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

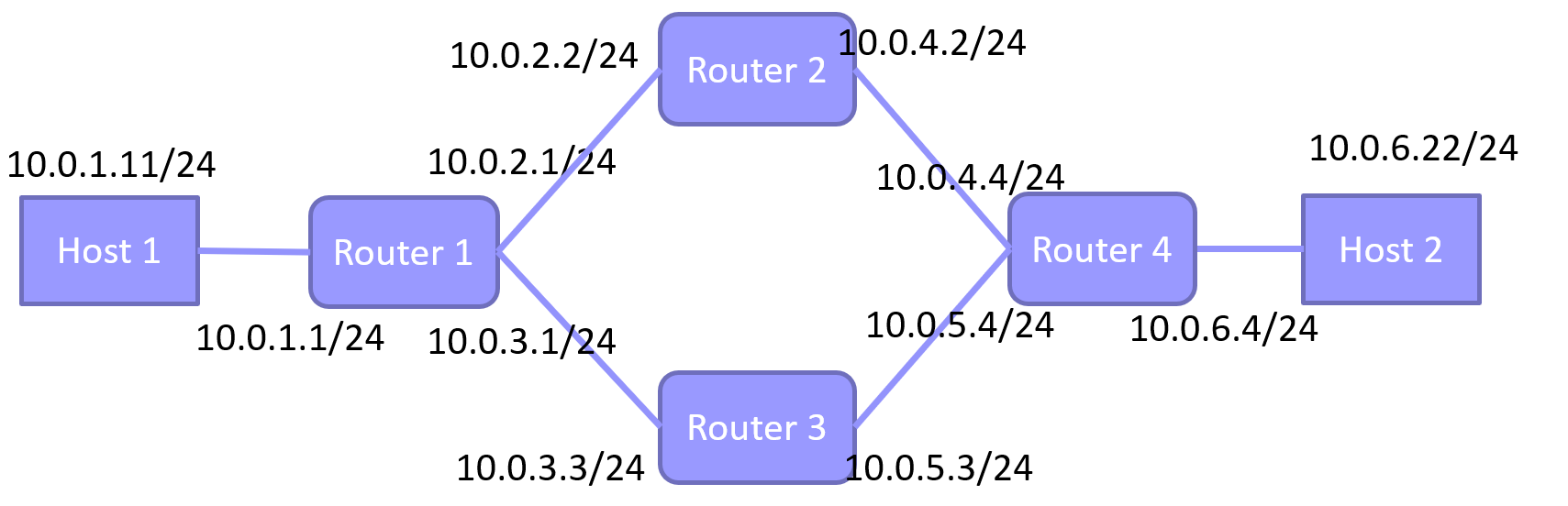
libipstack.a：上次实验实现的ARP和路由表相关函数编译出的库

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：



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：

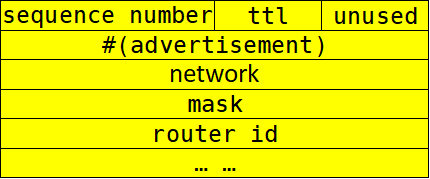


mask:生成该Hello消息的端口的网段掩码

hello interval：两次发送hello之间间隔的时间

padding：对齐，0。

mOSPF LSU：



sequence number：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

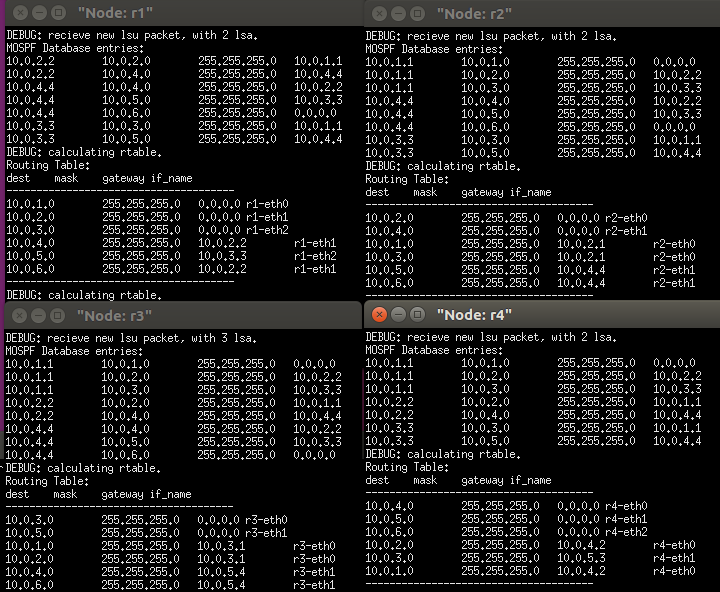
等待一段时间

h1# traceroute 10.0.6.22 -m 6

mininet> quit

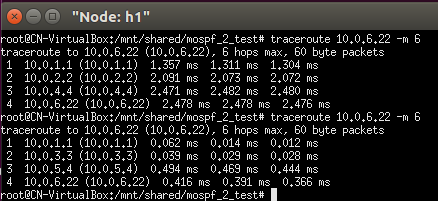
三、实验结果及分析

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。

1. 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。

1. 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。