_SYN_time	L_ASYN_time	L_time

本地链路信息表格式

第 2 行，定义 local_iface_addr 记录本地节点的接口地址。

第 3 行，定义 neighbor_iface_addr 记录邻节点的接口地址。

第 8 行，使用 ASYN_time 来确保在此之前的链路被认为是单向的。
使用 SYN_time 来确定在此之前，链路被认为是对称的。
第 6 行，定义 link_timer 链路维护时刻，链路在该时刻失效，必须被删除。
当 SYN_time 和 ASYN_time 都过期，链路被声明为丢失。
第 19 行，定义一个 link_hello_timer 结构体指针，用来判定何时接收 Hello 消息
第 33 行，利用 linkcost 计算节点开销

4.2.2 一跳邻居表

每个节点根据接收和发送 HELLO 分组获得关于其两跳以内的邻居的信息，维护着一个一跳邻居表。

```
1.  struct neighbor_entry {
2.    union olsr_ip_addr neighbor_main_addr;
3.    uint8_t status;
4.    uint8_t willingness;
5.    bool is_mpr;
6.    bool was_mpr;                         /* Used to detect changes in MPR */
7.    bool skip;
8.    int neighbor_2_nocov;
9.    int linkcount;
10.   struct neighbor_2_list_entry neighbor_2_list;
11.   struct neighbor_entry *next;
12.   struct neighbor_entry *prev;
13. };
```

N_neighbor_main_addr	N_status	N_willingness

邻居表格式

第 1 行，利用 neighbor_main_addr 记录节点 i 的一跳邻居地址。
第 2 行，status 判断节点 i 与其一跳邻居之间的链路状态。
取值可为 ASYM_LINK 、SYM_LINK 或 MPR_LINK 。链路状态 MPR_LINK 意味着与邻居节点 N_neighbor_main_addr 的链路是对称的且节点被选择为 MPR 。
第 3 行，利用 willingness 来表示邻居节点为其他节点转发分组的意愿程度，包括 WILL_NEVER ， WILL_LOW ， WILL_DEFAULT ， WILL_HIGH ， WILL_ALWAYS 五个级别。
第 6 行，定义 bool 变量 was_timer 用于检测 MPR 中的更改

4.2.3 二跳邻居表

节点存储一个两跳邻居表，描述邻居节点与对称两跳邻节点间的对称链路。

```
1.  struct neighbor_2_list_entry {
2.    struct neighbor_entry *nbr2_nbr;     /* backpointer to owning nbr entry */
3.    struct neighbor_2_entry *neighbor_2;
4.    struct timer_entry *nbr2_list_timer;
5.    struct neighbor_2_list_entry *next;
6.    struct neighbor_2_list_entry *prev;
7.  };
```

N_neighbor_main_addr	N_2hop_addr	N_time

两跳邻居表格式

第 2 行，neighbor_entry 表示邻节点的地址。

第 3 行，neighbor_2_entry 表示与 N_neighbor_main_addr 有对称链路的两跳邻节点的地址。

第 4 行，timer_entry 表示表项到期必须被移除的时间.

4.2.3 MPR Selector 表

每个节点维护一个 MPR Selector 表，记录选择自己为 MPR 结点的结点列表

```
1.  struct mpr_selector {
2.    union olsr_ip_addr MS_main_addr;
3.    struct timer_entry *MS_timer;
4.    struct mpr_selector *next;
5.    struct mpr_selector *prev;
6.  };
```

MS_main_addr	MS_time

MPR Selector 表格式

第 2 行，MS_main_addr 表示 MPR Selector 节点的地址。

第 3 行，timer_entry 记录该 MPR Selector 集表项的保持时间，当 MPR Selector 集过期时要及时删除。

第 4、5 行，分别表示之前和之后结点

4.2.4 拓扑控制信息表

网络中每一个节点都维护一张拓扑表，表中记录了从 TC 分组获得的网络拓扑信息。节点根据这一信息计算路由表。节点将网络中其他节点的多点中继的信息作为拓扑表项记录在拓扑表中。

```
1.  struct tc_entry {
2.    struct avl_node vertex_node;          /* node keyed by ip address */
3.    union olsr_ip_addr addr;              /* vertex_node key */
4.    struct avl_node cand_tree_node;       /* SPF candidate heap, node keyed by path_etx
     */
5.    olsr_linkcost path_cost;              /* SPF calculated distance, cand_tree_node ke
    y */
6.    struct list_node path_list_node;      /* SPF result list */
7.    struct avl_tree edge_tree;            /* subtree for edges */
8.    struct avl_tree prefix_tree;          /* subtree for prefixes */
9.    struct link_entry *next_hop;          /* SPF calculated link to the 1st hop neighbo
    r */
10.   struct timer_entry *edge_gc_timer;    /* used for edge garbage collection */
11.   struct timer_entry *validity_timer;   /* tc validity time */
12.   uint32_t refcount;                    /* reference counter */
13.   uint16_t msg_seq;                     /* sequence number of the tc message */
14.   uint8_t msg_hops;                     /* hopcount as per the tc message */
15.   uint8_t hops;                         /* SPF calculated hopcount */
16.   uint16_t ansn;                        /* ANSN number of the tc message */
17.   uint16_t ignored;                     /* how many TC messages ignored in a sequence

18.                                            (kindof emergency brake) */
19.   uint16_t err_seq;                    /* sequence number of an unplausible TC */
20.   bool err_seq_valid;                   /* do we have an error (unplauible seq/ansn)
    */
21. };
```

T_dest_addr	T_last_addr	T_seq	T_time

拓扑控制信息表格式

4.2.5 TC 消息重记录表

MPR 节点广播和转发 TC 分组，在这个过程中，一个节点可能会多次收到同一个 TC 分组，为了避免重新处理已经收到并处理过的 TC 分组，每个节点维护一个 TC 分组重复记录表。

```
1.  struct dup_entry {
2.    struct avl_node avl;
3.    union olsr_ip_addr ip;
4.    uint16_t seqnr;
5.    uint16_t too_low_counter;
```

```
6.    uint32_t array;
7.    uint32_t valid_until;
8.  };
```

第 4 行，seqnr 表示 TC 分组序列号

弟 7 行，valid_until 该表项有效期

4.2.6 路由表

网络中每个节维护一个路由表，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。

```
1.  struct route_entry {
2.    union olsr_ip_addr gw;
3.    union olsr_ip_addr dst;
4.    olsr_u16_t hopcnt;
5.    char if_name[MAX_IF_NAMESIZ];
6.    struct route_entry *next;
7.    struct route_entry *prev;
8.  };
```

R_dest_addr	R_next_addr	R_dist	R_iface_addr

路由表格式

第 2 行，gw 用来表示网关 IP 地址

第 3 行，dst 表示目的结点 IP 地址

第 4 行，hopcnt 用来记录跳数

第 6、7 行，记录之前和之后的 route_entry

5 OLSR 协议运行流程

OLSR 协议运行的大概流程如下所示：

- 节点通过周期性的交换 HELLO 消息进行链路监测，构建链路集
- 通过 HELLO 消息及链路集构建邻居集
- 每个节点利用邻居集计算自己的 MPR 节点，建立 MPR 集和 MPR Selector 集
- 每个节点广播自己的 TC 消息，利用收到的其他节点的 TC 消息进行拓扑建立

- 节点利用已知的网络拓扑结构，采用迪杰斯特拉算法建立路由表

接下来，我们将针对每个环节，基于代码进行具体的分析。

5.1 链路监测

本地链接信息库存储有关到邻居的链接的信息。协议在 link_set.h 头文件中定义了很多和链路相关的函数，并且在 link_set.c 文件中给出了实现。其中包括定义、删除、修改、替代、重置等一系列相关函数，下面挑出一些函数进行代码解析。分别为更新、重置和删除函数。

5.1.1 链路信息表更新

每时每刻都在通过相关的链路变化来更新链路信息表。一旦满足如下规则就会进行更新。

1. 收到 HELLO 消息，且本地链路中不存在满足的表项。
2. 如果存在表项，执行更新操作，代码如下。

```
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.
7.    entry = add_link_entry(local, remote, &message->source_addr, message->vtime, messa
    ge->htime, in_if);
8.
9.     entry->vtime = message->vtime;
10.   entry->ASYM_time = GET_TIMESTAMP(message->vtime);
11.
12.   entry->prev_status = check_link_status(message, in_if);
13.
14.   switch (entry->prev_status) {
15.   case (LOST_LINK):
16.     olsr_stop_timer(entry->link_sym_timer);
17.     entry->link_sym_timer = NULL;
18.     break;
19.   case (SYM_LINK):
20.   case (ASYM_LINK):
21.
22.     olsr_set_timer(&entry->link_sym_timer, message->vtime, OLSR_LINK_SYM_JITTER, OLS
    R_TIMER_ONESHOT, &olsr_expire_link_sym_timer,  entry, 0);
23.
24.     olsr_set_link_timer(entry, message->vtime + NEIGHB_HOLD_TIME * MSEC_PER_SEC);
25.     break;
```

```
26.   default:;
27.   }
28.
29.   if (entry->link_timer && (entry->link_timer->timer_clock < entry->ASYM_time)) {
30.     olsr_set_link_timer(entry, TIME_DUE(entry->ASYM_time));
31.   }
32.
33.   if (olsr_cnf->use_hysteresis)
34.     olsr_process_hysteresis(entry);
35.
36.   update_neighbor_status(entry->neighbor, get_neighbor_status(remote));
37.
38.   return entry;
39.
40. }
```

第 7 行，如果链路中不存在表项时执行，增加一个节点

第 9 行，开始更新 ASYM_time

第 22 行，更改信息，使得 L_SYM_time = current time + validity time

第 24 行，更改 L_time，条件为 L_time = L_SYM_time + NEIGHB_HOLD_TIME

第 29 行，再次更改 L_time，使得其值为 L_time 和 L_ASYM_time 中较大一方

第 33 行，跟新滞后值

第 36 行，更新邻居表

5.1.2 重置链路信息函数

```
1.  void olsr_reset_all_links(void) {
2.    struct link_entry *link;
3.
4.    OLSR_FOR_ALL_LINK_ENTRIES(link) {
5.      link->ASYM_time = now_times-1;
6.
7.      olsr_stop_timer(link->link_sym_timer);
8.      link->link_sym_timer = NULL;
9.
10.     link->neighbor->is_mpr = false;
11.     link->neighbor->status = NOT_SYM;
12.   } OLSR_FOR_ALL_LINK_ENTRIES_END(link)
13.
14.
15.   OLSR_FOR_ALL_LINK_ENTRIES(link) {
16.     olsr_expire_link_sym_timer(link);
17.     olsr_clear_hello_lq(link);
18.   } OLSR_FOR_ALL_LINK_ENTRIES_END(link)
19. }
```

第 4 行~第 12 行，遍历整个表项，分别修改 link_ASYM_time、link_sym_timer、邻居表的 is_mpr、status 值。让它们分别清空或变为空指针

第 15 行~第 18 行，再次遍历， 通过 olsr_expire_link_sym_timer(link)回调链接符号计时器，通过 olsr_clear_hello_lq(link)函数处理 hello 消息的相关重置。

5.1.3 删除、取消链接并释放链接条目

当需要删除某些链接时，调用此函数

```
1.  static void
2.  olsr_delete_link_entry(struct link_entry *link)
3.  {
4.    struct tc_edge_entry *tc_edge;
5.
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.    }
10.
11.   if (link->neighbor->linkcount == 1) {
12.     olsr_delete_neighbor_table(&link->neighbor->neighbor_main_addr);
13.   } else {
14.     link->neighbor->linkcount--;
15.   }
16.
17.   olsr_stop_timer(link->link_timer);
18.   link->link_timer = NULL;
19.   olsr_stop_timer(link->link_sym_timer);
20.   link->link_sym_timer = NULL;
21.   olsr_stop_timer(link->link_hello_timer);
22.   link->link_hello_timer = NULL;
23.   olsr_stop_timer(link->link_loss_timer);
24.   link->link_loss_timer = NULL;
25.   list_remove(&link->link_list);
26.
27.   free(link->if_name);
28.   free(link);
29.
30.   changes_neighborhood = true;
31. }
```

第 6 行，删除为 SPF 制作的 tc_edge

第 12 行，删除邻居条目

第 18 行开始，结束掉正在运行的计时器，释放资源

5.1.4 删除与给定接口地址匹配的所有链接项

当需要删除某些特定接口的地址或者说特定 IP 的链接时，就会调用此函数

```
1.  void
2.  olsr_delete_link_entry_by_ip(const union olsr_ip_addr *int_addr)
```

```
3.  {
4.    struct link_entry *link;
5.
6.    if (list_is_empty(&link_entry_head)) {
7.      return;
8.    }
9.
10.   OLSR_FOR_ALL_LINK_ENTRIES(link) {
11.     if (ipequal(int_addr, &link->local_iface_addr)) {
12.       olsr_delete_link_entry(link);
13.     }
14.   }
15.   OLSR_FOR_ALL_LINK_ENTRIES_END(link);  }
```

第 6 行，先判读要删除的值是否为空，为空停止删除

第 11 行，进行比较，判断出是所要删除的给定端口，则调用

olsr_delete_link_entry(link)函数进行删除。

5.2 邻居发现

我们上文介绍，每个节点的邻居表中包含邻居的主地址及节点与这个邻居的链路状态。通过广播 HELLO 消息（包含该节点的邻居列表、邻居类型与链路类型），每个节点建立并维护邻居表，节点从而根据邻居表进行 MPR 计算。邻居发现的对应过程如下：

1. 每隔一段固定的时间，网络中的每个节点都向它的一跳邻居广播一次 HELLO 消息；
2. 通过 HELLO 消息的交互，各个节点更新自己的一跳二跳邻居表；
3. 每个节点得知一跳邻居与二跳邻居的信息，实现邻居发现。

我们对源代码涉及到的操作进行分析：

5.2.1 generate_hello 函数:产生 HELLO 消息

在 olsr_scheduler()设置定时器后，定义在 generate_msg.c 中的 generate_hello 函数定时广播 HELLO 消息。

```
1.  /*generate_msg.c*/
2.  void generate_hello(void *p)
3.  {
4.    struct hello_message hellopacket;
5.    struct interface *ifn = (struct interface *)p;
6.
7.    olsr_build_hello_packet(&hellopacket, ifn);
8.
9.    if (queue_hello(&hellopacket, ifn))
10.     net_output(ifn);
```

```
11.
12.   olsr_free_hello_packet(&hellopacket);
13. }
```

其中，构造 HELLO 消息包的 olsr_build_hello_packet 函数定义在 packet.c 中

```
1.  /*packet.c*/
2.  int
3.  olsr_build_hello_packet(struct hello_message *message, struct interface *outif)
4.  {
5.    struct hello_neighbor *message_neighbor, *tmp_neigh;
6.    struct link_entry *links;
7.    struct neighbor_entry *neighbor;
8.
9.  #ifdef DEBUG
10.   OLSR_PRINTF(3, "\tBuilding HELLO on interface \"%s\"\n", outif->int_name ? outif->
    int_name : "<null>");
11. #endif
12.
13.   message->neighbors = NULL;
14.   message->packet_seq_number = 0;
```

12-13 初始化邻居列表为空，并把序列号初始化为 0.

```
15.   /* Set willingness */
16.
17.   message->willingness = olsr_cnf->willingness;
18. #ifdef DEBUG
19.   OLSR_PRINTF(3, "Willingness: %d\n", olsr_cnf->willingness);
20. #endif
21.
22.   /* Set TTL */
23.
24.   message->ttl = 1;
25.   message->source_addr = olsr_cnf->main_addr;
```

17-25 初始化消息的意愿值与源地址，分别设置为 olsr_config.h 中定义的意愿值与主地址，TTL 设置为 1

```
26.   /* Walk all links of this interface */
27.   OLSR_FOR_ALL_LINK_ENTRIES(links) {
28.  if (!ipequal(&links->local_iface_addr, &outif->ip_addr)) {
29.       continue;
30.     }
31.
32.     message_neighbor = olsr_malloc_hello_neighbor("Build HELLO");
```

这里使用了宏定义来增加代码的复用性：OLSR_FOR_ALL_LINK_ENTRIES(links)与 OLSR_FOR_ALL_LINK_ENTRIES_END(links)是一个 for 循环的头部与尾部，并在头部遍历了所有的链接，使用宏定义时可以对其做不同操作。生成 HELLO 消息时，需要对

每个链接做检查，看他们是否已经在传出接口上注册。之后，计算该接口的邻居状态，确定 HELLO 消息里的 Link Code 字段。同理，根据邻居表中表项，HELLO 消息中的其他字段被逐个填充，HELLO 消息生成。
值得一提的是，HELLO 消息中的邻居类型除了是否为 MPR 外，还取决于这个邻居是否关于此节点对称链接。生成 HELLO 消息时，这也根据邻居表中的表项直接决定。而邻居表计算链路对称与否的过程将体现在邻居表操作相关的代码中。

5.2.2 process_message_neighbors 函数:处理 HELLO 消息

process_message_neighbors 函数定义在 process_package.c 中，处理来自 HELLO 消息的邻居地址列表。如前所述，首先更新链接集；之后更新邻居集来实现邻居发现。
首先，为了防止把节点本身也加入二跳邻居，对所有接口进行检查。

```
1.  if (if_ifwithaddr(&message_neighbors->address) != NULL)
2.    continue;
```

之后，从消息中得到邻居的主地址。

```
1.      /* Get the main address */
2.      neigh_addr = mid_lookup_main_addr(&message_neighbors->address);
3.
4.      if (neigh_addr != NULL) {
5.        message_neighbors->address = *neigh_addr;
6.      }
7.
8.      if (((message_neighbors->status == SYM_NEIGH) || (message_neighbors->status == M
    PR_NEIGH))) {
9.        struct neighbor_2_list_entry *two_hop_neighbor_yet = olsr_lookup_my_neighbors(
    neighbor, &message_neighbors->address);
```

调用 mid_lookup_main_addr 函数，对消息中的主地址在 mid 集中查找，如果 mid 集中存在这个主地址，则修改消息中地址指向函数内的 neigh_addr。之后，查看邻居状态字段，并调用 olsr_lookup_my_neighbors 函数，如果确认这个地址和 N 的对称邻居具有对称链接，说明它是二跳邻居，与之建立连接，并为这个邻居设置定时器。之后遍历一跳邻居，看通过哪些一跳邻居可以到达这个二跳邻居。

```
1.      if (((message_neighbors->status == SYM_NEIGH) || (message_neighbors->status == M
    PR_NEIGH))) {
2.        struct neighbor_2_list_entry *two_hop_neighbor_yet = olsr_lookup_my_neighbors(
    neighbor, &message_neighbors->address);
3.
4.        if (two_hop_neighbor_yet != NULL) {
5.          /* Updating the holding time for this neighbor */
```

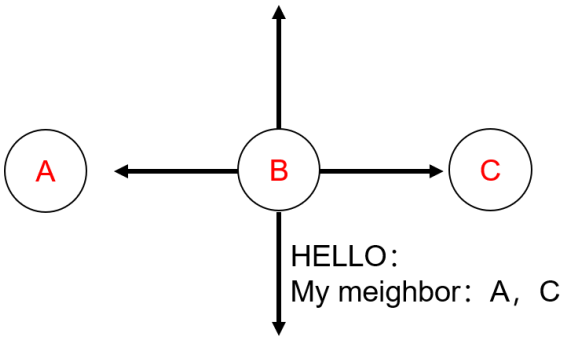
```
6.          olsr_set_timer(&two_hop_neighbor_yet->nbr2_list_timer, message->vtime, OLSR_
    NBR2_LIST_JITTER, OLSR_TIMER_ONESHOT,
7.                         &olsr_expire_nbr2_list, two_hop_neighbor_yet, 0);
8.          two_hop_neighbor = two_hop_neighbor_yet->neighbor_2;
```

（源文件 178-235）另外，对于链路质量的 OLSR，接下来需要对这个确定的路径重设链路质量。对消息中提到的每一个二跳邻居，计算出最佳的二跳路径开销。

5.2.3 update_neighbor_status 函数:邻居状态更新

在 neighbor_table.c 中定义了许多对邻居表增删改操作的函数，被不同的入口函数调用。update_neighbor_status 函数就是其一，通过调用 olsr_lookup_two_hop_neighbor_table 与 olsr_delete_two_hop_neighbor_table 更新二跳邻居表，计算链路是否对称并设置其 status。

5.2.4 发现二跳邻居的一个例子

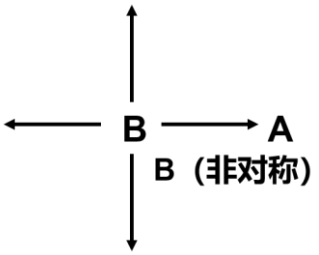


B 节点发送 HELLO 消息给 A，C 两个节点，HELLO 消息里声明自己的邻居为 A 和 C，这样，A 收到消息以后就直到了自己的二跳邻居为 C，C 也知道了自己的二跳邻居为 A。

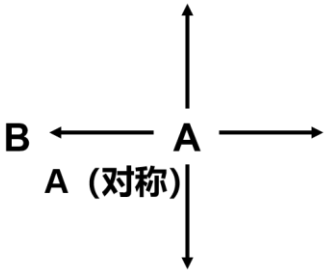
5.2.5 发现对称邻居的一个例子

A 和 B 是网络上相邻的节点，接下来将通过一个例子说明如何发现对称邻居。

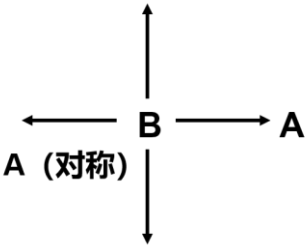
- B 向邻居发送自己的 HELLO 消息，A 收到 B 的 HELLO 消息，知道 B 是自己的一跳邻居，但是不知道 B 是不是一个对称邻居，也就是说不知道自己发的 HELLO 消息 B 能不能收到



- A 再想邻居发送自己的 HELLO 消息，B 收到了 A 的消息，将 A 标记为自己的邻居，此时，B 知道链路是双向可达的了，自己可以发给 A，也能收到 A 的消息，就把 A 标记为自己的对称邻居，并且在下一个 HELLO 消息中将 LINKCODE 里 A 的 NEIGHBORTYPE 标记为对称：



- B 再次发送 HELLO 消息，A 收到以后发现自己的 LINKCODE 里的 NEIGHBORTYPE 标记为了对称，所以知道自己刚刚发的消息被 B 收到了，于是知道自己即可以发给 B 消息，也能收到 B 的消息，于是将 B 的类型标记为对称。



5.3 MPR 选择

建立邻居表以及二跳邻居表以后，OLSR 协议会使每个节点选择自己的部分一跳节点作为自己的 MPR 节点，只有这些 MPR 节点可以转发该节点的消息，一个节点的所有 MPR 节点构成 MPR 集，对于 MPR 集的要求为：

- 该节点能通过自己选择的 MPR 集中的节点，访问到任何一个自己的二跳节点
- MPR 集越小，整个协议的开销越小
- 节点的 MPR 集为每个接口的 MPR 集的并集

5.3.1 MPR 选择算法

针对上述要求，OLSR 协议中的 MPR 选择算法如下：

1. 将一跳邻居中的所有 willingness 值为 WILL_ALWAYS 的节点加入 MPR 集
2. 将一跳邻居集中满足如下条件的节点加入 MPR 集：
   - 节点 P 有一个二跳节，能且仅能由此一跳节点访问到
3. 如果当前二跳节点已经完全可以由 MPR 集访问到，那么就终止算法，若不能则：
   - 计算当前未被选为 MPR 节点的一跳邻居对于二跳邻居的可达性，即，该一跳邻居能访问到的还未被 MPR 集覆盖的二跳邻居的数目
   - 选择可达性非零而且 willingness 最高的一跳邻居作为 MPR，若有多种选择，则
   - 选择可达性最高的作为 MPR，若可达性相同，则
   - 选择深度最大的节点作为 MPR
4. 删除 MPR 集里冗余的部分，即，如果在 MPR 集里删除一个节点，剩余的节点仍可以将所有二跳邻居访问到，那么就删除它
5. 对所有接口选择的 MPR 集取并集，作为节点的 MPR 集

OLSR 在 MPR.C 文件的 olsr_calculate_mpr 函数中描述了该 MPR 选择算法，具体代码分析如下：

```
1.  void  olsr_calculate_mpr(void)  {
2.    uint16_t two_hop_covered_count;
3.    uint16_t two_hop_count;
4.    int i;
5.    OLSR_PRINTF(3, "\n**RECALCULATING MPR**\n\n");
6.    olsr_clear_mprs();
7.    two_hop_count = olsr_calculate_two_hop_neighbors();
8.    two_hop_covered_count = add_will_always_nodes();
```

2 定义变量 two_hop_covered_count 记录已被 MPR 集覆盖的二跳邻居数量

3 定义变量 two_hop_count 记录所有二跳邻居数量

6 调用 olsr_clear_mprs 函数，清除现有的 MPR 集，以便重新生成

7 调用 olsr_calculate_two_hop_neighbors 函数，计算二跳邻居数量，结果返回到 two_hop_covered_count 变量中

8 调用 add_will_always_nodes 函数，将所有 willingness 为 WILL_ALWAYS 的节点加入 MPR 集，将返回值赋给变量 two_hop_covered_count

```
1.    for (i = WILL_ALWAYS - 1; i > WILL_NEVER; i--) {
2.      struct neighbor_entry *mprs;
```

```
3.      struct neighbor_2_list_entry *two_hop_list = olsr_find_2_hop_neighbors_with_1_li
    nk(i);
4.      while (two_hop_list != NULL) {
5.        struct neighbor_2_list_entry *tmp;
6.        if (!two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor->is_mpr)
7.          olsr_chosen_mpr(two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor,
    &two_hop_covered_count);
8.          tmp = two_hop_list;
9.        two_hop_list = two_hop_list->next;
10.       free(tmp);
11.     }
12.     if (two_hop_covered_count >= two_hop_count) {
13.       i = WILL_NEVER;
14.       break;
15.     }
16.     while ((mprs = olsr_find_maximum_covered(i)) != NULL) {
17.       olsr_chosen_mpr(mprs, &two_hop_covered_count);
18.
19.       if (two_hop_covered_count >= two_hop_count) {
20.         i = WILL_NEVER;
21.         break;  }}}
```

1 对所有 willingness 值进行遍历，从高到低

2 建立邻居表格式的 mpr 集

3 调用 olsr_find_2_hop_neighbors_with_1_link 函数，将在当前 willingness 下，找到存在直连链路的二跳邻居，并生成一个二跳邻居集返回给 two_hop_list

4-11 将 two_hop_list 中的二跳邻居依次进行处理，将他们未被选为 MPR 的对应的一跳邻居利用 olsr_chosen_mpr 函数选为 MPR

12-15 如果 two_hop_covered_count >= two_hop_count，也就是说，所有的二跳邻居都已经被 MPR 集覆盖到了，那么就停止循环

16-18 通过 olsr_find_maximum_covered 函数找出可达性最高的节点并将他选为 MPR

19-21 如果 two_hop_covered_count >= two_hop_count，也就是说，所有的二跳邻居都已经被 MPR 集覆盖到了，那么就停止循环

```
1.    olsr_optimize_mpr_set();
2.
3.    if (olsr_check_mpr_changes()) {
4.      OLSR_PRINTF(3, "CHANGES IN MPR SET\n");
5.      if (olsr_cnf->tc_redundancy > 0)
6.        signal_link_changes(true); }}
```

1 调用 olsr_optimize_mpr_set 函数对 MPR 集进行优化

3-6 调用 olsr_check_mpr_changes 函数检查是否有 MPR 变化，如果有，就用 signal_link_changes 函数发出状态改变的信号

接下来，我将详细介绍几个在 MPR 选择算法中调用过的函数。

5.3.1.1 olsr_clear_mprs 函数

```
1.  static void
2.  olsr_clear_mprs(void)
3.  {
4.    struct neighbor_entry *a_neighbor;
5.    struct neighbor_2_list_entry *two_hop_list;
6.
7.    OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
8.      if (a_neighbor->is_mpr) {
9.        a_neighbor->was_mpr = true;
10.       a_neighbor->is_mpr = false;
11.     }
12.       for (two_hop_list = a_neighbor->neighbor_2_list.next; two_hop_list != &a_neigh
    bor->neighbor_2_list;
13.          two_hop_list = two_hop_list->next) {
14.       two_hop_list->neighbor_2->mpr_covered_count = 0;
15.     }
16.
17.   }
18.   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
19. }
```

该函数用来重置该节点的 MPR 选择记录

8-11 将邻居集中所有节点取消 MPR 标签

12-15 将二跳邻居的已被 MPR 覆盖的标签置零

5.3.1.2 olsr_calculate_two_hop_neighbors 函数

```
1.  static uint16_t olsr_calculate_two_hop_neighbors(void)
2.  {
3.    struct neighbor_entry *a_neighbor, *dup_neighbor;
4.    struct neighbor_2_list_entry *twohop_neighbors;
5.    uint16_t count = 0;
6.    uint16_t n_count = 0;
7.    uint16_t sum = 0;
8.    /* Clear 2 hop neighs */
9.    olsr_clear_two_hop_processed();
10.   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
11.
12.     if (a_neighbor->status == NOT_SYM) {
13.       a_neighbor->neighbor_2_nocov = count;
14.       continue;
15.     }
16.     for (twohop_neighbors = a_neighbor->neighbor_2_list.next; twohop_neighbors != &a
    _neighbor->neighbor_2_list;
17.          twohop_neighbors = twohop_neighbors->next) {
18.
19.       dup_neighbor = olsr_lookup_neighbor_table(&twohop_neighbors->neighbor_2->neigh
    bor_2_addr);
20.
```

```
21.       if ((dup_neighbor == NULL) || (dup_neighbor->status != SYM)) {
22.         n_count++;
23.         if (!twohop_neighbors->neighbor_2->processed) {
24.           count++;
25.           twohop_neighbors->neighbor_2->processed = 1;
26.         }
27.       }
28.     }
29.     a_neighbor->neighbor_2_nocov = n_count;
30.     /* Add the two hop count */
31.     sum += count;
32.   }
33.   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
34.   OLSR_PRINTF(3, "Two hop neighbors: %d\n", sum);
35.   return sum;
36. }
```

这个函数用来计算所有的二跳邻居数量，其中的 OLSR_FOR_ALL_NBR_ENTRIES 是一个宏定义，它通过一个 for 循环把之后代码的操作应用于所有邻居节点上，OLSR_FOR_ALL_NBR_ENTRIES_END 是结束这个操作的标志。

9 调用 olsr_clear_two_hop_processed 函数将所有的二跳邻居的 processed 项清零

10-33 计算每个邻居的二跳节点的数量，并加和

5.3.1.3 add_will_always_nodes 函数

```
1.  static uint16_t
2.  add_will_always_nodes(void)
3.  {
4.    struct neighbor_entry *a_neighbor;
5.    uint16_t count = 0;
6.
7.    OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
8.      struct ipaddr_str buf;
9.      if ((a_neighbor->status == NOT_SYM) || (a_neighbor->willingness != WILL_ALWAYS))
    {
10.       continue;
11.     }
12.     olsr_chosen_mpr(a_neighbor, &count);
13.
14.     OLSR_PRINTF(3, "Adding WILL_ALWAYS: %s\n", olsr_ip_to_string(&buf, &a_neighbor->
    neighbor_main_addr));
15.
16.   }
17.   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
18.
19.   return count;
20. }
```

该函数将所有 willingness 值为 WILL_ALWAYS 的一跳邻居节点加入 MPR 集中，并返回加入的节点的数量

7 利用 OLSR_FOR_ALL_NBR_ENTRIES 对所有邻居节点进行操作

9-11 如果邻居非对称或者邻居的转发意愿值不是最高，就跳过该邻居

12 否则，就调用 olsr_chosen_mpr 函数将邻居选为 MPR

5.3.1.4 olsr_find_2_hop_neighbors_with_1_link 函数

```
1.  static struct neighbor_2_list_entry *
2.  olsr_find_2_hop_neighbors_with_1_link(int willingness)
3.  {
4.
5.    uint8_t idx;
6.    struct neighbor_2_list_entry *two_hop_list_tmp = NULL;
7.    struct neighbor_2_list_entry *two_hop_list = NULL;
8.    struct neighbor_entry *dup_neighbor;
9.    struct neighbor_2_entry *two_hop_neighbor = NULL;
10.
11.   for (idx = 0; idx < HASHSIZE; idx++) {
12.
13.     for (two_hop_neighbor = two_hop_neighbortable[idx].next; two_hop_neighbor != &tw
    o_hop_neighbortable[idx];
14.          two_hop_neighbor = two_hop_neighbor->next) {
15.
16.       dup_neighbor = olsr_lookup_neighbor_table(&two_hop_neighbor->neighbor_2_addr);

17.
18.       if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
19.         continue;
20.       }
21.       if (two_hop_neighbor->neighbor_2_pointer == 1) {
22.         if ((two_hop_neighbor->neighbor_2_nblist.next->neighbor->willingness == will
    ingness)
23.             && (two_hop_neighbor->neighbor_2_nblist.next->neighbor->status == SYM))
    {
24.           two_hop_list_tmp = olsr_malloc(sizeof(struct neighbor_2_list_entry), "MPR
    two hop list");
25.
26.           /* Only queue one way here */
27.           two_hop_list_tmp->neighbor_2 = two_hop_neighbor;
28.
29.           two_hop_list_tmp->next = two_hop_list;
30.
31.           two_hop_list = two_hop_list_tmp;}}}}
32.   return (two_hop_list_tmp);
33. }
```

该函数用于选出符合当前意愿值条件的同时是当前节点一跳邻居的二跳邻居，参数是意愿值，返回的是一个二跳邻居链表的指针。

11-20 遍历两跳邻居表

21-31 寻找符合条件的二跳邻居节点并把它加入链表中

32 返回链表指针

5.3.1.5 olsr_chosen_mpr 函数

```
1.  static int
2.  olsr_chosen_mpr(struct neighbor_entry *one_hop_neighbor, uint16_t * two_hop_covered_
    count)
3.  {
4.    struct neighbor_list_entry *the_one_hop_list;
5.    struct neighbor_2_list_entry *second_hop_entries;
6.    struct neighbor_entry *dup_neighbor;
7.    uint16_t count;
8.    struct ipaddr_str buf;
9.    count = *two_hop_covered_count;
10.
11.   OLSR_PRINTF(1, "Setting %s as MPR\n", olsr_ip_to_string(&buf, &one_hop_neighbor->n
    eighbor_main_addr));
12.   one_hop_neighbor->is_mpr = true;      //NBS_MPR;
13.
14.   for (second_hop_entries = one_hop_neighbor->neighbor_2_list.next; second_hop_entri
    es != &one_hop_neighbor->neighbor_2_list;
15.        second_hop_entries = second_hop_entries->next) {
16.     dup_neighbor = olsr_lookup_neighbor_table(&second_hop_entries->neighbor_2->neigh
    bor_2_addr);
17.
18.     if ((dup_neighbor != NULL) && (dup_neighbor->status == SYM)) {
19.       continue;
20.     }
21.     /*
22.        Now the neighbor is covered by this mpr
23.      */
24.     second_hop_entries->neighbor_2->mpr_covered_count++;
25.     the_one_hop_list = second_hop_entries->neighbor_2->neighbor_2_nblist.next;
26.     if (second_hop_entries->neighbor_2->mpr_covered_count >= olsr_cnf->mpr_coverage)
27.       count++;
28.     while (the_one_hop_list != &second_hop_entries->neighbor_2->neighbor_2_nblist) {
29.       if ((the_one_hop_list->neighbor->status == SYM)) {
30.         if (second_hop_entries->neighbor_2->mpr_covered_count >= olsr_cnf->mpr_cover
    age) {
31.           the_one_hop_list->neighbor->neighbor_2_nocov--;
32.         }
33.       }
34.       the_one_hop_list = the_one_hop_list->next;
35.     }
36.   }
37.
38.   *two_hop_covered_count = count;
39.   return count;
40. }
```

此函数用于将一个节点选为 MPR 节点第一个参数是一个一跳邻居的结构体，第二个参数是当前被覆盖的二跳节点的数量，是一个值-结果变量，调用函数时，将一个变量的引用传到第二个参数，函数体会对它做一定的修改。

12 将第一个变量对应的一跳邻居节点的类型标志为 MPR

14-36 计算该 MPR 节点覆盖的二跳节点的数量，改数量变量为 count

38 将 count 值传回给第二个参数（一个变量的引用）

39 返回 count 值

5.3.1.6 `olsr_find_maximum_covered` 函数

```
1.  static struct neighbor_entry *
2.  olsr_find_maximum_covered(int willingness)
3.  {
4.    uint16_t maximum;
5.    struct neighbor_entry *a_neighbor;
6.    struct neighbor_entry *mpr_candidate = NULL;
7.
8.    maximum = 0;
9.
10.   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
11. #if 0
12.     printf("[%s] nocov: %d mpr: %d will: %d max: %d\n\n", olsr_ip_to_string(&buf, &a
    _neighbor->neighbor_main_addr),
13.            a_neighbor->neighbor_2_nocov, a_neighbor->is_mpr, a_neighbor->willingness
    , maximum);
14. #endif
15.     if ((!a_neighbor->is_mpr) && (a_neighbor->willingness == willingness) && (maximu
    m < a_neighbor->neighbor_2_nocov)) {
16.
17.       maximum = a_neighbor->neighbor_2_nocov;
18.       mpr_candidate = a_neighbor;
19.     }
20.   }
21.   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
22.
23.   return mpr_candidate;
24. }
```

该函数选出在特定 willingness 值下可覆盖二跳邻居最多的一跳邻居作为 MPR 候选者，参数为 willingness 值，返回值是指向该一跳邻居的指针。

10 对于所有一跳邻居进行操作

15 判断有没有一个一跳邻居它现在还不是 MPR 节点，他的 willingness 值为参数给出的 willingness 值，并且他的 neighbor_2_nocov 值（覆盖的二跳邻居数量）大于当前记录的最大值 maximum

17 将 maximum 值改为当前一跳邻居覆盖的二跳邻居的数量

18 将该节点选为 MPR 候选节点

23 返回这个一跳邻居节点的指针

5.3.1.7 olsr_optimize_mpr_set 函数

```
1.  static void
2.  olsr_optimize_mpr_set(void)
3.  {
4.    struct neighbor_entry *a_neighbor, *dup_neighbor;
5.    struct neighbor_2_list_entry *two_hop_list;
6.    int i, removeit;
7.    for (i = WILL_NEVER + 1; i < WILL_ALWAYS; i++) {
8.
9.      OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
10.
11.       if (a_neighbor->willingness != i) {
12.         continue;
13.       }
14.
15.       if (a_neighbor->is_mpr) {
16.         removeit = 1;
17.
18.         for (two_hop_list = a_neighbor->neighbor_2_list.next; two_hop_list != &a_nei
    ghbor->neighbor_2_list;
19.              two_hop_list = two_hop_list->next) {
20.
21.           dup_neighbor = olsr_lookup_neighbor_table(&two_hop_list->neighbor_2->neigh
    bor_2_addr);
22.
23.           if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
24.             continue;
25.           }
26.           if (two_hop_list->neighbor_2->mpr_covered_count <= olsr_cnf->mpr_coverage)
     {
27.             removeit = 0;  }}}
28.         if (removeit) {
29.           struct ipaddr_str buf;
30.           OLSR_PRINTF(3, "MPR OPTIMIZE: removiong mpr %s\n\n", olsr_ip_to_string(&bu
    f, &a_neighbor->neighbor_main_addr));
31.           a_neighbor->is_mpr = false;
32.         }
33.       }
34.     } OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
35.   }
36. }
```

该函数可以优化 MPR 集合，通过删除一部分被选为 MPR 的节点，这些节点有如下特征，该节点覆盖的二跳邻居节点都可以被其余 MPR 节点访问到。

7 从 willingness 值最低的 MPR 节点里先开始进行挑选删除

9 对于所有一跳邻居进行操作

11-13 若当前的一跳邻居的 willness 值不符合 for 循环里的 willingness 值，就跳过
15-16 若节点为 MPR 节点，先将 removeit 置为 1，假设我们要删除它
18-27 若当前邻居可以访问到一个大家都没法访问到的二跳邻居，就把 removeit 置为 0
28-31 若 removeit 值未被置为 0，就将该一跳邻居节点的邻居类型设为不是 MPR

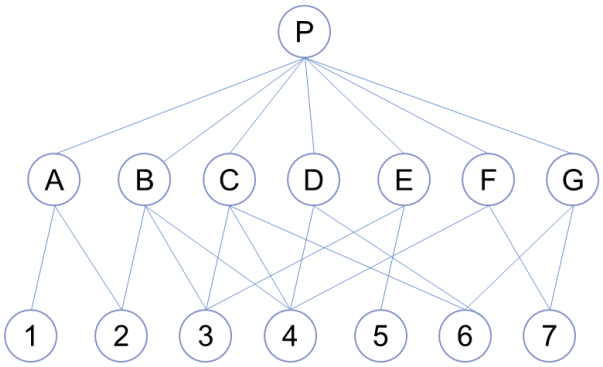
5.3.1.8 `olsr_check_mpr_changes` 函数

```
1.  static int
2.  olsr_check_mpr_changes(void)
3.  {
4.    struct neighbor_entry *a_neighbor;
5.    int retval;
6.
7.    retval = 0;
8.
9.    OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
10.
11.     if (a_neighbor->was_mpr) {
12.       a_neighbor->was_mpr = false;
13.
14.       if (!a_neighbor->is_mpr) {
15.         retval = 1;
16.       }
17.     }
18.   }
19.   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
20.
21.   return retval;
22. }
```

该函数用来检测 MPR 集有没有变化，若有变化则返回 1，若无变化返回 0
7 将 retval 置为 0，默认没有变化
9 对所有一跳邻居进行操作
11-14 判断该节点是不是曾经是 MPR 但是现在不是了
15 如果有，将 retval 置为 1
21 返回 retval

5.3.2 MPR 选择的一个例子

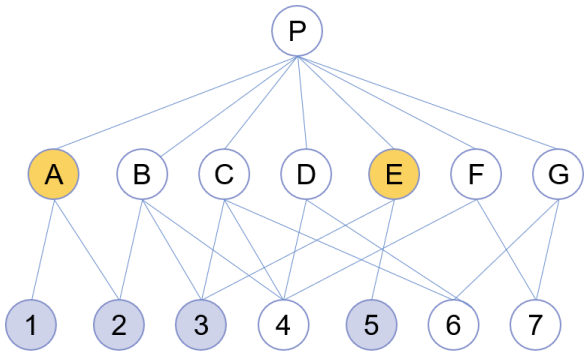
针对刚才的 MPR 选择算法，我们制作了一个例子，来形象的阐述 MPR 选择算法的运作流程，网络拓扑图如下，我们要选出节点 P 的 MPR 集第一行为一跳邻居节点，第二行为二跳邻居节点。



- 首先，计算 P 的所有一跳邻居 Y 的度，即为可以通过它访问到的所有二跳节点的数量，记为 D（Y）

D (A)	D (B)	D (C)	D (D)	D (E)	D (F)	D (G)
2	3	3	2	2	2	2

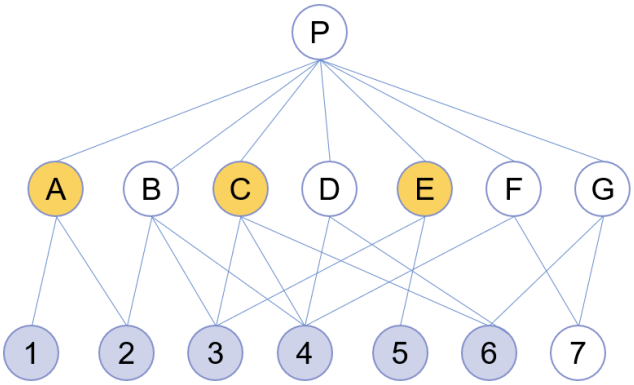
- 我们假设所有一跳邻居的 willingness 的值都小于 WILL_ALWAYS
- 如果有一个二跳节点，他只能通过一个一跳邻居访问到，那么久选择该一跳邻居为 MPR，重复该步骤，此时网络拓扑图如下，深色的一跳邻居节点是已被选为 MPR 的节点，深色的二跳邻居节点是被 MPR 访问覆盖到的二跳邻居节点。



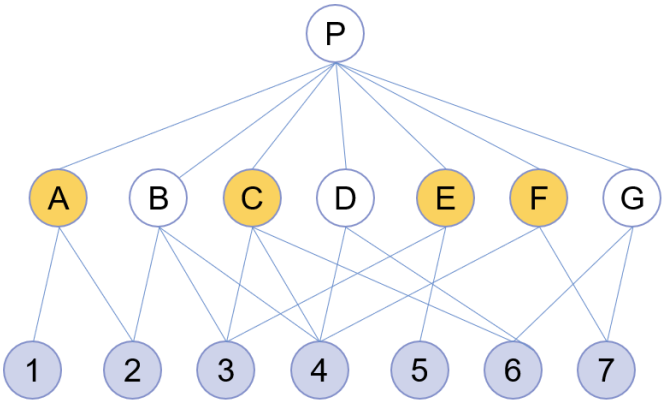
- 计算当前未被选为 MPR 节点的一跳邻居对于二跳邻居的可达性，即，该一跳邻居能访问到的还未被 MPR 集覆盖的二跳邻居的数目记为 R（Y）

R (A)	R (B)	R (C)	R (D)	R (E)	R (F)	R (G)
0	1	2	2	0	2	2

- 选择可达性非零而且 willingness 最高的一跳邻居作为 MPR，若有多种选择，则选择可达性最高的作为 MPR，若可达性相同，则选择深度最大的节点作为 MPR，下面为下一次更新的网络拓扑图，此时选择了 C 为新的 MPR，因为 C 和 D 的可达性一样，但是 C 的深度大：



- 此时，还有二跳节点 7 未被覆盖到，我们假设 F 的 willingness 值为 WILL_ALWAYS-1，G 为 WILL_ALWAYS-2，所以选择 F 为 MPR，最终的网络拓扑图如下：



5.3.3 考虑链路质量的 MPR 选择

在 lq_mpr.c 文件中的 olsr_calculate_lq_mpr 函数描述了 OLSRd 使用的考虑链路质量的 MPR 选择算法，这个算法的目的之一仍是选择 willingness 最高的一跳邻居作为 MPR，其次选择拥有可以到达二跳邻居的最好的链路状态的一跳邻居作为 MPR，具体代码分析如下：

```
1.  void olsr_calculate_lq_mpr(void)
2.  {
3.    struct neighbor_2_entry *neigh2;
4.    struct neighbor_list_entry *walker;
5.    int i, k;
6.    struct neighbor_entry *neigh;
7.    olsr_linkcost best, best_1hop;
8.    bool mpr_changes = false;
9.
10.   OLSR_FOR_ALL_NBR_ENTRIES(neigh) {
11.     neigh->was_mpr = neigh->is_mpr;
12.     neigh->is_mpr = false;
```

10 对所有一跳邻居进行如下操作
11 将 MPR 现在的状态保存为过去的状态，把现在的状态空出来一会进行更新
12 将现在的状态设为该节点不是 MPR

```
1.      if (neigh->status == NOT_SYM || neigh->willingness != WILL_ALWAYS) {
2.        continue;
3.      }
4.
5.      neigh->is_mpr = true;
6.
7.      if (neigh->is_mpr != neigh->was_mpr) {
8.        mpr_changes = true;
9.      }
10.   }
11.   OLSR_FOR_ALL_NBR_ENTRIES_END(neigh);
```

1-3 如果该邻居不是对称邻居或者该邻居的 willingness 不是 ALWAYS，就跳过
5 否则，将该节点选为 MPR 节点
7-9 如果该邻居节点的过去状态和现在状态不一致，说明 MPR 产生变化

```
1.  for (i = 0; i < HASHSIZE; i++) {
2.      for (neigh2 = two_hop_neighbortable[i].next; neigh2 != &two_hop_neighbortable[i]
    ; neigh2 = neigh2->next) {
3.        best_1hop = LINK_COST_BROKEN;
4.        neigh = olsr_lookup_neighbor_table(&neigh2->neighbor_2_addr);
5.        if (neigh != NULL && neigh->status == SYM) {
6.          struct link_entry *lnk = get_best_link_to_neighbor(&neigh->neighbor_main_add
    r);
7.          if (!lnk)
8.            continue;
9.
10.         best_1hop = lnk->linkcost;
```

1 遍历所有的二跳邻居节点
3 将 best_1hop 的链路开销（当前最好的链路开销）先设置为链路断开的开销，LINK_COST_BROKEN 的定义为(1<<22)也就是开销无穷大
4 利用 olsr_lookup_neighbor_table 函数进行遍历，查看该二跳邻居是否也为一跳邻居
5 如果 neigh 不为空（意味着的确也为一跳邻居），而且是对称邻居
6 利用 get_best_link_to_neighbor 函数返回到这个邻居最好的直连链路到 lnk 变量
7-8 如果没有更好的直连链路，那就跳过
10 现在将最好的连接状态设为 lnk 链路的链路开销

```
1.          for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_
    nblist; walker = walker->next)
2.            if (walker->path_linkcost < best_1hop)
3.              break;
```

```
4.          if (walker == &neigh2->neighbor_2_nblist)
5.            continue;
6.        }
7.
```

1 遍历一跳节点检查我们是不是找到了合适的 MPR

2-3 如果有更好的链路开销，那么就终止循环

4 如果搜索到链表尾部仍没有更好的链路开销，那么就继续外层的大循环

```
1.        for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nb
    list; walker = walker->next)
2.          walker->neighbor->skip = false;
3.
4.        for (k = 0; k < olsr_cnf->mpr_coverage; k++) {
5.          neigh = NULL;
6.          best = LINK_COST_BROKEN;
7.
8.          for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_
    nblist; walker = walker->next)
9.            if (walker->neighbor->status == SYM && !walker->neighbor->skip && walker->
    path_linkcost < best) {
10.             neigh = walker->neighbor;
11.             best = walker->path_linkcost;
```

1-2 将所有一跳邻居的状态标记为未选择过

5-6 预置操作，同上

8-12 不断循环，直到找到拥有最好链路开销的邻居节点

```
1.          if ((neigh != NULL) && (best < best_1hop)) {
2.            neigh->is_mpr = true;
3.            neigh->skip = true;
4.
5.            if (neigh->is_mpr != neigh->was_mpr)
6.              mpr_changes = true;
7.          }
8.          else
9.            break;
10.       }
11.     }
12.   }
13.   if (mpr_changes && olsr_cnf->tc_redundancy > 0)
14.     signal_link_changes(true);
```

1-12 如果这个节点到刚刚提到的二跳邻居节点的链路比直连链路都要好的话，就把这个一跳邻居节点选为 MPR

13-14 发送链路改变信号

5.3.4 更新 MPR Selector 集

假设节点 X 选择 Y 作为自己的 MPR 后，会将自己的邻居表中 Y 的邻居类型改为 MPR 邻居，并且再之后发送的 HELLO 消息中的 Y 的邻居类型中会对其声明，所以，当 Y 再次接到 X 的 HELLO 消息后，就会如此对消息进行处理，下面是 process_package.c 文件中的 olsr_hello_tap 函数，在这里仅截取和 Y 节点更新自己的 MPR Selector 集相关的部分代码。

```
1.  if (lookup_mpr_status(message, in_if))
2.     /* source_addr is always the main addr of a node! */
3.     olsr_update_mprs_set(&message->source_addr, message->vtime);
```

1 调用 lookup_mpr_status 函数检查这条 HELLO 消息的主人有没有将自己设置为 MPR
3 如果有，就调用 olsr_update_mprs_set 更新 MPR Selector 集，将消息的主人加入进去。

5.3.4.1 lookup_mpr_status 函数

其中 lookup_mpr_status 函数在 process_package.c 文件中定义，如下所示

```
1.  static bool
2.  lookup_mpr_status(const struct hello_message *message, const struct interface *in_if
    )
3.  {
4.    struct hello_neighbor *neighbors;
5.
6.    for (neighbors = message->neighbors; neighbors; neighbors = neighbors->next) {
7.      if ( neighbors->link != UNSPEC_LINK
8.          && (olsr_cnf->ip_version == AF_INET
9.              ? ip4equal(&neighbors->address.v4, &in_if->ip_addr.v4)
10.             : ip6equal(&neighbors->address.v6, &in_if->int6_addr.sin6_addr))) {
11.
12.       if (neighbors->link == SYM_LINK && neighbors->status == MPR_NEIGH) {
13.         return true;
14.       }
15.       break;
16.     }
17.   }
18.   /* Not found */
19.   return false;
20. }
```

该函数用来检查 HELLO 消息的来源是不是把自己设置为 MPR，两个参数分别是 Message 消息和一个用来存放链路状态的 buffer，如果来源把自己设置为 MPR 则返回 true，若没有则返回 false
6 遍历 HELLO 消息中提到的所有 neighbor

7-10 判断链路状态是否不明确，ip 版本是否正确
12-13 如果自己和来源是对称的，而且自己（来源的邻居）被设为 MPR，返回真
19 如果不是 返回假

5.3.4.2 olsr_update_mprs_set 函数

该函数在 mpr _selector_set.c 中定义，第一个参数是要被加入 Selector 集的节点的地址，第二个参数是消息的 v_time，返回 1 为成功，0 为失败，解释如下：

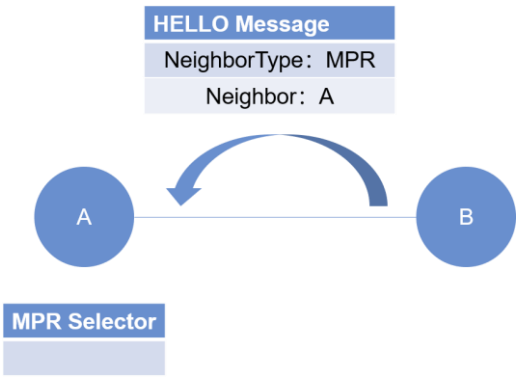
```
1.  int
2.  olsr_update_mprs_set(const union olsr_ip_addr *addr, olsr_reltime vtime)
3.  {
4.    struct ipaddr_str buf;
5.    struct mpr_selector *mprs = olsr_lookup_mprs_set(addr);
6.
7.    OLSR_PRINTF(5, "MPRS: Update %s\n", olsr_ip_to_string(&buf, addr));
8.
9.    if (mprs == NULL) {
10.     olsr_add_mpr_selector(addr, vtime);
11.     signal_link_changes(true);
12.     return 1;
13.   }
14.   olsr_set_mpr_sel_timer(mprs, vtime);
15.   return 0;
16. }
```

5 利用 olsr_lookup_mprs_set 函数搜索现有的 MPR Selector 集是不是已经存在第一个参数指向的节点
9 如果有，使用 olsr_add_mpr_selector 函数向集合里添加节点，声明链路改变
14 如果没有，更新时间

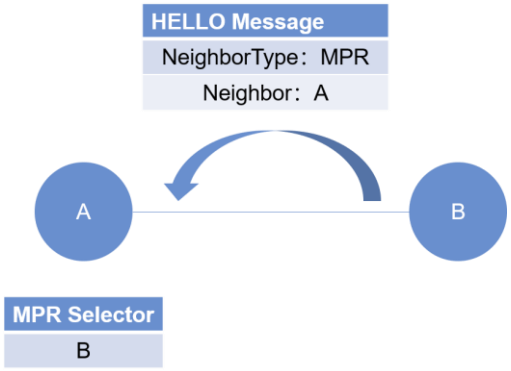
5.3.5 更新 MPR Selector 集的一个例子

针对更新 MPR Selector 集，我们准备了一个例子形象的描述这段代码是怎么运作的。例子如下：

- A 和 B 是网络中的连个相邻节点，B 把 A 选为了 MPR 节点，此时，在 B 的 HELLO 消息中，描述邻居 A 的状态变为 MPR 邻居：



- A 收到 HELLO 消息后，发现自己被 B 标记为 MPR，于是在 MPRSeletor 集中加入 B



5.4 拓扑建立

通过交换 TC 消息，每个节点更新拓扑表建立拓扑结构。拓扑建立涉及到对 TC 消息及拓扑表的操作如下：

5.4.1 generate_tc 函数:产生 TC 消息

```
1.  void generate_tc(void *p)
2.  {
3.    struct tc_message tcpacket;
4.    struct interface *ifn = (struct interface *)p;
5.
6.    olsr_build_tc_packet(&tcpacket);
7.
8.    if (queue_tc(&tcpacket, ifn) && TIMED_OUT(ifn->fwdtimer)) {
9.      set_buffer_timer(ifn);
10.   }
11.   olsr_free_tc_packet(&tcpacket);
12. }
```

与 HELLO 消息的生成类似，在 generate_msg.c 中定义的 generate_tc 函数调用 olsr_build_tc_packet 来构造 TC 包。不同的是， TC 消息的释放需要判断之前的 TC 消息

时间戳是否到期，若到了时间，则调用 set_buffer_timer()重新设置定时器，并从函数输入的参数所对应的接口释放消息。
因为同样的原因，在节点的通告链路集为空集期间，节点仍然持续发送空的 TC 消息，以保证让先前的 TC 消息无效。

5.4.2 olsr_input_tc:处理 TC 消息

定义在 tc_set.c 中。在做处理之前，先判断一下要不要丢弃这个消息包：

```
1.  /* We are only interested in TC message types. */
2.    pkt_get_u8(&curr, &type);
3.    if ((type != LQ_TC_MESSAGE) && (type != TC_MESSAGE)) {
4.      return false;
5.    }
```

是否是 TC 消息，不是直接丢弃

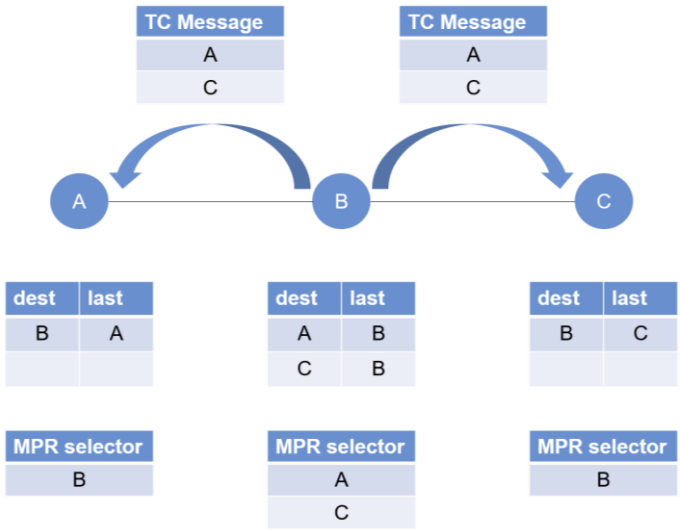
```
1.  if (check_neighbor_link(from_addr) != SYM_LINK) {
2.    OLSR_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr_ip_to_string(&buf, f
    rom_addr));
3.    return false;
4.  }
```

判断发送接口的信息，如果发送者不是对称一跳邻居，包也被丢弃
（源文件 842-848）接受到 TC 消息后，节点根据消息头的 Vtime 字段计算 TC 消息的“有效时间”；（源文件 853-872）并对序列号做判断，据此生成新的 tc 条目加入到拓扑表。

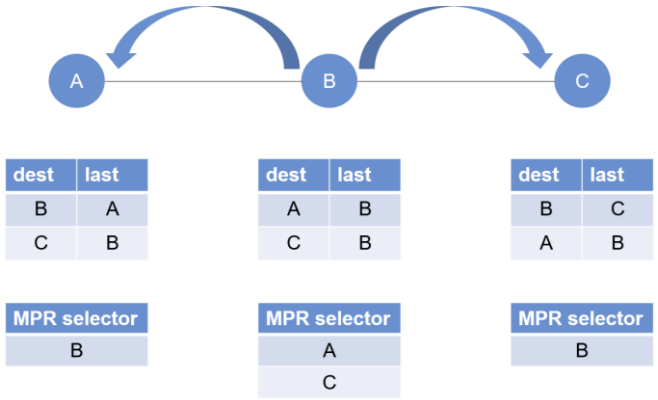
5.4.3 拓扑建立的一个小例子

针对拓扑消息的生成，发送和节点处理收到的拓扑控制消息的过程，我们制作了一个小例子，来形象的演示拓扑建立的过程：

- A，B，C 为网络中的三个节点，改图从上到下分别为 B 发送的 TC 消息，网络拓扑结构，TC 表，以及 MPR selector 表，当前状态为，A 和 C 选择 B 作为 MPR，B 选择 A 和 C 作为 MPR，此时，B 发送自己生成的 TC 消息，广播自己的 MPR selector 集到网络中，如图所示：



- A 收到消息后发现，C 把 B 选为了 MPR，这个消息是它之前不知道的，于是它就跟新了他的 TC 表，C 节点同理，更新完如下图：



5.5 路由计算

在所有节点交换完自己的 TC 消息以后，每个节点都在 TC 表里存储了足够还原网络拓扑结构的信息，现在每个节点就可以通过迪杰斯特拉算法来建立路由，以此来填充自己的路由表，说明该节点如何转发收到的数据包。下面，我先通过代码描述一些和路由计算相关的函数。

5.5.1 路由算法

在 OLSR 的 RFC 文档中提供了 OLSR 计算路由的路由算法，如下所示：

每个节点维护- -张保存到每个目的节点的路由表。路由表的计算是基于节点存储的拓扑结构的。接收到 TC 分组的节点,分析并存储[last - hop,node]连接对，其中"node" 是

在发送 TC 分组列表中发现的节点地址。简单来说，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 Djkstra 算法。当邻居表和拓扑表发生变化或路由失效时都需要更新路由表。

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=l;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

对于任意两跳邻居，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--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_ gaddr ==两跳邻居地址

R_ next _ gaddr ==已有的具有如下特征的路由表表项的 R_ next_ addr :其 R_ ,dest_ gaddr 为该两跳邻居表项的 N_ neighbor_ main_ addr

R_ dist==2

R _iface_ gaddr --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，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_ gaddr 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。

此算法在 olsr_spf.c 中通过构建 spf 树进行迪杰斯特拉算法来实现。

5. 5. 2 路由计算相关函数

在 route_table.c 文件中定义了许多路由计算的相关函数，包括初始化路由表，比较最短路径等，我将挑选一些重要的函数代码进行简要分析。

5.5.2.1 olsr_init_routing_table 函数

该函数用来初始化路由树，代码分析如下

```
1.  void
2.  olsr_init_routing_table(void)
3.  {
4.    OLSR_PRINTF(5, "RIB: init routing tree\n");
5.
6.    /* the routing tree */
7.    avl_init(&routingtree, avl_comp_prefix_default);
8.    routingtree_version = 0;
9.
10.   /*
11.    * Get some cookies for memory stats and memory recycling.
12.    */
13.   rt_mem_cookie = olsr_alloc_cookie("rt_entry", OLSR_COOKIE_TYPE_MEMORY);
14.   olsr_cookie_set_memory_size(rt_mem_cookie, sizeof(struct rt_entry));
15.
16.   rtp_mem_cookie = olsr_alloc_cookie("rt_path", OLSR_COOKIE_TYPE_MEMORY);
17.   olsr_cookie_set_memory_size(rtp_mem_cookie, sizeof(struct rt_path));
18. }
```

7-8 调用 `avl_init` 函数生成一个 avl 树，`routingtree_version` 用于检测过时的信息，在这里先置为零

13-17 为 rt_entry 和 rt_ path 分配内存，并创建相应的 cookie

5.5.2.2 olsr_lookup_routing_table 函数

该函数用于从路由表中查找特定的条目，参数是要查找的转发地址，返回值是该表项的指针，具体代码分析如下：

```
1.  struct rt_entry *
2.  olsr_lookup_routing_table(const union olsr_ip_addr *dst)
3.  {
4.    struct avl_node *rt_tree_node;
5.    struct olsr_ip_prefix prefix;
6.
7.    prefix.prefix = *dst;
8.    prefix.prefix_len = olsr_cnf->maxplen;
9.
10.   rt_tree_node = avl_find(&routingtree, &prefix);
11.
12.   return rt_tree_node ? rt_tree2rt(rt_tree_node) : NULL;
13. }
```

10 调用 avl_find 函数在 avl 树中查找该叶子

12 返回查找结果

5.5.2.3 olsr_lookup_routing_table 函数

该函数用于为提供的特定 IP 创造一条路由表项，并将它加入 avl 树中，参数为 IP，返回值为路由表项的指针

```
1.  static struct rt_entry *
2.  olsr_alloc_rt_entry(struct olsr_ip_prefix *prefix)
3.  {
4.    struct rt_entry *rt = olsr_cookie_malloc(rt_mem_cookie);
5.    if (!rt) {
6.      return NULL;
7.    }
8.
9.    memset(rt, 0, sizeof(*rt));
10.
11.   /* Mark this entry as fresh (see process_routes.c:512) */
12.   rt->rt_nexthop.iif_index = -1;
13.
14.   /* set key and backpointer prior to tree insertion */
15.   rt->rt_dst = *prefix;
16.
17.   rt->rt_tree_node.key = &rt->rt_dst;
18.   avl_insert(&routingtree, &rt->rt_tree_node, AVL_DUP_NO);
19.
20.   /* init the originator subtree */
21.   avl_init(&rt->rt_path_tree, avl_comp_default);
22.
23.   return rt;
24. }
```

4 为一个新的路由表项 rt 开辟空间

12 标识这个路由表项为新建立的路由表项

15 把该表项的 ip 地址设置为参数给的 ip 地址

17-21 将节点插入到路由表中，并初始化树

5.5.2.4 olsr_insert_rt_path 函数

该函数为一个给定的路由路径创建一个路由表项，并将它插入 RIB 树中

```
1.  void
2.  olsr_insert_rt_path(struct rt_path *rtp, struct tc_entry *tc, struct link_entry *lin
    k)
3.  {
4.    struct rt_entry *rt;
5.    struct avl_node *node;
6.    if (tc->path_cost == ROUTE_COST_BROKEN) {
7.      return;
8.    }
9.
10.   if (rtp->rtp_dst.prefix_len > olsr_cnf->maxplen) {
11.     return;
12.   }
13.   node = avl_find(&routingtree, &rtp->rtp_dst);
14.   if (!node) {
15.     rt = olsr_alloc_rt_entry(&rtp->rtp_dst);
16.       if (!rt) {
17.       return;
18.     }
19.   } else {
20.     rt = rt_tree2rt(node);
21.   }
22.   /* Now insert the rt_path to the owning rt_entry tree */
23.   rtp->rtp_originator = tc->addr;
24.
25.   /* set key and backpointer prior to tree insertion */
26.   rtp->rtp_tree_node.key = &rtp->rtp_originator;
27.     /* insert to the route entry originator tree */
28.   avl_insert(&rt->rt_path_tree, &rtp->rtp_tree_node, AVL_DUP_NO);
29.     /* backlink to the owning route entry */
30.   rtp->rtp_rt = rt;
31.     /* update the version field and relevant parameters */
32.   olsr_update_rt_path(rtp, tc, link);
33. }
```

4-12 判断路由路径的参数是不是符合要求，比如链路是不是断的，如果是，就返回

13-14 判断树中是不是已经有此节点了

15 如果没有，调用 olsr_alloc_rt_entry 函数建立新的路由表，参数传路径的目的地址，即为目标的 IP

23-32 将新的路由表项插入路由表中，并更新整个树

5.5.2.5 olsr_cmp_rtp 函数

比较两个路径的开销优劣，两个参数分别是两个路径的指针，第一个路径好返回 true，第二个路径好返回 false

```
1.  static bool
2.  olsr_cmp_rtp(const struct rt_path *rtp1, const struct rt_path *rtp2, const struct rt
    _path *inetgw)
3.  {
4.    olsr_linkcost etx1 = rtp1->rtp_metric.cost;
5.    olsr_linkcost etx2 = rtp2->rtp_metric.cost;
6.    if (inetgw == rtp1)
7.      etx1 *= olsr_cnf->lq_nat_thresh;
8.    if (inetgw == rtp2)
9.      etx2 *= olsr_cnf->lq_nat_thresh;
10.   if (etx1 < etx2) {
11.     return true;
12.   }
13.   if (etx1 > etx2) {
14.     return false;
15.   }
16.   if (rtp1->rtp_metric.hops < rtp2->rtp_metric.hops) {
17.     return true;
18.   }
19.   if (rtp1->rtp_metric.hops > rtp2->rtp_metric.hops) {
20.     return false;
21.   }
22.   if (memcmp(&rtp1->rtp_originator, &rtp2->rtp_originator, olsr_cnf->ipsize) < 0) {
23.       return true;
24.   }
25.
26.   return false;
27. }
```

10-15 先比较两个路由路径的花销，花销更小更优

16-21 开销一样的跳数越小越优

22-23 再一样的话源地址越小越优

5.5.2.6 olsr_rt_best 函数

如果给出的路由比最优路由好，就把最好路由替换成参数中的路由

```
1.  void
2.  olsr_rt_best(struct rt_entry *rt)
3.  {
4.    /* grab the first entry */
5.    struct avl_node *node = avl_walk_first(&rt->rt_path_tree);
6.    assert(node != 0);            /* should not happen */
7.    rt->rt_best = rtp_tree2rtp(node);
8.    /* walk all remaining originator entries */
```

```
9.    while ((node = avl_walk_next(node))) {
10.     struct rt_path *rtp = rtp_tree2rtp(node);
11.
12.     if (olsr_cmp_rtp(rtp, rt->rt_best, current_inetgw)) {
13.       rt->rt_best = rtp;
14.     }
15.   }
16.   if (0 == rt->rt_dst.prefix_len) {
17.     current_inetgw = rt->rt_best;
18.   }
19. }
```

9 遍历所有节点

12-13 比较开销，更新最好的路径

6 OLSR 协议改进

阅读了一些论文及文献后，我们学习了一种可以生成最小的 MPR 集的 MPR 选择方案：

OLSR 协议中的 MPR 集算法只是用一跳结点为优先，因此往往存在一些冗余，此文献中的 O_MPR 集算法避免了 MPR 节点的冗余，在满足要求的情况下，得出最优解。

1. 设 S 为节点 Ni 的 MPR 集,且 S 初始化为除 N_willingness=WILL_NEVER 的全部节点的集合;
2. 分别计算 M1(Ni)中全部节点在 M2(Ni)中所能覆盖的节点数量;
3. S 中没有被选定的节点按以下条件先后退出:
   N_willingness 较低的优先;
   其次是 Reachability 较小的;
   最后是节点覆盖数量较少的
   检查此刻 M2(Ni)中还有没有 S 中某个节点所不能覆盖的二跳邻居节点,若存在则表示不能从 S 集合删除该节点,反之可删除;
4. 重复执行步骤 3,直到 M2(Ni)中的节点全被 S 中节点所覆盖。
   其中,N_willingness 是 0 和 7 之间的一个整数,表示节点携带和转发分组的意愿;WILL_NEVER 表示决不能被选为 MPR 节点;Reachability 是该节点一跳对称邻居节点的个数,但不包括节点本身和此时参与计算 MPR 的节点。
   该算法的时间复杂度,理想情况下是 O(n×log n),在最坏情况下是 O(n2)。