

研究型网络实验报告

**OSPF协议设计实现**

|  |  |
| --- | --- |
| 学生姓名 | 童浩 |
| 学生学号 | 10231016 |
| 指导教师 | 张力军 |
| 培养院系 | 计算机学院 |

2013年7月

目录

[1 绪论 1](#_Toc361618841)

[1.1 路由协议简介 1](#_Toc361618842)

[1.2 OSPF路由协议简介 1](#_Toc361618843)

[1.3 OSPF主要名词术语 2](#_Toc361618844)

[1.4 实现基本目标和完成情况 4](#_Toc361618845)

[2 OSPF概述 5](#_Toc361618846)

[2.1 OSPF区域划分 5](#_Toc361618847)

[2.2 OSPF连接方式 6](#_Toc361618848)

[2.3 OSPF网络类型 7](#_Toc361618849)

[2.4 OSPF数据包 7](#_Toc361618850)

[2.4.1 包头格式 7](#_Toc361618851)

[2.4.2 Hello包 8](#_Toc361618852)

[2.4.3 数据库描述包/DD 包 10](#_Toc361618853)

[2.4.4 连接状态请求包/LSR 包 12](#_Toc361618854)

[2.4.5 连接状态更新包/LSU 包 12](#_Toc361618855)

[2.4.6 连接状态确认包/LSAck 包 13](#_Toc361618856)

[2.5 链路状态宣告（LSA） 14](#_Toc361618857)

[2.5.1 Router-LSA 14](#_Toc361618858)

[2.5.2 Network-LSA 15](#_Toc361618859)

[2.5.3 Network Summary LSA 16](#_Toc361618860)

[2.5.4 ASBR Summary LSA 16](#_Toc361618861)

[2.5.5 AS-external-LSA 16](#_Toc361618862)

[2.6 LSA 格式 16](#_Toc361618863)

[2.6.1 LSA 头部 16](#_Toc361618864)

[2.6.2 Router-LSA 17](#_Toc361618865)

[2.6.3 Network-LSA 18](#_Toc361618866)

[2.6.4 Summary-LSA 18](#_Toc361618867)

[2.6.5 AS-external-LSA 19](#_Toc361618868)

[2.7 最短路径算法 20](#_Toc361618869)

[3 OSPF结构实现 21](#_Toc361618870)

[3.1 整体架构设计 21](#_Toc361618871)

[3.1.1 整体结构示意图 21](#_Toc361618872)

[3.1.2 主要数据结构 22](#_Toc361618873)

[3.1.3 文件组织 27](#_Toc361618874)

[3.2 OSPF实用类 27](#_Toc361618875)

[3.2.1 基于AVL树的列表 27](#_Toc361618876)

[3.2.2 Patricia树 28](#_Toc361618877)

[3.2.3 优先级队列 29](#_Toc361618878)

[3.2.4 计时器 29](#_Toc361618879)

[3.3 维护系统设计 29](#_Toc361618880)

[3.4 邻居路由器维护 32](#_Toc361618881)

[3.4.1 邻居状态机 32](#_Toc361618882)

[3.4.2 接口状态机 34](#_Toc361618883)

[3.5 链路状态数据库 36](#_Toc361618884)

[3.5.1 LSA及数据库实现 36](#_Toc361618885)

[3.5.2 LSA创建 36](#_Toc361618886)

[3.5.3 LSA列表 36](#_Toc361618887)

[3.5.4 LSA老化 36](#_Toc361618888)

[4 OSPF过程实现 36](#_Toc361618889)

[4.1 分组组装过程 36](#_Toc361618890)

[4.2 发现邻居路由器与Hello处理过程 36](#_Toc361618891)

[4.2.1 发现邻居路由与接收Hello分组 36](#_Toc361618892)

[4.2.2 发送Hello分组 38](#_Toc361618893)

[4.2.3 DR选举过程 38](#_Toc361618894)

[4.3 数据库交换与Database Description处理过程 42](#_Toc361618895)

[4.3.1 数据库交换 42](#_Toc361618896)

[4.3.2 接收DD分组 42](#_Toc361618897)

[4.3.3 发送DD分组 45](#_Toc361618898)

[4.4 Link State Request处理过程 47](#_Toc361618899)

[4.4.1 发送LSR分组 47](#_Toc361618900)

[4.4.2 接收LSR分组 48](#_Toc361618901)

[4.5 泛洪与Link State Update处理过程 50](#_Toc361618902)

[4.5.1 单个路由泛洪过程 50](#_Toc361618903)

[4.5.2 接收LSU分组 50](#_Toc361618904)

[4.5.3 泛洪LSA 50](#_Toc361618905)

[4.5.4 响应LSR而发送LSU分组 50](#_Toc361618906)

[4.5.5 重发LSA 50](#_Toc361618907)

[4.6 Link State Acknowledgment处理过程 50](#_Toc361618908)

[4.6.1 接收LS ACK分组 50](#_Toc361618909)

[4.6.2 发送LS ACK分组 50](#_Toc361618910)

[4.7 路由选择计算 50](#_Toc361618911)

[4.7.1 触发路由重新计算 50](#_Toc361618912)

[4.7.2 路由计算 50](#_Toc361618913)

[4.7.3 与内核同步路由 50](#_Toc361618914)

[结论 51](#_Toc361618915)

[致谢 52](#_Toc361618916)

[参考文献 53](#_Toc361618917)

[附录A 54](#_Toc361618918)

[附录B 55](#_Toc361618919)

# 绪论

## 路由协议简介

路由协议通过在路由器之间共享路由信息来支持可路由协议。路由信息在相邻路由器之间传递，确保所有路由器知道到其它路由器的路径。总之，路由协议创建了路由表，描述了网络拓扑结构；路由协议与路由器协同工作，执行路由选择和数据包转发功能。

路由分为静态路由和动态路由，其相应的路由表称为静态路由表和动态路由表。静态路由表由网络管理员在系统安装时根据网络的配置情况预先设定，网络结构 发生变化后由网络管理员手工修改路由表。动态路由随网络运行情况的变化而变化，路由器根据路由协议提供的功能自动计算数据传输的最佳路径，由此得到动态路由表。

根据路由算法，动态路由协议可分为距离向量路由协议（Distance Vector Routing Protocol）和链路状态路由协议（Link State Routing Protocol）。距离向量路由协议基于Bellman-Ford算法，主要有RIP、IGRP（IGRP为 Cisco公司的私有协议）；链路状态路由协议基于图论中非常著名的Dijkstra算法，即最短优先路径（Shortest Path First， SPF）算法，如OSPF。在距离向量路由协议中，路由器将部分或全部的路由表传递给与其相邻的路由器；而在链路状态路由协议中，路由器将链路状态信息传 递给在同一区域内的所有路由器。 根据路由器在自治系统（AS）中的位置，可将路由协议分为内部网关协议 （Interior Gateway Protocol，IGP）和外部网关协议（External Gateway Protocol，EGP，也叫域 间路由协议）。域间路由协议有两种：外部网关协议（EGP）和边界网关协议（BGP）。EGP是为一个简单的树型拓扑结构而设计的，在处理选路循环和设置 选路策略时，具有明显的缺点，目前已被BGP代替。

## OSPF路由协议简介

OSPF(Open Shortest Path First开放式最短路径优先）是一个内部网关协议(Interior Gateway Protocol，简称IGP），用于在单一自治系统（autonomous system,AS）内决策路由。是对链路状态路由协议的一种实现，隶属内部网关协议（IGP），故运作于自治系统内部。著名Dijkstra算法被用来计算最短路径树。

## OSPF主要名词术语

**Router路由器**

一种三层IP分组交换机。

**Autonomous System 自治系统**

一组使用相同路由协议来交换路由信息的路由器。简写为AS。

**Interior Gateway Protocol 内部网关协议**

被一个AS 内的路由器所使用的路由协议，缩写为IGP。每个AS 使用单一的IGP，不同的AS 会使用不同的IGP。

**Router ID 路由器标识**

一个用来识别每台运行OSPF协议路由器的32位数。在一个自治系统中，这个数唯一地标志一个路由器。

**Network 网络**

表示IP 网络/子网/超网。一个物理网络上可能设置有多个网络/子网号，我们把它们按照独立的网络来对待。

**Network mask 网络掩码**

一个 32 位的数字，表示IP 地址的范围来说明这是一个IP 网络/子网/超网。本文以16 进制来表示网络掩码。

**Point-to-point networks 点对点网络**

仅仅连接一对路由器的网络。

**Broadcast networks 广播网络**

支持多台（大于两台）路由器接入的网络，同时有能力发送一条信息就能到所有接入的路由器（广播）。网络上邻居路由器可以通过OSPF 的Hello协议来动态发现。如果可能，OSPF协议将进一步使用多播。广播网络上的每一对路由器都被认为可以直接通讯。

**Non-broadcast networks 非广播网络**

支持多台（大于两台）路由器接入的网络，但没有广播能力。网络上的邻居路由器通过OSPF的Hello 协议来维持。但由于缺乏广播能力，需要一些配置信息的帮助来发现邻居。在非广播网络上，OSPF协议的数据通常需要被轮流发送到每一台邻居路由器上。

**Interface 接口**

是指路由器与所接入的网络之间的一个连接。接口通过下层协议和路由协议获取与其相关的状态信息。指向网络的接口只和单一的IP 地址及掩码相关（除非是无编号的点对点网络）。接口有时也被称为连接。

**Neighboring routers 邻居路由器**

在同一网络中都有接口的两台路由器。邻居关系是由OSPF 的Hello 协议来维持，并通常依靠Hello协议来动态发现。

**Adjacency 邻接**

用以在所选择的邻居路由器之间交换路由信息的关系。不是每对邻居路由器都会成为邻接。

**Link state advertisement 链路状态宣告**

描述路由器或网络自身状态的数据单元。对路由器来说，这包含它的接口和邻接状态。每一项连接状态宣告都被泛洪到整个路由域中。所有路由器和网络链路状态宣告的集合形成了协议的连接状态数据库。

**Hello Protocol Hello协议**

在 OSPF 协议中，用于建立和维持邻居关系的部分。在广播网络中还被用于动态发现邻居路由器。

**Flooding 泛洪**

在 OSPF 协议中，用于OSPF 路由器之间发送及同步连接状态数据库的部分。

**Designated Router 指定路由器**

在每个接入了至少两台路由器的广播和NBMA 网络中都有一台作为指定路由器。指定路由器生成Network-LSA并在运行协议时完成其他特定职责。指定路由器通过Hello协议选举。指定路由器的概念减少了广播和NBMA网络上所需要的邻接数量。同时也减少了路由协议所需要的流量及连接数据库的大小。常缩写为DR。

**Lower-level protocols 下层协议**

为 IP 及OSPF协议提供服务的下层网络接入协议。如为以太网服务的以太网数据链路层。

## 实现基本目标和完成情况

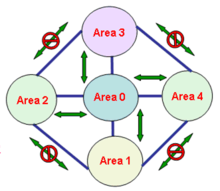
* + - 1. 骨干区域（区域号为0）中广播网络中的DR、BDR选举。
      2. OSPF分组的封装和发送。
      3. Router LSA和Network LSA的实现。
      4. 最短路径的计算。
      5. 将计算出的路由表存入Linux系统内核。

## 实现难点

* + - 1. 实现了较为独立的类，提供了一定的扩展性。详见3.1.2“主要数据结构”。
      2. 实现了较为完整的LSA老化，详见3.5.2 “LSA老化”。
      3. 利用特定的数据结构，实现了最短路径算法。详见4.6.1 “路由计算”。
      4. 利用Linux IPv4 routing socket，直接向内核添加或删除路由，用IP转发实现路由功能。详见4.6.2 “与内核同步路由”。

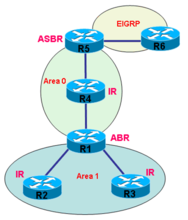
# OSPF概述

## OSPF区域划分

[](http://baike.baidu.com/picview/6234950/6326149/0/0bd162d9f2d3572c3426a8ad8b13632763d0c3fe.html)因为OSPF路由器之间会将所有的链路状态（LSA）相互交换，毫不保留，当网络规模达到一定程度时，LSA将形成一个庞大的数据库，势必会给OSPF计算带来巨大的压力；为了能够降低OSPF计算的复杂程度，缓存计算压力，OSPF采用分区域计算，将网络中所有OSPF路由器划分成不同的区域，每个区域负责各自区域精确的LSA传递与路由计算，然后再将一个区域的LSA简化和汇总之后转发到另外一个区域，这样一来，在区域内部，拥有网络精确的LSA，而在不同区域，则传递简化的LSA。区域的划分为了能够尽量设计成无环网络，所以采用了Hub-Spoke的拓朴架构，也就是采用核心与分支的拓朴，如下图：

区域的命名可以采用整数数字，如1、2、3、4，也可以采用IP地址的形式，0.0.0.1、0.0.0.2，因为采用了Hub-Spoke的架构，所以必须定义出一个核心，然后其它部分都与核心相连，OSPF的区域0就是所有区域的核心，称为BackBone 区域（骨干区域），而其它区域称为Normal 区域（常规区域），在理论上，所有的常规区域应该直接和骨干区域相连，常规区域只能和骨干区域交换LSA，常规区域与常规区域之间即使直连也无法互换LSA，如上图中Area 1、Area 2、Area 3、Area 4只能和Area 0互换LSA，然后再由Area 0转发，Area 0就像是一个中转站，两个常规区域需要交换LSA，只能先交给Area 0，再由Area 0转发，而常规区域之间无法互相转发。

OSPF区域是基于路由器的接口划分的，而不是基于整台路由器划分的，一台路由器可以属于单个区域，也可以属于多个区域，如下图：

[](http://baike.baidu.com/picview/6234950/6326149/0/9358d109b3de9c8274b0cdfe6d81800a18d843ed.html)

## OSPF连接方式

如果一台OSPF路由器属于单个区域，即该路由器所有接口都属于同一个区域，那么这台路由器称为Internal Router（IR），如上图中的R2，R3和R4；如果一台OSPF路由器属于多个区域，即该路由器的接口不都属于一个区域，那么这台路由器称为Area Border Router （ABR），如上图中的R1，ABR可以将一个区域的LSA汇总后转发至另一个区域；如果一台OSPF路由器将外部路由协议重分布进OSPF，那么这台路由器称为Autonomous System Boundary Router （ASBR），如上图中，R5将EIGRP重分进OSPF，那么R5就是ASBR，但是如果只是将OSPF重分布进其它路由协议，则不能称为ASBR。可以配置任何OSPF路由器成为ABR或ASBR。

下面给出这四种路由器的详细定义：

* IAR(Interal Area Router)区域内路由器：

所有接口属于一个区域，只生成一条LSA，只有一个LSDB。

* ABR(Area Border Router)区域边界路由器：

同时属于两个以上区域，为所属的每个区域生成一条LSA和保存一个LSDB，根据需要能够生成第三、第四类LSA。

* ASBR(AS Boundery Router)自治系统边界路由器：

引入其它路由协议的路由器。不一定在AS的边界。生成第五类LSA。

* BBR(BackBone Router)骨干路由器

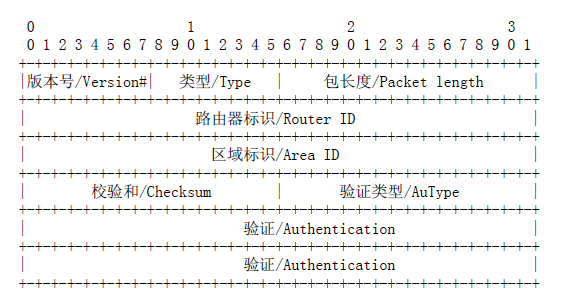
## OSPF网络类型

* **点到点网络(point-to-point）**，由cisco提出的网络类型，自动发现邻居，不选举DR/BDR，hello时间10s。
* **广播型网络(broadcast）**，由cisco提出的网络类型，自动发现邻居，选举DR/BDR，hello时间10s。
* **非广播型（NBMA）网络 (non-broadcast）**，由RFC提出的网络类型，手工配置邻居，选举DR/BDR，hello时间30s。
* **点到多点网络 (point-to-multipoint）**，由RFC提出，自动发现邻居，不选举DR/BDR，hello时间30s。
* **点到多点非广播**，由cisco提出的网络类型，手动配置邻居，不选举DR/BDR，hello时间30s。

## OSPF数据包

### 包头格式

每个OSPF 包都以24 字节的头部开始。头部所包含的所有信息用于决定包如何进行下一步操作。



**版本号/Version #**

OSPF 的版本号，本规范所说明的协议版本号为2。

**类型/Type**

OSPF 包的类型按如下定义。具体细节见附录A.3.2 到附录A.3.6。

**类型描述**

1 Hello 包

2 数据库描述包/Database Description

3 连接状态请求包/Link State Request

4 连接状态更新包/Link State Update

5 连接状态确认包/Link State Ack

**包长度/Packet length：**

整个OSPF 包的字节长度，包括标准的OSPF 头部。

**路由器标识/Router ID：**

生成包的路由器标识。

**区域标识/Area ID：**

一个32 位数表示包所属于的区域。所有的OSPF 包都关联到一个区域。大多数都只传输一跳。

在虚拟通道中的传输关联到骨干区域0.0.0.0。

**校验和/Checksum：**

从OSPF 包头开始，除了64 位的验证域外，整个包的标准IP 校验和。该校验和作为16 位1补足校验和而计算包中除验证域外的所有16 位字。如果包的长度不满足16 位字，就在校验和前加上一个字节的0 来补足。校验和也被作为正确性验证的一部分，在某些OSPF 验证类型下，校验和计算被忽略。

**验证类型/AuType：**

说明所使用验证过程的类型。

### Hello包

Hello 包的OSPF 包类型为1。这些包被周期性的从各个接口（包含虚拟通道）发送，以建立和维持邻居关系。此外，在支持多播或广播的物理网络上，Hello 包使用多播来动态的发现邻居路由器。

所有接入同一网络的路由器必须使用相同的参数（网络掩码、HelloInterval 和RouterDeadInterval）。这些参数都包含在Hello 包中，参数的不同将阻止形成邻居关系。



**网络掩码/Network mask：**

该接口所关联的网络掩码。例如，接口上设定的是B 类网络并使用第三个字节作为子网，则网络掩码为0xffffff00。

**选项/Options：**

在附录A.2 中说明的路由器所支持的选项。

**HelloInterval：**

路由器发送Hello 包的间隔秒数

**路由器优先级/Rtr Pri：**

路由器的优先级。用于DR、BDR 的选举。如果设为0，路由器就不能成为DR 或BDR。

**RouterDeadInterval：**

在宣告安静的路由器为断开前所需要等待的秒数。

**指定路由器/Designated Router：**

以发送路由器的视角认为网络上的DR。DR以其网络上的接口IP地址作为标识。设定为0.0.0.0表示没有DR。

**备份指定路由器/Backup Designated Router：**

以发送路由器的视角认为网络上的BDR。BDR 以其网络上的接口IP 地址作为标识。设定为0.0.0.0 表示没有BDR。

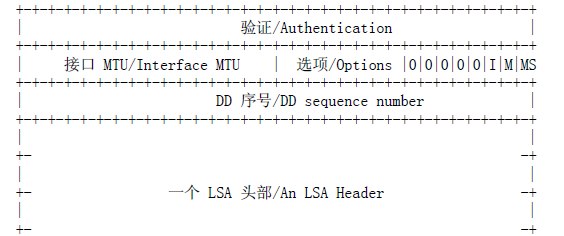
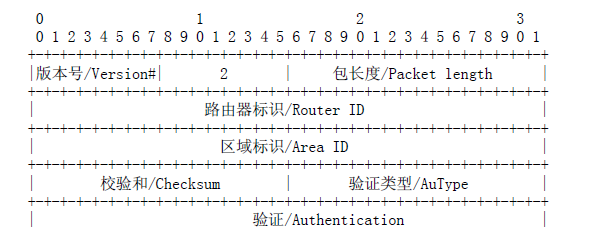
**邻居/Neighbor：**

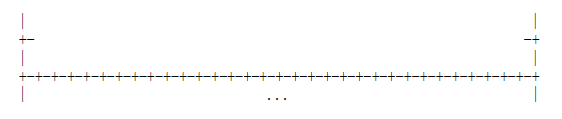
通过有效的Hello 包，从网络上新近收到的路由器标识。新近是指在RouterDeadInterval 内。

### 数据库描述包/DD 包

DD 包的OSPF 包类型为2。当初始化邻接时交换这种包，它描述了连接状态数据库的目录。可能使用多个包来描述数据库，所以使用一种发送—响应的过程。一台路由器被指定为主机，而另一台为从机。主机发送的DD 包，被从机的DD 包所确认，之间是通过包中的DD 序号进行联系。

DD 包的格式与LSR 包和LSAck 包类似。这三种包的主要内容为描述连接状态数据库内容的一系列列表。





**接口MTU/Interface MTU：**

从所关联的接口不分片而能发送的最大IP 包字节大小。在虚拟通道上的DD 包里，接口MTU 被设为0。

**选项/Options：**

按附录A.2 的描述，路由器所支持的选项。

**I 位/I-bit：**

初始位，在第一个DD 包中设定为1。

**M 位/M-bit：**

更多位。当后面还有更多的DD 包时设定为1。

**MS 位/MS-bit：**

主从位。在数据库交换过程中的主机设定为1，否则该路由器为从机。

**DD 序号/DD sequence number：**

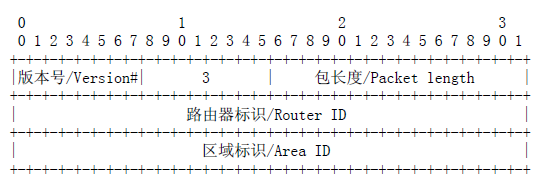
用于描述DD 包的序号。其初始值（设定了初始位）应当唯一。在整个数据库描述过程中，所发送的DD 需要应当线形增加。

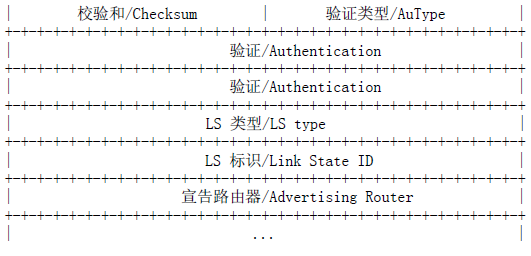
包中的其他部分是包含了连接状态数据库内容的列表（可能是局部列表）。每个LSA 由其LSA 头部来描述。这些信息可以唯一的识别出该LSA 及其当前实例。

### 连接状态请求包/LSR 包

LSU 包的OSPF 包类型为3。在与邻居交换了DD 包后，路由器后发现它的一部分连接状态数据库已经过期。这时就使用LSR 包来取得邻居数据库中较新的部分。也许需要使用多个LSR 包。

发送LSR 包的路由器确切的知道所请求的实例。每个实例由LS 序号、LS 校验和以及LS 时限来定义，虽然这些域不是在LSR 包中说明。在响应中，路由器会收到最新的实例。



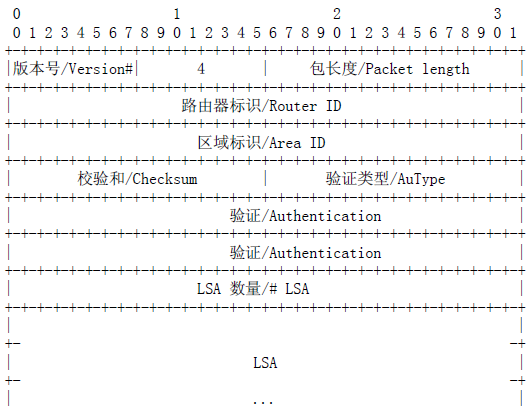


所请求的每个LSA 由LS 类型、LS 标识和宣告路由器来说明。这唯一的识别出一个LSA，但没有指明实例。LSR 包被理解为请求该LSA 最新的实例（而且也确实如此）。

### 连接状态更新包/LSU 包

LSU 包的OSPF 包类型为4。此包实现了LSA 的洪泛。每个LSU 包将其包含的LSA 传送到距其起源更远的一跳。多个LSA 可能被包含在一个包中。

LSU 包在支持多播/广播的物理网络上使用多播。为了使洪泛过程可靠，洪泛的LSA 由LSAck 包所确认。如果需要重传当前的LSA，重传的LSA 始终被直接发送到邻居。



**LSA 数量/# LSA：**

该更新包中包含的LSA 数量。

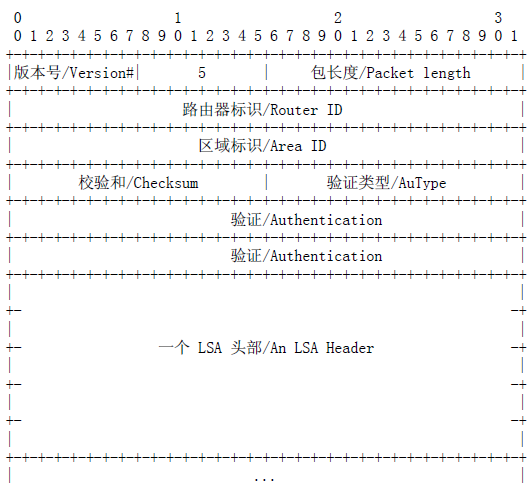
LSU 包内包含有一系列LSA。每个LSA 的开始是20 个字节的通用头部。

### 连接状态确认包/LSAck 包

LSAck 包的OSPF 包类型为5。通过明确的确认，该包使洪泛LSA 变为可靠。确认是通过发送和接收LSAck 包而实现。多个LSA 的确认可以包含在一个LSAck 包中。

根据当前发送接口的状态以及LSU 包的发送路由器，LSAck 包可能发送到多播地址AllSPFRouters或AllDRouters 或者使用单播。

包的格式与DD 包类似。包中包含了LSA 头部的列表。



通过LSA 头部的描述确认每个LSA。

## 链路状态宣告（LSA）

### Router-LSA

路由器为其所接入的每个区域生成Router-LSA。每个LSA 描述路由器接入该区域的接口状态。该LSA 仅在特定的区域内被洪泛，而不是更远。Router-LSA的前20 个字节为一般的LSA头部，Router-LSA的LS 类型= 1。通过 Router-LSA 中的特定位，路由器说明自己是否为ABR 或ASBR（分别通过B 位和E 位）。这使得在路由表中保留到达这些类型路由器的路径，并为后面处理Summary-LSA 和AS-external-LSA 打好了基础。当路由器接入两个或更多区域时设定B 位，即使路由器当前没有接入OSPF骨干区域。在存根区域中决不能在Router-LSA 中设置E 位（ASBR 不能存在于存根区域中）。此外，当且仅当路由器配置的一条或多条以区域A 为传输区域的虚拟通道达到完全邻接时，要在区域A的Router-LSA中设置V位。V位的设定使得区域中的其他路由器能够发现，区域是否支持传输流量。Router-LSA描述了路由器在区域中运作的连接（及接口或通道）。依据所接入的网络类型，而决定连接类型。每个连接都使用连接标识来识别。连接标识向连接的另一端给出了名称。下表汇总了类型值和连接标识。

|  |  |
| --- | --- |
| 链接标识 | 链接类型 |
| 1 | 点对点连接邻居路由器标识 |
| 2 | 连接到传输网络DR 的接口地址 |
| 3 | 连接到存根网络网络IP地址 |
| 4 | 虚拟通道邻居路由器标识 |

表1. Router-LSA 中的连接描述

此外，每个连接有连接数据域，给出了此连接的32 位附加信息。对于传输网络、编号点对点连接和虚拟通道，该域给出了关联接口的IP地址；对于存根网络，该域给出了存根网络的IP 地址掩码；对于无编号点对点连接，该域被设为无编号接口的MIB-II接口索引值。

最后，是该连接的输出值，连接的输出数值可配置。除了到存根网络的连接，输出数值始终不为0。为了进一步描述建立连接列表的过程，假设路由器需要为区域A 建立Router-LSA。路由器开始检查接口数据结构，对于每个接口，执行下面的步骤：

如果所接入的网络不属于区域A，不在LSA中加入连接，开始检查下一个接口。

如果接口状态为Down，不增加连接。

如果接口状态为Loopback，只要不是无编号点对点接口，就增连接（存根网络）。

连接标识为接口IP 地址，连接数据设为掩码0xffffffff（表示主机路径），其距离值设为0。否则，将连接描述按OSPF接口类型加入Router-LSA。

### Network-LSA

为每个传输的广播或NBMA 网络生成Network-LSA（传输网络是指接入了两台或更多路由器的网络）。Network-LSA 描述了所有接入网络的路由器。网络上的DR生成这个LSA。仅当DR与网络上至少一台路由器完全邻接后，才生成该LSA。该Network-LSA仅在包含该传输网络的区域中洪泛，而不是更远。Network-LSA中包含了与DR 完全邻接的邻居列表，各台路由器由其OSPF路由器标识来识别。DR 自己也包含在列表中。Network-LSA 的LS标识是DR 的接口IP 地址。将该值与网络地址掩码（也包含在该Network-LSA 中）进行运算可得出网络IP地址。当路由器不再是网络上的DR，就要删除以前生成的Network-LSA。该LSA 不再用于路由表计算。这通过将LSA提前老化到MaxAge 并重新洪泛而实现。此外还有一种罕见的情况，当路由器标识改变时，必须删除以旧路由器标识所生成的Network-LSA。该Network-LSA 的LS标识等于某一接口IP地址，但宣告路由器不为路由器标识，而路由器不可能知道以前的路由器标识。

### Network Summary LSA

由ABR路由器始发，用于通告该区域外部的目的地址.当其他的路由器收到来自ABR的Network Summary LSA以后，它不会运行SPF算法，它只简单的加上到达那个ABR的开销和Network Summary LSA中包含的开销,通过ABR，到达目标地址的路由和开销一起被加进路由表里，这种依赖中间路由器来确定到达目标地址的完全路由（full route）实际上是距离矢量路由协议的行为。

### ASBR Summary LSA

由ABR发出，ASBR汇总LSA除了所通告的目的地是一个ASBR而不是一个网络外，其他同Network Summary LSA.

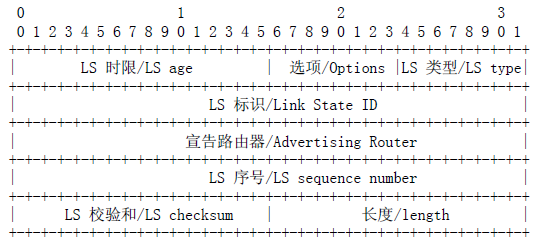
### AS-external-LSA

AS-external-LSA 描述了到达AS 外部目标的路径。大多数AS-external-LSA 描述了到达特定外部目标的路径，这时LSA的LS 标识被设为目标网络的IP地址。然而，AS 的默认路径可以被描述为LS 标识设为Default Destination（0.0.0.0）的AS-external-LSA。AS-external-LSA 是由ASBR所生成。不论是从其他路由协议（如BGP），或通过配置信息得到，ASBR 为每条外部路径生成单独的一个AS-external-LSA。发自ASBR路由器，用来通告到达OSPF自治系统外部的目的地，或者OSPF自治系统那个外部的缺省路由的LSA.这种LSA将在全AS内泛洪（4个特殊区域除外）。

## LSA 格式

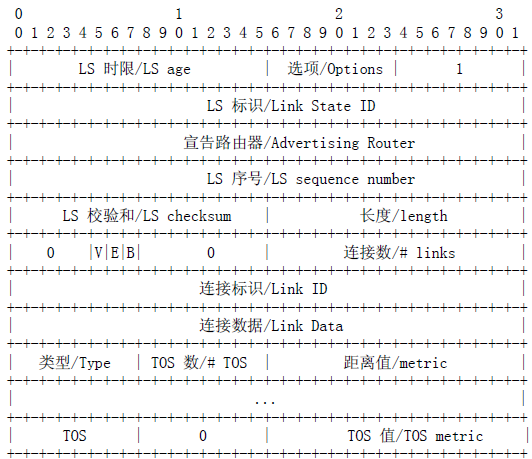
### LSA 头部

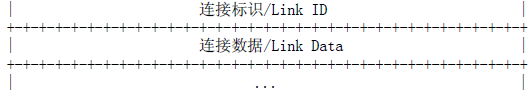
所有的LSA 开始于20 个字节的头部。头部包含的信息可以唯一的识别出LSA（LS 类型、LS 标识和宣告路由器）。在同一时间里，可能存在LSA 的多个实例。必须判定哪个实例较新。这通过检查LSA 头部中的LS 时限、LS 序号和LS 校验和来判定。



### Router-LSA

Router-LSA 是类型1 LSA。每台路由器生成一个Router-LSA 。该LSA 描述了路由器连接（接口）到区域的状态和距离值。连接到一个区域的所有接口必须在一个Router-LSA 中描述。

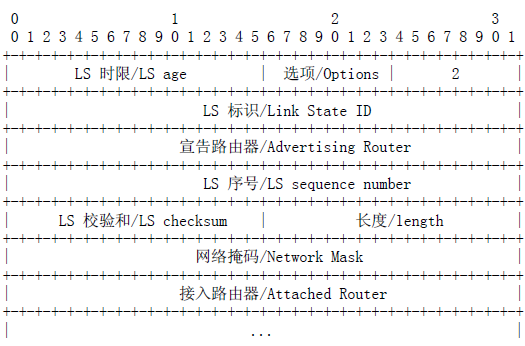




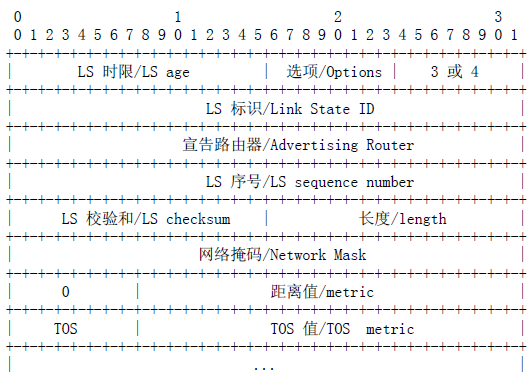
在Router-LSA 中，LS 标识被设为路由器的OSPF 路由器标识。Router-LSA 仅在一个区域内洪泛。

### Network-LSA

Network-LSA 是类型2 LSA。Network-LSA 是为区域中接入了两个或多个路由器的广播和NBMA 网络而生成。Network-LSA 由网络中的DR 生成。每个LSA 描述了接入网络的所有路由器，包括DR自身。在该LSA 中，LS 标识域表示DR 的接口IP 地址。从网络到所接入路由器的距离为零。所以在Network-LSA 中不需要距离域。



### Summary-LSA

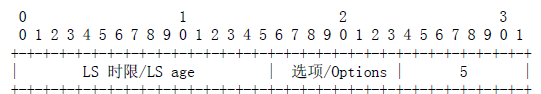
Summary-LSA 是类型3 和4 的LSA。这些LSA 由ABR 生成，描述区域间目标。当目标为IP 网络时使用类型3 的Summary-LSA。这时LSA 的LS 标识为IP 网络号。当目标为ASBR时，使用类型4 的Summary-LSA，这时LS 标识是ASBR 的OSPF 路由器标识。除了LS 标识域外，类型3 和4 的Summary-LSA 格式相同。

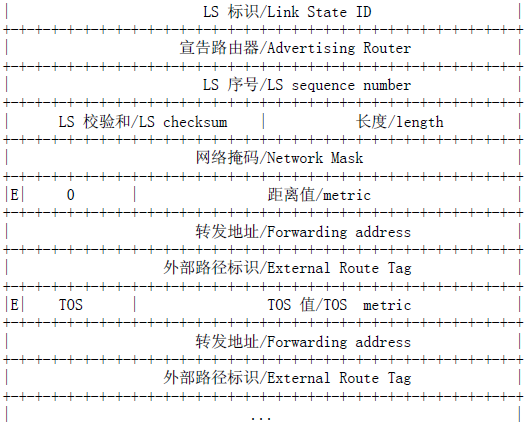
对于存根区域，类型3 的Summary-LSA 同样被用于描述默认路径（每区域）。在存根区域中，不使用一系列的外部路径，而使用默认路径。当描述默认路径时，Summary-LSA 的LS 标识始终为DefaultDestination（0.0.0.0）并且其网络掩码被设为0.0.0.0。

### AS-external-LSA

AS-external-LSA 是类型5 的LSA。是由ASBR 所生成，并描述AS 外部的路径。

AS-external-LSA 通常描述外部目标。这些LSA 的LS 标识域为网络的IP 地址。AS-external-LSA 同样被用于描述默认路径。当一个目标没有特定路径时使用默认路径。描述默认路径时，LS 标识始终被设定为DefaultDestination（0.0.0.0），并且其网络掩码被设为0.0.0.0。





## 最短路径算法

OSPF在选择路由的时候，会根据耗费(Metric)的长短来选择最短的路径进行报文的转发。而在最短路径的实现方案上，OSPF协议选择了Dijkstra(迪杰斯特拉)算法是典型的最短路径路由算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法能得出最短路径的最优解，但由于它遍历计算的节点很多，所以效率低。Dijkstra算法是很有代表性的最短路算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。

# OSPF结构实现

## 整体架构设计

### 整体结构示意图



1. 分组处理

当在接口上收到一个OSPF协议分组时，主循环通过调用解析函数来解析该分组，通过解析分组，来进行各种动作。对于收到的Hello分组，OSPF会建立邻居关系，而如果在一定时间内没有收到Hello分组，则会破坏邻居关系。对于收到的数据库描述（Database Description, DD）分组和链路状态请求（Link State Request, LSR/LS Request）分组会通过邻居状态机中的一系列状态和动作促进链接状态的建立，并保证相邻路由器的数据库保持同步，建立起完全的邻接关系。

在与一个路由器建立邻接关系之后，相邻路由器之间的数据库同步就通过可靠泛洪进行维护。收到的链路状态更新（Link State Update, LSU/LS Update）会被解析并将新的LSA存入到链路状态数据库。收到的链路状态确认（Link State Acknowledgment, LS ACK）分组会告诉本路由LSA已经被收到。详细可靠泛洪过程见。。。

路由器定期从每个接口发出Hello分组。LS Update在路由器生成新的LSA时发送，其他类型的分组在响应接收分组的时候才会发送。

如果发送的分组传输失败，则路由器内部的计时器会在重传时间到达时重新发送分组。

1. LSA处理

路由器保有的LSA集合组成了链路状态数据库。链路状态数据库的组织和实现见3.4。

当路由器从邻居路由器接收到一个LS Update分组时，或者自己创建一个新的LSA或LS Update时，都需要将LSA存入到数据库。

数据库中每一个LSA都会连续老化（age），这样保证了路由器创建的每一个LSA会被定期重新创建，同时也保证已经断开（down）的路由器创建的LSA最终可以从数据库中删除。LSA老化机制详见3.4.3。

每当一个新的LSA存入数据库中时，该LSA都会被泛洪到该路由器邻接的邻居路由器，同时，也会激发重新计算路由表。

1. 计时器处理

计时器协助实现了重传机制，保证了OSPF协议分组和LSA能够可靠的传送到邻居路由器。此外，计时器也维持了Hello分组定时发送和死掉的邻居结点的计时。

计时器同时也维护着OSPF路由器的一些功能，LSA数据库连续老化，最短路由计算都是通过1秒的计时器来维持。

### 主要数据结构

OSPF实现中主要数据结构的互相关系如图所示。

一个OSPF路由器会有多个接口与外部链接，每一个启动的接口，会被记录在OSPF类的接口表中。由于本实现并未实现多区域，所有接口都属于骨干区（区号0.0.0.0, Backbone），所以，所有有关区域的操作都由OSPF类完成。

每一个接口用一个类Interface表示，在每一个接口上OSPF路由器都尝试与其相邻的路由器建立会话。每一个会话都用一个Neighbor类表示，其中包含着维护数据库同步的状态信息。



1. OSPF类

class OSPF

**{**

const rtid\_t myid**;** //路由器ID

uns16 max\_retrans\_window**;** //最大传输窗口

byte max\_dds**;** //最多同时发送DD

//动态数据

Interface **\***ifcs**;**//接口列表

Table phyints**;** //物理接口列表

Table krtdeletes**;** //需要删除的内核路由

Table ospf\_member**;** //OSPF组播成员列表，现在仅有224.0.0.5和224.0.0.6的组播有用

uns16 ospf\_mtu**;**//所有接口最大的MTU

//以下两个构成一个等待建立邻接关系邻居链表

Neighbor **\***g\_adj\_head**;**//头

Neighbor **\***g\_adj\_tail**;**//尾

//DD处理

int n\_dbx\_nbrs**;** //正在进行数据库交换的邻居数

int local\_inits**;** //本地启动的数据库交换，表示是由于本路由器首先发送的DD引起的交换

int remote\_inits**;**//远程启动的数据库交换

//LSA处理

byte **\***build\_area**;**//组建LSA缓冲区（网络格式）

uns16 build\_size**;**//LSA缓冲区大小

同时，还提供了一些调用函数，来完成必要的功能：

byte **\***orig\_buff**;**//LSA缓冲区

uns16 orig\_size**;**//缓冲区大小

bool orig\_buff\_in\_use**;**//缓冲区是否被使用

//LSA数据库相关

uns32 db\_xsum**;**//LS数据库校验和

Table rtrLSAs**;**//router-LSAs

Table netLSAs**;**//network-LSAs

LsaList replied\_list**;**//最近发送的LSA，响应收到的旧的LSA

LsaList MaxAge\_list**;**//达到最大老化的LSA，需要被刷新

uns32 total\_lsas**;**//数据库中LSA总数

LsaList dbcheck\_list**;**//验证通过的LSA

LsaList pending\_refresh**;**//等待被刷新的LSA

//路由处理

bool need\_remnants**;** //是否需要内核上传路由信息

int full\_sched**:**1**;**//是否要进行路由规划

uns32 n\_dijkstras**;**//进行Dijkstras算法的次数

//泛洪

Pkt a\_update**;** //当前进行泛洪的update

Pkt a\_demand\_upd**;**//当前在需要泛洪的接口进行泛洪的update

public**:**

class rtrLSA **\***mylsa**;**//本路由产生的LSA

int n\_routers**;**//可到达的路由

class Interface **\*\***ifmap**;**//router-LSA与接口的映射关系

int ifmap\_valid**;**//映射关系有效？

int size\_ifmap**;**//映射关系表大小

int n\_ifmap**;**//当前映射关系表条目数

**}**

* int run\_fsm(FSMmachine \*table, int &i\_state,int event);

输入的状态转移表，状态和时间，来得到产生的动作和改变后的状态

* Interface \*find\_ifc(uns32 addr, int phyint = -1);

根据地址和物理端口号来查找端口

* Interface \*next\_ifc(uns32 addr, int phyint);

找到比所给出地址和物理端口号大的最小的一个端口

* Interface \*find\_ifc(Pkt \*pdesc);

根据数据包查找端口

* Interface \*find\_nbr\_ifc(InAddr nbr\_addr);

找到所给邻居地址的端口

* int getpkt(Pkt \*pkt,int type,uns16 size);

生成基本的IP包

* void freepkt(Pkt \*pkt);

释放包

* void delete\_down\_neighbors();

遍历邻居表，删除断开的邻居

* void multi\_join(int phyint,InAddr group);

加入多播组

* void multi\_leave(int phyint,InAddr group);

离开多播组

* void phy\_attach(int phyint);

添加启用的物理端口到记录中

* void phy\_detach(int phyint, InAddr addr);

删除端口记录

* void ConfigInterface(struct InterfaceConfig \*m,int opt);

设置端口

* void phy\_up(int phyint);

开启物理端口

* void phy\_down(int phyint);

关闭物理端口

* void resolvePkt(int phyint, InPkt \*pkt, int plen);

解析数据包

1. Interface类

class Interface

**{**

protected**:**

//需设置的变量

InMask if\_mask**;** //接口掩码

uns16 mtu**;** //接口mtu

int if\_IfIndex**;** //接口序号

int if\_type**;**//接口类型，目前只支持广播接口

int if\_cost**;**//接口花费

byte if\_rxmt**;**//接口重传间隔时间

byte if\_xdelay**;**//传输延迟时间

uns16 if\_hellointerval**;**//Hello发送间隔

uns32 if\_deadinterval**;**//邻居Dead时间

byte if\_drpri**;** //优先级

autyp\_t if\_autype**;**//验证类型 目前只支持空

byte if\_passwd**[**8**];**//验证密码

int passive**;** //被动接口，目前没有实现

int if\_mcfwd**;** //多播转发

bool igmp\_enabled**;** //是否开启组播

//根据设置和运行过程决定的

InAddr if\_net**;** //网段

InAddr if\_multi**;**//多播地址

InAddr if\_flood**;**//泛洪地址

InAddr if\_dr**;**//DR地址

InAddr if\_bdr**;**//BDR地址

class Neighbor **\***if\_dr\_n**;**//作为DR的邻居

int if\_state**;**//接口状态

Interface **\***next**;**//接口链表，下一个接口

Pkt if\_update**;**//需要发送的update

Pkt if\_dack**;**//延迟发送的ACK

bool recv\_update**;**//收到了Update，正在处理

bool flood**;**//是否要泛洪

//一些维护功能的计时器

WaitTimer if\_waitt**;**//等待计时器

HelloTimer\_Ifc if\_hellot**;**//Hello计时器

DAckTimer if\_ackt**;**//延迟发送ACK计时器

public**:**

InAddr if\_addr**;**//接口地址

int if\_physic**;**//对应的物理接口编号

同时，还提供了一些调用函数，来完成必要的功能：

class Neighbor **\***if\_Neighborlist**;**//邻居列表

int if\_Neighborcnt**;**//邻居数

int if\_fullcnt**;**//完全邻接的邻居数

**}**

* virtual void if\_send(Pkt \*, InAddr);

从端口发出数据包

* virtual void Neighbor\_send( Pkt \*, Neighbor \*);

向某邻居发数据包

* virtual Neighbor \*find\_Neighbor(InAddr, rtid\_t);

找到某邻居

* virtual void set\_IdAddr(Neighbor \*, rtid\_t, InAddr); //制止

邻居ID或地址

* inline int state();

得到状态

* inline InAddr net();

得到网络段

* inline InAddr mask();

得到掩码

* inline uns16 cost();

得到花费

* inline InAddr dr();

得到DR

* inline InAddr bdr();

得到BDR

* inline byte rxmt\_interval();

得到重传间隔

1. Neighbor类

class Neighbor

**{**

InAddr n\_addr**;** //邻居地址

rtid\_t n\_id**;** //邻居路由器ID

int n\_state**;**//邻居状态

byte n\_opts**;**//邻居发送分组的option段

byte n\_imms**;**//邻居发送DD的imms段

uns32 n\_ddseq**;**//DD序列号

bool database\_sent**;**//需要发送整个数据库？

int n\_adj\_pend**:**1**,**//等待建立邻接关系？

remote\_init**:**1**;**//由该邻居发起的DD交换？

Neighbor **\***next\_pend**;**//下一个等待建立邻接关系的邻居

//LSA重传列表

class LsaList pend\_retrans**;**//最近需要重传的LSA

class LsaList retrans**;**//已经泛洪的LSA，但还没有到重传时间

class LsaList failed\_retrans**;**//发送了LSA还没有收到确认

uns32 retrans\_cnt**;**//所有重传的LSA

uns16 retrans\_window**;**//重传窗口，一次性发送LSA最大数目

class LsaList ddlist**;**//DD表

class LsaList reqlist**;**//LSA Request表

int req\_goal**;**//还需发送的请求数

Pkt n\_update**;**//需要发送的update

Pkt n\_imack**;**//立即回复的ACK

Pkt n\_ddpkt**;**//当前发送的DD

InactTimer inactt**;**//判断邻居死掉的计时器

HelloTimer hellot**;**//发送hello的计时器

HoldTimer holdt**;**//用来保持slave模式发送出去的最后的DD，知道master收到了该分组为止

DDTimer ddt**;**//DD重传计时器

LSRTimer lsrt**;**//LSReq重传计时器

LSATimer lsat**;**//LSUpdate重传计时器

ProgressTimer progt**;**//触发时表示相邻路由的数据库交换被阻断，需要重启

protected**:**

Neighbor **\***next**;** //邻居列表中的下一个

Interface **\***n\_ifc**;**//与之相关的接口

public**:**

byte prior**;**//优先级

InAddr n\_dr**;**//DR地址

InAddr n\_bdr**;**//BDR地址

}

同时，还提供了一些调用函数，来完成必要的功能：

* inline Interface \*ifc();

得到对应的端口

* inline int state();

得到当前状态

* inline int declared\_dr();

宣称是DR？

* inline int declared\_bdr();

宣称是BDR？

* inline int is\_dr();

是DR？

* inline int is\_bdr();

是BDR？

* inline InAddr addr();

得到地址

* inline rtid\_t id();

得到路由ID

* inline int priority();

得到优先级

* inline void build\_imack(LShdr \*hdr);

组建立即的ACK

* void start\_adjacency();

开始建立邻接关系

* void AddPendAdj();

加入到等待建立邻接表

* void DelPendAdj();

从等待建立邻接表删除

* bool adv\_as\_full();

已经是FULL状态？

### 全局变量

OSPF \*ospf; OSPF实例

Queuetimerq; 全局计时器队列

SysCalls \*sys; 系统调用接口

INtbl \*inrttbl; IP路由表

FWDtbl \*fa\_tbl; 转发地址表

INrte \*default\_route; 默认路由表表项(0/0)

PatriciaTree EPath::nhdb; 下一跳数据库

### 源文件组织

|  |  |  |  |
| --- | --- | --- | --- |
| arch.h | LsaList.h | ospfDD.cpp | phyint.h |
| define.h | LShdr.h | ospfHello.cpp | Queue.cpp |
| include.h | main.cpp | ospfOrig.cpp | Queue.h |
| Interface.cpp | Neighbor.cpp | ospfPkt.h | rtrLSA.cpp |
| Interface.h | Neighbor.h | ospfRequest.cpp | RtrTblEntry.cpp |
| InterfaceFSM.cpp | NeighborFSM.cpp | ospfType.h | RtrTblEntry.h |
| InterfaceFSM.h | NeighborFSM.h | ospfUpdate.cpp | system.cpp |
| IP.h | netLSA.cpp | ospfUtil.cpp | system.h |
| List.h | OSPF.cpp | ospfUtil.h | Table.cpp |
| LSA.cpp | OSPF.h | Patricia.cpp | Table.h |
| LSA.h | ospfAck.cpp | Patricia.h | Timer.cpp |
| LsaList.cpp | ospfCalc.cpp | phyint.cpp | Timer.h |

本程的源文件大部分由与其相关的功能或结构来阻止，某源文件或头文件中实现或声明的函数或变量都与其文件名有关。

此外，arch.h定义了一些如版本号、默认掩码、骨干区域等与OSPF有关的常量，define.h定义了一些在本程序中用到的变量类型，以及实现了进行字节转换的函数。include.h描述了一个必须引用的头文件列表，它则会在其他源文件中引用，而不是将所有要引用的头文件写上去。

## OSPF实用类

这里介绍本程序使用的一些实用类。主要介绍这些类的数据结构和接口函数。关于实现细节可查阅相关资料或阅读代码。

### 基于AVL树的列表

AVL树是最先发明的自平衡二叉查找树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是O(log n)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

AVL树适用于分类排序和搜索具有固定长度且相对较短的关键字的节点。

相关代码如下：

class TblItem **{**

uns32 \_index1**;** // 首要的比较值

uns32 \_index2**;** // 次要的比较值

TblItem **\***right**;** // 右结点

TblItem **\***left**;** // 左节点

int16 balance**:**3**,** // AVL平衡因子

in\_db**:**1**;** // 在数据库中？

int16 refct**;** // 被引用数

public**:**

TblItem **\***sll**;** // 在顺序列表中的下一个结点Next in ordered list

**};**

//Table结构，对外表现出是一个顺序表，内部用AVL来维持

class Table **{**

TblItem **\***\_root**;** //AVL根结点

uns32 count**;** //树上节点数

uns32 instance**;** // 加入或减少动作

public**:**

TblItem **\***sllhead**;** // 顺序表的头部

TblItem **\***find**(**uns32 key1**,** uns32 key2**=**0**);**//查找一个结点

TblItem **\***previous**(**uns32 key1**,** uns32 key2**=**0**);**//找到所给关键字最靠近的前面的一个结点

void add**(**TblItem **\*);** // 向树上加一个结点

void remove**(**TblItem **\*);** // 从树上删除一个结点

void clear**();**

friend class TblSearch**;**

**};**

//搜索AVL树，深度优先搜索。

class TblSearch **{**

Table **\***tree**;** // 需要搜索的表

uns32 instance**;** // 相关的条目

uns32 c\_index1**;** // 当前的位置index1

uns32 c\_index2**;** // 当前的位置index2

在本程序中，如下的数据结构被组织成基于AVL树的列表：

TblItem **\***current**;**//当前的条目

public**:**

void seek**(**uns32 key1**,** uns32 key2**);** //设置起点

TblItem **\***next**();**//第一次调用时，返回最小关键字的结点，下一次返回次小关键字结点，依次类推

**};**

* 链路状态数据库。主要的比较值\_index1被设置为LSA的ID，次要的比较值是\_index2被设置为LSA通告路由器。
* 路由表。index1被设置为网短号，index2为网络掩码。
* 物理接口数据库。Index为物理接口编号，index2为0。

### Patricia树

Patricia树对具有长关键字的结点进行分类和排序效果十分好，是一种变基数的树结构。它允许关键字保存在内部节点和树叶上来减少对存储空间的占用。

//Patricia树中的结点

class PatriciaEntry

**{**

PatriciaEntry **\***zeroptr**;**//0指针

PatriciaEntry **\***oneptr**;**//1指针

uns32 chkbit**;**

public**:**

byte **\***key**;**//关键字

int keylen**;**//关键字长

inline bool bit\_check**(**int bit**);**

friend class PatriciaTree**;**

**};**

//Patricia树

class PatriciaTree

**{**

PatriciaEntry **\***root**;**

int size**;**

public**:**

PatriciaTree**();**

void init**();**

void add**(**PatriciaEntry **\*);**//更加一个条目

PatriciaEntry **\***find**(**byte **\***key**,** int keylen**);**//根据关键字找到条目

PatriciaEntry **\***find**(**char **\***key**);**

void remove**(**PatriciaEntry **\*);**//删除一个条目

void clear**();**

void clear\_subtree**(**PatriciaEntry **\*);**//删除子树

**};**

在本程序中，如下的数据结构被组织成基于Patricia树的列表：

* 在路径计算中，多路径数据库EPath::nhdb被组织成Patricia树。其中，关键字定义为每一个数据项即一个结点的下一条的数组。
* 物理接口数据库也是一个Patricia树，关键字是物理接口的名称。

### 优先级队列

优先级队列总是将具有最小代价的成员放在队首，并且对于添加、删除或者修改，操作代价都为O(log n)。

相关数据结构和接口函数如下：

//优先级队列的一个元素

class Q\_Elt

**{**

Q\_Elt **\***left**;**//左节点

Q\_Elt **\***right**;**//右结点

Q\_Elt **\***parent**;**

uns32 dist**;**

protected**:**//比较顺序：cost0 cost1 tie1 tie2

uns32 cost0**;**

uns32 cost1**;**

byte tie1**;**

uns32 tie2**;**

public**:**

friend class Queue**;**

friend class OSPF**;**

inline Q\_Elt**();**

inline bool cmp\_cost**(**Q\_Elt **\*);**//如果自己比传入的参数花费少则返回true

friend int main**(**int argc**,**char **\***argv**[]);**

**};**

class Queue

**{**

Q\_Elt **\***root**;**

void adjust**(**Q\_Elt **\***a**,**int del**);**

void combine**(**Queue **&**q**);**

public**:**

inline Queue**();**

inline Q\_Elt **\***gethead**();**//返回队列头部

Q\_Elt **\***rmhead**();**//将头部从队列去掉并返回原来的头部

void add**(**Q\_Elt **\***item**);**

void del**(**Q\_Elt **\***item**);**

**};**

在本程序中，如下的数据结构被组织成优先级队列：

* 路由计算中，备选列表是用优先队列实现的。cost0为从计算节点到目的地的路径代价；cost1为0；tie1是LSA的LS类型，tie2是LSA链路状态ID。
* 计时器。Cost0为到计时器激活时已经过去的秒数；cost1是毫秒数。

### 计时器

本程序实现了两种功能的计时器。一种是单触发计时器Timer，另一种是周期出发计时器ITimer，ITimer从Timer继承而来。

启动计时器时要调用start()方法。milliseconds 参数指定了计时器计时区间长度􀉖以毫秒为单位。在计时器计时停止或重新启动之前，如果时间量等于计时器的满期时间，则称计时器已到期限，且调用其方法action()。满期的单触发计时器将进入停止状态，而满期的间隔计时器则总是重新启动。

可以调用Timer::is\_running()来辨别一个计时器当前是否在运行。也可以调用Timer::restart()重新设置运行中计时器的计时区间。如果不再有restart() 或stop() 调用，则计时器从触发开始将一直运行直到将来满期。

本程序用到的计时器及其解释如下：

* WaitTimer：延迟接口上的指定路由器计算，直到发现后备指定路由器存在。
* HelloTimer\_Ifc：定期从某个端口发出Hello的计时器
* DAckTimer：延迟链路状态确认分组在接口上的组播，先会将要ACK的LSA加入到update中，直到到时间将其发出。
* InactTimer：用于邻居活跃计时，触发时表示在deadtime时间内没有收到hello，应该断开邻接关系
* HoldTimer：用来保持以slave模式送出去的最后的数据库描述分组，直到主路由器收到了该分组为止。
* DDTimer：该计时器触发导致主路由器重新发送其刚刚发送过的数据库描述分组
* LSRTimer：该计时器触发将导致一个路由器向相邻路由器重新发送链路状态请求分组。
* LSATimer：该计时器触发导致LSA重新传输到相邻路由器，从而实现可靠性泛洪算法。
* ProgressTimer：该计时器触发指示与相邻路由器的数据库交换处理被阻断，并且要求隔一段时间后重新启动。
* DBageTimer：每隔1s触发一次，完成链路状态数据库老化及其他数据库内部管理工作，包括在必要的时候运行完整的路由选择计算。

## 维护系统设计

维护系统在本程序中是SysCalls类，为OSPF路由进行了必要的网络设置，并且提供了一系列与内核交互的接口函数。

### 功能

维护系统主要记录了OSPF路由运行过程中的网络socket、UDP socket、组播socket和Router socket。其中网络socket主要用来获取和发送数据包；UDP socket主要用来与内核通信并进行设置；组播socket暂没有设置，但可以进行组播扩展；Router socket主要用来获取和设置内核路由。

此外，维护系统还记录了物理接口列表，接口与地址对应关系表，以及OSPF得到了一些直连路由。

### 初始化过程

在开始OSPF路由功能之前，需要初始化一个维护系统的实例。初始化过程中，需要对如系统时间等全局变量进行赋值，并且需要打开上述socket并进行设置。

### 重要的接口函数

1. 打开IP转发功能

函数原型： void SysCalls**::**ip\_forward**(**bool enable**)**

传入参数： bool enable; 打开或者关闭

函数功能： 使用sysctl，根据传入参数enable开启或关闭IP转发功能。

1. 发送数据包

函数原型： void SysCalls**::**sendpkt**(**InPkt **\***pkt**,**int phyint**,**InAddr gw**)**

传入参数： InPkt \*pkt; 需要发送的数据包

int phyint; 发送通过的屋里端口号。

InAddr gw; 目的地址。

函数功能： 该函数首先为pkt设置目的地址和校验和，然后判断目的地址是否是组播地址，如果是，将socket设置为组播。然后，设置好msg，调用sendmsg发送消息。

1. 设置路由ID

函数原型： int SetRouterID**(**char **\***buf**)**

传入参数： char \*buf; 从设置文件读取的ID

函数功能： 由于设置文件中的ID是用点分十进制表示，首先需要对ID进行转化，然后会创建一个OSPF实例。返回设置的路由ID。

1. 设置端口

函数原型： void SysCalls**::**SetInterface**(**char **\***buf**)**

传入参数： char \*buf; 从设置文件读取端口名称或IP

函数功能： 设置文件中，可以写端口的IP或名称。首先判断传入的参数是否是IP，再查找相应的端口，并设置一些列参数。并调用ospf的ConfigInterface为路由设置端口。

1. 读取配置

函数原型： void SysCalls**::**configure**()**

传入参数： 无

函数功能： 读取配置文件，设置路由ID和端口。

1. 读取内核物理端口

函数原型： void SysCalls**::**read\_kernel\_interfaces**()**

传入参数： 无

函数功能： 通过ioctl来读取内核端口，并存入系统的端口表中。

1. 接受原始数据包

函数原型： void SysCalls**::**raw\_receive**(**int fd**)**

传入参数： int fd; socket ID

函数功能： 调用recvmsg从fd收取消息。recvmsg是阻塞调用，但如果没有收到消息会在一定时间内被系统终止。收取到消息后，调用ospf类的resolvePkt函数对OSPF协议消息进行后续处理。

1. 加入多播组

函数原型： void SysCalls**::**join**(**InAddr group**,**int phyint**)**

传入参数： InAddr group; 多播组地址

Int phyint; 要加入多播组的端口号

函数功能： 通过设置socket让指定端口加入某多播组。本程序暂时只让端口加入OSPF多播组。

1. 离开多播组

函数原型： void SysCalls**::**leave**(**InAddr group**,**int phyint**)**

传入参数： InAddr group; 多播组地址

Int phyint; 要离开多播组的端口号

函数功能： 通过设置socket让指定端口离开某多播组。

1. 向内核添加路由

函数原型： void SysCalls**::**rtadd**(**InAddr net**,** InMask mask**,** EPath **\***mpp**,**

EPath **\***ompp**,** bool reject**)**

传入参数： InAddr net; 网段

InMast mask; 掩码

EPath \*mpp; 最新的后继路径

EPath \*ompp; 旧的后继路径

Bool reject; 是否是不可达路由

函数功能： 通过Linux IPv4 routing socket来添加路由。如果存在直接连接的路由，会根据旧的路径删除直连路由。否则，向内核添加最新的路由。如果reject为true，则添加的路由类型被声明为不可达。

1. 从内核删除路由表

函数原型： void SysCalls**::**rtdel**(**InAddr net**,** InMask mask**,** EPath **\***ompp**)**

传入参数： InAddr net; 网段

InMast mask; 掩码

EPath \*ompp; 旧的后继路径

函数功能： 通过Linux IPv4 routing socket来删除路由。

1. 为开启的端口添加直连路由

函数原型： void SysCalls**::**add\_interface\_direct**(**InAddr net**,** InMask mask**,**

InAddr addr**,**bool reject**)**

传入参数： InAddr net; 网段

InMast mask; 掩码

InAddr addr; 目的地址

bool reject; 是否是不可达路由

函数功能： reject一般为false。函数先检查直连路由表，如果已经存在该直连路由，先删除，然后向其中添加两条路由。第一条路由是路由器某接口网段的路由，路由目的地址为该接口IP。第二条路由是到路由器接口的路由，掩码为32位，目的地址为回环地址。

## 邻居路由器维护

### 邻居状态机

每一个相邻路由器由一个Neighbor类表示。它的状态保存在n\_state中。程序中定义的状态如下：邻居状态机状态编码

enum

**{**

NBE\_HELLORCVD **=** 1**,** // 接收到Hello分组

NBE\_START**,** // 开始发送Hello

NBE\_2WAYRCVD**,** // 双向

NBE\_NEGDONE**,** // 协商主从

NBE\_EXCHANGEDONE**,** // 完成了数据库交换

NBE\_BADLSREQ**,** // 收到的LS request有错

NBE\_LOADINGDONE**,** // 读取完毕

NBE\_EVAL**,** // 评估是否要形成邻接，与本OSPF允许的最多同时进行数据库交换数有关

NBE\_DDRCVD**,** // 收到有效的DD

NBE\_SEQNOMISM**,** // DD序号错误

NBE\_1WAYRCVD**,** // 单向

NBE\_KILLNBR**,** // 断开邻居

NBE\_INACTIVE**,** // 一段时间内没有收到Hello

NBE\_LLDOWN**,** // 链路断开

enum

**{**

NBS\_DOWN **=** 0x01**,** // 邻居断开

NBS\_ATTEMPT **=** 0x02**,** // 尝试发送Hello(NBMA)

NBS\_INIT **=** 0x04**,** // 单向通信

NBS\_2WAY **=** 0x08**,** // 双向通信

NBS\_EXSTART **=** 0x10**,** // 协商主从

NBS\_EXCHANGE **=** 0x20**,** // 开始发送DD

NBS\_LOADING **=** 0x40**,** // DD发送完毕，现在只有LS请求

NBS\_FULL **=** 0x80**,** // 完全邻接

NBS\_ACTIVE **=** 0xFE**,** // 除断开之外的所有状态

NBS\_FLOOD **=** NBS\_EXCHANGE **|** NBS\_LOADING **|** NBS\_FULL**,** //泛洪

NBS\_ADJFORM **=** NBS\_EXSTART **|** NBS\_FLOOD**,** //邻接形成

NBS\_BIDIR **=** NBS\_2WAY **|** NBS\_ADJFORM**,** //双向

NBS\_PRELIM **=** NBS\_DOWN **|** NBS\_ATTEMPT **|** NBS\_INIT**,** //双向连接建立之前的状态

NBS\_ANY **=** 0xFF**,** // All states

**};**

邻居状态机事件编码

NBE\_ADJTMO**,** // 形成邻接超时

**};**

邻居状态机动作编码

enum

**{**

NBA\_START **=** 1**,** // 开始与邻居连接

NBA\_RESTART\_INACTT**,** // 重置邻居不活动计时器

NBA\_START\_INACTT**,** // 开始邻居不活动计时器

NBA\_EVAL1**,** // 本机开始发送DD

NBA\_EVAL2**,** // 收到对方DD，本机开始发送DD

NBA\_SNAPSHOT**,** // 进行数据库快照

NBA\_EXCHANGEDONE**,** // 数据库交换完毕后进行一系列动作

NBA\_REEVAL**,** // 重新评估是否应该与邻居形成邻接

NBA\_RESTART\_DD**,** // 重新开始数据库发送

NBA\_DELETE**,** // 删除邻居

NBA\_CLR\_LISTS**,** // 清空数据库

NBA\_HELLOCHK**,**

**};**

本程序实现的邻居状态机状态转移图：



本程序对于新的邻居状态处理：

**if** **(**n\_state **==** NBS\_FULL**)**

**{**

n\_ifc->if\_fullcnt++;

ospf**->**n\_dbx\_nbrs**--;**//需要交换数据库的邻居数-1

progt**.**stop**();**

exit\_dbxchg**();**

ospf**->**rl\_orig**(false);**//形成邻接关系了，就需要生成router LSA

**}**

//开始进行数据库交换

**else** **if** **(**n\_ostate **<=** NBS\_EXSTART **&&** n\_state **>** NBS\_EXSTART**)**

ospf**->**n\_dbx\_nbrs**++;**

**else** **if** **(**n\_ostate **==** NBS\_FULL**)**//从FULL变到其他状态

**{**

n\_ifc**->**if\_fullcnt**--;**

ospf**->**rl\_orig**(false);**

**}**

**else** **if** **(**n\_state **<=** NBS\_2WAY **&&** n\_ostate **>=** NBS\_EXSTART**)**

**{**

exit\_dbxchg**();**

**if** **(**n\_ostate **>** NBS\_EXSTART**)**

ospf**->**n\_dbx\_nbrs**--;**

**}**

// 如果是DR，重新生成network lsa

**if** **(**n\_ifc**->**state**()** **==** IFS\_DR**)**

n\_ifc**->**nl\_orig**(false);**

**if** **((**n\_state **>=** NBS\_2WAY **&&** n\_ostate **<** NBS\_2WAY**)** **||**

**(**n\_state **<** NBS\_2WAY **&&** n\_ostate **>=** NBS\_2WAY**))**

n\_ifc**->**ifc\_fsm**(**IFE\_NCHG**);**//邻居可能发生变化，驱动接口状态机

### 接口状态机

接口状态机状态编码

enum

**{**

IFS\_DOWN **=** 0x01**,** //接口被关闭

IFS\_LOOP **=** 0x02**,** //接口循环

IFS\_WAIT **=** 0x04**,** //接口等待确定哪一个为BDR

IFS\_PP **=** 0x08**,** //点到点

接口状态机事件编码

enum

**{**

IFE\_UP **=** 1**,** //端口开启

IFE\_WTIM**,** //等待计时器出发

IFE\_BSEEN**,** //发现BDR

IFE\_NCHG**,** //邻居状态发生变化

IFE\_LOOP**,** //接口循环

IFE\_UNLOOP**,** //接口解除循环

IFE\_DOWN**,** //接口关闭

N\_IF\_EVENTS **=** IFE\_DOWN**,** //接口事件数

**};**

IFS\_OTHER **=** 0x10**,** //既不是DR也不是BDR

IFS\_BACKUP **=** 0x20**,** //BDR

IFS\_DR **=** 0x40**,** //DR

N\_IF\_STATES **=** 7**,** //接口状态数

IFS\_MULTI **=** **(**IFS\_OTHER **|** IFS\_BACKUP **|** IFS\_DR**),**//多路状态

IFS\_ANY **=** 0x7F**,** //所有状态

**};**

接口状态机动作编码

enum

**{**

IFA\_START **=** 1**,** //接口开启

IFA\_ELECT**,** //接口选举

IFA\_RESET**,** //接口重置

**};**

本程序实现的接口状态机状态转移图：



接口状态变化后采取的动作：

**if(**if\_ostate**==**if\_state**)**

**return;**

//设置泛洪中使用的地址

if\_flood**=(**if\_state**==**IFS\_OTHER**)?**AllDRouters**:**AllSPFRouters**;**

//接口关闭，则清除数据库

**if(**if\_ostate**==**IFS\_DOWN**||**if\_state**==**IFS\_DOWN**)**

**{**

delete\_lsdb**();**

**}**

//加入多播组，状态是DR或BDR

**if(**if\_state**>**IFS\_OTHER**&&**if\_ostate**<=**IFS\_OTHER**)**

ospf**->**multi\_join**(**if\_physic**,**AllDRouters**);**

//离开多播组

**else** **if(**if\_state**<=**IFS\_OTHER**&&**if\_ostate**>**IFS\_OTHER**)**

ospf**->**multi\_leave**(**if\_physic**,**AllDRouters**);**

//重新创建LSA

ospf**->**rl\_orig**(false);**

**if(**if\_state**==**IFS\_DR**||**if\_ostate**==**IFS\_DR**)**

**{**

nl\_orig**(false);**

**}**

## 链路状态数据库

链路状态数据库是OSPF实现的核心。数据库为路由计算提供了原始数据，并且OSPF中大部分协议都与路由器之间的数据库同步有关。

### LSA及数据库实现

1. LSA数据结构

本程序将LSA表示为一个类。主要的属性如下：

class LSA **:** public TblItem

**{**

protected**:**

// LS头部，机器字节序存储

age\_t lsa\_rcvage**;** // 收到LSA时的年林

byte lsa\_opts**;** // LS 选项

const byte lsa\_type**;** // 类型

seq\_t lsa\_seqno**;** // 序列号

xsum\_t lsa\_xsum**;** // 校验和

uns16 lsa\_length**;** // LSA字节长度

byte **\***lsa\_body**;** // LSA 内容

class Interface **\***lsa\_ifp**;**// 链接范围内LSA的接口

LSA **\***lsa\_agefwd**;** // 老化列表中向前列表

LSA **\***lsa\_agerv**;** // 老化列表中向后列表

uns16 lsa\_agebin**;** // 老化列表

uns16 lsa\_rxmt**;** // 所在重发列表的数目，也是必须对泛洪进行确认的相邻路由的数目

uns16 in\_agebin**:**1**,** // 在老化列表中？

deferring**:**1**,** // 等待延迟重新创建

changed**:**1**,** // 自从上次泛洪之后发生了变化

exception**:**1**,** //2-类外部路由度量

parsed**:**1**,** // 被解析进行路由计算

sent\_reply**:**1**,** // 发送对接收到旧的LSA的回复

checkage**:**1**,** // 排队等待校验

min\_failed**:**1**,** // MinArrival失败

we\_orig**:**1**;** // 自己生成

uns16 lsa\_hour**;** // 停留在数据库中的小时数

**}**

此外，还提供了一些虚函数来实现不同类型的LSA必要的功能。

* void LSA::parse()。当一个LSA 添加到链路状态数据库中时，对其预解析，以用于后面的路由选择计算
* void LSA::unparse()。从数据库中移出一个LSA 时，取消以前对其所做的处理，即对其反解析，从而使得LSA 不再用于路由选择计算。
* void LSA::build()。从数据库中的LSA 副本中重新建立LSA 的网络格式，从而保证该LSA 能被泛洪到路由器的相邻路由器中。
* void LSA::delete\_actions()。从数据库中删除一个LSA 时，根据不同类型的LSA 作不同的删除处理。
* void LSA::reoriginate (int forced)。根据反映本地路由新状态的内容重新创建LSA。如果forced 为true，则即使LSA 的内容没有改变也要对其重新创建。

1. LSA数据库组织

因为本程序只实现单区域，所以将所有LSA数据库均设置到OSPF类中。用rtrLSAs存储router-LSA，netLSAs存储network-LSA，db\_xsum是数据库的校验和。

1. LSA数据库操作

* 在数据库中查找LSA

一个LSA 在链路状态数据库中的位置由函数

LSA \*OSPF::FindLSA(Interface \*ip,byte lstype, lsid\_t lsid, rtid\_t rtid)确定。参数ldtype、lsid 和rtid 分别是LSA 的LS 类型、链路状态ID 号以及通告该LSA 的路由器。

为查找路由器自创建的LSA，需要使用

LSA \*OSPF::myLSA(Interface \*ip,byte lstype,lsid\_t lsid)函数。

当只需要查找LSA 所属于的数据库时，则使用函数

Table \*OSPF::FindLSdb(Interface \*ip,byte lstype)

* 向数据库添加一个LSA

//向数据库中添加LSA

LSA **\***OSPF**::**AddLSA**(**Interface **\***ip**,**LSA **\***current**,**LShdr **\***hdr**,**bool changed**)**

**{**

LSA **\***lsap**;**

int blen**;**

RtrTblEntry **\***old\_rte **=** 0**;**

bool min\_failed**=false;**

blen **=** ntoh16**(**hdr**->**ls\_length**)** **-** **sizeof(**LShdr**);**

**if** **(**current**)**

//如果存在一个当前的LSA，则将其替换掉

**{**

min\_failed **=** current**->**since\_received**()** **<** MinArrival**;**

old\_rte **=** 0**;**

current**->**stop\_aging**();**

update\_lsdb\_xsum**(**current**,** **false);**

**}**

**if** **(**current **&&** current**->**refct **==** 0**)**

//如果存在一个LSA，而且不被任何LSA列表引用，

//则直接存入或更新

**{**

// 更新

**if** **(**changed**)**

UnParseLSA**(**current**);**

lsap **=** current**;**

lsap**->**hdr\_parse**(**hdr**);**

lsap**->**start\_aging**();**

lsap**->**changed **=** changed**;**

lsap**->**deferring **=** **false;**

lsap**->**rollover **=** current**->**rollover**;**

lsap**->**min\_failed **=** min\_failed**;**

**if** **(!**changed**)**

**{**

update\_lsdb\_xsum**(**lsap**,** **true);**

**return(**lsap**);**

**}**

**}**

**else**

//如果不能使用当前的LSA，则新建一个

**{**

**switch** **(**hdr**->**ls\_type**)**

**{**

**case** LST\_RTR**:**

lsap **=** **new** rtrLSA**(**hdr**,** blen**);**

**break;**

**case** LST\_NET**:**

lsap **=** **new** netLSA**(**hdr**,** blen**);**

**break;**

**default:**

lsap **=** 0**;**

**break;**

**}**

**if** **(!**current**)**

lsap**->**changed **=** **true;**

**else**

// 如果先前存在，则需要反解析

**{**

lsap**->**changed **=** changed**;**

lsap**->**rollover **=** current**->**rollover**;**

lsap**->**min\_failed **=** min\_failed**;**

**if** **(**current**->**lsa\_rxmt **!=** 0**)**

lsap**->**changed **|=** current**->**changed**;**

lsap**->**update\_in\_place**(**current**);**

UnParseLSA**(**current**);**

**}**

//开始老化

lsap**->**start\_aging**();**

**}**

//解析新的LSA

ParseLSA**(**lsap**,** hdr**);**

update\_lsdb\_xsum**(**lsap**,** **true);**

//如果发生变化，则需要重新计算路由

**if** **(**changed**)**

**{**

rtsched**(**lsap**,** old\_rte**);**

**}**

**return(**lsap**);**

**}**

* 预解析LSA

//解析LSA，将网络格式的LSA转化为本机存储的LSA

void OSPF**::**ParseLSA**(**LSA **\***lsap**,** LShdr **\***hdr**)**

**{**

int blen**;**

//如果已经被解析过，则不需要再解析了

**if** **(**lsap**->**parsed**)**

**return;**

blen **=** ntoh16**(**hdr**->**ls\_length**)** **-** **sizeof(**LShdr**);**

//除了到达最大年龄的LSA之外

//其他LSA需要调用与自己类型合适的解析程序解析

**if** **(**lsap**->**lsa\_age**()** **!=** MaxAge**)**

**{**

lsap**->**exception **=** **false;**

lsap**->**parsed **=** **true;**

lsap**->**parse**(**hdr**);**

total\_lsas**++;**

**}**

**else**

lsap**->**exception **=** **true;**

**delete** **[]** lsap**->**lsa\_body**;**

lsap**->**lsa\_body **=** 0**;**

**if** **(**lsap**->**exception**)**

**{**

//需要保留整个LSA内容

lsap**->**lsa\_body **=** **new** byte**[**blen**];**

memcpy**(**lsap**->**lsa\_body**,** **(**hdr **+** 1**),** blen**);**

**}**

**}**

* 从数据库删除一个LSA

//当LSA被新的实例代替时，旧的就要从LSA删除，

void OSPF**::**DeleteLSA**(**LSA**\*** lsap**)**

**{**

Table **\***table**;**

//更新数据库校验和

update\_lsdb\_xsum**(**lsap**,** **false);**

//停止老化

lsap**->**stop\_aging**();**

//反解析

UnParseLSA**(**lsap**);**

table **=** FindLSdb**(**lsap**->**lsa\_ifp**,** lsap**->**lsa\_type**);**

//从数据库表中移除

table**->**remove**((**TblItem **\*)** lsap**);**

lsap**->**delete\_actions**();**

//检查引用，并删除

lsap**->**chkref**();**

**}**

* 反解析LSA

//当从数据库删除一个LSA时，需要撤销对他的解析处理，

//从而保证路由计算时不会被使用，也不会被其他的引用

//之后，便从数据库中删除

void OSPF**::**UnParseLSA**(**LSA **\***lsap**)**

**{**

**if** **(**lsap**->**parsed**)**

**{**

//调用类型专用的反解析程序

lsap**->**parsed **=** **false;**

lsap**->**unparse**();**

total\_lsas**--;**

**}**

**}**

* 组建网络格式LSA

//组建网络格式的LSA

LShdr **\***OSPF**::**BuildLSA**(**LSA **\***lsap**,** LShdr **\***hdr**)**

**{**

int blen**;**

**if** **(**hdr **==** 0**)**

**{**

**if** **(**lsap**->**lsa\_length **>** build\_size**)**

**{**

**delete** **[]** build\_area**;**

build\_size **=** lsap**->**lsa\_length**;**

build\_area **=** **new** byte**[**lsap**->**lsa\_length**];**

**}**

hdr **=** **(**LShdr **\*)** ospf**->**build\_area**;**

**}**

//填入标准的头部

**\***hdr **=** **\***lsap**;**

//填入内容

**if** **(!**lsap**->**exception**)**

//调用类型专用的组建函数

lsap**->**build**(**hdr**);**

**else**

//之前已经存过内容，直接拷贝

**{**

blen **=** lsap**->**lsa\_length **-** **sizeof(**LShdr**);**

memcpy**((**hdr **+** 1**),** lsap**->**lsa\_body**,** blen**);**

**}**

**return(**hdr**);**

**}**

### LSA老化

1. LSA老化的实现

所有的LSA 包含一个LS 年龄字段，该字段指明从创建时起LSA 已经生存的时间，以秒计。OSPF可以通过年龄知道LSA合适应该被刷新，合适应该删除旧的LSA。

数据库中每一个LSA的年龄每秒必须加1，但不需要每秒都处理数据库中所有的LSA。可以将所有相同年龄的LSA放入一个老化列表，它的年龄就是列表的编号与年龄为0列表编号的差值。每一秒钟只需要修改年龄为0的列表即可。

1. 由老化触发的行为

//数据库老化行为

void OSPF**::**dbage**()**

**{**

// 每秒增加一次年龄

LSA**::**Bin0**++;**

**if** **(**LSA**::**Bin0 **>** MaxAge**)**

LSA**::**Bin0 **=** 0**;**

// 处理特定年龄的LSA

//重新创建为保持了两次创建间隔为MinLSInterval而被延迟创建的

//LSA

deferred\_lsas**();**

//检查特定年龄LSA校验和。

checkages**();**

//刷新LSA

refresh\_lsas**();**

//对达到最大年龄的LSA进行处理

maxage\_lsas**();**

do\_refreshes**();**

// 完成由于老化到最大年龄的LSA进行泛洪

send\_updates**();**

// Check to see whether any MaxAge can be deleted

free\_maxage\_lsas**();**

**}**

1. 释放LSA

达到最大年龄的LSA 在其移出条件满足以前，会一直留在链路状态数据库和LSA 最大年龄列表MaxAge\_list 中。移出条件是否满足由函数free\_maxage\_lsas()判断。达到最大年龄的LSA 不被装入任何的年龄列表中，也没有老化处理。

//遍历达到最大年龄LSA列表，看是否有符合条件的能够被释放

void OSPF**::**free\_maxage\_lsas**()**

**{**

LSA **\***lsap**;**

LsaListIterator iter**(&**MaxAge\_list**);**

**while** **((**lsap **=** iter**.**get\_next**()))**

**{**

**if** **(!**lsap**->**valid**())**

**{**

iter**.**remove\_current**();**

**continue;**

**}**

**if** **(**lsap**->**lsa\_rxmt **!=** 0**)**

**continue;**

**if** **(!**maxage\_free**(**lsap**->**ls\_type**()))**

**continue;**

//可以释放，从数据库中移除

iter**.**remove\_current**();**

**if** **(**lsap**->**rollover**)**

**{**

lsap**->**rollover **=** **false;**

lsap**->**refresh**(**InvalidLSSeq**);**

**}**

**else**

ospf**->**DeleteLSA**(**lsap**);**

**}**

**}**

### LSA创建

1. 创建对应类型的LSA

* Router LSA

//创建Router LSA

void OSPF**::**rl\_orig**(**int forced**)**

**{**

LSA **\***olsap**;**

LShdr **\***hdr**;**

RTRhdr **\***rtrhdr**;**

int maxlen**;**

int maxifc**;**

int length**;**

Interface **\***ip**;**

IfcList iiter**(this);**

seq\_t seqno**;**

RtrLink **\***rlp**;**

// 得到当前LSA数据库

olsap **=** ospf**->**myLSA**(**0**,**LST\_RTR**,** ospf**->**my\_id**());**

// 计算LSA最大长度

maxlen **=** **sizeof(**LShdr**)** **+** **sizeof(**RTRhdr**);**

**while** **((**ip **=** iiter**.**get\_next**()))**

maxlen **+=** ip**->**rl\_size**();**

// 得到一个序列号

seqno **=** ospf**->**ospf\_get\_seqno**(**LST\_RTR**,** olsap**,** forced**);**

**if** **(**seqno **==** InvalidLSSeq**)**

**return;**

maxifc **=** maxlen**/sizeof(**RtrLink**);**

**if** **(**maxifc **>** size\_ifmap**)**

**{**

**delete** **[]** ifmap**;**

ifmap **=** **new** Interface**\*** **[**maxifc**];**

size\_ifmap **=** maxifc**;**

**}**

n\_ifmap **=** 0**;**

ifmap\_valid **=** **true;**

// 组建LSA头部

hdr **=** ospf**->**orig\_buffer**(**maxlen**);**

hdr**->**ls\_opts **=**0**;**

hdr**->**ls\_opts **|=** SPO\_EXT**;**

hdr**->**ls\_type **=** LST\_RTR**;**

hdr**->**ls\_id **=** hton32**(**ospf**->**my\_id**());**

hdr**->**ls\_org **=** hton32**(**ospf**->**my\_id**());**

hdr**->**ls\_seqno **=** **(**seq\_t**)** hton32**((**uns32**)** seqno**);**

// 填入Router lsa 部分

rtrhdr **=** **(**RTRhdr **\*)** **(**hdr**+**1**);**

rtrhdr**->**rtype **=** 0**;**

rtrhdr**->**zero **=** 0**;**

rtrhdr**->**nlinks **=** 0**;**

rlp **=** **(**RtrLink **\*)** **(**rtrhdr**+**1**);**

iiter**.**reset**();**

//遍历端口，加入与端口邻接的路由信息

**while** **((**ip **=** iiter**.**get\_next**()))**

**{**

**if** **(**ip**->**state**()** **==** IFS\_DOWN**)**

**continue;**

**else** **if** **(**ip**->**state**()** **==** IFS\_LOOP**)**

**{**

rlp**->**link\_id **=** hton32**(**ip**->**if\_addr**);**

rlp**->**link\_data **=** hton32**(**0xffffffffL**);**

rlp**->**link\_type **=** LT\_STUB**;**

rlp**->**n\_tos **=** 0**;**

rlp**->**metric **=** hton16**(**ip**->**cost**());**

add\_to\_ifmap**(**ip**);**

rlp**++;**

rtrhdr**->**nlinks**++;**

**}**

**else**

rlp **=** ip**->**rl\_insert**(**rtrhdr**,** rlp**);**

**}**

//如果没有活动接口，则清除LSA。

**if** **(**rtrhdr**->**nlinks **==** 0**)**

**{**

lsa\_flush**(**olsap**);**

**delete** **[]** ifmap**;**

ifmap **=** 0**;**

size\_ifmap **=** 0**;**

ospf**->**free\_orig\_buffer**(**hdr**);**

**return;**

**}**

length **=** **((**byte **\*)** rlp**)** **-** **((**byte **\*)** hdr**);**

hdr**->**ls\_length **=** hton16**(**length**);**

rtrhdr**->**nlinks **=** hton16**(**rtrhdr**->**nlinks**);**

ospf**->**lsa\_reorig**(**0**,**olsap**,**hdr**,**forced**);**

ospf**->**free\_orig\_buffer**(**hdr**);**

**}**

* Network LSA

//端口生成network lsa

void Interface**::**nl\_orig**(**int forced**)**

**{**

LSA **\***olsap**;**

LShdr **\***hdr**;**

olsap **=** ospf**->**myLSA**(**0**,** LST\_NET**,** if\_addr**);**

hdr **=** nl\_raw\_orig**();**

hdr **=** nl\_raw\_orig**();**

**if** **(**hdr **==** 0**)**

lsa\_flush**(**olsap**);**

**else**

**{**

seq\_t seqno**;**

seqno **=** ospf**->**ospf\_get\_seqno**(**LST\_NET**,** olsap**,** forced**);**

**if** **(**seqno **!=** InvalidLSSeq**)**

**{**

hdr**->**ls\_seqno **=** hton32**(**seqno**);**

**(**void**)** ospf**->**lsa\_reorig**(**0**,** olsap**,** hdr**,** forced**);**

**}**

ospf**->**free\_orig\_buffer**(**hdr**);**

**}**

**}**

//创建原始的Network LSA头部

LShdr **\***Interface**::**nl\_raw\_orig**()**

**{**

LShdr **\***hdr**;**

NetLShdr **\***nethdr**;**

uns16 length**;**

rtid\_t **\***nbr\_ids**;**

NbrList iter**(this);**

Neighbor **\***np**;**

//只有路由器本身是网络的指定路由器，且有一个或更多的完全相邻路由

//器都与接口连接时，路由器才创建一个network-LSA。如果另外一个路由器

//是指定路由器，则本路由器的router-LSA，会指向那个指定路由器创建的

//network-LSA。如果不存在完全邻接的相邻路由器,则网络被当作一个STUB

//在router-LSA 中被通告。如果不打算创建一个network-LSA􀈪则必须清除

//以前创建的任何network-LSA。

**if** **(**if\_state **!=** IFS\_DR**)**

**return(**0**);**

**else** **if** **(**if\_fullcnt **==** 0**)**

**return(**0**);**

// 组建新的LSA

length **=** **sizeof(**NetLShdr**)** **+** **(**if\_fullcnt**+**1**)\*sizeof(**rtid\_t**);**

length **+=** **sizeof(**LShdr**);**

// 填入内容

hdr **=** ospf**->**orig\_buffer**(**length**);**

hdr**->**ls\_opts **|=** SPO\_EXT**;**

hdr**->**ls\_type **=** LST\_NET**;**

hdr**->**ls\_id **=** hton32**(**if\_addr**);**

hdr**->**ls\_org **=** hton32**(**ospf**->**my\_id**());**

hdr**->**ls\_length **=** hton16**(**length**);**

nethdr **=** **(**NetLShdr **\*)** **(**hdr **+** 1**);**

nethdr**->**netmask **=** hton32**(**if\_mask**);**

// 填入FULL状态的邻居ID

nbr\_ids **=** **(**rtid\_t **\*)** **(**nethdr **+** 1**);**

**\***nbr\_ids **=** hton32**(**ospf**->**my\_id**());**

**while** **((**np **=** iter**.**getnext**())** **!=** 0**)**

**{**

**if** **(**np**->**adv\_as\_full**())**

**\*(++**nbr\_ids**)** **=** hton32**(**np**->**id**());**

**}**

**return(**hdr**);**

**}**

1. 完成创建

LSA头部和LSA内容都已经创建好，调用lsa\_reorig组装LSA，完成创建。

//重新生成LSA

LSA **\***OSPF**::**lsa\_reorig**(**Interface **\***ip**,**LSA **\***olsap**,**LShdr **\***hdr**,**int forced**)**

**{**

int changes**;**

LSA **\***lsap**;** //年龄为0

hdr**->**ls\_age **=** 0**;**

changes **=** **(**olsap **?** olsap**->**cmp\_contents**(**hdr**)** **:** **true);**

//如果不是强制长生LSA，而且数据库中有这个LSA的副本，则不产生LSA

**if** **(!**changes **&&** **!**forced **&&** olsap**->**do\_not\_age**()** **==** hdr**->**do\_not\_age**())**

**{**

olsap**->**we\_orig **=** **true;**

**return(**0**);**

**}**

hdr**->**generate\_cksum**();**

//加入到数据库，并泛洪发送

lsap **=** AddLSA**(**ip**,** olsap**,** hdr**,** changes**);**

lsap**->**we\_orig **=** **true;**

lsap**->**flood**(**0**,** hdr**);**

**return(**lsap**);**

**}**

# OSPF过程实现

## 发现邻居路由器与Hello处理过程

OSPF中，邻居路由器被发现和随后的操作都是有发送和接受OSPF Hello分组来保证。

### 发现邻居路由与接收Hello分组

//省略了验证参数和储存邻居数据的过程

first\_hello **=** **(**np**->**n\_state **==** NBS\_DOWN**);**//从邻居收到的第一个Hello？

np**->**nbr\_fsm**(**NBE\_HELLORCVD**);** // 驱动状态机

**for** **(**idp **=** **(**rtid\_t **\*)** **(**hlopkt**+**1**);** **;** idp**++)**

**{**

**if** **((**byte **\*)** idp **>=** pdesc**->**end**)**//对方发过来的Hello还没有active Neighbor，自己不在其中

**{**

np**->**nbr\_fsm**(**NBE\_1WAYRCVD**);**//通过事件1WAY破坏邻接关系

**if** **(**first\_hello**)**//收到的第一个Hello，则驱动状态机

**{**

np**->**nbr\_fsm**(**NBE\_EVAL**);**

**}**

**return;**

**}**

**else** **if** **(**ntoh32**(\***idp**)** **==** ospf**->**my\_id**())**//自己在Hello体内，则2WAY

**break;**

**}**

np**->**nbr\_fsm**(**NBE\_2WAYRCVD**);**

np**->**negotiate**(**hlopkt**->**Hello\_opts**);**

// 检查是否DR变化

nbr\_change **=** **false;**

backup\_seen **=** **false;**

**if** **(**old\_pri **!=** np**->**prior**)**

**{**

np**->**nbr\_fsm**(**NBE\_EVAL**);**

nbr\_change **=** **true;**

**}**

//邻居的参数发生了变化？

**else** **if** **(**old\_id **!=** np**->**n\_id**)**

nbr\_change **=** **true;**

**else** **if** **(**was\_declaring\_dr **!=** np**->**declared\_dr**())**

nbr\_change **=** **true;**

**else** **if** **(**was\_declaring\_bdr **!=** np**->**declared\_bdr**())**

nbr\_change **=** **true;**

**if** **(**if\_state **==** IFS\_WAIT**)**

**{**

**if** **(**np**->**declared\_dr**()** **&&** ntoh32**(**hlopkt**->**Hello\_bdr**)** **==** 0**)**//缺少BDR

backup\_seen **=** **true;**

**if** **(**np**->**declared\_bdr**())**//存在BDR

backup\_seen **=** **true;**

**}**

**if** **(**backup\_seen**)**//驱动接口状态机，进行选举

ifc\_fsm**(**IFE\_BSEEN**);**

**else** **if** **(**nbr\_change**)**

ifc\_fsm**(**IFE\_NCHG**);**

// 生成新的network lsa

**if** **(**old\_id **!=** np**->**n\_id **&&** np**->**ifc**()->**state**()** **==** IFS\_DR**)**

np**->**ifc**()->**nl\_orig**(false);**

**}**

### 发送Hello分组

这里主要给出端口广播发送Hello的过程。目的地址为224.0.0.5。

//广播发送Hello

void Interface**::**send\_hello**(**bool empty**)**

**{**

NbrList lis**(this);**

Neighbor **\***nbr**;**

uns16 size**;**

Pkt pkt**;**

HelloPkt **\***hellopkt**;**

rtid\_t **\***hellonbr**;**

**if(**passive**)**//接口如果是被动，则不发送

**return;**

size**=** **sizeof(**HelloPkt**);**

//对于已经发现的邻居，都要加入到Hello中

**while((**nbr**=**lis**.**getnext**()))**

**{**

**if(**nbr**->**state**()>=**NBS\_INIT**)**

size**+=sizeof(**rtid\_t**);**

**}**

**if(**build\_hello**(&**pkt**,**size**)==**0**)**

**return;**

lis**.**reset**();**

hellopkt**=(**HelloPkt **\*)(**pkt**.**spfpkt**);**

hellonbr**=(**rtid\_t **\*)(**hellopkt**+**1**);**

//向Hello加入发现的邻居的RouterID

**while(!**empty**&&(**nbr**=**lis**.**getnext**()))**

**{**

**if(**nbr**->**state**()>=**NBS\_INIT**)**

**{**

**\***hellonbr**++=**hton32**(**nbr**->**id**());**

pkt**.**dptr**+=sizeof(**rtid\_t**);**

**}**

**}**

//广播发送

if\_send**(&**pkt**,**AllSPFRouters**);**

**}**

### DR选举过程

void Interface**::**ifa\_elect**()**

**{**

int pass**;**

InAddr prev\_dr**;**

InAddr prev\_bdr**;**

if\_waitt**.**stop**();**

prev\_dr**=**if\_dr**;**

prev\_bdr**=**if\_bdr**;**

**for(**pass**=**1**;** pass**<=**2**;** pass**++)**

**{**//两次，直到结果不变

int declared**;**

byte c\_pri**;**

rtid\_t c\_id**;**

Neighbor **\***np**;**

NbrList iter**(**this**);**

Neighbor **\***bdr\_p**;**

bdr\_p**=**0**;**

if\_bdr **=** if\_addr**;**

c\_pri **=** if\_drpri**;**

c\_id **=** ospf**->**my\_id**();**

c\_id **=** ospf**->**my\_id**();**

**}**

**else**

//否则设置BDR为0

**{**

if\_bdr **=** UnknownAddr**;**

c\_pri **=** 0**;**

declared **=** false**;**

c\_id **=** 0**;**

**}**

**while** **((**np **=** iter**.**getnext**()))**

**{**

//遍历邻居

**if** **(**np**->**state**()** **<** NBS\_2WAY**)**

**continue;**

**else** **if** **(**np**->**prior **==** 0**)**

**continue;**

**else** **if** **(**np**->**declared\_dr**())**

**continue;**

**else** **if** **(**declared**)**

//本机宣称自己是BDR，而邻居没宣称自己是BDR

//或者自己的优先级高于该邻居

//或者邻居ID小于本机ID

//则跳过该邻居

**{**

**if** **(!**np**->**declared\_bdr**())**

**continue;**

**if** **(**np**->**prior **<** c\_pri**)**

**continue;**

**if** **(**np**->**prior **==** c\_pri **&&** np**->**n\_id **<** c\_id**)**

**continue;**

**}**

**else** **if** **(!**np**->**declared\_bdr**())**

//本机没有宣称自己是BDR（自己是DR或本来就不是BDR）

//或者自己的优先级高于该邻居

//或者邻居ID小于本机ID

//则跳过该邻居

**{**

**if** **(**np**->**prior **<** c\_pri**)**

**continue;**

**if** **(**np**->**prior **==** c\_pri **&&** np**->**n\_id **<** c\_id**)**

**continue;**

**}**

// 当前的邻居被选为BDR

bdr\_p **=** np**;**

if\_bdr **=** np**->**n\_addr**;**

declared **=** np**->**declared\_bdr**();**//已经有BDR

c\_pri **=** np**->**prior**;**

c\_id **=** np**->**n\_id**;**

**}**

// 初始化DR选举

iter**.**reset**();**

if\_dr\_n **=** 0**;**

**if** **(**if\_dr **==** if\_addr **&&** if\_drpri **!=** 0**)**

**{**//如果自己是DR

c\_pri **=** if\_drpri**;**

c\_id **=** ospf**->**my\_id**();**

**}**

**else**

**{**

if\_dr **=** UnknownAddr**;**

c\_pri **=** 0**;**

**}**

// 遍历2WAY的邻居

**while** **((**np **=** iter**.**getnext**()))**

**{**

**if** **(**np**->**state**()** **<** NBS\_2WAY**)**

**continue;**

**else** **if** **(**np**->**prior **==** 0**)**

**continue;**

**else** **if** **(!**np**->**declared\_dr**())**

**continue;**

**else** **if** **(**np**->**prior **<** c\_pri**)**

**continue;**

**else** **if** **(**np**->**prior **==** c\_pri **&&** np**->**n\_id **<** c\_id**)**

**continue;**

// 找到一个适合当DR的邻居

if\_dr\_n **=** np**;**

if\_dr **=** np**->**n\_addr**;**

c\_pri **=** np**->**prior**;**

c\_id **=** np**->**n\_id**;**

**}**

//如果没有选出DR，则设置BDR为DR

**if** **(**if\_dr **==** UnknownAddr**)**

**{**

if\_dr **=** if\_bdr**;**

if\_dr\_n **=** bdr\_p**;**

**}**

//如果DR或BDR的变化是有关本机的，则再进行一次

**if** **(**if\_dr **!=**prev\_dr **&&** **(**if\_dr**==**if\_addr **||** prev\_dr**==**if\_addr**))**

**continue;**

**else** **if** **(**if\_bdr **==** prev\_bdr**)**

**break;**

**else** **if** **(**if\_bdr **==** if\_addr **||** prev\_bdr **==** if\_addr**)**

**continue;**

**else**

**break;**

**}**

//对于选举结果设置自己的状态

**if** **(**if\_dr **==** if\_addr**)**

if\_state **=** IFS\_DR**;**

**else** **if** **(**if\_bdr **==** if\_addr**)**

if\_state **=** IFS\_BACKUP**;**

**else**

if\_state **=** IFS\_OTHER**;**

ifa\_allNeighbors\_event**(**NBE\_EVAL**);**

**if** **(**if\_dr **!=** prev\_dr**)**//DR与之前的不同，重新生成Router lsa

ospf**->**rl\_orig**(**false**);**

**}**

## 数据库交换与Database Description处理过程

### 数据库交换

数据库交换是路由器与一个相邻路由器间的链路状态数据库初始同步化过程。数据库交换开始的条件是双向通信已经建立，路由器决定与相邻路由器建立完全邻接关系。并且路由器还没有达到其同时进行数据库交换的数目限制。对于已经达到数目限制的邻居，加入等待同步队列。

void Neighbor**::**recv\_dd**(**struct Pkt **\***pdesc**)**

**{**

### 接收DD分组

DDPkt **\***ddpkt**;**

byte new\_opts**;**

int init**;**

int more**;**

int claim**;**

int master**;**

uns32 seqno**;**

LShdr **\***hdr**;**

int is\_empty**;**

ddpkt **=** **(**DDPkt **\*)** pdesc**->**spfpkt**;**

new\_opts **=** ddpkt**->**dd\_opts**;**

init **=** **((**ddpkt**->**dd\_imms **&** DD\_INIT**)** **!=** 0**);**

more **=** **((**ddpkt**->**dd\_imms **&** DD\_MORE**)** **!=** 0**);**

claim **=** **((**ddpkt**->**dd\_imms **&** DD\_MASTER**)** **==** 0**);**//对方声称的主从关系

master **=** **(**ospf**->**my\_id**()** **>** n\_id**);**//id大的为主

seqno **=** ntoh32**(**ddpkt**->**dd\_seqno**);**

hdr**=(**LShdr **\*)(**ddpkt**+**1**);**

is\_empty**=(**pdesc**->**end**==(**byte**\*)**hdr**);**

**if((**ntoh16**(**ddpkt**->**dd\_mtu**)>**n\_ifc**->**mtu**))**

**{**

printf**(**"BADMTU\n"**);**

**}**

**switch** **(**n\_state**)**

**{**

**case** NBS\_DOWN**:**

**case** NBS\_ATTEMPT**:**

**default:**

**return;**

**case** NBS\_INIT**:**

**case** NBS\_2WAY**:**

nbr\_fsm**(**NBE\_DDRCVD**);**

//开始数据库交换

**if(**n\_state**!=**NBS\_EXSTART**)**

**return;**

**case** NBS\_EXSTART**:**

**if(**claim**!=**master**)**//对方声称的主从关系与本机不同

**return;**

//主从关系协商一致

**if** **(**master **&&** **(!**init**)** **&&** **(**seqno **==** n\_ddseq**))**

//本机是主，收到的DD分组init为0，序列号相同，表示协商完毕

**{**

n\_opts **=** new\_opts**;**

nbr\_fsm**(**NBE\_NEGDONE**);**

**break;**

**}**

**if((!**master**)&&**init**&&**more**&&**is\_empty**)**

//本机是从，收到的DD分组init和more为1，表示写上完毕

**{**

n\_opts**=**new\_opts**;**

nbr\_fsm**(**NBE\_NEGDONE**);**

**break;**

**}**

//跳入数据库交换状态

**return** **;**

**case** NBS\_EXCHANGE**:**

**if** **(**claim **!=** master **||** init **||** n\_opts **!=** new\_opts**)**//DD错误

**{**

nbr\_fsm**(**NBE\_SEQNOMISM**);**

**return;**

**}**

**else** **if** **(**master**)**//本机是主

**{**

**if** **(**seqno **==** n\_ddseq**)**//收到的序列号与发出的一致

**break;**

**if** **(**master **&&** seqno **==** **(**n\_ddseq **-** 1**))**//还未发来最新的回复

**return;**

**}**

**else**//本机是从

**{**

**if** **(**seqno **==** **(**n\_ddseq **+** 1**))**//主增加了序列号

**break;**

**if** **(**seqno **==** n\_ddseq**)**//说明主没有收到DD分组，重传DD

**{**

retrans\_dd**();**

**return;**

**}**

**}**

nbr\_fsm**(**NBE\_SEQNOMISM**);**

**return;**

**case** NBS\_LOADING**:**

**case** NBS\_FULL**:**//本机认为已经完全邻接

**if** **(**claim **!=** master **||** init **||** n\_opts **!=** new\_opts**)**//DD错误

**{**

nbr\_fsm**(**NBE\_SEQNOMISM**);**

**return;**

**}**

**else** **if** **(**master **&&** seqno **==** **(**n\_ddseq **-** 1**))**

//本机是主，对方发送正确的DD在,直接返回

**return;**

**else** **if** **((!**master**)** **&&** seqno **==** n\_ddseq**)**

//本机是从，收到的序列号没有改变，说明对方没有收到DD，重传

**{**

retrans\_dd**();**

**return;**

**}**

nbr\_fsm**(**NBE\_SEQNOMISM**);**

**return;**

**}**

//收到了有效的DD分组，

//释放准备发的DD

//处理收到的DD里面带有的LSA信息

//组成LS Req，并发送

negotiate**(**n\_opts**);**

n\_imms**=**ddpkt**->**dd\_imms**;**

dd\_free**();**

ddt**.**stop**();**

progt**.**restart**();**

process\_dd**(**hdr**,**pdesc**->**end**);**

send\_req**();**

//如果是主，增加序列号

**if(**master**)**

**{**

n\_ddseq**++;**

**if(**database\_sent**&&(!**more**))**

**{**

nbr\_fsm**(**NBE\_EXCHANGEDONE**);**

**return;**

**}**

**}**

//如果是从，拷贝序列号

**else**

n\_ddseq**=**seqno**;**

//没有请求，发送下一个DD

**if(**reqlist**.**is\_empty**())**

send\_dd**();**

**}**

### 发送DD分组

void Neighbor**::**send\_dd**()**

**{**

int master**;**

Interface **\***ip**;**

DDPkt **\***ddpkt**;**

LShdr **\***hdr**;**

LSA **\***lsap**;**

LsaListIterator iter**(&**ddlist**);**

master **=** **(**ospf**->**my\_id**()** **>** n\_id**);**

ip **=** n\_ifc**;**

//由ospf分配最基本的分组

**if** **(**ospf**->**getpkt**(&**n\_ddpkt**,** SPT\_DD**,** ip**->**mtu**)** **==** 0**)**

**return;**

//填入字段

ddpkt **=** **(**DDPkt **\*)** **(**n\_ddpkt**.**spfpkt**);**

ddpkt**->**dd\_mtu **=** hton16**(**ip**->**mtu**);**

ddpkt**->**dd\_seqno **=** hton32**(**n\_ddseq**);**

ddpkt**->**dd\_opts **|=** SPO\_EXT**;**

n\_ddpkt**.**dptr **=** **(**byte **\*)** **(**ddpkt **+** 1**);**

**if** **(**n\_state **==** NBS\_EXSTART**)**

//如果是数据库交换开始阶段

//imms设置为DD\_INIT | DD\_MORE | DD\_MASTER

//直接发送分组，并设定重传时间

**{**

n\_ddpkt**.**hold **=** **true;**

ddpkt**->**dd\_imms **=** DD\_INIT **|** DD\_MORE **|** DD\_MASTER**;**

database\_sent **=** **false;**

ip**->**Neighbor\_send**(&**n\_ddpkt**,** **this);**

ddt**.**start**(**ip**->**if\_rxmt**\***Timer**::**SECOND**,** **false);**

**return;**

**}**

**if** **(**master**)**

ddt**.**start**(**ip**->**if\_rxmt**\***Timer**::**SECOND**,** **false);**

progt**.**restart**();**

n\_ddpkt**.**hold **=** **true;**

ddpkt**->**dd\_imms **=** 0**;**

//加入LSA信息

**while** **((**lsap **=** iter**.**get\_next**()))**

**{**

hdr **=** **(**LShdr **\*)** n\_ddpkt**.**dptr**;**

**if** **(!**lsap**->**valid**())**

**{**

iter**.**remove\_current**();**

**continue;**

**}**

**else** **if** **(**lsap**->**lsa\_age**()** **==** MaxAge**)**

//LSA已经达到最大老化

**{**

iter**.**remove\_current**();**

add\_to\_retranslist**(**lsap**);**

hdr **=** ospf**->**BuildLSA**(**lsap**);**

**(**void**)** add\_to\_update**(**hdr**);**

**continue;**

**}**

**else** **if** **(**n\_ddpkt**.**dptr **+** **sizeof(**LShdr**)** **>** n\_ddpkt**.**end**)**

//已经达到最大DD长度

**break;**

**else**

**{**

iter**.**remove\_current**();**

**\***hdr **=** **\***lsap**;**

n\_ddpkt**.**dptr **+=** **sizeof(**LShdr**);**

**}**

**}**

//设置响应的参数

**if** **(**master**)**

ddpkt**->**dd\_imms **|=** DD\_MASTER**;**

**if** **(!**ddlist**.**is\_empty**())**

//仍有LSA没有传完

ddpkt**->**dd\_imms **|=** DD\_MORE**;**

**else**

database\_sent **=** **true;**

ip**->**Neighbor\_send**(&**n\_ddpkt**,** **this);**

ip**->**Neighbor\_send**(&**n\_update**,** **this);**

**if** **((!**master**)** **&&** database\_sent **&&** **(**n\_imms **&** DD\_MORE**)** **==** 0**)**

**{**

nbr\_fsm**(**NBE\_EXCHANGEDONE**);**

holdt**.**start**(**ip**->**if\_deadinterval**\***Timer**::**SECOND**);**

**}**

**}**

## Link State Request处理过程

### 发送LSR分组

void Neighbor**::**send\_req**()**

**{**

Interface **\***ip**;**

Pkt pkt**;**

ReqPkt **\***rqpkt**;**

LSRef **\***lsref**;**

LsaListIterator iter**(&**reqlist**);**

int count**;**

//要请求的LSA为空，则返回

**if** **(**reqlist**.**is\_empty**())**

**return;**

ip **=** n\_ifc**;**

pkt**.**iphead **=** 0**;**

//得到原始的LSR分组

**if** **(**ospf**->**getpkt**(&**pkt**,** SPT\_LSREQ**,** ip**->**mtu**)** **==** 0**)**

**return;**

rqpkt **=** **(**ReqPkt **\*)** **(**pkt**.**spfpkt**);**

pkt**.**dptr **=** **(**byte **\*)** **(**rqpkt **+** 1**);**

count **=** 0**;**

//加入需要知道的LSA

**for** **(**lsref **=** **(**LSRef **\*)** pkt**.**dptr**;** **;** **)**

**{**

LSA **\***lsap**;**

**if** **(!(**lsap **=** iter**.**get\_next**()))**

**break;**

**if** **(**pkt**.**dptr **+** **sizeof(**LSRef**)** **>** pkt**.**end**)**

//达到最大长度

**break;**

lsref**->**ls\_type **=** hton32**(**lsap**->**ls\_type**());**

lsref**->**ls\_id **=** hton32**(**lsap**->**ls\_id**());**

lsref**->**ls\_org **=** hton32**(**lsap**->**adv\_rtr**());**

pkt**.**dptr **+=** **sizeof(**LSRef**);**

lsref**++;**

count**++;**

**}**

req\_goal **=** reqlist**.**count**()** **-** count**;**

//发送

ip**->**Neighbor\_send**(&**pkt**,** **this);**

**}**

### 接收LSR分组

//收到LS Requuest

void Neighbor**::**recv\_req**(**Pkt **\***pdesc**)**

**{**

ReqPkt **\***rqpkt**;**

LSRef **\***lsref**;**

Interface **\***ip**;**

//还未进行数据库交换

**if** **(**n\_state **<** NBS\_EXCHANGE**)**

**return;**

ip **=** n\_ifc**;**

rqpkt **=** **(**ReqPkt **\*)** pdesc**->**spfpkt**;**

lsref **=** **(**LSRef **\*)** **(**rqpkt **+** 1**);**

//获取对方请求的LSA

**for** **(;** **(**byte **\*)** **(**lsref **+** 1**)** **<=** pdesc**->**end**;** lsref**++)**

**{**

lsid\_t ls\_id**;**

rtid\_t ls\_org**;**

LSA **\***lsap**;**

LShdr **\***hdr**;**

byte lstype**;**

lstype **=** ntoh32**(**lsref**->**ls\_type**);**

ls\_id **=** ntoh32**(**lsref**->**ls\_id**);**

ls\_org **=** ntoh32**(**lsref**->**ls\_org**);**

//查找请求的LSA

**if** **(!(**lsap **=** ospf**->**FindLSA**(**ip**,** lstype**,** ls\_id**,** ls\_org**)))**

**{**

nbr\_fsm**(**NBE\_BADLSREQ**);**

**return;**

**}**

hdr **=** ospf**->**BuildLSA**(**lsap**);**

//加入到update分组中

**(**void**)** add\_to\_update**(**hdr**);**

**}**

//发送update

ip**->**Neighbor\_send**(&**n\_update**,** **this);**

**}**

## 泛洪与Link State Update处理过程

### 单个路由泛洪过程

泛洪过程从路由器自邻接的相邻路由器接收到一个链路状态更新分组开始。路由器检查并确认更新分组中的每一个LSA。路由器辨别出更新的LSA，并将其装入链路状态数据库，添加到所有其他邻接的相邻路由器的链路状态重发表中。之后将其封装到链路状态更新分组中以从所有接口发送出去。

路由器自创建的LSA 也将被泛洪到邻接的相邻路由器。它所使用的泛洪逻辑与在链路状态更新分组中接收LSA 的泛洪逻辑是一致的。泛洪过程从装载LSA 到数据库中开始。

路由器定期将邻接的相邻路由器的链路状态重发列表上的LSA 重发到它的相邻路由器。只有当相邻路由器通过链路状态确认分组对接收的LSA 进行确认时􀈹路由器才会将这些LSA 从链路状态重发列表中删除。



### 接收LSU分组

//接收Update处理

void Neighbor**::**recv\_update**(**Pkt **\***pdesc**)**

**{**

Interface **\***ip**;**

int count**;**

UpdPkt **\***upkt**;**

LShdr **\***hdr**;**

byte **\***end\_lsa**;**

**if** **(**n\_state **<** NBS\_EXCHANGE**)**

**{**

**if** **(**n\_ifc**->**type**()** **==** IFT\_PP**)**

n\_ifc**->**send\_hello**(true);**

**return;**

**}**

ip **=** n\_ifc**;**

upkt **=** **(**UpdPkt **\*)** pdesc**->**spfpkt**;**

ip**->**recv\_update **=** **true;**

//LSA个数

count **=** ntoh32**(**upkt**->**upd\_no**);**

//第一个LSA头部

hdr **=** **(**LShdr **\*)** **(**upkt**+**1**);**

**for** **(;** count **>** 0**;** count**--,** hdr **=** **(**LShdr **\*)** end\_lsa**)**

**{**

int errval**=**0**;**

int lslen**;**

int lstype**;**

lsid\_t lsid**;**

rtid\_t orig**;**

age\_t lsage**;**

LSA **\***olsap**;**

int compare**;**

int rq\_cmp**;**

lstype **=** hdr**->**ls\_type**;**

lsage **=** ntoh16**(**hdr**->**ls\_age**);**

**if** **((**lsage **&** **~**DoNotAge**)** **>=** MaxAge**)**

lsage **=** MaxAge**;**

lslen **=** ntoh16**(**hdr**->**ls\_length**);**

end\_lsa **=** **((**byte **\*)**hdr**)** **+** lslen**;**

**if** **(**end\_lsa **>** pdesc**->**end**)**

**break;**

//校验

**if** **(!**hdr**->**verify\_cksum**())**

errval **=** 1**;**

//获取某种类型LSA的数据库

**else** **if** **(!**ospf**->**FindLSdb**(**ip**,** lstype**))**

errval **=** 2**;**

**if** **(**errval **!=** 0**)**

**{**

**continue;**

**}**

//找到当前的数据库副本

lsid **=** ntoh32**(**hdr**->**ls\_id**);**

orig **=** ntoh32**(**hdr**->**ls\_org**);**

olsap **=** ospf**->**FindLSA**(**ip**,**lstype**,** lsid**,** orig**);**

//如果数据库为空或已经到达最大时间，就发送ACK并释放LSA

**if** **(**lsage **==** MaxAge **&&** **(!**olsap**)** **&&** ospf**->**maxage\_free**(**lstype**))**

**{**

build\_imack**(**hdr**);**

**continue;**

**}**

/\*与LS数据库中的LSA比较，1表示接收到的更新，0表示一样，-1 表示接收到的比较旧\*/

compare **=** **(**olsap **?** olsap**->**cmp\_instance**(**hdr**)** **:** 1**);**

/\*收到的LSA更新\*/

**if** **(**compare **>** 0**)**

**{**

bool changes**;**

LSA **\***lsap**;**

**if** **(**olsap **&&** olsap**->**since\_received**()** **<** MinArrival**)**

**{**

**if** **(**olsap**->**min\_failed**)**

**{**

**continue;**

**}**

**}**

//如果是自己长生的LSA，强制重新创建

changes **=** **(**olsap **?** olsap**->**cmp\_contents**(**hdr**)** **:** **true);**

**if** **(**changes **&&** ospf**->**originated**(this,** hdr**,** olsap**))**

**continue;**

//否则加入数据库

lsap **=** ospf**->**AddLSA**(**ip**,**olsap**,** hdr**,** changes**);**

//泛洪发送

lsap**->**flood**(this,** hdr**);**

**}**

**else** **if** **(**ospf\_rmreq**(**hdr**,** **&**rq\_cmp**))**

//数据库交换出错

**{**

nbr\_fsm**(**NBE\_BADLSREQ**);**

**}**

**else** **if** **(**compare **==** 0**)**

**{**

**if** **(!**remove\_from\_retranslist**(**olsap**))**

build\_imack**(**hdr**);**

**}**

**else**

**{**

LShdr **\***ohdr**;**

//更新数据库副本

**if** **(**olsap**->**ls\_seqno**()** **==** MaxLSSeq**)**

**continue;**

**if** **(**olsap**->**sent\_reply**)**

**continue;**

ohdr **=** ospf**->**BuildLSA**(**olsap**);**

add\_to\_update**(**ohdr**);**

olsap**->**sent\_reply **=** **true;**

ospf**->**replied\_list**.**addEntry**(**olsap**);**

**}**

**}**

//泛洪回到接收端口

ip**->**if\_send\_update**();**

//从其他接口泛洪出去

ospf**->**send\_updates**();**

ip**->**Neighbor\_send**(&**n\_imack**,** **this);**

ip**->**Neighbor\_send**(&**n\_update**,** **this);**

ip**->**recv\_update **=** **false;**

//如果有必要，继续发送请求

**if** **(**reqlist**.**count**()** **&&** reqlist**.**count**()** **<=** req\_goal**)**

**{**

lsrt**.**restart**();**

send\_req**();**

**}**

**}**

### 泛洪LSA

//泛洪，需要用重传

void LSA**::**flood**(**Neighbor **\***from**,** LShdr **\***hdr**)**

**{**

IfcList ifcIter**(**ospf**);**

Interface **\***r\_ip**;**

Interface **\***ip**;**

bool flood\_it**=false;**

bool on\_demand**=false;**

bool on\_regular**=false;**

//得到要发送的接口

r\_ip **=** **(**from **?** from**->**ifc**()** **:** 0**);**

**if** **(!**hdr**)**

hdr **=** ospf**->**BuildLSA**(this);**

//遍历系统所有接口的邻居，记录需要发送的邻居

**while** **((**ip **=** ifcIter**.**get\_next**()))**

**{**

Neighbor **\***np**;**

NbrList nbrIter**(**ip**);**

int n\_nbrs**;**

n\_nbrs **=** 0**;**

**while** **((**np **=** nbrIter**.**getnext**()))**

**{**

int rq\_cmp**;**

**if** **(**np**->**state**()** **<** NBS\_EXCHANGE**)**

**continue;**

**if** **(**np**->**ospf\_rmreq**(**hdr**,** **&**rq\_cmp**)** **&&** rq\_cmp **<=** 0**)**

**continue;**

**if** **(**np **==** from**)**

**continue;**

n\_nbrs**++;**

np**->**add\_to\_retranslist**(this);**

**}**

//向update加入需要发送的LSA

**if** **(**ip **==** r\_ip **&&** **(**ip**->**state**()** **==** IFS\_DR **&&** **!**from**->**is\_bdr**()** **&&** n\_nbrs **!=** 0**))**

**{**

ip**->**add\_to\_update**(**hdr**);**

**}**

**else** **if** **((**r\_ip **==** 0 **&&** n\_nbrs **!=** 0**)&&**ip**->**recv\_update**)**

ip**->**add\_to\_update**(**hdr**);**

**else** **if** **(**ip **!=** r\_ip**)**

**{**

**if** **(**n\_nbrs **==** 0**)**

**continue;**

flood\_it **=** **true;**

ip**->**flood**=true;**

on\_regular **=** **true;**

**}**

**else**

ip**->**if\_build\_dack**(**hdr**);**

**}**

**if** **(!**flood\_it**)**

**return;**

**else**

**{**

**if** **(**on\_regular**)**

ospf**->**add\_to\_update**(**hdr**,** **false);**

**if** **(**on\_demand**)**

ospf**->**add\_to\_update**(**hdr**,** **true);**

**}**

**}**

### 重发LSA

本程序重发机制靠如下三个分离的链表完成。

class LsaList pend\_retrans;//最近需要重传的LSA

class LsaList retrans;//已经泛洪的LSA，但还没有到重传时间

class LsaList failed\_retrans;//发送了LSA还没有收到确认

每当收到确认，则从三个链表中移除收到确认的LSA。如果到了重传之间，则嗲用Neighbor::retrans\_updates()重传LS Update。

//重发LS Update

void Neighbor**::**retrans\_updates**()**

**{**

int space**;**

LSA **\***lsap**;**

LShdr **\***hdr**=**0**;**

int npkts**;**

LsaList **\***list**;**

uns32 nexttime**;**

space **=** 0**;**

npkts **=** 0**;**

// 从LS重发表得到要发的LSA

**while** **((**lsap **=** get\_next\_retrans**(**list**,** nexttime**)))**

**{**

bool full**;**

// 是否还有剩余空间

full **=** space **<** lsap**->**ls\_length**();**

**if** **(**full **&&** **++**npkts **>** retrans\_window**)**

**break;**

hdr **=** ospf**->**BuildLSA**(**lsap**);**

space **=** add\_to\_update**(**hdr**);**

// 将LSA移动到等待重发的列表

list**->**remove**(**lsap**);**

pend\_retrans**.**addEntry**(**lsap**);**

**}**

**if** **(**npkts **>** 0**)**

**{**

// 向邻居发送update

n\_ifc**->**Neighbor\_send**(&**n\_update**,** **this);**

// 重启重发计时器

lsat**.**start**(**n\_ifc**->**rxmt\_interval**()\***Timer**::**SECOND**);**

**}**

**else** **if** **(**nexttime**)**

**{**

lsat**.**start**(**nexttime**\***Timer**::**SECOND**);**

**}**

**}**

等待到达重传时间，则进行重传。

void LSATimer**::**action**()**

**{**

LsaList **\***list**;**

uns32 nexttime**;**

// 等待重发表移到重发过等待确认的列表

np**->**failed\_retrans**.**append**(&**np**->**pend\_retrans**);**

// 如果仍有LSA需要重发，则设置窗口为1

**if** **(**np**->**get\_next\_retrans**(**list**,** nexttime**))**

**{**

np**->**retrans\_window **=** 1**;**

**}**

np**->**retrans\_updates**();**

// 删除一些已经失效的LSA

np**->**retrans\_cnt **-=** np**->**retrans**.**garbage\_collect**();**

np**->**retrans\_cnt **-=** np**->**failed\_retrans**.**garbage\_collect**();**

**}**

## Link State Acknowledgment处理过程

### 接收LS ACK分组

从邻居收到ACK，会从邻居的重发表中查找有无对应的LSA，如果有，则从重传表中删除该LSA。

void Neighbor**::**recv\_ack**(**Pkt **\***pdesc**)**

**{**//接收ACK

Interface **\***ip**;**

AckPkt **\***apkt**;**

LShdr **\***hdr**;**

**if** **(**n\_state **<** NBS\_EXCHANGE**)**

**return;**

ip **=** n\_ifc**;**

apkt **=** **(**AckPkt **\*)** pdesc**->**spfpkt**;**

hdr **=** **(**LShdr **\*)** **(**apkt**+**1**);**

**for** **(;** **((**byte **\*)(**hdr**+**1**))** **<=** pdesc**->**end**;** hdr**++)**

**{**

int lstype**;**

lsid\_t lsid**;**

rtid\_t orig**;**

LSA **\***lsap**;**

int compare**;**

lstype **=** hdr**->**ls\_type**;**

lsid **=** ntoh32**(**hdr**->**ls\_id**);**

orig **=** ntoh32**(**hdr**->**ls\_org**);**

// 将接收到的ACK与LSA重发表中的LSA比较

lsap **=** ospf**->**FindLSA**(**ip**,** lstype**,** lsid**,** orig**);**

compare **=** lsap **?** lsap**->**cmp\_instance**(**hdr**)** **:** 1**;**

// 如果受到的ACK更新，从LSA重发中的LSA删除

**if(**compare**==**0**)**

remove\_from\_retranslist**(**lsap**);**

**}**

**}**

### 组建LS ACK分组

//组建ACK

void Interface**::**if\_build\_ack**(**LShdr **\***hdr**,** Pkt **\***pkt**,** Neighbor **\***np**)**

**{**

**if** **(!**pkt**)**

pkt **=** **&**if\_dack**;**

// 如果已经没有多的空间可以添加LSA，直接发送当前的ACK

**if** **(**pkt**->**iphead **&&** **(**pkt**->**dptr **+** **sizeof(**LShdr**))** **>** pkt**->**end**)**

**{**

**if** **(**np**)**

Neighbor\_send**(**pkt**,** np**);**

**else**

if\_send**(**pkt**,** if\_flood**);**

**}**

**if** **(!**pkt**->**iphead**)**

**{**

**if** **(**ospf**->**getpkt**(**pkt**,** SPT\_LSACK**,** mtu**)** **==** 0**)**

**return;**

**}**

// 将要确认的LSA的头部加入到包中

memcpy**(**pkt**->**dptr**,** hdr**,** **sizeof(**LShdr**));**

pkt**->**dptr **+=** **sizeof(**LShdr**);**

**}**

## 路由选择计算

### 路由计算

OSPF::dijkstra()计算所有路由器的区内路由。该函数使用Dijkstra 最短路径优先算法。该算法运行一次即可在路由器附属区内找到到达每个router-LSA 和network-LSA 的最短路径。

算法描述如下：

首先存在一个由router-LSA 和network-LSA 组成的备选列表，对列表中的任何一个LSA都存在一条到达该LSA 的路径，但是可能不是最优路径。每一次循环开始，到达最近的余下的备选LSA 的路径保证是最优的。因此该备选LSA 从备选列表中移出并添加到最短路径树。而被移到最短路径树的备选LSA 的相邻LSA 被检查，这些相邻LSA 可能正被添加到备选列表中，或者其在备选列表中的位置被调整。然后循环再重新开始，直到备选列表为空。

备选的LSA 由类TNode 实现。LSA 的备选状态由TNode::t\_state 表示。该值如果为DS\_ONCAND，则LSA 当前在备选列表中。如果为DS\_ONTREE，则表示已经找到。

当一个LSA 被添加到最短路径树上时，与该LSA 相联系的路由表表项通过调用RtrTblEntry::new\_intra()更新。如果LSA 是一个network-LSA，则其网络前缀的路由表表项被更新。如果是router-LSA，则为路由器设置的表项被更新，接着router-LSA内的STUB描述被处理。反复调用RTE::new\_intra()就是为了保证其路由表表项被更新。当发现更短的路径时，则RTE::new\_intra()放弃当前路径，并且如果存在多条最短路径，则多条最短路径都被保存。

//最短路径计算

void OSPF**::**dijkstra**()**

**{**

Queue cand**;**

TNode **\***V**;**

n\_dijkstras**++;**

rtrLSA **\***rtr**;**

netLSA **\***net**;**

n\_routers **=** 0**;**

rtr **=** **(**rtrLSA **\*)** rtrLSAs**.**sllhead**;**

**for** **(;** rtr**;** rtr **=** **(**rtrLSA **\*)** rtr**->**sll**)**

rtr**->**t\_state **=** DS\_UNINIT**;**

net **=** **(**netLSA **\*)** netLSAs**.**sllhead**;**

**for** **(;** net**;** net **=** **(**netLSA **\*)** net**->**sll**)**

net**->**t\_state **=** DS\_UNINIT**;**

// 初始化候选列表，加入自己产生的router LSA

dijk\_init**(**cand**);**

**while** **((**V **=** **(**TNode **\*)** cand**.**rmhead**()))**

**{**

RtrTblEntry **\***dest**;**

Link **\***lp**;**

TNode **\***W**;**

int i**;**

// 将V放到最小路径树上

V**->**t\_state **=** DS\_ONTREE**;**

dest **=** V**->**t\_dest**;**

dest**->**new\_intra**(**V**,** **false,** 0**,** 0**);**

**if** **(**V**->**ls\_type**()** **==** LST\_RTR**)**

n\_routers**++;**

// 遍历与V联系的节点，将可能的节点加入到候选列表

**for** **(**lp **=** V**->**t\_links**,** i **=** 0**;** lp **!=** 0**;** lp **=** lp**->**l\_next**,** i**++)**

**{**

TLink **\***tlp**;**

uns32 new\_cost**;**

//更新STUB路由表项

**if** **(**lp**->**l\_ltype **==** LT\_STUB**)**

**{**

SLink **\***slp**;**

slp **=** **(**SLink **\*)**lp**;**

**if** **(!**slp**->**sl\_rte**)**

**continue;**

slp**->**sl\_rte**->**new\_intra**(**V**,true,**slp**->**l\_fwdcst**,**i**);**

**continue;**

**}**

tlp **=** **(**TLink **\*)** lp**;**

//W没有与lp邻接

**if** **(!(**W **=** tlp**->**tl\_nbr**))**

**continue;**

//W已经在SPF树上

**if** **(**W**->**t\_state **==** DS\_ONTREE**)**

**continue;**

//从V到达W的代价

new\_cost **=** V**->**cost0 **+** tlp**->**l\_fwdcst**;**

**if** **(**W**->**t\_state **==** DS\_ONCAND**)**

**{**

**if** **(**new\_cost **>** W**->**cost0**)**

**continue;**

**else** **if** **(**new\_cost **<** W**->**cost0**)**

//找到 更好的路径，从备选列表删除W

cand**.**del**(**W**);**

**}**

// 找到更好的路径，设置路径的参数

**if** **(**W**->**t\_state **!=** DS\_ONCAND **||** new\_cost **<** W**->**cost0**)**

**{**

W**->**t\_direct **=** **(**ospf**->**mylsa**==(**rtrLSA **\*)**V**);**

W**->**cost0 **=** new\_cost**;**

W**->**cost1 **=** 0**;**

W**->**tie1 **=** W**->**lsa\_type**;**

cand**.**add**(**W**);**

W**->**t\_state **=** DS\_ONCAND**;**

W**->**t\_parent **=** V**;**

W**->**t\_epath **=** 0**;**

这里使用到了几个数据结构，TNode和TLink。类TNode 是router-LSA 和network-LSA 的基类，用于表示最短路径计算中担任中继点角色的LSA。

**}**

**else** **if** **(**ospf**->**mylsa**==(**rtrLSA **\*)**V**)**

W**->**t\_direct **=** **true;**

// 现在找到了到达W的最短路径，加入下一跳

W**->**add\_next\_hop**(**V**,** i**);**

**}**

**}**

**}**

LSA 的备选状态由TNode::t\_state 表示。该值如果为DS\_ONCAND，则LSA 当前在备选列表中；如果为DS\_ONTREE，则表示已经找到到达该LSA 的最短路径；如果为DS\_UNINIT，则表示还没有发现到达该LSA 的路径。

TLink是TNode构成的链表。

本程序使用的最短路径算法是Prim最小生成树算法。每次都会遍历与已经在树上的结点相关的所有边，找到一个另一端不在树上并且花费最小的边，并且将另一端放到树上。这样进行到最后，就可以找到一个最小生成树。

### 与内核同步路由

//向内核安装路由表

void IPrte**::**sys\_install**()**

**{**

TblItem **\***item**;**

**switch(**r\_type**)**

**{**

**case** RT\_DIRECT**:**

fa\_tbl**->**resolve**();**

**break;**

**case** RT\_NONE**:**

**case** RT\_STATIC**:**

**case** RT\_EXTT1**:**

**case** RT\_EXTT2**:**

fa\_tbl**->**resolve**(this);**

**break;**

**default:**

**break;**

**}**

**if** **((**item **=** ospf**->**krtdeletes**.**find**(**net**(),** mask**())))**

**{**

ospf**->**krtdeletes**.**remove**(**item**);**

**delete** item**;**

**}**

// 更新内核的转发表

**switch(**r\_type**)**

**{**

**case** RT\_NONE**:**

sys**->**rtdel**(**net**(),** mask**(),** last\_epath**);**

**break;**

**case** RT\_REJECT**:**

sys**->**rtadd**(**net**(),** mask**(),** 0**,** last\_epath**,** **true);**

**break;**

**default:**

sys**->**rtadd**(**net**(),** mask**(),** r\_epath**,** last\_epath**,** **false);**

**break;**

**}**

last\_epath **=** r\_epath**;**

**}**

上述代码中，有两个重要的函数，分别是rtdel和rtadd。这两个函数在3.3.3 “重要接口函数”中简单介绍过，他们会利用Linux IPv4 routing socket与内核通信，控制内核添加或删除路由条目。

详细的socket选项和控制可以参见源文件system.cpp，rtdel和rtadd函数体部分。

# 测试与检验

## 测试环境

本程序分两次测试检验，第一次在虚拟机和模拟器上进行，第二次用真机和真实路由器进行。

* 虚拟测试环境：

操作系统：Windows 8 64bit

虚拟机软件：VMware® Workstation 9.0.2 build-1031769

虚拟机系统： Ubuntu 13.04 64bit

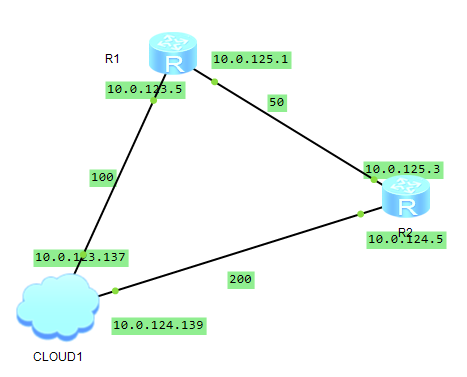
路由模拟器软件：华为eNSP 1.2.00.200

* 真实测试环境：

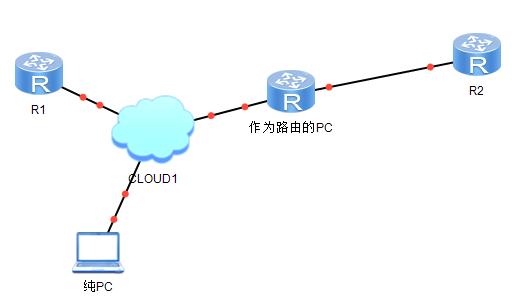
两台路由器，一台Linux台式双网卡电脑，一台Windows电脑

## 测试拓扑

* 虚拟测试拓扑（CLOUD1中有虚拟机两个网卡的接口）：

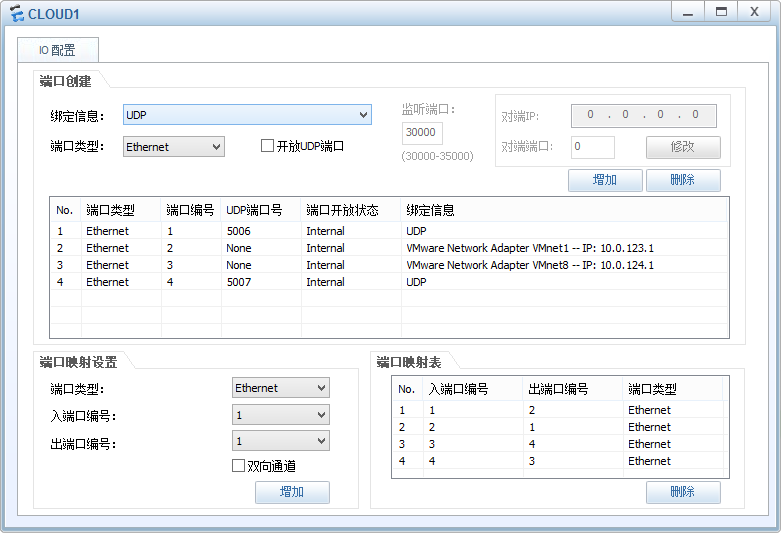


* 真实测试（CLOUD2构成一个广播网络）：



## 配置方法

* 虚拟测试
  + - 1. 打开eNPS，按拓扑图添加除了连线之外的器件，路由模拟器使用AR1220。
      2. 如图为CLOUD1添加接口和连接：



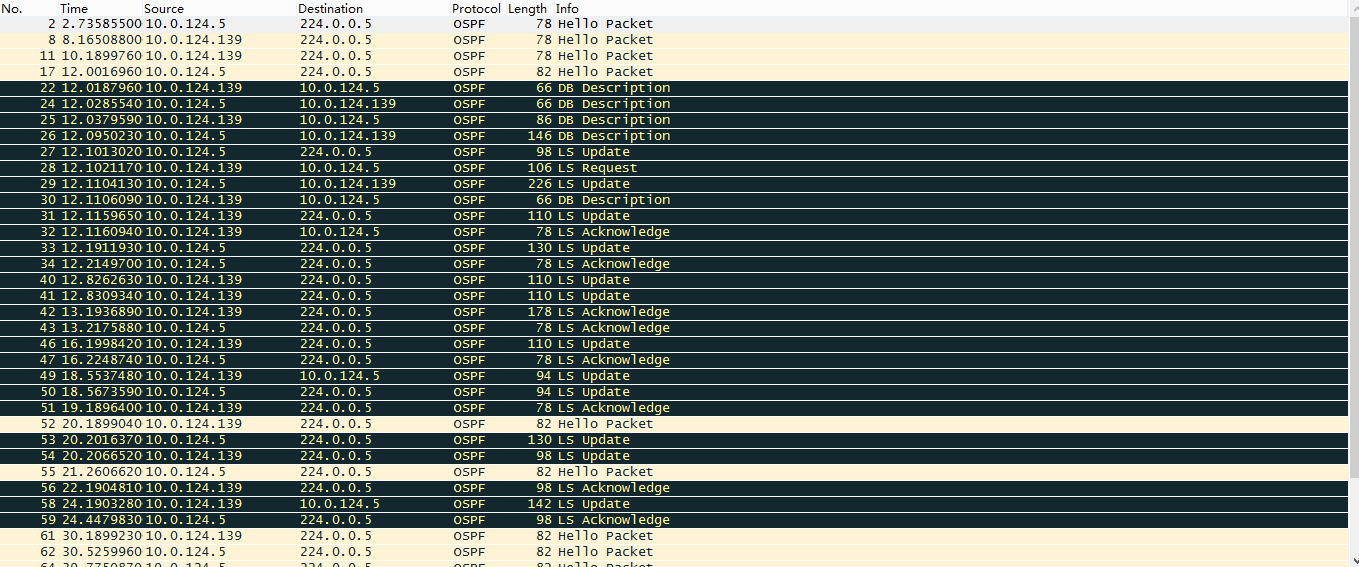
* + - 1. 连接其他器件
      2. 启动路由模拟器，按拓扑图设置路由器端口和IP以及链路开销，设置OSPF区域为0，验证模式为无验证模式。
      3. 按照安装指导，在虚拟机上安装路由软件，填写好配置文件。在Linux系统下配置好网卡IP，运行路由软件。
      4. 在虚拟机上截取报文，并查看路由表。
      5. 互相使用ping命令。
* 真实测试
  + - 1. 按照拓扑图连接网络，配置好路由、PC网卡IP。
      2. 按照安装指导，在虚拟机上安装路由软件，填写好配置文件。在Linux系统下配置好网卡IP，运行路由软件。
      3. 在各个设备上互相使用ping命令。

## 测试结果

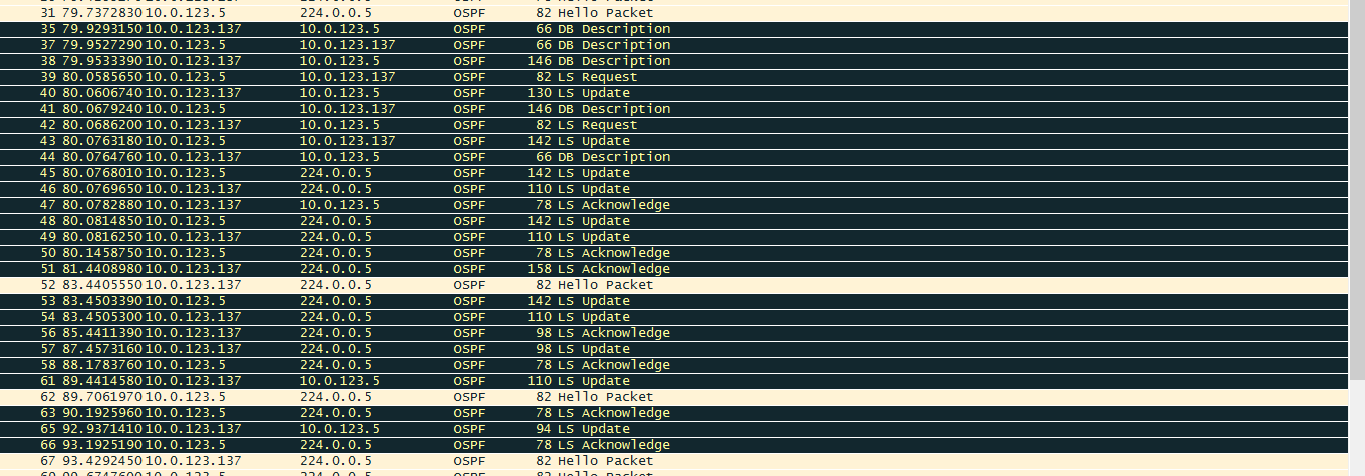
* 虚拟测试

虚拟机报文截取：

eth0：

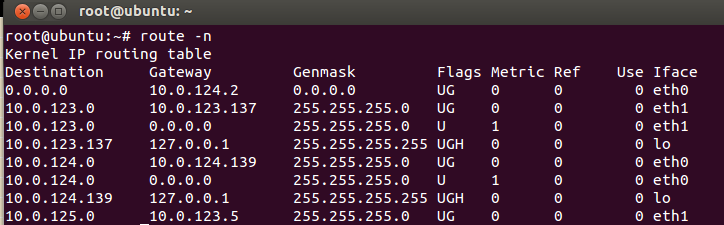


eth1：



以上报文说明正常建立了完全的邻接关系。

虚拟机路由表：



由路由表可以看到到达10.0.125.0网段的路由交付给了10.0.123.5，而不是直接交付 给与虚拟机相连的R2的端口10.0.124.5，说明正确计算了最短路径。

并且，虚拟机和两个路由模拟器之间可以互相PING通。

* 真实测试

纯PC，作为路由的PC和两台路由之间可以互相PING通。

# 结论

这次OSPF设计实验，耗费了大量的精力和时间，但也取得了较好的结果。主要实现了以下功能：

* + - 1. 骨干区域（区域号为0）中广播网络中的DR、BDR选举。
      2. OSPF分组的封装和发送。
      3. Router LSA和Network LSA的实现。
      4. 最短路径的计算。
      5. 将计算出的路由表存入Linux系统内核。

由此实现了最短路径的选择、路由表的生成，从而实现了全网的互通。

此外，在细节方面按照面向对象原则实现了一些主要的类，为以后的程序扩展提供了方便。

今后如果还有机会改进本次的OSPF的实现，可以很容易地在原有的基础上扩展区域的实现，并且加入其它LSA的实现。并且，制作一个控制台对OSPF进行实时控制也很有必要，这样就不需要每次更改设置都必须重启一遍程序。

当然程序还有不少的bug，仍然需要一一排查。

最后，在工程验收的时候受老师和助教启发，有机会需要对程序持久健壮性进行测试，看是否有累积而造成的内存泄漏情况。

这次课程设计遇到了不少的困难，虽然花费了很多的时间和精力，但还是很值得的。收获有很多，对于C++的运用也更熟悉了，对网络的理解也超出了平时在课堂上学习的深度，对于路由协议和OSPF也有了较全面的理解。

# 致谢

在这次课程设计当中，遇到了不少困难，同做OSPF实验的两名高工同学给予了我很大的帮助。

感谢殷鹏程同学提供了eNSP软件模拟路由器的信息，如果不知道这个软件，调试程序会变得十分困难。

感谢吴俣同学在最后时刻发现了我的问题，并且告知了我开启IP转发功能的方法。我因为没有打开IP转发而无法ping而连续几天卡在这上面，导致差点无法完成工程。

感谢GoAgent提供了稳定的Google连接，在Google上用英文搜索可以得到很多十分有用在百度上搜索不到或者翻译不靠谱的东西，这些不论对于完成这个程序还是对以后的研究学习都十分重要。

最后也要感谢在Stack Overflow技术论坛上那些不知真实姓名的编程者们，我在上面查找过问题，提出过问题，都找到了满意并且有效的回复。如果没有他们，程序也有可能无法按时完成。

# 参考文献

[1] RFC 2328

[2] 计算机网络. 第五版. 谢希仁

[3] 计算机网络实验教程. 北航出版社. 张力军

# 附录A

# 附录B