北京航空航天大學

研究型网络实验报告

OSPF 协议子集的具体设计与实现

学 生 姓 名　　黄 建 宇

学 生 学 号　　39061416

指 导 教 师　　张 力 军

培 养 院 系　　计算机学院（高等工程学院）

摘　要

OSPF 已成为目前 Internet 广域网和 Intranet 企业网采用最多、应用最广泛的路由协议之一，OSPF 是 Open Shortest Path First（开放最短路由优先协议）的缩写。本项目对 OSPF 的一个子集进行了实现，并对其进行了模拟测试来验证其正确性和有效性，本课程论文是对一学期的网络实验提高层次课程的总结。

首先，具体分析 OSPF 协议的相关内容，阅读 RFC2328 文档相关内容。针对 RFC 2328 进行一些简化，提取出能够实现的子集部分。

其次，基于 Linux 实现基本的 OSPF 协议：识别第 1、2、3、4、5 类报文种类；实现 OSPF 协议的格式数据的封装发送和接收解封装以及相应处理；实现 DR 的选举；利用 Hello 报文建立邻居->邻接关系；利用 DD 报文建立邻居 Master/Slave 关系；实现 LSA 的生成、交换与泛洪->建立 LSDB；生成最短路径树（SPF 算法，或称 Dijkstra 算法）；生成最终的路由表；实现路由转发功能。

最后，在 Linux 环境下对实现的 OSPF 协议进行测试。设计出四个实验进行全方面测试：通过实验 1（PC 与路由直接相连）验证与路由器等设备交互的正确性；通过实验 2（两个路由交互验证）验证路由器报文交互过程的正确性（五类报文）；通过实验 3（四个路由交互验证）验证生成的 LSDB、SPF 树、路由表的正确性；通过实验 4（PC 与路由多台验证）验证路由表转发的正确性。

关键词： OSPF　Linux　RFC　网络协议实现　测试

目　录

图表目录

第一章　绪论

1.1　引言

随着 Internet 技术在全球范围的飞速发展，OSPF 已成为目前 Internet 广域网和 Intranet 企业网采用最多、应用最广泛的路由协议之一。OSPF 是 Open Shortest Path First（开放最短路由优先协议）的缩写。它是 IETF 组织开发的一个基于链路状态的内部网关协议。目前针对 IPv4 协议使用的是 OSPF Version 2（RFC2328）。

OSPF 协议是由 Internet Engineering Task Force 的 OSPF 工作组所开发的，特别为 TCP/IP 网络而设计，包括明确的支持 CIDR 和标记来源于外部的路由信息。OSPF 也提供了对路由更新的验证，并在发送/接收更新时使用 IP 多播。此外，还作了很多的工作使得协议仅用很少的路由流量就可以快速地响应拓扑改变。

本项目的研究动机就是在上面所述的背景下产生的，在协议规定的基础上在 PC 机上实现 OSPF 协议，完成基本的功能。

图 1 即是计算实现的 OSPF 协议子集实现的测试图（**两个路由器验证发包收包的交互过程，四个路由器验证生成路由表的正确性**）。

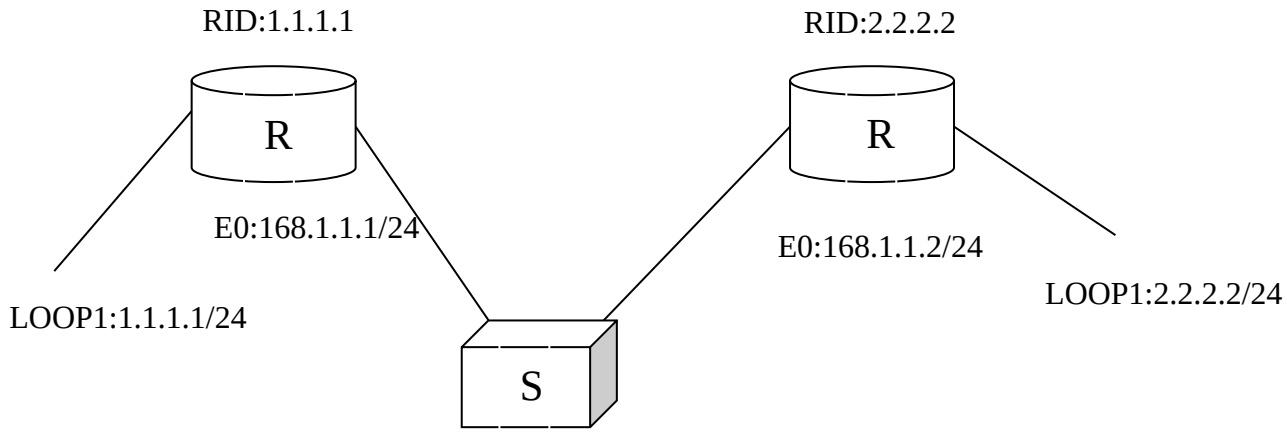


图 1. 1 OSPF 协议子集测试图 1

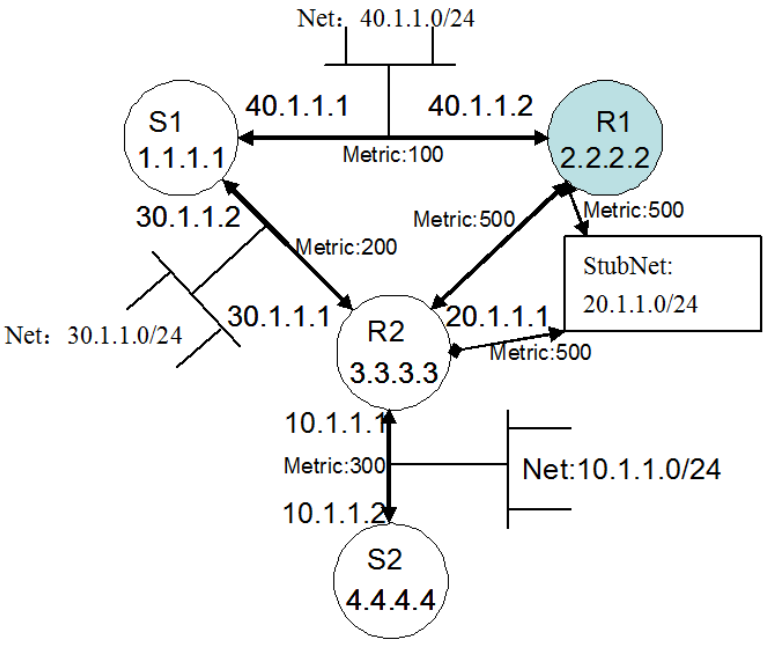


图 1. 2 OSPF 协议子集测试图 2

1.2 课题的来源和意义

本课题来源于网络实验提高层次题目，属于实验课程设计。在已有的网络实验课和前期学习的基础上自主独立实现 OSPF 协议，增进自己对协议的熟悉、提高协议编程的水平，并在这个过程中逐步积累相关经验。

1.3 相关技术的研究现状

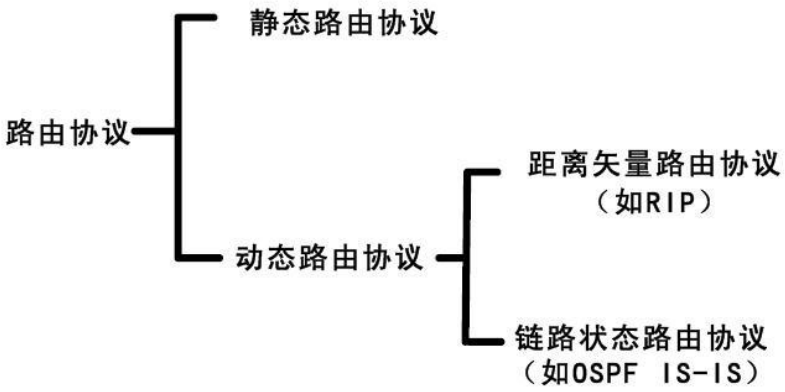


图 1. 3 路由协议分类

一般的路由表由这几种方法得出：

1　链路层协议发现的路由（也称为接口路由或直连路由）

2　由网络管理员手工配置的静态路由

3　动态路由协议发现的路由。

在小型网络当中，手工配置较为方便，但当路由表比较庞大的时候，使用静态路由显然不太现实了。

距离矢量协议也称为 Bellman-Ford 协议，网络中路由器向相邻的路由器发送它们的整个路由表。路由器在从相邻路由器接收到的信息的基础之上建立自己的路由表。然后，将信息传递到它的相邻路由器。这样一级级的传递下去以达到全网同步。每个路由器都不了解整个网络拓扑，它们只知道与自己直接相连的网络情况，并根据从邻居得到的路由信息更新自己的路由表，进行矢量行叠加后转发给其它的邻居。距离矢量算法存在的一个重要的问题就是会产生路由环路。路由环路问题产生的原因和距离矢量算法的原理有关，正如前面所讲的，每个路由器根据从其它路由器接收到的信息来建立自己的路由表。如果某个路由器出现“故障”或者因为别的原因而无法在网上使用时，就会造成路由环路。链路状态算法对路由的计算方法和距离矢量算法有本质的差别。距离矢量算法是一个平面式的,所有的路由表项学习完全依靠邻居,交换的是整个路由表项。链路状态是一个层次式的,执行该算法的路由器不是简单的从相邻的路由器学习路由，而是把路由器分成区域，收集区域内所有路由器的链路状态信息，根据链路状态信息生成网络拓扑结构，每一个路由器再根据拓扑结构图计算出路由。

1.4　研究目标和研究内容

1　学习 OSPF 协议，了解 OSPF 协议的原理；

2　基于 Linux 实现基本的 OSPF 协议；

3　实现 OSPF 协议规定格式数据的封装发送；

4　接受符合 OSPF 协议规定格式的数据包，并进行逻辑处理

5　根据 OSPF 原理生成单播路由表，并使得单播数据包能根据路由表进行转发

6　OSPF 需识别第 1、2、3、4、5 类报文种类

7　在 PC 机上实现 OSPF 路由协议

8　实现 OSPF 协议的格式数据的封装发送和接收解封装

9　实现 OSPF 路由协议的路由功能

10　在上面的基础上，实现 OSPF 协议的相关机制，逐步完善

第二章 OSPF（开放最短路径优先协议）概述

2.1 OSPF 概述

OSPF 协议是特别为 TCP/IP 网络而设计，包括明确的支持 CIDR 和标记来源于外部的路由信息。OSPF 也提供了对路由更新的验证，并在发送/接收更新时使用 IP 多播。此外，还作了很多的工作使得协议仅用很少的路由流量就可以快速地响应拓扑改变。

OSPF 用链路状态算法来计算在每个区域中到所有目的的最短路径，当一个路由器首先开始工作，或者任一个路由变化发生，这个配备给 OSPF 的路由器将 LSA 扩散到同一级区域内所有路由器，这些 LSA 包含这个路由器的链接状态和它与邻居路由器联系的信息，从这些 LSA 的收集中形成了链路状态数据库，在这个区域中的所有路由器都有一个特定的数据库来描述这个区域的拓扑结构。

在 OSPF 路由协议中，可以通过划分区域来分割整个自治系统，每一个区域都有着该区域独立的网络拓扑数据库及网络拓扑图。对于每一个区域，其网络拓扑结构在区域外是不可见的，同样，在每一个区域中的路由器对其域外的其余网络结构也不了解。这意味着 OSPF 路由域中的网络链路状态数据广播被区域的边界挡住了，这样做有利于减少网络中链路状态数据包在全网范围内的广播，也是 OSPF 将自治系统划分成很多个区域的重要原因。

OSPF 仅通过在 IP 包头中的目标地址来转发 IP 包。IP 包在 AS 中被转发，而没有被其他协议再次封装。OSPF 是一种动态路由协议，它可以快速地探知 AS 中拓扑的改变（例如路由器接口的失效），并在一段时间的收敛后计算出无环路的新路径。收敛的时间很短且只使用很小的路由流量。

2.2 OSPF 协议特点

适应范围—— OSPF 支持各种规模的网络，最多可支持几百台路由器。

快速收敛——如果网络的拓扑结构发生变化，OSPF 立即发送更新报文，使这一变化在自治系统中同步。

无自环——由于 OSPF 通过收集到的链路状态用最短路径树算法计算路由，故从算法本身保证了不会生成自环路由。

子网掩码——由于 OSPF 在描述路由时携带网段的掩码信息，所以 OSPF 协议不受自然掩码的限制，对 VLSM 提供很好的支持。

区域划分—— OSPF 协议允许自治系统的网络被划分成区域来管理，区域间传送的路由信息被进一步抽象，从而减少了占用网络的带宽。

等值路由—— OSPF 支持到同一目的地址的多条等值路由，即到达同一个目的地有多个下一跳，这些等值路由会被同时发现和使用。

路由分级—— OSPF 使用 4 类不同的路由，按优先顺序来说分别是：区域内路由、区域间路由、第一类外部路由、第二类外部路由。

支持验证——它支持基于接口的报文验证以保证路由计算的安全性。

组播发送—— OSPF 在有组播发送能力的链路层上以组播地址发送协议报文，不仅达到了广播的作用，而且最大程度的减少了对其他网络设备的干扰。

2.3 动态路由协议的几个要素

- 报文（或者叫消息）
  - 链路状态描述
  - 链路状态描述报文的格式
- 邻居的自动发现和维护机制
  - 清楚周围邻居的状态－邻居状态机
  - 有目的地发送邻居需要的链路状态信息
- 一套算法，根据搜集的信息计算最终结果
  - 最短路径优先算法－SPF 算法

2.4 OSPF 的五种协议报文

- Hello 报文
  - 发现及维持邻居关系，选举 DR，BDR。
- DD 报文
  - 本地 LSDB 的摘要
- LSR 报文
  - 向对端请求本端没有或对端的更新的 LSA

- LSU 报文
  - 向对方发送其需要的 LSA
- LSAck 报文
  - 收到 LSU 之后，进行确认

2.5 OSPF 协议根据链路层媒体不同分为以下四种网络类型

- Broadcast；Stub:只连接了一台路由器
- NBMA
- Point-to-Multipoint
- Point-to-Point

2.6 路由器根据在自治系统中的不同角色划分

- 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

2.7 LSA 分类

- Router-LSA 由每个路由器生成，描述了路由器的链路状态和花费，传递到整个区域
- Network-LSA，由 DR 生成，描述了本网段的链路状态，传递到整个区域
- Net-Summary-LSA，由 ABR 生成，描述了到区域内某一网段的路由，传递到相关区

域

- Asbr-Summary-LSA，由 ABR 生成，描述了到 ASBR 的路由，传递到相关区域
- AS-External-LSA，由 ASBR 生成，描述了到 AS 外部的路由，传递到整个 AS（STUB 区域除外）

2.8 SPF 算法与最短路径树

在 OSPF 路由协议中，最短路径树的树干长度，即 OSPF 路由器至每一个目的地路由器的距离，称为 OSPF 的 Cost 值。Cost 值应用于每一个启动了 OSPF 的链路，它是一个 16 bit 的整数，范围是 1～65535。SPF 算法也被称为 Dijkstra 算法，是 OSPF 路由协议的基础。SPF 算法将每一个路由器作为根（Root）来计算到每一个目的地路由器之间的距离，每一个路由器根据一个统一的数据库会计算出路由域的拓扑结构图，该结构图类似于一棵树，在 SPF 算法中，被称为最短路径树。

第三章 OSPF 路由的计算过程

3.1 环境初始化

根据自身配置初始化参数，主要是路由器和接口的配置信息，如 Router ID、IP、mask、area id 等等。

这些初始化信息是必须的，后面对报文进行封装时填写字段都需要使用这些信息，这些信息应该是有路由器底层支持提供给 OSPF 协议模块使用。

3.2 建立邻接关系

建立邻接关系比较复杂，主要为建立一个邻居状态机，然后进行数据的交换。这里需要注意的是邻居和邻接是两个完全不同的概念，收到 Hello 报文的 OSPF 路由器会检查报文中所定义的一些参数，如果双方一致就会形成邻居关系。形成邻居关系的双方不一定都能形成邻接关系，这要根据网络类型而定。只有当双方成功交换 DD 报文，并能交换 LSA 之后，才形成真正意义上的邻接关系。

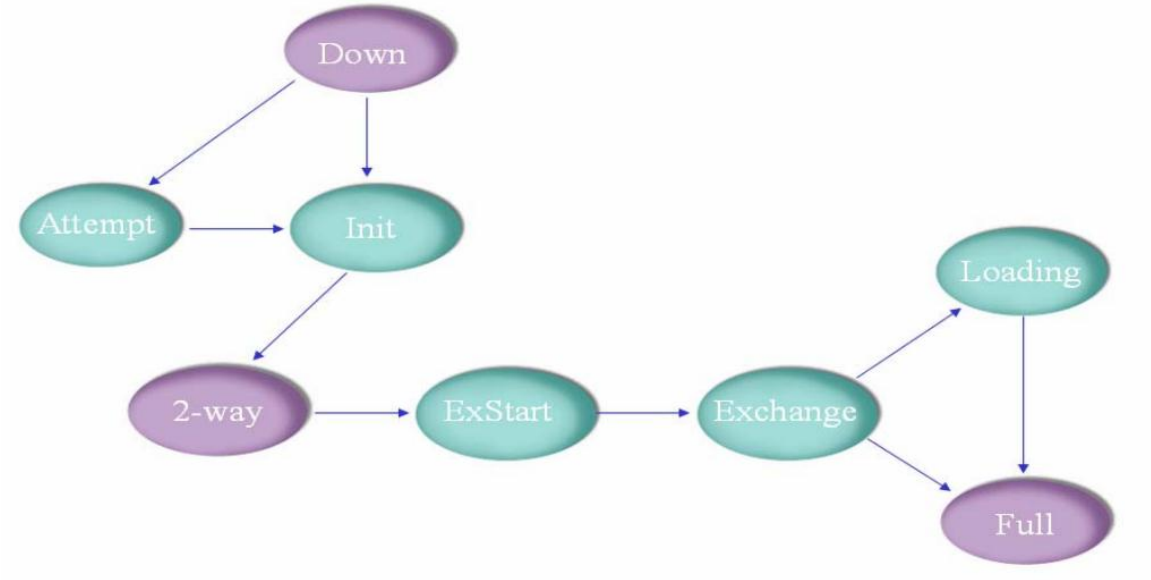


图 3. 1 OSPF 邻居状态机

HELLO Interval：接口上发送报文的时间间隔，以秒为单位。OSPF 邻居之间的 Hello 定时器的时间间隔要保持一致。Hello 定时器的值与路由收敛速度、网络负荷大小成反

比。

如果两路由器不具有相同的呼叫周期，则不能成为邻接关系。

DEAD Interval：如果在 DEAD TIME 指定的秒数内没有从已建立的邻居处收到报文，那么，邻居被宣布为故障状态。

邻接关系建立的图示：

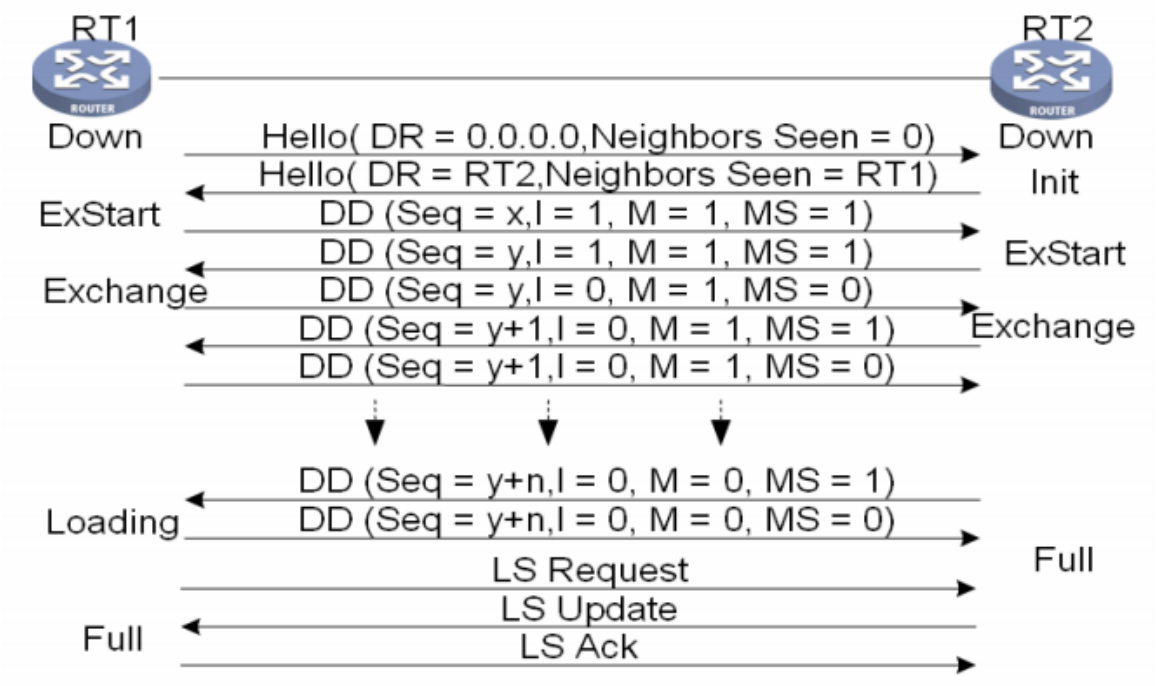


图 3. 2 OSPF 邻居关系建立图示

- Down
  - 邻居状态机的初始状态，是指在过去的 Dead-Interval 时间内没有收到对方的 Hello 报文
- Attempt
  - 只适用于 NBMA 类型的接口，处于本状态时，定期向那些手工配置的邻居发送 HELLO 报文
- Init
  - 本状态表示已经收到了邻居的 HELLO 报文，但是该报文中列出的邻居中没有包含本 Router ID（对方并没有收到我发的 HELLO 报文）
- 2-Way
  - 本状态表示双方互相收到了对端发送的 HELLO 报文，建立了邻居关系。在广播和 NBMA 类型的网络中，两个接口状态是 DROther 的路由器之间将停留在此状

态。其他情况状态机将继续转入高级状态。

- ExStart
  - 在此状态下，路由器和它的邻居之间通过互相交换 DD 报文（该报文并不包含实际的内容，只包含一些标志位）来决定发送时的主/从关系。建立主/从关系主要是为了保证在后续的 DD 报文交换中能够有序的发送。
- Exchange
  - 路由器将本地的 LSDB 用 DD 报文来描述，并发给邻居
- Loading
  - 路由器发送 LSR 报文向邻居请求对方的 LSU 报文
- Full
  - 在此状态下，邻居路由器的 LSDB 中所有的 LSA 本路由器全都有了。本路由器和邻居建立了邻接状态

3.3 DR 选举

在广播和 NBMA 类型的网络上，任意两台路由器之间都需要传递路由信息（flood），如果网络中有 N 台路由器，则需要建立 N *（N-1）/2 个邻接关系。

为了解决这个问题，OSPF 协议指定一台路由器 DR（Designated Router）来负责传递信息。所有的路由器都只将路由信息发送给 DR，再由 DR 将路由信息发送给本网段内的其他路由器。两台不是 DR 的路由器（DROther）之间不再建立邻接关系，也不再交换任何路由信息。

选举过程：

- 登记选民（所有 OSPF 路由器都定期发 hello 报文）
- 登记候选人（Priority>0）
- 竞选（声称自己是 DR）
- 投票（首先选 Priority 最大的，然后选 Router ID 最大的）

只有在广播和 NBMA 类型的接口上才会选举 DR，在 point-to-point 和 point-to-muiltipoint 类型的接口上不需要选举。路由器接口的优先级 Priority 将影响接口在选举 DR 时所具有的资格。优先级为 0 的路由器不会被选举为 DR 或 BDR。网段中的 DR 并不一定是 priority 最大的路由器；同理， BDR 也并不一定就是 priority 第

二大的路由器。若 DR、BDR 已经选择完毕，即使有一台 Priority 值更大的路由器加入，它也不会成为该网段中的 DR。 DR 是指某个网段中概念，是针对路由器的接口而言的。某台路由器在一个接口上可能是 DR，在另一个接口上可能是 BDR，或者是 DROther。两台 DROther 路由器之间不进行路由信息的交换，但仍旧互相发送 HELLO 报文。他们之间的邻居状态机停留在 2-Way 状态。在广播的网络上必须存在 DR 才能够正常工作，但 BDR 不是必需的。

3.4 LSA 的生成

本项目实现的是在同一个区域的 OSPF，即所有的 Router 都处于同一个 area 当中。所以只需要有一类和二类 LSA 一类为 Router LSA，每个路由器都会产生，二类 LSA 为 Network LSA，为 DR 周期性地产生，用来描述这个网段内的连接的路由器的 Router ID。生成 LSA 的时机：

- 路由器自己生成的 LSA，其 LS 时限达到 LSRefreshTime。这时，生成 LSA 的新实例，即使其 LSA 体的内容（除了 LSA 头）完全相同。这将保证周期性的生成所有的 LSA，这种周期性的 LSA 刷新增强了连接状态算法的强壮性。仅仅描述不可到达目标的 LSA 不会被重新刷新，而将被从路由域中废止。当 LSA 所描述的内容改变时，生成新的 LSA。然而，不能在时间 MinLSInterval 内连续生成同一 LSA 的两个实例。这意味着可能会延迟至多 MinLSInterval 秒以生成新的实例。下面的事件会导致 LSA 内容的改变。当且仅当内容有所不同时，才会生成 LSA 的新实例。
- 当接口状态改变，这可能需要生成一个 Router-LSA 的新实例。
- 当所接入网络的 DR 改变时，生成一个新的 Router-LSA。此外，如果路由器新成为 DR 的话，生成一个新的 Network-LSA；如果路由器不再成为 DR，其原来为该网络生成的任何 Network-LSA 都应当被从路由域中废止。
- 其邻居路由器的状态达到 FULL 状态、或不再是 FULL 状态，都将导致生成 Router-LSA 的新实例。此外，如果路由器是该网络的 DR 的话，还需要生成一个新的 Network-LSA。

3.5 LSA 的传播

传播时，分为以下几种情况：

- 在两个处于邻接状态的路由器之间交换 LSA
- 在 LSDB 发生变化时进行 flooding（泛洪）
- DR 根据当前网段的情况生成新的 2 类 LSA 并泛洪

3.6 区域内路由的计算

路由表的计算在项目中设计为周期性地更新，根据协议思想，我们需要首先计算出 SPF（最短路径树），然后再由 SPF 的计算结果得出路由表。

SPF 算法也被称为 Dijkstra 算法，是 OSPF 路由协议的基础。SPF 算法将每一个路由器作为根（Root）来计算到每一个目的地路由器之间的距离，每一个路由器根据一个统一的数据库会计算出路由域的拓扑结构图，该结构图类似于一棵树，在 SPF 算法中，被称为最短路径树。

3.7 路由转发功能的实现

这里实现路由转发功能的方式比较简单，直接通过 ioctl 的 SIOCADDRT 选项，实现系统调用，更改路由表，从而交由系统实现路由转发的功能。

第四章 OSPF 报文格式分析

4.1 IP 报文结构

数据结构定义：

```
struct ip_pkt
{
    u_char vhl;         /* version << 4 | header length >> 2 */
    u_char tos;         /* type of service */
    u_short len;        /* total length */
    u_short id;         /* identification */
    u_short offset;         /* fragment offset field */
    u_char ttl;         /* time to live */
    u_char pro;         /* protocol */
    u_short sum;            /* checksum */
    struct in_addr src,dst; /* source and dest address */
};
```

4.2 OSPF 报文格式

一共有 5 种不同的 OSPF 报文格式。所有的 OSPF 报文都以 24Byte 头部开始。这里首先描述报文头部，然后描述各种类型。

所有的 OSPF（除了 OSPF HELLO 报文）都处理 LSA 列表。例如，LSU 报文实现在 OSPF 路由域中 Flooding LSA。LSA 是整个系统的数据流的核心，生成的由 LSA 链接而成的 LSDB 是整个 OSPF 协议的精髓所在。所以，必须理解 LSA 的格式才能分析 OSPF 报文。

4.3 OSPF 报文头

数据结构定义：

```
/* OSPF header */
struct crypt {
    u_int16_t       dummy;
    u_int8_t    keyid;
    u_int8_t    len;
    u_int32_t       seq_num;
};
struct ospf_hdr {
    u_int8_t    version;
    u_int8_t    type;
    u_int16_t       len;
    u_int32_t       rtr_id;
    u_int32_t       area_id;
    u_int16_t       chksum;
    u_int16_t       auth_type;
    union {
        char    simple[MAX_SIMPLE_AUTH_LEN];
        struct crypt   crypt;
    } auth_key;
};
```

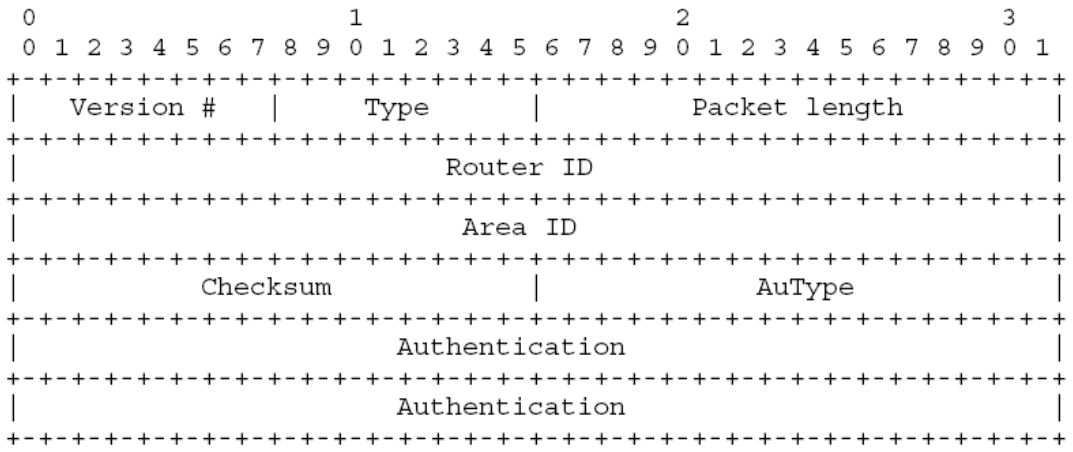


图 4. 1 OSPF 报文头图示

Version

OSPF 的版本号。对于OSPFv2 来说，其值为2。

Type

OSPF 报文的类型。数值从1到5，分别对应Hello 报文、DD 报文、LSR 报文、LSU 报文和LSAck 报文。

Packet length

OSPF 报文的总长度，包括报文头在内，单位为字节。

Router ID

报文起源的Router ID。

Area ID

一个32 位的数，标识报文属于哪个区域，所有OSPF 报文只属于单个区域，且只有一跳。当报文在虚链接上承载时，会打上骨干区域0.0.0.0 的标签。

Checksum

包的整个内容的校验，从OSPF 报文头部开始，但是除了64 位的认证字段。

AuType

认证类型包括四种：0（无需认证），1（明文认证），2（密文认证）和其他类型（IANA 保留）。当不需要认证时，只是通过Checksun 检验数据的完整性；当使用明文认证时，64 位的认证字段被设置成64 位的明文密码；当使用密文认证时，对于每一个OSPF 报文，共享密钥都会产生一个“消息位”加在OSPF 报文的后面，由于在网络上从来不以明文的方式发送密钥，所以提高了网络安全性。

Authentication

其数值根据验证类型而定。当验证类型为0 时未作定义，为1 时此字段为密码信息，类型为 2 时此字段包括 Key ID、MD5 验证数据长度和序列号的信息。

4.4 Hello 报文（Hello Packet）

数据结构定义：

```
/* Hello header (type 1) */
struct hello_hdr {
    u_int32_t       mask;
    u_int16_t       hello_interval;
    u_int8_t    opts;
    u_int8_t    rtr_priority;
    u_int32_t       rtr_dead_interval;
    u_int32_t       d_rtr;
    u_int32_t       bd_rtr;
};
```

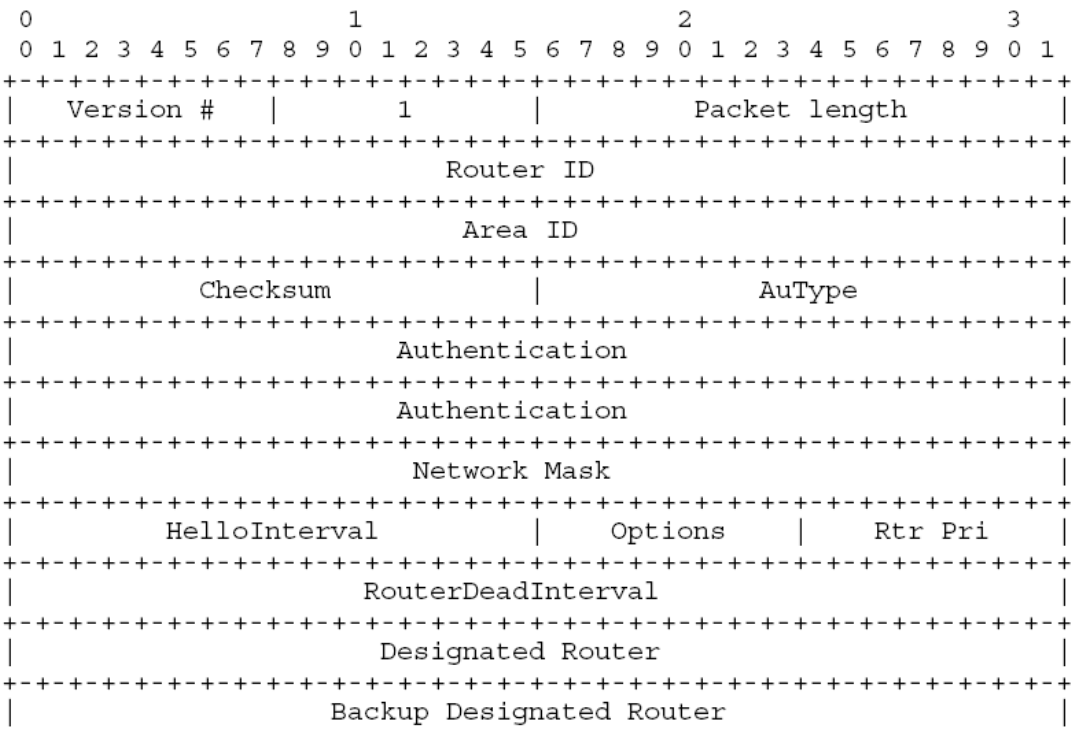


图 4. 2 OSPF Hello 报文图示

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：

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。

4.5 DD 报文（Database Description Packet）

数据结构定义：

```
/* Database Description header (type 2) */
struct db_dscrp_hdr {
    u_int16_t          iface_mtu;
```

```
    u_int8_t      opts;
    u_int8_t      bits;
    u_int32_t         dd_seq_num;
};
```

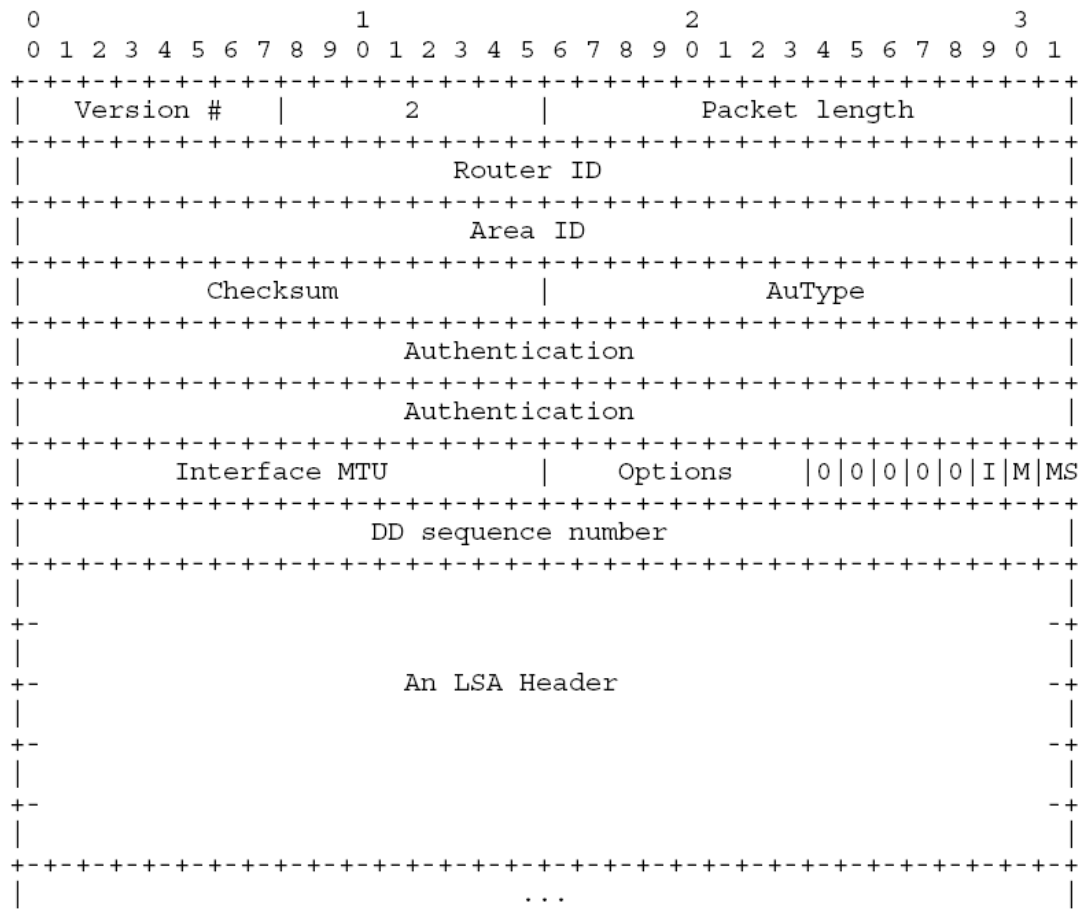


图 4. 3 OSPF DD 报文图示

路由信息(连接状态传送报文)只在形成邻接关系的路由器间传递。

首先，它们之间互发 DD（database description）报文，告之对方自己所拥有的路由信息，内容包括LSDB 中每一条LSA 的摘要（摘要是指LSA 的HEAD，通过该HEAD可以唯一标识一条LSA）。

这样做是为了减少路由器之间传递信息的量，因为LSA 的HEAD 只占一条LSA 的整个数据量的一小部分，根据HEAD，对端路由器就可以判断出是否已经有了这条LSA。

DD 报文有两种，一种是空DD 报文，用来确定Master/Slave 关系（避免DD 报文的无序发送）。确定Master/Slave 关系后，才发送有路由信息的DD 报文。

收到有路由信息的 DD 报文后，比较自己的数据库，发现对方的数据库中有自己需要的数据，则向对方发送LSR（Link State Request）报文，请求对方给自己发送数据。

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。

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。

4.6 LSR 报文（Link State Requirement Packet）

数据结构定义：

```
/*    Link State Request header (type 3) */
struct ls_req_hdr {
    u_int32_t          type;
    u_int32_t          ls_id;
    u_int32_t          adv_rtr;
};
```

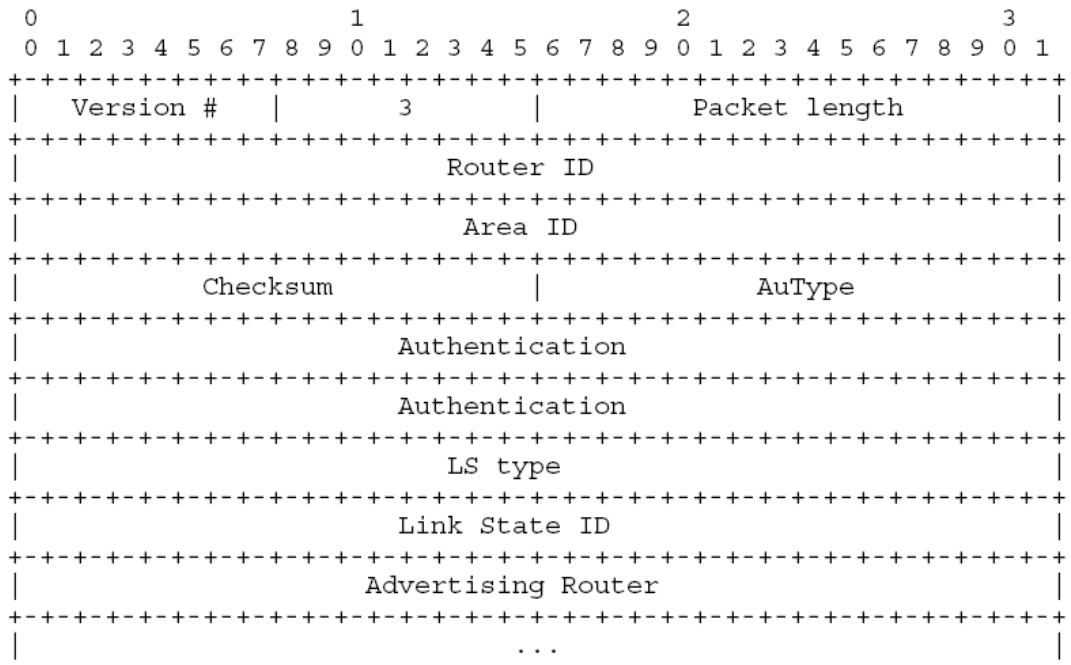


图 4. 4 OSPF LSR 报文图示

两台路由器互相交换过 DD 报文之后，知道对端的路由器有哪些LSA 是本地的LSDB 所缺少的或是对端更新的LSA，这时需要发送LSR 报文向对方请求所需的LSA。内容包括所需要的LSA 的摘要。

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。

4.7 LSU 报文（Link State Update Packet）

数据结构定义：

```
/* Link State Update header (type 4) */
struct ls_upd_hdr {
    u_int32_t          num_lsa;
     struct lsa lsu_lsa[1];
```

```
};
```

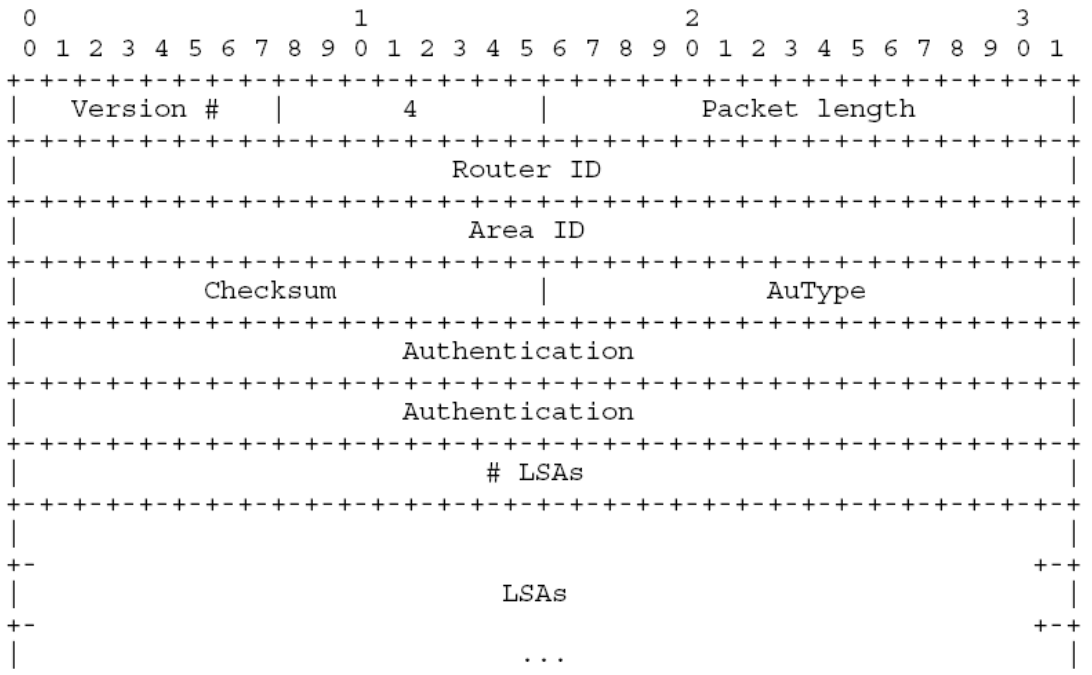


图 4. 5 OSPF LSU 报文图示

LSA 数量/# LSA：

该更新包中包含的 LSA 数量。

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。

4.8 LSAck 报文（Link State Acknowledgment Packet）

数据结构定义：

```
/* Link State Acknowledgment header (type 5) */
struct ls_ack_hdr {
    struct lsa_hdr lsa_lshdr[1];
};
```

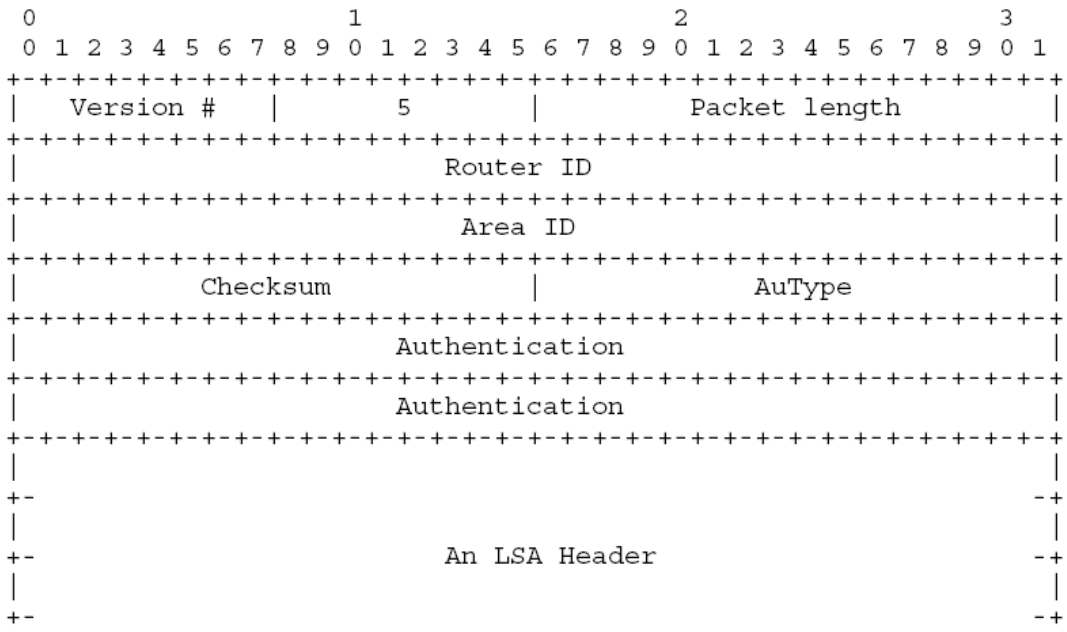


图 4. 6 OSPF LSAck 报文图示

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。

由于没有使用可靠的 TCP 协议，但是 OSPF 包又要求可靠的传输，所以就有了 LSAck 包。它用来对接收到的 LSU 报文进行确认。内容是需要确认的 LSA 的 HEAD（一个报文可对多个 LSA 进行确认）。

DD 报文、LSR 报文、LSU 报文发出后，在没有得到应有的对方相应的 LSR、LSU、LSAck 报文时，会重发。（例外：对 DD 报文若收到后发现没有必要产生连接状态请求报文，则不发连接状态请求报文。）同步后数据改变，则只向形成 Adjacency 关系的路由器发 LSU 报文。

4.9 LSA 头部

数据结构定义：

```
struct lsa_hdr {
    u_int16_t       age;
    u_int8_t        opts;
```

```
    u_int8_t      type;
    u_int32_t         ls_id;
    u_int32_t         adv_rtr;
    u_int32_t         seq_num;
    u_int16_t         ls_chksum;
    u_int16_t         len;
};
```

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。

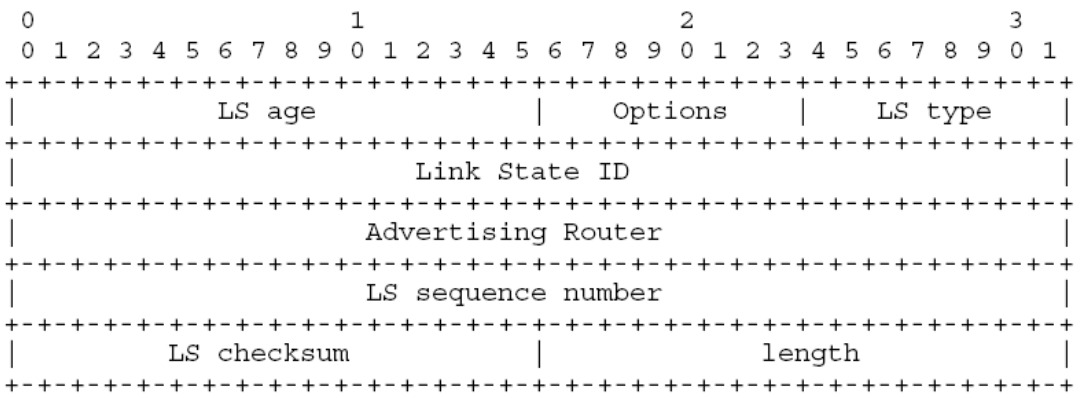


图 4. 7 OSPF LSA 头部图示

LS 类型/LS type：

LSA 的类型。各种类型的 LSA 使用不同的格式。本备忘录定义的 LSA 类型如下。

LS 类型	描述
1	Router-LSA
2	Network-LSA
3	Summary-LSA （IP 网络）
4	Summary-LSA （ASBR）
5	AS-external-LSA

LS 标识/Link State ID：

该域描述由 LSA 所描述的网络部件。具体的内容取决于 LSA 中的 LS 类型。例如，Network-LSA 中的 LS 标识为网络上 DR 的接口 IP 地址（从中可以计算出网络的 IP 地址）。

宣告路由器/Advertising Router：

生成该 LSA 的路由器标识。例如，Network-LSA 中该域等于网络上 DR 的路由器标识。

LS 序号/LS sequence number：

用于判定旧的或重复的 LSA。连续的 LSA 实例使用连续的 LS 序号。

LS 校验和/LS checksum：

整个 LSA 的 Fletcher 校验和，包括除 LSA 时限域外的 LSA 头部。更多细节见第 12.1.7 节。

长度/length：

LSA 的字节长度。包含 20 字节的 LSA 头部。

4.10 Router LSA

数据结构定义：

```
struct lsa_rtr {
    u_int8_t      flags;
    u_int8_t      dummy;
    u_int16_t         nlinks;
};

struct lsa_rtr_link {
    u_int32_t         id;
    u_int32_t         data;
    u_int8_t      type;
```

```
    u_int8_t       num_tos;
    u_int16_t          metric;
};
```

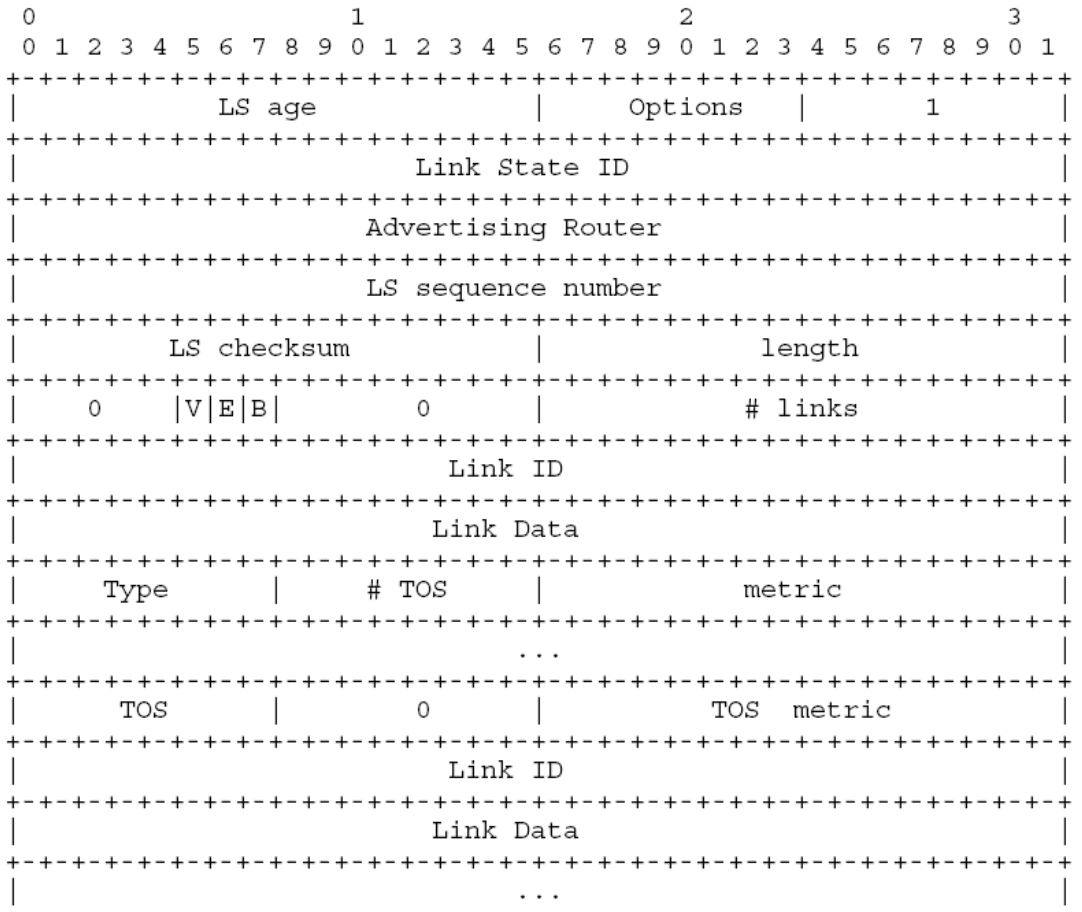


图 4. 8 OSPF Router LSA 图示

连接数/# links：

该 LSA 所描述的路由器连接数量。必须是该区域路由器连接（接口）的总和。

下面的各域用于描述各个路由器连接（接口）。每个接口都有类型（见下面的类型域），类型域说明了所描述连接的种类。连接可能为传输网络、到达另一路由器或存根网络。其他描述连接的域都取决于连接类型。例如，每个连接有相关的 32 为连接数据域。对存根区域的连接来说表示网络的 IP 地址掩码，而对其他类型来说，表示的是路由器接口 IP 地址。

类型/Type：

路由器连接的基本描述，可为以下值。注意，主机路径被看作网络掩码为 0xffffffff

的存根网络，

类型	描述
1	点对点连接到另一路由器
2	连接到传输网络
3	连接到存根网络
4	虚拟通道

连接标识/Link ID：

类型	LS 标识
1	邻居的路由器标识
2	DR 的 IP 接口地址
3	IP 网络/子网号
4	邻居的路由器标识

连接数据/Link Data：

其值同样取决于连接的类型。对于存根网络连接，连接数据说明的是网络的 IP 地址；对于其他类型的连接，说明的是路由器接口的 IP 地址。在计算路由表过程中，计算下一跳的 IP 地址时需要使用这些信息。

TOS 数/# TOS：

该连接不同 TOS 的数量，不包括所需要的连接距离。例如，如果没有给出额外的 TOS 距离，该域被设定为 0。

距离值/metric：

路由器连接的距离。

4.11 Network-LSA

数据结构定义：

```
struct lsa_net {
    u_int32_t          mask;
```

```
    u_int32_t          att_rtr[1];
};
struct lsa_net_link {
    u_int32_t          att_rtr;
};
```

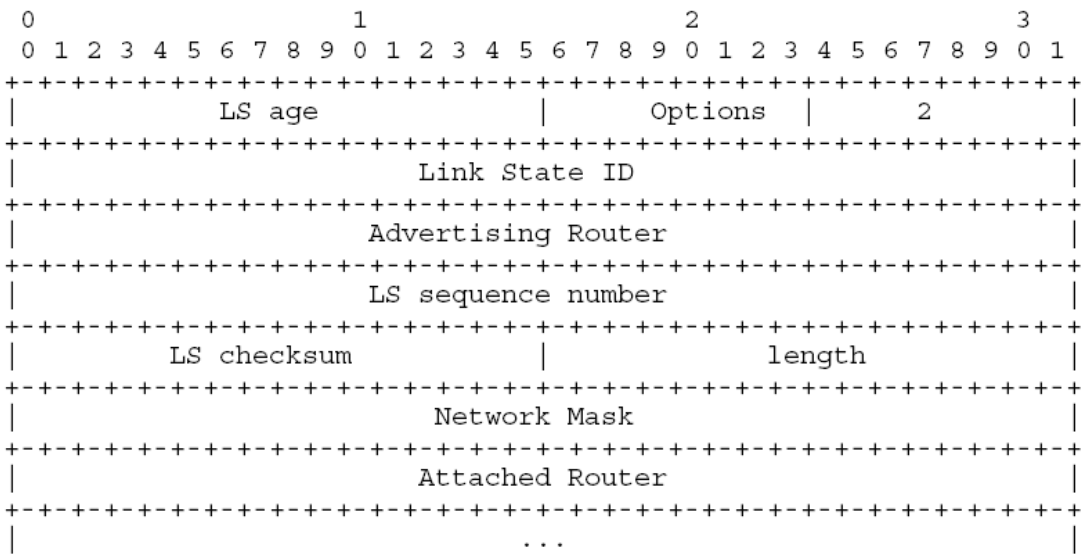


图 4. 9 OSPF Network LSA 图示

网络掩码/Network Mask：

该网络的 IP 地址掩码。例如 A 类网络的掩码为 0xff000000。

接入路由器/Attached Router：

接入该网络的各个路由器的路由器标识。注意，只有与 DR 达到完全邻接的路由器才被列出。DR 自身也被包含在列表中。列表中的路由器数量可以从 LSA 头部中的长度域中计算而得到。

4.12 LSA 定义合成

数据结构定义：

```
struct lsa {
    struct lsa_hdr          hdr;
```

```
    union {
        struct lsa_rtr      rtr;
        struct lsa_net          net;
        struct lsa_sum          sum;
        struct lsa_asext    asext;
    }data;
};
```

4.13 常量定义

```
/* misc */
#define OSPF_VERSION      2
#define IPPROTO_OSPF      89
#define AllSPFRouters         "224.0.0.5"
#define AllDRouters       "224.0.0.6"

#define DEFAULT_METRIC       10
#define DEFAULT_REDIST_METRIC     100
#define MIN_METRIC        1
#define MAX_METRIC        65535   /* sum & as-ext lsa use 24bit metrics */

#define DEFAULT_PRIORITY    1
#define MIN_PRIORITY      0
#define MAX_PRIORITY          255

#define DEFAULT_HELLO_INTERVAL   10
#define MIN_HELLO_INTERVAL     1
#define MAX_HELLO_INTERVAL    65535
```

```
/* msec */
#define DEFAULT_FAST_INTERVAL       333
#define MIN_FAST_INTERVAL   50
#define MAX_FAST_INTERVAL 333

#define DEFAULT_RTR_DEAD_TIME     40
#define FAST_RTR_DEAD_TIME      1
#define MIN_RTR_DEAD_TIME 2
#define MAX_RTR_DEAD_TIME       2147483647

#define DEFAULT_RXMT_INTERVAL     5
#define MIN_RXMT_INTERVAL 5
#define MAX_RXMT_INTERVAL       3600

#define DEFAULT_TRANSMIT_DELAY  1
#define MIN_TRANSMIT_DELAY     1
#define MAX_TRANSMIT_DELAY   3600

#define DEFAULT_ADJ_TMOUT       60

#define DEFAULT_NBR_TMOUT       86400    /* 24 hours */

/* msec */
#define DEFAULT_SPF_DELAY  1000
#define MIN_SPF_DELAY           10
#define MAX_SPF_DELAY         10000

/* msec */
#define DEFAULT_SPF_HOLDTIME 5000
```

```
#define MIN_SPF_HOLDTIME    10
#define MAX_SPF_HOLDTIME  5000

/* msec */
#define KR_RELOAD_TIMER          250
#define KR_RELOAD_HOLD_TIMER     5000

#define MIN_MD_ID           0
#define MAX_MD_ID           255

#define MAX_SIMPLE_AUTH_LEN 8

/* OSPF compatibility flags */
#define OSPF_OPTION_MT       0x01
#define OSPF_OPTION_E        0x02
#define OSPF_OPTION_MC       0x04
#define OSPF_OPTION_NP       0x08
#define OSPF_OPTION_EA       0x10
#define OSPF_OPTION_DC       0x20
#define OSPF_OPTION_O        0x40    /* only on DD options */
#define OSPF_OPTION_DN       0x80    /* only on LSA options */

/* OSPF packet types */
#define PACKET_TYPE_HELLO 1
#define PACKET_TYPE_DD       2
#define PACKET_TYPE_LS_REQUEST   3
#define PACKET_TYPE_LS_UPDATE    4
#define PACKET_TYPE_LS_ACK    5
```

```
/* OSPF auth types */
#define  AUTH_TYPE_NONE          0
#define AUTH_TYPE_SIMPLE    1
#define  AUTH_TYPE_CRYPT         2

#define MIN_AUTHTYPE         0
#define MAX_AUTHTYPE         2

/* LSA */
#define LS_REFRESH_TIME      1800
#define MIN_LS_INTERVAL      5
#define MIN_LS_ARRIVAL       1
#define DEFAULT_AGE      0
#define MAX_AGE              3600
#define CHECK_AGE            300
#define MAX_AGE_DIFF         900
#define LS_INFINITY          0xffffff
#define RESV_SEQ_NUM         0x80000000 /* reserved and "unused" */
#define INIT_SEQ_NUM     0x80000001
#define MAX_SEQ_NUM          0x7fffffff
```

第五章 程序设计与代码实现

5.1 程序结构设计

程序的模块设计如下：

图 5. 1 程序模块设计

程序所有文件的相应功能如下：

表 5. 1 程序结构设计

文件名称	文件功能
init.cpp	初始化相关配置
hello.cpp	发送、接收 HELLO 报文
dd.cpp	发送、接收 DD 报文
lsr.cpp	发送、接受 LSR 报文
lsu.cpp	发送、接收 LSU 报文
lsack.cpp	发送、接收 LSACK 报文
pkt.cpp	发送、接收所有 IP packet 报文
errlog.cpp	错误 LOG 定义

out.cpp	输出 SPFtree、路由表、LSDB 等相关信息
pkt.h	OSPF 报文格式定义
global.h	全局变量与相关接口定义
precv.cpp	收包相关函数
precv.h	收报相关函数
stdnet.cpp	发包相关函数
stdnet.h	发包相关函数

5.2 全局数据结构设计

5.2.1 错误类型与错误处理函数

```
typedef enum {
    INITERROR,                  //初始化错误
    SENDPKTERROR,               //发送错误
        RECVPKTERROR            //接收错误
} error_type_t;                 //定义的枚举错误类型
void errorprint(error_type_t errNum, char errstring[]);        //错误打印函数
```

5.2.2 邻居结构

```
struct nbr_struct
{
    struct inf_struct * infNow;     //当前邻居关系对应的接口信息
    u_int32_t seqNum;               //DD 报文用到，用于记录对方的 sequence number
    u_int32_t startSeqNum;          //DD 报文用到，用于记录对方的 sequence number
    u_int32_t router_id;        //邻居路由的 Router ID
    u_int32_t area_id;              //邻居的 Area ID
   u_int32_t inf_ip;            //邻居接口的 IP
```

```
    u_int32_t inf_mask;       //邻居接口的 MASK
    int exchangeNum;          //用于记录 exchange 的数目，用于 DD 报文的重发与 Exchange 状态的记录
    int master;               //用于记录邻居是否是 Master（在 DD 报文中用到）
    long lastHelloTime;       //用于记录邻居最后一次的 hello 报文时间，在 KEEP 进程中用到，若时间差超过 DeadInterval（这里是 40s），则判为邻居已经 Dead
    bool xchg;                //用于记录邻居状态，是否已经到 Exchange 状态
};
```

5.2.3 LSDB 结构

```
struct lsdb_struct
{
    u_int32_t seq;            //LSA 的 sequence number
    u_int32_t bornRtId;       //产生 LSA 信息的路由的 Router ID
    u_int32_t ls_id;          //LSA 的 ID
    u_int16_t length;         //LSA 的长度
    int ls_type;              //LSA 的类型（ROUTER/NETWORK/…）
   long bornTime;             //LSA 产生的时间
   //下面是具体的 LSA 信息
    …..
};
```

deque<struct lsdb_struct *> lsaDequeInArea;

双端队列 Deque 说明

这里我用了 C++ Deque 的结构来实现 LSDB 的存取。其实 C++里面有三种 STL 模板来实现链式结构：vector，list 和 deque。上网查询结果如下：

vector 和 built-in 数组类似，它拥有一段连续的内存空间，并且起始地址不变，

因此 它能非常好的支持随机存取，即[]操作符，但由于它的内存空间是连续的，所以在中间进行插入和删除会造成内存块的拷贝，另外，当该数组后的内存空间不够时，需要重新申请一块足够大的内存并进行内存的拷贝。这些都大大影响了 vector 的效率。

list 就是数据结构中的双向链表(根据 sgi stl 源代码)，因此它的内存空间可以是不连续的，通过指针来进行数据的访问，这个特点使得它的随机存取变的非常没有效率，因此它没有提供[]操作符的重载。但由于链表的特点，它可以以很好的效率支持任意地方的删除和插入。

deque 是一个 double-ended queue，它的具体实现不太清楚，但知道它具有以下两个特点：它支持[]操作符，也就是支持随机存取，并且和 vector 的效率相差无几，它支持在两端的操作：push_back,push_front,pop_back,pop_front 等，并且在两端操作上与 list 的效率也差不多。

因此在实际使用时，如何选择这三个容器中哪一个，应根据你的需要而定，一般应遵循下面的原则：

1、如果你需要高效的随机存取，而不在乎插入和删除的效率，使用 vector

2、如果你需要大量的插入和删除，而不关心随机存取，则应使用 list

3、如果你需要随机存取，而且关心两端数据的插入和删除，则应使用 deque。

由于 LSDB 有显著的随机存取的特点（由于路由可能突然加入或退出，或者 LSU 信息到达的不确定性），所以我选用双端队列 Deque 实现 LSDB 的存取）。

5.2.4 接口信息

```
struct infDeque
{
    struct infDeque * next;                 //简单的链式结构存取接口信息
    struct inf_struct * inf;                //具体的接口信息
};
struct inf_struct
{
```

```
    deque<struct nbr_struct *> nbrDeque; //接口所对应的邻居队列
    int idx;                    //接口编号 ID，这里我取的是 11/12 之类的信息，
而实际可以是 E0/0，E0/1 这样的编号
    u_int32_t ip;               //接口对应的 IP
    u_int32_t mask;             //接口对应的 MASK
    int type;                   //接口类型，这里取 BRDCAST 类型
    u_int32_t area_id;          //接口对应的 area 的 ID
    u_int32_t dr;               //接口的 DR
    u_int32_t bdr;              //接口的 BDR
    char name[10];              //接口的网卡名（eth0/eth1/lo 等等）
    bool drFlag;                //标志接口处是否有 DR，方便判断是否要进行 DR
选举
    struct area * areain;       //接口所处的 area
    int cost;                   //接口线路的 cost
};
```

5.2.5 路由信息

```
struct Rt_struct
{
    u_int32_t router_id;        //路由对应的 Router ID
    int priority;               //路由的优先级，初始化时所有路由优先级都设为 1。
设置这个结构是为了方便以后的扩展
};
```

5.2.6 区域信息

```
struct area
{
```

```
    struct infDeque * infDequeInArea;                    //所处区域的接口队列
    deque<struct lsdb_struct *> lsaDequeInArea;  //区域的 LSA 队列（LSDB 数据库）
    u_int32_t areaId;                                    //区域对应的 Area ID
};
```

5.2.7 路由表结构

```
struct routingtable
{
    struct in_addr destination;
    struct in_addr mask;
    struct in_addr nexthop;
    char interface_num[5];
};
```

5.3 全局变量与全局函数设计

5.3.1 全局变量设计

表 5. 2 全局变量设计

全局变量	功能
char outlsdbfile[100];	打印的 LSDB 文件名
char outfile[100];	输出 SPF VERTEX、SPFtree、路由表信息的文件名
uint32_t rtrRtrCnnt[RT_MAX][RT_MAX];	路由连接信息，rtrRtrCnnt[a][b] 表示从 Router ID 为 a 的路由到 Router ID 为 b 的直连路由下一跳的 IP 为 rtrRtrCnnt[a][b]

char outRTtablefile[100];	输出的路由表信息的文件名
deque<struct ospf_area *> areaDeque;	区域 area 链表
deque<struct inf_struct *> infDeque;	接口链表
struct Rt_struct nowRt;	当前的路由信息
map <int,int> cnctMap;	路由连接的映射表
char Rn[10];	路由名（R1/R2/R3/R4/…）

5.3.2 全局函数设计

表 5. 3 全局函数设计

全局函数名	功能
hello.cpp	
void * sendHelloPkt(void * nowInf);	发送 Hello 报文
void * sendHelloSeen(void * nowInf,u_int32_t seenNbr);	发送 Hello Seen 报文
void * keep(void * none);	刷新 Dead Interval，维持邻居关系的处理进程
void recv_hello(const u_char * packet);	接收 Hello 报文并处理
void * DRdetect(void * none);	判断 DR 是否存在的处理线程
dd.cpp	
void * send_db_description(void * nowInf, struct nbr_struct * nbrStct, struct std_lsa_hdr * pLsaSend, bool initBool,bool moreBool,bool msBool);	发送 DD 报文
void recv_db_description(const u_char * packet);	接收 DD 报文并处理

lsr.cpp	
void * send_ls_req(void * nowInf, struct nbr_struct * nbrStct, uint lsa_typeF, uint lsa_idF, uint32_t lsa_advs_addrF);	发送 LSR 报文
void recv_ls_req(const u_char * packet);	接收 LSR 报文并处理
lsu.cpp	
void * send_ls_update(void * nowInf, u_int32_t sendToIp, struct lsdbDequeStruct * lsaSent ,struct std_lsa_hdr * pLsaSend);	发送 LSU 报文
void recv_ls_update(const u_char * packet);	接收 LSU 报文并处理
lsack.cpp	
void * send_ls_ack(void * nowInf, struct std_lsa_hdr * pLsaAck);	发送 LSACK 报文
void recv_ls_ack(const u_char * packet);	接收 LSACK 报文并处理
pkt.cpp	
void * recvPkt(void * none);	接收 IP 报文的处理线程
void recv_packet(u_char *args,const struct pcap_pkthdr * header,const u_char * packet);接收 IP 报文并处理	
init.cpp	
void init();	初始化处理

out.cpp	
void * printLsdb(void * uselessF);	打印输出 SPF VERTEX、SPFtree、路由表信息
genrt.cpp	
int addRouteIterm(uint32_t dstIp, uint32_t mask, uint32_t nextHop, char* infName);	增加 Linux 系统路由表条目

5.4 主要流程描述

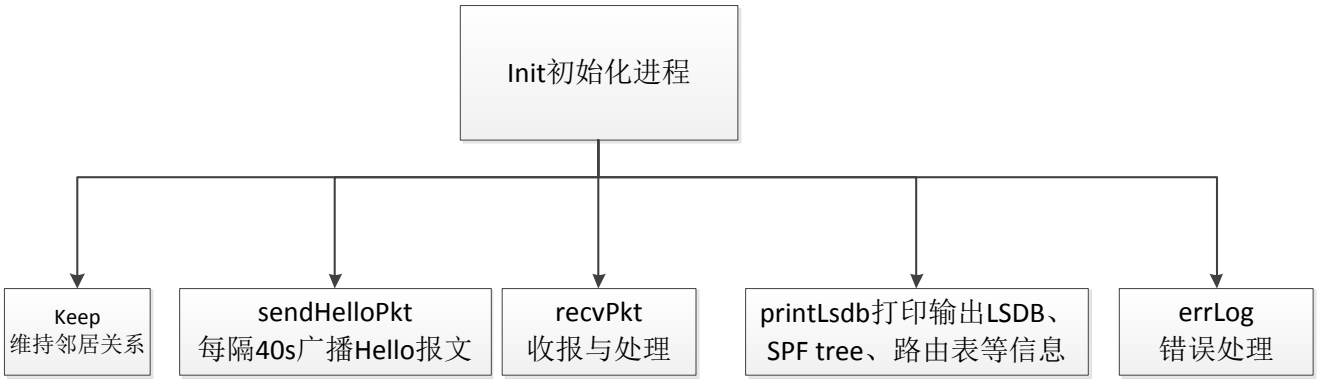


图 5. 2 主要流程描述

5.5 初始化

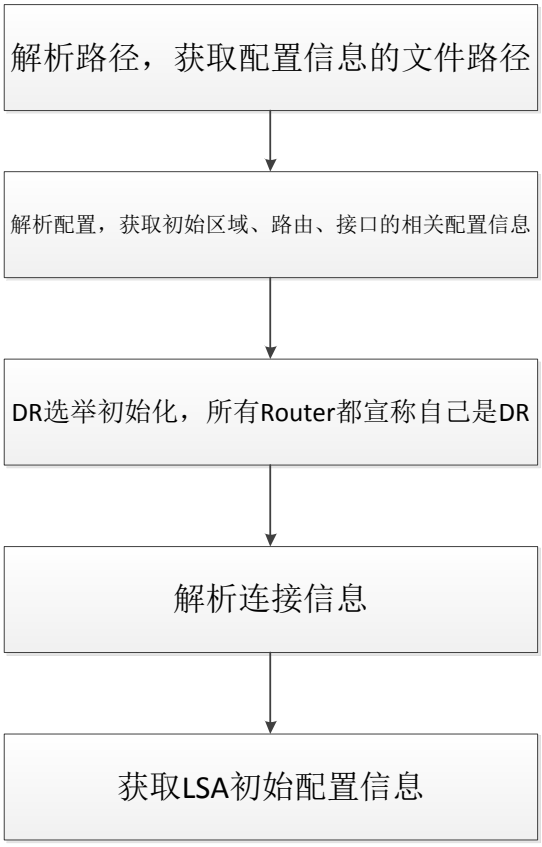


图 5. 3 初始化流程图

5.6 DR 选举

只有在广播和 NBMA 类型的接口上才会选举 DR，在 point-to-point 和 point-to-muiltipoint 类型的接口上不需要选举。

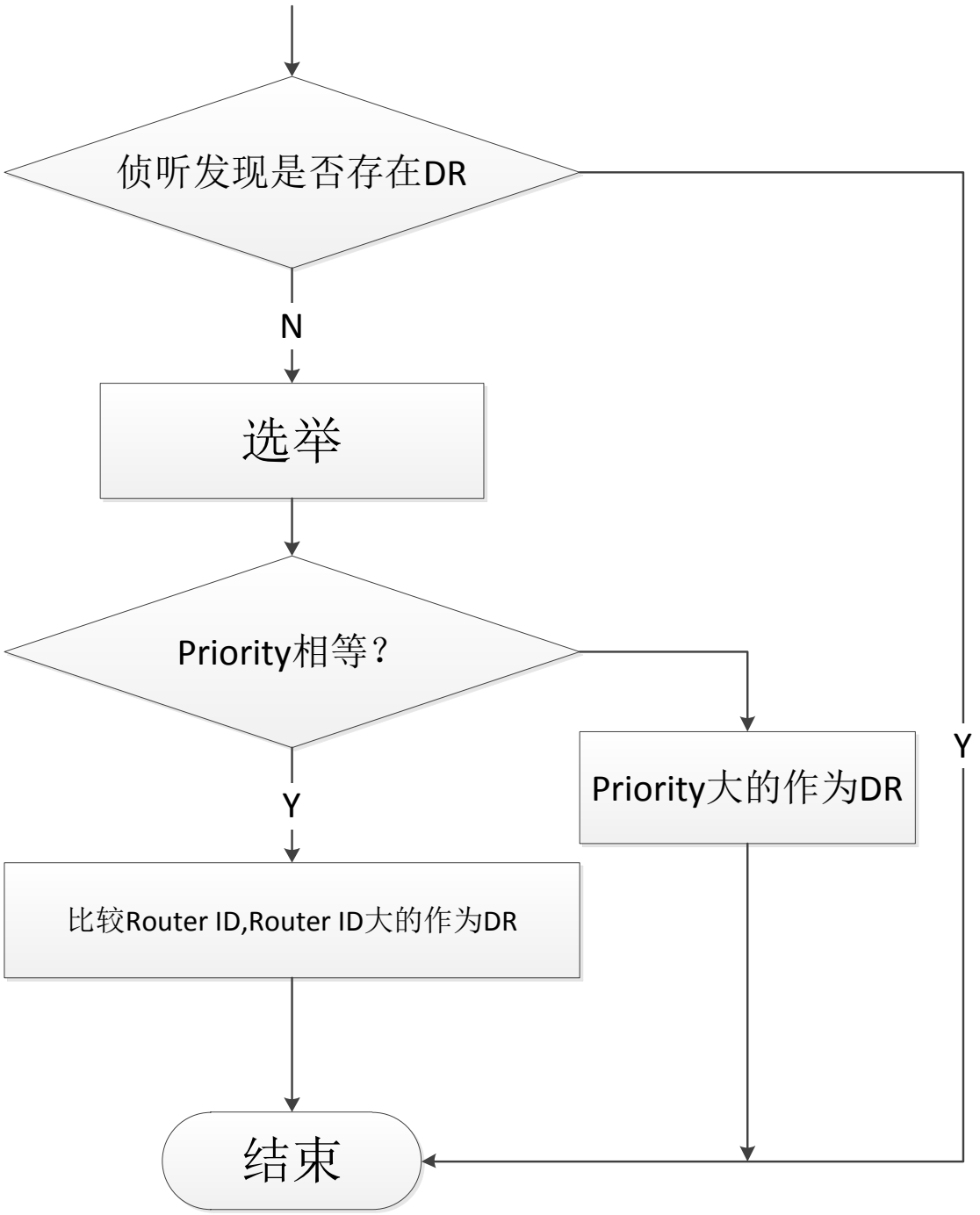


图 5. 4 DR 选举流程图

5.7 LSDB 的同步过程

LSDB 的同步过程如下：

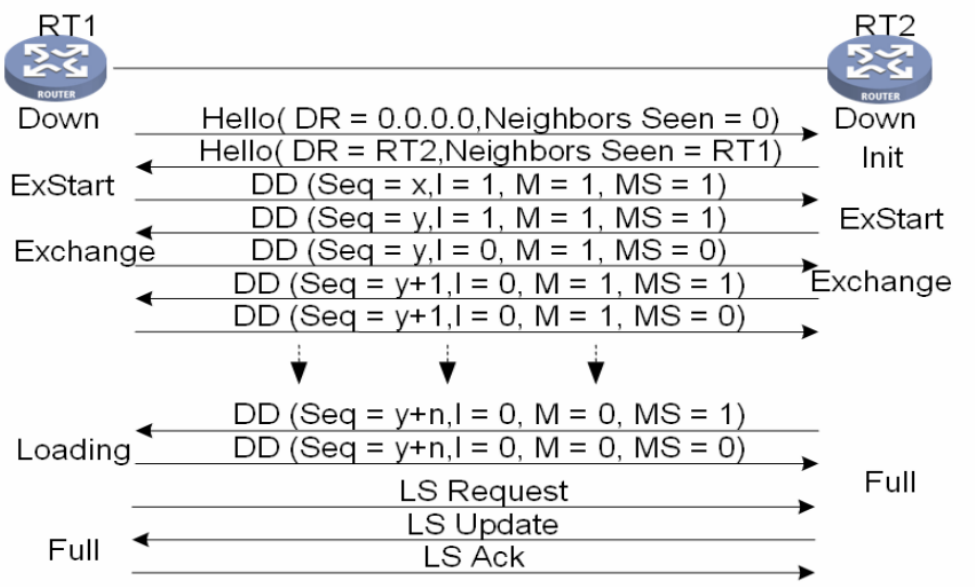


图 5. 5 LSDB 同步过程

5.8 DD 报文的处理

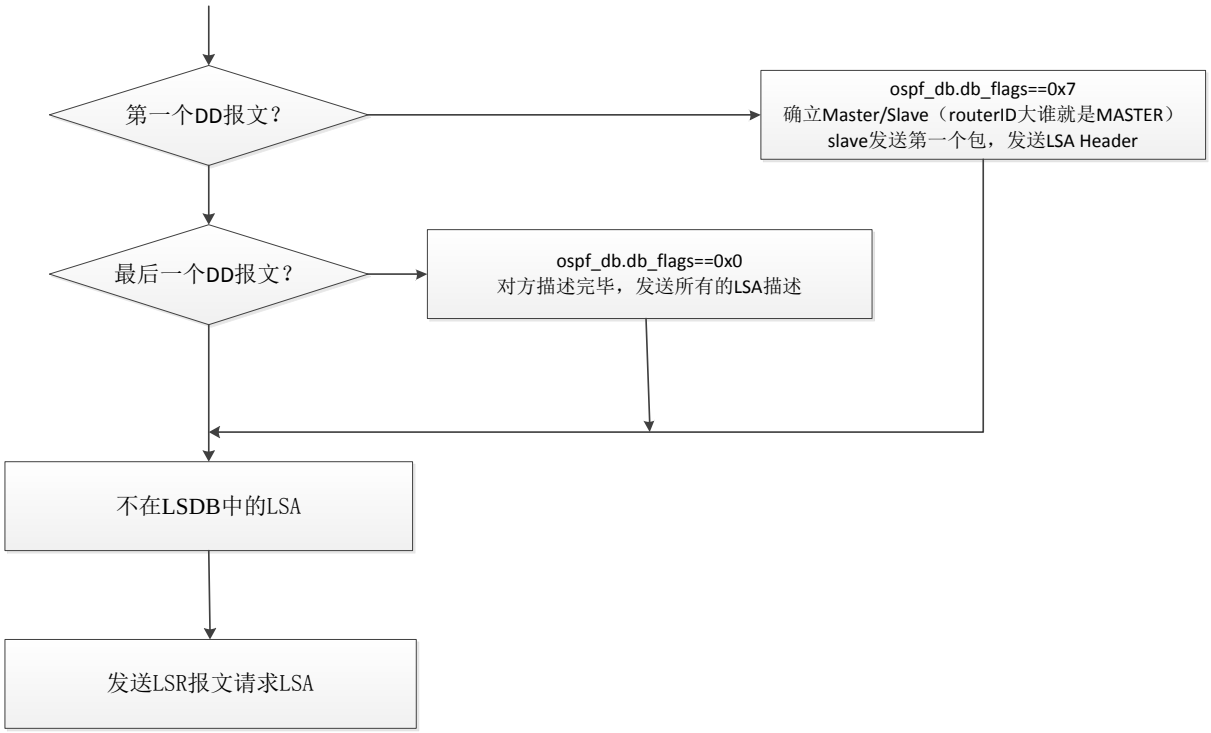


图 5. 6 DD 报文的处理流程图

可靠传输的保证的三个机制：

- Master/Slave
- 主/从关系协商出 SeqNum
- 传输先后交替通信
- SeqNum 递增
- 超时重传
- 如果超过一定时间没有回应则可能是发生了丢包
- 因为 IP 层没有提供可靠传输，所以需要自行进行重传

5.9 LSR 报文的处理

LSR 报文的处理较为简单，即收到请求后立即查找自己的 LSDB，找到对应的 LSA 数据，然后回送到目的路由器。

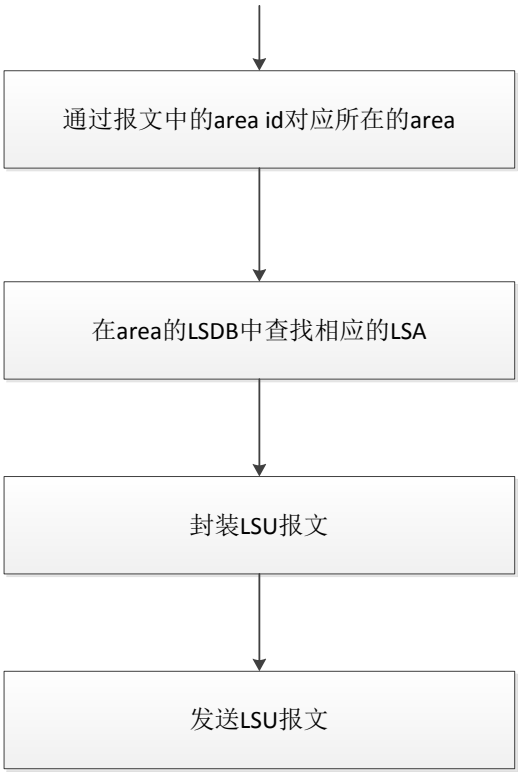


图 5. 7 LSR 报文处理的流程图

5.10 LSU 与 LSAck 报文的处理

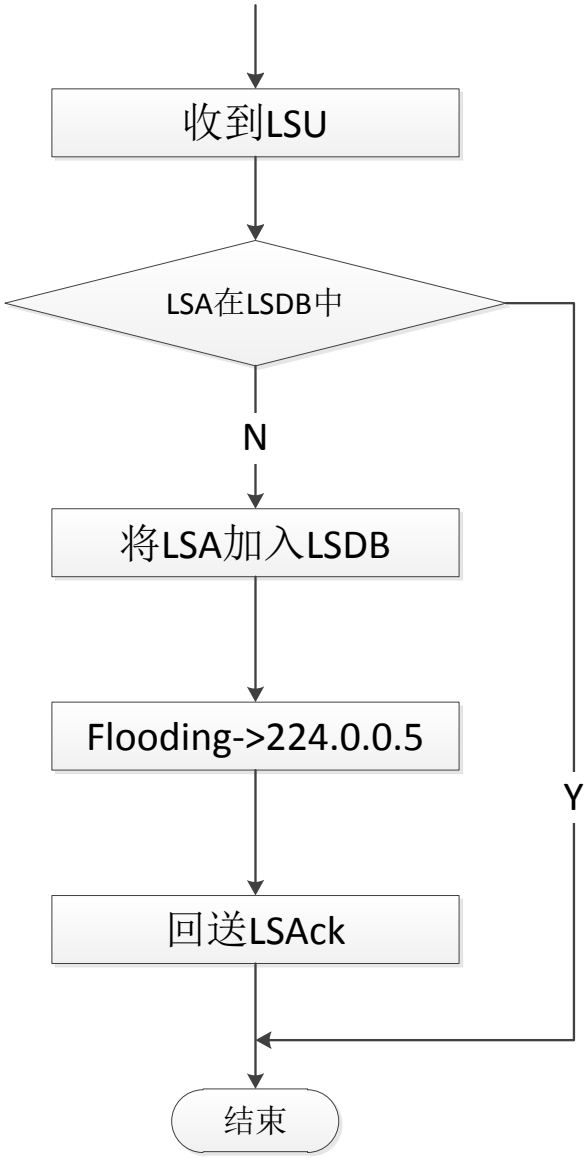


图 5. 8 LSU 与 LSAck 报文的处理流程图

5.11 LSA 生成时机

LSA 是整个系统的数据流的核心，生成的由 LSA 链接而成的 LSDB 是整个 OSPF 协议的精髓所在。

在本系统中的 LSA 生成时机如下（有 **3** 处）：

- 一类 LSA（Router LSA）：

- 初始化完成时，通过 LSA 初始化配置文件的读取生成系统无法自动生成的 LSA（Type：StubNet），在 DR 选举完成后与 DR 交换。
- DR 选举完成时，由所有路由各自产生产生描述直连路由信息的 LSA（Type：TransNet），产生后都与 DR 交换。

➢ 二类 LSA（Network LSA）：

- 生成 LSDB 要输出到文件前，由 DR 产生 Network LSA 描述网段内的链路状态，并在区域内传播（这里 Flooding）。

5.12 SPF 算法

这里给出的是程序的伪代码：

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                // Initializations
        dist[v] := infinity ;                  // Unknown distance function from source to v
        previous[v] := undefined ;             // Previous node in optimal path from source
    end for ;
    dist[source] := 0 ;                        // Distance from source to source
    Q := the set of all nodes in Graph ;
    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                      // The main loop
        u := vertex in Q with smallest dist[] ;
        if dist[u] = infinity:
            break ;                            // all remaining vertices are inaccessible from source
        end if ;
        remove u from Q ;
        for each neighbor v of u:              // where v has not yet been removed
```

```
from Q.
                alt := dist[u] + dist_between(u, v) ;
                if alt < dist[v]:                    // Relax (u,v,a)
                    dist[v] := alt ;
                    previous[v] := u ;
                    decrease-key v in Q;          // Reorder v in the Queue
                end if ;
            end for ;
        end while ;
        return dist[] ;
end Dijkstra.
```

5.13 路由转发功能的实现

这里采用的方法是直接通过 ioctl 的 SIOCADDRT 选项，实现系统调用，更改系统的路由表，从而交由系统实现路由转发的功能。

```
int addRouteIterm(uint32_t dstIp, uint32_t mask, uint32_t nextHop, char* infName)
{
    //struct routingtable *item = malloc(sizeof(struct routingtable));
    struct routingtable *item = new routingtable();
    item->destination.s_addr = dstIp;
    item->mask.s_addr = mask;
    item->nexthop.s_addr = nextHop;
    strcpy(item->interface_num,infName);//"eth0");
    printf("%s\n",item->interface_num);
    int sockfd;
    struct rtentry rm;
    int err;
```

```
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1)
    {
        printf("socket is -1\n");
        return -1;
    }
    memset(&rm, 0, sizeof(rm));

    (( struct sockaddr_in*)&rm.rt_dst)->sin_family = AF_INET;
    (( struct sockaddr_in*)&rm.rt_dst)->sin_addr.s_addr = (item->destination.s_addr
& item->mask.s_addr);
    (( struct sockaddr_in*)&rm.rt_dst)->sin_port = 0;
    printf("dest: %x\n", item->destination.s_addr & item->mask.s_addr);
    printf("%s\n",inet_ntostr( item->destination.s_addr & item->mask.s_addr));

    (( struct sockaddr_in*)&rm.rt_genmask)->sin_family = AF_INET;
    (( struct sockaddr_in*)&rm.rt_genmask)->sin_addr.s_addr =
(item->mask.s_addr);
    (( struct sockaddr_in*)&rm.rt_genmask)->sin_port = 0;
    printf("mask: %x\n", item->mask.s_addr);
    printf("%s\n",inet_ntoa(item->mask));

    (( struct sockaddr_in*)&rm.rt_gateway)->sin_family = AF_INET;
    (( struct sockaddr_in*)&rm.rt_gateway)->sin_addr.s_addr =
(item->nexthop.s_addr);
    (( struct sockaddr_in*)&rm.rt_gateway)->sin_port = 0;
    printf("gateway: %x\n", item->nexthop.s_addr);
    printf("%s\n",inet_ntoa(item->nexthop));
```

```
    rm.rt_dev = item->interface_num;


    rm.rt_flags = RTF_GATEWAY | RTF_UP;
    if ((err = ioctl(sockfd, SIOCADDRT, &rm)) < 0)
    {
        close(sockfd);
        perror("ioctl");
        perror("SIOCADDRT");
        printf("Add New Route failed, ret->%d\n", err);
        return -1;
    }
    close(sockfd);
    printf("Successfully Add New Route!\n");
    return 1;
}
```

第六章 实验设计与结果分析

6.1 实验目标

- 综合测试：
  - 测试 OSPF 协议实现的正确性
  - 验证 OSPF 协议规定格式数据的封装发送
  - 验证符合 OSPF 协议规定格式的数据包的接收与逻辑处理
  - 验证 Dead Interval 维护、邻居关系建立及报文交换过程
- 单独测试：
  - 与路由器等设备交互的正确性：实验 1：PC 与路由直接相连
  - 路由器交互过程的正确性（五类报文）：实验 2：两个路由交互验证
  - 验证生成的 LSDB、SPF 树、路由表的正确性：实验 3：四个路由交互验证
  - 验证路由表转发的正确性：实验 4：PC 与路由多台验证

6.2 实验环境

操作系统：Ubuntu 11.10（机器上直接运行）、Ubuntu9.10（Vmware 中运行）

编译器：g++ 4.4.3

调试环境：gdb7.1

集成开发环境：Netbeans IDE 7.1.2

抓包程序：Wireshark 1.2.7

6.3 实验 1：PC 与路由实验室组网测试

6.3.1 组网图

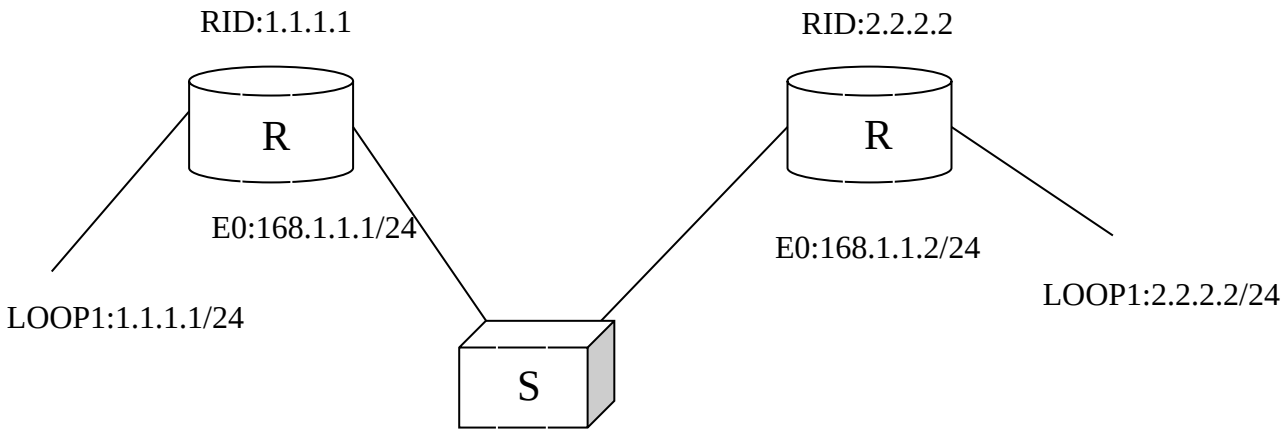


图 6. 1 实验 1 组网图

详细结果与动态交互过程请参见附件中的***视频***。（网络实验室中录制）

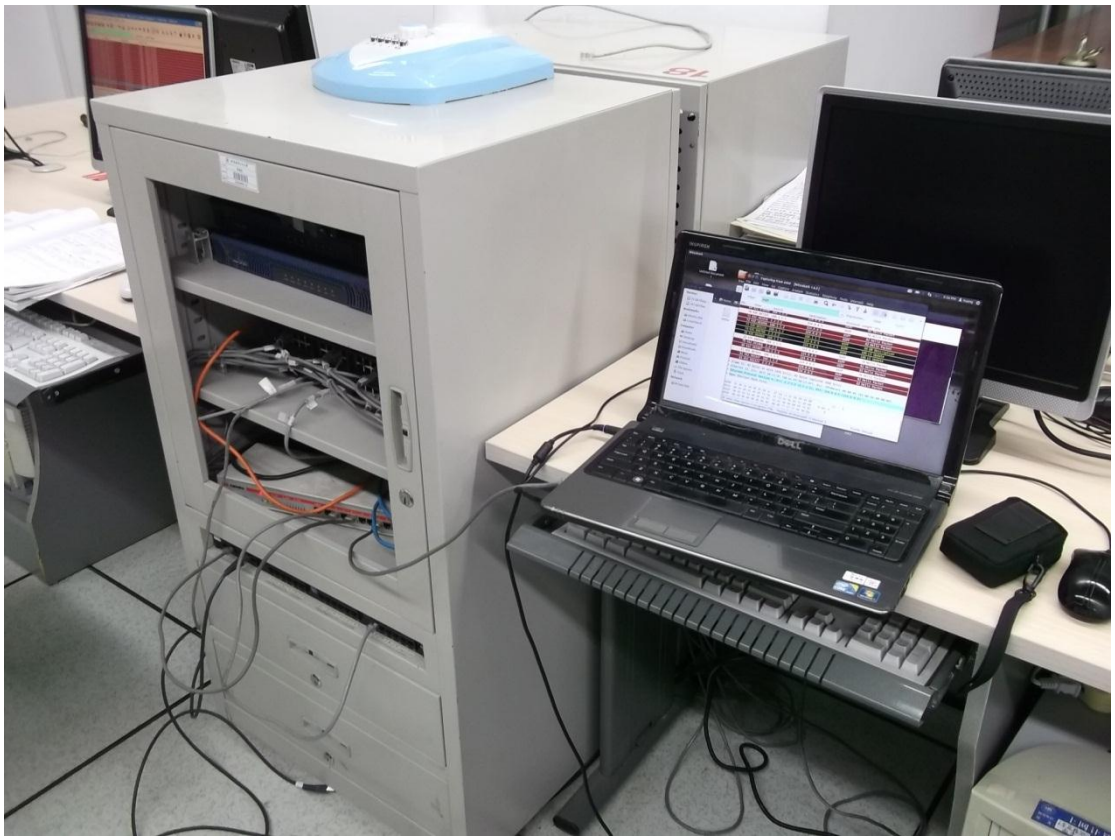


图 6. 2 实验 1 实际测试场景

6.4 实验 2：两个路由终端模拟

6.4.1 实验组网图

（下图只是用终端模拟的示意图，实际两个路由器之间应该有交换机（或集线器），因为网线接头不一致。）

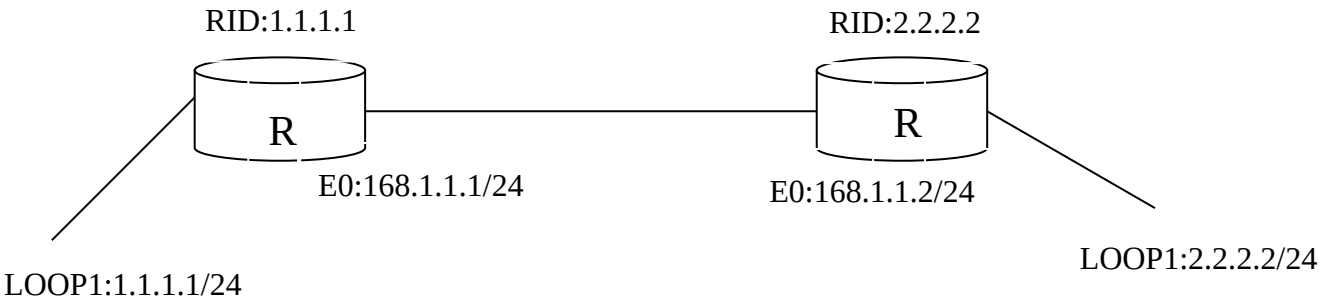


图 6.3 实验 2 组网图

在本实验中采用两台 PC 机模拟路由器，初始化文件由程序从开头读入，主要包括区域、接口、路由的配置（setting），无法自动生成的 LSA 的配置（initlsa）。由于这里测试时是用 PC 直接模拟，同一块网卡可能产生不同 IP 源地址的包。收包时，为区分开不是发自相连接口的包，还需引入互联初始化文件 initcnnt。

6.4.2 配置文件

R1 的初始化路由器基本配置（文件 init/setting 中）

```
setting
#用来初始化路由器的基本配置
R1      #路由器名称
1.1.1.1         #Router ID
0          #area ID
2          #interface  个数
11         #interface 编号，相当于 E0/0
168.1.1.1    #interface  配置的 IP 地址
```

```
255.255.255.0     #interface 配置的子网掩码 mask
100     #interface 对应的 Cost
eth1       #interface  对应的网卡名称
12     #interface 编号，相当于 E0/1
1.1.1.1    #interface  配置的 IP 地址
255.255.255.0    #interface 配置的子网掩码 mask
500       #interface 对应的 Cost
eth1       #interface  对应的网卡名称
```

```
initlsa
0           #配置初始的 LSA 条数
```

```
initcnnt
1           #1 对，2 行一组
168.1.1.1
168.1.1.2
2          #2 组，3 行一组
1.1.1.1
2.2.2.2
168.1.1.2
2.2.2.2
1.1.1.1
168.1.1.1
```

初始化成功后程序会在终端打印初始化信息：

```
BEGIN:SETTING.....................
Now Router is: R1
router ID: 1.1.1.1
AREA ID: 0
2 interfaces
168.1.1.1-255.255.255.0
```

cost: 100

NIC: eth1

1.1.1.1-255.255.255.0

cost: 500

NIC: eth1

END:SETTING.....

BEGIN: CONNECT.....

168.1.1.1-168.1.1.2

END:CONNECT.....

generate initial LSA.....

End:generate lsas

6.4.3 DR 选举展示

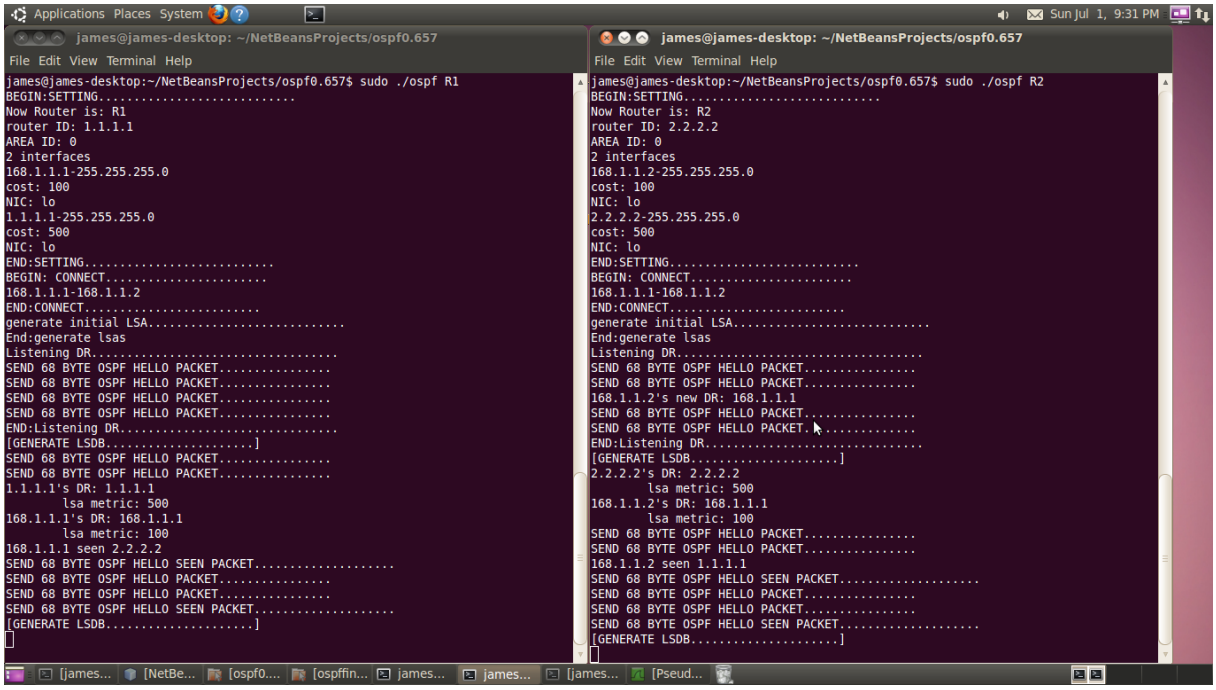
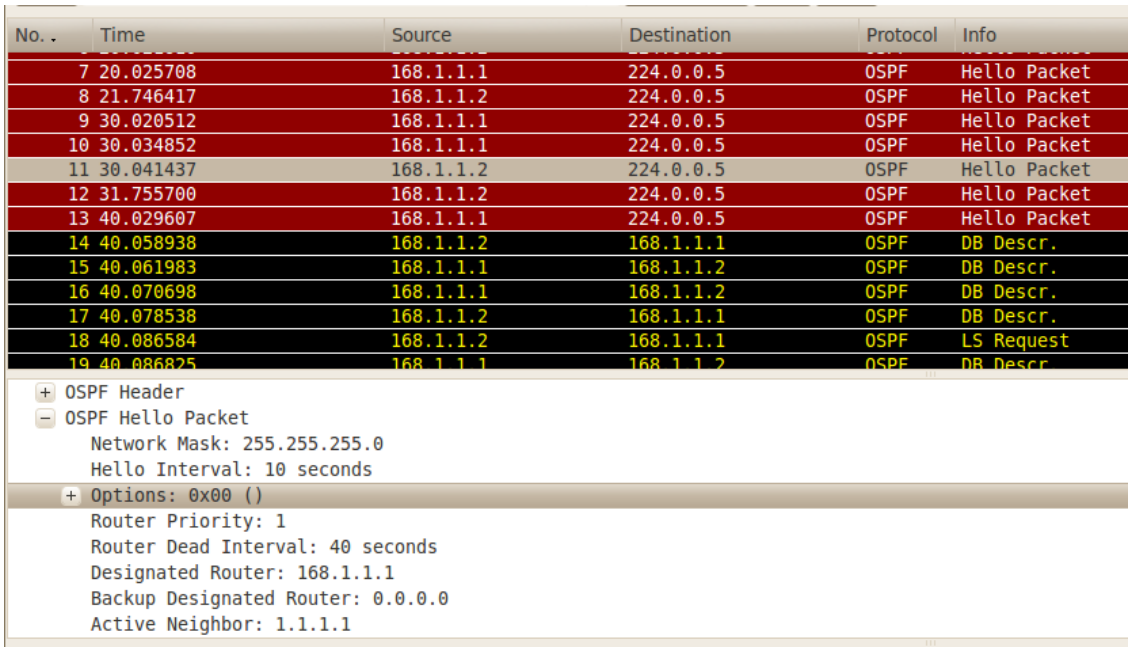


图 6. 4 DR 选举展示

从上图可以看出，最后选举出 DR：1.1.1.1 的 DR 是 1.1.1.1；168.1.1.1 的 DR 是 168.1.1.1；2.2.2.2 的 DR 是 2.2.2.2；168.1.1.2 的 DR 是 168.1.1.1。

6.4.5 路由交互过程

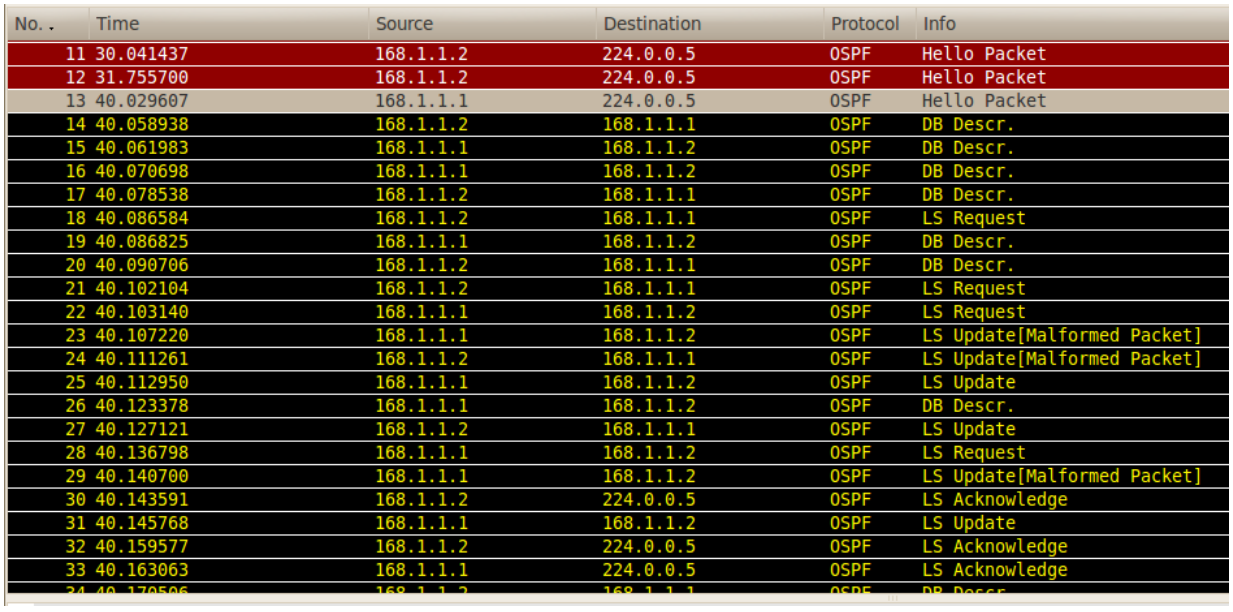


No.	Time	Source	Destination	Protocol	Info
7	20.025708	168.1.1.1	224.0.0.5	OSPF	Hello Packet
8	21.746417	168.1.1.2	224.0.0.5	OSPF	Hello Packet
9	30.020512	168.1.1.1	224.0.0.5	OSPF	Hello Packet
10	30.034852	168.1.1.1	224.0.0.5	OSPF	Hello Packet
11	30.041437	168.1.1.2	224.0.0.5	OSPF	Hello Packet
12	31.755700	168.1.1.2	224.0.0.5	OSPF	Hello Packet
13	40.029607	168.1.1.1	224.0.0.5	OSPF	Hello Packet
14	40.058938	168.1.1.2	168.1.1.1	OSPF	DB Descr.
15	40.061983	168.1.1.1	168.1.1.2	OSPF	DB Descr.
16	40.070698	168.1.1.1	168.1.1.2	OSPF	DB Descr.
17	40.078538	168.1.1.2	168.1.1.1	OSPF	DB Descr.
18	40.086584	168.1.1.2	168.1.1.1	OSPF	LS Request
19	40.086825	168.1.1.1	168.1.1.2	OSPF	DB Descr.

```
+ OSPF Header
- OSPF Hello Packet
    Network Mask: 255.255.255.0
    Hello Interval: 10 seconds
  + Options: 0x00 ()
    Router Priority: 1
    Router Dead Interval: 40 seconds
    Designated Router: 168.1.1.1
    Backup Designated Router: 0.0.0.0
    Active Neighbor: 1.1.1.1
```

图 6. 5 路由交互过程 1

Hello 报文中已经选出 DR，这里的 Hello 报文还是 Hello Seen 报文（发现了邻居）。



No.	Time	Source	Destination	Protocol	Info
11	30.041437	168.1.1.2	224.0.0.5	OSPF	Hello Packet
12	31.755700	168.1.1.2	224.0.0.5	OSPF	Hello Packet
13	40.029607	168.1.1.1	224.0.0.5	OSPF	Hello Packet
14	40.058938	168.1.1.2	168.1.1.1	OSPF	DB Descr.
15	40.061983	168.1.1.1	168.1.1.2	OSPF	DB Descr.
16	40.070698	168.1.1.1	168.1.1.2	OSPF	DB Descr.
17	40.078538	168.1.1.2	168.1.1.1	OSPF	DB Descr.
18	40.086584	168.1.1.2	168.1.1.1	OSPF	LS Request
19	40.086825	168.1.1.1	168.1.1.2	OSPF	DB Descr.
20	40.090706	168.1.1.2	168.1.1.1	OSPF	DB Descr.
21	40.102104	168.1.1.2	168.1.1.1	OSPF	LS Request
22	40.103140	168.1.1.1	168.1.1.2	OSPF	LS Request
23	40.107220	168.1.1.1	168.1.1.2	OSPF	LS Update[Malformed Packet]
24	40.111261	168.1.1.2	168.1.1.1	OSPF	LS Update[Malformed Packet]
25	40.112950	168.1.1.1	168.1.1.2	OSPF	LS Update
26	40.123378	168.1.1.1	168.1.1.2	OSPF	DB Descr.
27	40.127121	168.1.1.2	168.1.1.1	OSPF	LS Update
28	40.136798	168.1.1.1	168.1.1.2	OSPF	LS Request
29	40.140700	168.1.1.1	168.1.1.2	OSPF	LS Update[Malformed Packet]
30	40.143591	168.1.1.2	224.0.0.5	OSPF	LS Acknowledge
31	40.145768	168.1.1.1	168.1.1.2	OSPF	LS Update
32	40.159577	168.1.1.2	224.0.0.5	OSPF	LS Acknowledge
33	40.163063	168.1.1.1	224.0.0.5	OSPF	LS Acknowledge
34	40.170506	168.1.1.2	168.1.1.1	OSPF	DB Descr.

图 6. 6 路由交互过程 2

上图显示报文交互过程。其中，部分 LSU 发生错误，程序能实现***重发***功能。

6.4.6 DD 报文确定主从关系

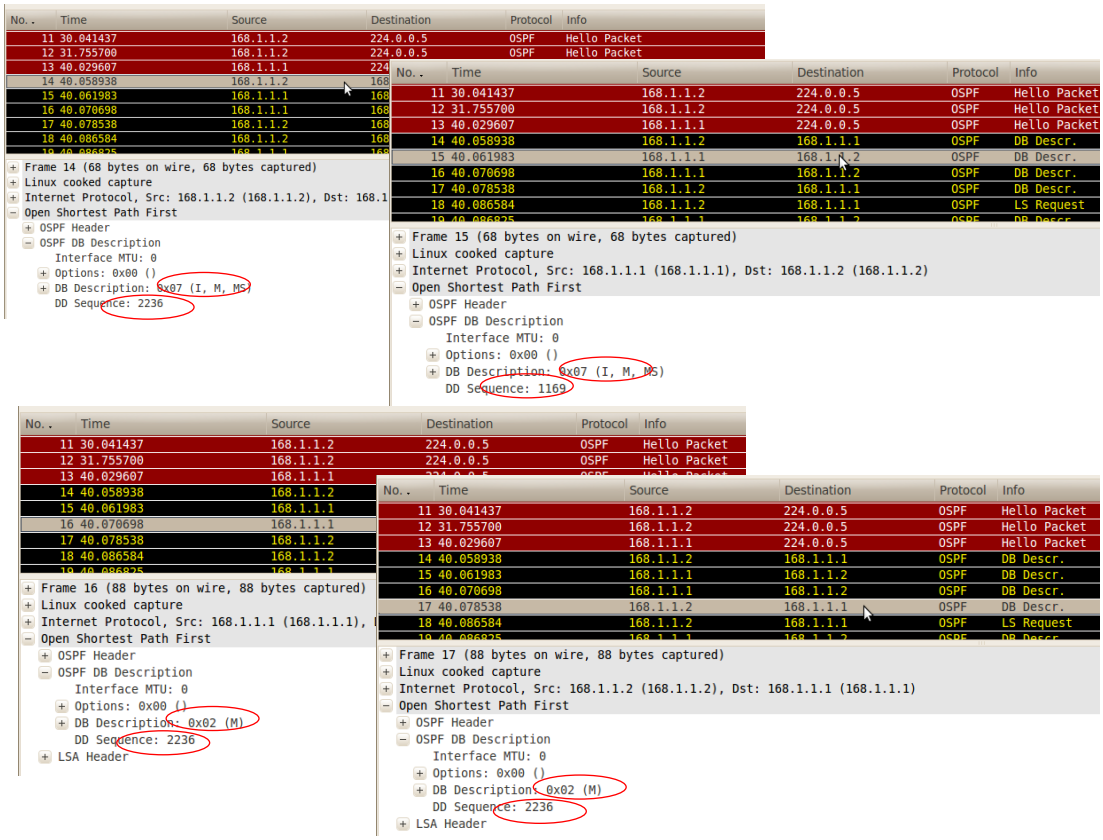


图 6. 7 DD 报文确定主从关系

上面序号依次为 14、15、16、17 的报文显示了主、从选举过程。

6.4.7 LSR 报文

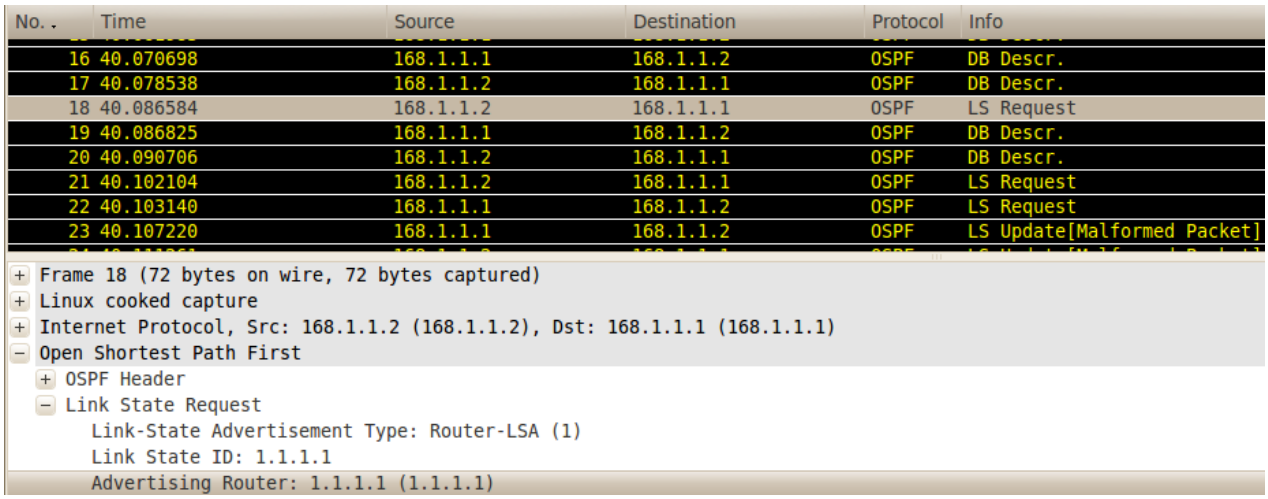


图 6. 8 LSR 报文

当一台路由器发现 DD 报文中描述的 LSA 在本地的 LSDB 中不存在的话就会发送 LSR 报文对其进行请求。

6.4.8 LSU 报文

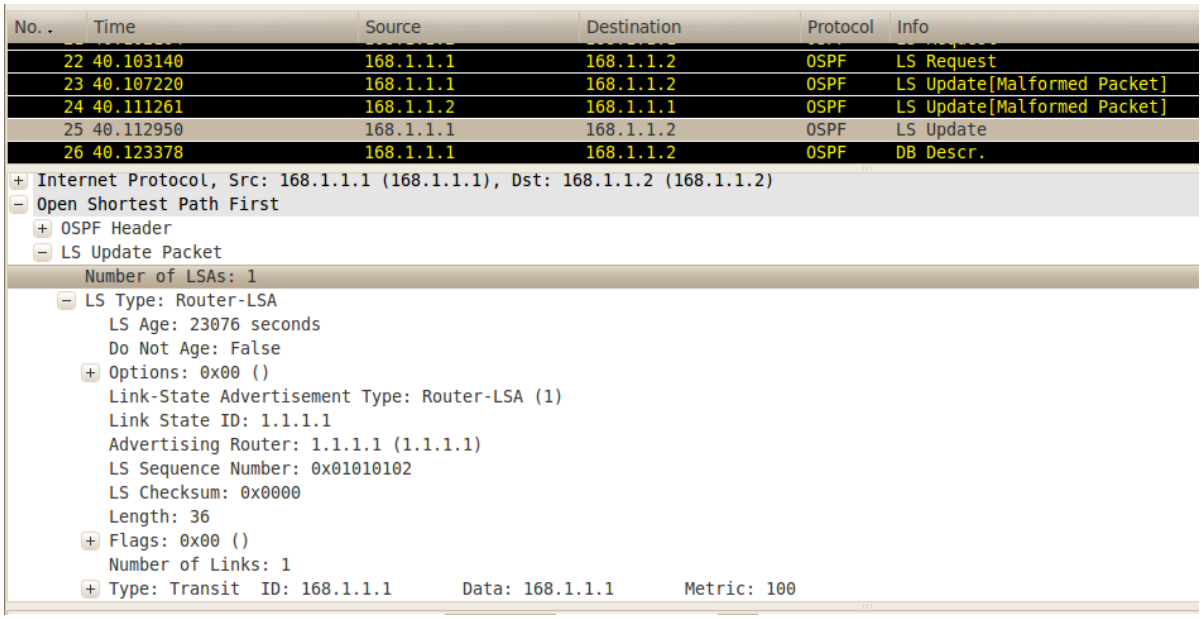


图 6. 9 LSU 报文

收到 LSR 后，被请求的路由器会在本地查找到相应的 LSA，然后发送 LSU 回应请求。

6.4.9 LSAck 报文

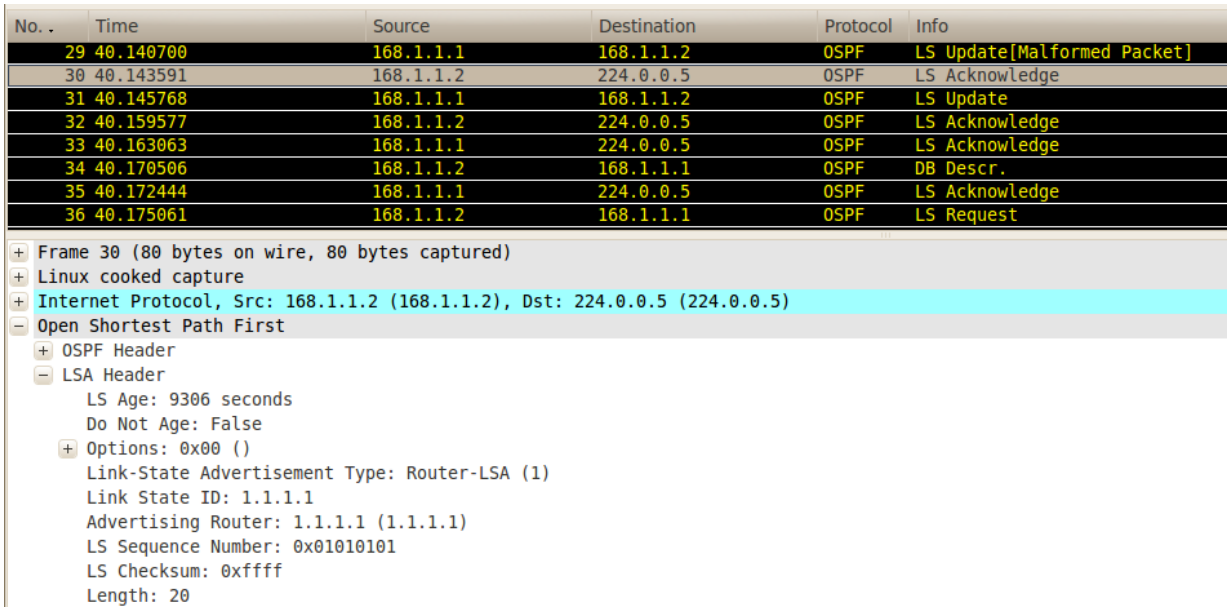


图 6. 10 LSAck 报文

收到 LSA 以后，发出请求的路由器会返回 LSAck 表示自身收到了 LSU 报文。

6.4.10 输出的 LSDB 信息

在 lsdb 文件中：

=====area: 0====	[5]
[1]	ls_type:network lsa
ls_type:router lsa	bornTime:1340892858
bornTime:1340892836	seqNum:16843013
seqNum:16843009	bornRouterId:1.1.1.1
bornRouterId:1.1.1.1	ls_id:1.1.1.1
ls_id:1.1.1.1	**
**	mask:255.255.255.0
link_id:1.1.1.1	router:1.1.1.1
link_data:1.1.1.1	----------
link_type:TRANS	

link_tos_count:8

tos_metric:500

---

[2]

ls_type:router lsa

bornTime:1340892836

seqNum:16843010

bornRouterId:1.1.1.1

ls_id:1.1.1.1

**

link_id:168.1.1.1

link_data:168.1.1.1

link_type:TRANS

link_tos_count:0

tos_metric:100

---

[3]

ls_type:router lsa

bornTime:26274

seqNum:33686018

bornRouterId:2.2.2.2

ls_id:2.2.2.2

**

link_id:2.2.2.2

link_data:2.2.2.2

[6]

ls_type:network lsa

bornTime:1340892858

seqNum:16843014

bornRouterId:1.1.1.1

ls_id:168.1.1.1

**

mask:255.255.255.0

router:1.1.1.1

router:2.2.2.2

---

[7]

ls_type:network lsa

bornTime:26298

seqNum:33686024

bornRouterId:2.2.2.2

ls_id:168.1.1.2

**

mask:255.255.255.0

router:2.2.2.2

router:1.1.1.1

---

link_type:TRANS link_tos_count:0 tos_metric:500 ---------- [4] ls_type:router lsa bornTime:26274 seqNum:33686019 bornRouterId:2.2.2.2 ls_id:2.2.2.2 ** link_id:168.1.1.2 link_data:168.1.1.2 link_type:TRANS link_tos_count:0 tos_metric:100 ----------	

6.4.11 输出路由表信息

在文件 out 中（这里结果相对较简单）：

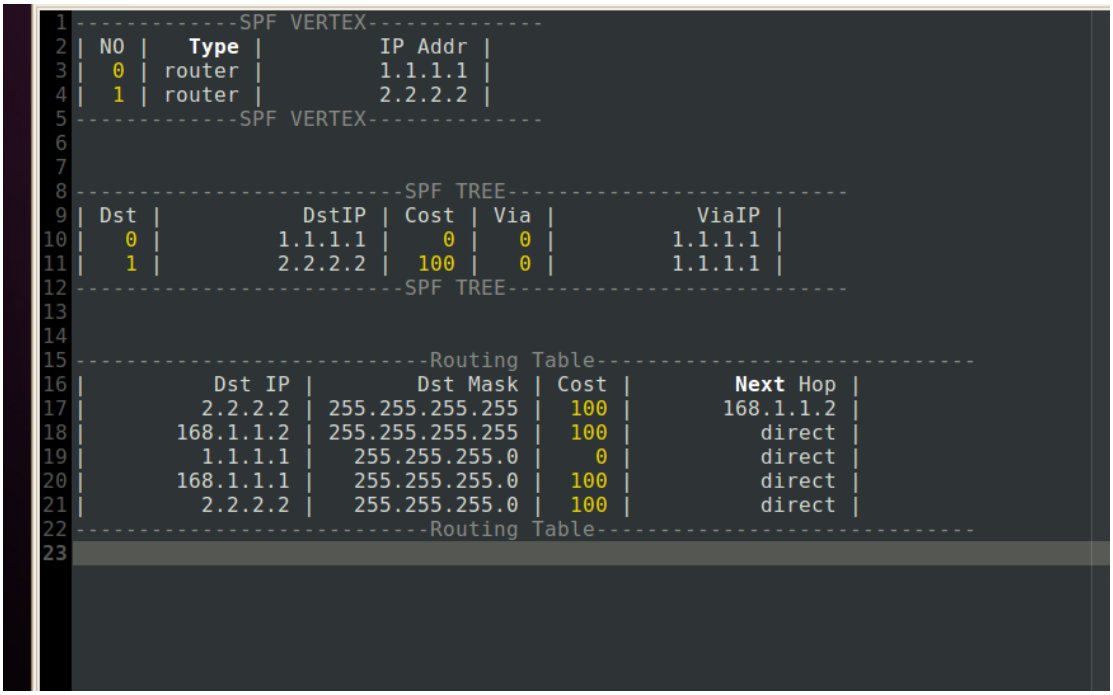


图 6. 11 实验 2 输出路由表信息

6.4.12 填入系统路由表，实现路由转发

由于这里是单网卡，所以填入时会有 ioctl:No Such Process 错误。

在与路由器交互（实验 1）时才能填写入系统。（虚拟出的 IP 地址与实际的 IP 地址一致）

6.4.13 验证 Dead Interval 维护机制

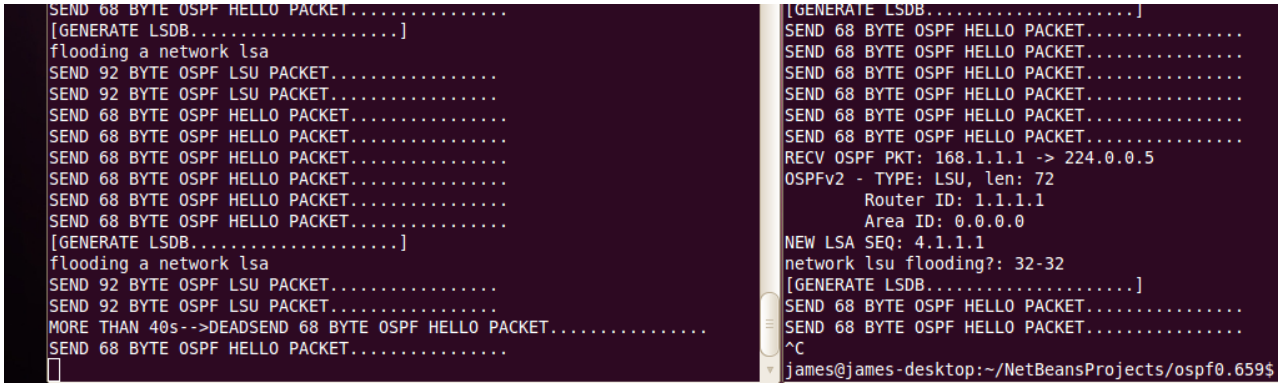


图 6. 12 验证 Dead Interval 维护机制

如上图，两个终端还在运行程序时，杀死其中的一个终端运行的 ospf 程序，则另一终端过 40s 会标记邻居为 Dead。

6.5 实验 3：四个路由终端模拟

6.5.1 实验组网图

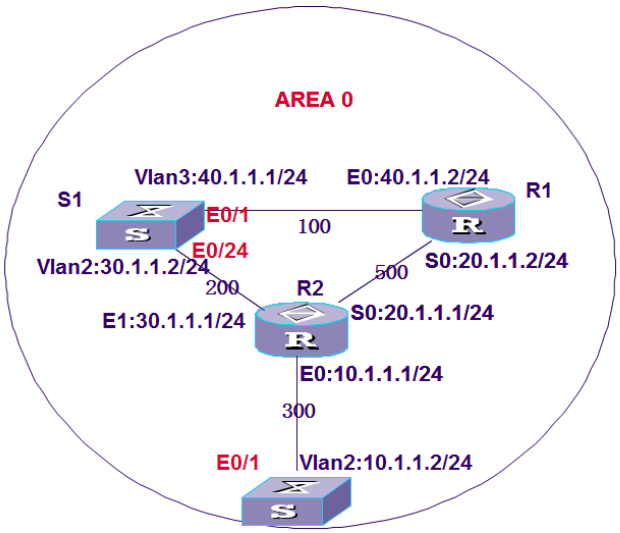


图 6. 13 实验 3 实验组网图

在本实验中采用两台 PC 机模拟路由器，初始化文件由程序从开头读入，主要包括区域、接口、路由的配置（setting），无法自动生成的 LSA 的配置（initlsa）。由于这里测试时是用 PC 直接模拟，同一块网卡可能产生不同 IP 源地址的包。收包时，为区分开不是发自相连接口的包，还需引入互联初始化文件 initcnnt。

6.5.2 配置文件

R1 的初始化配置文件：

setting 文件

#用来初始化路由器的基本配置

R1

2.2.2.2

0

2

11

40.1.1.2

255.255.255.0 100 eth1 12 20.1.1.2 255.255.255.0 500 eth1
initlsa 文件 2 0 3.3.3.3 20.1.1.2 1 500 0 20.1.1.0 255.255.255.0 3 500
initcnnt 文件 4 40.1.1.1 40.1.1.2 30.1.1.1 30.1.1.2 20.1.1.1 20.1.1.2 10.1.1.1

10.1.1.2

8

1.1.1.1

2.2.2.2

40.1.1.2

2.2.2.2

1.1.1.1

40.1.1.1

1.1.1.1

3.3.3.3

30.1.1.1

3.3.3.3

1.1.1.1

30.1.1.2

2.2.2.2

3.3.3.3

20.1.1.1

3.3.3.3

2.2.2.2

20.1.1.2

3.3.3.3

4.4.4.4

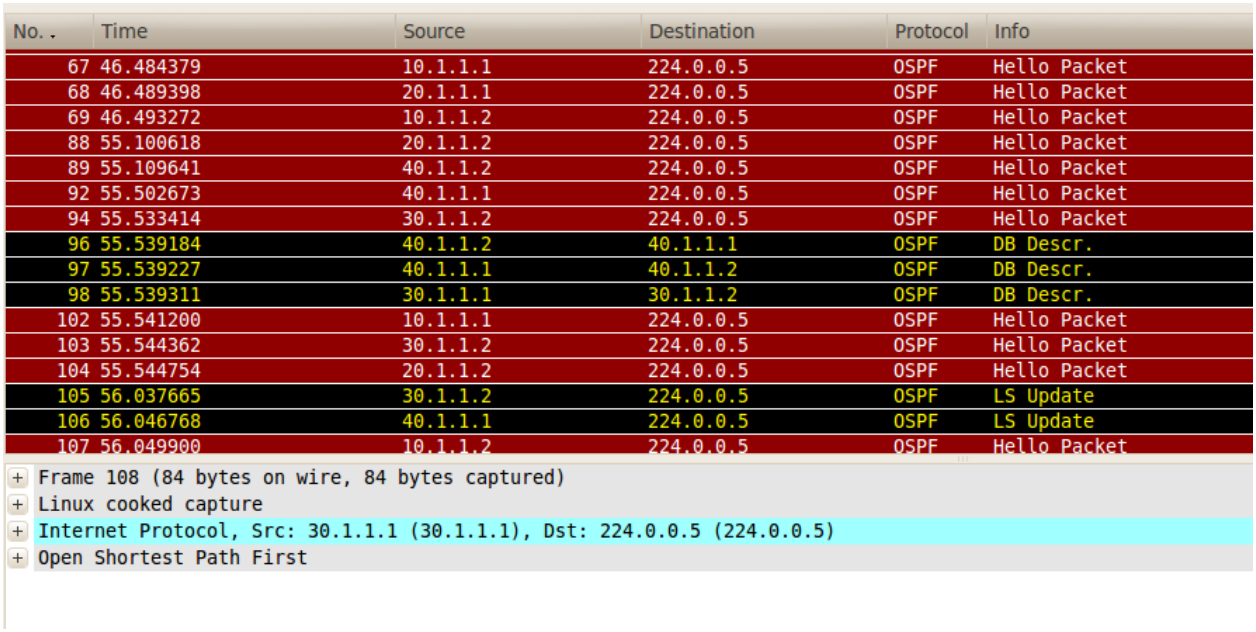
10.1.1.2

4.4.4.4

3.3.3.3

10.1.1.1

路由交互过程、DD 报文确定主从关系、LSR 报文、LSU 报文等细节与实验 2 类似，这里不再赘述。



No.	Time	Source	Destination	Protocol	Info
67	46.484379	10.1.1.1	224.0.0.5	OSPF	Hello Packet
68	46.489398	20.1.1.1	224.0.0.5	OSPF	Hello Packet
69	46.493272	10.1.1.2	224.0.0.5	OSPF	Hello Packet
88	55.100618	20.1.1.2	224.0.0.5	OSPF	Hello Packet
89	55.109641	40.1.1.2	224.0.0.5	OSPF	Hello Packet
92	55.502673	40.1.1.1	224.0.0.5	OSPF	Hello Packet
94	55.533414	30.1.1.2	224.0.0.5	OSPF	Hello Packet
96	55.539184	40.1.1.2	40.1.1.1	OSPF	DB Descr.
97	55.539227	40.1.1.1	40.1.1.2	OSPF	DB Descr.
98	55.539311	30.1.1.1	30.1.1.2	OSPF	DB Descr.
102	55.541200	10.1.1.1	224.0.0.5	OSPF	Hello Packet
103	55.544362	30.1.1.2	224.0.0.5	OSPF	Hello Packet
104	55.544754	20.1.1.2	224.0.0.5	OSPF	Hello Packet
105	56.037665	30.1.1.2	224.0.0.5	OSPF	LS Update
106	56.046768	40.1.1.1	224.0.0.5	OSPF	LS Update
107	56.049900	10.1.1.2	224.0.0.5	OSPF	Hello Packet

图 6. 14 实验 3 路由交互过程

6.5.3 输出的 LSDB 信息

//

=====area: 0====	[10]
[1]	ls_type:network lsa
ls_type:router lsa	bornTime:9038
bornTime:1340613401	seqNum:50529032
seqNum:33686018	bornRouterId:3.3.3.3
bornRouterId:2.2.2.2	ls_id:10.1.1.1
ls_id:2.2.2.2	**
**	mask:255.255.255.0
link_id:3.3.3.3	router:3.3.3.3
link_data:20.1.1.2	router:4.4.4.4
link_type:PTP	----------
link_tos_count:0	
tos_metric:500	
----------	[11]

[2]

ls_type:router lsa

bornTime:1340613401

seqNum:33686019

bornRouterId:2.2.2.2

ls_id:2.2.2.2

**

link_id:20.1.1.0

link_data:255.255.255.0

link_type:STUB

link_tos_count:0

tos_metric:500

---

[3]

ls_type:router lsa

bornTime:1340613411

seqNum:33686020

bornRouterId:2.2.2.2

ls_id:2.2.2.2

**

link_id:20.1.1.2

link_data:20.1.1.2

link_type:TRANS

link_tos_count:19

tos_metric:500

ls_type:network lsa

bornTime:9038

seqNum:50529034

bornRouterId:3.3.3.3

ls_id:30.1.1.1

**

mask:255.255.255.0

router:3.3.3.3

router:1.1.1.1

---

[12]

ls_type:router lsa

bornTime:8986

seqNum:50529027

bornRouterId:3.3.3.3

ls_id:3.3.3.3

**

link_id:2.2.2.2

link_data:20.1.1.1

link_type:PTP

link_tos_count:0

tos_metric:500

---

[13]

ls_type:router lsa

---

[4]

ls_type:router lsa

bornTime:1340613411

seqNum:33686021

bornRouterId:2.2.2.2

ls_id:2.2.2.2

**

link_id:40.1.1.1

link_data:40.1.1.2

link_type:TRANS

link_tos_count:0

tos_metric:100

---

[5]

ls_type:network lsa

bornTime:9037

seqNum:33686022

bornRouterId:2.2.2.2

ls_id:20.1.1.2

**

mask:255.255.255.0

router:2.2.2.2

router:3.3.3.3

---

bornTime:8986

seqNum:50529028

bornRouterId:3.3.3.3

ls_id:3.3.3.3

**

link_id:20.1.1.0

link_data:255.255.255.0

link_type:STUB

link_tos_count:0

tos_metric:500

---

[14]

ls_type:router lsa

bornTime:8995

seqNum:50529029

bornRouterId:3.3.3.3

ls_id:3.3.3.3

**

link_id:10.1.1.1

link_data:10.1.1.1

link_type:TRANS

link_tos_count:0

tos_metric:300

---

[15]

[6]

ls_type:router lsa

bornTime:8995

seqNum:16843009

bornRouterId:1.1.1.1

ls_id:1.1.1.1

**

link_id:30.1.1.2

link_data:30.1.1.2

link_type:TRANS

link_tos_count:0

tos_metric:200

---

[7]

ls_type:router lsa

bornTime:8995

seqNum:16843010

bornRouterId:1.1.1.1

ls_id:1.1.1.1

**

link_id:40.1.1.1

link_data:40.1.1.1

link_type:TRANS

link_tos_count:231

tos_metric:100

ls_type:router lsa

bornTime:8995

seqNum:50529030

bornRouterId:3.3.3.3

ls_id:3.3.3.3

**

link_id:20.1.1.1

link_data:20.1.1.1

link_type:TRANS

link_tos_count:0

tos_metric:500

---

[16]

ls_type:router lsa

bornTime:8995

seqNum:50529031

bornRouterId:3.3.3.3

ls_id:3.3.3.3

**

link_id:30.1.1.1

link_data:30.1.1.1

link_type:TRANS

link_tos_count:0

tos_metric:200

---

[8]

ls_type:network lsa

bornTime:9038

seqNum:16843011

bornRouterId:1.1.1.1

ls_id:30.1.1.2

**

mask:255.255.255.0

router:1.1.1.1

router:3.3.3.3

---

[9]

ls_type:network lsa

bornTime:9038

seqNum:16843012

bornRouterId:1.1.1.1

ls_id:40.1.1.1

**

mask:255.255.255.0

router:1.1.1.1

router:2.2.2.2

---

[17]

ls_type:router lsa

bornTime:8996

seqNum:67372036

bornRouterId:4.4.4.4

ls_id:4.4.4.4

**

link_id:10.1.1.1

link_data:10.1.1.2

link_type:TRANS

link_tos_count:0

tos_metric:300

---

[18]

ls_type:network lsa

bornTime:9038

seqNum:50529033

bornRouterId:3.3.3.3

ls_id:20.1.1.1

**

mask:255.255.255.0

router:3.3.3.3

由上面的 LSDB 可以得出网络全貌：

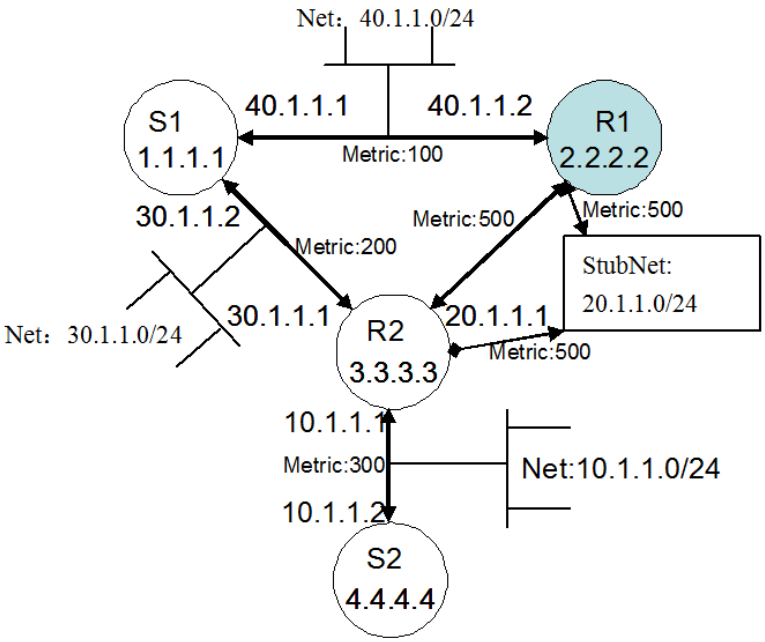


图 6. 15 实验 3LSDB 所得网络全景图

6.5.4 输出路由表信息

```
-------------SPF VERTEX--------------

--------------------------SPF TREE-----------------------------
| Dst |           DstIP | Cost | Via |           ViaIP |
|   0 |         3.3.3.3 |  300 |   3 |         1.1.1.1 |
|   1 |         2.2.2.2 |    0 |   1 |         2.2.2.2 |
|   2 |        20.1.1.0 |  500 |   1 |         2.2.2.2 |
|   3 |         1.1.1.1 |  100 |   1 |         2.2.2.2 |
|   4 |         4.4.4.4 |  600 |   0 |         3.3.3.3 |
--------------------------SPF TREE-----------------------------


---------------------------Routing Table-------------------------------
|          Dst IP |         Dst Mask | Cost |         Next Hop |
|        10.1.1.1 | 255.255.255.255 |  300 |         40.1.1.1 |
|        20.1.1.1 | 255.255.255.255 |  300 |         40.1.1.1 |
|        30.1.1.1 | 255.255.255.255 |  300 |         40.1.1.1 |
|        20.1.1.0 |   255.255.255.0 |  500 |           direct |
|        30.1.1.2 | 255.255.255.255 |  100 |         40.1.1.1 |
|        40.1.1.1 | 255.255.255.255 |  100 |           direct |
|        10.1.1.2 | 255.255.255.255 |  600 |         40.1.1.1 |
|        20.1.1.2 |   255.255.255.0 |  300 |         40.1.1.1 |
|        30.1.1.2 |   255.255.255.0 |  300 |         40.1.1.1 |
|        30.1.1.1 |   255.255.255.0 |  300 |         40.1.1.1 |
|        40.1.1.1 |   255.255.255.0 |  100 |           direct |
|        10.1.1.1 |   255.255.255.0 |  600 |         40.1.1.1 |
|        20.1.1.1 |   255.255.255.0 |  300 |         40.1.1.1 |
---------------------------Routing Table-------------------------------
```

图 6. 16 实验 3 输出路由表信息

```
-------------SPF VERTEX--------------
| NO |      Type |              IP Addr |
|    0 | router |              3.3.3.3 |
|    1 | router |              2.2.2.2 |
|    2 |      stub |              20.1.1.0 |
|    3 | router |              1.1.1.1 |
|    4 | router |              4.4.4.4 |
-------------SPF VERTEX--------------



--------------------------SPF TREE---------------------------
| Dst |                DstIP | Cost | Via |                ViaIP |
|     0 |              3.3.3.3 |    300 |     3 |              1.1.1.1 |
|     1 |              2.2.2.2 |        0 |     1 |              2.2.2.2 |
|     2 |            20.1.1.0 |    500 |     1 |              2.2.2.2 |
|     3 |              1.1.1.1 |    100 |     1 |              2.2.2.2 |
|     4 |              4.4.4.4 |    600 |     0 |              3.3.3.3 |
--------------------------SPF TREE---------------------------



----------------------------Routing Table-----------------------------
|                Dst IP |             Dst Mask | Cost |             Next Hop |
|             10.1.1.1 | 255.255.255.255 |    300 |             40.1.1.1 |
|             20.1.1.1 | 255.255.255.255 |    300 |             40.1.1.1 |
|             30.1.1.1 | 255.255.255.255 |    300 |             40.1.1.1 |
|             20.1.1.0 |      255.255.255.0 |    500 |                direct |
|             30.1.1.2 | 255.255.255.255 |    100 |             40.1.1.1 |
```

```
|          40.1.1.1 | 255.255.255.255 |   100 |            direct |
|          10.1.1.2 | 255.255.255.255 |   600 |          40.1.1.1 |
|          20.1.1.2 |   255.255.255.0 |   300 |          40.1.1.1 |
|          30.1.1.2 |   255.255.255.0 |   300 |          40.1.1.1 |
|          30.1.1.1 |   255.255.255.0 |   300 |          40.1.1.1 |
|          40.1.1.1 |   255.255.255.0 |   100 |            direct |
|          10.1.1.1 |   255.255.255.0 |   600 |          40.1.1.1 |
|          20.1.1.1 |   255.255.255.0 |   300 |          40.1.1.1 |
----------------------------Routing Table-----------------------------
```

从上面的 SPF tree 可以得出最短路径生成树：

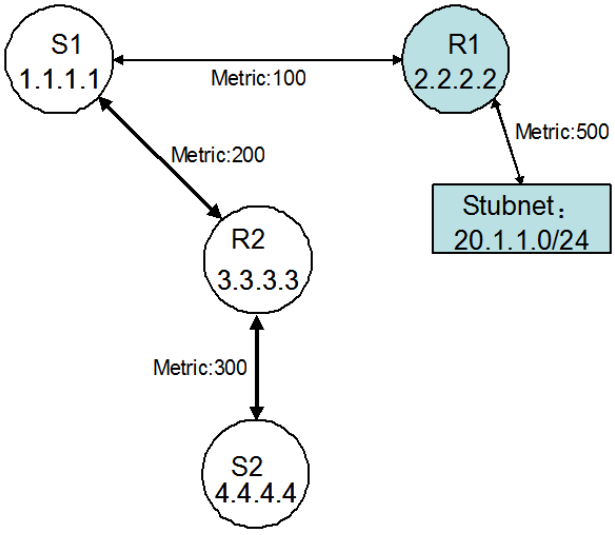


图 6. 17 实验 3 所得最短路径生成树

6.5.5 填入系统路由表，实现路由转发

由于这里是单网卡，所以填入时会有 ioctl:No Such Process 错误。

在与路由器交互（实验 4）时才能填写入系统。（虚拟出的 IP 地址与实际的 IP 地址一致）

6.6 实验 4：四个路由实验室组网测试

6.6.1 实验组网图

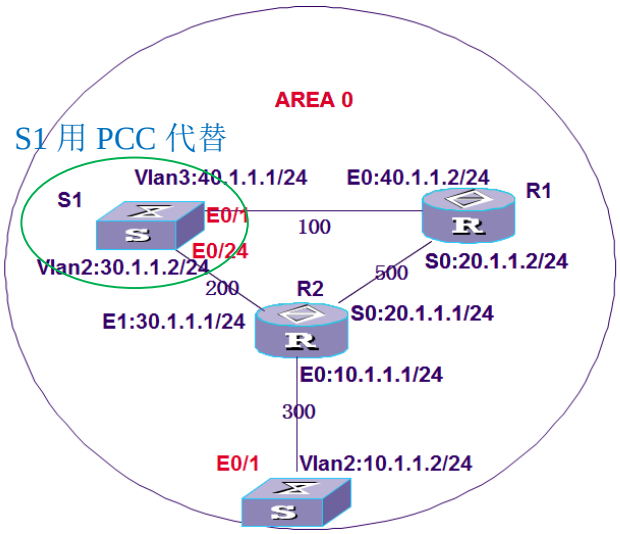


图 6. 18 实验 4 实验组网图

将 S1 用机房中的 PCC 代替，双网卡进行测试。但测试时不太稳定，有时不能实现 ping 通。实验 1 的交互比较稳定。

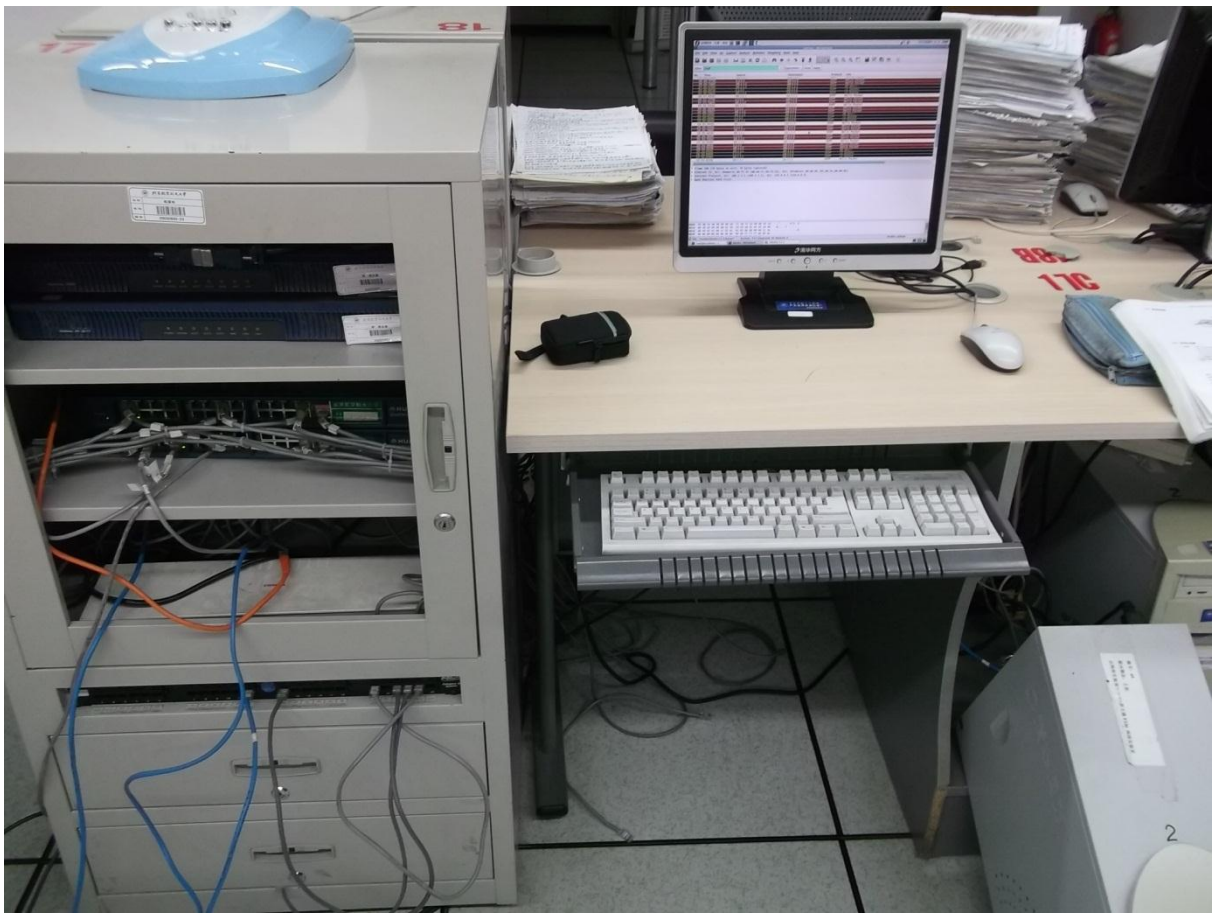


图 6. 19 实验 4 实际测试场景 1

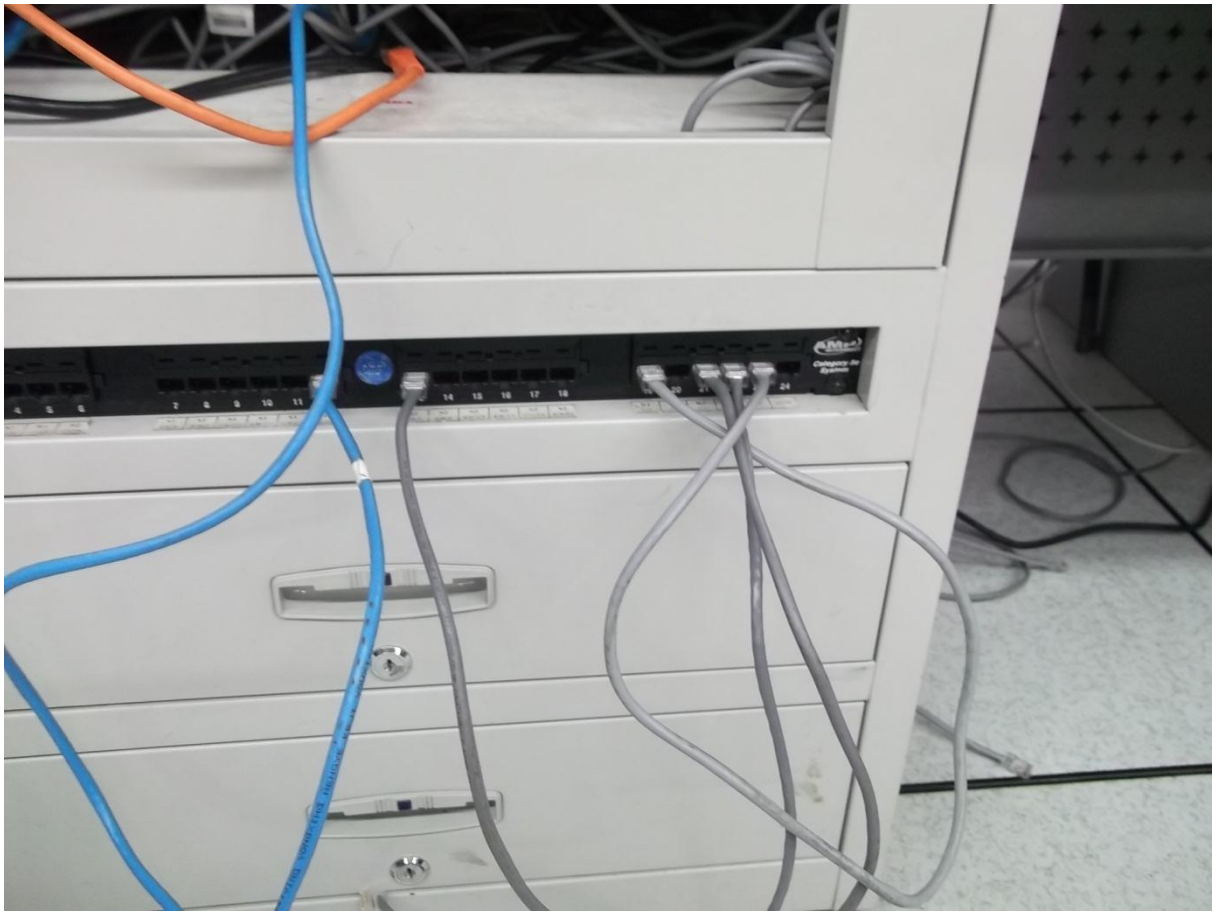


图 6. 20 实验 4 实际测试场景 2

6.7 结果分析与验证

- 测试 OSPF 协议实现的正确性
  - 上面 4 个实验均可得到理想的结果。
- 验证 OSPF 协议规定格式数据的封装发送
  - 上面 4 个实验的设计，均可通过 Wireshark 抓包截取相关报文，解析出来的结果均可读，没有出现乱序或无意义字段。
- 验证符合 OSPF 协议规定格式的数据包的接收与逻辑处理
- 验证 Dead Interval 维护、邻居关系建立及报文交换过程
- 与路由器等设备交互的正确性：实验 1：电脑与路由直接相连

- 路由器交互过程的正确性（五类报文）：实验 2：两个路由交互验证
  - DR 选举过程

    参见图 6. 4 DR 选举展示
  - DD 主从选举

    参见图 6. 7 DD 报文确定主从关系
  - DD 报文交换

    参见图 6. 7 DD 报文确定主从关系
  - LSR、LSU、LSAck 维护 LSDB 过程

    参见图 6. 8 LSR 报文、图 6. 9 LSU 报文、图 6. 10 LSAck 报文
  - 重传机制

    参见图 6. 6 路由交互过程 2
- 验证生成的 LSDB、SPF 树、路由表的正确性：实验 3：四个路由交互验证
  - 可以根据收集到的 LSA 正确构建出 LSDB
  - 可以根据 LSDB 得出最短路径生成树

    参见图 6. 17 实验 3 所得最短路径生成树
  - 由最短路径生成树和两类 LSA 导出了正确的路由表

    参见图 6. 16 实验 3 输出路由表信息
- 验证路由表转发的正确性：实验 4：电脑与路由多台验证
  - 实现路由转发功能

    填入系统路由表实现

    相互之间能 ping 通

结论

工作总结

- 学习 OSPF 协议，了解 OSPF 协议的原理；

具体分析 OSPF 协议的相关内容，阅读 RFC2328 文档相关内容。针对 RFC 2328 进行一些简化，提取出能够实现的子集部分：

- 报文交互过程
  - HELLO 报文交互
  - DR 选举过程
  - DD 主从选举
  - 邻接结构 DD 报文交换
  - LSR、LSU、LSAck 维护 LSDB 过程
- 泛洪过程
- 可以根据收集到的 LSA 构建出 LSDB
- SPF 算法（Dijkstra 算法）
- 导出路由表
- 路由转发功能

- 基于 Linux 实现基本的 OSPF 协议
  - 识别第 1、2、3、4、5 类报文种类
  - 实现 OSPF 协议的格式数据的封装发送和接收解封装以及相应处理。
  - DR 的选举
  - 利用 Hello 报文建立邻居->邻接关系
  - 利用 DD 报文建立邻居 Master/Slave 关系
  - LSA 的生成、交换与泛洪->建立 LSDB
  - 生成最短路径树（SPF 算法，或称 Dijkstra 算法）
  - 生成最终的路由表
  - 实现路由转发

- 在 Linux 环境下对实现的 OSPF 协议进行测试

设计出四个实验进行全方面测试:

- 与路由器等设备交互的正确性：实验 1：电脑与路由直接相连
- 路由器交互过程的正确性（五类报文）：实验 2：两个路由交互验证
  - DR 选举过程

    参见图 6. 4 DR 选举展示
  - DD 主从选举

    参见图 6. 7 DD 报文确定主从关系
  - DD 报文交换

    参见图 6. 7 DD 报文确定主从关系
  - LSR、LSU、LSAck 维护 LSDB 过程

    参见图 6. 8 LSR 报文、图 6. 9 LSU 报文、图 6. 10 LSAck 报文
  - 重传机制

    参见图 6. 6 路由交互过程 2
- 验证生成的 LSDB、SPF 树、路由表的正确性：实验 3：四个路由交互验证
  - 可以根据收集到的 LSA 正确构建出 LSDB
  - 可以根据 LSDB 得出最短路径生成树

    参见图 6. 17 实验 3 所得最短路径生成树
  - 由最短路径生成树和两类 LSA 导出了正确的路由表

    参见图 6. 16 实验 3 输出路由表信息
- 验证路由表转发的正确性：实验 4：电脑与路由多台验证
  - 实现路由转发功能

    相互之间能 ping 通

工作展望

由于时间和水平限制，目前设计和实现还存在很多不足之处，需要进一步研究和改进。主要包括如下的几个方面：

- 协议细节方面还有待完善

- 实验 4 有时会出现 bug，因时间有限，还没能调试成功。

- 初始配置文件可以去除
  - 初始配置文件可以通过系统调用读出网卡的相关参数得到，而本实验中我还是手动配置。
- 协议的扩展实现内容
  - 目前只是基于 Linux 实现基本的 OSPF 协议，在下面的工作中应该对其功能上进行进一步的扩展，比如使之支持多 area 或者是多种网络类型等等
  - 程序的头文件定义中已经留下这样的可扩展接口，方便进一步扩展

参考文献

[1]“计算机网络实验教程（第二版）”钱德沛主编 北航内部印刷 2011 年

[2]“计算机网络实验报告”北航内部印刷 2011 年

[3]TCP/IP 详解 (美)W.R.史蒂文斯(W.RichardStevens)著机械工业出版社

[4]RFC 2328

致 谢

首先，感谢张力军老师在一学期中对我的指导和鼓励。不论是课上的谆谆教诲或课下的循循善诱，张老师始终保持灿烂的微笑，让我感受到北航计算机学院老师独特的人格魅力和师者风范。张老师对每一个细节的严格要求与对每一个时间点的准确把握，让我能保证项目的进度，将压力转化为动力，并真真切切的实现了一个小型的实用系统。张老师给我的帮助与支持，让我能真正的从实践中踏入计算机网络的大门。

然后，感谢很多助教的辛苦工作与热心答疑。助教们牺牲自己的假期时间，在实验室和我一起完成一些网络实验，和我们进行研究和讨论，在技术细节上对我进行和指引。这里特别感谢袁园助教和袁永鑫助教，在网络实验机房时陪着我一起进行调试，平时也在很多方面对我进行指导。

其次，感谢张老师开设的网络的提高课程本身，这门课让我从实践中学习到计算机网络的协议的分析与实现。现在，我正在微软亚洲研究院（MSRA）无线网络组实习，从事网络相关的学习与研究。从张老师开设的这门课收获的许多知识与经验，让我有机会拿到这么好的机会。

最后，再对张老师和各位助教表示感谢，感谢你们一学期以来给予我的帮助和耐心教导，谢谢！