**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 | | 201692051 | 商迪  （组长） | 软网1604 | 1.代码介绍  2.OLSR工作流程  3.MPR集、MID信息表、重复消息信息表和MID消息的数据结构分析，参与部分邻居集的数据结构分析。  4.MPR集生成、多端口节点功能处理  5.虫洞攻击分析建模  6.文档编写和最后的汇总排版总结。 |  | | 201692231 | 陈雨涵 | 软网1604 | 1.引言适用性部分编写  2.路由表、HNA消息的数据结构分析。  3.路由计算、HNA消息处理  5.参与文档的编写和资料查找 |  | | 201692346 | 刘棪 | 软网1604 | 1.协议概述  2.OLSR数据包、HELLO消息、TC消息、链路状态信息表、邻居集和拓扑路由表的数据结构分析  3.邻居集的生成、TC控制  5.参与文档的编写 |  | |

目录

[1 引言 1](#_Toc534647163)

[1.1 OLSR相关术语 1](#_Toc534647164)

[1.2 协议概述 2](#_Toc534647165)

[1.3 适用性 3](#_Toc534647166)

[2 代码介绍 4](#_Toc534647167)

[2.1 代码文件介绍 4](#_Toc534647168)

[2.2 全局变量分析 6](#_Toc534647169)

[2.3 配置变量分析 7](#_Toc534647170)

[3 数据结构 8](#_Toc534647171)

[3.1 消息类数据结构 8](#_Toc534647172)

[3.1.1 OLSR数据包 8](#_Toc534647173)

[3.1.2 HELLO消息 10](#_Toc534647174)

[3.1.3 MID消息 12](#_Toc534647175)

[3.1.4 TC消息 13](#_Toc534647176)

[3.1.5 HNA消息 15](#_Toc534647177)

[3.2 信息表类数据结构 17](#_Toc534647178)

[3.2.1 链路状态信息表 17](#_Toc534647179)

[3.2.2 邻居集信息表（一跳邻居表中包含两跳邻居表） 17](#_Toc534647180)

[3.2.3 MPR集信息表 20](#_Toc534647181)

[3.2.4 MID信息表 21](#_Toc534647182)

[3.2.5 拓扑信息表 23](#_Toc534647183)

[3.2.6 路由表 23](#_Toc534647184)

[3.2.7 重复消息信息表 24](#_Toc534647185)

[4 OLSR工作流程 25](#_Toc534647186)

[4.1 main函数的分析 25](#_Toc534647187)

[4.2 接收数据包处理过程 27](#_Toc534647188)

[4.3 周期性发送数据包处理过程 28](#_Toc534647189)

[5 OLSR重点功能分析 30](#_Toc534647190)

[5.1 邻居信息表的生成 30](#_Toc534647191)

[5.1.1 链路状态感知 30](#_Toc534647192)

[5.1.2 邻居探测 34](#_Toc534647193)

[5.2 MPR集的生成 38](#_Toc534647194)

[5.2.1 MPR选择计算中用到的重要参数 39](#_Toc534647195)

[5.2.2 MPR选择算法 39](#_Toc534647196)

[5.3 多端口（MID）节点功能处理 45](#_Toc534647197)

[5.3.1 MID消息的产生 45](#_Toc534647198)

[5.3.2 MID消息的处理 46](#_Toc534647199)

[5.3.3 MID消息的转发 50](#_Toc534647200)

[5.3.4 多端口节点功能对OLSR中其他功能的支持 50](#_Toc534647201)

[5.4 TC控制 50](#_Toc534647202)

[5.5 路由计算 53](#_Toc534647203)

[5.5.1 相关数据结构 54](#_Toc534647204)

[5.5.2 路由表计算算法描述 56](#_Toc534647205)

[5.5.3 路由表计算 57](#_Toc534647206)

[5.6 HNA消息处理 67](#_Toc534647207)

[5.6.1 HNA包的生成 67](#_Toc534647208)

[5.6.2 HNA包的发送 68](#_Toc534647209)

[5.6.3 HNA包的处理 69](#_Toc534647210)

[6 OLSR路由协议的安全问题 71](#_Toc534647211)

[6.1虫洞攻击简介 71](#_Toc534647212)

[6.2 针对OLSR虫洞攻击建模 72](#_Toc534647213)

[7 感悟和总结 74](#_Toc534647214)

[7.1 总结 74](#_Toc534647215)

[7.2 陈雨涵感想 74](#_Toc534647216)

[7.3 刘棪感想 74](#_Toc534647217)

[7.4 商迪感想 75](#_Toc534647218)

1 引言

最佳链路状态路由协议（The Optimized Link State Routing Protocol, OLSR）是专门为移动自组网络（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。它是对传统链路状态路由算法的优化。OLSR路由协议不仅继承了传统链路状态算法的稳定性 、快速收敛速度和最短路径路由，而且利用其特有的多点中继节点（MPR）机制优化了洪泛算法，大大降低了协议开销。同时由于其是先应式路由协议，查找路由时延小。因此，OLSR路由协议被广泛应用于无线移动自组织网络中。

## OLSR相关术语

本文在分析说明OLSR路由协议的时候可能会用到以下术语。

* 节点

实现本文档中指定的优化链路状态路由协议的MANET路由器。

* OLSR接口

参与运行OLSR的MANET的网络设备。一个节点可能有几个OLSR接口，每个接口分配一个唯一的IP地址。

* 非OLSR接口

一种网络设备，不参与运行OLSR的MANET。一个节点可能有几个非OLSR接口(无线和/或有线)。来自这些接口的路由信息可以注入到OLSR路由域中。

* 单个OLSR接口节点

具有单一OLSR接口的节点，参与OLSR路由域。

* 多个OLSR接口节点

具有多个OLSR接口的节点，参与OLSR路由域。

* 主IP地址

节点的主地址，在OLSR控制流量中用作该节点发出的所有消息的“发送方地址”。它是节点的一个OLSR接口的地址。单个OLSR接口节点必须使用其唯一的OLSR接口的地址作为主地址。多个OLSR接口节点必须选择其中一个OLSR接口地址作为其“主地址”(相当于“路由器ID”或“节点标识符”)。选择哪个地址并不重要，但是节点应该始终使用与其主地址相同的地址。

* 邻居节点

如果节点Y能听到节点X(即节点X上的OLSR接口与Y上的OLSR接口之间存在链路)。则X,Y互为邻居节点。

* 2-hop邻居

邻居节点的邻居节点。

* 严格2-hop邻居

一个2跳邻居，它不是节点本身，也不是节点的邻居，而且是节点的willingness值与WILL\_NEVER不同的邻居的邻居。

* 多点继电器(MPR)

一个节点，由它的单跳邻居节点X选择，以“重新传输”它从X接收到的所有广播消息，前提是消息不是重复的，并且消息的生存时间字段大于1。

* 多点继电器选择器(MPR选择器，MS)

选择其单跳邻居节点X作为其多点继电器的节点称为节点X的多点继电器选择器。

* 链接

一个链接是一对OLSR接口(来自两个不同的节点)，它们很容易听到彼此的声音(例如，其中一个可能会接收来自另一个的流量)。当一个节点的一个接口具有到另一个节点的一个接口的链接时，该节点称为具有到另一个节点的链接。

* 对称的链接

两个OLSR接口之间经过验证的双向链接。

* 不对称的链接

两个OLSR接口之间的链接，仅在一个方向上验证。

* 对称1-hop邻域

任意节点X的对称单跳邻域是至少有一个到X的对称链路的节点的集合。

* 对称2-hop邻域

X的对称2跳邻域是节点集，不包括X本身，它与X的对称1跳邻域有对称的链路。

* 严格对称的2-hop邻域

X的对称严格2跳邻域是不包括X本身及其邻域的节点集，这些节点与某个对称的1跳邻域具有对称的链路，其意愿与WILL\_NEVER不同

## 协议概述

OLSR是为移动自组网络设计的一种主动路由协议。该协议继承了链路状态算法的稳定性，由于其主动性，具有即时计算路由的优点。OLSR通过选择特定节点（MPR结点）来重新传输控制消息，从而最大限度地减少了网络中流量控制的开销。与传统的链路状态路由协议相比，OLSR显著减少了将消息泛洪到网络中所有节点所需的重新传输次数。其次，为了向网络结点提供最短路径路由信息，OLSR仅需要对部分链路状态的信息进行泛洪。为了实现路由，每个结点需要将其最小链路状态集泛洪到整个网络。最小链路状态信息集是，所有选择为MPR的节点必须声明到其MPR Selector的链路。OLSR协议通过减少周期控制消息传输的间隔时间来改善结点对网络拓扑结构变化的不良反应。而且，由于OLSR协议使得结点一直维护着到网络中所有结点的路由信息，因此该协议特别适合于一个大集群的结点与另一个大集群结点通信的情况，尤其当目的地址源地址在不停变化的时候。此外该协议特别适合大型且密集的网络场景，因为MPR机制在上述环境中工作会极大的改善链路状态协议的工作性能。网络规模越大越密集，该协议与传统的链路状态协议相比就越具有优越性。从结构上来看，OLSR算一个分布式的网络协议，并不需要任何控制实体。该协议并不需要每条消息都必须可靠的传输：因为每个结点都会周期性的发送一次控制消息，因此可以在一定程度内容忍消息的丢失。最后，OLSR不需要对IP数据包的格式进行任何更改。 因此，任何现有的IP栈都可以按原样使用

## 1.3 适用性

最佳链路状态路由协议是用于移动自组织网络的先验式路由协议。它十分适用于大且密集的移动网络，由于优化的实现，中继节点机制能够很好地适用于移动网络以及越大越密集的网络。和传统的链路状态算法相比，优化性能达到更大。

OLSR还十分适用于那些流量在较大的节点集之间是随机的和零星的，而不是几乎仅在区域特定的节点之间的网络。作为一个先验式的协议。

OLSR也适用于那些通信随时间变化的场景。因为路由是由已知的而且所有节点时刻维护的，它不会有额外的流量控制产生。

2 代码介绍

## 2.1 代码文件介绍

OLSR路由协议代码文件由99个代码子文件组成（包括头文件和源文件）。文件组成繁杂，但根据不同功能可将文件分类梳理，便于分析。图2.1展示了大部分重要文件及它们之间的关系。

图2.1 OLSR中代码文件关系图

## 2.2 全局变量分析

OLSR路由协议定义多个全局变量以支持协议的正常运行，如表2.1就部分重要全局变量进行解析。

|  |  |  |
| --- | --- | --- |
| 数据类型 | 全局变量 | 描述 |
| struct olsrd\_config \* | olsr\_cnf | 定义了节点的全局OLSR的相关配置参数 |
| const char | olsrd\_version [] | 定义了OLSR协议的版本 |
| const char | build\_date [] | 定义了本协议建立日期 |
| const char | build\_host [] | 定义了本协议建立主机 |
| FILE \* | debug\_handle | 调试信息输出到的文件的文件指针 |
| struct olsrd\_config \* | olsr\_cnf | 节点配置信息 |
| uint32\_t | now\_times | 当前时间 |
| struct mid\_address | reverse\_mid\_set [HASHSIZE] | 反向MID信息表用于根据端口信息反向查找节点主IP地址 |
| struct neighbor\_2\_entry | two\_hop\_neighbortable [HASHSIZE] | 两跳邻居信息表 |
| struct neighbor\_entry | neighbortable [HASHSIZE] | 一跳邻居节点信息表 |
| struct avl\_tree | duplicate\_set | 类型为平衡二叉树数据结构的重复列表，主要用于记录消息的序列号，防止重复处理冗余消息。 |
| struct parse\_function\_entry \* | parse\_functions  preprocessor\_functions  packetparser\_functions | 解析器功能列表，预处理功能列表，数据包解析器功能列表，这些全局变量主要记录了所有的消息处理和预处理函数 |

表2.1 OLSR中部分全局变量

## 2.3 配置变量分析

OLSR路由协议中的任何具体实现都必须支持表2.2列出的配置变量，且必须支持通过系统管理可以修改这些配置变量取值的某种机制。

|  |  |  |
| --- | --- | --- |
| 配置变量 | 默认值 | 描述 |
| OLSR\_LINK\_JITTER | 5s | OLSR消息抖动时间 |
| OLSR\_LINK\_HELLO\_JITTER | 0s | HELLO消息抖动时间 |
| OLSR\_LINK\_SYM\_JITTER | 0s | 系统抖动时间 |
| OLSR\_LINK\_LOSS\_JITTER | 0s | 数据包丢失抖动时间 |
| DEF\_IP\_VERSION | AF\_INET | 缺省IP协议域 |
| DEF\_USE\_HYST | False | 缺省消息迟滞 |
| DEF\_LQ\_LEVEL | 2 | 缺省链路质量等级 |
| DEF\_OLSRPORT | 698 | 缺省OLSR端口号 |
| DEF\_MIN\_TC\_VTIME | 0.0 | TC 消息vtime最小取值 |
| DEF\_GW\_TYPE | GW\_UOLINK\_IPV46 | 缺省网关类型 |
| DEF\_DOWNLINK\_SPEED | 1024 / kb s | 缺省的链路下载速度 |
| MAX\_LQ\_LEVEL | 2 | 缺省链路质量最高等级 |
| MIN\_LQ\_LEVEL | 0 | 缺省链路质量最低等级 |
| DEF\_RTPROTO | 0 | 缺省的路由协议 |
| COOKIE\_ID\_MAX | 25 | 系统cookie数量最大值 |
| MAXMESSAGESIZE | 1500kb | 广播数据包大小的最大值 |
| MAX\_TTL | 0.165ms | 定义TTL最大值 |
| MAXJITTER | HELLO\_INTERVAL/4 | OLSR消息传播最大抖动 |

表2.2 OLSR中的配置变量

3 数据结构

OLSR路由协议维护有多种数据结构。它们根据功能可大致分为两类：一类是用于节点间传递不同消息的消息类数据结构；一类是存储网络当前各种信息的信息表结构。它们是整个路由协议中消息处理和转发以及其他重要功能的基础。因此，本章主要介绍OLSR路由协议中包含的一些重要的数据结构，为读者阅读下面章节打下基础。同时，如在阅读下面章节时，对某些数据结构不熟悉，也可参考翻阅。

## 3.1 消息类数据结构

### 3.1.1 OLSR数据包

OLSR使用统一的数据包格式与所有与协议相关的数据进行通信。这样做的目的是在不破坏向后兼容性的情况下促进协议的可扩展性。这还提供了将不同“类型”信息装配到一条消息中的简单方式，因此优化了传送效率，最大化利用由网络提供的最大帧大小。这些数据包在网络传输中嵌入在UDP数据报中。在本文分析中的网络地址为IPv4格式。

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

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

表3.1介绍OLSR包的格式。

1 2 3 4

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

表3.1 OLSR包格式

数据包结构解析：

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

olsr\_protocol.h

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

图3.1 OLSR包代码定义

图3.1是OLSR定义的OLSR包结构体变量，这里各项代表的含义和上述一样，就不再做表述。图3.2为此数据结构组织图。

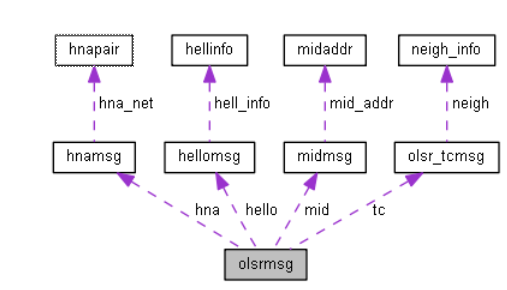


图3.2 OLSR包组织图

### 3.1.2 HELLO消息

在路由器上激活了OLSR协议之后，只有通过路由协议消息“探索”到了邻居路由器，才能与其建立邻接关系。运行OLSR协议的路由器会同时发送并侦听Hello消息。Hello协议消息所起的作用有，链路感知，邻居探测和MPR集的选择。

表3.2为OLSR Hello消息的格式。

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

表3.2 OLSR Hello消息的格式

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

数据包结构解析：

* Reserved：永远被设置为“0000000000000”。
* Htime：Hello消息发送的间隔时间，根据首部的Vtime得到，其计算公式为，emission interval=C\*(1+a/16)\*2^b，在这里a是OLSR首部的Vtime的高四位，b是Vtime的低四位，C是一个特殊的数字。
* Willgnes：表明结点的成为MPR结点的程度。
* WILL\_NEVER：不会被选择为MPR结点。
* WILL\_ALWAYS：总是会被选择为MPR结点。
* Link\_code：该项是一个8比特的数据项，其值永远<15,当一个消息中的Link\_code<15时，代表其具有2个域：0~1位代表链路类型，2~3位代表邻居类型。OLSR中的链路类型有4种：不确定链路UNSPEC\_LINK（0），非对称链路ASYM\_LINK（1），对称链路SYM\_LINK（2），丢失链路LOST\_LINK（3）。3个邻居类型：非邻居NOT\_NEIGH（0），对称邻居SYM\_NEIGH（1），MPR邻居MPR\_NEIGH（2）。
* Neighbor Interface Address：源节点的邻居结点的地址。

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

在代码中，Hello消息实现如图3.3。

olsr\_protocol.h

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

图3.3 Hello消息的代码实现

在packet.h中的hello消息的组织图如图3.4。

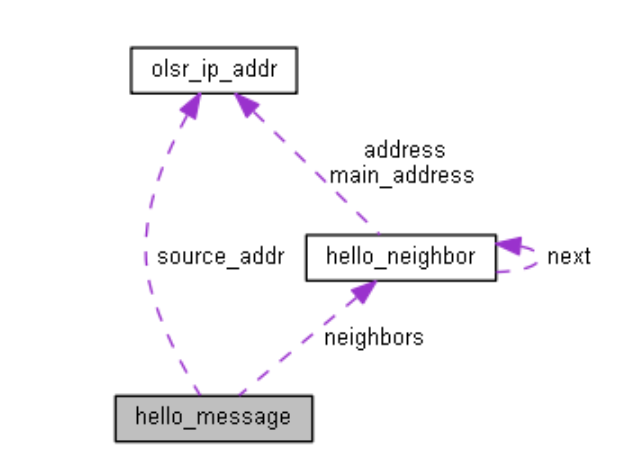


图3.4 Hello消息与其他结构的关系

### 3.1.3 MID消息

MID（Multiple Interface Declaration）消息是节点用来维护网络中多接口节点的主IP地址和其他接口地址之间关系的重要工具，也是维护3.2节中提到的MID信息表的重要通信工具。

|  |
| --- |
| OLSR interface address |
| OLSR interface address |
| OLSR interface address |
|  |

表3.3 MID消息结构

经3.1.1节中对OLSR数据包的分析可知，四种消息共享一类消息表头。MID消息对表头的设置是将MessageType设置为MID\_MESSAGE。Time应设置为255(最大值)，以扩散消息传入整个网络，Vtime相应地设置MID\_HOLD\_TIME。

如表3.3，数据包结构解析：

* OLSR interface address：MID消息中的message部分如图所示主要存储的是节点其他接口的地址信息。

图3.5为代码中MID消息的数据结构。

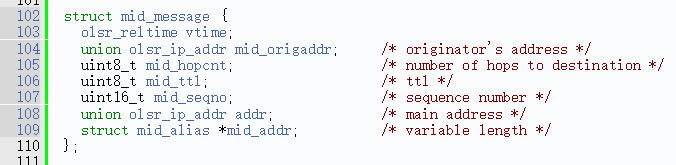


图3.5 MID消息代码实现

图3.6为MID消息的组织图。

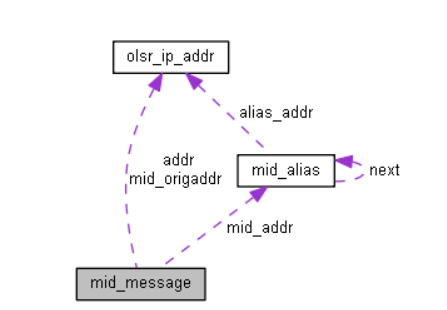


图3.6 MID消息组织图

### 3.1.4 TC消息

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



图3.7 结点和它的MPR结点

如图3.7所示，节点 B、C、D和 E 是节点 A的邻居节点，其中节点 B、C和D又是节点A的 MPR节点。当节点 B、C、D和E 收到节点A发送来的TC 控制消息时首先，它们都需要判断自己是不是节点 A的 MPR节点。节点 B、C和D 发现自己是节点A的 MPR 节点，它们再判断TC控制消息是否是最新的，如果是，则转发该TC 控制消息，否则丢弃该TC控制消息；而节点E发现自己不是节点A的MPR节点，则不转发该TC控制消息。

1 2 3 4

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

表3.4 TC消息结构

图3.4是TC消息格式，在消息首部MessageType中为TC\_MESSAGE，消息首部的Time应该设为255（最大值）目的是为了让消息能够在网络中更充分的传播。

数据包结构解析：

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

图3.8为消息在代码中的实现：

Packet.h

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

图3.8 TC消息代码实现

图3.8代码是一个完整的TC消息，包含了消息类型和Vtime。

图3.9是TC消息与其他的结构的关系图。

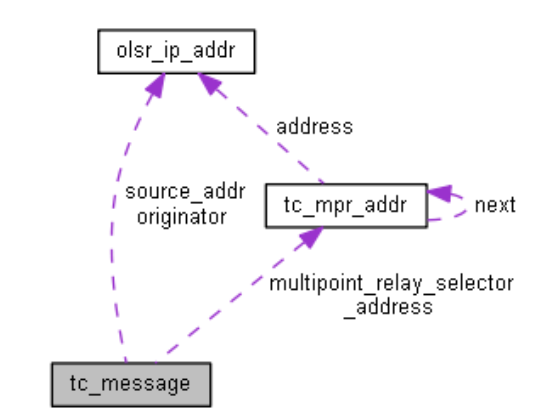


图.3.9 TC消息的关系图

### 3.1.5 HNA消息

包含主机的网络节点需要定期生成HNA消息，并向网络中其它节点发送该消息，说明该网络节点下的主机能否接入到其它网络中。

HNA消息的结构如表3.5所示。

|  |
| --- |
| Network Address |
| Netmask |
| Network Address |
| Netmask |
| ··· |

表3.5 HNA消息结构

数据包结构解析：

* Network address：网络地址
* Netmask：子网掩码

每个节点都维护一个信息：哪些节点可能是关联主机和网络的“网关”，并将他们记录为“关联组”。

在HNA消息机制中，通过网络向一个地址序列宣布可达性，显然比向一个个主机地址宣布要来的高效。

TC消息和HNA消息之间的一个重要的区别就是，TC消息可以在之前的信息中含有取消指令，而HNA消息只会在消息失效才被删除。

图3.10是此初始化HNA数据集函数分析

hna\_set.c

63 int olsr\_init\_hna\_set(void)

64 {

65 int idx;

66

67 for (idx = 0; idx < HASHSIZE; idx++) {

68 hna\_set[idx].next = &hna\_set[idx];

69 hna\_set[idx].prev = &hna\_set[idx];

70 }

71

72 hna\_net\_timer\_cookie = olsr\_alloc\_cookie("HNA Network", OLSR\_COOKIE\_TYPE\_TIMER);

73

74 hna\_net\_mem\_cookie = olsr\_alloc\_cookie("hna\_net", OLSR\_COOKIE\_TYPE\_MEMORY);

75 olsr\_cookie\_set\_memory\_size(hna\_net\_mem\_cookie, sizeof(struct hna\_net));

76

77 hna\_entry\_mem\_cookie = olsr\_alloc\_cookie("hna\_entry", OLSR\_COOKIE\_TYPE\_MEMORY);

78 olsr\_cookie\_set\_memory\_size(hna\_entry\_mem\_cookie, sizeof(struct hna\_entry));

79

80 return 1;

81 }

图3.10 初始化HNA数据集函数代码实现

olsr\_init\_hna\_set()函数功能：初始化HNA数据集。

72-74:hna\_net\_timer\_cookie记录的是数据包头部的Vtime值，类似的，下面的cookie都是对包内信息的记录。

75-78: olsr\_cookie\_set\_memory\_size记录的是发消息的表，根据HNA消息的接收消息的规则，对于通信过的节点和未通信过的节点通信方式不同。

## 3.2 信息表类数据结构

### 3.2.1 链路状态信息表

本地链路信息表用于存储该节点和邻居节点的链路信息，该信息表通常与链路感知与邻居探测有关

表3.6为链路信息结构。

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

表3.6 链路信息结构

信息表结构解析：

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

### 3.2.2 邻居集信息表（一跳邻居表中包含两跳邻居表）

|  |  |  |  |
| --- | --- | --- | --- |
| N\_neighbor\_main\_addr | | N\_status | N\_willingness |
| Is\_mpr | Was\_mpr | Neighbor\_2\_list | [neighbor\_2\_nocov](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\structneighbor__entry.html#aeb324de89af103041cc6756113fb6e6d) |

表3.7 邻居信息表结构

表3.7表示该结点的一条一跳邻居信息。

信息表结构解析：

* N\_neighbor\_main\_addrL：表示一跳邻居的主地址
* N\_status：表示邻居的类型，在OLSR里取值有三种分别是MPR\_NEIGH,SYM\_NEIGH,ASYM\_NEIGH
* N\_willingnes：表示邻居结点愿意为邻居结点转发分组的愿意程度，如果为WILL\_ALWAYS,那么该结点一定会被其他结点选择为MPR结点，具有WILL\_NEVER意愿的节点将永远不会被任何节点选择为MPR,必须始终选择愿意使用WILL\_ALWAYS的节点作为MPR。默认情况下，节点应该声明愿意使用WILL\_DEFAULT。
* Is\_mpr：表示现在此节点是否是MPR节点。
* Was\_mpr：表示此节点曾经是否是MPR节点，用来感知mpr的变化。
* Neighbor\_2\_list：指向的是此邻居节点的所有邻居节点，即所谓的两跳邻居节点表。其数据结构为neighbor\_2\_list\_entry.
* Neighbor\_2\_nocov：表示的是此邻居节点覆盖的两跳邻居节点个数。

图3.11为一跳邻居信息表组织图。

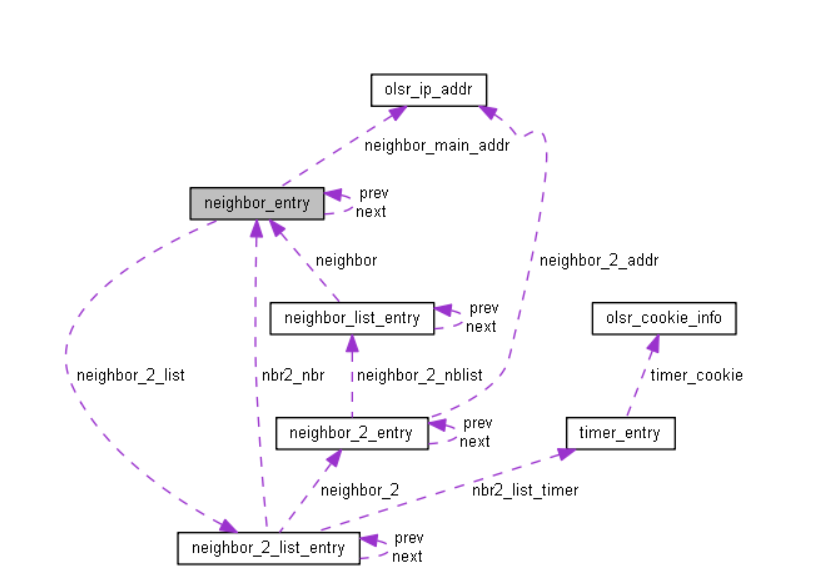


图3.11 一跳邻居信息表组织图。

neighbor\_2\_list\_entry数据结构

表结构如表3.8

|  |  |  |
| --- | --- | --- |
| Nbr2\_nbr | Neighbor\_2 | Nbr\_list\_timer |

表3.8 表结构

信息表结构解析：

* Nbr2\_nbr：反向指向其关联的一跳节点。
* Neighbor\_2：表示与 nbr2\_nbr有对称链路的两跳邻节点的地址，其类型neighbor\_2\_entry数据结构。
* Nbr\_list\_timer：表示表项到期必须被移除的时间。

图3.12为此数据结构组织图。

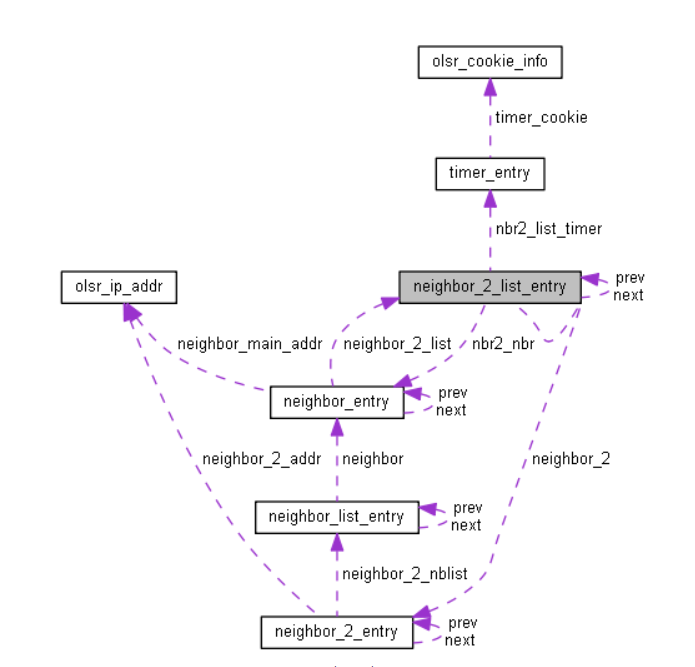


图3.12 数据结构组织图

Neighbor\_2\_entry数据结构

表结构如表3.9。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| neighbor\_2\_addr | mpr\_covered\_count | processed | neighbor\_2\_pointer | neighbor\_2\_nblist |

表3.9 数据结构

信息表结构解析：

* Neighbor\_2\_addr：此两跳节点的IP地址。
* mpr\_covered\_count：表示此两跳节点关联的节点的一跳邻居个数，用于MPR计算。
* processed：用于MPR选择计算
* neighbor\_2\_pointer：此两跳节点连接有多少个邻居节点
* neighbor\_2\_nblist：此两跳节点的相关路径花费

图3.13为此数据结构组织图。

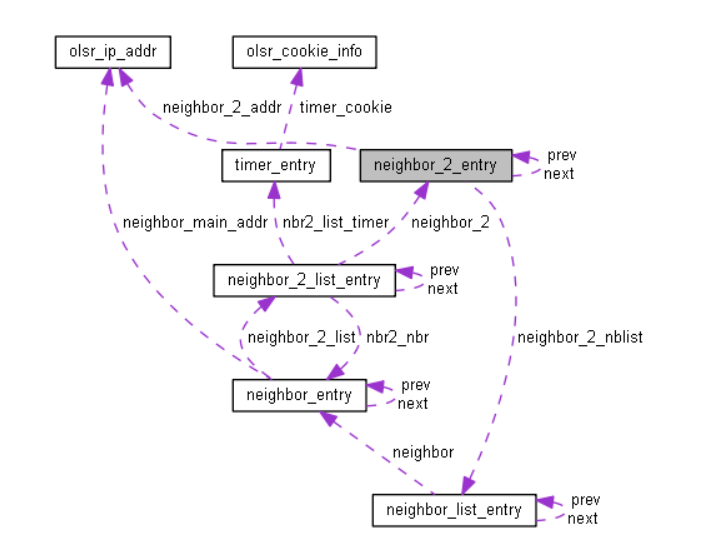


图3.13 数据结构组织图

### 3.2.3 MPR集信息表

MPR Selector Set是每个运行OLSR路由协议的节点保存的一个存储被本节点选为MPR节点的一跳邻居节点的信息集合。每个MPR节点的信息被保存在一个mpr\_selector的结构体中。其结构组成如表3.10。

|  |  |  |  |
| --- | --- | --- | --- |
| MS\_main\_addr | MS\_timer | next | pre |

表3.10 MPR信息表结构

信息表结构解析：

* MS\_main\_addr：类型为定义在<[olsr\_types.h](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\olsr__types_8h_source.html)>文件中的联合体olsr\_ip\_addr，主要存储MPR节点的主IP地址（包括IPv4和IPv6的地址）；
* MS\_timer：类型为定义在<[scheduler.h](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\scheduler_8h_source.html)>文件中的结构体timer\_entry的指针，主要明确了此节点信息失效和被强制移除的时间；
* next 和 prev ：都是mpr\_selector结构体的指针类型，主要记录此节点信息在整个MPR Seclector节点信息集中的前后节点地址信息，以便对MPR Selector节点信息集进行增、删、改、查操作。

图3.14为此数据结构组织图。

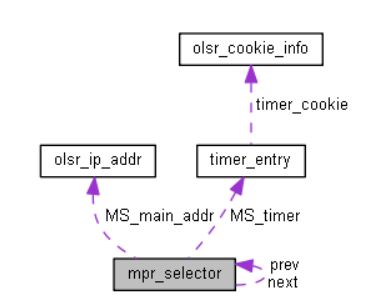


图3.14 数据结构组织图

### 3.2.4 MID信息表

在MANET网络中很多节点除了主IP地址外，还有多个网络接口。对于只有一个接口的节点，主IP地址与接口地址之间的关系无关紧要。但是，对于多接口的节点来说，它们之间的关系就显得尤为重要。因此，OLSR协议规定网络中节点通过向网络中其他节点洪泛MID消息来通知自己的多接口信息。并且每个节点通过获取网络中的MID消息在本节点存储其他节点的多接口信息，存储这些信息的容器就是MID Set。

MID Set中的每个节点信息保存在mid\_entry结构体中，其结构组成如表3.11。

|  |
| --- |
| main\_addr |
| aliases |
| prev |
| next |
| mid\_timer |

表3.11 MID信息表结构

信息表结构解析：

* main\_addr：类型为定义在<[olsr\_types.h](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\olsr__types_8h_source.html)>文件中的联合体olsr\_ip\_addr，主要存储网络节点的主IP地址（包括IPv4和IPv6的地址）。
* aliases：类型为<mid\_set.h>文件中的结构体mid\_address的指针, 意思是别名，主要存储此网络节点的其他的所有接口地址，指向一个接口地址列表。结构体mid\_address 是与mid\_entry 一同在同一文件中定义的结构体，其结构组成如表3.12。

|  |
| --- |
| alias |
| main\_entry |
| prev |
| next |
| next\_alias |
| vtime |

表3.12 数据结构

数据结构解析：

* alias：类型为定义在<[olsr\_types.h](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\olsr__types_8h_source.html)>文件中的联合体olsr\_ip\_addr，主要存储网络节点的其他接口地址信息（包括IPv4和IPv6的地址）。
* main\_entry：类型为结构体mid\_entry的指针类型，主要指向此接口信息对应节点的主IP地址信息条目。
* prev 和next：类型为mid\_address的指针类型，主要记录此接口信息条目在整个aliases接口信息集中的前后节点地址信息，以便对aliases接口信息集进行增、删、改、查操作。
* next\_alias：类型为mid\_address的指针类型。
* vtime：类型为无符号4字节的整数，接口信息条目根据此变量来计算自己的有效时间。

图3.15为mid\_entry的组织图，图3.16为mid\_address的组织图。

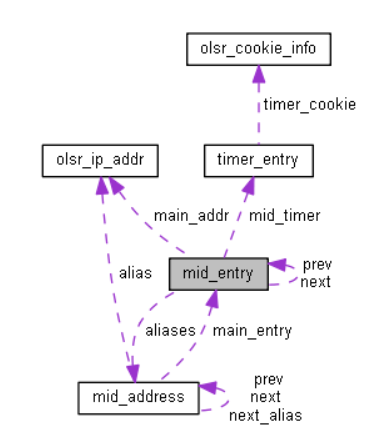


图3.15 mid\_entry的组织图

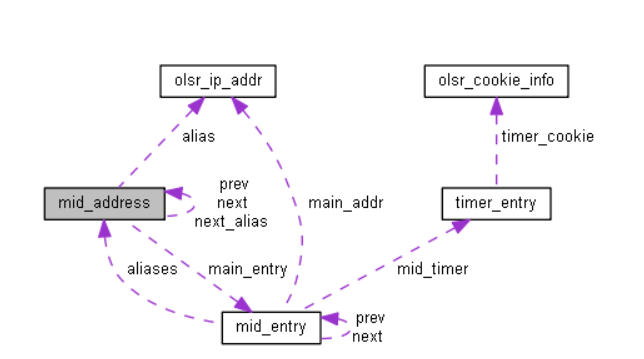


图3.16 mid\_address的组织图

### 3.2.5 拓扑信息表

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

表3.13 拓扑信息表结构

每个结点都会维护一张拓扑表，表3.13中记录了从TC消息中获得网络拓扑信息。图为拓扑信息表结构。

信息表结构解析：

* T\_dest\_addr：MPR选择结点的地址，表示该结点已经选择结点T\_last作为MPR。
* T\_last：被T\_dest选为MPR结点的结点地址。
* T\_seq：表示T\_last已经发布了它保存的序列号为T\_seq的MPR Selector集合的控制信息。
* T\_time：表项的保持时间，过期后就失效。

### 3.2.6 路由表

网络中每个节点维护一个路由表，表中保存了节点到网路中所有可达目的节点的路由，对于路由已知的网络中的每一个目的地，表项被存储在路由表中，所有路由未到达或部分已知的表项不被记录在表中。表项格式如表3.14所示：

|  |  |  |  |
| --- | --- | --- | --- |
| R\_dest\_addr | R\_next\_addr | R\_dist | R\_iface\_addr |

表3.14 路由表结构

信息表结构解析：

* R\_dest\_addr：路由目的节点地址。
* R\_next\_addr：路由的下一跳节点地址。
* R\_dist：本节点到目的节点的距离。
* R\_iface\_addr：表示下一跳节点通过本地接口R\_iface\_addr到达。

### 3.2.7 重复消息信息表

消息处理过程中，难免因为各种原因会收到冗余数据包，为了防止节点重复处理相同的消息，降低路由协议的开销。OLSR路由协议为每个节点维护一个重复信息表，表中记录每个消息的重复信息，为防止重复处理相同信息功能提供支持。此重复信息表存储在一个平衡二叉树中。

表3.15是重复信息表项的数据结构。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Avl | ip | seqnr | too\_low\_counter | Array | Valid\_until |

表3.15 重复信息表结构

信息表解析：

* Avl：表示此表项在对应的平衡二叉树节点。
* IP：表示发送消息的源地址。
* Seqnr：表示收到消息的序列号。
* too\_low\_counter：表示的是此消息是否被转发。
* Array：记录的是收到此消息的一系列端口地址。
* Valid\_until：表示的是此表项的有效时长。

4 OLSR工作流程

尽管我们在第一章已经对OLSR路由协议做了一个简要概述，但是主要还是停留在对OLSR各种功能的离散描述上，不能具体了解OLSR路由协议在实际运行中的具体流程。本章我们基于整体OLSR路由协议代码文件，详细分析一下OLSR路由协议实际运行时的工作流程。

OLSR路由协议以main函数为主入口函数，主要完成路由协议运行后的一系列基本工作，比如初始化、中断处理函数和读取外部参数等等。通过中断信号处理来完成对接收数据包的处理和周期性发送数据包的工作。三种处理过程相辅相成，共同完成OLSR路由协议的多种功能，最终实现网络路由的目的。

下面我们针对这三个工作流程进行详细分析。

## 4.1 main函数的分析

Main函数是OLSR路由协议的主入口函数，主要功能是协议运行前的基本准备工作、处理交互输入参数、于网络建立接口、初始化所有数据结构和设定信号处理程序。它是路由协议得以正常运行的基础，同时也是协调OLSR路由协议所有功能已完成最后网络路由目的的“指挥官”。图4.1是main函数内部主要处理流程。



图4.1 main函数内部主要处理流程

## 4.2 接收数据包处理过程

OLSR路由协议通过信号处理，来完成接收数据包的处理过程。对接收数据包处理过程主要分为两大部分，对数据包的解析和对数据包的转发操作。



图4.2 接收数据包处理工程主要函数的调用图

如图4.2是接收数据包处理工程主要函数的调用。根据第三章的分析可知，OLSR路由协议的四种消息均被包裹在一个OLSR数据包中，通过OLSR数据包的转发和广播来完成节点间消息的通信。因此，OLSR路由协议当收到一个OLSR数据包后，首先系统调用olsr\_input()函数对收到OLSR数据包进行处理,然后此函数将数据包传递给parse\_packet()函数，完成对OLSR数据包的进一步处理。此函数主要完成两部分功能，一是循环匹配消息类型，并调用对应处理函数（如图所示四种消息处理函数）来完成对不同消息的处理，以及各种功能的实现。二是根据上述处理结果选择是否调用数据包转发函数，如果调用，函数则会对数据包进行判断并转发。

下面是对此部分涉及到的函数和结构的分析。

* Parse\_functions 全局变量：类型为结构体[parse\_function\_entry](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\structparse__function__entry.html)的指针，此变量是一个功能表，主要存储消息处理函数。在main函数初始化包解析器时，将四种消息处理函数加入列表。表4.1解释了其中记录的四种包处理函数以及其对应完成的功能。

|  |  |
| --- | --- |
| 消息处理函数 | 完成功能 |
| olsr\_input\_hello() | 调用函数更新Mid\_set，维护节点多接口信息。 |
| olsr\_input\_tc() | 调用函数更新（两跳范围内）邻居表，维护节点邻居信息；更新MPR表，维护节点MPR集信息。 |
| olsr\_input\_mid() | 调用函数更新hna\_set，维护节点网络信息 |
| olsr\_input\_hna（） | 调用函数更新路由表，维护节点路由信息 |

表4.1 四种包处理函数以及其对应完成的功能

* olsr\_input()函数

void olsr\_input (int fd, void\* data\_\_attribute\_\_(unused), unsigned int flags \_\_attribute\_\_(unused) )

从套接字处理OLSR数据。读取数据，设置wich接口接收到消息，发送IPC(如果使用)，并将数据包传递给parse\_packet()。

* <parser.c> parse\_packet()

void parse\_packet (struct olsr \* olsr, int size, struct interface \* in\_if,

union olsr\_ip\_addr \* from\_addr )

处理新收到的OLSR包。检查类型和必要的转换，并调用相应的函数来处理信息。

* <olsr.c> olsr\_forward\_message()

int olsr\_forward\_message ( union olsr\_message \* m, struct interface \* in\_if,

union olsr\_ip\_addr \* from\_addr )

检查是否要转发消息，包括TTL的检查、是否来自节点MPR节点和IP地址有效性等；并在转发前更新数据包参数，包括TTL和跳数的更新等，最后转发消息。输入参数为OLSR数据包，进入接口信息，源地址信息。

## 4.3 周期性发送数据包处理过程

经过分析OLSR路由协议，可知OLSR路由协议在调用shutdown()函数时产生发送HELLO消息和TC消息，通过信号处理程序，每隔一定周期，发送特定信号，迫使系统调用shutdown函数，而此函数又调用shutdown\_message()函数，shutdown\_message()函数又调用两种消息的产生函数，生成发送两种消息。

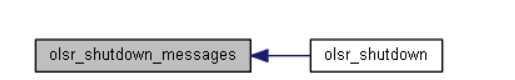


图4.3 shutdown()函数调用关系

而MID 消息和HNA消息则通过信号处理程序，周期性调用消息生成函数，产生和发送两种消息。

这样OLSR路由协议就完成了周期性发送四种消息的功能。表4.2是对四种消息产生函数的解释。

|  |  |
| --- | --- |
| 消息产生函数 | 完成功能 |
| Generate\_hello() | 构造hello消息并发送 |
| Generate\_tc() | 构造tc消息并发送 |
| Generate\_mid() | 构造mid消息并发送 |
| Generate\_hna() | 构造hna消息并发送 |

表4.2 四种消息产生函数

5 OLSR重点功能分析

## 5.1 邻居信息表的生成

### 5.1.1 链路状态感知

在介绍链路感知之前首先我们需要明确一下一条链路应当包含哪些信息，表5.1是结点存储的关于链路的信息。

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

表5.1 结点存储的关于链路的信息

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

我们该怎样确定一条链路的具体状态呢？具体规则如下：

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

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

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

链路感知部分主要由四个部分的代码完成，如图5.1所示。



图5.1 链路感知函数调用关系

Check\_link\_status()函数通过查找收到的Hello消息查看链路的状态，add\_link\_entry()函数为链路集中添加一条新的链路信息，lookup\_link\_entry()根据指定的IP地址在链路集中查找一条链路信息，update\_link\_entry()根据收到的hello消息更新新的链路信息。如图5.2我们来看看链路感知机制是怎样的。

link\_set.c

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

图5.2 链路感知机制代码实现

7：通过给定的信息查找链路，如果存在返回该链路；如果不存在则新建立一条链路并返回

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

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

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

* 如果链路类型是LOST\_LINK类型那么

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

* 或者链路类型为：SYM\_LINK 或者ASYM\_LINK类型那么

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

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

link\_set.c

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

图5.3 代码实现

10-11 如果在链路集合中找到链路，则返回

29-40 如果没有一条链路信息为

L\_neighbor\_iface\_addr = Source Address

那么就建立一条新的链路信息

L\_neighbor\_iface\_addr = Source Address

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

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

L\_time = current time + validity time

如图5.4我们再看看关于update函数的调用模型。

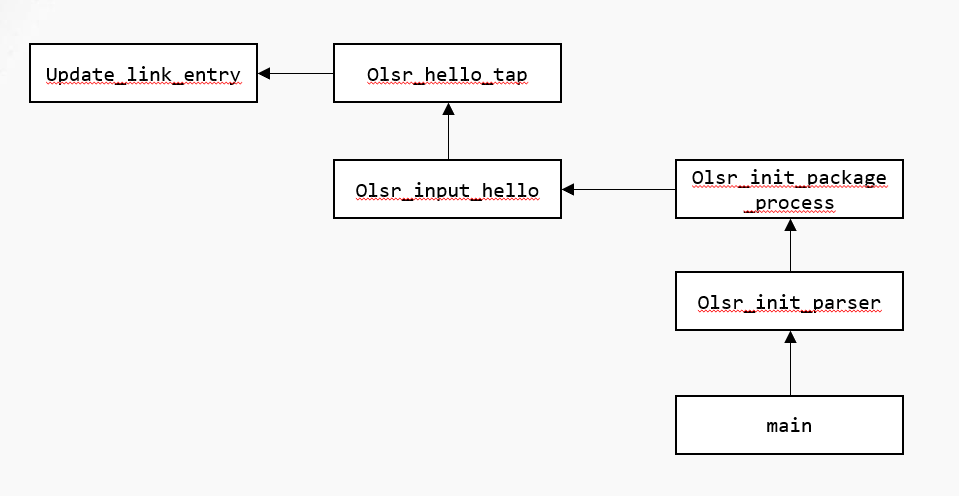


图5.4 函数调用

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

### 5.1.2 邻居探测

表5.2是结点存储的关于邻居集的信息。下面是一条存储的邻居信息，各项含义都已经在上述数据机构部分已经描述过了

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

表5.2 邻居集数据结构

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

在任何情况下，链路与邻居之间的对应关系如下定义：

一条链路集合的有关邻居需要满足的是

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

一个邻居的相关链路需要满足：

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

邻居检测主要由三个部分组成：创建邻居，更新邻居，移除邻居。如图5.4



图5.4 函数调用

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

Neighbor\_table.c

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

图5.5 代码实现

11-13 在邻居集中查找是否有满足下列条件的邻居：

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

如果有就直接返回，如果没有的话就会创建出一个新的邻居元组

21-30 建立一个新的邻居元组，并对其各项进行赋予初值

Link\_set.c

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

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

图5.6 代码实现

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

关于邻居探测涉及到的函数调用如图5.7所示

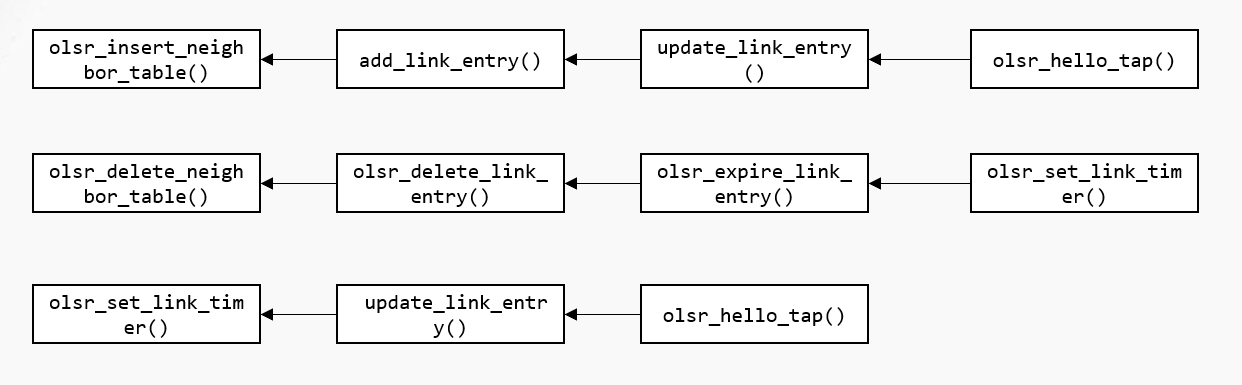


图5.7 函数调用

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

下面简单介绍一下关于二跳邻居表的更新过程，更新机制与一跳邻居机制相似。

二跳邻居表的信息结构如表5.3

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

表5.3 结构

2跳邻居集的维护机制也是周期交换hello消息

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

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

* 如果Hello消息中的地址为收到消息的主地址，那么跳过该地址
* 如果上述条件不成立，那么建立一个二跳邻居元组

N\_neighbor\_main\_addr = Originator Address;

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

N\_time = current time + validity time

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

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

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

## 5.2 MPR集的生成

MPR洪泛机制是OLSR路由协议的最大特点,也是此路由协议较之传统链路状态路由协议的最大改进之处. OLSR路由协议不再像传统路由协议一样广播发送自己的携带自己链路状态的数据包,而是选择邻居节点中的其中一部分节点作为MPR节点,成为转发自己数据包的对象,而且只保存MPR节点与本节点的链路状态信息.这样的洪泛机制大大降低了路由协议的开销.其中,MPR覆盖率(MPR节点占总邻居节点的比率)是影响路由协议开销的关键因素,他通过影响记录链接的数量、记录链接的节点数量和MPR泛洪机制的效率来影响开销.所以如何使MPR覆盖率最小,即在节点邻居节点中选择最少的MPR节点显得尤为重要.所以本节主要分析MPR节点的选择功能.

OLSR路由协议通过维护1.1中我们提到的MPR Selector信息表来存储节点的MPR节点集.MPR节点的选择的两条原则为: 第一,指定节点应该选择尽可能小的MPR集，以减少协议开销。第二,所有严格的2跳节点必须至少通过一个MPR节点到达,即经选择的MPR节点洪泛出去的数据包应覆盖到指定节点所有的2跳邻居节点。

虽然一般来说，最小MPR集提供的开销最小，但在某些情况下，开销可以用其他收益来交换。例如，一个节点可能会决定增加它的MPR覆盖率，如果它观察到它的邻居信息库由于移动性而发生了许多变化.在这里我们不考虑这种情况.

### 5.2.1 MPR选择计算中用到的重要参数

根据第三章数据结构的整理，我们了解到MPR计算用到的一些重要参数，这里再重申一遍，以便MPR选择算法的阅读理解。

* willingnes：表示邻居结点愿意为邻居结点转发分组的愿意程度，如果为WILL\_ALWAYS,那么该结点一定会被其他结点选择为MPR结点，具有WILL\_NEVER意愿的节点将永远不会被任何节点选择为MPR,必须始终选择愿意使用WILL\_ALWAYS的节点作为MPR。默认情况下，节点应该声明愿意使用WILL\_DEFAULT。
* Is\_mpr：表示现在此节点是否是MPR节点。
* Was\_mpr：表示此节点曾经是否是MPR节点，用来感知mpr的变化。
* mpr\_covered\_count：表示此两跳节点关联的节点的一跳邻居个数，用于MPR计算。
* processed：用于MPR选择计算

### 5.2.2 MPR选择算法

OLSR中MPR选择算法的大致思想是：

* 先选择邻居节点中willingness值为WILL\_ALWAYS的节点为MPR
* 然后根据willingness值将剩余邻居节点按降序依次选择进MPR节点集。
* 若是willingness值相同优先选择mpr\_covered\_count为1的节点（即mpr\_covered\_count值最小节点）
* 若mpr\_covered\_count不为1，则优先选择覆盖两跳邻居节点最多的节点。
* 不断重复2-3步，直到邻居节点全部覆盖两跳邻居节点。
* 初步选择完毕后，要对MPR节点集进一步优化。根据willingness值将MPR节点按升序一次检查是否存在多余节点（即其所有两跳邻居节点已经被其他节点覆盖），删除多余节点。得到最小MPR集

图5.8为路由协议中olsr\_calculate\_mpr()函数的程序流程图。



|  |
| --- |
| 图5.8 olsr\_calculate\_mpr()函数的程序流程图  图5.9 olsr\_calculate\_mpr()函数调用关系图 |

如图5.9，此函数调用了九个函数进行MPR计算。下面对这九个函数进行分析。

* <mpr.c>

static void olsr\_clear\_mprs (void )

此函数的功能主要是将邻居表中一跳节点的MPR标记位初始化(is\_mpr和was\_mpr),并且将两跳邻居节点的mpr\_covered\_count初始化为0。

* <mpr.c>

static uint16\_t olsr\_calculate\_two\_hop\_neighbors (void )

此函数功能为计算此节点的两跳邻居的数量。

* <mpr.c>

static uint16\_t add\_will\_always\_nodes (void )

此函数功能将所有willingness值为WILL\_ALWAYS的节点加入到MPR节点集中。返回值为加入的节点个数。

* <mpr.c>

Static struct neighbor\_2\_list\_entry \* olsr\_find\_2\_hop\_neighbors\_with\_1\_link (int willingness)

此函数的作用是根据给定的willingness值,来得到所有具有此willingness并且只与本节点的一个邻居相连的所有两跳邻居节点.

函数输入参数为指定的willingness的值. 返回值为所有符合条件的两跳邻居节点集。

* <mpr.c>

static int olsr\_chosen\_mpr ( struct neighbor\_entry \* one\_hop\_neighbor,uint16\_t \* two\_hop\_covered\_count )

此函数的功能是将指定一跳邻居节点选择进MPR集,并且更新MPR节点覆盖的两条邻居节点计数器。

函数输入参数为一跳邻居节点和MPR节点覆盖的两条邻居节点计数器。

* <mpr.c>

static struct neighbor\_entry \* olsr\_find\_maximum\_covered ( int willingness)

此函数的作用是得到符合给定willingness值的不是MPR节点的一跳邻居节点集中覆盖两跳邻居节点最多的节点。

函数输入参数为指定的willingness的值. 返回值为所有符合条件的一跳邻居节点。

函数流程图如图5.10。



图5.10 函数流程图

* <mpr.c>

static void olsr\_optimize\_mpr\_set (void )

此函数的功能是进一步优化已得到的MPR节点集,使其含有最少的节点数量。

函数流程图如图5.11。

图5.11 函数流程图

* <link\_set.c>

void signal\_link\_changes(bool val)

此函数的作用是设定link\_changes的值,来显示是否发生链路变化。

函数输入参数是bool值表示是否发生链路状态变化。

* <mpr.c>

static int olsr\_check\_mpr\_changes (void )

此函数的作用是检查MPR节点集是否发生变化。

## 5.3 多端口（MID）节点功能处理

在无线自组网中，有些节点的多个接口都运行OLSR路由协议。对于只有一个端口运行OLSR路由协议的端口来说节点的主IP地址和接口地址之间的关系十分微小，但是，对于有多个接口运行OLSR路由协议的节点来说，节点主IP地址与接口地址之间的关系就显得尤为重要，他们之间的关系直接影响到路由表的计算问题。因此，OLSR协议设置多端口（MID）节点功能处理部分来帮助节点时刻了解网络中多端口节点的主IP地址与端口地址之间的关系，为路由表的计算提供支持。

此部分功能涉及到的主要数据结构为1.1和1.2提到的MID消息和MID信息表。此功能利用MPR洪泛机制周期性广播MID消息以通知网络中所有节点本节点的主IP地址和接口信息，并且利用收到的MID消息在节点内部更新维护一个记录其他节点主IP地址和接口信息关系的MID信息表，以便为其他功能提供支持（如路由表的计算）。

此部分功能主要分为三个部分，分别是MID消息的产生、MID消息的处理和MID消息的转发。下面将根据这三部分内容进行详细解析。

### 5.3.1 MID消息的产生

正如4.3节介绍的那样，MID消息通过信号处理程序调用MID生成和发送函数进行MID消息的周期性发送。图5.12为MID生成和发送函数。

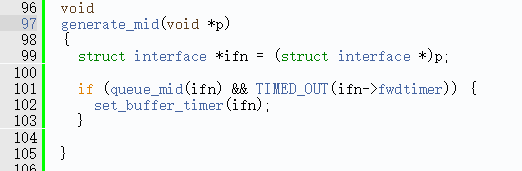


图5.12 MID生成和发送函数

100-101：生成MID消息的节点将生成的MID消息包放进一个队列中，接 收消息的节点依次读取队列中的MID消息，当读取到某一消息时，首先判断该 消息是否过期，若过期，则不予读取，直接丢弃该数据包。

### 5.3.2 MID消息的处理

节点的MID信息表主要通过收到的MID消息进行更新维护。所以MID消息的处理主要是对MID信息表的更新处理。大致包括MID信息的提取、MID信息有效性的验证、利用MID信息更新MID信息表、利用MID信息更新路由前缀表和MID信息的销毁几个处理部分。

图5.12为函数中各处理部分的关系和其调用的函数：

图5.12 函数流程图

下面是各函数的详细介绍：

* ＜rebuild\_packet.c＞

void mid\_chgestruct　(struct mid\_message \* mmsg,　const union olsr\_message \* m )

此函数的作用是提取OLSR数据包中的MID信息，并存储在内部MID信息结构中，以便下一步处理。

Mmsg指向一个mid\_message类型的指针，这是一个OLSR内部存储MID节点多接口信息的数据结构。M是一个指向olsr\_message类型的指针，指向的是节点收到的OLSR数据包。

* <link\_set.c>

int check\_neighbor\_link ( const union olsr\_ip\_addr \* int\_addr)

此函数的主要功能为在函数邻居链路状态信息表中查找指定地址的链路状态.

函数参数为指定地址的IP地址

* <net\_olsr.c>

bool olsr\_validate\_address (const union olsr\_ip\_addr \* adr)

函数主要功能为判断IP地址是否有效.

函数参数为指定IP地址

* ＜mid\_set.c＞

Int [olsr\_update\_mid\_table](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\mid__set_8c.html#a664b31b4f35eacc80ca33e3c47233111)(const union [olsr\_ip\_addr](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\unionolsr__ip__addr.html) \*adr, [olsr\_reltime](file:///D:\%E5%AD%A6%E4%B9%A0\%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE%E6%A0%88\html\mantissa_8h.html#a2982f3029e50006108ca03c607781240) vtime)

用于更新MID中对应条目的定时器，ａｄｒ指向源地址，vtime是用于计算有效生存期的时间变量

* ＜mid\_set.c＞

void insert\_mid\_alias(union olsr\_ip\_addr \* main\_add,　const union olsr\_ip\_addr \* alias,　olsr\_reltime　vtime )

此函数功能是在MID信息表中的指定节点条目插入其对应的别名地址，如果不存在指定源地址条目，则新建条目。

函数参数ｍａｉｎ＿ａｄｄ为主IP地址，ａｌｉａｓ为别名地址，vtime用于计算条目生存周期。

* ＜mid\_set.c＞

union olsr\_ip\_addr\* mid\_lookup\_main\_addr ( const union olsr\_ip\_addr \* adr)

此函数功能是在反向MID信息表中查找别名对应的主IP地址。这就是多接口地址的反向解析功能.

函数参数ａｄｒ为别名的IP地址。

* ＜routing\_table.c＞

struct rt\_path\* olsr\_insert\_routing\_table (union olsr\_ip\_addr\* dst,　int plen,　union olsr\_ip\_addr \* originator,

int origin )

此函数的主要功能是在路由前缀树中加入别名信息，如果不存在主IP地址对应的节点，则新创建一个条目。

函数参数为ｄｓｔdst是目的地IP地址，ｐｌｅｎ是前缀长度，originator为源IP地址。

* ＜mid\_set.c＞

static void olsr\_prune\_aliases ( struct mid\_message \*message)

此函数主要功能为删除MID信息表中此MID信息源地址对应条目中没有出现在此MID消息中的别名信息。

函数参数为类型函数参数为类型为内部MID信息存储结构mid\_message的指针，它指向正在处理的MID消息。

* ＜packet.c＞

void olsr\_free\_mid\_packet ( struct mid\_message \* message)

此函数的主要功能是释放MID消息所占内存空间。

函数参数为类型为内部MID信息存储结构mid\_message的指针，它指向要被删除的MID消息。

主要处理算法流程如图5.13所示：

图5.13 函数流程图

### 5.3.3 MID消息的转发

节点通过广播和转发MID消息将其洪泛到整个网络,所以MID消息处理完成后要由节点转发出去。此过程已经再4.2节中详细分析过了，这里不再赘述。

### 5.3.4 多端口节点功能对OLSR中其他功能的支持

OLSR这种允许多端口节点存在的功能,为其协议中其他功能提供了支持.上文中提到的对路由表的计算支持,主要通过上文提到的olsr\_insert\_routing\_table()函数完成,为网络中节点添加多个路由前缀,以便路由计算。

另外, OLSR中需要使用“接口地址”的部分是链路感知。OLSR的其余部分对节点进行操作，这些节点由它们的“主地址”唯一标识(实际上，节点的主地址是其“节点id”——为了方便起见，它对应于其中一个接口的地址)。在一个只有单个接口节点的网络中，根据定义，一个节点的主地址等于该节点的接口地址。在一个公共OLSR区域内运行着多个接口节点的网络中，需要能够将任何接口地址映射到相应的主地址。这一功能可以由上文提到的mid\_lookup\_main\_addr()函数解决,实现对主IP地址的反向查找。

反向查找的一般查找结果为:

* 如果根据别名地址反向查找到主IP地址,则返回主IP地址。
* 如果没有查找到,则默认端口地址为节点的主IP地址。

## 5.4 TC控制

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

TC消息机制分为两个部分：生成和处理。

Generate\_msg.c

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

图5.14 代码实现

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

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

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

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

接下来要介绍的是关于TC消息的处理过程：当一个结点收到了一条，首先根据消息首部的Vtime域计算其有效时间。然后按照图5.15的代码步骤更新

Tc\_set.c

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

图5.15 代码实现

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

Tc\_set.c

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

图5.16 代码实现

853-855: 对于在拓扑集中存在满足:

T\_last\_addr == originator address AND

T\_seq > ANSN,

直接丢弃该TC消息，因为消息顺序发生了错误。

859-875 只接受那些正确序列号的消息。

图5.17是关于tc消息处理过程的调用。

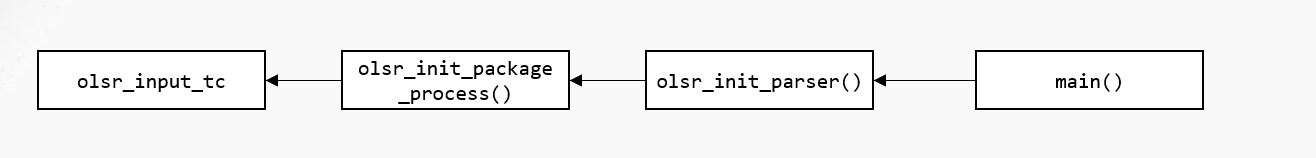


图5.17 函数调用

我们可以看到在tc消息的处理过程在包处理函数中被调用。

## 5.5 路由计算

节点通过TC消息的扩散获得全网拓扑图，再根据邻居表、两跳邻居表和拓扑表，独立地按照Dijkstra算法计算出路由表。每个节点都有一张路由表，通过路由表寻找路径信息。对于路由已知的网络中的每个目的地，将路由信息记录在路由表中。所有路由被破坏或仅部分已知的目的地的路由信息不被记录在表中。

每一条路由信息都包含信息目的地址、下一跳地址、总跳数、下一跳接口地址，格式如表5.4

|  |  |  |  |
| --- | --- | --- | --- |
| R\_dest\_addr | R\_next\_addr | R\_dist | R\_iface\_addr |

表5.4 结构

如果该节点所维护的这些信息表中的任何信息发生改变，则重新计算路由表以更新关于网络中的每个目的地的路由信息。即更新路由表的条件为：

* 邻居表变化，需要重新选择路由，根据路由计算结果更新路由表。
* 两跳邻居表变化，需要重新选择路由，根据路由计算结果更新路由表。
* 拓扑表发生变化，需要重新选择路由，根据路由计算结果更新路由表。

路由表的更新既不在网络中，也不在一跳邻居域中生成或触发任何消息。

操作系统的路由体系结构按功能可以分成两个部分。一部分是负责与其它节点交换信息，计算到其它节点的正确路由，称之为“路由功能模块”；一部分则是根据内核路由表，将需要发送到网络中的数据分组，通过正确的网络接口发送到下一跳节点，称为“转发功能模块”。这样，操作系统就可以在“转发功能模块”保持不变的情况下，通过修改“路由功能模块”，从而实现不同的路由协议。

OLSR 协议的实现通过端口号为698的UDP端口收发路由控制分组，然后维护邻居表，进行逻辑计算，最后生成路由表并反映到内核路由表中。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本节点时，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 相关数据结构

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 };

71

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 };

图5.18

67:路由选择时采用复合矩阵。

68:两个路由节点之间的花销。

69:两个路由节点之间的跳数。

74:下一跳网关。

75: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 };

图5.19

85-93: 每一个RIB节点都会有一个路由的接口，这个接口很重要，里面包含了最佳路径的下一个网关信息，而且是rt\_path\_tree的根，同样也包含了一个在所有路由信息里的一个最好的路径。rt\_dst包含了该信息的路由地址与前缀长度。rt\_tree\_node包含了val\_tree的一些信息。该结构体里面也包含了父节点、子节点、左右节点、表示一棵树的信息值。

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 };

图5.20

106-117: 该结构体包含了rt\_path的信息，接收到的rt\_path会被加入到路由信息表中，然后用于计算best\_path。

routing\_table.h

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 };

图5.21

182-194: 核心路由表的结构。核心路由表表项的添加和删除主要受目的地、网关（下一跳）和标志设置的影响。标志位的设置非常重要，决定着传输数据时，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 };

图5.22

OLSR协议中有三种路由类型，INT（内部路由信息）是由接收TC\_Message生成的，而MID类型的路由信息是由接收MID\_Message生成的，HNA路由信息是来自HNA公告。

### 5.5.2 路由表计算算法描述

每个节点维护一张保存到每个目的节点的路由表。路由表是基于本地链路信息库中所包含的信息和拓扑集。因此，任何这些数据集发生了变动，路由表就需要重新计算以更新到网络中各目的节点的路由。路由条目在路由表中按照如下格式被记录：

1. R\_dest\_addr R\_next\_addr R\_dist R\_iface\_addr

2. R\_dest\_addr R\_next\_addr R\_dist R\_iface\_addr

3. ,, ,, ,, ,,

准确地说，在以下这些数据集或库被检测到的情况下下，路由表需要更新：

* 链路集
* 邻居集
* 2跳邻居集
* 拓扑集
* 多接口关联信息库

为了找到从 给定源节点到较远节点R的路径，必须找到连接对（X , R ），然后是找到连接对（Y , X ），以此类推，直到发现节点Y就是源节点的邻居节点时结束连接对的查找。

为了使得到的路径是最优的，转发节点仅选择最小路径上的连接对。路径选择算法在拓扑图的基础上采用了多重Dijkstra算法。

在 OLSR 标准协议中，协议根据最小跳数建立每个节点的路由表。任意一个节点路由表的添加过程可分为三部分：首先，添加自己的邻节点进入路由表，即跳数h=1；其次，添加自己的两跳邻节点进入路由表，即h=2；最后，循环添加跳数等于h+1（h=2开始）的节点进入路由表。不考虑多接口时，具体过程描述如下：

* + 首先清除现有路由表中的所有表项。
  + 将具有对称链路的邻居作为目的地址，一个如下的新的表项加入路由表：
    - * R\_dest\_addr ==邻节点地址
      * R\_next\_addr ==邻节点地址
      * R\_dist==1
      * R\_iface\_addr ==本地地址
  + 对于任意一个两跳邻居节点，其两跳邻居集中必然存在着这样一个表项，其 N\_neighbor\_main\_addr 等于本地的一个一跳邻居且其willingness不为WILL\_NEVER 。则将每个两跳邻居加入路由表如下：
    - * R\_dest\_addr ==两跳邻居地址
      * R\_next\_addr ==已有的具有如下特征的路由表表项的R\_next\_addr：其R\_dest\_addr为该两跳邻居表项的N\_neighbor\_main\_addr
      * R\_dist==2
      * R\_iface\_addr ==已有的具有如下特征的路由表表项的R\_iface\_addr：其R\_dest\_addr 为该两跳邻居表项的N\_neighbor\_main\_addr
  + 将目的地为 h+1 跳的路由表项记录在路由表中，对于h的每个值，下面的程序必须被执行，h从2开始，每次加1，在迭代中没有新表项被记录时，执行停止。对于拓扑集中的每个拓扑表项，如果其T\_dest\_addr不符合于路由表中任何表项的R\_dest\_addr且其T\_dest\_addr符合于一个R\_dist为h的路由表项的R\_dest\_addr，则一个新的路由表项被记录如下：
    - * R\_dest\_addr==T\_dest\_addr
      * R\_next\_addr==已有的具有如下特征的路由表表项的R\_next\_addr：其 R\_dest\_addr==T\_last\_addr
      * R\_dist==h+1
      * R\_iface\_addr==已有的具有如下特征的路由表表项的R\_iface\_addr：其 R\_dest\_addr==T\_last\_addr

计算路由表之后，为节约内存空间，要删除没用于计算的拓扑表中的表项。否则，这些表项将会提供到同一目的节点的多条路由。

### 5.5.3 路由表计算

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 }

图5.23

函数功能：初始化路由表（初始化平衡二叉树）。

173:调用avl\_init( )初始化一个平衡二叉树。

174:用以检测每一个rt\_entry和rt\_path子树中过时的信息。

179-183: 为rt\_entry和rt\_path分配内存，创建相应的cookie。

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 }

图5.24

函数功能：从avl树里面找到一个地址的路由表条目，根据参数地址并配上设置里的最长前缀长度。在avl树里调用avl\_find函数从routingtree里面找到该地址的rt\_entry。

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 }

图5.25

546:先调用olsr\_locate\_tc\_entry函数根据源地址判断该tc\_entry是否可连接，因为tc\_entry作为所有路由的连接点，如果它是不可连接的，则在最短路径优先计算时它不会被考虑。

551-563: 首先检查该前缀是否已经存在一条路径。如果没有的话，则调用olsr\_alloc\_rt\_path函数创建一条该前缀和源地址的路径。

routing\_table.c

582 void

583 olsr\_delete\_routing\_table(union olsr\_ip\_addr \*dst, int plen, union olsr\_ip\_addr \*originator)

584 {

585 #ifdef DEBUG

586 struct ipaddr\_str dstbuf, origbuf;

587 #endif

588

589 struct tc\_entry \*tc;

590 struct rt\_path \*rtp;

591 struct avl\_node \*node;

592 struct olsr\_ip\_prefix prefix;

593

594 /\*

595 \* No bogus prefix lengths.

596 \*/

597 if (plen > olsr\_cnf->maxplen) {

598 return;

599 }

600

601 tc = olsr\_lookup\_tc\_entry(originator);

602 if (!tc) {

603 return;

604 }

605

606 /\*

607 \* Grab the rt\_path for the prefix.

608 \*/

609 prefix.prefix = \*dst;

610 prefix.prefix\_len = plen;

611

612 node = avl\_find(&tc->prefix\_tree, &prefix);

613

614 if (node) {

615 rtp = rtp\_prefix\_tree2rtp(node);

616 olsr\_delete\_rt\_path(rtp);

617

618 #ifdef DEBUG

619 OLSR\_PRINTF(1, "RIB: del prefix %s/%u from %s\n", olsr\_ip\_to\_string(&dstbuf, dst), plen,

620 olsr\_ip\_to\_string(&origbuf, originator));

621 #endif

622

623 /\* overload the hna change bit for flagging a prefix change \*/

624 changes\_hna = true;

625 }

626 }

图5.26

函数功能：从字典树中删除前缀，并重载HNA消息。

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 }

图5.27

函数功能：此函数用于为一条给定的rt\_path创建一个route entry，并将其插入到RIB树中和计算路由表。

215:每条路径都是周期性更新的，修改所维护的routingtree\_version值。

217-225:将相应的网关地址、接口地址、跳数和路径花销置为新接收到的值。

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 }

图5.28

函数功能：创建一个可用的路由条目，对提供的参数ip前缀分配一个路由条目空间，并做一些相应的初始化，并把该条目插入到avl树里。

234-237:申请内存空间并把空间内清零。

241-245:表示该入口为新分配入口，并把该入口的目的地址设置成为参数提供的入口地址。

247-251:把入口的树节点插入到整个路由表中，并初始化avl树。

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 }

图5.29

函数功能：对给定的rt\_path创建一个路由条目，并将它插入到全局RIB树中。

301-310:如果传入的tc\_entry为ROUTE\_COST\_BROKEN，或者传入的rtp的目的地址长度大于所设置的最大地址长度，则直接返回。

315-328:调用avl\_find( )函数检查传入的rtp的节点是否在路由表中，如果节点不在路由表的avl树中则重新分配一个节点，如果在就把节点类型从avl\_node类型转化成rt\_entry类型。

330-343:把新路由条目添加进avl树里，改变相应的参数，更新路径。

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 }

图5.30

函数功能：从路由表中删除rtp树，并把它从tc树里删除，然后改变路由表版本。

354-357:把rtp所指向的树节点从所在的树里删除，并把指向的树的根置空。

360-364:将其从字典树（前缀树）中移除，并解锁相应的tc\_entry。

367-372:把rtp所指向的树节点再从tc树里删除，释放cookie所占用的内存。

routing\_table.c

435 static bool

436 olsr\_cmp\_rtp(const struct rt\_path \*rtp1, const struct rt\_path \*rtp2, const struct rt\_path \*i netgw)

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 }

图5.31

函数功能：比较两条路由路径，如果第一条更优，则返回true，否则返回false。

445-451:比较两个路由路径的花销，花销小的路径更优。

453-459:如果两个路由路径花销相同，则比较两条路径的跳数，跳数少的更优。

461-464:如果两个路由路径的花销和跳数都相同，则比较两条路由的源地址，地址更小的更优。

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 }

图5.32

函数功能：运行最优路径，首先得到第一条目，然后遍历所有条目，找到一条最优路径并修改当前网关路径到最佳路径。

488-493: 通过调用avl\_walk\_first函数从rt-> rt\_path\_tress里找到树里的第一个条目并保证不为零，然后把节点类型转化为rt\_entry类型。

495-506: 历整棵树并比较当前路径与最佳路径，如果当前路径比最佳路径还要好则把当前路径记录为最佳路径。

路由计算过程：路由节点启动后，首先执行对相邻节点的发现工作，并获取它们的地址。然后路由节点主动测试到每一个相邻路由器的链路时延或成本，并根据测试结果设置相关链路的状态。然后路由节点构造自己的链路状态信息包，链路状态信息的内容包括本路由器的标号、本路由器的邻居路由器列表、本路由器到各邻居路由器的链路状态（时延或成本）、该链路状态信息包的序号和生存时间等。然后该路由节点向所有参与链路状态交互的路由节点广播其链路状态信息，可以是周期性地发送，也可在链路状态发生变化时发送。每个路由节点在收到所有的链路状态信息后，可以构造或更新表示整个网络拓扑结构的图G（V, E），顶点V表示路由器，边E表示连接路由器的链路。这时路由节点就可以用 Dijkstra 算法从图G中计算出到所有目的路由器的最短路径，也就是构造以自己为根节点的最短路径优先树。

## 5.6 HNA消息处理

### 5.6.1 HNA包的生成

在OLSR网络中，应用HNA消息的节点需周期性的生成HNA消息，该消息内容应包括该节点的IP地址及子网掩码，而发包时周期则根据该节点的Vtime来确定。

从该函数中可以看出，节点生成的HNA消息包被存放进一个队列中，并打上时间戳，对于未过期的HNA消息，将其置入缓冲区中，等待发送。

generate\_msg.c

107 void

108 generate\_hna(void \*p)

109 {

110 struct interface \*ifn = (struct interface \*)p;

111

112 if (queue\_hna(ifn) && TIMED\_OUT(ifn->fwdtimer)) {

113 set\_buffer\_timer(ifn);

114 }

115 }

图5.33

112-114: 生成HNA消息的节点将HNA消息放在队列中，当读取到该消息时，判断该消息是否过期，如未过期，则读取消息内容。

### 5.6.2 HNA包的发送

一般情况下，HNA消息生成的包为OLSR标准包，所以发送HNA运用的是OLSR协议的洪泛发包方式。下图所示为HNA消息的发送函数，从函数中可以看到，HNA消息发送时在消息中记录了初始化过程中的vtime，msgtype，msgsize等数据，而hop\_count则用于接收方进行判断消息跳数，接收方借此来判断如何处理这条HNA消息。

hna\_set.c

367 bool

368 olsr\_input\_hna(union olsr\_message \*m, struct interface \*in\_if \_\_attribute\_\_ ((unused)), unio n olsr\_ip\_addr \*from\_addr)

369 {

370

371 uint8\_t olsr\_msgtype;

372 olsr\_reltime vtime;

373 uint16\_t olsr\_msgsize;

374 union olsr\_ip\_addr originator;

375 uint8\_t hop\_count;

376 uint16\_t msg\_seq\_number;

图5.34

函数中的一些判断是在发消息之前确保该节点所发的节点为标准的HNA消息包，确保该消息符合OLSR协议的规范。

hna\_set.c

386 /\* Check if everyting is ok \*/

387 if (!m) {

388 return false;

389 }

390 curr = (const uint8\_t \*)m;

391

392 /\* olsr\_msgtype \*/

393 pkt\_get\_u8(&curr, &olsr\_msgtype);

394 if (olsr\_msgtype != HNA\_MESSAGE) {

395 OLSR\_PRINTF(1, "not a HNA message!\n");

396 return false;

397 }

398 /\* Get vtime \*/

399 pkt\_get\_reltime(&curr, &vtime);

400

401 /\* olsr\_msgsize \*/

402 pkt\_get\_u16(&curr, &olsr\_msgsize);

403

404 hnasize = olsr\_msgsize - 8 - olsr\_cnf->ipsize;

405 curr\_end = (const uint8\_t \*)m + olsr\_msgsize;

406

407 /\* validate originator \*/

408 pkt\_get\_ipaddress(&curr, &originator);

409 /\*printf("HNA from %s\n\n", olsr\_ip\_to\_string(&buf, &originator)); \*/

410

411 /\* ttl \*/

412 pkt\_ignore\_u8(&curr);

413

414 /\* hopcnt \*/

415 pkt\_get\_u8(&curr, &hop\_count);

416

417 /\* seqno \*/

418 pkt\_get\_u16(&curr, &msg\_seq\_number);

图5.35

392-397: 检测该HNA消息包是否符合OLSR消息包的规范，如规范则可继续发送，如不符合规范，则返回false，不发送包。

398-399:调用pkt\_get\_reltime( )函数，获取vtime。

407-409:验证发包源地址的有效性。

### 5.6.3 HNA包的处理

位于OLSR协议中的每个节点持续的对所接收的消息头进行判断，当检测到所需的消息是，就读取该消息。每个节点都会多次接收到同一消息，为了避免因读取重复消息而造成资源浪费，每个节点都维持了一个防重读表。在该表中，节点自动记录近期接收到的消息，并将新接受的消息与该表中的信息进行对比，如果新消息不在改防重读表中，才会读取新消息的具体内容。

每一个包含HNA包的消息的有效时间都是根据包含在该包头部的Vtime数据来计算的，当一个节点读取一个HNA消息包时，所经历的具体步骤如下：

* 若发送HNA包的根节点与接收该包的节点的距离超过一跳，则丢弃这个包；
* 所发送HNA包的根节点与接收该包的节点的距离不超过一跳，读取这个包的源IP与子网掩码；
* 若之前曾与该节点通信，则将该元组的保存时间设为现有时间与有效时间之和；
* 若之前不曾与该节点通信，则需初始化该元组的数据，初始化内容如下：

A\_gateway\_addr = originator address

A\_network\_addr = network address

A\_netmask = netmask

A\_time = current time + validity time

6 OLSR路由协议的安全问题

网络安全问题无处不再，尤其是在MANET这种无线、动态网络中，更容易发起安全攻击。因此，在分析OLSR代码的过程中我们注意到了OLSR路由协议的安全问题。我们发现虽然OLSR路由协议中考虑到了部分安全问题，但是OLSR路由协议并没有提出一整套有效的安全防御检测方案。尽管在协议中加入安全加密方案可以抵御大部分攻击，但对于虫洞攻击这种危害性极大的攻击，OLSR路由协议还没有提出有效的安全防御检测方案。所以，本节对针对OLSR路由协议的虫洞攻击进行讨论，以进一步理解OLSR工作原理，并深入体会其中蕴含的安全隐患。

## 6.1虫洞攻击简介

虫洞是1916年由奥地利物理学家[路德维希](https://baike.baidu.com/item/%E8%B7%AF%E5%BE%B7%E7%BB%B4%E5%B8%8C/9134423)·弗莱姆首次提出的概念，意思是连结两个遥远时空的多维空间隧道。而网络中的虫洞和物理意义上的虫洞有异曲同工之妙。网络中的虫洞是网络中两个节点在两个相距很远的节点间建立的“隧道”。它通过“隧道”的特殊功能，使其体现出很好的路由度量，在网络层极大缩短了两个遥远区域的通信距离。

这样的虫洞建立在某种意义上是对网络通信有利的，因为它极大缩短了两个通信区域间的通信距离。但是这种虫洞一旦被恶意攻击者所利用，就会对网络产生巨大威胁，这就是所谓的虫洞攻击。

两个潜伏在不同通信区域的恶意节点，一旦相互建立“隧道”，由于“隧道”较好的路由度量，就会破坏“隧道”附近的路由信息，将大部分的路由流量吸引到“隧道”中去。也就是说两个通信区域间的所有通信信息都将经过恶意节点和其中间建立的“隧道”。恶意节点通过这一特性，可以对两个通信区域间通信的任意两个节点实施被动攻击，如监测流量信息和监听通信信息等，更有甚者可以发起主动攻击，例如破坏、拦截、篡改和重放通信信息。

虫洞攻击的关键就是“隧道”的建立。网络层路由算法衡量节点间通信距离一般采用的度量有：路径长度、可靠性、时延和带宽等，并没有考察实际数据链路层节点间距离。即便攻击节点间的秘密“隧道”链路的距离比正常单跳无线传输距离要更远，但只要使被“隧道”传输的分组在路由上比合法传递的分组要体现更好的路由度量，就能建立理想“隧道”。一般有两种建立“隧道”的方法。第一种利用网络本身存在的多跳传输，通过分组封装来实现相互之间的通信，使多跳路由在网络层只表现为一跳。恶意节点获取非法数据包后，对其进行封装，使其变为在网络中可以正常传输的数据包，等到另一个恶意节点后再将其解封装，发送给目标节点。使用这种方法建立“隧道”的前提是，恶意节点首先要侵入网络，获取合法身份，而且要求数据包没有加密传输。还有一种较为容易的建立“隧道”的方法。就是恶意节点间建立的一条高带宽的带外信道，这条私有链路在物理距离上远很多，但攻击节点通过高功率发射达到更好的路由度量来完成秘密通信。这种攻击方法需要来自硬件方面的支持，但是其发动攻击比较简单，而且即使在数据包加密的情况下，虫洞攻击也可以发起。并且，这种虫洞攻击较难监测的到。

## 6.2 针对OLSR虫洞攻击建模



图6.1 针对OLSR路由协议的虫洞攻击模型

虽然无线自组网中的虫洞攻击一般大同小异，但是针对不同的路由协议，虫洞攻击采取的策略也一般不同。本节将针对OLSR协议提出一种有效的一般性的虫洞攻击模型，下面的虫洞攻击安全监测和防御策略也将在这种攻击模型的基础上提出。

根据上文中的OLSR路由协议简介可知，OLSR是无线自组网中的一个基于链路状态路由算法的先应式路由协议。基于链路状态路由算法这一特点表示网络中拓扑图的构建是基于每个运行OLSR的节点的邻居信息。先应式路由协议表示网络中运行OLSR路由协议的节点通过发送hello包来发现自己的邻居信息。依据这两个特点，我们将采用转发hello包，以建立伪邻居节点的方式进行虫洞攻击。同时，OLSR路由协议的一个巨大特点就是他的MPR洪泛机制，针对OLSR的虫洞攻击必须保证通过恶意节点建立的伪邻居节点必须总是被被攻击节点选为MPR节点，来进行数据包的转发。

因此，我们提出下面的针对OLSR的虫洞攻击模型。

首先此处虫洞攻击建立的“隧道”是两个攻击节点间建立的一条高带宽的带外信道。

如图2.1通信区域1和通信区域2在物理位置上相隔很远，节点A和C位于通信区域1内，节点B和D位于通信区域2内，节点A和节点B间建立有一条“隧道”。当节点C广播HELLO分组时，攻击节点A捕获该HELLO分组并通过“隧道” 传给合谋节点B，B将A的HELLO分组重放到网络中。当节点D收到重放的 HELLO分组后，认为节点C是它的1跳邻居。同样，C在收到D的HELLO分组后，也会认为D是邻居。这样，C和D均成为对方邻居表中的伪邻居。同时，因为通信区域1和通信区域2相距很远，D附近的其他节点与节点C之间只有节点D这一个一跳邻居节点，同理，节点C附近的节点也是同样情况。根据OLSR路由协议的MPR节点选择算法的第一条，节点会优先在节点的两跳邻居节点集合中选择那些只与节点的一跳邻居节点集合中的一个节点互为邻居的节点，节点D和节点C将互为对方的MPR节点。

这样“隧道”将表现除更好的路由度量，并且通过OLSR协议的运行，这条虚假路由信息将进入到节点用于路由的网络拓扑图中。到此，虫洞攻击建立，两通信区域间的大量路由流量将被吸引到此“隧道”中去。两通信区域间的大量TC分组和数据分组都将通过“隧道”传输，分组信息直接暴露在攻击节点下，它们即可发动丢弃、篡改、伪造等恶意行为。因此，针对虫洞攻击，必须对OLSR路由协议作相应安全改进，加强其防御能力。

7 感悟和总结

## 7.1 总结

经过将近一个月的对OLSR路由协议的分析理解，我们对OLSR路由协议的运行功能有了更深一步的理解。

OLSR路由协议整体的两大功能是接收数据包处理以及周期性发送数据包，节点通过周期性发送数据包像网络中通知自己的节点信息（包括链路、接口、网络和拓扑等信息），此部分涉及到了OLSR消息的生成以及数据包的生成。而接收数据包处理部分在辅助网路中消息的传播的同时，主要是通过消息处理函数协调完成OLSR路由协议的多种功能（包括邻居集的生成、MPR集的生成、多端口节点功能处理、TC控制、路由计算和HNA消息的处理等功能）最终完成网络中的路由目的。这两大功能都是通过信号处理程序调用完成。

## 7.2 陈雨涵感想

从短短的几周大作业时间里，我还是学到了不少东西。从最开始的不知如何分工，去网上搜索资料，粗略了解OLSR协议的过程；到将协议的最主要部分分工，各自完成相应模块的代码阅读和理解；到最后的整合，以及和成员之间相互交流各自部分的功能。从最开始的一脸茫然，无从下手，到最后基本理解了协议的整个流程，以及大部分的细节，满满的都是成就感。

在代码分析的过程中，最大的难点就在于我们各自的内容之间多多少少都会穿插着其他成员的内容，没有小组其他成员的代码讲解，我不会这么快得理解自己部分的协议流程。而最开心的，莫过于理解协议功能和流程的那一刻。每每在仔细阅读RFC文档内容后，这样的成就感总能让人满足。

最后，感谢覃振权老师课上的精彩讲解，感谢助教学长们的认真、细心的帮助，还有，感谢两位小组成员在大作业过程中对我的帮助，使我对OLSR协议的理解更加深刻。

## 7.3 刘棪感想

通过这接近一个月的时间，我们组完成了对于OLSR路由协议的简易分析。根据自己学习的经验，在这简单说明一下怎样系统的学习一种协议：

入门时，看一些相关的书籍会好一些，在这里我看的是Jeff Doyle的OSPF和IS-IS详解，如果英文不好，直接看英文标准，不太好理解一些抽象的基础概念，结合书籍和网络上的教程，可以对所要学习的东西有一个大致的了解。接下来，动手实验，在实验中可以对协议的内容有一个更深的理解。 然后，再看一些相关的论文。 最后，想要了解某一点的详细情况时，再去查阅相关的标准。

## 7.4 商迪感想

在我们三位成员的共同努力下，我们在将近一个月的时间内完成了对OLSR路由协议的简易分析。这一个月的时间对我来说意义非凡。从最开始的无从下手，不知所措到如今的了然于心，中间经历的困难与喜悦都将成为我人生中宝贵的财富。

最开始拿到代码时，面对庞杂的文件，以及复杂的代码，实在无从下手。于是我们采用先了解在细化的方式，经过多天的搜集和研习资料初步对OLSR路由协议的主要功能和算法有了大致了解。随后我们从功能出发，采用自顶向下的方式从整体上分析协议文件，和基本处理流程。与此同时，我们将主要功能分配下去，由小到大的分析方法同步开展。这样系统全面的分析方法，最终是我们对OLSR路由协议有了较为细致的理解。

最后，感谢覃振权老师课上的精彩讲解，感谢助教学长们的认真、细心的帮助，感谢两位小组成员在大作业过程中对我的支持和帮助。