网络路由实验报告

李昊宸

2017K8009929044

（一）网络路由机制实现

一、实验内容

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。

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. 基于 1，实现路由器计算路由表项的相关操作。

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

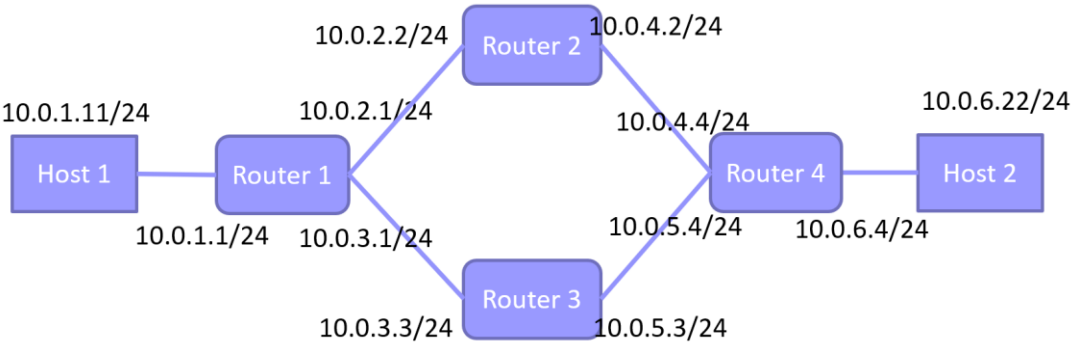
5）在节点 h1 上 ping/traceroute 节点 h2

6）关掉某节点或链路，等一段时间后，再次用 h1 去 traceroute 节点 h2

二、实验流程

1. 搭建实验环境

include：相关头文件
scripts：禁止协议栈的数据包处理
main.c： 路由器的代码实现，编译后在路由器结点上运行
ip.c：处理 IP 数据包，包括转发
libipstack.a：上次实验实现的 ARP 和路由表相关函数编译出的库
mospf_daemon.c：处理 Hello、LSU 数据包
mospf_proto.c：mOSPF 协议函数
mospf_database.c：链路状态数据库相关函数
topo.py：实现如下图的四节点拓扑



图一 六节点网络拓扑

本次实验，在上周 ARP 缓存的基础上，增加了构建链路状态数据库的 mOSPF 协议的支持。

OSPF：Open Shortest Path First，一种内部网关协议，用于在单一自治系统（autonomous system，AS）内决策路由。它是对链路状态路由协议的一种实现，隶属内部网关协议（IGP）。特殊说明的是，只有**路由器**参与链路状态数据库的构建。

链路：路由器接口的另一种说法。因此 OSPF 也称为接口状态路由协议。OSPF 通过路由器之间通告网络接口的状态来建立链路状态数据库，生成最短路径树，每个 OSPF 路由器使用这些最短路径构造路由表。

OSPF 路由协议是一种典型的链路状态（Link-state）的路由协议，一般用于同一个路由域内。在这里，路由域是指一个自治系统（Autonomous System），即 AS，它是指一组通过统一的路由政策或路由协议互相交换路由信息的网络。在这个 AS 中，所有的 OSPF 路由器都维护一个相同的描述这个 AS 结构的数据库，该数据库中存放的是路由域中相应链路的状态信息，OSPF 路由器正是通过这个数据库计算出其 OSPF 路由表的。

作为一种链路状态的路由协议，OSPF 将链路状态组播数据 LSA（Link State Advertisement）传送给在某一区域内的所有路由器，这一点与距离矢量路由协议不同。运行距离矢量路由协议的路由器是将部分或全部的路由表传递给与其相邻的路由器

本次实验用到的协议为组播扩展 OSPF（MOSPF）。它在原 OSPF 第二版本的基础上作了增强使之支持 IP 组播路由。它与 OSPFv2 的区别在于：

1）OSPFv2 的 protocol number 为 89，而 mOSPF 为 90。

2）mOSPF 对数据包格式进行了适当简化。

3）OSPFv2 基于可靠洪泛：收到 LSU 数据包后需要回复 ACK。

4）OSPFv2 有更多的消息类型。例如，链路状态数据库 Summary。

5）OSPFv2 有安全认证机制（鉴别）

mOSPF 协议格式：

mOSPF Header：

version	type	length
router id		
area id		
checksum		padding

IP protocol number：90

version：2

type：

Hello：1
LSU ：4

length：mOSPF 消息的长度（首部+内容）
router id：生成本消息的路由器 id
area id：传播区域，本实验中设置为 0。
checksum：校验和（校验首部+内容）
padding：对齐，0。

mOSPF Hello：

mask	
hello interval	padding

mask:生成该 Hello 消息的端口的网段掩码
hello interval：两次发送 hello 之间间隔的时间
padding：对齐，0。

mOSPF LSU：

sequence number	ttl	unused
#(advertisement)		
network		
mask		
router id		
… …		

sequence number：序列号
ttl：生存时间
unused：未使用
advertisement：该路由器节点的邻居路由器数（如果某个端口没有邻居，按 1 算）
network：邻居所在网段
mask：邻居所在网段的掩码
router id：邻居路由器 id

链路状态数据库算法：

邻居发现：

每个节点周期性（ hello-interval ：5 秒）宣告自己的存在，发送 mOSPF Hello 消息，包括节点 ID, 端口的子网掩码。目的 IP 地址为 224.0.0.5，目的 MAC 地址为 01:00:5E:00:00:05。

节点收到 mOSPF Hello 消息后，对消息内容进行解析：如果发送该消息的节点不在邻居列表中，添加至邻居列表；如果已存在，更新其生存时间。

同时，有另一个进程执行邻居列表老化操作（Timeout）：如果列表中的节点在 3* hello-interval 时间内未更新，则将其删除。

链路状态扩散和更新：

生成并洪泛链路状态：当节点邻居列表发生变动时，或超过 lsu interval (30

秒)未发送过链路状态信息时，向每个邻居节点发送链路状态信息，包含该节点 ID (mOSPF Header)、邻居节点 ID、网络和掩码 (mOSPF LSU)。当端口没有相邻路由器时，也要表达该网络，邻居节点 ID 为 0。序列号(sequence number)，每次生成链路状态信息时加 1。目的 IP 地址为邻居节点相应端口的 IP 地址，目的 MAC 地址为该端口的 MAC 地址

收到链路状态信息后，如果之前未收到该节点的链路状态信息，或者该信息的序列号更大，则更新链路状态数据库。随后将 TTL 减 1，如果 TTL 值大于 0，则向除该端口以外的端口转发该消息。

同时，有另一个进程执行数据库节点老化操作：当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

路由计算：不同节点经过交换链路状态信息，获得一致性链路状态数据库；每个节点独立计算路由条目，从而保证网络的可达性。

具体算法比较直观：

1） 每个节点将自己的链路状态数据库和邻居节点库抽象成拓扑图，相连的两个路由器之间有一条边，边的值即为路径开销。
2） 每个节点以自己为根，利用 Dijkstra 算法计算出到其他每个节点的最短路径，并在期间计算出每个节点最短路径的前一跳节点，并将其记录下来。
3） 通过对记录的前一条节点表的修改，可以得到通往每个节点最短路径的第一跳，根据该信息生成网络路由。

路由计算与最短路径算法的不同：

	最短路径算法	路由计算
目的	计算到每个节点的路径	计算到每个网络的路由
结果形式	路径长度和前一跳节点	下一跳网关和转发端口

也就是说，最短路径算法给出了从根节点到达每个节点的路径长度和前一跳节点，那么有已知的前一跳节点，可以递归向上确定从根路由器出发的第一跳节点，也就是下一跳网关和转发端口。而每个节点的链路状态信息又被保存在根节点的链路状态数据库中，于是可以查找前往任何一个网段的下一跳网关和转发端口，进而生成路由表。

从**最短路径到路由表项**算法：

按照路径长度从小到大依次遍历每个节点，对于节点端口对应的每个网络，如果该网络对应的路由未被计算过，就查找从源节点到该节点的下一跳节点，确定下一跳网关地址、源节点的转发端口。

2. 启动脚本

1)六节点拓扑链路状态数据库建立测试

```
make all
sudo python topo.py
mininet> xterm h1 h2 r1 r2 r3 r4
r1# ./mospfd
r2# ./mospfd
r3# ./mospfd
r4# ./mospfd
mininet> quit
```

2)六节点拓扑路由测试

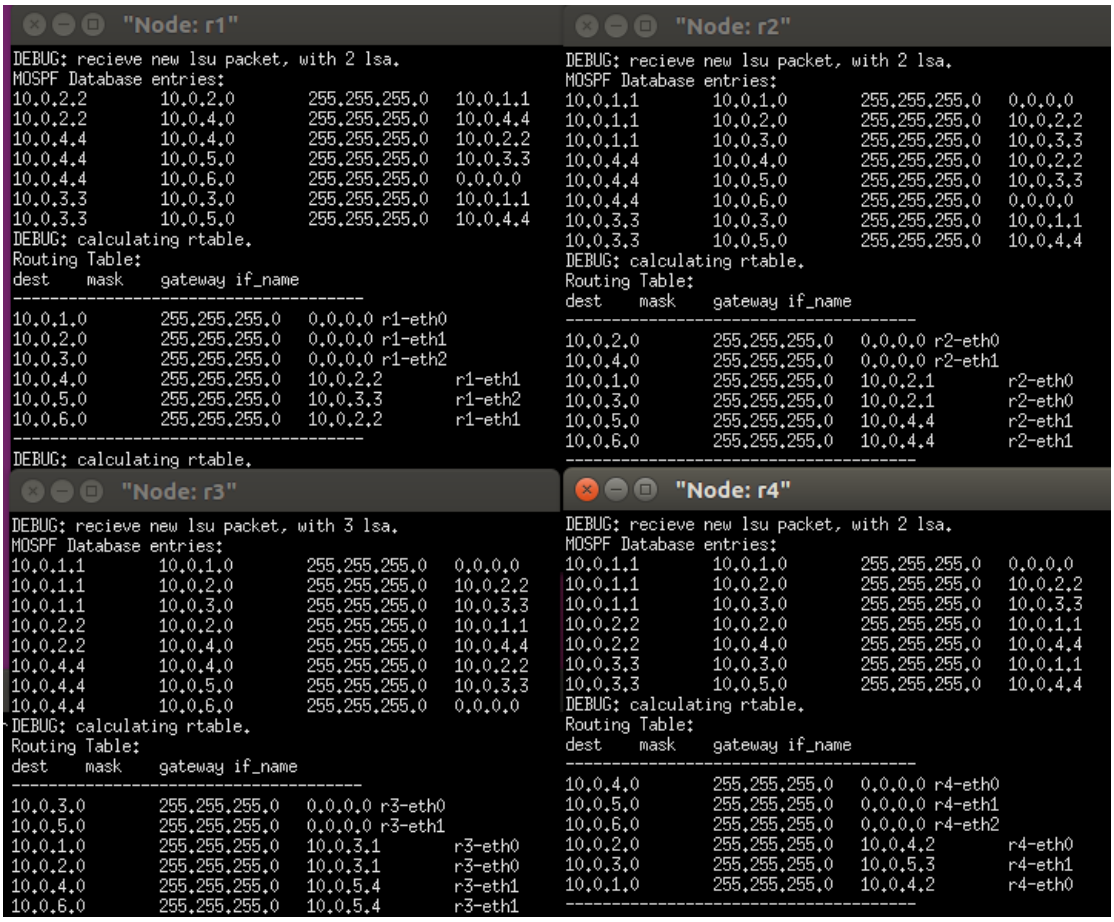
```
make all
sudo python topo.py
mininet> xterm h1 h2 r1 r2 r3 r4
r1# ./mospfd
r2# ./mospfd
r3# ./mospfd
r4# ./mospfd

等待一段时间
h1# traceroute 10.0.6.22 -m 6
mininet> link r2 r4 down

等待一段时间
h1# traceroute 10.0.6.22 -m 6
mininet> quit
```

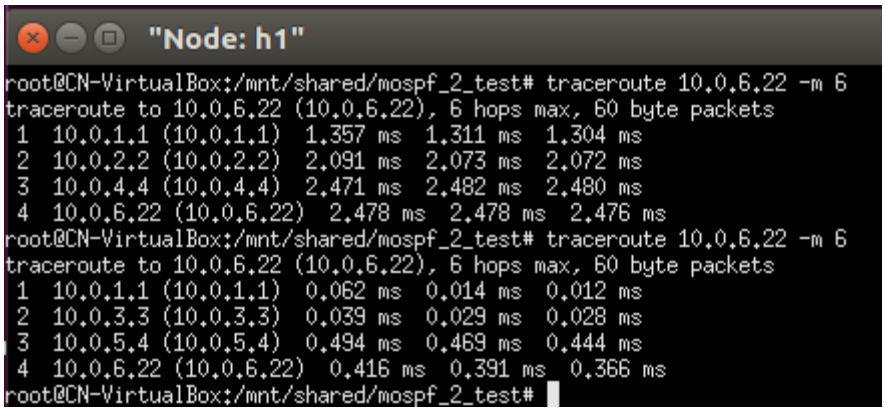
三、实验结果及分析

1. 实验结果



图三 六节点拓扑链路状态数据库和路由表结果

可以看到，每个节点终端都打印出了链路状态数据库，和基于数据库和内核信息构建的路由表。



图四 六节点改变拓扑前后路由结果

在关闭 r2 与 r4 之间的 link 后，链路拓扑重构，重新生成了新的路由表，并且 traceroute 得到了不同的路由结果，证明此次实验满足要求。

（二）实验代码详解

本次实验代码过长，不再赘述代码的内容。

一、 ip.c：处理收到的 IP 数据包

（1）void handle_ip_packet(iface_info_t *iface, char *packet, int len)：处理 IP 数据包

解析 IP 报文的 IP 首部。如果目的 IP 是本端口 IP 的话，判断协议类型：

如果是 ICMP 报文，就检查 ICMP type，如果是 ECHOREQUEST 就应答 ICMP 报文，否则丢弃。

如果是 MOSPF 报文，就调用 handle_mospf_packet 进行处理。

如果目的 IP 不是本端口 IP，就检查目的 IP 是不是 224.0.0.5，如果是，就调用 handle_mospf_packet 进行处理。

如果目的 IP 不是本端口 IP，并且目的 IP 不是 224.0.0.5，就将该包转发。

二、 mospf_daemon.c:发送 ARP 请求和应答

（1）void *sending_mospf_hello_thread(void *param)：周期性发送 Hello 报文的进程

首先设置报文大小为 ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE。

接下来是一个每 5s 执行一次的循环：

获取 mospf 互斥锁，遍历每个端口执行以下操作：

分配报文大小的空间，进行初始化：

以太网首部：目的 mac 地址设置为 01:00:5E:00:00:05，源 mac 地址设置为端口 mac 地址，type 设置为 IP。

IP 首部：调用上次实验中用到的 ip_init_hdr 初始化首部，其中目的 IP 地址为 224.0.0.5，IP 协议号为 90。

设置 mospf 首部：version 为 2，type 为 1（Hello），message 长度为 htons（MOSPF_HDR_SIZE + MOSPF_HELLO_SIZE），router id 为 instance 中记录的 router_id（htonl（instance->router_id）），area id 为 0.0.0.0，padding 为 0。

设置 Hello 段：mask 为该端口的掩码（htonl（iface->mask）），周期设置为 htons（5），padding 设置为 0。

设置校验和。需要注意的是，IP 首部校验和只校验首部，mospf 校验和要校验 mospf 首部和 message。

最后释放 mospf 互斥锁，睡眠 5s。

（2）void *sending_mospf_lsu_thread(void *param)：周期性发送 LSU 报文的进程

循环：首先获取 mospf 互斥锁，然后调用 sending_mospf_lsu_func，然后释放互

斥锁，睡眠 30s。

（3）void sending_mospf_lsu_func(void *param)：发送 LSU 报文

首先设置报文非数据区 size_without_lsu_data 大小为 ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_SIZE + MOSPF_LSU_SIZE。

接下来调用 list_for_each_entry,遍历所有端口,计算该节点所有的邻居数目 num（此处注意，一个端口如果记录的 num_nbr 非 0，就将 num_nbr 加到求和变量 num 上；如果为 0，就将求和变量 num 加 1，因为即使某端口没有相邻路由器，该网络也是客观存在的，也需要表达该网络）

分配一个 mospf_lsa 型数组 lsu_data，空间大小为 num * MOSPF_LSA_SIZE。

再次遍历所有端口，对于 num_nbr 非 0 的端口，遍历其邻居链表 nbr_list，将每一个邻居的所在网段，网段掩码和路由 id 记录到 lsu_data 中（htonl）；对于 num_nbr 为 0 的端口，所在网段记录为 htonl（iface->IP & iface->mask），掩码为 htonl（iface->mask），router id 为 0。

再次遍历所有端口：

如果该端口 num_nbr 非 0,也就是有相邻路由器,就要向这些路由器发送 LSU 消息：设置报文大小为 size_without_lsu_data + num * MOSPF_LSA_SIZE。遍历 nbr_list 的每一个节点：

分配报文大小的空间，进行初始化：

以太网首部：源 mac 地址设置为端口 mac 地址，type 设置为 IP。目的 mac 地址会在 ip_send_packet 时设置。

IP 首部：调用上次实验中用到的 ip_init_hdr 初始化首部，其中目的 IP 地址为当前遍历到的邻居路由器的 IP,, IP 协议号为 90。

设置 mospf 首部：version 为 2，type 为 4（LSU），message 长度为 htons（MOSPF_HDR_SIZE + MOSPF_LSU_SIZE + num * MOSPF_LSA_SIZE），router id 为 instance 中记录的 router_id(htonl(instance-)router_id)),area id 为 0.0.0.0，padding 为 0。

设置 LSU 段首部：seq 为 htons（instance->sequence_num），unused 为 0，ttl 为 MOSPF_MAX_LSU_TTL，nadv 为 htonl（num）。

设置 LSU 数据段：调用 memcpy 将 lsu_data 拷贝到 LSU 首部的后面。

设置校验和，最后调用 ip_send_packet 将 packet 发送出去。

如果该端口 num_nbr 为 0，就不用发送了，因为非路由器节点不参与链路状态数据库的构建。

（4）void handle_mospf_hello(iface_info_t *iface, const char *packet, int len)：处理收到的 Hello 消息

首先解析 IP 首部和 mospf 首部，获取 mospf 互斥锁。

遍历收到消息端口的 nbr_list，寻找和 mospf 首部中 router id 相同的邻居。如果找到了，说明之前该路由器已经向本路由器发送过 hello 消息，就将该 nbr 节点的 alive 分量修改为 0，重新开始计时。

如果没找到，就要向该端口的邻居链表新增该路由器信息。先将 num_nbr 加 1，然

后解析 mospf 信息到 hello_message，分配一个 mospf_nbr_t 节点，修改邻居 ID 为 mospf 首部中记录的 router id（ntohl），邻居 IP 为 IP 首部中记录的 saddr（ntohl），网段掩码为 ntohl（hello_message->mask），alive 为 0。调用 list_add_tail 将该节点添加到 nbr_list 的尾部。此时邻居列表得到更新，就调用 sending_mospf_lsu_func 函数向每个邻居路由器发送 LSU 信息。随后调用 calculating_rtable_func，重新计算最短路径，重新生成路由表。

最后释放互斥锁。

(5)void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)：处理收到的 LSU 消息

首先获取 mospf_database_lock 互斥锁（此处设置两个锁的原因为，hello 的发送与应答与处理收到的 lsu 之间数据是独立的，互不影响），解析 IP 首部、mospf 首部和 LSU 首部。遍历数据库中的每一项：

如果当前遍历到的数据库节点记录的 router id 与收到的 LSU 消息中经过 ntohl 转换的 mospf 首部中的 router id 相同，说明之前该路由器发送过的 LSU 消息经扩散后被本路由器收到过。比较序列号，如果收到的 LSU 序列号较大，就需要对本机数据库进行更新：更新该数据库节点的序列号，生存期 alive 重置为 0，nadv 为 LSU 首部中的 nadv（ntohl），array 首先调用 realloc 进行扩容，大小增至 num * MOSPF_LSA_SIZE，随后将 LSU 中的数据部分拷贝到 array 数组下（这里有一个 trick，如果新的 num 比原来的 num 小，realloc 会使原来 array 数组的部分数据丢失。但是，每一次 array 都是重置的，所以丢失的数据都是无用数据）。如果序列号相等或者较小，就不需要进行上述操作。然后调用 calculating_rtable_func(NULL)重新计算路由表。

如果遍历全部数据库节点都没有找到匹配的节点，说明该路由器发送的 LSU 消息第一次被本路由器收到。新分配一个 mospf_db_entry_t 型的数据库节点，router id 设置为 mospf 首部中的 router id（ntohl），序列号设置为 LSU 首部中的序列号，生存期 alive 设置为 0，nadv 为 LSU 首部中的 nadv（ntohl），array 用 malloc 分配 num * MOSPF_LSA_SIZE 的空间，随后将 LSU 中的数据部分拷贝到 array 数组下。之后调用宏 list_add_tail 将该节点增添到数据库链表上，调用 calculating_rtable_func(NULL)重新计算路由表。为了检查运行的正确性，仿照 print_rtable 的方式，调用宏增加了打印数据库的功能。

释放互斥锁，此时对数据库的更新结束。

对 LSH 首部中的 ttl 字段减 1，如果 ttl 大于 0，说明生存期未耗尽，需要继续进行转发。遍历所有端口：

如果某个端口的 num_nbr 非 0，并且不是收到该 LSU 消息的端口，就要从该端口向外转发该包。复制一个与收到 LSU 报文相同的包，修改以太网首部的源 mac 地址为该端口的 mac 地址。遍历 nbr_list，如果 nbr_id 与 mospf 首部中的 router id（ntohl）相同，就跳过（因为可能该路由器同时在本路由器不同端口连接的网络上）；否则，就修改 mospf 校验和，修改 IP 首部源 IP 为 htonl（iface->ip），目的 IP 为 htonl（nbr->nbr_ip），修改 IP 首部校验和，调用 ip_send_packet 将包发送出去。

（6） void handle_mospf_packet(iface_info_t *iface, char

*packet, int len)：处理收到的 mospf 协议报文

首先做一些基本检查，检查 mospf_verssion，校验和以及作用域。

检查无误后，解析 mospf 协议类型，如果是 hello，就调用 handle_mospf_hello；如果是 LSU，就调用 handle_mospf_lsu，否则 log 错误信息。

（7）void *checking_db_thread(void *param)：检查数据库节点失效进程

该程序通过 sleep（1）控制，每 1 秒执行一次：

首先获取 mpspf_database_lock 互斥锁，因为要对数据库进行访问和修改。调用宏 list_for_each_entry_safe 对数据库 mospf_db 进行安全遍历：

将生存期 alive 加 1。如果 alive 超过生存时间 MOSPF_DATABASE_TIMEOUT（40s），需要将该数据中记录的该表项删除，以及路由表中相关表项删除，然后重新计算路由表：首先调用最长前缀匹配查找到该路由器 router id 的转发表项，得到下一跳网关 gw，调用宏 list_for_each_entry_safe 安全遍历路由表，如果 gw 不为 0，且某一个表项的下一跳网关为 gw，就调用宏 remove_rt_entry 将该表项删除（这么处理比较方便）。结束对路由表的遍历后，调用宏 list_delete_entry 将该节点从数据库链表上删除，释放该节点，调用 calculating_rtable_func 重新计算路由表。

数据库遍历结束后，释放互斥锁，睡眠 1s。

（8）void init_graph()：初始化拓扑图

函数在计算路由表时使用。

用 router_list[router_num]记录拓扑中每个路由器的 router id，将记录拓扑信息的二维数组 graph 初始化为全 0。

初始化数组遍历 i 为 1。

之后调用 list_for_each_entry 遍历每个端口：

对于一个端口，遍历其邻居节点链表：

router_list[i]设置为 nbr_id,graph[0][i]和 graph[i][0]都设置为 1,i 自增 1。

需要注意的是，访问 nbr 的操作应该申请互斥锁 mospf，但是为了无锁化版本的 calculating_rtable_func 的调用，这里不再加锁。

该初始化将所有邻居路由拓扑加入到拓扑表中。

（9）int min_dist(u8 *dist, int *visited)：查找距离已访问节点集合最近的节点

函数在计算路由表时使用。用一维数组 visited[router_num]记录节点是否被访问过。用一维数组 dist[router_num]记录通过目前已访问的节点集合到达的其他节点的最短距离。初始化距离变量_min 为 255。

外层循环 i 遍历 visit，如果节点 i 被访问过，就进入内层循环：

内层循环 j 遍历 visit，如果节点 j 没被访问过，且 graph[i][j]>0，并且 graph[i][j] + dist[i] < _min，就更新_min 的值为 graph[i][j] + dist[i]，记录 j

的值为 result。

循环结束后，记录 dist[result]的值为_min；将 result 返回。

因此，函数会返回距离已经访问过的节点集合最近的节点。

（10）void Dijkstra(int *prev)：生成最短路径，prev 数组返回目标节点最短路径的上一跳

首先初始化 dist 数组为 255，dist[0]为 0，visited 数组为 0，visited[0]为 1。

随后外层循环 i 遍历所有节点：

调用 min_dist 返回最近节点 u，visited[u]设置为 1，进入内层循环 j 遍历所有节点：

如果 visited[j]等于 0,graph[u][j] > 0,且 dist[u] + graph[u][j] < dist[j]，就将 dist[j]的值赋为 dist[u] + graph[u][j]，并且设置 u 的上一跳 prev[j]为 u。

因此，函数会完成最短路径的计算，并修改 prev[d]为当前路由器到编号为 d 的路由器的最短路径的上一跳路由器编号。

（11）void *calculating_rtable_thread(void *param)：周期性重新计算路由表进程

全部代码在一个由 sleep（10）控制的 while 循环中，每 10s 执行一次。

首先获取 mospf 互斥锁，然后调用 init_graph 初始化拓扑表，然后释放 mospf 互斥锁。

再获取 mospf_database_lock 互斥锁。遍历数据库节点 db_entry：

内层循环以 i=1 开始，查找 router_list[i]与 db_entry 的 route id 的关系。如果 route_list[i]等于 0 或者等于当前数据库节点的 route id，就跳出内层循环。

跳出/结束循环后，如果 route_list[i]为 0，就将 db_entry->rid 的值赋给 route_list[i]。

遍历结束后，如果链路数据洪泛达到饱和，route_list 应该会被填满。接下来补充拓扑表。遍历数据库节点：

遍历 router_list，找到与节点 route id 相同的 router_list[t1]。如果没找到这样的 t1，就 continue。遍历节点的 array 数组，如果 array[k]的 route id 为 0，说明该端口不连接任何路由器，就跳过；然后再次遍历 router_list，找到与 array[k]的 route id 相同的 router_list[t2]。如果没找到这样的 t2，就 continue。设置 graph[t1][t2] = 1，graph[t2][t1] = 1。

遍历结束后，拓扑表构建完成，释放互斥锁。

设置好 prev 的初始值，调用函数 Dijkstra，返回一个 prev 数组。考虑到要构建路由表，对于任何一个节点，路由表中只需要保存从本路由器到该节点最短路径的下一跳即可。所以对 prev 做一下修改，如果 prev[t1]不为 0，且 prev[prev[t1]]也不为 0，那么可以修改 prev[t1]的值为 prev[prev[t1]]，重复以上修改，直到 prev[prev[t1]]为 0 为止。这样一来，prev[t1]保存的是到编号为 t1 的路由器的下一跳路由器编号。

给路由器表项设置一个有效信号 valid，遍历路由表项，如果表项的下一跳网关 gw 为 0，说明该表项记录的设备与本路由器直接相连，该表项从内核中导入，就设置 valid 为 1。否则，设置 valid 为 0。也就是说，每次计算路由表，都是重新计算非内核导入

项。

接下来从 t1=1 开始，遍历除了本路由器外的其他路由器。用 rid 记录 router_list[t1]，gw_rid 记录从本地路由器到遍历当前路由器的下一跳路由器：如果该路由器是本路由器的邻居，那么 gw_rid 记录的是该路由器的 router id，否则记录的是下一跳路由器的 router id。二重遍历所有端口下的所有邻居，必然会找到某个端口下记录的某个邻居的 router id 与 gw_rid 相等，找到后停止。此时 iface 为该端口，nbr 为该邻居。

找到了下一跳路由器，接下来的内容是将数据库中由该路由器发来的 LSU 消息，并将消息内容加入路由表，新加入的项的下一跳网关就是下一跳路由器的对应端口。遍历数据库，找到 router id 相同的数据库节点。遍历该节点下的 array 数组：

遍历路由表，如果路由表某一项的网段号 dest 与 array[t1].network 相同，并且 gw 与 nbr->nbr_ip 不同（即新计算出的下一跳网关），并且 valid 项为 0，那么该项属于需要更新的非内核导入表项，调用 remove_rt_entry 将该表项从路由表中删除。接下来，如果 array.[t1]的掩码为 0，说明该数据无效，网段不存在，就 continue。否则就调用 new_rt_entry (db_entry->array[t1].network, db_entry->array[t1].mask, nbr->nbr_ip, iface)新建一个表项，并调用 add_rt_entry 将其加入路由表。

如果路由表某一项的网段号 dest 与 array[t1].network 相同，并且 gw 与 nbr->nbr_ip 相同，说明该表项不需要更新，就将 valid 置 1。

如果路由表没有网段号 dest 与 array[t1].network 相同的项，说明该项在路由表中不存在。接下来，如果 array.[t1]的掩码为 0，说明该数据无效，网段不存在，就 continue 。否则就调用 new_rt_entry (db_entry->array[t1].network, db_entry->array[t1].mask, nbr->nbr_ip, iface) 新建一个表项，并调用 add_rt_entry 将其加入路由表。

全部结束后，打印路由表，sleep（10）。

（12）void calculating_rtable_func(void *param)：无锁化计算路由表进程

该函数与（11）基本一致，唯一区别在于去掉锁，方便其他有锁进程调用。