网络协议栈分析与设计课程大作业

OLSR 路由协议代码分析

学号	姓名	班级	负责模块	成绩
201693097	季天冬	软网 1604	第二章 运行总图 第四章 数据结构 第五章 5.1、5.3 第六章 设计与实践	
201692330	张振宇	软网 1604	第三章 文件与变量 第五章 5.2、5.4 第六章 设计与实践	
201692286	刘雨晴	软网 1602	第一章 OLSR 简介 第五章 5.5 第六章 设计与实践	

目录

1 OLSR 简介 3
1.1 OLSR 基本概念 3
1.2 OLSR 核心机制 3
2 OLSR 协议运行总图 4
3 文件与变量 5
3.1 文件列表 5
3.2 变量列表 5
3.3 配置变量 6
4 OLSR 数据结构 7
4.1 OLSR 消息结构 7
4.1.1 消息基本格式 7
4.1.2 HELLO 消息格式 7
4.1.3 TC 消息格式 8
4.2 OLSR 表结构 8
4.2.1 本地链路信息表 8
4.2.2 邻居表 9
4.2.3 MPR Selector 表 10
4.2.4 分组重复记录表 10
4.2.5 拓扑表、路由 11
5 OLSR 协议代码分析 12
5.1 链路感知 12
5.1.1 链路信息表更新 12
5.1.2 其他链路操作 13
5.2 邻居发现 14
5.2.1 邻居表表项状态 14
5.2.2 邻居表的操作 15
5.3 MPR 处理 17
5.3.1 MPR 选举算法 17
5.3.2 1MPR 函数 18
5.4 拓扑控制消息处理 19
5.5 路由计算 22
5.5.1 相关结构体 23
5.5.2 路由表计算 25
6 设计与实践 32
6.1 动机 32
6.2 简化 OLSR 实现 – PyOLSR 32
6.2.1 总体设计 32
6.2.2 具体实现 32
7 小结 35

1 OLSR 简介

1.1OLSR 基本概念

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。它是一种标准化的表驱动式路由协议，它是为了适应无线自组网的需求，对经典链路状态算法进行优化而形成的。协议用到的核心概念是多点中继机制，通过此机制，协议显著的减少了分组消息的开支。具体来讲，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洪泛范围；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个节点当其第一次收到一个控制消息时转发每一个消息，OLSR 协议中只有节点的 MPR 节点才转发该节点所发送的控制分组，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组的数量，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节点相连的链路信息，而只发布与其多点中继选择节点间的链路。OLSR 协议中，每个控制分组都携带有序列号，可以用来区分新旧信息，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 MPR Selector 信息，以帮助其他节点建立到它的路由，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保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。

1.2OLSR 核心机制

多点中继是 OLSR 协议的核心思想，目的是通过减少同一区域内相同控制分组的重复转发次数来减少网络中广播分组的数量。网络中某一节点 N 的邻居节点被分为两类：MPR 节点和非 MPR 节点。每个节点从其一跳邻居中选择自己的 MPR 集，使得该节点通过这个集合转发的 TC 分组能够覆盖该节点的所有两跳节点。节点 N 的 MPR 集是该节点邻居节点集的子集且满足以下条件：N 的每个两跳邻节点都必须有到达 MPR(N)的双向链路，且节点 N 与 MPR(N)间的链路也是双向的。MPR 集越小，协议性能越好。节点 N 发送的广播分组通过其 MPR 节点转发到达其他节点。而非 MPR 节点只接收控制分组并不转发。协议根据 MPR 集来计算到所有目的地的路由。网络中每个节点周期性的广播它的 MPR Selector 信息。每个节点收到后更新自己到每个已知节点的路由。因此，路由就是通过源到目的节点的 MPR 的逐跳节点序列。

采用 MPR 机制有效的减少了网络内控制分组重发的数量，避免了广播风暴。MRP 下的洪泛机制如下图所示：

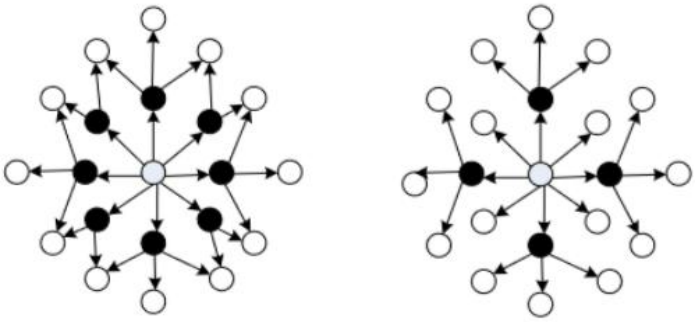


图 无选择性洪范与选择性洪范

2 OLSR 协议运行总图

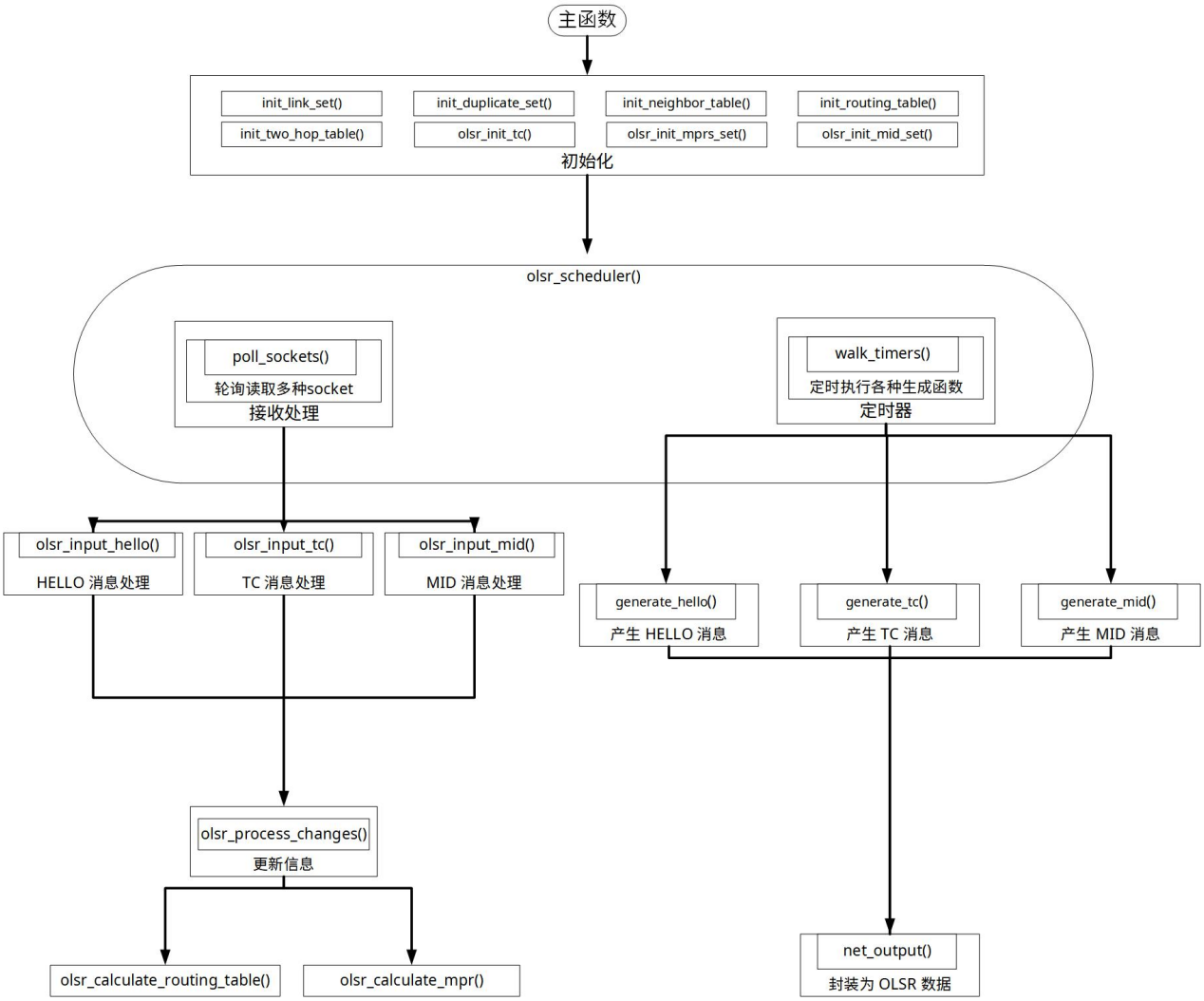


图 OLSR 协议运行总图

为了对复杂的协议有一个整体上的理解，我绘制了一个协议总图。该图从函数调用层面的大致描绘了整个协议的运行。从图中我们可以看出，OLSR 采用了轮询机制。程序进入主函数后，首先调用一系列初始化函数对协议用到的数据结构进行初始化。初始化完成后，主程序调用 olsr_scheduler 函数进入 scheduler 循环。

在 scheduler 循环中，主程序依次调用 poll_sockets 函数和 walk_timers 函数分别进行消息接收操作和产生操作。当进行消息接收操作时，程序轮询读取多种 socket。当发现可识别 socket，就调用相应的处理函数处理。其中，olsr_input_hello, olsr_input_hello 和 olsr_input_mid 分别处理对应的消息。随后程序会根据之前消息处理的结果更新自己存储的数据结构；当进行消息产生操作的时候，程序调用 walk_timers 函数，并根据到期情况，用指定的函数产生各种消息。其中， generate_hello、generate_tc 和 generate_mid 函数分别产生对应的消息。

除了上图中所列函数。OLSRD 中还存在许多其他的函数实现，比如一组处理终端字符信号 SIGINT 的函数，用于在终端结束程序时，释放 OLSR 程序运行时所产生的资源。限于篇幅，未能一一罗列。

3 文件与变量

3.1 文件列表

OLSR 代码规模巨大，现将后续代码分析会用到的以及一些重要文件罗列为下表：

文件	内容
lq_packet.h/c	定义了 OLSR、HELLO、TC 等数据首部以及其他数据结构
neighbor_table.h/c	定义了邻居表和一些相关处理函数
two_hop_neighbor_table.h/c	定义了二跳邻居表和一些相关处理函数
mpr_selector_set.h/c	定义了 MPR Selector 表和一些相关处理函数
duplicate_set.h/c	定义了分组重复信息表和一些相关处理函数
tc_set.h/c	定义了拓扑表和一些相关处理函数
routing_table.h/c	定义了路由表和一些相关处理函数
link_set.h/c	定义了链路信息表和一些相关处理函数
lq_mpr.c/h	nMPR 的一些操作
mpr.c/h	1MPR 的一些操作
generate_msg.c	HELLO、TC 等消息的生成函数
packet.h	消息数据包结构定义
olsr.c	定义一些全局函数，路由计算更新、错误处理等
olsr_cfg.h	定义一些全局常量
olsr_spf.c	spf 树的相关函数
process_routes.c	处理核心路由表
process_package.c	处理 HELLO 消息包
schedule.c	定时器处理

3.2 变量列表

一些重要的全局变量如下表：

全局变量	类型	描述
olsrport	uint16_t	OLSR 消息发送端口
rt_proto	uint8_t	路由的协议
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	拓扑消息是否改动 flag
changes_neighborhood	bool	邻居表信息是否改动 flag

3.3配置变量

OLSR 路由协议的一些配置变量罗列如下：

配置变量名	描述	缺省值
DEF_IP_VERSION	缺省 IP 协议域	AF_INET
DEF_USE_HYST	缺省消息迟滞	false
DEF_LQ_LEVEL	缺省链路质量等级	2
DEF_OLSRPORT	缺省 olsr 端口号	698
DEF_MIN_TC_VTIME	TC 消息 vtime 最小取值	0.0
DEF_GW_TYPE	缺省网关类型	GW_UPLINK_IPV46

4 OLSR 数据结构

4.1OLSR 消息结构

4.1.1 消息基本格式

File: lq_packet.h

```
55  struct olsr_common {
56    uint8_t type;
57    olsr_reltime vtime;
58    uint16_t size;
59    union olsr_ip_addr orig;
60    uint8_t ttl;
61    uint8_t hops;
62    uint16_t seqno;
63  };
```

为了协议的扩展性和兼容性，OLSR 的数据包采用了统一的格式。每个 OLSR 分组封装了多个消息，这些消息使用同样的首部。OLSR 协议基本数据包首部的代码定义如上图所示：

56 type：消息类型，

57 vtime：分组所携带消息的有效期，

58 size：消息长度，单位 byte，从消息类型开始处到下一个消息类型开始出的长度，

59 orig: 产生该消息节点的主地址，

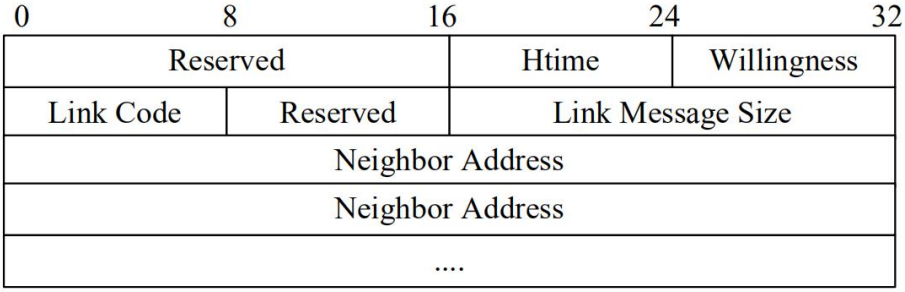
60 ttl: 消息被传送的最大跳数。消息每被重传前 ttl 减一，ttl 为 0 和 1 的消息不会被重传，

61 hops: 跳数，初始化为 0，消息每被重传前 hops 加一，

62 seqno: 消息标识号，由节点维护，节点每产生一个消息，此表示号加一，用来保证每个消息不会被该节点重传两次。

4.1.2 HELLO 消息格式

在 OLSR 消息中，HELLO 消息被用来进行邻居发现。每个节点周期性的向一跳范围节点广播 HELLO 分组，分组格式如下图：



具体的，相关代码定义如下：

File: lq_packet.h

```
107  struct lq_hello_info_header {
108    uint8_t link_code;
109    uint8_t reserved;
110    uint16_t size;
111  };

113  struct lq_hello_header {
114    uint16_t reserved;
115    uint8_t htime;
116    uint8_t will;
117  };
```

108 link_code: 链路类型：描绘了发送 HELLO 的节点与随后邻居列表中的邻节点间的链路类型，
109 reserved: 保留字段，初始化为全 0，
110 size: 本链路消息大小，
114 reserved: 保留字段，初始化为全 0，
115 htime: HELLO 消息发送时间间隔，
116 will: 节点意愿，比如意愿为 WILLING_NEVER 的节点不会被选为 MPR，具有 WILLING_ALWAYS 意愿的节点总是会被选为 MPR，默认值为 WILLING_DEFAULT。

4.1.3 TC 消息格式

TC 消息提供了拓扑信息，用于计算路由，TC 消息的代码定义如下：

File: lq_packet.h

```
120  struct lq_tc_message {
121    struct olsr_common comm;
122    union olsr_ip_addr from;
123    uint16_t ansn;
124    struct tc_mpr_addr *neigh;
125  };
```

123 ansn: 本节点收到的最近一个 TC 分组的序列号，用于防重。

4.2 OLSR 表结构

4.2.1 本地链路信息表

本地链路信息表记录了本节点与邻居节点的链路信息。具体的，在 link_set 中定义了一个结构体 link_entry，代码节选如下：

File: link_set.h

```
58  struct link_entry {
59    union olsr_ip_addr local_iface_addr;
60    union olsr_ip_addr neighbor_iface_addr;
61    const struct interface *inter;
62    char *if_name;
63    struct timer_entry *link_timer;
64    struct timer_entry *link_sym_timer;
65    uint32_t ASYM_time;
66    olsr_reltime vtime;
67    struct neighbor_entry *neighbor;
68    uint8_t prev_status;
```

59 local_iface_addr: 本地节点接口地址，
60 neighbor_iface_addr: 邻节点接口地址，
66 vtime: 链路信息有效时间。

4.2.2 邻居表

每个节点通过收发 HELLO 消息获取邻居信息，用于维护邻居表。具体的，neighbor_table.h 和 two_hop_neighbor_table.h 中分别定义了结构体 neighbor_entry 和 neighbor_2_list_entry，其代码定义如下：

File: neighbor_table.h

```
58  struct neighbor_entry {
59    union olsr_ip_addr neighbor_main_addr;
60    uint8_t status;
61    uint8_t willingness;
62    bool is_mpr;
63    bool was_mpr;                    /* Used to detect changes in MPR */
64    bool skip;
65    int neighbor_2_nocov;
66    int linkcount;
67    struct neighbor_2_list_entry neighbor_2_list;
68    struct neighbor_entry *next;
69    struct neighbor_entry *prev;
70  };
```

59 neighbor_main_addr: 一跳邻居的主地址，
60 status: 与该邻居之间的链路状态，
61 willingness: 该邻居节点的意愿，、

File: two_hop_neighbor_table.h

```
60  struct neighbor_2_entry {
61    union olsr_ip_addr neighbor_2_addr;
62    uint8_t mpr_covered_count;          /*used in mpr calculation */
63    uint8_t processed;                  /*used in mpr calculation */
64    int16_t neighbor_2_pointer;         /* Neighbor count */
65    struct neighbor_list_entry neighbor_2_nblist;
66    struct neighbor_2_entry *prev;
67    struct neighbor_2_entry *next;
68  };
```

59 neighbor_2_addr: 二跳邻居地址。

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 节点的节点列表，代码实现如下：

File: mpr_selector_set.h

```
47  struct mpr_selector {
48    union olsr_ip_addr MS_main_addr;
49    struct timer_entry *MS_timer;
50    struct mpr_selector *next;
51    struct mpr_selector *prev;
52  };
```

48 MS_main_addr: MPR Selector 节点主地址，
49 MS_timer: 用于维护信息时间有效性。

4.2.4 分组重复记录表

为了避免重复，节点维护分组分组重复记录表，其代码定义如下：

File: duplicate_set.h

```
55  struct dup_entry {
56    struct avl_node avl;
57    union olsr_ip_addr ip;
58    uint16_t seqnr;
59    uint16_t too_low_counter;
60    uint32_t array;
61    uint32_t valid_until;
62  };
```

58 seqnr: TC 分组序列号，用于区分，
61 valid_until: 该表项有效期。

4.2.5 拓扑表、路由

节点从 TC 消息中获取网络拓扑和路由信息，存储在拓扑表和路由中，这两种表的代码定义如下：

File: tc_set.h

```
71  struct tc_entry {
72    struct avl_node vertex_node;          /* node keyed by ip address */
73    union olsr_ip_addr addr;              /* vertex_node key */
74    struct avl_node cand_tree_node;       /* SPF candidate heap, node keyed by path_etx */
75    olsr_linkcost path_cost;              /* SPF calculated distance, cand_tree_node key *
76    struct list_node path_list_node;      /* SPF result list */
77    struct avl_tree edge_tree;            /* subtree for edges */
78    struct avl_tree prefix_tree;          /* subtree for prefixes */
79    struct link_entry *next_hop;          /* SPF calculated link to the 1st hop neighbor *
80    struct timer_entry *edge_gc_timer;    /* used for edge garbage collection */
81    struct timer_entry *validity_timer;   /* tc validity time */
82    uint32_t refcount;                    /* reference counter */
83    uint16_t msg_seq;                     /* sequence number of the tc message */
84    uint8_t msg_hops;                     /* hopcount as per the tc message */
85    uint8_t hops;                         /* SPF calculated hopcount */
86    uint16_t ansn;                        /* ANSN number of the tc message */
87    uint16_t ignored;                     /* how many TC messages ignored in a sequence
88                                             (kindof emergency brake) */
89    uint16_t err_seq;                     /* sequence number of an unplausible TC */
90    bool err_seq_valid;                   /* do we have an error (unplauible seq/ansn) */
91  };
```

File: routing_table.h

```
85  struct rt_entry {
86    struct olsr_ip_prefix rt_dst;
87    struct avl_node rt_tree_node;
88    struct rt_path *rt_best;              /* shortcut to the best path */
89    struct rt_nexthop rt_nexthop;         /* nexthop of FIB route */
90    struct rt_metric rt_metric;           /* metric of FIB route */
91    struct avl_tree rt_path_tree;
92    struct list_node rt_change_node;      /* queue for kernel FIB add/chg/del */
93  };
```

5 OLSR 协议代码分析

整体上来说，OLSR 协议是这样一个运行流程：节点间通过周期性的交换 HELLO 消息包，完成链路感知和邻居发现；节点利用邻居信息，进行 MRP 计算；节点间通过周期性的交换 TC 消息包，计算拓扑结构。最终，节点根据拓扑结构，基于 MRP 利用迪杰斯特拉算法进行路由计算。

本章将通过具体代码分析，对上述流程的各个环节进行解读。

5.1链路感知

OLSR 协议中，链路感知是通过周期性收发 HELLO 消息包实现。节点通过维护本地链路信息表存储与邻节点的链路信息。

5.1.1 链路信息表更新

具体的，每当节点收到一个 HELLO 消息包时，节点需要更新其链路信息表，更新规则如下：

1. 一旦收到一个 HELLO 消息，在本地链路信息表中不存在表项满足：
   neighbor_iface_addr == HELLO 分组的源地址
   则创建一个新的表项，使得 neighbor_iface_addr == HELLO 分组的源地址。
2. 如果存在表项，则执行更新操作，更新的函数代码如下：

File: link_set.c

```
696   entry->vtime = message->vtime;
697   entry->ASYM_time = GET_TIMESTAMP(message->vtime);

709     /* L_SYM_time = current time + validity time */
710     olsr_set_timer(&entry->link_sym_timer, message->vtime, OLSR_LINK_SYM_JITTER, OLSR_
711                    entry, 0);
712
713     /* L_time = L_SYM_time + NEIGHB_HOLD_TIME */
714     olsr_set_link_timer(entry, message->vtime + NEIGHB_HOLD_TIME * MSEC_PER_SEC);
715     break;
716   default:;
717   }
718
719   /* L_time = max(L_time, L_ASYM_time) */
720   if (entry->link_timer && (entry->link_timer->timer_clock < entry->ASYM_time)) {
721     olsr_set_link_timer(entry, TIME_DUE(entry->ASYM_time));
722   }
```

697 L_ASYM_TIME = 有效时间戳，即 当前时间 + 有效时间，

710 L_SYM_TIME = 当前时间 + 有效时间，

714 L_TIME = L_SYM_TIME + NEIGHB_HOLD_TIME，

720 L_TIME = MAX(L_TIME, L_ASYM_TIME)。

其中，有效时间通过消息首部的 vtime 计算。

5.1.2 其他链路操作

重置链路信息函数，代码如下：

File: link_set.c

```
92  void olsr_reset_all_links(void) {
93    struct link_entry *link;
94
95    OLSR_FOR_ALL_LINK_ENTRIES(link) {
96      link->ASYM_time = now_times-1;
97
98      olsr_stop_timer(link->link_sym_timer);
99      link->link_sym_timer = NULL;
100
101     link->neighbor->is_mpr = false;
102     link->neighbor->status = NOT_SYM;
103   } OLSR_FOR_ALL_LINK_ENTRIES_END(link)
104
105
106   OLSR_FOR_ALL_LINK_ENTRIES(link) {
107     olsr_expire_link_sym_timer(link);
108     olsr_clear_hello_lq(link);
109   } OLSR_FOR_ALL_LINK_ENTRIES_END(link)
110 }
```

95 遍历所有节点，

96 ASYM_TIME 设置 now_times – 1，

99 link_sym_timer 设置为空指针，

101-102 邻居的 is_mpr 置为 false，status 置为 NOT_SYM

删除链路表项函数，代码如下：

File: link_set.c

```
360 static void
361 olsr_delete_link_entry(struct link_entry *link)
362 {
363   struct tc_edge_entry *tc_edge;
364
365   /* delete tc edges we made for SPF */
366   tc_edge = olsr_lookup_tc_edge(tc_myself, &link->neighbor_iface_addr);
367   if (tc_edge != NULL) {
368     olsr_delete_tc_edge_entry(tc_edge);
369   }
370
371
```

```
372     /* Delete neighbor entry */
373     if (link->neighbor->linkcount == 1) {
374       olsr_delete_neighbor_table(&link->neighbor->neighbor_main_addr);
375     } else {
376       link->neighbor->linkcount--;
377     }
378
379     /* Kill running timers */
380     olsr_stop_timer(link->link_timer);
381     link->link_timer = NULL;
382     olsr_stop_timer(link->link_sym_timer);
383     link->link_sym_timer = NULL;
384     olsr_stop_timer(link->link_hello_timer);
385     link->link_hello_timer = NULL;
386     olsr_stop_timer(link->link_loss_timer);
387     link->link_loss_timer = NULL;
388     list_remove(&link->link_list);
389
390     free(link->if_name);
391     free(link);
392
393     changes_neighborhood = true;
394   }
```

366-369 删除 SPF 的 TC 边表项，该表项使用邻居端口地址调用 olsr_lookup_tc_edge 函数找到，

373-377 删除邻居表项：如果邻居表项的链路计数>1 则该技术减 1，否则调用 olsr_delete_neighbor_table 删除该邻居表项，

380-391 清空定时器，释放资源，

393 将 change_neighborhood 的 flag 置为真，后续程序中会因此执行更新邻居表。

5.2邻居发现

在 OLSR 协议中，每个节点都会检测其与哪个节点互为邻居。每隔一个固定的时间，节点就会向一跳范围内的节点广播一次 HELLO 消息，用于更新拓扑消息。HELLO 消息包含了本节点的邻居以及链路状态信息。因而，通过 HELLO 消息的交互，各节点都知道了自己的一跳和两跳邻居信息。

5.2.1 邻居表表项状态

OLSR 协议的链路有对称和非对称两种状态。在不考虑有效期的情况下，当节点 X 收到邻居节点 Y 的 HELLO 消息时，X 将 Y 放入邻居集中，并将其状态设置为非对称状态。之后，当 X 向 Y 发送 HELLO 消息时，Y 从消息中得知 Y 到 X 为非对称链路的信息，Y 随后将 X 在邻居表中的状态更新为对称状态。同理 X 在收到 Y 的 HELLO 消息时得知 Y 到 X 为对称状态链路，于是 X 更新 Y 的状态为对称状态。

具体的，在 OLSRD 代码中，邻居状态被定义为：NOT_SYM: 0 和 SYM: 1：

```
118   /*
119    *Neighbor status
120    */
121
122   #define NOT_SYM              0
123   #define SYM                  1
```

5.2.2 邻居表的操作

初始化邻居表函数，代码如下：

File: neighbor_table.c

```
56   void
57   olsr_init_neighbor_table(void)
58   {
59     int i;
60
61     for (i = 0; i < HASHSIZE; i++) {
62       neighbortable[i].next = &neighbortable[i];
63       neighbortable[i].prev = &neighbortable[i];
64     }
65   }
```

61-64 通过 For 循环，将每一项的前驱和后继表项设置为自身。

删除二跳邻居节点记录，代码如下：

File: neighbor_table.c

```
70   static void
71   olsr_del_nbr2_list(struct neighbor_2_list_entry *nbr2_list)
72   {
73     struct neighbor_entry *nbr;
74     struct neighbor_2_entry *nbr2;
75
76     nbr = nbr2_list->nbr2_nbr;
77     nbr2 = nbr2_list->neighbor_2;
78
79     if (nbr2->neighbor_2_pointer < 1) {
80       DEQUEUE_ELEM(nbr2);
81       free(nbr2);
82     }
```

```
83    /*
84     * Kill running timers.
85     */
86    olsr_stop_timer(nbr2_list->nbr2_list_timer);
87    nbr2_list->nbr2_list_timer = NULL;
88    /* Dequeue */
89    DEQUEUE_ELEM(nbr2_list);
90    free(nbr2_list);
91    /* Set flags to recalculate the MPR set and the routing table */
92    changes_neighborhood = true;
93    changes_topology = true;
94  }
```

76-77 将两个备用指针指向邻居的邻居和邻居

79-81 如果这个节点没有两跳邻居，就删除并释放内存

86-93 杀掉计时器进程，并且将该节点删除，然后将邻居改变的变量设为 True。

根据地址删除两条邻居节点，代码如下：

File: neighbor_table.c

```
104  int
105  olsr_delete_neighbor_2_pointer(struct neighbor_entry *neighbor, struct neighbor_2_entry *neigh2)
106  {
107    struct neighbor_2_list_entry *nbr2_list;
108
109    nbr2_list = neighbor->neighbor_2_list.next;
110
111    while (nbr2_list != &neighbor->neighbor_2_list) {
112      if (nbr2_list->neighbor_2 == neigh2) {
113        olsr_del_nbr2_list(nbr2_list);
114        return 1;
115      }
116      nbr2_list = nbr2_list->next;
117    }
118    return 0;
119  }
```

111-117 进行循环。如果这个两跳邻居不是核心点的两跳邻居，那么就进行循环。首先判断是不是两跳邻居节点。然后判断这个节点的邻居是不是都是两跳节点。如果是的话，就删除这个邻居节点。

5.3 MPR 处理

5.3.1 MPR 选举算法

MPR 选举算法的原则是本节点可以通过选出的 MPR 节点到达所有的二跳邻居节点，且 MPR 节点的数量尽可能少。RFC3626 中提出了一种 MPR 贪婪选择算法，被广泛使用，该算法定义如下：

(1) 一跳邻居节点中所有 willingness 为 WILL_ALWAYS 的节点构成初始 MPR 集。

(2) 将 N 中满足如下条件的节点加入 MPR 集：它是到达二跳邻居表中某节点的唯一节点。移除二跳邻居表中被 MPR 集覆盖的节点。

(3) 当二跳邻居表中不存在不被 MPR 集中的任何节点覆盖的节点时，算法结束，否则：

① 计算一跳邻居表中每个节点的可达性。

② 从可达性非 0 且 willingness 最高的节点中选择一个 MPR，有多个选择时，选择可达性高的，可达性相同时，选择深度高的节点。

(4) 将节点每个接口的 MPR 集组合在一起，即为该节点的 MPR 集合。

具体的，OLSRD 中通过在 lq_mpr.c 中定义的 olsr_calculate_lq_mpr 函数计算 MPR，该函数较长，逐块分析如下：

```
17    bool mpr_changes = false;
18
19    OLSR_FOR_ALL_NBR_ENTRIES(neigh) {
20      neigh->was_mpr = neigh->is_mpr;
21      neigh->is_mpr = false;
22      if (neigh->status == NOT_SYM || neigh->willingness != WILL_ALWAYS) {
23        continue;
24      }
25      neigh->is_mpr = true;
26      if (neigh->is_mpr != neigh->was_mpr) {
27        mpr_changes = true;
28      }
29    }
30    OLSR_FOR_ALL_NBR_ENTRIES_END(neigh);
```

17 设置 mpr_changes 的 flag 为假，

19-30 遍历所有邻居表项，进行一些初始工作，

20 保存之前的 MPR 状态于 was_mpr，

21 清空 is_mpr，

22-24 对于非对称和 willing 不为 WILL_ALWAYS 的节点，直接跳过，

25 否则，初始化这些节点为 MPR 节点，

26-27 更新 mpr_changes 的 flag。

```
32  for (i = 0; i < HASHSIZE; i++) {
33    for (neigh2 = two_hop_neighbortable[i].next; neigh2 != &two_hop_neighbortable[i]; neigh2 = neigh2->next) {
34      best_1hop = LINK_COST_BROKEN;
35      neigh = olsr_lookup_neighbor_table(&neigh2->neighbor_2_addr);
36
37      if (neigh != NULL && neigh->status == SYM) {
38        struct link_entry *lnk = get_best_link_to_neighbor(&neigh->neighbor_main_addr);
39        if (!lnk)
40          continue;
41        best_1hop = lnk->linkcost;
42
43        for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
44          if (walker->path_linkcost < best_1hop)
45            break;
46
47        if (walker == &neigh2->neighbor_2_nblist)
48          continue;
49      }
```

33 遍历邻居环形表的所有二跳邻居，
35 通过主地址查询该二条邻居是否同时是一跳邻居，
37-49 如果的确如此，且邻居状态是对对称的，则检查链路质量。如果直连链路比通过 MPR 的最佳路由要好，则使用直连链路而不把该邻居看作二条邻居进而为之选择 MPR 节点。

否则，

```
51        for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
52          walker->neighbor->skip = false;
53        for (k = 0; k < olsr_cnf->mpr_coverage; k++) {
54          neigh = NULL;
55          best = LINK_COST_BROKEN;
56          for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
57            if (walker->neighbor->status == SYM && !walker->neighbor->skip && walker->path_linkcost < best) {
58              neigh = walker->neighbor;
59              best = walker->path_linkcost;
60            }
61          if ((neigh != NULL) && (best < best_1hop)) {
62            neigh->is_mpr = true;
63            neigh->skip = true;
64            if (neigh->is_mpr != neigh->was_mpr)
65              mpr_changes = true;
66          }
67          else
68            break;
69        }
70      }
71    }
```

51-52 将所有一跳节点标记为未选择，
56-60 选择还未被选择的最佳节点，即拥有较少 path_linkcost 的节点，
61-63 当存在上述节点，且该二跳路径优于最佳一跳路径时，将该节点设置为 MPR 节点，
64-65 更新 mpr_change 的 flag。

5.3.2 1MPR 函数

在 mpr.h 中，定义了一些 1MPR 操作的函数，这些操作与 nMPR 情况类似，举例如下：

1MPR 的 WILL_ALWAYS 节点添加函数，代码如下：

```
357  static uint16_t
358  add_will_always_nodes(void)
359  {
360    struct neighbor_entry *a_neighbor;
361    uint16_t count = 0;
362    OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
363      struct ipaddr_str buf;
364      if ((a_neighbor->status == NOT_SYM) || (a_neighbor->willingness != WILL_ALWAYS)) {
365        continue;
366      }
367      olsr_chosen_mpr(a_neighbor, &count);
368    }
369    OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
370    return count;
371  }
```

364 忽略非对称且 willingness 不是 WILL_ALWAYS 的节点，

367 其余的添加为 MPR 节点。

1MPR 的清除 MPR 节点函数，代码如下：

```
236  static void
237  olsr_clear_mprs(void)
238  {
239    struct neighbor_entry *a_neighbor;
240    struct neighbor_2_list_entry *two_hop_list;
241
242    OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
243
244      /* Clear MPR selection. */
245      if (a_neighbor->is_mpr) {
246        a_neighbor->was_mpr = true;
247        a_neighbor->is_mpr = false;
248      }
249
250      /* Clear two hop neighbors coverage count/ */
251      for (two_hop_list = a_neighbor->neighbor_2_list.next; two_hop_list != &a_neighbor->neighbor_2_list;
252           two_hop_list = two_hop_list->next) {
253        two_hop_list->neighbor_2->mpr_covered_count = 0;
254      }
255
256    }
257    OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
258  }
```

245-248 清空 is_mpr 且填写 was_mpr，

251-254 遍历将两条邻居节点计数置 0。

1MPR 的 MPR 选择函数 olsr_calculate_mpr 与上一节所分析的 olsr_calculate_lq_mpr 相似，故不赘述。

5.4拓扑控制消息处理

TC 消息的生成，代码如下：

File: generate_msg.c

```
81  void
82  generate_tc(void *p)
83  {
84    struct tc_message tcpacket;
85    struct interface *ifn = (struct interface *)p;
86
87    olsr_build_tc_packet(&tcpacket);
88
89    if (queue_tc(&tcpacket, ifn) && TIMED_OUT(ifn->fwdtimer)) {
90      set_buffer_timer(ifn);
91    }
92
93    olsr_free_tc_packet(&tcpacket);
94  }
```

84-90 MID 消息通过 olsr_build_tc_packet() 函数生成之后放在 MID 队列中。时间戳满的时候，调用 set_buffer_timer()设置定时器。最后从给定接口 ifn 释放消息，最后调用 olsr_free_tc_packet()释放所占内存。

为了构建拓扑信息库，每个被选择为 MPR 的节点必须广播拓扑控制 TC 消息。这些通过 TC 消息扩散到网络中的信息将有所帮助每个节点计算其路由表。

当节点的通告链路集变为空时，该节点应当在等于其先前发送 TC 消息的“有效时间”的持续时间期间仍然发送空 TC 消息，以便使先前的 TC 消息无效，直到有节点加入到通告链路集。

拓扑表集合的初始化与删除，代码如下：

File: tc_set.c

```
185  void
186  olsr_init_tc(void)
187  {
188    OLSR_PRINTF(5, "TC: init topo\n");
189
190    avl_init(&tc_tree, avl_comp_default);
191
192    /*
193     * Get some cookies for getting stats to ease troubleshooting.
194     */
195    tc_edge_gc_timer_cookie = olsr_alloc_cookie("TC edge GC", OLSR_COOKIE_TYPE_TIMER);
196    tc_validity_timer_cookie = olsr_alloc_cookie("TC validity", OLSR_COOKIE_TYPE_TIMER);
197
198    tc_edge_mem_cookie = olsr_alloc_cookie("tc_edge_entry", OLSR_COOKIE_TYPE_MEMORY);
199    olsr_cookie_set_memory_size(tc_edge_mem_cookie, sizeof(struct tc_edge_entry) + active_lq_handler->tc_lq_size);
200
201    tc_mem_cookie = olsr_alloc_cookie("tc_entry", OLSR_COOKIE_TYPE_MEMORY);
202    olsr_cookie_set_memory_size(tc_mem_cookie, sizeof(struct tc_entry));
203
204    /*
205     * Add a TC entry for ourselves.
206     */
207    tc_myself = olsr_add_tc_entry(&olsr_cnf->main_addr);
208  }
```

函数功能：初始化拓扑集。

189 调用 avl_init()初始化一个 avl 树。

195-198 对拓扑表集合的初始化，从 cookie 中获取相应的值为拓扑表集合的属性赋值.

202 调用 olsr_add_tc_entry()配置一个 entry，并将其插入到 tc 树中，同时会导致一条 rt_path 插入到路由表中。

```
278  void
279  olsr_delete_tc_entry(struct tc_entry *tc)
280  {
281    struct tc_edge_entry *tc_edge;
282    struct rt_path *rtp;
283  #if 0
284    struct ipaddr_str buf;
285    OLSR_PRINTF(1, "TC: del entry %s\n", olsr_ip_to_string(&buf, &tc->addr));
286  #endif
287
288    /* delete gateway if available */
289  #ifdef LINUX_NETLINK_ROUTING
290    olsr_delete_gateway_entry(&tc->addr, FORCE_DELETE_GW_ENTRY);
291  #endif
292    /*
293     * Delete the rt_path for ourselves.
294     */
295    olsr_delete_routing_table(&tc->addr, olsr_cnf->maxplen, &tc->addr);
296
297    /* The edgetree and prefix tree must be empty before */
298    OLSR_FOR_ALL_TC_EDGE_ENTRIES(tc, tc_edge) {
299      olsr_delete_tc_edge_entry(tc_edge);
300    } OLSR_FOR_ALL_TC_EDGE_ENTRIES_END(tc, tc_edge);
301
302    OLSR_FOR_ALL_PREFIX_ENTRIES(tc, rtp) {
303      olsr_delete_rt_path(rtp);
304    } OLSR_FOR_ALL_PREFIX_ENTRIES_END(tc, rtp);
305
306    /* Stop running timers */
307    olsr_stop_timer(tc->edge_gc_timer);
308    tc->edge_gc_timer = NULL;
309    olsr_stop_timer(tc->validity_timer);
310    tc->validity_timer = NULL;
311
312    avl_delete(&tc_tree, &tc->vertex_node);
313    olsr_unlock_tc_entry(tc);
314  }
```

函数功能：删除一条 TC entry。

278-289 若宏定义 LINUX_NETLINK_ROUTING 条件，则删除网关信息。其中网关的删除操作需要对网关协议(IPv4 或 IPv6)、网关信息是否为空以及网关信息保存时间是否为空进行判断；

294 删除本地的路由表中相应的 rt_path；

302-306 停止相应的定时器，将 timers 参数都设置为空，保证数据和设置的彻底删除；

307-308 从 avl 树中删除相应的节点，并将该 tc_entry 的引用值减 1。

TC 消息处理，代码如下：

File: tc_set.c

```
789  bool
790  olsr_input_tc(union olsr_message * msg, struct interface * input_if __attribute__ ((unused)), union olsr_ip_addr * from_addr)
791  {
792    struct ipaddr_str buf;
793    uint16_t size, msg_seq, ansn;
794    uint8_t type, ttl, msg_hops, lower_border, upper_border;
795    olsr_reltime vtime;
796    union olsr_ip_addr originator;
797    const unsigned char *limit, *curr;
798    struct tc_entry *tc;
799    bool emptyTC;
800
801    union olsr_ip_addr lower_border_ip, upper_border_ip;
802    int borderSet = 0;
803
804    curr = (void *)msg;
805    if (!msg) {
806      return false;
807    }
808
809    /* We are only interested in TC message types. */
810    pkt_get_u8(&curr, &type);
811    if ((type != LQ_TC_MESSAGE) && (type != TC_MESSAGE)) {
812      return false;
813    }
814
815    /*
816     * If the sender interface (NB: not originator) of this message
817     * is not in the symmetric 1-hop neighborhood of this node, the
818     * message MUST be discarded.
819     */
820    if (check_neighbor_link(from_addr) != SYM_LINK) {
821      OLSR_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr_ip_to_string(&buf, from_addr));
822      return false;
823    }
```

810-813 当节点接收到 TC 消息时，只关心其消息类型。当其类型不等于 LQ_TC_MESSAGE 或 TC_MESSAGE 时，直接返回 false，将包丢弃。

814-818 TC 消息接收者在接收到消息时判断发送者接口信息，若发送者并非是对称一跳邻居，那么该包将会被丢弃。

819-822 一旦接收到 TC 消息，必须根据消息头的 Vtime 字段计算有效时间。

5.5路由计算

节点通过 TC 消息的扩散即可获得全网拓扑图，之后就可以根据邻居表，两跳邻居表以及拓扑表，独立的按照 Dijkstra 算法计算出路由表。通过每个节点都有的路由表，可以寻找路径信息。路由表记录已知网络中的每个目的地，被破坏或者仅部分已知的路由信息不会被记录。

路由信息格式如下：

R_dest_addr	R_next_addr	R_dist	R_iface_addr

如果信息表中任何信息有所改变，则必须重新计算路由表，更新网络中每个目的地址的路由信息。即当邻居表，两跳邻居表或者拓扑表发生变化的时候，都需要重新选择路由，根据计算结果重新更新路由表。这个更新不在网络中进行，也不发送任何消息。

操作系统的路由体系可以分为负责与其他节点交换信息，计算到其他节点的正确路由的“路由功能模块”，另一部分则是根据内核路由表，将数据分组通过正确的网络端口发送到下一跳的“转发功能模块”。这样，即使不改变“转发功能模块”，也可以通过修改“路由功能模块”来实现不同的路由协议。

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

5.5.1 相关结构体

File: routing_table.h

```
66  /* a composite metric is used for path selection */
67  struct rt_metric {
68    olsr_linkcost cost;
69    uint32_t hops;
70  };
```

此结构体为路由选择时使用的矩阵：

`68 cost`: 两个路由点之间的花销，

`69 hops`: 两个路由点之间的跳数。

File: routing_table.h

```
72  /* a nexthop is a pointer to a gateway router plus an interface */
73  struct rt_nexthop {
74    union olsr_ip_addr gateway;          /* gateway router */
75    int iif_index;                       /* outgoing interface index */
76  };
77
```

`74 gateway`: 下一跳的网关（IPv4 或者 IPv6），

`75 iif_index`: 接口索引。

File: routing_table.h

```
85  struct rt_entry {
86    struct olsr_ip_prefix rt_dst;
87    struct avl_node rt_tree_node;
88    struct rt_path *rt_best;             /* shortcut to the best path */
89    struct rt_nexthop rt_nexthop;        /* nexthop of FIB route */
90    struct rt_metric rt_metric;          /* metric of FIB route */
91    struct avl_tree rt_path_tree;
92    struct list_node rt_change_node;     /* queue for kernel FIB add/chg/del */
93  };
94
```

每个 RIB 节点都会有一个路由接口，包含了最佳路径的下一个网关信息，而且是 rt_path_tree 的根，同样也包含了在所有路由信息里的最佳路径：

86 `rt_dst`: 该信息的路由地址与前缀长度，

87 `rt_tree_node`：包含了 val_tree 的信息。

File: routing_table.h

```
106  struct rt_path {
107    struct rt_entry *rtp_rt;             /* backpointer to owning route head */
108    struct tc_entry *rtp_tc;             /* backpointer to owning tc entry */
109    struct rt_nexthop rtp_nexthop;
110    struct rt_metric rtp_metric;
111    struct avl_node rtp_tree_node;       /* global rtp node */
112    union olsr_ip_addr rtp_originator;   /* originator of the route */
113    struct avl_node rtp_prefix_tree_node; /* tc entry rtp node */
114    struct olsr_ip_prefix rtp_dst;       /* the prefix */
115    uint32_t rtp_version;                /* for detection of outdated rt_paths */
116    uint8_t rtp_origin;                  /* internal, MID or HNA */
117  };
118
```

该结构体包含了 rt_path 的信息，接受到的 rt_path 会被加入到路由信息表中，用于计算 best_path。

File: routing_table.h

```
179  /**
180   * IPv4 <-> IPv6 wrapper
181   */
182  union olsr_kernel_route {
183    struct {
184      struct sockaddr rt_dst;
185      struct sockaddr rt_gateway;
186      uint32_t metric;
187    } v4;
188
189    struct {
190      struct in6_addr rtmsg_dst;
191      struct in6_addr rtmsg_gateway;
192      uint32_t rtmsg_metric;
193    } v6;
194  };
195
```

上图为 IPv4 和 IPv6 的核心路由表结构。核心路由表表现添加和删除主要受目的地，网关和标志设置的影响。标志位的设置还决定了传输数据时，添加的路由表项是否发挥作用。

File: routing_table.h

```
129  enum olsr_rt_origin {
130    OLSR_RT_ORIGIN_MIN,
131    OLSR_RT_ORIGIN_INT,
132    OLSR_RT_ORIGIN_MID,
133    OLSR_RT_ORIGIN_HNA,
134    OLSR_RT_ORIGIN_MAX
135  };
```

OLSR 协议中有 3 种路由类型。INT 类型（内部路由）通过 TC 消息接受生成，MID 则是接受 MID 消息生成，而 HNA 路由来自 HNA 公告。

5.5.2 路由表计算

File: routing_table.c

```
167  void
168  olsr_init_routing_table(void)
169  {
170    OLSR_PRINTF(5, "RIB: init routing tree\n");
171
172    /* the routing tree */
173    avl_init(&routingtree, avl_comp_prefix_default);
174    routingtree_version = 0;
175
176    /*
177     * Get some cookies for memory stats and memory recycling.
178     */
179    rt_mem_cookie = olsr_alloc_cookie("rt_entry", OLSR_COOKIE_TYPE_MEMORY);
180    olsr_cookie_set_memory_size(rt_mem_cookie, sizeof(struct rt_entry));
181
182    rtp_mem_cookie = olsr_alloc_cookie("rt_path", OLSR_COOKIE_TYPE_MEMORY);
183    olsr_cookie_set_memory_size(rtp_mem_cookie, sizeof(struct rt_path));
184  }
```

此函数为初始化路由表。先通过 avl_init()初始化一个 avl 树，维护一个版本号（初始化为 0）routingtree_version 用以检测 rt_entry 和 rt_path 子树中信息是否过时。接下来则为 rt_entry 和 rt_path 分配内存，创建相应的 cookie。

File: routing_table.c

```
194  struct rt_entry *
195  olsr_lookup_routing_table(const union olsr_ip_addr *dst)
196  {
197    struct avl_node *rt_tree_node;
198    struct olsr_ip_prefix prefix;
199
200    prefix.prefix = *dst;
201    prefix.prefix_len = olsr_cnf->maxplen;
202
203    rt_tree_node = avl_find(&routingtree, &prefix);
204
205    return rt_tree_node ? rt_tree2rt(rt_tree_node) : NULL;
206  }
```

此函数为在 avl 树里找到一个地址的路由表条目。根据参数地址并配上设置的最长前缀长度，调用 avl_fing()函数在 routingtree 里找到该地址的 rt_entry。如果找到了，则利用 rt_tree2rt()函数将返回节点转化为 et_entry 类型，没找到则返回空。

File: routing_table.c

```
211  void
212  olsr_update_rt_path(struct rt_path *rtp, struct tc_entry *tc, struct link_entry *link)
213  {
214
215    rtp->rtp_version = routingtree_version;
216
217    /* gateway */
218    rtp->rtp_nexthop.gateway = link->neighbor_iface_addr;
219
220    /* interface */
221    rtp->rtp_nexthop.iif_index = link->inter->if_index;
222
223    /* metric/etx */
224    rtp->rtp_metric.hops = tc->hops;
225    rtp->rtp_metric.cost = tc->path_cost;
226  }
```

此函数用于更新 rt_path。每条路径都是周期性更新的。修改维护的 routingtree_version 的值，更新网关地址（rtp_nexthop.geteway），接口地址（rtp_nexthop.iif_index），跳数（rtp_metric.hops）和路径花销（rtp_metric.cost），都改为新接受到的值。

File: routing_table.c

```
231  static struct rt_entry *
232  olsr_alloc_rt_entry(struct olsr_ip_prefix *prefix)
233  {
234    struct rt_entry *rt = olsr_cookie_malloc(rt_mem_cookie);
235    if (!rt) {
236      return NULL;
237    }
238
239    memset(rt, 0, sizeof(*rt));
240
241    /* Mark this entry as fresh (see process_routes.c:512) */
242    rt->rt_nexthop.iif_index = -1;
243
244    /* set key and backpointer prior to tree insertion */
245    rt->rt_dst = *prefix;
246
247    rt->rt_tree_node.key = &rt->rt_dst;
248    avl_insert(&routingtree, &rt->rt_tree_node, AVL_DUP_NO);
249
250    /* init the originator subtree */
251    avl_init(&rt->rt_path_tree, avl_comp_default);
252
253    return rt;
254  }
```

此函数为创建一个可用的路由条目，对于提供的参数（IP 前缀）分配一个路由条目空间，初始化后插入到 avl 树里。首先是 olsr_cookie_malloc()与 memset()函数申请内存空间并清 0。之后为改入口分配入口，并把该入口的目的地址设置成为参数提供的入口地址。函数 avl_insert()讲入口的树节点插入到路由表中，在用函数 avl_init()初始化树。

File: routing_table.c

```
292  void
293  olsr_insert_rt_path(struct rt_path *rtp, struct tc_entry *tc, struct link_entry *link)
294  {
295    struct rt_entry *rt;
296    struct avl_node *node;
297
298    /*
299     * no unreachable routes please.
300     */
301    if (tc->path_cost == ROUTE_COST_BROKEN) {
302      return;
303    }
304
305    /*
306     * No bogus prefix lengths.
307     */
308    if (rtp->rtp_dst.prefix_len > olsr_cnf->maxplen) {
309      return;
310    }
311
312    /*
313     * first check if there is a route_entry for the prefix.
314     */
315    node = avl_find(&routingtree, &rtp->rtp_dst);
316
317    if (!node) {
318
319      /* no route entry yet */
320      rt = olsr_alloc_rt_entry(&rtp->rtp_dst);
321
322      if (!rt) {
323        return;
324      }
325
326    } else {
327      rt = rt_tree2rt(node);
328    }
329
330    /* Now insert the rt_path to the owning rt_entry tree */
331    rtp->rtp_originator = tc->addr;
332
333    /* set key and backpointer prior to tree insertion */
334    rtp->rtp_tree_node.key = &rtp->rtp_originator;
335
336    /* insert to the route entry originator tree */
337    avl_insert(&rt->rt_path_tree, &rtp->rtp_tree_node, AVL_DUP_NO);
338
339    /* backlink to the owning route entry */
340    rtp->rtp_rt = rt;
341
342    /* update the version field and relevant parameters */
343    olsr_update_rt_path(rtp, tc, link);
344  }
345
```

此函数为对每个 rt_path 创建一个路由入口，把它加入到全局的 RIB 树里。首先两个 if 判断参数是否满足条件，如果传入的 tc_entry 为 ROUTE_COST_BROKEN 或者传入的 rtp 的目的地址长度（rtp_dst.prefix_len）

大于所设定的最大地址长度，则判断参数不符合条件，直接返回。接着调用 avl_find()函数，检查传入的 rtp 节点是都在路由表中。如果节点在路由表中则将节点类型从 avl_node 改为 rt_nentry 类型，不在则在路由表中重新分配一个节点。最后将节点添加进 avl 树里，改变相应参数的值，更新整个路由表。

File: routing_table.c

```
349  void
350  olsr_delete_rt_path(struct rt_path *rtp)
351  {
352
353    /* remove from the originator tree */
354    if (rtp->rtp_rt) {
355      avl_delete(&rtp->rtp_rt->rt_path_tree, &rtp->rtp_tree_node);
356      rtp->rtp_rt = NULL;
357    }
358
359    /* remove from the tc prefix tree */
360    if (rtp->rtp_tc) {
361      avl_delete(&rtp->rtp_tc->prefix_tree, &rtp->rtp_prefix_tree_node);
362      olsr_unlock_tc_entry(rtp->rtp_tc);
363      rtp->rtp_tc = NULL;
364    }
365
366    /* no current inet gw if the rt_path is removed */
367    if (current_inetgw == rtp) {
368      current_inetgw = NULL;
369    }
370
371    olsr_cookie_free(rtp_mem_cookie, rtp);
372  }
373
```

此函数可以删除 rtp 树，并吧它从 TC 树中删除，改变路由表版本。首先，avl_delete()把 rto 指向的树节点从所在的树里面删除。rtp_rt 指向的树的根置空。接着再用函数 avl_delete()将 rtp 在前缀里删除，olsr_unlock_tc_entry()解锁相应的 tc_entry。最后删除 TC 树里 rtp 所指向的树节点函数 olsr_cookie_free()删除 cookie 所占用的内存。

File: routing_table.c

```
435  static bool
436  olsr_cmp_rtp(const struct rt_path *rtp1, const struct rt_path *rtp2, const struct rt_path *inetgw)
437  {
438    olsr_linkcost etx1 = rtp1->rtp_metric.cost;
439    olsr_linkcost etx2 = rtp2->rtp_metric.cost;
440    if (inetgw == rtp1)
441      etx1 *= olsr_cnf->lq_nat_thresh;
442    if (inetgw == rtp2)
443      etx2 *= olsr_cnf->lq_nat_thresh;
444
445    /* etx comes first */
446    if (etx1 < etx2) {
447      return true;
448    }
449    if (etx1 > etx2) {
450      return false;
451    }
```

```
452
453   /* hopcount is next tie breaker */
454   if (rtp1->rtp_metric.hops < rtp2->rtp_metric.hops) {
455     return true;
456   }
457   if (rtp1->rtp_metric.hops > rtp2->rtp_metric.hops) {
458     return false;
459   }
460
461   /* originator (which is guaranteed to be unique) is final tie breaker */
462   if (memcmp(&rtp1->rtp_originator, &rtp2->rtp_originator, olsr_cnf->ipsize) < 0) {
463     return true;
464   }
465
466   return false;
467 }
```

此函数为比较两个路由的路由，如果为第一个参数更好的话返回值 true。首先比较两个路由的花销，花销小的路径更优；在花销一样的情况下则比较路由跳数，跳数小的更优；跳数一样，则比较两条路由的源地址，源地址小的更优。

File: routing_table.c

```
485 void
486 olsr_rt_best(struct rt_entry *rt)
487 {
488   /* grab the first entry */
489   struct avl_node *node = avl_walk_first(&rt->rt_path_tree);
490
491   assert(node != 0);            /* should not happen */
492
493   rt->rt_best = rtp_tree2rtp(node);
494
495   /* walk all remaining originator entries */
496   while ((node = avl_walk_next(node))) {
497     struct rt_path *rtp = rtp_tree2rtp(node);
498
499     if (olsr_cmp_rtp(rtp, rt->rt_best, current_inetgw)) {
500       rt->rt_best = rtp;
501     }
502   }
503
504   if (0 == rt->rt_dst.prefix_len) {
505     current_inetgw = rt->rt_best;
506   }
507 }
```

此函数通过遍历找到最优路径，并且把当前网关路径改为最佳路径。首先是调用 avl_walk_first()函数找到树里的第一个条目，当然要保证不为 0。随后讲节点类型转化为 rt_entry。之后遍历整棵树，比较当前路径与记录的最佳路径，如果当前路径优与记录的最佳路径，则将当前路径记录为最佳路径。

```c
olsr_insert_routing_table(union olsr_ip_addr *dst, int plen,
                          union olsr_ip_addr *originator, int origin)
{
#ifdef DEBUG
  struct ipaddr_str dstbuf, origbuf;
#endif
  struct tc_entry *tc;
  struct rt_path *rtp;
  struct avl_node *node;
  struct olsr_ip_prefix prefix;

  /*
   * No bogus prefix lengths.
   */
  if (plen > olsr_cnf->maxplen) {
    return NULL;
  }

  /*
   * For all routes we use the tc_entry as an hookup point.
   * If the tc_entry is disconnected, i.e. has no edges it will not
   * be explored during SPF run.
   */
  tc = olsr_locate_tc_entry(originator);

  /*
   * first check if there is a rt_path for the prefix.
   */
  prefix.prefix = *dst;
  prefix.prefix_len = plen;

  node = avl_find(&tc->prefix_tree, &prefix);

  if (!node) {

    /* no rt_path for this prefix yet */
    rtp = olsr_alloc_rt_path(tc, &prefix, origin);

    if (!rtp) {
      return NULL;
    }
#ifdef DEBUG
    OLSR_PRINTF(1, "RIB: add prefix %s/%u from %s\n", olsr_ip_to_string(&dstbuf, dst), plen,
                olsr_ip_to_string(&origbuf, originator));
#endif

    /* overload the hna change bit for flagging a prefix change */
    changes_hna = true;

  } else {
    rtp = rtp_prefix_tree2rtp(node);
  }

  return rtp;
}
```

该函数是将一个前缀插入 TC 的前缀树。先调用 olsr_locate_tc_entry()函数判断该 tc_entry 是否可以连接，tc_entry 作为所有路由的连接点，如果它不可连接，在计算最短路径时不会被考虑。之后则是检查该前缀是否已经存在一条路径，如果没有，则调用 olsr_alloc_rt_path 函创建一条该前缀和源地址的路径。

整个路由计算过程如下：路由节点启动后，首先发现相邻节点，获取相邻节点的地址。之后路由节点则会主动开始测试到相邻节点的链路时延或成本，根据测试结果则可设置相关链路状态。这样路由器节点就可以构造出自己的链路状态信息包，内容包括了路由器的标号、路由器的邻居路由器列表、路由器到各邻居路由器的链路状态（时延或成本）、链路状态信息包的序号和生存时间等。之后则由该节点向所有参与链路状态交互的节点广播链路状态信息，包括了周期性的广播和链路状态发生变化时的广播。节点收到所有的链路状态信息后，则可以构造出整个网络的拓扑结构图 G（V，E）。V 表示路由器，E 则是链路路径。这样一来，节点就可以利用 Dijkstra 算法在图 G 中计算到所有目的地的最短路径，即构造以自己为根节点的最短路径优先树。这样一来，路由计算就完成了。

6 设计与实践

6.1动机

这次老师提供的 OLSR daemon 是当前最流行的 OLSR 实现版本，实现了 OLSR 协议的全部功能，支持多种操作系统。然而，正因为此，OLSRD 代码规模巨大，依赖关系错综复杂，不利于阅读和修改，也不利于实验。所以在设计与实践环节，我们小组准备编写实现一个 OLSR 协议，实现 OLSR 协议标准功能的一个子集，藉此巩固网络协议知识和提高协议设计能力。

6.2简化 OLSR 实现 – PyOLSR

6.2.1 总体设计

为了简化，进程间的通信通过读写文本文件模拟。
节点使用多线程技术：
1. 发送消息：主线程使用循环，利用 sleep 函数，根据时间戳周期性的发送 HELLO/TC 消息。
2. 接受消息：另有二个线程构成生产者\消费者模式。一个线程读取代表接受的文本文件，并通过 yields 封装成生成器。另一个线程调用生成器，实时接受消息。
3. 路由：路由模块定义了一些算法和更新函数，根据收到的消息更新各种结构。
4. 计时器：使用单例范式定义了一个简单的计时器。

6.2.2 具体实现

关于代码的具体实现，我们许多地方都参考了 OLSRD 中的实现。下面对 MPR 选择算法和路由计算算法的代码实现做简要展示：

6.2.2.1 MPR 选择算法递归版

类似 RFC 中定义的贪婪算法，使用一个递归函数计算 MPR，代码如下：

```
3   two_hop_set = set.union(*list(neighbor_map.values()))
4
5   def select_mpr(neighbor_map,two_hop_set):
6       if two_hop_set == set():
7           return set()
8       count = lambda x: len(x & two_hop_set)
9       convert = lambda x: (count(x[1]), x[0])
10      interset = set(map(convert, neighbor_map.items()))
11      _, mpr = max(interset)
12      two_hop_set = two_hop_set - neighbor_map[mpr]
13      neighbor_map.pop(mpr)
14      return {mpr} | select_mpr(neighbor_map, two_hop_set)
```

3 对一跳邻居表（字典）进行解包操作，然后使用集合的 union 函数生成二跳邻居表（集合），
6-7 如果二条邻居表为空，说明所有二跳邻居都已经覆盖，**递归返回**，
8 使用 lambda 表达式定义 count 函数，功能是计算参数 x 与二跳邻居表的交集个数，
9 使用 lambda 表达式定义 convert 函数，返回一个 tuple(x[1]即某 value 与二跳交集个数，x[0]某 key)，
10 使用 map 函数，对一跳邻居的每一项调用 convert，生成一个 tuple(某节点二跳邻居与二跳邻居表交集个数，该节点 ID)，
11 使用解包操作和 max 函数，选取上述交集最多的结点为 mpr 节点，
12 二跳邻居表中删除该 mpr 节点的邻居，
13 邻居表中删除该 mpr 节点，
14 递归调用，返回{mpr 节点} 与 子调用返回的集合 的交集。

经测试，该递归函数能够有效的计算出 MPR 节点集合。

6.2.2.2 路由计算函数

路由计算函数使用了迪杰斯特拉算法，代码如下：

```
1   def calc_route_table(self, topo, bidir):
2       """ 计算路由表
3       参数:
4           - topo (Set[(str,str)]: 从TC表中获得的tuple表 {(dst,last_hop)}
5           - bidir (Set(str)): 双向邻居集合
6       返回:
7           路由表 (Dict(dst:(next_hop, hops))).
8       """
9
10      route = dict()
11      for node in bidir:
12          route[node] = (node, 1)
13      h = 1
14      changed = True
15      while changed:
```

```
16          changed = False
17          for dst, last_hop in topo:
18              if (dst not in route and last_hop in route
19                      and route[last_hop][1] == h and dst != self.nid):
20                  route[dst] = (route[last_hop][0], h + 1)
21                  changed =  True
22          h += 1
23      return route
```

10 初始化 route 表，

11-12 根据二跳邻居表，初始化路由表中所有二跳邻居项为经由该邻居，跳数 1，

15-22 循环将距离目标节点 h+1 跳的路由条目添加到路由表中，

17 对于拓扑表的每一项，

18 如果，其目的地址节点不包含在路由表项中，且其最后一跳节点与目标节点之间跳数等于 h 时，

20 添加一个新的路由表项：目的地（路由表 dst）为拓扑表的 dst，下一跳地址为上述最后一跳节点，跳数设置为 h+1。

经测试，该函数能够有效的计算出路由表。

7 小结

通过八周的工作，我们最终完成了这份 OLSR 协议分析与设计文档。在该文档中，我们将 OLSRD 代码中包含的主要函数调用关系总图、数据结构、变量、函数算法做了简要的介绍与分析，并分化成了合乎逻辑的章节，希望能让读者通过阅读快速了解这个协议。

根据我们分析的部分代码，我们将 OLSR 协议分为五个主要部分：链路感知、邻居发现、MPR 处理、拓扑控制消息处理和路由计算。节点间通过周期性的交换 HELLO 消息包，完成链路感知和邻居发现；节点利用邻居信息，进行 MRP 计算；节点间通过周期性的交换 TC 消息包，计算拓扑结构。最终，节点根据拓扑结构，基于 MRP 利用路由计算算法进行路由计算。链路感知中节点通过周期性收发 HELLO 消息包，维护本地链路信息表存储与邻节点的链路信息；邻居发现中，每隔一个固定的时间，节点就会向一跳范围内的节点广播一次 HELLO 消息，用于更新拓扑消息；MPR 处理中，节点通过 MPR 选举算法计算维护 MPR 表；在拓扑控制消息处理中，节点通过 TC 广播，维护网络拓扑信息；在路由计算中，节点根据 TC 消息中包含的全网拓扑信息，计算、维护各种路由表项。

然后由于该代码规模大，RFC 文档篇幅长，我们也遗憾的未能将代码协议的方方面面都理解和阐述。通过阅读 RFC 文档，我们除了获取了 OLSR 协议的方方面面的知识，也知道了一个协议制订时所需要考虑的各种问题。而通过分析一个大规模的项目，我们一方面对 OLSR 协议有了更深入的理解，另一方面也获得了大量项目编写、网络编程、C 语言的知识与小技巧。

在我们的设计板块中，我们设想自己设计实现一个简化版的 OLSR 路由协议。但由于时间问题，项目中并没有完全完成，还有许多有待解决的问题。并且由于相关网络编程技术不够熟练，我们也没有真正实现网络交互而是用文件读写来模拟。在课程结束后，我们小组也会继续学习网络、协议知识，继续完成、维护这个项目。

感谢覃老师和助教们这八周的精彩讲解和耐心指导！这段时间的学习让不仅让我们对网络协议分析设计有了深入的理解，还锻炼了我们的团队协作能力。另外通过阅读没有人翻译的原版 RFC 文档，我们的英语阅读水平也有了不小的提高。