# 目录

[目录 1](#_Toc1791289500)

[1 OSPF协议 4](#_Toc1950831599)

[1.1 协议简介 4](#_Toc1202854838)

[1.2 术语定义 5](#_Toc1971307520)

[2 软件设计 7](#_Toc1793475369)

[2.1 功能设计 7](#_Toc1116070391)

[2.2 架构设计 8](#_Toc1763476057)

[3 软件实现 10](#_Toc956055698)

[3.1 模块分析 10](#_Toc1356877437)

[3.1.1 Main 10](#_Toc2093085875)

[3.1.2 Recv 11](#_Toc1246131731)

[3.1.3 Send 11](#_Toc846207625)

[3.1.4 IO 11](#_Toc4720806)

[3.1.5 Hello 12](#_Toc1143993178)

[3.1.6 DD 13](#_Toc1726443077)

[3.1.7 LSR 13](#_Toc121189528)

[3.1.8 LSU 14](#_Toc1316994892)

[3.1.9 LSAck 14](#_Toc160033862)

[3.1.10 LSDB 15](#_Toc1675815068)

[3.1.11 NSM 16](#_Toc1878982323)

[3.1.12 SPF 16](#_Toc827493499)

[3.2 代码分析 17](#_Toc38216099)

[3.2.1 接口初始化 17](#_Toc891759360)

[3.2.2 Router-LSA生成 18](#_Toc307760641)

[3.2.3 最短路径树 20](#_Toc27523187)

[3.2.4 邻居状态机 21](#_Toc670965451)

[3.2.5 OSPF协议包接收 22](#_Toc1256621693)

[4 实验成果 24](#_Toc986520294)

[4.1 使用方法 24](#_Toc300714908)

[4.2 实验结果 25](#_Toc459046447)

[5 不足与期望 27](#_Toc1190021041)

****

计算机网络实验

提高层次实验报告

OSPF协议实现

学号：13061119

姓名：李周洋

2016年7月

# 1 OSPF协议

## 1.1 协议简介[[1]](#footnote-0)

开放式最短路径优先（Open Shortest Path First，OSPF）是对链路状态路由协议的一种实现，隶属[内部网关协议](https://zh.wikipedia.org/wiki/%E5%86%85%E9%83%A8%E7%BD%91%E5%85%B3%E5%8D%8F%E8%AE%AE" \o "内部网关协议)（IGP），故运作于[自治系统](https://zh.wikipedia.org/wiki/%E8%87%AA%E6%B2%BB%E7%B3%BB%E7%BB%9F" \o "自治系统)内部。OSPF协议是大中型网络上使用最为广泛的IGP（Interior Gateway Protocol）协议。采用迪杰斯特拉算法（Dijkstra's algorithm）来计算[最短路径树](https://zh.wikipedia.org/w/index.php?title=%E6%9C%80%E7%9F%AD%E8%B7%AF%E5%BE%84%E6%A0%91&action=edit&redlink=1" \o "最短路径树（页面不存在）)。它使用“代价（Cost）”作为路由度量。链路状态数据库（LSDB）用来保存当前[网络拓扑](https://zh.wikipedia.org/wiki/%E7%BD%91%E7%BB%9C%E6%8B%93%E6%89%91" \o "网络拓扑)结构，[路由器](https://zh.wikipedia.org/wiki/%E8%B7%AF%E7%94%B1%E5%99%A8" \o "路由器)上属于同一区域的链路状态数据库是相同的（属于多个区域的路由器会为每个区域维护一份链路状态数据库）。

OSPF最主要的特征是使用分布式的链路状态协议（link state protocol），而不是相RIP（路由信息协议）那样的距离向量协议。OSPF向本自治系统的所有路由器发送信息。这里是用的方法是洪泛法（flooding），这就是路由器通过所有输出端口向所有相邻的路由器发送信息。而每一个相邻路由器又再将此信息发往其所有的相邻路由器（不再发送给刚刚发来信息的那个路由器）。这样，最终整个区域中所有的路由器都得到了这个信息的一个副本。

发送的信息就是与本路由器相邻的所有路由器的链路状态，但这只是路由器所知道的部分信息。所谓“链路状态”就是说明本路由器和哪些路由器相邻，以及该链路的“度量”（metric）。OSPF将这个“度量”用来表示费用、距离、时延、带宽，等等。这些都是由网络管理人员来决定。因此较为灵活。

OSPF提出了“区域（Area）”的概念，一个网络可以由单一区域或者多个区域组成。其中，一个特别的区域被称为骨干区域（Backbone Area），该区域是整个OSPF网络的核心区域，并且所有其他的区域都与之直接连接。所有的内部路由都通过骨干区域传递到其他非骨干区域。所有的区域都必须直接连接到骨干区域，如果不能创建直接连接，那么可以通过虚拟链路（Virtual-link）和骨干区域创建虚拟连接。

同一个广播域（Broadcast Domain）的路由器或者一个点对点（Point To Point）连接的两端的路由器，在发现彼此的时候，创建邻接（Adjacencies）。多路访问网络以及非广播多路访问网络的路由器会选举指定路由器（Designated Router，DR）和备份指定路由器（Backup Designated Router，BDR），DR和BDR作为网络的中心负责路由器之间的信息交换从而降低了网络中的信息流量。OSPF协议同时使用单播（Unicast）和组播（Multicast）来发送Hello包和链路状态更新（Link State Updates），使用的组播地址为224.0.0.5和224.0.0.6。OSPF协议不使用TCP或者UDP协议而是承载在IP协议之上，IP协议号为89，工作在OSI模型的传输层。节点在创建邻接，接受链路状态通告（Link-state Advertisement，LSA）时，可以通过MD5或者明文进行安全验证。

## 1.2 术语定义

**路由器/Router:**

一种三层IP包的交换设备。在早期的IP文献中被称为网关/gateway。

**自制系统/Autonomous System:**

一组使用相同路由协议交换路由信息的路由器，缩写为AS。

**内部网关协议/Interior Gateway Protocol:**

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。

**路由器标识/Router ID:**

一个32位的数字，用以识别每台运行OSPF协议的路由器。在一个AS中，这个数字可以唯一地表示出一台路由器。

**网络/Network:**

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。物理点对点/point-to-point网络是个例外--无论在上面

设置了多少网络/子网号(如果有的话)，都将其看作是一个网络。

**网络掩码/Network mask:**

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。如将C类IP地址的网络掩码显示为0xffffff00，这一掩码在其他文献中经常被表示为255.255.255.0。

**点对点网络/Point-to-point networks:**

仅仅连接一对路由器的网络。56k的串行线路是一个点对点网络的例子。

**广播网络/Broadcast networks:**

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。以太网/ethernet是一个广播网络的例子。

**非广播网络/Non-broadcast networks:**

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。X.25公用数据网/Public Data Network(PDN)是一个非广播网络的例子。

在非广播网络上运行的OSPF有两种模式。第一种被称为非广播多路接入/non-broadcast multi-access(NBMA)，模拟OSPF在广播网络上的操作;第二种被称为点对多点/Point-to-Multipoint，将非广播网络看作是一系列点对点的连接。非广播网络被作为NBMA网络还是点对多点网络，取决于OSPF在该网络上所配置的运行模式。

**接口/Interface:**

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/link。

**邻居路由器/Neighboring routers:**

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。

**邻接/Adjacency:**

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。

**连接状态宣告/Link state advertisement:**

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洪泛到整个路由域中。所有路由器和网络连接状态宣告的集合形成了协议的连接状态数据库。在本备忘录中，连接状态宣告被缩写为LSA。

**Hello协议/Hello Protocol:**

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。

**洪泛/Flooding:**

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。

**指定路由器/Designated Router:**

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。

**下层协议/Lower-level protocols:**

为IP及OSPF协议提供服务的下层网络接入协议。如为X.25PDN服务的X.25packetandframelevel，以及为以太网服务的以太网数据链路层。

# 2 软件设计

## 2.1 功能设计

本实验要实现的是一个简化的OSPF协议路由器软件，软件名称为uOSPF。软件将支持OSPF协议中最基本的功能，并能与H3C的Quidway系列商用路由器进行通信和协议包交换并配合完成IP包的路由。

**实现的主要功能：**

* 支持多个接口，即多块网卡。可为每个接口配置区域标识。
* 支持广播网络，下层协议为以太网Ethernet。
* 通过Hello协议包发现并维持邻居路由器。
* 通过Database Description协议包与邻接路由器交换链路状态数据库的信息。
* 在交换链路状态数据库的信息过程中支持主/从两种模式。
* 通过Link State Request协议包向邻路由器接请求链路状态信息。
* 通过Link State Update协议包向邻接更新链路请求信息。
* 通过Link State Acknowledge协议包向邻居确认收到的链路状态信息。
* 支持生成本路由器的Router-LSA。
* 维护链路状态数据库，定期更新LSA。
* 支持解析1,2,3,4类LSA（Router-LSA、Network-LSA、Summary-LSA）。
* 分析链路状态信息，通过Dijkstra算法生成最短路径树，进而生成路由表。
* 通过route命令将路由表项写入操作系统内核路由表，实现包的转发。
* 支持输出部分中间状态信息，用于监视和调试。

**未实现的主要功能：**

* 不支持非广播多路访问网络（NBMA）、点到多点网络和虚链路。
* 不支持对OSPF协议包的验证（Authentication）。
* 不支持成为指定路由器DR和备用指定路由器BDR。
* 不支持成为边界路由器ABR和自治系统边界路由器ASBR。
* 不支持老化链路状态数据库。
* 不支持生成Network-LSA、Network-LSA、Summary-LSA及AS-external-LSA。
* 不支持等值多路经的负载均衡。

**软件与硬件环境：**

软件运行在网络实验室的电脑上，电脑配有两块以太网卡，运行Ubuntu-10.04操作系统，gcc版本4.4.3。电脑可以与其它电脑及路由器组网。运行uOSPF软件的电脑可以作为路由器使用。

## 2.2 架构设计

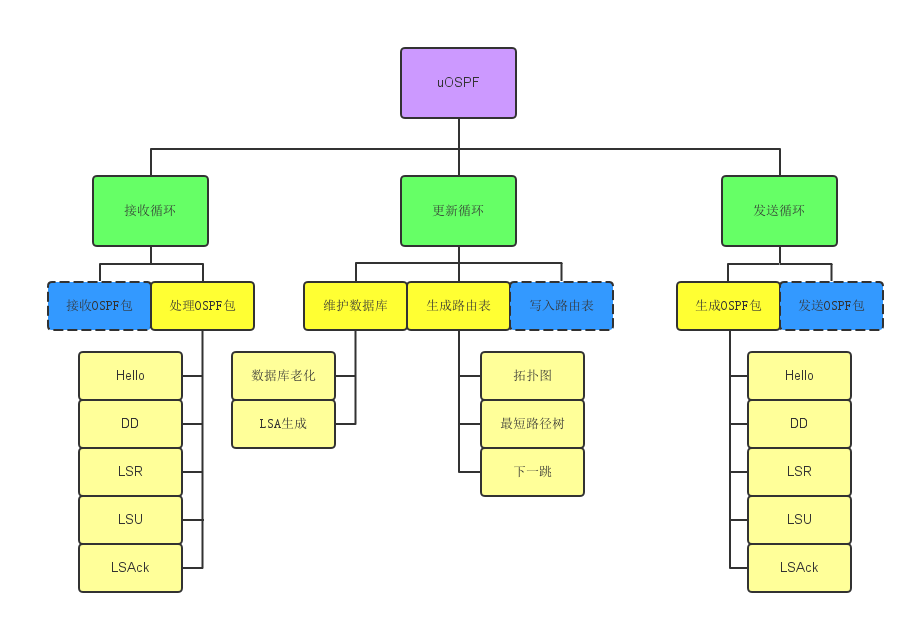


图 1 uOSPF整体架构

如图所示，uOSPF分为三个主要流程：接收循环，发送循环和更新循环，它们分别由三个线程来执行。在执行三个线程之前，由主线程进行接口的初始化以及相关的配置工作。

**接收循环：**

接收循环负责从每一个接口接收OSPF协议包，然后将协议包分为5类（Hello、DD、LSR、LSU、LSAck），分别调用不同的处理过程进行处理。

Hello包的处理过程包括发现新的邻居、将新邻居加入接口数据结构的邻居列表、初始化邻居数据结构、初始化邻居状态机、更新邻居状态。

DD包的处理过程包括确定主从关系、更新邻居状态、将区域数据结构的链路状态数据库中没有的LSA头部加入邻居数据结构的LSA请求列表。

LSR包的处理过程包括将LSR包的所有LSR加入邻居数据结构的LSR接收列表。

LSU包的处理过程包括提取所有LSA、验证LSA校验和、将LSA加入区域数据结构的链路状态数据库、将LSA头部加入邻居数据结构的Ack发送列表。

LSAck包的处理过程包括从邻居数据结构的LSR接收列表中删除被确认的LSR。

**发送循环：**

发送循环负责定期发送需要发送的OSPF协议包。发送循环分别调用5类OSPF包（Hello、DD、LSR、LSU、LSAck）的生成过程来生成需要发送的OSPF协议包，然后将包发别发送到对应接口的对应目地地址。对于需要超时重新发送的包，发送循环会触发对应的计时器然后重新发送。

Hello包的生成过程包括填写协议选项、填写路由器优先级等参数、将最近接收到Hello包的路由器列表写入Hello包。

DD包的生成过程包括填写协议选项、填写主从、更多、初始标识，填写DD包序号，将区域数据结构的链路状态数据库LSA头部写入DD包。

LSR包的生成过程包括根据邻居数据结构的LSA请求列表将LSA请求信息写入LSR包。

LSU包的生成过程包括根据邻居数结构的LSR接收列表查找到对应LSA并将其加入LSU包、将新生成的LSA加入LSU包。

LSAck包的生成过程包括将邻居数据结构的Ack发送列表写入LSAck包。

**更新循环：**

更新循环负责维护数据库，生成路由表和写入路由表。更新循环定期调用数据库老化过程对链路数据库进行老化（尚未实现），调用LSA生成过程生成新的LSA（仅支持生成Router-LSA），然后调用路由表生成过程生成新的路由表，最后通过route命令将新生成的路由表项写入操作系统内核的路由表。

Router-LSA生成过程包括填写LS序号、LS时限、LS选项等相关参数、为该区域的接口接入的每一个网络生成连接信息、计算LSA校验和、将新生成的LSA加入区域数据结构的链路状态数据库。

路由表生成过程包括读取区域数据结构的链路状态数据库、根据链路状态信息建立网络拓扑图、调用Dijkstra算法生成网络拓扑图的最短路径树、为每一个目标计算下一跳、为每一个可达目标生成一条路由表项并写入区域数据结构的路由表。

# 3 软件实现

## 3.1 模块分析

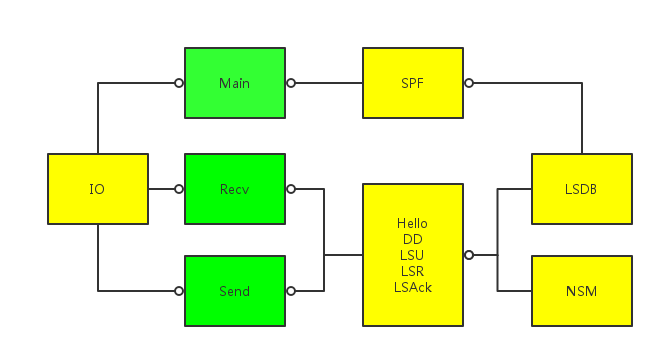


图 2 uOSPF模块关系

### 3.1.1 Main

文件：main.c

引用：IO、LSDB、Recv、Send、SPF

描述：主模块，通过main()函数实现程序的初始化和退出工作。

函数：

**main()**

程序的入口。程序启动启动时负责定义协议相关参数、初始化网卡、初始化接口数据结构、初始化区域数据结构。然后启动t\_recv和t\_send两个线程，分别执行接收循环和发送循环，自己作为主线程执行更新循环。程序结束前终结两个分支线程。

### 3.1.2 Recv

文件：recv.h、recv.c

引用：DD、Hello、IO、LSAck、LSR、LSU

描述：接收模块，实现接收循环。

函数：

**recv\_loop()**

接收循环。通过IO模块提供的recv\_ospf()函数接收OSPF协议包。找到包对应邻居后对接收到的包进行分类，调用Hello、DD、LSR、LSU、LSAck模块提供的对应包处理函数进行处理。

### 3.1.3 Send

文件：send.h、send.c

引用：DD、Hello、IO、LSAck、LSDB、LSR、LSU、NSM

描述：发送模块，实现发送循环。

函数：

**send\_loop()**

发送循环。定期删除接口数据结构的不活跃邻居。定期调用Hello、DD、LSR、LSU、LSAck模块提供的包生成函数生成对应类型的OSPF协议包，然后调用IO模块提供的send\_ospf()函数发送生成的OSPF协议包达对应目的地址。如果包超时需要重传，则触发超时计数器，执行重传。包成功发送后，将对应数据结构的需要发送包的列表清空。调用NSM模块提供的add\_event()函数更新邻居状态。

### **3.1.4 IO**

文件：io.h、io.c

引用：无

描述：IO模块，实现物理接口（网卡）初始化、接收OSPF协议包和发送OSPF协议包。

函数：

**if\_init()**

网卡初始化函数。通过Linux操作系统提供的ioctl接口获取网卡信息、设置网卡参数、打开网卡混杂模式（pro-misc mode）。

**cksum()**

校验和计算函数。使用标准16位1补足算法计算OSPF协议包的校验和。校验和的计算不包含64位验证域。

**recv\_ospf()**

OSPF协议包接收函数。使用Linux操作系统提供的recvfrom接口接收IP包，如果是OSPF协议包并且校验和正确，则接受这个OSPF协议包并记录源地址，最后返回接收到包的接口地址。

**send\_ospf()**

OSPF协议包发送函数。为传入的OSPF协议包填写头部，并计算校验和。然后将OSPF协议包封装成IP协议包，最后使用Linux操作系统提供的sendto接口发送IP协议包到传入的目的地址。

### 3.1.5 Hello

文件：hello.h、hello.c

引用：NSM

描述：Hello包模块，实现Hello包的处理和生成。

函数：

**process\_hello()**

Hello包处理函数。解析Hello包的报文，将新的邻居插入接口数据结构的邻居列表，初始化邻居数据结构，初始化邻居状态机，检查报文的邻居路由器列表中是否有自己并更新邻居状态。更新邻居状态时调用NSM模块提供的add\_event()函数。

**produce\_hello()**

Hello包生成函数。生成Hello包的报文，填入自己的路由器信息和选项参数，将接口数据结构的邻居列表中的邻居路由器标识（router\_id）填入报文。

### 3.1.6 DD

文件：dd.h、dd.c

引用：LSDB、NSM

描述：Database Description包模块，实现DD包的处理和生成。

函数：

**process\_dd()**

DD包处理函数。解析DD包的报文，根据I标识位判断是否需要协商，根据MS标识位确定主/从关系，根据主/从关系确定邻居数据结构的DD包序号，根据M标识位确定是否还有更多DD包。然后根据主/从关系和是否有更多包更新邻居数据结构。更新邻居状态时调用NSM模块提供的add\_event()函数。最后将DD包的报文中的LSA头部与区域数据结构的链路状态数据库中的LSA头部对比，将链路状态数据库中没有的LSA头部加入邻居数据结构的LSA请求列表。查询链路状态数据库时调用LSDB模块的提供的find\_lsa()函数，比较LSA头部时调用LSDB模块提供的cmp\_lsa()函数。

**produce\_dd()**

DD包生成函数。生成DD包的报文，填入DD包选项参数，根据邻居数据结构的主/从关系、邻居状态、是否还有更多DD包等填写标识位，填入邻居数据结构的DD包序号。将区域数据结构的链路状态数据库中的LSA头部填入DD包的报文。

### 3.1.7 LSR

文件：lsr.h、lsr.c

引用：无

描述：Link State Request包模块，实现LSR的比较，LSR包的处理和生成。

函数：

**lsr\_eql()**

LSR判等函数。判断两个LSR是否相等。

**process\_lsr()**

LSR包处理函数。解析LSR包的报文，提取出LSR，将新的LSR加入邻居数据结构的LSR接收列表。判断LSR是与已有的LSR相同时调用lsr\_eql()函数。

**produce\_lsr()**

LSR包生成函数。生成LSR包的报文，将邻居数据结构的LSA请求列表的请求信息填入报文。

### 3.1.8 LSU

文件：lsu.h、lsu.c

引用：LSDB

描述：Link State Update包模块，实现LSU包的处理和生成。

函数：

**process\_lsu()**

LSU包处理函数。解析LSU包的报文，提取出LSA。调用LSDB模块提供fletcher16()函数计算LSA的校验和，若校验和不正确，则舍弃这个LSA。将新的LSA加入区域数据结构的链路状态数据库。查找已存在的相同LSA头部时调用LSDB模块的lsah\_eql()函数。将LSA头部加入邻居数据结构的Ack发送列表。

**produce\_lsu()**

LSU包生成函数。生成LSU包的报文，根据邻居数据结构的LSR接收列表的请求信息，在区域数据结构的链路状态数据库中查找对应的LSA，如果找到，则将对应LSA填入报文。

**produce\_upd()**

LSU包生成函数。生成LSU包的报文，将传入的Router-LSA填入报文。

### 3.1.9 LSAck

文件：lsack.h、lsack.c

引用：无

描述：Link State Acknowledge包模块，实现LSAck的处理和生成。

函数：

**process\_ack()**

LSAck包处理函数。解析LSAck包的报文，提取出被确认的LSA头部，将邻居数据结构的LSR接收列表中对应的LSR删去。

**produce\_ack()**

LSAck包生成函数。生成LSAck包的报文，将邻居数据结构的Ack发送列表中的LSA头部填入报文。

### 3.1.10 LSDB

文件：lsdb.h、lsdb.c

引用：无

描述：链路状态数据库模块，实现LSA的校验和计算、Router-LSA生成、LSA头部比较以及加入LSA头部，查找LSA和插入LSA。

函数：

**get\_lsa\_seq()**

LSA序号函数。返回下一个LSA序号，初始值0x80000001。

**fletcher16()**

校验和计算函数。使用16位Fletcher算法计算校验和。

**gen\_router\_lsa()**

Router-LSA生成函数。填写LSA头部的参数、选项等域。填写Router-LSA的标识位。根据区域数据结构中各个接口接入的网络的具体情况填入连接信息，主要分为残根网络（StubNet）和传输网络（TransNet）两类。填入LSA头部的长度域，调用fletcher16()函数计算校验和并填入LSA头部。调用insert\_lsa()函数将新生成的Router-LSA插入区域数据结构的链路状态数据库。最后返回新生成的Router-LSA。

**lsah\_eql()**

LSA头部判等函数。判断两个LSA头部是否相等。

**cmp\_lsah()**

LSA头部比较函数。根据OSPF协议的比较细则的比较两个LSA头部的新旧关系。

**add\_lsah()**

LSA头部加入函数。将传入的LSA头部加入传入的邻居数据结构的LSA请求列表。

**find\_lsa()**

LSA查找函数。在传入的区域数据结构的链路状态数据库中查找和传入的LSA头部相等的LSA。判断LSA头部相等时调用lsah\_eql()函数。

**insert\_lsa()**

LSA插入函数。在传入的区域数据结构的链路状态数据库中查找与传入的LSA头部相等的LSA。若没有查找到LSA或查找到的LSA比传入的LSA更旧，就将新的LSA插入链路状态数据库，并返回插入的LSA。否则返回NULL。

### 3.1.11 NSM

文件：nsm.h、nsm.c

引用：无

描述：邻居状态机模块，定义邻居状态和状态转移矩阵，实现邻居状态机。

函数：

**add\_event()**

添加事件函数。将时间加入邻居状态机，邻居转台机根据状态转移矩阵确定新的邻居状态，并进一步更新接口状态。

### 3.1.12 SPF

文件：spf.h、spf.c

引用：无

描述：最短路径有限算法模块，实现网络拓扑图的建立、最短路径树的生成、路由表的生成。

函数：

**find\_node()**

节点查找函数。根据输入的目标标识在网络拓扑图中查找对应的节点。

**dijkstra()**

Dijkstra算法实现函数。使用Dijkstra算法生成网络拓扑图的最短路径树。为每一个非叶子节点计算下一跳节点（路由器标识）。在最短路径树上加入叶子节点。查找目标标识对应节点时调用find\_node()函数。

**spt()**

最短路径树生成函数。分析传入的所有LSA，为第1、2类LSA描述的网络目标建立对应的节点，即建立网络拓扑图。调用dijkstra()函数生成网络拓扑图的最短路径树，并得到叶子节点。调用find\_node()函数查找第3、4类LSA描述的网络目标对应的叶子节点，并填入节点信息。第5类LSA描述的网络目标的计算尚未实现。

**find\_nbr\_ip()**

邻居查找函数。在传入的区域数据结构的接口数据结构的邻居列表中查找传入的路由器标识（Router ID）对应的邻居，并返回其接口的IP地址。

**find\_if\_name()**

接口查找函数。在传入的区域数据结构的接口中查找传入的IP地址对应的接口，并返回其名称。

**update\_table()**

路由表生成函数。调用spt()函数生成传入区域的最短路径树。然后将生成的最短路径树转化为路由表项加入到路由表中。简历路由表时调用find\_nbr\_ip()函数和find\_if\_name()函数。

## 3.2 代码分析

### 3.2.1 接口初始化

void if\_init() {

struct ifconf ifc;

struct ifreq ifrs[NUM\_INTERFACE];

sock = socket(AF\_PACKET, SOCK\_DGRAM, htons(ETHERTYPE\_IP));

ifc.ifc\_len = sizeof(ifrs);

ifc.ifc\_req = ifrs;

ioctl(sock, SIOCGIFCONF, &ifc);

int num = ifc.ifc\_len / sizeof(struct ifreq);

num\_if = 0;

for (int i = 0; i < num; ++i) {

/\* except loop-back \*/

if (strcmp(ifrs[i].ifr\_name, "lo")) {

/\* interface name \*/

strcpy(ifs[num\_if].name, ifrs[i].ifr\_name);

/\* ip address \*/

ioctl(sock, SIOCGIFADDR, ifrs + i);

ifs[num\_if].ip =

((struct sockaddr\_in \*)&ifrs[i].ifr\_addr)->sin\_addr.s\_addr;

/\* subnet mask \*/

ioctl(sock, SIOCGIFNETMASK, ifrs + i);

ifs[num\_if].mask =

((struct sockaddr\_in \*)&ifrs[i].ifr\_netmask)->sin\_addr.s\_addr;

/\* turn on promisc mode \*/

ioctl(sock, SIOCGIFFLAGS, ifrs + i);

ifrs[i].ifr\_flags |= IFF\_PROMISC;

ioctl(sock, SIOCSIFFLAGS, ifrs + i);

/\* bind socket to this interface \*/

ifs[num\_if].sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_OSPF);

setsockopt(ifs[num\_if].sock, SOL\_SOCKET, SO\_BINDTODEVICE,

ifrs + i, sizeof(struct ifreq));

++num\_if;

}

}

}

函数首先使用Linux操作系统提供的socket接口创建一个用于IP网络的套接字sock，用于接收OSPF协议包。然后使用这个套接字和ioctl接口来初始化接口。使用struct ifconf来获取接口配置信息和所有接口。使用struct ifreq来获取每一个接口的信息。并且使用socket接口和setsockopt接口为每一个接口创建并绑定一个套接字，用于发送OSPF协议包。

### 3.2.2 Router-LSA生成

lsa\_header \*gen\_router\_lsa(area \*a) {

uint8\_t buff[BUFFER\_SIZE];

lsa\_header \*lsah = (lsa\_header \*)buff;

router\_lsa \*rlsa = (router\_lsa \*)(buff + sizeof(lsa\_header));

struct link \*l = rlsa->links;

lsah->age = 0;

lsah->options = OPTIONS;

lsah->type = OSPF\_ROUTER\_LSA;

lsah->id = myid;

lsah->ad\_router = myid;

lsah->seq\_num = get\_lsa\_seq();

rlsa->flags = 0x00;

rlsa->zero = 0x00;

for (int i = 0; i < a->num\_if; ++i) {

if (a->ifs[i]->num\_nbr == 0) {

/\* StubNet \*/

l->type = ROUTERLSA\_STUB;

l->id = a->ifs[i]->ip & a->ifs[i]->mask;

l->data = a->ifs[i]->mask;

l->tos = 0;

l->metric = a->ifs[i]->cost;

++l;

} else {

/\* Broadcast \*/

if (a->ifs[i]->state == 1) {

/\* TransNet \*/

l->type = ROUTERLSA\_TRANSIT;

l->id = a->ifs[i]->dr;

l->data = a->ifs[i]->ip;

l->tos = 0;

l->metric = a->ifs[i]->cost;

++l;

} else {

/\* StubNet \*/

l->type = ROUTERLSA\_STUB;

l->id = a->ifs[i]->ip & a->ifs[i]->mask;

l->data = a->ifs[i]->mask;

l->tos = 0;

l->metric = a->ifs[i]->cost;

++l;

}

}

}

rlsa->num\_link = htons((l - rlsa->links));

size\_t len = sizeof(lsa\_header) + sizeof(router\_lsa) +

(l - rlsa->links) \* sizeof(struct link);

lsah->length = htons(len);

lsah->checksum = 0;

lsah->checksum = htons(fletcher16(buff + sizeof(lsah->age),

ntohs(lsah->length) - sizeof(lsah->age)));

return insert\_lsa(a, lsah);

}

函数首先填写LSA的头部，然后根据区域数据结构中各个接口接入的网络的具体情况产生连接信息。若接口连接道的网络上没有接入邻居路由器，则该网络是一个残根网络（StubNet），否则是一个广播网络，若接口与网络的指定路（DR）由器建立了完全邻接，则是一个传输网络（TransNet），否则是一个残根网络（StubNet）。之后，计算LSA的校验和，使用的fletcher16()函数来自GNU Zebra。最后，将新生成的Router-LSA插入区域数据结构的链路状态数据库。

### 3.2.3 最短路径树

void spt(lsa\_header \*const lsas[], int num\_lsa) {

int root;

num\_node = 0;

/\* lsa type 1 and lsa type 2 \*/

for (int i = 0 ; i < num\_lsa; ++i) {

if (lsas[i]->type == OSPF\_ROUTER\_LSA || lsas[i]->type == OSPF\_NETWORK\_LSA) {

nodes[num\_node].id = lsas[i]->id;

nodes[num\_node].lsa = lsas[i];

nodes[num\_node].dist = DISTANCE\_INF;

if (nodes[num\_node].id == myid) root = num\_node;

++num\_node;

}

}

dijkstra(root);

/\* lsa type 3 ans lsa type 4 \*/

for (int i = 0 ; i < num\_lsa; ++i) {

if (lsas[i]->type == OSPF\_SUMMARY\_LSA ||

lsas[i]->type == OSPF\_ASBR\_SUMMARY\_LSA) {

summary\_lsa \*slsa = (summary\_lsa \*)((uint8\_t \*)lsas[i] +

sizeof(lsa\_header));

if (find\_node(lsas[i]->id) < num\_node) continue;

int k = find\_node(lsas[i]->ad\_router);

if (k == num\_node || nodes[k].dist == DISTANCE\_INF) continue;

nodes[num\_node].id = lsas[i]->id;

nodes[num\_node].mask = slsa->mask;

nodes[num\_node].next = nodes[k].next;

nodes[num\_node].dist = nodes[k].dist + ntohl(slsa->metric >> 4 << 4);

nodes[num\_node].lsa = lsas[i];

++num\_node;

}

}

/\* lsa type 5 \*/

//TODO

}

分析传入的所有LSA，为第1、2类LSA描述的网络目标建立对应的节点，并填入节点属性，即建立网络拓扑图。调用dijkstra()函数生成网络拓扑图的最短路径树，并得到叶子节点。调用find\_node()函数查找第3、4类LSA描述的网络目标对应的叶子节点，并填入节点属性。第5类LSA描述的网络目标的计算尚未实现。

### 3.2.4 邻居状态机

const transition nsm[] = {

{S\_Down, E\_HelloReceived, S\_Init},

{S\_Init, E\_2WayReceived, S\_2Way},

{S\_2Way, E\_1WayReceived, S\_Init},

{S\_2Way, E\_AdjOK, S\_ExStart},

{S\_ExStart, E\_NegotiationDone, S\_Exchange},

{S\_Exchange, E\_ExchangeDone, S\_Loading},

{S\_Loading, E\_LoadingDone, S\_Full}

};

void add\_event(interface \*iface, neighbor \*nbr, nbr\_event event) {

const int num = 8;

puts("--------------------------------");

printf("Interface: %s\n", iface->name);

printf("Neighbor: %s\n", inet\_ntoa((struct in\_addr){nbr->ip}));

printf("Old State: %s\n", state\_str[nbr->state]);

printf("Event: %s\n", event\_str[event]);

for (int i = 0; i < num; ++i)

if ((nbr->state == nsm[i].src\_state) && (event == nsm[i].event)) {

nbr->state = nsm[i].dst\_state;

if (nbr->state == S\_Full)

if (iface->dr == nbr->ip) iface->state = 1;

break;

}

printf("New State: %s\n", state\_str[nbr->state]);

puts("--------------------------------");

}

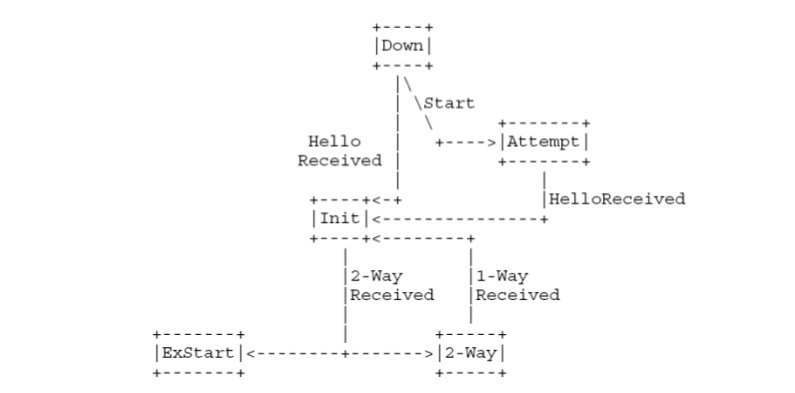


图 3 邻居状态转移图

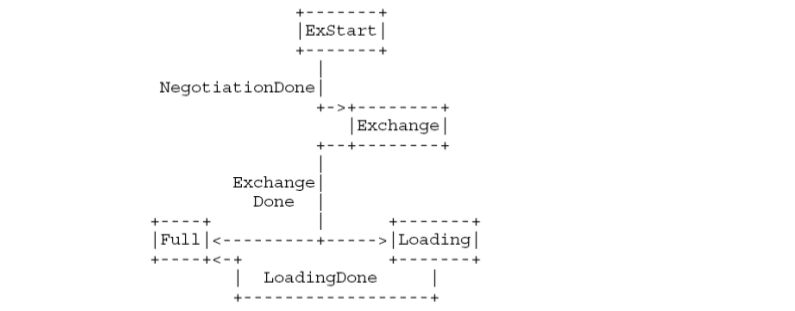


图 4 邻居状态转移图

程序实现了邻居状态机，是上图所示邻居状态转移图的抽象和简化。状态转移矩阵nsm定义初始状态，触发事件和目标状态。当新的事件发生时，调用add\_event()函数。函数检索状态转移矩阵，若初始状态是输入的邻居数据结构的状态，且触发事件是输入的事件，则将输入的邻居数据结构的状态更新为目标状态。进一步更新接口状态。函数会输出状态转移的过程信息到控制台。

### 3.2.5 OSPF协议包接收

uint16\_t cksum(const uint16\_t \*data, size\_t len) {

uint32\_t sum = 0;

while (len > 1) {

sum += \*data++;

len -= 2;

}

/\* mop up an odd byte, if necessary \*/

if (len) sum += \*(uint8\_t \*)data;

sum = (sum >> 16) + (sum & 0xffff);

sum += (sum >> 16);

return ~sum;

}

interface \*recv\_ospf(int sock, uint8\_t buf[], int size, in\_addr\_t \*src) {

struct iphdr \*iph;

ospf\_header \*ospfhdr;

for (;;) {

if (recvfrom(sock, buf, size, 0, NULL, NULL) < sizeof(struct iphdr))

continue;

iph = (struct iphdr \*)buf;

\*src = iph->saddr;

if (iph->protocol == IPPROTO\_OSPF) {

ospfhdr = (ospf\_header \*)(buf + sizeof(struct iphdr));

memset(ospfhdr->auth\_data, 0, sizeof(ospfhdr->auth\_data));

if (cksum((uint16\_t \*)ospfhdr, ntohs(ospfhdr->length))) continue;

printf("%s received from %s\n", ospf\_type\_name[ospfhdr->type],

inet\_ntoa((struct in\_addr){\*src}));

for (int i = 0; i < num\_if; ++i)

if ((iph->saddr & ifs[i].mask) == (ifs[i].ip & ifs[i].mask)) {

return ifs + i;

}

}

}

}

函数首先使用Linux操作系统提供的recvfrom接口和套接字sock接收IP包，如果包中封装的不是OSPF协议包，则丢弃该IP包。然后连同校验和域一起计算ospf协议包的校验和，若和不为零，即校验和不正确，则丢弃该OSPF协议包。否则接受这个OSPF协议包并记录下包的源地址。最后在所有接口中查找接入到源地址所在网络中的接口并返回这个接口。

# 4 实验成果

## 4.1 使用方法

首先使用make命令构建软件，得到可执行文件uospf。Makefile文件内容如下：

CC = gcc

CFLAGS = -std=gnu99 -Wall -g -lpthread

SRC = $(wildcard \*.c)

HEAD = $(wildcard \*.h)

OBJ = $(SRC:.c=.o)

uospf: $(HEAD) $(OBJ)

$(CC) $(CFLAGS) $(LIBS) -o $@ $(OBJ)

$(OBJ): $(HEAD) $(SRC)

$(CC) -c $(CFLAGS) $(SRC)

.PHONY: clean

clean:

rm -f \*.o \*.gch uospf

由于实验室的gcc版本较旧，只能使用gnu99标准通过编译，但仍可能造成BUG。建议使用c11或更新的标准。程序还需要链接到pthread库。

在运行uospf程序之前，以root用户身份执行以下命令，使得Linux操作系统内核可以根据内核中的路由表转发接收到的IP包：

echo 1 > /proc/sys/net/ipv4/ip\_forward

然后以root用户身份使用./uospf命令启动程序，在程序提示下配置接口的区域标识（Area ID）。然后等待程序与网络中的其它路由器进行连接。运行状态如下图。

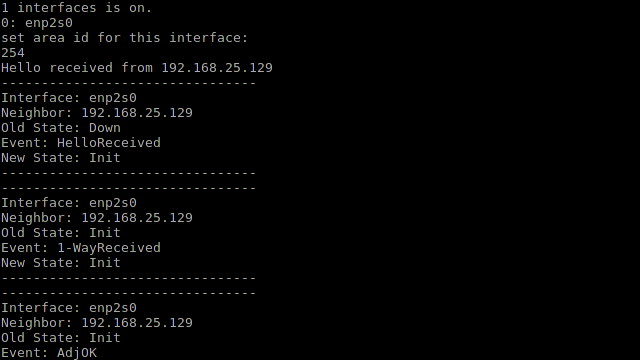
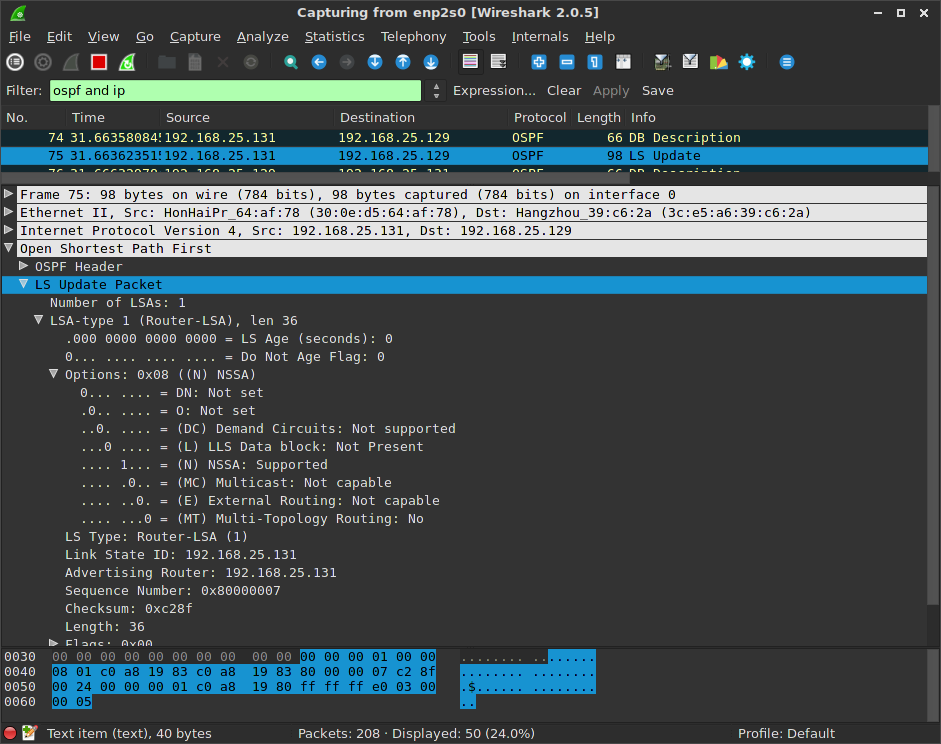
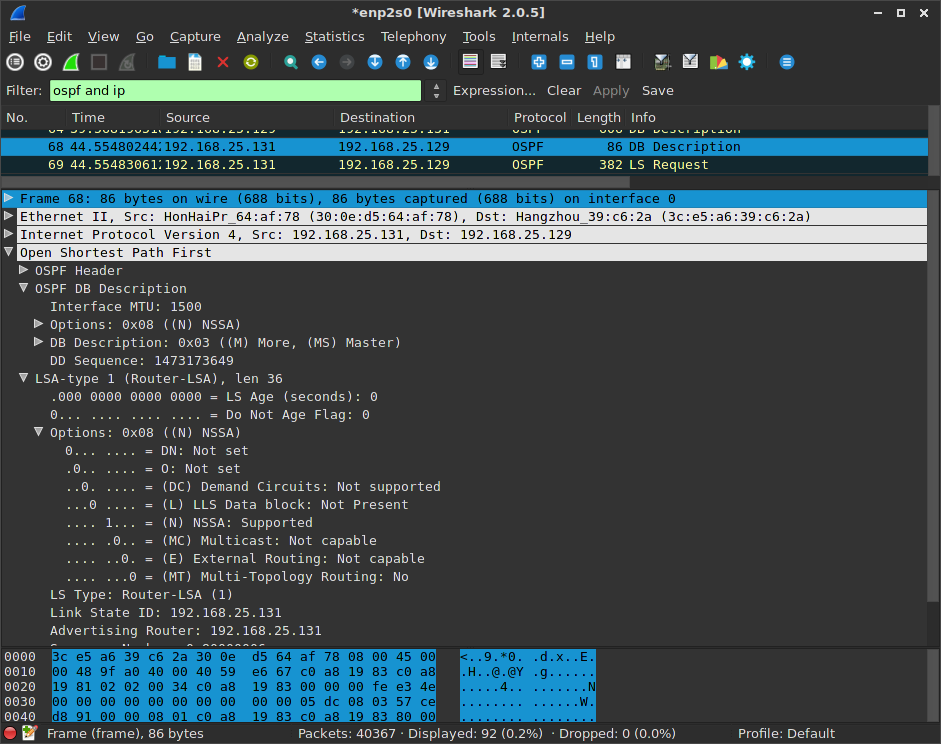


图 5 uOSPF程序运行状态截图

## 4.2 实验结果

实验过程中使用了两台路由器和三台PC，一台PC使用两块网卡并运行uOSPF软件，作为实验路由器。实验按照《计算机网络实验教程》第4章中的实验进行组网，实验路由器工作良好，与邻居路由器建立了正常连接，与指定路由器（DR）建立了完全邻接，并同步了链路状态数据库，将路由表写入了系统内核路由表中，最终实现了各网络间包的转发和路由。实验还尝试了其他多区域的网络划分的组网方式，实验路由器亦能完成跨区域的包转发。

实验过程中使用Wireshark软件截获了相关OSPF报文，截图如下：



# 5 不足与期望

uOSPF只实现了OSPF协议的一个功能子集。

程序只支持了1-4类LSA，第5类乃至更多的LSA可以在将来实现。

程序使用了多线程设计，但多个线程在访问临界资源是存在冲突隐患。将来应该加入互斥锁等结构来防止冲突。

程序没有实现数据库老化，这可以在将来加入。

程序的接口可配置选项较少，需要另行通过命令行工具配置。将来可以将入程序中，使得路由软件的配置更加方便。

程序使用route命令来操作系统内核路由表，这依赖于命令。将来可以使用Linux操作系统提供的相应接口来操作内核路由表，实现更多的功能和更高的效率。

程序在设计上没有支持成为指定路由器（DR）或备份指定路由器（BDR），也没有留出相应接口。但是经过一定的修改，加入指定路由器所需有的特定功能，也许可以在将来实现。

1. OSPF协议的详细描述可以参见文件「FC2328 OSPF v2 中文版」。 [↑](#footnote-ref-0)