优化链路状态路由协议代码分析

《网络协议栈分析与设计》课程大作业

学号	姓名	班级	负责模块	成绩
201771050	陈倩	软网 1703	1. 引言 2. 代码介绍 3. 数据结构 4. RFC 文档阅读 5. 小结 6. 排版	
201792423	马嘉笛	软网 1703	1. MPR 选举 2. 路由计算	
201792112	杜锦滢	软网 1701	1. 链路感知 2. 邻居发现 3. 拓扑消息处理	

目录

1 引言

最佳链路状态路由协议(Optimized Link State Routing Protocol，OLSR)是为解决经典链路状态路由协议（Link State，LS）泛洪（Flooding）问题优化而来的，在无线移动自组网络（Mobile Ad Hoc Network，MANET）中使用 LS 会由于重复转发数据包造成许多不必要的电池损耗，因此，OLSR 是专门为 MANET 提出来的一种标准化的先验式的优化链路状态路由协议。

1.1 OLSR 标准术语

节点（node）：MANET 路由器，可实现 OLSR 协议；
OLSR 接口（OLSR interface）：参与运行 OLSR 的 MANET 网络设备。一个节点可能具有多个 OLSR 接口，每个接口分配了唯一的 IP 地址；
单 OLSR 接口节点（single OLSR interface node）:只具有一个 OLSR 接口的节点；
多 OLSR 接口节点（multiple OLSR interface node）:具有多个 OLSR 接口的节点；
主地址（main address）:节点的主地址，用作该节点发出的所有消息的“发起者地址”，它是某一个 OLSR 接口的地址。单 OLSR 接口节点的主地址必须使用其唯一的 OLSR 接口的地址，多 OLSR 接口节点选取其中一个 OLSR 接口的地址作为主地址，节点始终使用与其主地址相同的地址。
邻居节点（neighbor node）: 如果节点 X 能侦听到节点 Y，那节点 Y 就是节点 X 的邻居节点；
两跳邻居节点（2-hop neighbor）:能被邻居节点侦听到的节点为该节点的两跳邻居节点；
严格两跳邻居节点（strict 2-hop neighbor）:在两跳邻居节点中，排除掉节点本身和节点的邻居，剩余的集合为该节点的严格两跳邻居节点；
多点中继节点（Multipoint relay, MPR）：由节点的邻居节点 X 选择的节点，以转发它从节点 X 接受到的所有不重复的广播消息。
多点中继节点选择器（Multipoint relay selector, MS）:选择了其一跳邻居节点 Y 作为 MPR 的节点 X 被称为 Y 的 MS；
链路（link）：连接是一对邻居节点能够相互侦听到对方，当一个节点的一个接口具有到另一个节点的接口之一的链接时，该节点被称为具有到另一节点的链接；
对称链路（symmetric link）:在两个 OLSR 接口间的双向链路；
非对称链路（asymmetric link）：在两个 OLSR 接口间的单向链路；

1.2 OLSR 简介

OLSR 是专为 MANET 开发的，它是表驱动的主动协议，即与网络中的其他节点定期交换拓扑信息。每个节点选择一组其邻居节点作为 MPR，只有被选作 MPR 的节点才负责转发数据包，直到数据包扩散到整个网络。MPR 通过减少所需的传输次数，提供了一种用于泛洪控制流量的有效机制。

当被选为 MPR 的节点周期性的声明链路状态信息时，它们还有一个特别的任务。实际上，对于使用 OLSR 提供通往全部目的地的最短路径路由而言，唯一的需求就是 MPR 节点为它们的 MS 声明链路状态信息。冗余的可用的链路信息也可能被使用。

OLSR 网络中一个节点会周期性的声明它可以连接哪些 MS 节点，在路由计算中，MPR 被用来构成从已知节点到网络中任意目的地的路径。

一个节点从它的一跳邻居节点中选择具有对称链路的节点作为 MPR，因此经由 MPR 选择路径，能够自动避免与单向链路中传输数据包相关的问题。OLSR 能够独立于其他协议进行工作，它对下层链路不

作任何假设。OLSR 中的数据传输全部使用 UDP 协议，所以由于冲突产生的丢包在所难免，OLSR 的端口为 698.

OLSR 以路由跳数提供最优路径，因此，OLSR 尤其适合大而密集型网络。

1.3 OLSR 工作原理

OLSR 网络中主要有三种控制消息分组：HELLO 消息分组、TC（Topology Control）消息分组和 MID（Multiple Interface Declaration）消息。

每个节点周期性的向一跳邻居节点发送 HELLO 消息，用来侦听邻居节点状态；在初始化阶段，节点 B 对外广播 HELLO 消息，当节点 B 收到这个消息分组时，将 B 放进自己的邻居节点集中，并标记到 B 链路状态为非对称的，然后，A 对广播 HELLO 分组，在该 HELLO 分组中包含 B 是 A 的非对称邻居节点这个消息，当 B 收到这个分组时，会在邻居节点集中将 A 的状态更新为对称，当 B 再次广播 HELLO 分组时，HELLO 分组中就包含了 A 是 B 的对称邻居节点这个信息，当 A 再收到这个 HELLO 分组时，B 的状态被 A 更新为对称。

侦听到邻居的信息后，节点分别选择自己的 MPR，后面的 HELLO 消息中便包含了各自的 MPR 信息。

每个节点根据自己收到的 TC 分组消息和自己的邻居节点消息来计算出网络的拓扑。为了使得全网的节点建立起一张完整的网络拓扑。OLSR 采用 MPR 机制对数据包进行选择性的泛洪，可以有效的减少整个网络内的数据包数量。

通过消息的选择性泛洪，每个节点都拥有一张路由表。通过路由表寻找路径消息。任何一个网络变化都会导致路由表的更新。另外，计算路径采用 Dijkstra 最短生成树算法，跳数，链路带宽，时延，队列长度等都可以作为路径长度的判据。

OLSR 不要求控制消息按顺序收发，因为每个控制消息都带有自己的序列号，每发一个消息，序列号增加。接收到分组的节点通过判断序列号的大小就可以知道该控制消息的新旧程度。

1.4 MPR 机制

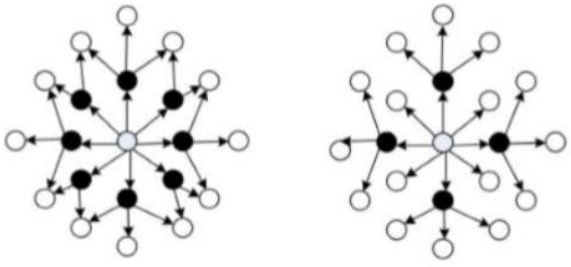
MPR 机制是 OLSR 协议中的核心机制，目的是通过减少同一区域内相同控制分组的重复转发次数来减少网络中广播分组的数量。网络中某一节点 X 的邻居被分为两类：MPR 节点和非 MPR 节点。每个节点从其一跳邻居中选择自己的 MPR 集，使得该节点通过这个集合转发的 TC 分组能够覆盖该节点所有的两跳邻居节点。

节点 X 的 MPR(X)节点必须满足以下条件：（1）MPR(X)必须是 X 的一跳邻居节点子集；（2）X 的每个两跳邻居节点都必须有到达 MPR(X)的对称链路；（3）X 与 MPR(X)之间的也必须有对称链路。

MPR 集越小，协议性能越好，节点 X 发送的广播消息通过 MPR(X)转发到其他节点，而非 MPR 节点只处理消息而不转发。

协议根据 MPR 集来计算路由。OLSR 网络中每个 MPR 节点周期性的广播它的 MS 集，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 机制选择性泛洪的 OLSR 有效的减少了网络内控制分组重复转发的数量，避免了广播风暴。



泛洪与选择性泛洪

2 代码介绍

2.1 文件介绍

在 OLSR 协议中，在/src 中共有 150 个源文件，我们挑选了一些重要的源文件进行介绍。

文件	描述
Address.c	IP 数据包表征功能
Avl.h	定义了 AVL 树
Avl.c	封装了对 AVL 树的操作
Bmf.c	组播转发功能
Duplicate_set.c	定义了分组重复信息表和一些相关处理
Gateway.c	网关初始化和配置
Generate_msg.c	HELLO,TC,MID,HNA 消息的生成函数
Hysteresis.c	实现 Hysteresis 技术
Ifnet.c	接口网络的操作
Interface.c	接口实现
Ipcalc.c	IP 前缀的转换
Kernel_routes.c	增删路由
Link_set.h	定义了存储节点自身信息的结构体
Link_set.c	确定邻居表的信息
List.c	链表操作
Main.c	程序初始化
Mpr.c	封装了一些对 MPR 的操作
Mpr_selector_set.h	定义了结构体 MS
Neighbor_table.h	对邻居信息数据结构的定义
Neighbor_table.c	对一跳邻居表的处理
Net.c	不同内核的基本网络操作
Olsr.c	实现了一些全局函数
Olsr_conf.c	配置文件操作
Olsr_cfg.h	定义了一些常量
Olsr_spf.c	Dijkstra 最小生成树算法计算路由表
Packet.c	各种基本数据包发送和释放
Parser.c	分析器操作
Process_routes.c	封装了对核心路由表的处理
Routing_table.c	封装了对路由表的操作
Scheduler.c	封装了对定时器的处理
TC_set.c	TC 消息的泛洪
Timer.h	基本的计时器计算
Two_hop_neighbor_table.c	对二跳邻居表的处理

表 2.1.1 OLSR 中部分源文件介绍

2.2 全局变量

同样，这里只列出一些重要的全局变量

全局变量	数据类型	描述
Olsrport	Nit16_t	OLSR 消息发送、接收的端口号
Rt_proto	Nit8_t	路由表计算所遵循的协议
Willingness	Nit8_t	WILL_ALWAYS 的邻居节点集合
Min_tc_vtime	Float	TC 消息 vtime 的最小取值
Max_tc_vtime	Float	TC 消息 vtime 的最大取值
Changes_topology	Bool	判断拓扑信息是否变化
Changes_neighborhood	bool	判断邻居信息是否变化

表 2.2.1 OLSR 中部分全局变量介绍

2.3 配置变量

OLSR 路由协议中的任何具体实现都必须支持下列配置变量，并且必须支持通过系统管理可以修改这些配置变量取值.

名称	描述	默认值
DEF_IP_VERSION	缺省 IP 协议域	AF_INET
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 消息最小 vtime 取值	0.0
DEF_GW_TYPE	缺省网关类型	GW_UOLINK_IPV46
DEF_DOWNLINK_SPEED	缺省链路下载速度	1024kb/s
DEF_PROTOCOL	缺省的路由协议	0
MAX_TTL	TTL 最大值	0.165ms
COOKIE_ID_MAX	Cookie 数量最大值	25
MAXMESSAGESIZE	广播数据包的最大值	1500kb
OLSR_LINK_JITTER	OLSR 消息抖动的时间	5s
OLSR_LINK_HELLO_JITTER	HELLO 消息抖动的时间	0s
OLSR_LINK_SYM_JITTER	系统抖动时间	0s
OLSR_LINK_LOSS_JITTER	数据包丢失抖动的时间	0s

表 2.3.1 OLSR 中部分配置变量

2.4 系统量

统计量名	说明
Spam_orig_counter	无效的 orig 分组数量
Spam_hna_counter	无效的 hna 分组数量
Spam_mid_counter	无效的 mid 分组数量
Seq	路由信息序列号

表 2.4.1 OLSR 统计量

2.5 OLSR 协议框图

为了对复杂的 OLSR 协议有一个整体上的认识，我们从 main.c 函数出发，从函数调用层面对整个协议画了一个 OLSR 协议框图。

OLSR 采用轮询机制，首先调用一系列初始化函数对协议需要用到的数据结构进行初始化。然后，调用 olsr_scheduler 函数进入调度循环。在调度循环中，主程序可以调用 poll_sokets 函数接受消息，也可以调用 walk_time 函数产生消息。Poll_sokets 函数进行消息接收时，轮询 sokets，若发现可以识别的 soket，就调用相应的函数处理，处理完成之后，调用 olsr_process_changes 更新路由表或者重新计算 MPR；walk_time 函数根据定时器随机产生调用指定的函数产生消息，generate_hello, generate_tc, generate_mid 分别产生 HELLO, TC 和 MID 消息，产生消息之后，调用 net_output 函数将消息封装为 OLSR 数据包发送出去。

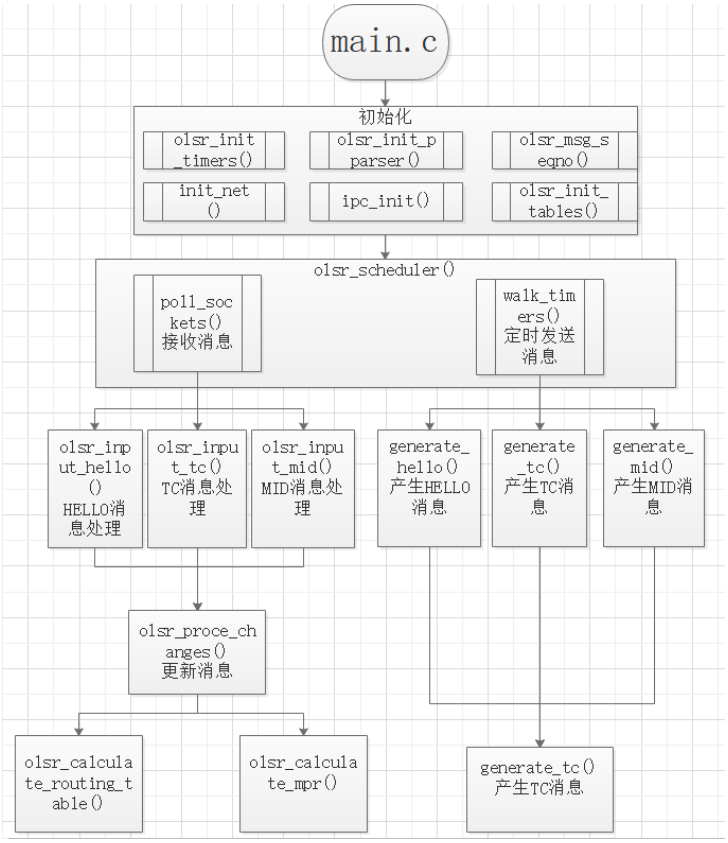


图 2.5.1 olsr 协议框图

3 数据结构

3.1 链路存储器

链路存储器里面存储着各个链路的信息，以下是 olsrd-0.6.0 中对链路存储器的定义：

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;
69:
70:   /*
71:    * Hysteresis
72:    */
73:   float L_link_quality;
74:   int L_link_pending;
75:   uint32_t L_LOST_LINK_time;
76:   struct timer_entry *link_hello_timer; /* When we should receive a new HELLO */
77:   olsr_reltime last_htime;
78:   bool olsr_seqno_valid;
79:   uint16_t olsr_seqno;
80:
81:   /*
82:    * packet loss
83:    */
84:   olsr_reltime loss_helloint;
85:   struct timer_entry *link_loss_timer;
86:
87:   /* user defined multiplies for link quality, multiplied with 65536 */
88:   uint32_t loss_link_multiplier;
89:
90:   /* cost of this link */
91:   olsr_linkcost linkcost;
92:
93:   struct list_node link_list;          /* double linked list of all link entries */
94:   uint32_t linkquality[0];
95: };
```

Link_set.h

58 local_iface_addr：本地接口的地址，
59 neighbor_iface_addr：邻居接口的地址，
61 inter：本地接口的信息，
62 if_name:：接口名称，
63-66：计时器，
67 neighbor：邻居节点的信息，
68 prev_status：原来的链路状态，
73-79：链路质量相关参数，
84 loss_helloint：丢失的 hello 包数量，
91 linkcost：链路花销。

3.2 链路存储器

邻居节点以双向链表的方式进行存储，除了存储着自己本身的主地址、状态、是否为 MPR 等信息，还存储着经由它可以连接的两跳邻居的信息。

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: };
```

neighbor_table.h

59 neighbor_main_addr：邻居的主地址，
60 status：邻居的状态（SYM_LINK,ASYM_LINK,MPR_LINK）
61 willingness：邻居愿意为本节点转发数据包的意愿值，
62 is_mpr：是否为 MPR 节点，
63 was_mpr：之前是否是 MPR 节点，检测 MPR 节点是否改变，
65 neighbor_2_nocov：没有覆盖的两跳邻居节点的个数。

3.3 TC 存储器

TC 存储器中记录着拓扑信息，以 avl 树的形式保存节点信息，以链路开销作为权重，然后用 DIJKSTRA 最短路径算法计算出最短路径。此外 olsrd-0.6.0 中还定义了其他用于记录拓扑状态的必要统计量。

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: };
```

tc_set.h

3.4 路由存储器

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: };
```

routing_table.h

86 rt_dst：以 IP 前缀的方式记录路由目的地，

88 rt_best：最短路径，

89 rt_nexthop：下一跳节点。

3.5 OLSR 首部

OLSR 使用统一的数据包格式，使用 UDP 通信，数据包嵌入到 UDP 数据报。每个数据包封装一个或多个消息，邮件分享通用报头格式，使节点能够正确接受或者重传一个未知类型的消息。每一个消息都会分配一个唯一的标识号，用来确保消息不会被重传。

OLSR 对数据包使统一的格式，这样做的目的时促进协议的扩展性且不损害其反向兼容型。每个分组封装一个或多个消息，这些消息共用一个 OLSR 首部。

OLSR 协议分组的首部如图 3.1.1 所示：

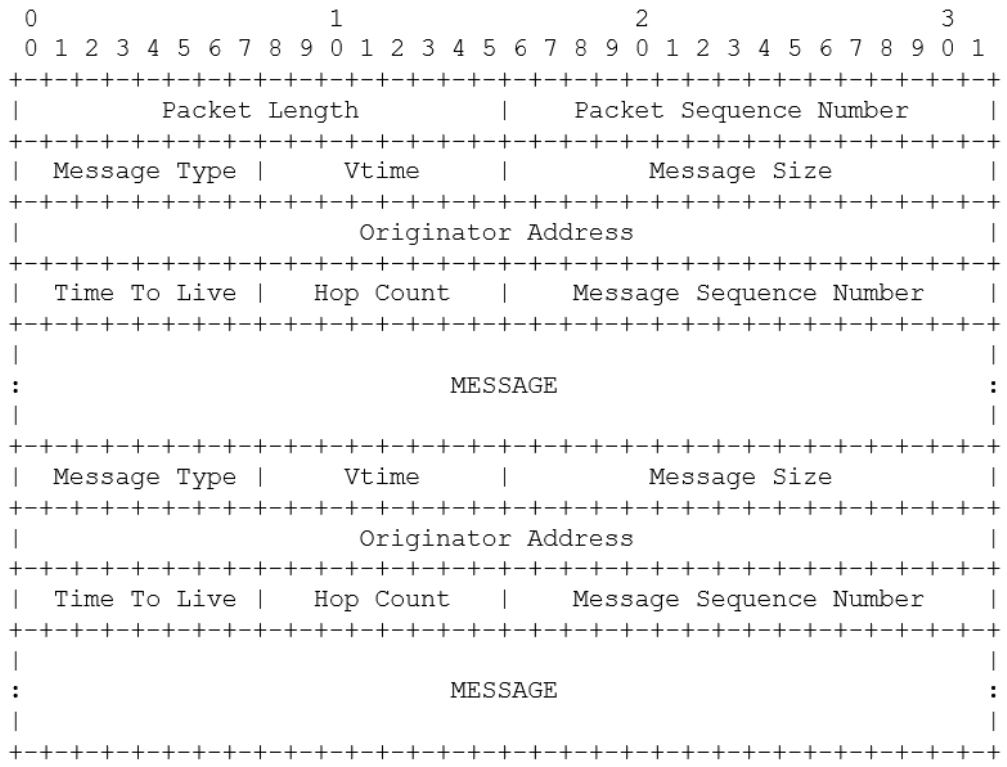


图 3.1.1 OLSR 分组首部

Packet Length:分组长度。

Packet Sequence Number: 分组序列号，当一个 OLSR 分组被发送时，序列号加一，因此分组不必

被有序的传送。

Message Type: 表示消息类型，范围在 0-127 之间。

Vtime: 分组中所携带的消息的有效期。

Message Size: 消息长度。

Originator Address: 产生该消息节点的主地址。

Time to live: 消息被传送的最大跳数。

Hop Count:一个消息已被传送的跳数。

Message Sequence Number: 消息序列号，产生一个新的消息时，序列号加一。消息序列号用来保证一个消息不会被任何节点重传两次。

下面列出 olsrd-0.6.0 中对 OLSR 首部的定义：

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: };
```

Lq_packet.h

56 type: 消息类型，

57 vtime: 分组所携带的消息的有效期，

58 size: 消息长度，

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，

61 hops: 消息已被传送的跳数，

62 seqno: 消息标识号，由节点维护。

在传输时，这些分组被嵌入 UDP 中，最后加上 IP 报头用来在网络中传输。

3.6 HELLO 消息

HELLO 消息用来建立一个节点的邻居表，包括邻居节点的地址以及本节点到邻居节点的延时和开销，OLSR 采用周期性的广播 HELLO 分组来侦听邻居节点的状态。HELLO 消息只在一跳的范围内广播，不能被转发。

OLSR 只关心对称链路，HELLO 消息建立对称链路的原理：在初始化阶段，节点 B 对外广播 HELLO 消息，当节点 B 收到这个消息分组时，将 B 放进自己的邻居节点集中，并标记到 B 链路状态为非对称的，然后，A 对广播 HELLO 分组，在该 HELLO 分组中包含 B 是 A 的非对称邻居节点这个消息，当 B 收到这个分组时，会在邻居节点集中将 A 的状态更新为对称，当 B 再次广播 HELLO 分组时，HELLO 分组中就包含了 A 是 B 的对称邻居节点这个信息，当 A 再收到这个 HELLO 分组时，B 的状态被 A 更新为对称。

HELLO 消息分组格式如下：

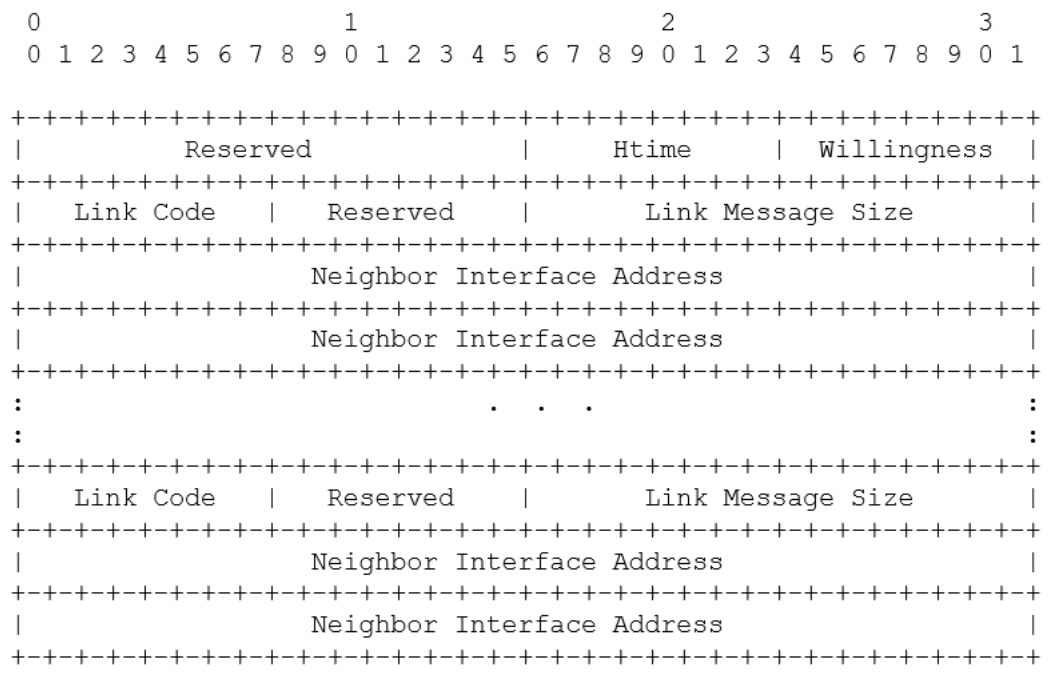


图 3.2.1 HELLO 分组格式

Reserved: 保留字段，设为“000000000000”。

Htime: 描述此接口的 HELLO 消息发送时间间隔。

Willingness: 描述一个节点为其他节点携带网络流量的意愿。

Link Code: Link Code 包含 Link Type 和 Neighbor Type.链路类型有四种：UNSPEC_LINK 即没有提供有关链路类型特定信息的链路，ASYM_LINK 即非对称链路，SYM_LINK 即对称链路，LOST_LINK 即丢失的链路；邻居类型有三种：SYM_NEIGH 即对称邻居，MPR_NEIGH 即被该节点选为 MPR 节点的邻居，NOT_NEIGH 即节点已经不存在或者尚未成为对称邻居。

Link Message Size: 本链路消息的大小，从 Link Code 字段开始一直到下一个 Link Code 字段之前（若无下一个 Link Code 字段，则直到分组结尾）。

Neighbor Interface Address: 邻居地址列表，每一种 Link Code 之后都紧随一个邻居列表地址，表示发送该 HELLO 分组的节点到这个邻居列表中的所有节点的链路类型都是相同的。

Lq_packet.h

```
107 struct lq_hello_info_header {
108:    uint8_t link_code;
109:    uint8_t reserved;
110:    uint16_t size;
111: };
112:
113 struct lq_hello_header {
114:    uint16_t reserved;
115:    uint8_t htime;
116:    uint8_t will;
117: };
```

Lq_packet.h

结构体 lq_hello_info_header 和 lq_hello_header 共同组成了 HELLO 消息数据包的头部。

3.7 MID 消息

MID 消息的主要功能是实现协议中多重 OLSR 接口之间的通信，结构如下：

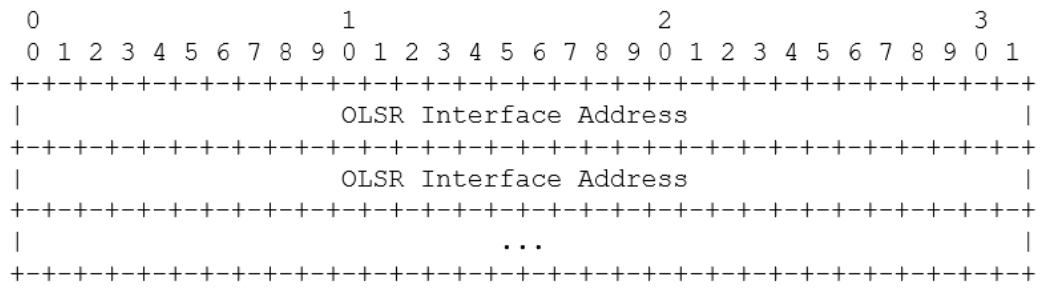


图 3-3 MID 分组格式

为了将消息扩散到整个网络，MID 消息的 TTL 被设置为 255（最大值），MID 的 vtime 根据 MID_HOLD_TIME 来设置。

Olsr-0.6.0 中使用一个双向链表存储节点接口，mid_set[idx].next 表示下一跳节点的接口，mid_set[idx].prev 表示上一跳节点接口，Olsrd-0.6.0 中对 MID 消息的定义：

mid_set.c

```
66: int
67: olsr_init_mid_set(void)
68: {
69:   int idx;
70:
71:   OLSR_PRINTF(5, "MID: init\n");
72:
73:   for (idx = 0; idx < HASHSIZE; idx++) {
74:     mid_set[idx].next = &mid_set[idx];
75:     mid_set[idx].prev = &mid_set[idx];
76:
77:     reverse_mid_set[idx].next = &reverse_mid_set[idx];
78:     reverse_mid_set[idx].prev = &reverse_mid_set[idx];
79:   }
80:
81:   return 1;
82: }
```

mid_set.c

3.8 TC 消息

TC(Topology Control)消息对于路由选择具有十分重要的意义，在 OLSR 协议网络中，每个节点周期性的发送 TC 分组。当一个节点接收到 TC 消息时，处理它，如果　这个节点是发送这个 TC 消息节点的 MPR 节点，则判断该 TC 消息是否最新，如果是，转发它，否则，丢弃它。当这个 TC 消息中包含一条有效的链路，则把这条链路加到节点自身所维护的拓扑表中，用以计算路由（拓扑信息维护）；当这个 TC 消息中，有链路发生变化时，更改自身所维护的拓扑表并重新计算路由（路由建立与维护）。

下图为 TC 消息分组格式：

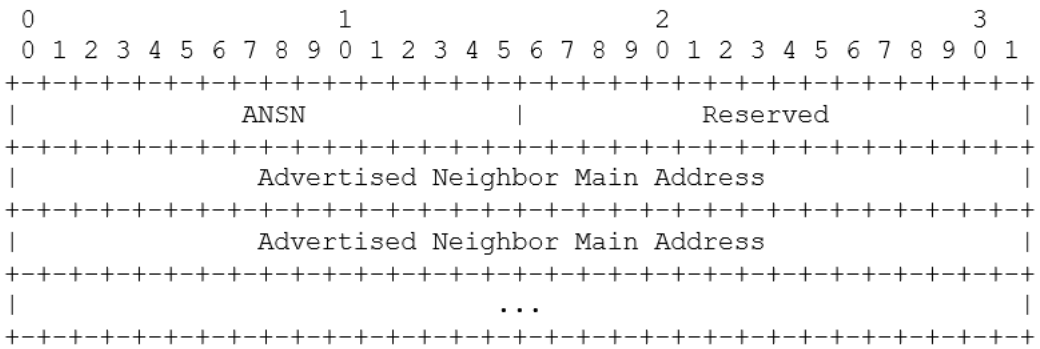


图 3-4 TC 消息分组格式

TC 消息的 OLSR 报头中的 Message Type 应该被设为 TC_MESSAGE，TTL 被设为 255（最大值），以便消息被扩散到整个网络中去，Vtime 应该被设为 TOP_HOLD_TIME。

ANSN：Advertised Neighbor Sequence Number，当节点在它的邻居集中检测到更改时，它会增加此序列号。ANSN 用来跟踪最新消息，当节点收到一个 TC 消息时，它基于 ANSN 来判断这个消息是否是最新的 TC 消息。

Reserved：保留字段，设置为“00000000000000000000000000”。

Advertised Neighbor Main Address：保存邻居节点的主地址。

以下为在 olsrd-0.6.0 中对 TC 消息的定义：

packet.h

```
77: struct tc_message {
78:   olsr_reltime vtime;
79:   union olsr_ip_addr source_addr;
80:   union olsr_ip_addr originator;
81:   uint16_t packet_seq_number;
82:   uint8_t hop_count;
83:   uint8_t ttl;
84:   uint16_t ansn;
85:   struct tc_mpr_addr *multipoint_relay_selector_address;
86: };
```

packet.h

tc_mesasge 是 TC 消息数据包格式，

78 vitime：被设为 TOP_HOLD_TIME，

79 source_addr：上一跳地址，

80 originator：产生该 TC 消息的节点主地址，

81 packet_seq_number：数据包序号，用于接收方排序重组，

83 ttl：time to live，被设为 255，

84 ansn：该 TC 消息的 ANSN 序号，用来判断是否最新，

85 multipoint_relay_Selector_address：MS 节点地址。

3.9 HNA 消息

HNA(Host and Network Association)将 OLSR 的移动自组网与没有接口的 OLSR 自组网相连。下图为 HNA 消息的分组格式：

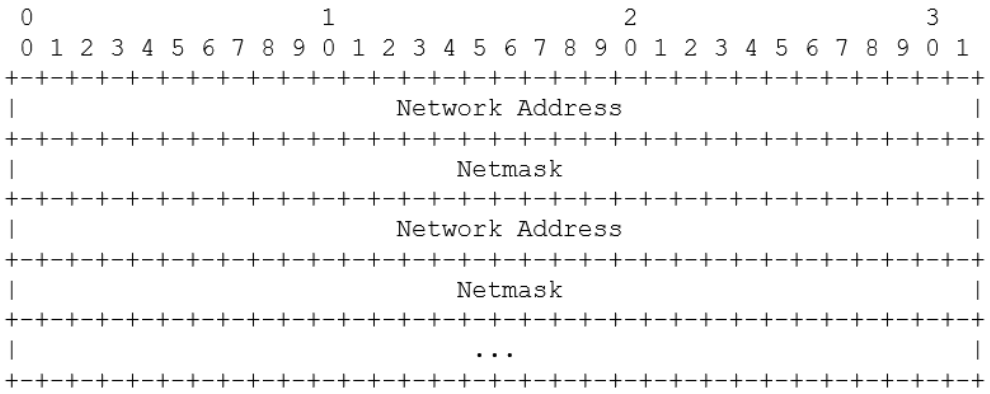


图 3-5 HNA 消息分组格式

Network Address 是移动自组网的网络地址，Netmask 是该地址的子网掩码。HNA 消息从一定程度上来说是 TC 消息的“通用版本”，HNA 消息和 TC 消息的发起者（originator）都宣布对其他某些主机的“可达性”。在 TC 消息中，不需要子网掩码，因为所有可达性都是按主机宣布的，在 HNA 消息中，通过网络地址和子网掩码来声明地址序列的可达性。TC 消息和 HNA 消息之间的重要区别是：TC 消息可以

根据 ANSN 来对先前的信息具有取消作用，而 HNA 消息中的信息仅在到期时才被删除。

在转发 HNA 消息时，上图作为正常封包的数据部分，该包的报头数据类型为 HNA_MESSAGE，TTL 设为 225，Vtime 设为 HNA_HOLD_TIME.

4 OLSR 路由算法

4.1 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。

4.1.1 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。

Link_set.c

```
686: struct link_entry *
687: update_link_entry(const union olsr_ip_addr *local, const union olsr_ip_addr *remote,
688:                   const struct hello_message *message,const struct interface *in_if)
689: {
690:   struct link_entry *entry;
691:
692:   /* Add if not registered */
693:   entry = add_link_entry(local, remote, &message->source_addr, message->vtime, message->htime, in_if);
694:
695:   /* Update ASYM_time */
696:   entry->vtime = message->vtime;
697:   entry->ASYM_time = GET_TIMESTAMP(message->vtime);
698:
699:   entry->prev_status = check_link_status(message, in_if);
700:
701:   switch (entry->prev_status) {
702:   case (LOST_LINK):
703:     olsr_stop_timer(entry->link_sym_timer);
704:     entry->link_sym_timer = NULL;
705:     break;
706:   case (SYM_LINK):
707:   case (ASYM_LINK):
708:
708:
709:     /* L_SYM_time = current time + validity time */
710:     olsr_set_timer(&entry->link_sym_timer, message->vtime, OLSR_LINK_SYM_JITTER, OLSR_TIMER_ONESHOT,
711:                    &olsr_expire_link_sym_timer,entry, 0);
712:     /* L_time = L_SYM_time + NEIGHB_HOLD_TIME */
713:     olsr_set_link_timer(entry, message->vtime + NEIGHB_HOLD_TIME * MSEC_PER_SEC);
714:     break;
715:   default:;
716:   }
717:
718:   /* L_time = max(L_time, L_ASYM_time) */
719:   if (entry->link_timer && (entry->link_timer->timer_clock < entry->ASYM_time)) {
720:     olsr_set_link_timer(entry, TIME_DUE(entry->ASYM_time));
721:   }
722:
723:   /* Update hysteresis values */
724:   if (olsr_cnf->use_hysteresis)
725:     olsr_process_hysteresis(entry);
726:
727:   /* Update neighbor */
728:   update_neighbor_status(entry->neighbor, get_neighbor_status(remote));
729:
730:   return entry;
731: }
```

Link_set.c

本函数根据获取的 hello数据包对于链路节点信息进行更新。
693-698：若该链路信息尚未注册，则添加该节点，并更新时间戳
700-722：检查链路状态，若该链路已经损坏则停止对称链路计时器；否则启动计时器
728：获取并更新数据包源端的邻居状态

4.1.2 其他链路操作

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: }
```

Link_set.c

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

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: }
```

link_set.c

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大于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

4.2 邻居发现

在OLSR协议中，各个节点周期性地向它的一跳节点发送 hello数据包，这些数据包记录着它和它的邻居节点的信息，使得收到 hello数据包的节点不但能更新源端节点的信息以及链路状态信息，还能获得两跳邻居节点的信息。

Process_packege.c

```
405: void
406: olsr_hello_tap(struct hello_message *message, struct interface *in_if, const union olsr_ip_addr *from_addr)
407: {
408:   struct neighbor_entry *neighbor;
409:
410:   /*
411:    * Update link status
412:    */
413:   struct link_entry *lnk = update_link_entry(&in_if->ip_addr, from_addr, message, in_if);
```

```
492:   /* Don't register neighbors of neighbors that announces WILL_NEVER */
493:   if (neighbor->willingness != WILL_NEVER)
494:     process_message_neighbors(neighbor, message);
495:
496:   /* Process changes immedeatly in case of MPR updates */
497:   olsr_process_changes();
498:
499:   olsr_free_hello_packet(message);
500:
501:   return;
502: }
```

Process_packege.c

本函数处理到来的 hello数据包并根据接收到的数据包进行各项信息的更新和维护。

413：调用 update_link_entry函数对链路信息进行更新

493-494：调用 process_message_neighbors根据 hello小心对邻居节点的信息进行更新和维护

Link_set.c

```
523: static struct link_entry *
524: add_link_entry(const union olsr_ip_addr *local, const union olsr_ip_addr *remote,
525:     const union olsr_ip_addr *remote_main,olsr_reltime vtime, olsr_reltime htime, const struct interface *local_if)
526: {
527:   struct link_entry *new_link;
528:   struct neighbor_entry *neighbor;
529:   struct link_entry *tmp_link_set;
530:
531:   tmp_link_set = lookup_link_entry(remote, remote_main, local_if);
532:   if (tmp_link_set) {
533:     return tmp_link_set;
534:   }
549:   new_link = olsr_malloc_link_entry("new link entry");
550:
551:   /* copy if_name, if it is defined */
552:   if (local_if->int_name) {
553:     size_t name_size = strlen(local_if->int_name) + 1;
554:     new_link->if_name = olsr_malloc(name_size, "target of if_name in new link entry");
555:     strscpy(new_link->if_name, local_if->int_name, name_size);
556:   } else
557:     new_link->if_name = NULL;
558:
559:   /* shortcut to interface. XXX refcount */
560:   new_link->inter = local_if;
561:
562:   /*
563:    * L_local_iface_addr = Address of the interface
564:    * which received the HELLO message
565:    */
566:   new_link->local_iface_addr = *local;
567:
568:   /* L_neighbor_iface_addr = Source Address */
569:   new_link->neighbor_iface_addr = *remote;
570:
571:   /* L_time = current time + validity time */
572:   olsr_set_link_timer(new_link, vtime);
573:
574:   new_link->prev_status = ASYM_LINK;
600:   list_add_before(&link_entry_head, &new_link->link_list);
601:
602:   /*
603:    * Create the neighbor entry
604:    */
605:
606:   /* Neighbor MUST exist! */
607:   neighbor = olsr_lookup_neighbor_table(remote_main);
608:   if (!neighbor) {
609: #ifdef DEBUG
610:     struct ipaddr_str buf;
611:     OLSR_PRINTF(3, "ADDING NEW NEIGHBOR ENTRY %s FROM LINK SET\n", olsr_ip_to_string(&buf, remote_main));
612: #endif
613:     neighbor = olsr_insert_neighbor_table(remote_main);
614:   }
615:
616:   neighbor->linkcount++;
617:   new_link->neighbor = neighbor;
618:
619:   return new_link;
620: }
```

Link_set.c

本函数检查一个链路实体是否存在，若不存在则新建一个实体并加入存储链表。

531-534：先查找该链路是否已经存在，若是则直接返回

549-574：新建一个链路实体并给它的各个属性赋值

600：将新建的链路加入到保存链路实体的链表中

607-618：为确保链路的另一端存在邻居的信息，先查找该邻居，若未能找到则新建一个，并将这个邻居加入到邻居表中，同时更新该链路的信息。

Neighbor_table.c

```
300: update_neighbor_status(struct neighbor_entry *entry, int lnk)
301: {
302:   /*
303:    * Update neighbor entry
304:    */
305: 
306:   if (lnk == SYM_LINK) {
307:     /* N_status is set to SYM */
308:     if (entry->status == NOT_SYM) {
309:       struct neighbor_2_entry *two_hop_neighbor;
310: 
311:       /* Delete posible 2 hop entry on this neighbor */
312:       if ((two_hop_neighbor = olsr_lookup_two_hop_neighbor_table(&entry->neighbor_main_addr)) != NULL) {
313:         olsr_delete_two_hop_neighbor_table(two_hop_neighbor);
314:       }
315: 
316:       changes_neighborhood = true;
317:       changes_topology = true;
318:       if (olsr_cnf->tc_redundancy > 1)
319:         signal_link_changes(true);
320:     }
321:     entry->status = SYM;
322:   } else {
323:     if (entry->status == SYM) {
324:       changes_neighborhood = true;
325:       changes_topology = true;
326:       if (olsr_cnf->tc_redundancy > 1)
327:         signal_link_changes(true);
328:     }
329:     /* else N_status is set to NOT_SYM */
330:     entry->status = NOT_SYM;
331:     /* remove neighbor from routing list */
332:   }
```

Neighbor_table.c

本函数对邻居节点的状态进行更新和维护。

306-320：如果被更新的链路是对称链路且该链路原本的状态是非对称状态， 则修改其状态为对称链路，并且从 2跳邻居表中搜索该地址，如果它被划归为 2 跳邻居，就把它从 2跳邻居表中删除。进行修改操作之后将记录修改的全局变量设为true。

322-332：如果被更新的链路是非对称链路而原本状态是对称状态，也要进行修改，并将记录修改的全局变量设为 true.

Process_packege.c

```
73: static void
74: process_message_neighbors(struct neighbor_entry *neighbor, const struct hello_message *message)
75: {
76:   struct hello_neighbor *message_neighbors;
77:
78:   for (message_neighbors=message->neighbors; message_neighbors!= NULL; message_neighbors=message_neighbors->next) {
79:     union olsr_ip_addr *neigh_addr;
80:     struct neighbor_2_entry *two_hop_neighbor;
81:
82:     /*
83:      *check all interfaces
84:      *so that we don't add ourselves to the
85:      *2 hop list
86:      *IMPORTANT!
87:      */
88:     if (if_ifwithaddr(&message_neighbors->address) != NULL)
89:       continue;
90:
91:     /* Get the main address */
92:     neigh_addr = mid_lookup_main_addr(&message_neighbors->address);
93:
94:     if (neigh_addr != NULL) {
95:       message_neighbors->address = *neigh_addr;
96:     }
97:
98:     if (((message_neighbors->status == SYM_NEIGH) || (message_neighbors->status == MPR_NEIGH))) {
99:       struct neighbor_2_list_entry *two_hop_neighbor_yet=olsr_lookup_my_neighbors(neighbor,
100:         &message_neighbors->address);
101:       if (two_hop_neighbor_yet != NULL) {
102:         /* Updating the holding time for this neighbor */
103:         olsr_set_timer(&two_hop_neighbor_yet->nbr2_list_timer, message->vtime, OLSR_NBR2_LIST_JITTER,
104:         OLSR_TIMER_ONESHOT,&olsr_expire_nbr2_list, two_hop_neighbor_yet, 0);
105:         two_hop_neighbor = two_hop_neighbor_yet->neighbor_2;
133:       } else {
134:         two_hop_neighbor = olsr_lookup_two_hop_neighbor_table(&message_neighbors->address);
135:         if (two_hop_neighbor == NULL) {
136:           changes_neighborhood = true;
137:           changes_topology = true;
138:
139:           two_hop_neighbor = olsr_malloc(sizeof(struct neighbor_2_entry), "Process HELLO");
140:
141:           two_hop_neighbor->neighbor_2_nblist.next = &two_hop_neighbor->neighbor_2_nblist;
142:
143:           two_hop_neighbor->neighbor_2_nblist.prev = &two_hop_neighbor->neighbor_2_nblist;
144:
145:           two_hop_neighbor->neighbor_2_pointer = 0;
146:
147:           two_hop_neighbor->neighbor_2_addr = message_neighbors->address;
148:
149:           olsr_insert_two_hop_neighbor_table(two_hop_neighbor);
150:
151:           linking_this_2_entries(neighbor, two_hop_neighbor, message->vtime);
152:         } else {
153:           /*
154:              linking to this two_hop_neighbor entry
155:            */
156:           changes_neighborhood = true;
157:           changes_topology = true;
158:
159:           linking_this_2_entries(neighbor, two_hop_neighbor, message->vtime);
160:         }
161:       }
162:     }
163:   }
```

Process_packege.c

本函数处理从 hello数据包获取的一系列邻居节点的信息。

88-92：获得邻居节点的主地址，并将其赋值给邻居节点的地址元素。

98-105：如果与这个邻居节点是对称邻居或 MPR，就在邻居表中查找这个节点， 如果这个节点存在于表中，则更新该节点的计时器。

133-159：在两跳邻居表中查找该节点，若该节点不存在则给这个节点赋值一些基本信息，并加入到两跳邻居表中，将一跳邻居与两跳邻居连接；若该节点已经存在，直接将一跳邻居与两跳邻居连接。

4.3 MPR 操作

4.3.1 MPR 选举算法

OLSR 路由协议与传统的链路状态路由算法最明显的区别在于 MPR 的引入。MPR 选举机制在全网有效地扩散拓扑信息的同时，大大降低了路由控制信息的洪泛规模。MPR 的选举原则是可以通过选出的 MPR 节点到达所有的二跳邻居节点，并且要使 MPR 节点的数量尽可能得少。RFC3626 中提出了一种 MPR 贪婪算法，被广泛应用，该算法定义如下：

（1）将一跳邻居节点中所有 willingness 为 WILL_ALWAYS 的节点构成初始 MPR 集。

（2）选出 N 中节点是到达二跳邻居表中某节点的唯一节点，将该节点加入 MPR 集中。并将能通过该节点到达的二跳邻居节点从二跳邻居表中删除。

（3）当二跳邻居表中不存在不能通过 MPR 集到达的二跳邻居节点时，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加入 MPR 集，若有多个选择，选择可达性高的，可达性相同时，选择深度高的节点加入 MPR 集。

（4）将节点每个接口的 MPR 集组合在一起，就是该节点的 MPR 集合。

Olsrd-0.6.0 中通过在 lq_mpr.c 中定义的 olsr_calculate_lq_mpr 函数计算 MPR，分析如下：

lq_mpr.c

```
58:   bool mpr_changes = false;
59:
60:   OLSR_FOR_ALL_NBR_ENTRIES(neigh) {
61:
62:     /* Memorize previous MPR status. */
63:
64:     neigh->was_mpr = neigh->is_mpr;
65:
66:     /* Clear current MPR status. */
67:
68:     neigh->is_mpr = false;
69:
70:     /* In this pass we are only interested in WILL_ALWAYS neighbours */
71:
72:     if (neigh->status == NOT_SYM || neigh->willingness != WILL_ALWAYS) {
73:       continue;
74:     }
75:
76:     neigh->is_mpr = true;
77:
78:     if (neigh->is_mpr != neigh->was_mpr) {
79:       mpr_changes = true;
80:     }
81:
82:   }
83:   OLSR_FOR_ALL_NBR_ENTRIES_END(neigh);
```

lq_mpr.c

58 设置 mpr_changes 的 flag 为假 false

60-83 遍历了所有的邻居表项，进行初始工作

64 将之前的 MPR 状态保存在 was_mpr 中

68 将 is_mpr 清空

72-74 对于非对称或 willingness 不为 WILL_ALWAYS 的节点，直接跳过

76 如果节点不满足以上条件，初始化这些节点为 MPR 节点

78-80 更新 mpr_changes 的 flag，如果 MPR 状态发生了变化，将 mpr_changes 置为 true.

Lq_mpr.c

```
85:   for (i = 0; i < HASHSIZE; i++) {
86:     /* loop through all 2-hop neighbours */
87:
88:     for (neigh2 = two_hop_neighbortable[i].next; neigh2 != &two_hop_neighbortable[i]; neigh2 = neigh2->next) {
89:       best_1hop = LINK_COST_BROKEN;
90:
91:       /* check whether this 2-hop neighbour is also a neighbour */
92:
93:       neigh = olsr_lookup_neighbor_table(&neigh2->neighbor_2_addr);
94:
95:       /* if it's a neighbour and also symmetric, then examine
96:          the link quality */
97:
98:       if (neigh != NULL && neigh->status == SYM) {
99:         /* if the direct link is better than the best route via
100:         * an MPR, then prefer the direct link and do not select
101:         * an MPR for this 2-hop neighbour */
102:
103:        /* determine the link quality of the direct link */
104:
105:        struct link_entry *lnk = get_best_link_to_neighbor(&neigh->neighbor_main_addr);
106:
107:        if (!lnk)
108:          continue;
109:
110:        best_1hop = lnk->linkcost;
111:
112:        /* see wether we find a better route via an MPR */
113:
114:        for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
115:          if (walker->path_linkcost < best_1hop)
116:            break;
117:
118:        /* we've reached the end of the list, so we haven't found
119:         * a better route via an MPR - so, skip MPR selection for
120:         * this 1-hop neighbor */
121:
122:        if (walker == &neigh2->neighbor_2_nblist)
123:          continue;
124:      }
```

Lq_mpr.c

88 遍历邻居环行表的所有二跳邻居

93 通过主地址查询该二跳邻居是否同时是一跳邻居

98-124 如果是，并且该邻居状态是对称的，则检查链路质量。如果直连链路比通过 MPR 的最佳路由要好，就使用直连链路，而不会把该邻居看做二跳邻居进而给它选择 MPR 节点。

如果不是：

lq_mpr.c

```
131:      for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
132:        walker->neighbor->skip = false;
133:
134:      for (k = 0; k < olsr_cnf->mpr_coverage; k++) {
135:        /* look for the best 1-hop neighbour that we haven't
136:         * yet selected */
137:
138:        neigh = NULL;
139:        best = LINK_COST_BROKEN;
140:
141:        for (walker = neigh2->neighbor_2_nblist.next; walker != &neigh2->neighbor_2_nblist; walker = walker->next)
142:          if (walker->neighbor->status == SYM && !walker->neighbor->skip && walker->path_linkcost < best) {
143:            neigh = walker->neighbor;
144:            best = walker->path_linkcost;
145:          }
146:
147:        /* Found a 1-hop neighbor that we haven't previously selected.
148:         * Use it as MPR only when the 2-hop path through it is better than
149:         * any existing 1-hop path. */
150:        if ((neigh != NULL) && (best < best_1hop)) {
151:          neigh->is_mpr = true;
152:          neigh->skip = true;
153:
154:          if (neigh->is_mpr != neigh->was_mpr)
155:            mpr_changes = true;
156:        }
157:
158:        /* no neighbour found => the requested MPR coverage cannot
159:         * be satisfied => stop */
160:
161:        else
162:          break;
163:      }
164:    }
165:  }
```

lq_mpr.c

131-132 将所有的一跳节点都标记为未选择，置为 false

141-145 选择还没有被选择的最佳节点，即拥有较少 path_linkcost 的节点

150-152 当存在上述节点并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

154-155 更新 mpr_changes 的 flag，如果 MPR 状态发生了变化，将 mpr_changes 置为 true

除了 lq_mpr.c 中对 MPR 算法有描述，在 mpr.c 中也有相应的功能：
olsr_calculate_mpr 函数为 mpr.c 中核心函数，功能为计算出 MPR 邻居。

mpr.c

```
389: void
390: olsr_calculate_mpr(void)
391: {
392:   uint16_t two_hop_covered_count;
393:   uint16_t two_hop_count;
394:   int i;
395:
396:   OLSR_PRINTF(3, "\n**RECALCULATING MPR**\n\n");
397:
398:   olsr_clear_mprs();
399:   two_hop_count = olsr_calculate_two_hop_neighbors();
400:   two_hop_covered_count = add_will_always_nodes();
401:
402:   /*
403:    *Calculate MPRs based on WILLINGNESS
404:    */
405:
406:   for (i = WILL_ALWAYS - 1; i > WILL_NEVER; i--) {
407:     struct neighbor_entry *mprs;
408:     struct neighbor_2_list_entry *two_hop_list = olsr_find_2_hop_neighbors_with_1_link(i);
409:
410:     while (two_hop_list != NULL) {
411:       struct neighbor_2_list_entry *tmp;
412:       //printf("CHOSEN FROM 1 LINK\n");
413:       if (!two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor->is_mpr)
414:         olsr_chosen_mpr(two_hop_list->neighbor_2->neighbor_2_nblist.next->neighbor, &two_hop_covered_count);
415:       tmp = two_hop_list;
416:       two_hop_list = two_hop_list->next;;
417:       free(tmp);
418:     }
419:
420:     if (two_hop_covered_count >= two_hop_count) {
421:       i = WILL_NEVER;
422:       break;
423:     }
424:     //printf("two hop covered count: %d\n", two_hop_covered_count);
425:
426:     while ((mprs = olsr_find_maximum_covered(i)) != NULL) {
427:       //printf("CHOSEN FROM MAXCOV\n");
428:       olsr_chosen_mpr(mprs, &two_hop_covered_count);
430:       if (two_hop_covered_count >= two_hop_count) {
431:         i = WILL_NEVER;
432:         break;
433:       }
434:
435:     }
436:   }
437:
438:   /*
439:      increment the mpr sequence number
440:    */
441:   //neighbortable.neighbor_mpr_seq++;
442:
443:   /* Optimize selection */
444:   olsr_optimize_mpr_set();
445:
446:   if (olsr_check_mpr_changes()) {
447:     OLSR_PRINTF(3, "CHANGES IN MPR SET\n");
448:     if (olsr_cnf->tc_redundancy > 0)
449:       signal_link_changes(true);
450:   }
451:
452: }
```

Mpr.c

398 将被选为 MPR 节点的记录清除
399 计算二跳邻居节点个数
400 添加 willingness 为 WILL_ALWAYS 的节点加入 MPR 集
406-436 以各节点的 willingness 级别为基础计算 MPR。具体实现方法如下：
先做如下循环：
（1）首先定义了一个二跳邻居节点链表（two_hop_list），里面的数据为存在一条连接并且 willingness 级别为 i 的二跳邻居节点。
（2）当 two_hop_list 不为空时，如果 two_hop_list 链表中当前二跳节点的邻居节点链表中的邻居并不是 MPR 节点的话，那么选择此邻居成为新的 MPR 节点，并更新节点覆盖计数器。two_hop_list 指向下一个二跳邻居节点链表，重复（2）操作
（3）如果已经覆盖的二跳节点数大于总的二跳节点数，将 i 置为 WILL_NEVER 并退出循环（1）
（4）当在当前 willingness 级别下覆盖到最多二跳节点的节点还存在时，将它添加到 MPR 集合

中。如果已覆盖的二跳节点数大于总的二跳节点数，就退出循环（1）

444-452 这段函数是用来优化 MPR 集合的。首先调用 olsr_optimize_mpr_set()函数进行 MPR 集合优化。然后判断，如果 MPR 集合表变化了，就弹出更改提示；如果全局变量中 tc_redundancy 大于 0，就将链路变化状态置为真 true.

add_will_always_nodes 函数功能为将所有 willingness 值为 WILL_ALWAYS 的节点设置为 MPR.

mpr.c

```
358: add_will_always_nodes(void)
359: {
360:   struct neighbor_entry *a_neighbor;
361:   uint16_t count = 0;
362:
363: #if 0
364:   printf("\nAdding WILL ALWAYS nodes....\n");
365: #endif
366:
367:   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
368:     struct ipaddr_str buf;
369:     if ((a_neighbor->status == NOT_SYM) || (a_neighbor->willingness != WILL_ALWAYS)) {
370:       continue;
371:     }
372:     olsr_chosen_mpr(a_neighbor, &count);
373:
374:     OLSR_PRINTF(3, "Adding WILL_ALWAYS: %s\n", olsr_ip_to_string(&buf, &a_neighbor->neighbor_main_addr));
375:
376:   }
377:   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
378:
379: #if 0
380:   OLSR_PRINTF(1, "Count: %d\n", count);
381: #endif
382:   return count;
383: }
```

mpr.c

367-371 若邻居节点是非对称的或邻居节点的 willingness 不是 WILL_ALWAYS 的，则直接跳过

372-377 将其他剩余邻居节点添加到 MPR 集合中，并返回所添加的节点数量 count.

olsr_chosen_mpr 函数功能为用来处理被选择成为 MPR 的节点并更新那些在 MPR 计算时用到的计数器信息。

mpr.c

```
140: static int
141: olsr_chosen_mpr(struct neighbor_entry *one_hop_neighbor, uint16_t * two_hop_covered_count)
142: {
143:   struct neighbor_list_entry *the_one_hop_list;
144:   struct neighbor_2_list_entry *second_hop_entries;
145:   struct neighbor_entry *dup_neighbor;
146:   uint16_t count;
147:   struct ipaddr_str buf;
148:   count = *two_hop_covered_count;
149:
150:   OLSR_PRINTF(1, "Setting %s as MPR\n", olsr_ip_to_string(&buf, &one_hop_neighbor->neighbor_main_addr));
151:
152:   //printf("PRE COUNT: %d\n\n", count);
153:
154:   one_hop_neighbor->is_mpr = true;      //NBS_MPR;
155:
156:   for (second_hop_entries = one_hop_neighbor->neighbor_2_list.next; second_hop_entries != &one_hop_neighbor->neighbor_2_list;
157:        second_hop_entries = second_hop_entries->next) {
158:     dup_neighbor = olsr_lookup_neighbor_table(&second_hop_entries->neighbor_2->neighbor_2_addr);
159:
160:     if ((dup_neighbor != NULL) && (dup_neighbor->status == SYM)) {
161:       //OLSR_PRINTF(7, "(2)Skipping 2h neighbor %s - already 1hop\n", olsr_ip_to_string(&buf, &second_hop_entries->neighbor_2->
162:       continue;
163:     }
164:     //      if(!second_hop_entries->neighbor_2->neighbor_2_state)
165:     //if(second_hop_entries->neighbor_2->mpr_covered_count < olsr_cnf->mpr_coverage)
166:     //{
167:     /*
168:        Now the neighbor is covered by this mpr
169:      */
170:     second_hop_entries->neighbor_2->mpr_covered_count++;
171:     the_one_hop_list = second_hop_entries->neighbor_2->neighbor_2_nblist.next;
172:
173:     //OLSR_PRINTF(1, "[%s](%x) has coverage %d\n", olsr_ip_to_string(&buf, &second_hop_entries->neighbor_2->neighbor_2_addr), s
174:
175:     if (second_hop_entries->neighbor_2->mpr_covered_count >= olsr_cnf->mpr_coverage)
176:       count++;
177:
178:     while (the_one_hop_list != &second_hop_entries->neighbor_2->neighbor_2_nblist) {
179:
180:       if ((the_one_hop_list->neighbor->status == SYM)) {
```

```
181:         if (second_hop_entries->neighbor_2->mpr_covered_count >= olsr_cnf->mpr_coverage) {
182:           the_one_hop_list->neighbor->neighbor_2_nocov--;
183:         }
184:       }
185:       the_one_hop_list = the_one_hop_list->next;
186:     }
187:
188:     //}
189:   }
190:
191:   //printf("POST COUNT %d\n\n", count);
192:
193:   *two_hop_covered_count = count;
194:   return count;
195:
196: }
```

Mpr.c

154 将 one_hop_neighbor 的 is_mpr 置为真

156-158 for 循环：对 second_hop_entries 进行遍历。循环内容如下：

160-163 如果节点已经存在或状态是对称的，则跳过该节点

170-171 将二跳邻居节点的 MPR 覆盖数加 1，同时将二跳邻居节点的邻居赋给一跳节点链表。（the_one_hop_list）

175-176 如果二跳节点的 MPR 数大于 mpr_coverage,则 count++.

178-188 如果一跳节点链表和二跳节点的邻居列表不同，就去判断一跳邻居是否是否是对称的。若是对称就判断二跳节点的 mpr_covered_count 是否大于等于全局变量 mpr_coverage,，若是则将一跳邻居节点的二跳覆盖减 1。接着一跳链表指向下一邻居，重复这一循环。

193-194 给传进来的参数即二跳邻居节点覆盖数赋值为 count 并返回 count 值

olsr_find_2_hop_neighbor_with_1_link 函数功能，基于一个确定的 willingness 级别查找所有存在一条链路的二跳节点。

mpr.c

```
86: static struct neighbor_2_list_entry *
87: olsr_find_2_hop_neighbors_with_1_link(int willingness)
88: {
89:
90:   uint8_t idx;
91:   struct neighbor_2_list_entry *two_hop_list_tmp = NULL;
92:   struct neighbor_2_list_entry *two_hop_list = NULL;
93:   struct neighbor_entry *dup_neighbor;
94:   struct neighbor_2_entry *two_hop_neighbor = NULL;
95:
96:   for (idx = 0; idx < HASHSIZE; idx++) {
97:
98:     for (two_hop_neighbor = two_hop_neighbortable[idx].next; two_hop_neighbor != &two_hop_neighbortable[idx];
99:          two_hop_neighbor = two_hop_neighbor->next) {
100:
101:       //two_hop_neighbor->neighbor_2_state=0;
102:       //two_hop_neighbor->mpr_covered_count = 0;
103:
104:       dup_neighbor = olsr_lookup_neighbor_table(&two_hop_neighbor->neighbor_2_addr);
105:
106:       if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
107:
108:         //OLSR_PRINTF(1, "(1)Skipping 2h neighbor %s - already 1hop\n", olsr_ip_to_string(&buf, &two_hop_neighbor->neighbor_2_
109:
110:         continue;
111:       }
112:
113:       if (two_hop_neighbor->neighbor_2_pointer == 1) {
114:         if ((two_hop_neighbor->neighbor_2_nblist.next->neighbor->willingness == willingness)
115:             && (two_hop_neighbor->neighbor_2_nblist.next->neighbor->status == SYM)) {
116:           two_hop_list_tmp = olsr_malloc(sizeof(struct neighbor_2_list_entry), "MPR two hop list");
117:
118:           //OLSR_PRINTF(1, "ONE LINK ADDING %s\n", olsr_ip_to_string(&buf, &two_hop_neighbor->neighbor_2_addr));
119:
120:           /* Only queue one way here */
121:           two_hop_list_tmp->neighbor_2 = two_hop_neighbor;
122:
123:           two_hop_list_tmp->next = two_hop_list;
124:
125:           two_hop_list = two_hop_list_tmp;
126:         }
127:       }
128:
129:     }
130:
131:   }
132:
133:   return (two_hop_list_tmp);
134: }
```

mpr.c

96-133 循环遍历二跳邻居表 two_hop_neighbortable，具体功能如下；

104 查询已经存在的二跳邻居节点，并赋给 dup_neighbor

106-111 如果节点已经存在或节点不是非对称的，则直接跳过这一轮循环，进入下一轮

113-126 如果二跳节点存在就判断二跳邻居节点的 willingness 是否满足给定的 willingness 级别并且同时二跳邻居节点的状态是否是对称的。如果是的话，就新建一个空间，将二跳邻居节点添加到二跳邻居节点链表 two_hop__list 中。

133 返回二跳节点链表

olsr_find_maximum_covered 函数功能是找出最多覆盖二跳邻居节点的邻居节点并满足给定的 willingness 级别。

mpr.c

```
206: static struct neighbor_entry *
207: olsr_find_maximum_covered(int willingness)
208: {
209:   uint16_t maximum;
210:   struct neighbor_entry *a_neighbor;
211:   struct neighbor_entry *mpr_candidate = NULL;
212:
213:   maximum = 0;
214:
215:   OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
216:
217: #if 0
218:     printf("[%s] nocov: %d mpr: %d will: %d max: %d\n\n", olsr_ip_to_string(&buf, &a_neighbor->neighbor_main_addr),
219:            a_neighbor->neighbor_2_nocov, a_neighbor->is_mpr, a_neighbor->willingness, maximum);
220: #endif
221:
222:     if ((!a_neighbor->is_mpr) && (a_neighbor->willingness == willingness) && (maximum < a_neighbor->neighbor_2_nocov)) {
223:
224:       maximum = a_neighbor->neighbor_2_nocov;
225:       mpr_candidate = a_neighbor;
226:     }
227:   }
228:   OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
229:
230:   return mpr_candidate;
231: }
```

mpr.c

222-226 如果 a_neighbor 不是 mpr 并且它的 willingness 不是给定的级别，且同时覆盖的二跳节点的数量值 maximum 小于 a_neighbor 所覆盖的二跳节点数。那么将 maximum 赋值为 a_neighbor 所覆盖的二跳节点数。并将此节点设为 MPR 的候选者（以此类推，如果循环遍历后拥有更多二跳覆盖的节点，那么这个节点会替换掉前者称为新的候选者）。

olsr_optimize_mpr_set 函数功能是通过删除二跳节点中被足够多的的 MPR 覆盖的节点来优化 MPR 集合。

mpr.c

```
463: static void
464: olsr_optimize_mpr_set(void)
465: {
466:   struct neighbor_entry *a_neighbor, *dup_neighbor;
467:   struct neighbor_2_list_entry *two_hop_list;
468:   int i, removeit;
469:
470: #if 0
471:   printf("\n**MPR OPTIMIZING**\n\n");
472: #endif
473:
474:   for (i = WILL_NEVER + 1; i < WILL_ALWAYS; i++) {
475:
476:     OLSR_FOR_ALL_NBR_ENTRIES(a_neighbor) {
477:
478:       if (a_neighbor->willingness != i) {
479:         continue;
480:       }
481:
482:       if (a_neighbor->is_mpr) {
483:         removeit = 1;
484:
485:         for (two_hop_list = a_neighbor->neighbor_2_list.next; two_hop_list != &a_neighbor->neighbor_2_list;
486:              two_hop_list = two_hop_list->next) {
487:
488:           dup_neighbor = olsr_lookup_neighbor_table(&two_hop_list->neighbor_2->neighbor_2_addr);
489:
490:           if ((dup_neighbor != NULL) && (dup_neighbor->status != NOT_SYM)) {
491:             continue;
492:           }
493:           //printf("\t[%s] coverage %d\n", olsr_ip_to_string(&buf, &two_hop_list->neighbor_2->neighbor_2_addr), two_hop_list->
494:           /* Do not remove if we find a entry which need this MPR */
495:           if (two_hop_list->neighbor_2->mpr_covered_count <= olsr_cnf->mpr_coverage) {
496:             removeit = 0;
497:           }
498:         }
499:
```

```
500:         if (removeit) {
501:           struct ipaddr_str buf;
502:           OLSR_PRINTF(3, "MPR OPTIMIZE: removiong mpr %s\n\n", olsr_ip_to_string(&buf, &a_neighbor->neighbor_main_addr));
503:           a_neighbor->is_mpr = false;
504:         }
505:       }
506:     } OLSR_FOR_ALL_NBR_ENTRIES_END(a_neighbor);
507:   }
508: }
```

mpr.c

474-503 若邻居节点的 willingness 级别不是所指定的，则跳过这一轮，进入下一轮循环。如果覆盖二跳邻居的 MPR 数量值小于全局变量 olsr_cnf->mpr_coverage 的值，那么就不需要移除，否则，该 MPR 就是冗余的，会被除去。

4.4 拓扑消息处理

拓扑控制消息即 TC消息，它由一个网络中的节点发出，用以声明通告链路集(advertised link set)，集合中必须至少包含通往其 MPR选择者集中所有节点，也就是选取发送节点作为 MPR的邻居的链路。为建立拓扑信息库，每一个被选为 MPR的节点广播 TC消息，使得网络中的所有节点都能得到 TC消息并建立全网拓扑。非 MPR的节点不会转发 TC消息，从而避免了广播风暴。MPR机制提供了拓扑信息分发中更好的可扩展性。

tc.set.c

```
789: bool
790: olsr_input_tc(union olsr_message * msg, struct interface * input_if __attribute__ ((unused)),
791:               union olsr_ip_addr * from_addr)
792: {
793:   struct ipaddr_str buf;
794:   uint16_t size, msg_seq, ansn;
795:   uint8_t type, ttl, msg_hops, lower_border, upper_border;
796:   olsr_reltime vtime;
797:   union olsr_ip_addr originator;
798:   const unsigned char *limit, *curr;
799:   struct tc_entry *tc;
800:   bool emptyTC;
801:
802:   union olsr_ip_addr lower_border_ip, upper_border_ip;
803:   int borderSet = 0;
804:
805:   curr = (void *)msg;
806:   if (!msg) {
807:     return false;
808:   }
809:
810:   /* We are only interested in TC message types. */
811:   pkt_get_u8(&curr, &type);
812:   if ((type != LQ_TC_MESSAGE) && (type != TC_MESSAGE)) {
813:     return false;
814:   }
```

tc.set.c

本函数处理接收到的 TC消息，如果消息有价值，则更新拓扑图中边的信息， 触发最短路径计算，最后将其洪泛给邻居节点

806-809：将TC消息指针强制转换类型以便后续的字节操作，若消息为空则直接退出

812-814：如果消息类型不是 TC消息则直接退出

tc.set.c

```
821:  if (check_neighbor_link(from_addr) != SYM_LINK) {
822:    OLSR_PRINTF(2, "Received TC from NON SYM neighbor %s\n", olsr_ip_to_string(&buf, from_addr));
823:    return false;
824:  }
825:
826:  pkt_get_reltime(&curr, &vtime);
827:  pkt_get_u16(&curr, &size);
828:
829:  pkt_get_ipaddress(&curr, &originator);
830:
831:  /* Copy header values */
832:  pkt_get_u8(&curr, &ttl);
833:  pkt_get_u8(&curr, &msg_hops);
834:  pkt_get_u16(&curr, &msg_seq);
835:  pkt_get_u16(&curr, &ansn);
836:
837:  /* Get borders */
838:  pkt_get_u8(&curr, &lower_border);
839:  pkt_get_u8(&curr, &upper_border);
840:
841:  tc = olsr_lookup_tc_entry(&originator);
```

tc.set.c

821-825：检查链路类型，如果与邻居节点的链路不是对称链路就直接退出
826-839：从数据包中根据数据报结构按字节获取各部分数据
841：根据 ip地址从 TC树上获取 TC实体

tc.set.c

```
882:  if (!tc) {
883:    tc = olsr_add_tc_entry(&originator);
884:  }
885:
886:  /*
887:   * Update the tc entry.
888:   */
889:  tc->msg_hops = msg_hops;
890:  tc->msg_seq = msg_seq;
891:  tc->ansn = ansn;
892:  tc->ignored = 0;
893:  tc->err_seq_valid = false;
```

tc.set.c

882-893：如果没有找到这个 TC实体，那么就新建一个并加入到 TC树中，之后对该 TC实体进行属性的更新

tc.set.c

```
901:  limit = (unsigned char *)msg + size;
902:  borderSet = 0;
903:  emptyTC = curr >= limit;
904:  while (curr < limit) {
905:    if (olsr_tc_update_edge(tc, ansn, &curr, &upper_border_ip)) {
906:      changes_topology = true;
907:    }
908:
909:    if (!borderSet) {
910:      borderSet = 1;
911:      memcpy(&lower_border_ip, &upper_border_ip, sizeof(lower_border_ip));
912:    }
913:  }
914:
915:  /*
916:   * Calculate real border IPs.
917:   */
918:  if (borderSet) {
919:    borderSet = olsr_calculate_tc_border(lower_border, &lower_border_ip, upper_border, &upper_border_ip);
920:  }
```

tc.set.c

901-913：遍历 TC数据包中的通告链路并更新拓扑图中的边的信息
918-920：计算边界 IP的个数

tc.set.c

```
935:   if (borderSet) {
936:
937:     /*
938:      * Delete all old tc edges within borders.
939:      */
940:     olsr_delete_revoked_tc_edges(tc, ansn, &lower_border_ip, &upper_border_ip);
941:   } else {
942:
943:     /*
944:      * Kick the the edge garbage collection timer. In the meantime hopefully
945:      * all edges belonging to a multipart neighbor set will arrive.
946:      */
947:     olsr_set_timer(&tc->edge_gc_timer, OLSR_TC_EDGE_GC_TIME, OLSR_TC_EDGE_GC_JITTER, OLSR_TIMER_ONESHOT,
948:                    &olsr_expire_tc_edge_gc,tc, tc_edge_gc_timer_cookie);
949:   }
950:
951:   if (emptyTC && borderSet) {
952:     /* cleanup MIDs and HNAs if all edges have been erased by
953:      * an empty TC, then alert the duplicate set and kill the
954:      * tc entry */
955:     olsr_cleanup_mid(&originator);
956:     olsr_cleanup_hna(&originator);
957:     olsr_cleanup_duplicates(&originator);
958:
959:     olsr_delete_tc_entry(tc);
960:   }
961:   /* Forward the message */
962:   return true;
963: }
```

tc.set.c

935-940：删除边界值以内的没有得到更新的 TC边

951-963：如果所有的TC边都被一个空的 TC消息删除，那么清空 mid和hna，向冗余集发送警告，然后删除TC实体，最后将 TC消息转发。

tc.set.c

```
622: static int
623: olsr_tc_update_edge(struct tc_entry *tc, uint16_t ansn, const unsigned char **curr, union olsr_ip_addr *neighbor)
624: {
625:   struct tc_edge_entry *tc_edge;
626:   int edge_change;
627:
628:   edge_change = 0;
629:
630:   /*
631:    * Fetch the per-edge data
632:    */
633:   pkt_get_ipaddress(curr, neighbor);
634:
635:   /* First check if we know this edge */
636:   tc_edge = olsr_lookup_tc_edge(tc, neighbor);
637:
638:   if (!tc_edge) {
639:
640:     /*
641:      * Yet unknown - create it.
642:      * Check if the address is allowed.
643:      */
644:     if (!olsr_validate_address(neighbor)) {
645:       return 0;
646:     }
647:
648:     tc_edge = olsr_add_tc_edge_entry(tc, neighbor, ansn);
649:
650:     olsr_deserialize_tc_lq_pair(curr, tc_edge);
651:     edge_change = 1;
652:
653:   } else {
654:
655:     /*
656:      * We know this edge - Update entry.
657:      */
658:     tc_edge->ansn = ansn;
```

```
659:
660:     /*
661:      * Update link quality if configured.
662:      */
663:     if (olsr_cnf->lq_level > 0) {
664:       olsr_deserialize_tc_lq_pair(curr, tc_edge);
665:     }
666:
667:     /*
668:      * Update the etx.
669:      */
670:     if (olsr_calc_tc_edge_entry_etx(tc_edge)) {
671:       edge_change = 1;
672:     }
673: #if DEBUG
674:     if (edge_change) {
675:       OLSR_PRINTF(1, "TC:   chg edge entry %s\n", olsr_tc_edge_to_string(tc_edge));
676:     }
677: #endif
678:
679:   }
680:
681:   return edge_change;
682: }
```

tc.set.c

本函数根据接收到的 TC消息更新 TC实体中边的信息

633-636：获取数据包源端的 IP地址并查找这条边是否已经存在于 TC实体中

638-651：如果这条边不存在，先验证 IP地址的有效性，若无效则直接返回，否则将这条边加入到 TC实体中，并将标记变量 edge_change记为1

653-672：如果这条边已经存在，则更新它的ANSN和etx，并将标记变量edge_change记为1

681：如果边的信息被修改过返回 1，否则返回 0.

4.5 路由计算

OLSR 协议在路由发现策略上与传统路由协议类似。通过在全网范围内周期性地交换网络拓扑信息和链路状态信息，OLSR 协议的每个节点都掌握了全网拓扑图的最新信息。移动节点运用 Dijkstra 算法，在已有的网络拓扑图的基础上，主动发现去往网络中其他节点的路由。链路状态信息洪泛将会引起路由开销，OLSR 主要采用两种方法：一种是多跳中继，每个节点在自己的一跳邻居节点中选择一部分节点作为自己的 MPR，又利用 MPR 代替所有节点转发链路状态消息，实现路由控制消息的选择性洪泛；另一种是压缩链路状态信息，这是因为链路状态信息只描述了与 MPR 之间的链路，而没有描述与所有的一跳邻居节点之间的链路。

本文介绍的 OLSR 协议采用的是 olsrd-0.6.0 版本。

4.5.1 相关结构体

下面结构体是路由选择时使用的矩：

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: };
```

routing_table.h

68 cost 为两个路由点之间的花销

69 hops 为两个路由点之间的跳数

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: };
```

routing_table.h

74 gateway 为下一跳的网关（IPv4 或 IPv6）
75 iif_index 为接口索引

routing_table.h

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

routing_table.h

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。
86 rt_dst 该信息的路由地址和前缀长度
87 rt_tree_node 包含了 val_tree 的信息

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: };
```

routing_table.h

以上结构体包含了 rt_path 的信息，接收到的 rt_path 会被加入到路由信息表中，用于计算 best_path.

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: };
```

routing_table.h

上图结构体是 IPv4 和 IPv6 的核心路由表结构。核心路由表表现出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。

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: };
```

routing_table.h

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

4.5.2 路由表的计算

olsr_init_routing_table 函数功能是初始化路由表。

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: }
```

routing_table.c

173 通过 avl_init()函数初始化一个 avl 树

174 维护一个版本号 routingtree_version 用来检测 rt_entry 和 rt_path 子树中信息是否过时，初始化为 0

179-183 为 rt_entry 和 rt_path 分配内存，创建相应的 coolie

olsr_lookup_routing_table 函数功能是在 avl 树里找到一个地址的路由表条目。

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: }
```

routing_table.c

200-201 根据给定的参数地址并排上设置的最长前缀长度

203 调用 avl_find()函数在 routingtree 里找到该地址的的 rt_entry

205 如果找到了，就利用 rt_tree2rt()函数将返回节点转化为 rt_entry 类型，没找到则返回空

olsr_update_rt_path 函数功能为更新 rt_path。每条路径都是周期性更新。

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: }
```

routing_table.c

215 修改维护 routingtree_version 的值
218 更新网关地址 rtp_nexthop.gateway
221 更新接口地址 rtp_nexthop.iif_index
224 更新跳数 rtp_metric.hops
225 更新路径花销 rtp_metric.cost

olsr_alloc_rt_entry 函数功能是创建一个可用的路由条目，对于提供的参数即 IP 前缀分配一个路由条目空间，初始化后插入到 avl 树里。

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: }
```

routing_table.c

234 函数 olsr_cookie_malloc()申请内存空间并清零
239 函数 memset()申请内存空间并清零
245 将该入口的目的地址设置为参数提供的入口地址
248 函数 avl_insert()将入口的树节点插入到路由表中
251 用函数 avl_init()初始化树

olsr_insert_rt_path 函数功能为每个 rt_path 创建一个路由入口，把它加到全局 RIB 树里。

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: }
```

routing_table.c

301-303 利用 if 函数判断参数传入的 tc_entry 是否是 ROUTE_COST_BROKEN，如果不是，则判断参数不符合条件，直接返回

308-310 利用 if 函数判断参数传入的 rtp 的目的地址长度（rtp_dst.prefix_len）是否大于所设定的最大地址长度，如果不是，则判断参数不符合条件，直接返回

315 调用函数 avl_find()检查传入的 rtp 节点是否都在路由表中

317-328 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。

337 将节点添加进 avl 树里

343 更新参数的值，并更新整个路由表

olsr_delete_rt_path 函数功能为删除 rtp 树，并将其从 TC 树中删除，改变路由表版本。

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: }
```

routing_table.c

355 函数 avl_delete()将 rtp 指向的树节点从所在树里删除
356 rtp_rt 指向的树的根置空
361 函数 avl_delete()将 rtp 从前缀里删除
362 函数 olsr_unlock_tc_entry()解锁相应的 tc_entry
363 删除 TC 树里 rtp 所指向的树节点
371 函数 olsr_cookie_free()删除 cookie 所占用的内存

olsr_cmp_rtp 函数功能是比较两个路由，如果第一个参数更好的话返回值为 true。

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: }
```

routing_table.c

446-451 比较两个路由的花销，花销小的路径更优
454-459 若花销一样的话，比较两个路由的跳数，跳数小的更优
462-466 若跳数还一样，则比较两个路由的源地址，源地址小的更优

olsr_rt_best 函数功能是通过遍历找到最优路径，并把当前网关路径改为最佳路径。

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: }
```

routing_table.c

489 调用函数 avl_walk_first()查找树里的第一个条目
491 保证找到的不为零
493 将节点类型转化为 rt_entry
496-502 遍历整棵树，比较当前路径与记录的最佳路径，如果当前路径优于记录的最佳路径，则将当前路径记录为最佳路径

olsr_insert_routing_table 函数功能是将一个前缀插入 TC 的前缀树。

routing_table.c

```
523: struct rt_path *
524: olsr_insert_routing_table(union olsr_ip_addr *dst, int plen, union olsr_ip_addr *originator, int origin)
525: {
526: #ifdef DEBUG
527:   struct ipaddr_str dstbuf, origbuf;
528: #endif
529:   struct tc_entry *tc;
530:   struct rt_path *rtp;
531:   struct avl_node *node;
532:   struct olsr_ip_prefix prefix;
533:
534:   /*
535:    * No bogus prefix lengths.
536:    */
537:   if (plen > olsr_cnf->maxplen) {
538:     return NULL;
539:   }
540:
541:   /*
542:    * For all routes we use the tc_entry as an hookup point.
543:    * If the tc_entry is disconnected, i.e. has no edges it will not
544:    * be explored during SPF run.
545:    */
546:   tc = olsr_locate_tc_entry(originator);
547:
548:   /*
549:    * first check if there is a rt_path for the prefix.
550:    */
551:   prefix.prefix = *dst;
552:   prefix.prefix_len = plen;
553:
554:   node = avl_find(&tc->prefix_tree, &prefix);
555:
556:   if (!node) {
557:
558:     /* no rt_path for this prefix yet */
559:     rtp = olsr_alloc_rt_path(tc, &prefix, origin);
560:
561:     if (!rtp) {
562:       return NULL;
563:     }
564: #ifdef DEBUG
565:     OLSR_PRINTF(1, "RIB: add prefix %s/%u from %s\n", olsr_ip_to_string(&dstbuf, dst), plen,
566:                 olsr_ip_to_string(&origbuf, originator));
567: #endif
568:
569:     /* overload the hna change bit for flagging a prefix change */
570:     changes_hna = true;
571:
572:   } else {
573:     rtp = rtp_prefix_tree2rtp(node);
574:   }
575:
576:   return rtp;
577: }
```

routing_table.c

546 调用函数 olsr_locate_tc_entry()判断给 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。

556-576 检查该前缀是否已经存在一条路径，如果没有，就调用函数 olsr_alloc_rt_path()创建一跳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，根据测试结果可以设置相关链路状态。这样路由器节点就可以构造出自己的链路状态信息包，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。

最后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。节点接收到所有的链路状态信息包后，就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，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。

以上，路由计算就完成了。

5 小结

通过为期八周的工作，在小组成员的共同努力下，我们终于完成了优化链路状态路由协议代码分析文档。在本文档中，我们从 OLSR 简介、olsrd-0.6.0 代码介绍、数据结构、算法四个方面对 OLSR 协议进行了介绍与分析。

OLSR 协议和传统的 LS 路由协议一样，仍是通过周期性的广播 HELLO 消息与其他节点建立连接，通过 TC 消息获取拓扑信息并建立拓扑表，然后用 DIJKSTRA 最短路径算法计算路由。OLSR 协议的特别之处在于使用 MPR 机制选择性泛洪，每个节点根据 MPR 选择算法选出自己的 MPR 集，只有 MPR 节点才可以转发 TC 消息，从而避免了广播风暴。

虽然描述起来就一两句话，但是 olsrd-0.6.0 的代码量十分庞大，仅 src 文件里就有 123 个文件，因此我们也未能将代码分析得十分全面。在代码分析过程中，我们学会了使用 source insight 的跳转功能去查看函数的定义，十分方便，也帮助我们理清了这个庞大的代码的总框图；另外，OLSR 的 RFC 文档也十分的长，共计 75 页，不过对照文档分析代码让我们的整个分析十分的有条理。

最后感谢覃老师八周的耐心指导！通过上课学习和大作业的实践分析，我们对网络协议有了进一步深入的了解，一个协议虽然描述起来很简单，但是考虑到诸多的现实因素，从代码本身容错性到硬件的能耗和寿命都有可能成为网络协议实现的障碍，所以 OLSR 协议几经更新，才有了 olsrd-0.6.0 如此庞大的体量。这段时间我们不仅对 OLSR 协议有了一个深入的了解，也获得了大量的代码阅读训练以及对网络编程和高级 C 语言的了解；另外通过对 RFC3626 的阅读，也提高了我们英文文献阅读的能力。