**09-网络路由实验**

范子墨

2021K8009929006

1. **实验内容**
   1. 实验一
      1. 基于已有代码框架，实现路由器生成和处理mOSPF Hello/LSU消息的相关操作，构建一致性链路状态数据库
      2. 运行实验
         1. 运行网络拓扑(topo.py)
         2. 在各个路由器节点上执行disable\_arp.sh, disable\_icmp.sh, disable\_ip\_forward.sh)，禁止协议栈的相应功能
         3. 运行./mospfd，使得各个节点生成一致的链路状态数据库
   2. 实验二
      1. 基于实验一，实现路由器计算路由表项的相关操作
      2. 运行实验
         1. 运行网络拓扑(topo.py)
         2. 在各个路由器节点上执行disable\_arp.sh, disable\_icmp.sh, disable\_ip\_forward.sh)，禁止协议栈的相应功能
         3. 运行./mospfd，使得各个节点生成一致的链路状态数据库
         4. 等待一段时间后，每个节点生成完整的路由表项
         5. 在节点h1上ping/traceroute节点h2
         6. 关掉某节点或链路，等一段时间后，再次用h1去traceroute节点h2
2. **总体设计思路**

基于上一个实验，对数据包进行进一步的分类处理。若目的地址与接口地址匹配，则增加判断上层协议是否是IPPROTO\_MOSPF字段，将其交给 handle\_mospf\_packet 函数进行处理。若目的地址与接口地址不匹配，则判断是否为等于多播 OSPF 所使用的 IP 地址 MOSPF\_ALLSPFRouters，若等于，则断言其为多播OSPF数据包，同样交由handle\_mospf\_packet 函数进行处理。如果目的 IP 地址既不等于接口 IP 地址，也不等于 MOSPF\_ALLSPFRouters，则说明这个数据包需要进行路由转发，将其交给 ip\_forward\_packet 函数进行进一步处理。这里需要自己将上个实验中对路由转发的处理代码复制到ip\_forward函数中。

在handle\_mospf\_packet函数中，首先检验该数据包是否有错误，而后根据mospf->type进行分类。如果是MOSPF\_TYPE\_HELLO类型，则为hello数据包，跳转至handle\_mospf\_hello。若是MOSPF\_TYPE\_LSU类型数据包，则调用handle\_mospf\_lsu函数处理。

handle\_mospf\_lsu会处理三类lsu消息，包括新的链路状态信息、原有状态信息的定期发送、已记录或过期的状态信息。根据数据包，对本路由器的链路状态数据库进行更新。而后由于洪泛链路，因此在处理好消息后，如果TTL值还大于0，向除本端口以外的端口转发消息，一层层传递下去。

与此同时，每个节点需要周期性发送MOSPF Hello（ hello-interval ：5秒）宣告自己的存在，同时周期性发送MOSPF LSU数据包。此外，要定期检查邻居列表和MOSPF数据库表。由于本次实验没有默认路由表，因此需要根据Dijkstra算法进行路由表的生成。即同时创造以下线程，同时运行。

pthread\_create(&hello, NULL, sending\_mospf\_hello\_thread, NULL);

    pthread\_create(&lsu, NULL, sending\_mospf\_lsu\_thread, NULL);

    pthread\_create(&nbr, NULL, checking\_nbr\_thread, NULL);

    pthread\_create(&db, NULL, checking\_database\_thread, NULL);

    pthread\_create(&rtable, NULL, generate\_rtable\_thread, NULL);

1. **生成和处理mOSPF Hello/LSU**



图表 1：mOSPF表头

1. 发送mOSPF Hello

遍历路由器的接口列表，最后将构造好的数据包通过接口发送给邻居，数据包赋值过程与上一个实验类似。需要注意由于mOSPF为多播，因此发包的目的 mac 地址为 01:00:5E:00:00:05，ip 地址为 224.0.0.5。每次循环前，需要先休眠一段时间，

            u8 dhost[ETH\_ALEN] = {0x01, 0x00, 0x5e, 0x00, 0x00, 0x05};

            eh->ether\_type = htons(ETH\_P\_IP);

            memcpy(eh->ether\_shost, iface->mac, ETH\_ALEN);

            memcpy(eh->ether\_dhost, dhost, ETH\_ALEN);



图表 2：Hello数据包

1. 处理mOSPF Hello

收到mOSPF Hello消息后，对数据包进行处理。首先遍历邻居列表，检查是否有邻居ID与收到的ID相匹配，若匹配，则设置found=1，表示该数据包处理完成，同时设置该条目的alive计数器为0。若未找到匹配，则表明邻居列表中需要新增条目，按照PPT上的数据结构进行构建即可，最后将该邻居添加到该接口的邻居列表中，由于信息发生变动，因此向其他邻居发送lsu信息。

typedef struct {

struct list\_head list;

u32 nbr\_id; // neighbor ID

u32 nbr\_ip; // neighbor IP

u32 nbr\_mask; // neighbor mask

u8 alive; // alive for #(seconds)

} mospf\_nbr\_t;

1. 定期检查邻居列表

首先遍历该路由器的所有接口，然后遍历该接口的所有邻居，将alive实践增加，如果该邻居已经超过15s没有更新过，则删除。

    if (nbr->alive > 3 \* iface->helloint)

    {

        list\_delete\_entry(&nbr->list);

        free(nbr);

        iface->num\_nbr--;

        sending\_mospf\_lsu();

    }

1. 生成mOSPF LSU

首先是在sending\_mospf\_lsu函数中定期发送lsu信息，即每40s跳转到sending\_mospf\_lse函数中。

在sending\_mospf\_lsu函数中，首先遍历路由器的接口，得到该路由器的总邻居数。而后创建数组lsa，填充每个接口的邻居信息，包括掩码、网络地址和路由器ID。

list\_for\_each\_entry (iface, &instance->iface\_list, list)

    {

        if (iface->num\_nbr == 0)

        {

            lsa\_array[i].mask = htonl(iface->mask);

            lsa\_array[i].network = htonl(iface->ip & iface->mask);

            lsa\_array[i].rid = 0;

            i++;

        }

        else

        {

            mospf\_nbr\_t \*ptr = NULL;

            list\_for\_each\_entry (ptr, &iface->nbr\_list, list)

            {

                lsa\_array[i].mask = htonl(ptr->nbr\_mask);

                lsa\_array[i].network = htonl(ptr->nbr\_ip & ptr->nbr\_mask);

                lsa\_array[i].rid = htonl(ptr->nbr\_id);

                i++;

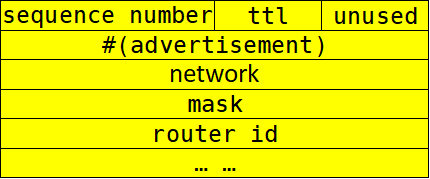
            }

        }

    }

而后更新该路由协议的序列号。

最后再次遍历该接路由器的所有邻居，将lsu数据包赋值并发给所有邻居。处理方式与hello数据包类似。



图表 3：lsu数据包

1. 处理mOSPF LSU

将受到的数据包拆分出ip首部、mospf首部、lsu部分和lsa部分。而后遍历数据链路库，如果存在匹配的路由器rid，则设置found=1，如果序列号大于原序列号，则更新当前路由器的所有邻居信息。如果没有找到匹配的条目，则创建一个新的数据库条目，填充RID、序列号、邻居数目，并为每个LSA信息分配内存并填充。

        mospf\_db\_entry\_t \*new\_db = (mospf\_db\_entry\_t \*)malloc(sizeof(mospf\_db\_entry\_t));

        new\_db->rid = ntohl(mospf\_header->rid);

        new\_db->seq = ntohs(lsu->seq);

        new\_db->nadv = ntohl(lsu->nadv);

        new\_db->alive = 0;

        new\_db->array = (struct mospf\_lsa\*)malloc(new\_db->nadv \* MOSPF\_LSA\_SIZE);

        for (int i = 0; i < new\_db->nadv; i ++ )

        {

            new\_db->array[i].network = ntohl(lsa[i].network);

            new\_db->array[i].mask = ntohl(lsa[i].mask);

            new\_db->array[i].rid = ntohl(lsa[i].rid);

        }

        list\_add\_tail(&new\_db->list, &mospf\_db);

将ttl-1，如果ttl还大于0，将该数据包更改ip首部和以太网首部后再次发出。

    iph->saddr = htonl(iface->ip);

    iph->daddr = htonl(nbr->nbr\_ip);

    mospfh->checksum = mospf\_checksum(mospfh);

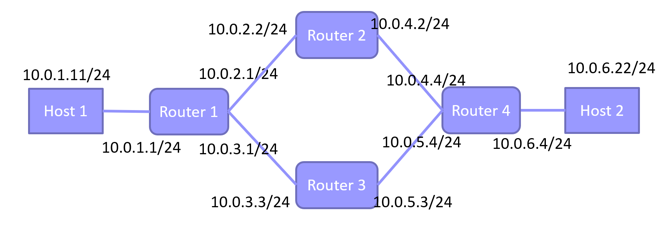
    iph->checksum = ip\_checksum(iph);

    ip\_send\_packet(forward, len);

1. 定期检查mOSPF数据库表

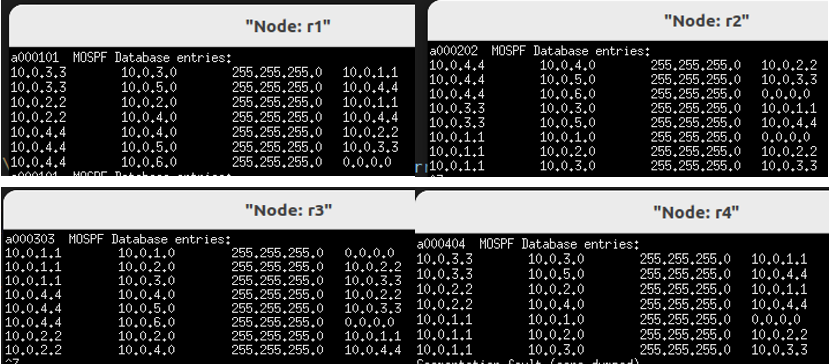
遍历数据链路库中的所有条目，如果数据库中一个节点的链路状态超过40秒未更新时，表明该节点已失效，将对应条目删除。

1. 结果



图表 4：拓扑

可以看出根据上图，数据链路库生成正确。



图表 5：链路状态数据库

1. **实现路由器计算路由表项的相关操作**
2. 生成路由表

首先初始化一个图数据结构，在init\_graph函数中根据链路状态数据库中的信息建立路由器中的拓扑图，而后执行Dijkstra算法，找到最短路径的下一个节点，将其标记为已访问。

而后遍历路由表，如果路由表项有下一跳地址，则移除该路由表项，即每个路由器的路由表只有其端口所连接的局域网。

对于每个节点，会根据 Dijkstra 算法匹配到前序节点。用递归的方式前递可以找到对于本节点而言，每一个其他的节点的下一条节点是多少，并以此更新路由表，从而确定到其他网络的下一跳网关地址、源节点的转发端口。

1. Dijkstra算法

伪代码在讲义中已经给出，实现思路为首先初始化 dist 数组（dist[i] 表示从起点到节点 i 的最短距离），prev 数组（prev[i] 表示节点 i 的前驱节点），visited 数组（标记节点是否已经被访问）。使用外循环遍历每一个路由器，在每次循环中，找到距离起点距离最短且未被访问的节点，这里使用min\_dist函数实现。而后遍历所有节点 v，如果节点 v 未被访问且从起点经过节点 u到达节点 v 的距离小于当前记录的 dist[v] 距离，同时存在一条边，则更新 dist[v]和prev[v]，使得dist[v]记录更短的距离，同时将 prev[v] 设置为节点u。

for(int i = 0; i < 4; i++)

    {

        dist[i] = INT8\_MAX;

        prev[i] = -1;

        visited[i] = 0;

    }

    dist[0] = 0;

    for(int i = 0; i < num; i++)

    {

        int u = min\_dist(dist, visited,num);

        visited[u] = 1;

        for (int v = 0; v < num; v++)

        {

            if (visited[v] == 0 && dist[u] + graph[u][v] < dist[v] && graph[u][v] > 0)

            {

                dist[v] = dist[u] + graph[u][v];

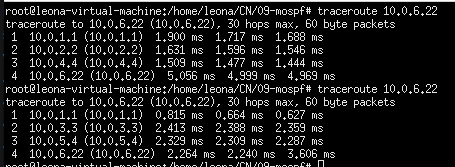
                prev[v] = u;

            }

        }

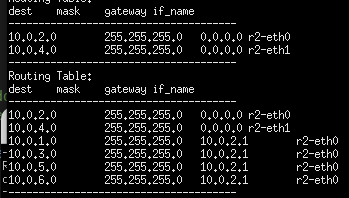
    }

2、结果



图表 6：h1和h2路径

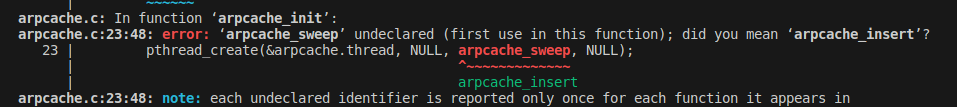
h1 节点对 h2 节点执行 traceroute 操作，可见路径查找正常。执行` link r2 r4 down `命令，取消 r2 和 r4节点间的链路。继续等待约 40s，重新执行 h1 到 h2 的 traceroute 操作，可见路径发生改变且符合理论情况。



图表 7：路由器r2路由表

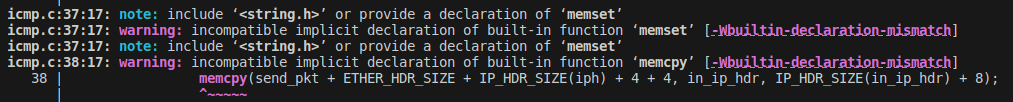
如图可见路由器r2在断开r2和r4的连接后路由表从只有两个局域网去向变为正常连接至所有局域网。

1. **遇到的问题**
2. 有一个函数未声明

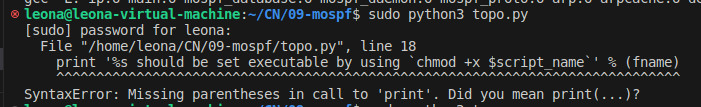


图表 8：函数未声明

1. 和上一个实验相同，编译代码时会出现很多warning，以及似乎拓扑文件的一些书写格式和python3不兼容

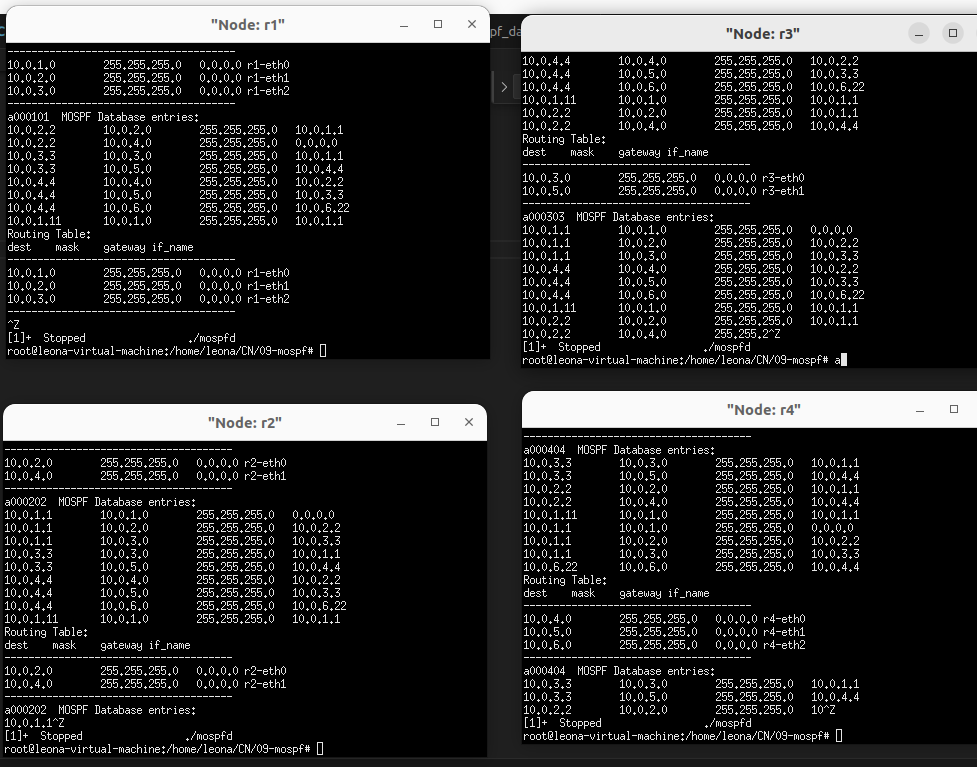


图表 9：warning



图表 10：部分语法错误

1. 发现链路状态数据库可以正常生成但是ping不对

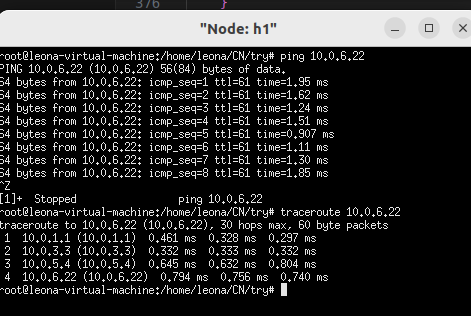


图表 11：链路状态数据库和路由表

链路状态数据库没有问题，但是和数据库一同输出rtable时，rtable没有成功生成，只有端口所对网络的信息。因此考虑时路由生成函数generate\_rtable\_thread，在该函数中输出rtable，结果发现各路由器端点没有输出，因此该函数可能根本没有运行，查找该函数相关语句，发现确实没有运行。检查mospf\_run函数，发现没有创建线程运行该函数。

pthread\_create(&rtable, NULL, generate\_rtable\_thread, NULL);

添加上后，成功运行。



图表 12：输出正常