RIP路由协议实现实验报告

1. 路由算法基础知识

1.1. 路由算法概述

1️⃣什么是路由算法：生成路由表的算法，路由表控制分组转发

2️⃣以是否能更具信息量/拓扑自适应调整，分为：

1. 静态/非自适应路由选择：手动搭建每条路由，多用于小网络，简单开销小
2. 动态/自适应路由选择：多用于复杂网络，复杂开销大，又分为：
    - 链路状态路由算法(LS)：具全局性，维护一个全局的拓扑图
    - 距离-向量路由算法(DV)：具有分布性，无需维护一个全局的拓扑图

3️⃣算法数据结构：用图，结点标识路由器，线表示链路

1.2. 链路状态路由算法

1️⃣算法概述：主动测试所有邻结点连接状态，定期传播链路状态给其他点，是的人每个系欸但那都有完全网络拓扑信息

2️⃣算法核心：Dijikstra算法，这个自不必多说

3️⃣其他

1. 算法改进：引入优先队列，复杂度可$O(n^2) \rightarrow O(n\log n)$
2. 算法缺点：存在震荡，会有随即延迟

1.3. 距离-向量路由算法

1.3.1. 算法特征

1️⃣分布式：每个结点从邻居获得信息→计算→将结果发给邻居

2️⃣迭代+自终止：重复上述过程直到邻居无更多信息要交换，算法结束

3️⃣异步：所有节点迭代的步伐不必一致

1.3.2. 算法思想&伪代码

1️⃣Bellman-Ford方程：$d_x(\mathrm{y}) = min_v\{c(x,v) + d_v(\mathrm{y})\}$

1. 符号：$d_x(\mathrm{y})$是$x \rightarrow y$路径最小开销，$c(x,v)$是x到其某一邻居v的路径
2. 思想：$x \rightarrow y$的最短路径，一定要经过邻居v中的个，遍历所有邻居就有可能求得

2️⃣结点x的距离向量：$\boldsymbol{D}_x = [\boldsymbol{D}_x(y) : y \in N]$

1. y：除x以外的其他所有节点之一
2. $\boldsymbol{D}_x(y)$：结点x到其他结点y的开销估计
3. $\boldsymbol{D}_x$：即$[\boldsymbol{D}_x(y_1), \boldsymbol{D}_x(y_2), \ldots\ldots]$

3️⃣结点x维护的信息：所有邻居的$c(x,v)$，自生的距离向量$\boldsymbol{D}_x$，所有邻居的距离向量$\boldsymbol{D}_v$

4️⃣ 算法操作

1. 每个节点周期性地，向每个邻居发送其距离向量副本
2. 当x收到v新的距离向量，则用Bellman-Ford方程更新距离向量

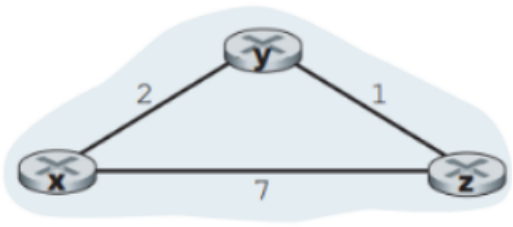
$$\boldsymbol{D}_x(\mathrm{y}) = min_v\{c(x,v) + \boldsymbol{D}_v(\mathrm{y})\}, y \in N$$

3. 如果$\boldsymbol{D}_x$因此改变，则向所有邻居立即更新其距离向量
4. 重复下去，最终迭代得到最低路径开销

5️⃣ 算法伪代码

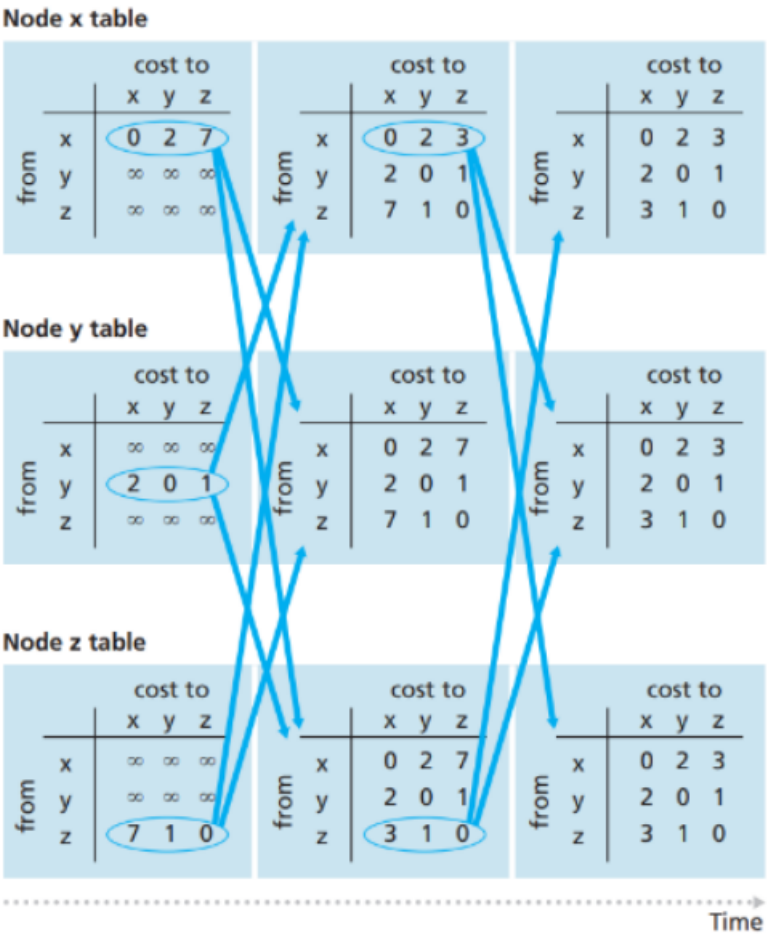
```
//初始化x的距离向量
for(所有其他结点y)
{
    if(y是邻居) Dx(y)=c(x,y);
    else Dx(y)=infinity;
}
//将x的距离向量发给所有邻居
for(所有邻居w)
{
    将 Dx=[Dx(y):y in N] 送给 w;
}
//无限循环，处理网络变化
while(1)
{
    wait_until(x到邻居w的链路成本变化||收到邻居w的更新信息)
    for(所有其他结点y)
    {
        Dx(y)=min_v {c(x,v)+D_v(y)};//更新x到其余所有节点距离，以此更新x距离向量
    }
    if(对任何结点y，Dx(y)变化)
    {
       将Dx(y)最小值发给所有邻居
    }
}
```

1.3.3. 算法示例



1️⃣ 初始化：初始化每个结点的路由选择表，包括结点自己的距离向量，邻居的距离向量全部设为无穷

完成后结果为图中第一列



2 周期性更新+计算：完成后结果为图中第二列

1. 每个结点向邻居更新距离向量：如图中箭头的指向
2. 结点收到更新后：重新计算自身的距离向量，以x结点为例有

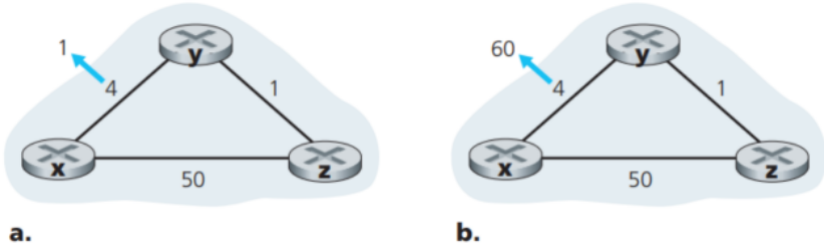
$$
\begin{aligned}
\boldsymbol{D}_x(x) &= 0 \\
\boldsymbol{D}_x(y) &= \min\{c(x,y) + \boldsymbol{D}_y(y), c(x,z) + \boldsymbol{D}_z(y)\} = \min\{2+0, 7+1\} = 2 \\
\boldsymbol{D}_x(z) &= \min\{c(x,y) + \boldsymbol{D}_y(z), c(x,z) + \boldsymbol{D}_z(z)\} = \min\{2+1, 7+0\} = 3
\end{aligned}
$$

3 改变后更新+计算：

1. x, z因为上一步，距离向量发生改变，所以立即让二者向邻居更新距离向量
2. 更新后再计算，算无可算算法静止，**直到一条链路开销发生改变**

1.3.4. 链路开销改变&链路故障

x与邻居v的开销改变：更新距离，在最低开销变化时，告诉所有邻居新距离向量



a. b.

1 好消息传达速度快：当4→1的变化发生时，以下时间挨个发生

1. y检测到开销变化，更新$\boldsymbol{D}_y$，向邻居更新新的距离向量
2. z收到来自y的更新距离表，计算出$z \to x$最小开销(从5减为2)，向邻居更新新的距离向量

3. y收到来自z的更新距离表，y开销未变所以不发送任何信息，算法静止

2. 坏消息传达速度慢：当4→60后，要迭代44此算法才会静止，这样容易造成无穷计数

1.3.5. 算法改进：增加毒性逆转

以$z \to x$为例

1. 操作：当路由是$z \to y \to x$时，z就持续给y撒谎that有$D_z(x) = \infty$

2. 好处：避免环路，加快收敛

1.4. LS/DV算法比较

算法	报文多少	收敛速度	健壮性
LS	多	快	大(错误只影响单个顶点)
DV	少	慢	小(错误会影响整个网络)

2. 路由选择

2.1. 自治系统(AS)：一堆路由器聚集

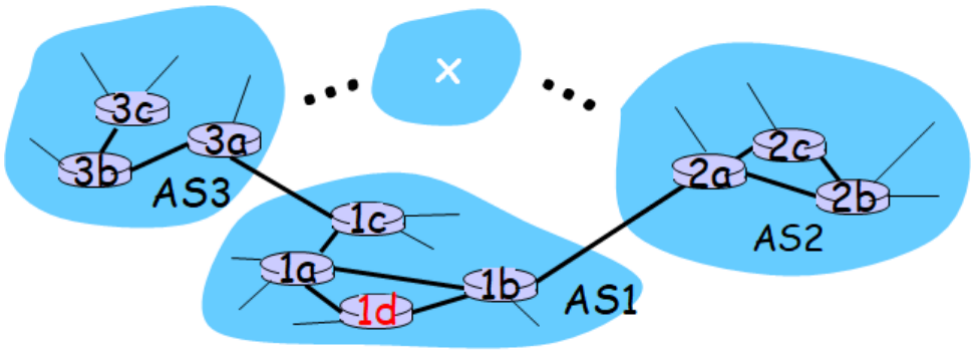
1. 协议运行：

   1. 同一AS内所有路由器运行相同协议，即内部网关协议(IGP)
   2. 不同AS的路由器运行不同协议
   3. 所有AS运行相同的AS间路由协议，即边界网关协议(BGP)

2. ASN：一个自治系统由一个ASN唯一标识

3. 网关路由器：直接连接另一个AS的路由器

2.2. 网关路由选择



2.2.1. 起因

1. AS1中路由器收到发往其他AS的数据报

2. AS1搞清，哪些路由可通过AS2/AS3访问，此处X都可通过AS2/AS3访问

3. 通过跨网关协议把可达性信息传给AS1内所有路由器

4. AS1内路由器将选一个网关路由器，转发数据报

2.2.2. 热土豆路由选择

1️⃣思想：以最小开销一股脑把分组送出AS1，至于送出AS1后分组到达目的地的成本，则完全不管

2️⃣示例：上图分组发往1d后，1d就会选择将分组转给1b(近)而不是1c(远)

2.3. 路由协议

特点	RIP	OSPF	BGP
网关协议	内部	内部	外部
路由表内容	目的网络，下一跳，距离	目的网络，下一跳，距离	目的网络，完整路径
最优通路依据	跳数	费用	多种有关策略
算法	距离-矢量算法	链路状态算法	路径-矢量算法
传送方式	UDP	IP数据报	TCP连接
其他	简单，效率低	效率高，规模大	\

2.3.1. 内部网关协议IGP

1️⃣路由信息协议RIP：基于DV算法(距离-向量)

1. AS内部最远两点不超过15跳，直接不超过15跳，周长不超过25跳
2. 每30s结点将距离向量广播给邻居
3. 180s内没收到邻居的广播则认为邻居已断开，重新计算BF方程，广播链路故障信息
4. 最优路径不唯一，则选其中一条
5. 算法由应用层的route-d进程管理

2️⃣开放最短路径优先OSPF：基于链路状态算法(Dijikstra)

1. 周期性地广播自己跟谁连，代价多少，广播给整个AS，周期30min
2. 运行在网络层上部，报文封装在IP报文中，是不可靠协议

2.3.2. 边界网关协议BGP

1️⃣边界网关维护的不是路由表，而是路径表

2️⃣BGP为每个AS提供如下方法

1. 向邻居应用服务器获取子网可达性信息
2. 将可达性信息传播到所有AS内部路由器
3. 根据可达性信息和策略确定到AS的好路由

3️⃣路由选择：优先使用本地策略，本地缺省时使用最短路径+热土豆

4️⃣四种报文：

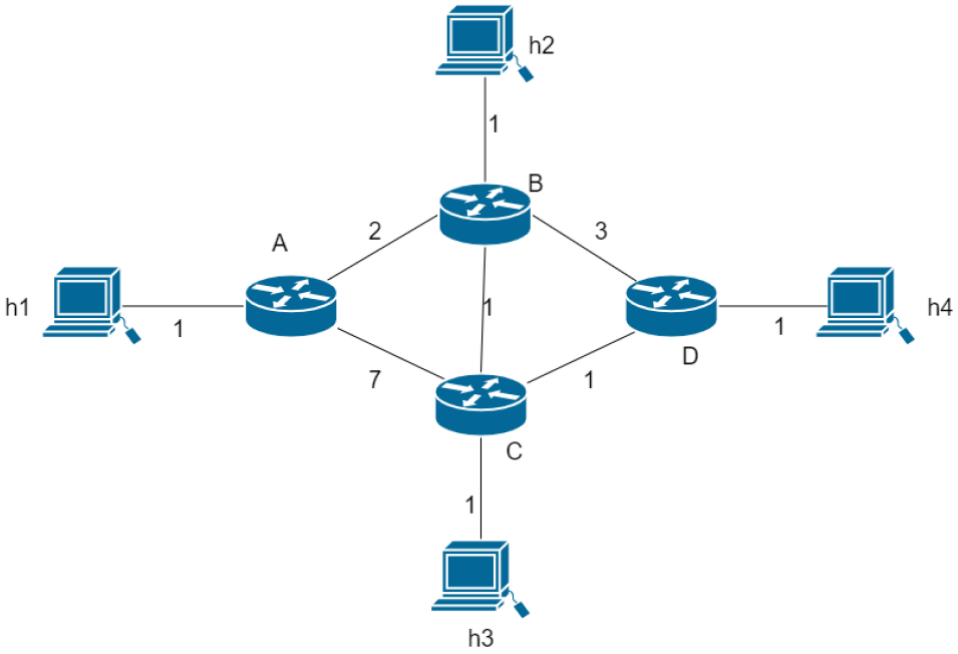
1. OPEN：与相邻的BGP建立联系

2. UPDATE：发送某一路由信息
3. KEEPALIVE：打开报文+周期性证实邻居关系
4. NOTIFICATION：报告报文错误，关闭TCP

3. 实验要求

2.1. 要修改的代码

1️⃣在`simulator/topos`文件夹下新建`myTopo.py`文件，实现如下拓扑结构



2️⃣补全文件`simulator/dv_router.py`

2.2. 要实现的功能

1️⃣实现静态路由

1. 实现 `add_static_route` 方法，用于在路由表中为每个直接连接的主机添加 `TableEntry` 对象
2. 效果
    - 之前：路由器无法自动记录直接连接的主机信息
    - 之后：路由器能够自动识别并记录每个直接连接的主机的路由信息，这些路由具有相应的延迟，并设置为永不过期

2️⃣转发

1. 实现 `handle_data_packet` 方法，以便在数据包到达路由器时适当地处理它们
2. 效果
    - 之前：路由器在收到数据包时不具备转发逻辑
    - 之后：路由器可以使用其路由表来正确转发数据包。如果没有路由信息或延迟太高，路由器将丢弃数据包

3️⃣发送路由表广播

1. 实现 `send_routes` 方法，以周期性地广告路由表

2. 效果：路由器定期向邻居广播路由表

4️⃣处理路由广播

1. 实现 `handle_route_advertisement` 方法，这个方法在路由器收到来自邻居的路由广播时被框架调用
2. 效果：路由器能够处理路由广播，并根据新路由信息更新路由表

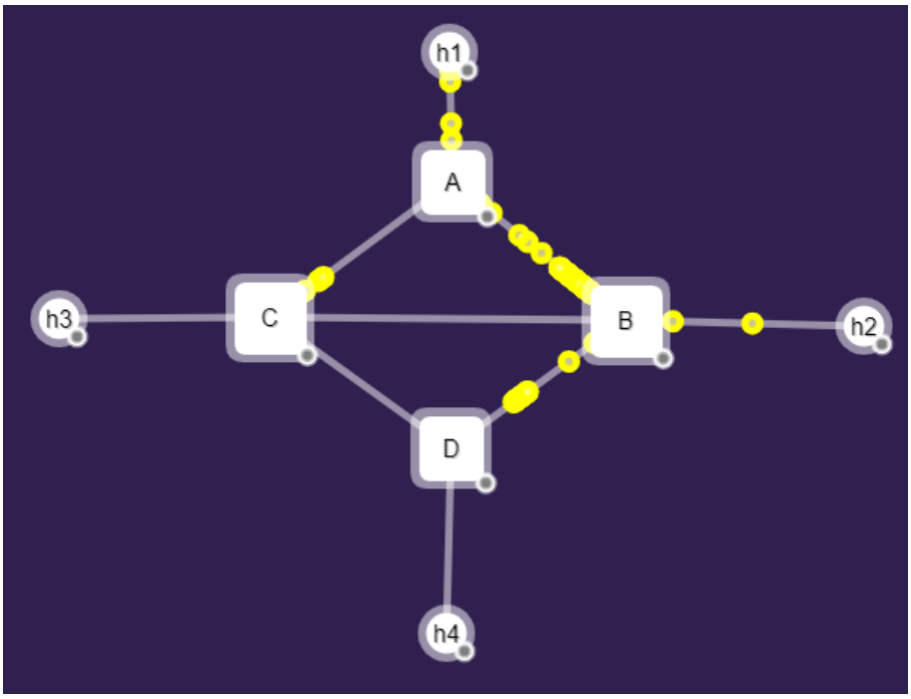
5️⃣处理路由表超时

1. 实现 `expire_routes` 方法，该方法用于清除过期的路由。该方法在路由器的计时器触发时自动调用
2. 效果：路由器能够自动识别并清除过期的路由，确保路由表的准确性和有效性

4. 拓扑实现

这个很简单，参照`simulator/topos/candy.py`就行了

```
import sim
def launch (switch_type = sim.config.default_switch_type, host_type =
sim.config.default_host_type):

  switch_type.create('A')
  switch_type.create('B')
  switch_type.create('C')
  switch_type.create('D')

  host_type.create('h1')
  host_type.create('h2')
  host_type.create('h3')
  host_type.create('h4')

  A.linkTo(h1, latency=1)
  B.linkTo(h2, latency=1)
  C.linkTo(h3, latency=1)
  D.linkTo(h4, latency=1)

  A.linkTo(B, latency=2)
  A.linkTo(C, latency=7)
  B.linkTo(C, latency=1)
  C.linkTo(D, latency=1)
  B.linkTo(D, latency=3)
```



5. dv_router.py

5.1. 阶段一add_static_route

1代码

```
#阶段1
#接收两参数，host为主机标识符，port为连接到主机的端口
def add_static_route(self, host, port):
    #确保传入的 port 是一个当前激活的端口
    assert port in self.ports.get_all_ports(), "Link should be up, but is not."

    #在路由器的路由表self.table中，为指定的host添加一个新的表项(TableEntry)
    #包含以下内容
    #  1.目的地dst设置为传入的host
    #  2.指定要使用的端口为传入端口
    #  3.self.ports.get_latency(port)会返回连接到指定端口的延迟值
    #  4.设置这条路由的过期时间为永久
    self.table[host] = TableEntry(dst=host, port=port, latency=self.ports.get_latency(port), expire_time=FOREVER)
```

2运行Demo

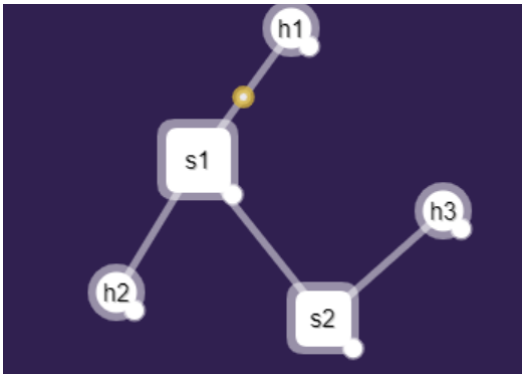
1. 打印路由表

```
$ python3 simulator.py --start --default-switch-type=dv_router
topos.simple
.....
>>> print(s1.table)
=== Table for s1 ===
name   prt lat  sec
------ --- ---- -----
h1     0   1    inf
h2     1   1    inf
```

2. ping操作

```
>>> h1.ping(h2)
```

数据从h1发出后，到达服务器s1后就没有然后了



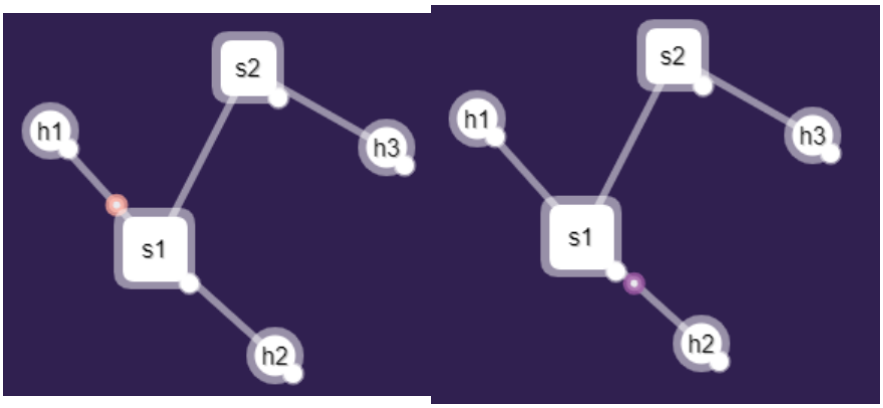
5.2. 阶段二handle_data_packet

1️⃣代码

```
#阶段2
#接收两参数，packet(接收到的数据包)和in_port(数据包进入的端口)
def handle_data_packet(self, packet, in_port):
    #self.table.get(packet.dst)用于获取与 packet.dst相关联的路由表条目
    #packet.dst不在self.table中，get方法将返回None
    export = self.table.get(packet.dst)

    #要求条目在路由表种存在，并且延迟小于INFINITY
    #当满足两个条件时，才不会丢弃数据包
    if not export or export.latency >= INFINITY: return

#两个条件都满足时，用self.send(packet, export.port)进行数据包转发
self.send(packet, export.port)
```

2️⃣运行Demo：ping操作有来有回了

```
>>> h1.ping(h2)
```



但是当主机所连接的服务器不用时，如

```
>>> h1.ping(h3)
```

则就无法完成路由了

5.3. 阶段三send_routes

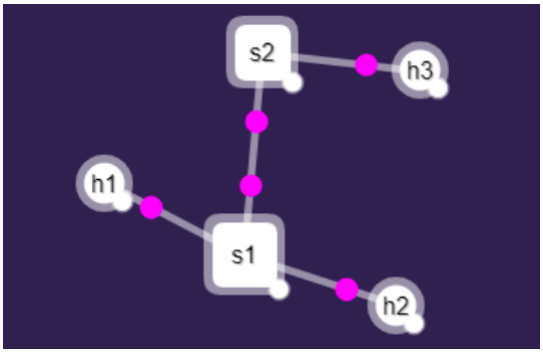
1️⃣代码

```
def send_routes(self, force=False, single_port=None):
    #决定路由信息应该发送到哪些端口
    #如果single_port被指定，路由信息将只发送到这个端口
    #否则，路由信息将发送到路由器上的所有端口
    if single_port:
        ports_to_send = [single_port]
    else:
        ports_to_send = self.ports.get_all_ports()

        #循环遍历所有需要发送路由信息的端口
        for port in ports_to_send:
            #遍历路由表中的每个条目。host 是目的地地址，entry是关联的路由表条目
            for host, entry in self.table.items():
                #应用两种路由优化：毒性逆转和分割视界
                #如果启用了毒性逆转且条目的端口与当前端口相同，则将延迟设置为无限大以防止路由循环
                #如果启用了分割视界且条目的端口与当前端口相同，则跳过此条目(不向来源端口广播)
                #否则，使用条目中的原始延迟值
                if self.POISON_REVERSE and entry.port == port:
                    latency_now = INFINITY
                elif self.SPLIT_HORIZON and port == entry.port:
                    continue
                else:
                    latency_now = entry.latency

                    # 发送路由，构造一个路由包，包含目的地dst，延迟值latency_now，并通过指定的端口发送
                    self.send_route(port=port, dst=host, latency=latency_now)

                    #更新了self.last_table，将其设置为当前路由表的副本
```

```
                    self.last_table = self.table.copy()
```

2️⃣运行Demo：

观察到路由广告数据包（以紫色点显示）定期从每个交换机发送



但是该操作还是不能够完成

```
>>>h1.ping(h3)
```

5.4. 阶段四expire_route

1️⃣代码

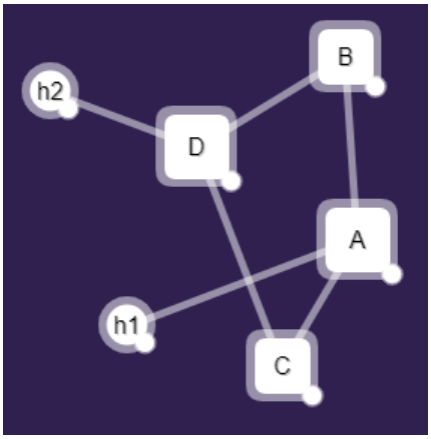
```
#阶段4
#当接收到邻居传来的路由广播后，用此方法更新路由器的路由表
#更新方法为：选择当前路由和新路由中较好的一个，如果两条路由性能相等则优先选择新路由
#接收三个参数：目标路由route_dst，到达该目标路由的延迟route_latency，接收广告的端口port
def handle_route_advertisement(self, route_dst, route_latency, port):
    #首先考虑延时无穷大(路由不可达)的情况
    if route_latency == INFINITY:
        #检查当前路由表的表项，如果目标路由已存在于路由表中，并且端口与收到广播的相同
        if route_dst in self.table and self.table[route_dst].port == port:
            entry = self.table[route_dst]
            #如果路由表中的该条目的延迟小于无穷大，说明之前这条路由是有效的，现在才变得不可达
            #因此，设置该路由的新过期时间为当前时间加上路由存活时间
            if (entry.latency < INFINITY):
                expire_time_now=self.ROUTE_TTL + api.current_time()
                #否则的话，说明该条目的延迟已经是无穷大，则维持原有的过期时间
            else:
                expire_time_now=entry.expire_time

                # 更新路由表，设置目的地为不可达(延迟为无穷大)并更新过期时间
                self.table[route_dst] = TableEntry(
                    dst = route_dst,
                    port=port,
                    latency=INFINITY,
                    expire_time=expire_time_now
                )

                #当收到的路由延迟不是无穷大时，计算新路由的总延迟
```

```
            else:
                #新延时包括接收到的路由延迟和当前端口的延迟
                #计算新路由的过期时间，为当前时间加上路由的存活时间
                new_latency = route_latency + self.ports.get_latency(port)
                new_expire_time = api.current_time()+self.ROUTE_TTL

                #创建一个新的路由表条目，包含目的地、端口、新计算的延迟和过期时间
                new_table = TableEntry(
                    dst=route_dst,
                    port=port,
                    latency=new_latency,
                    expire_time=new_expire_time
                )

                #如果路由表中不存在这个目的地的路由，直接在路由表中添加这个新路由
                if not self.table.get(route_dst):
                    self.table[route_dst] = new_table

                    #如果路由表中已经有这个目的地的路由，获取当前的路由条目
                else:
                    entry = self.table[route_dst]
                    #如果新路由是通过相同的端口接收或新路由的延迟更低
                    #则用新路由更新路由表中的这个目的地的路由
                    if new_table.port==entry.port or new_table.latency<entry.latency:
                        self.table[route_dst] = new_table
```

2️⃣运行demo：

在以下拓扑结构种运行以下指令，数据能够有来有回

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.square
........
>>>h1.ping(h2)
```



5.5. 阶段5`expire_routes`

1️⃣代码

```python
#阶段4
def expire_routes(self):
    #创建一个新的空路由表用于更新
    renew_table = Table()
    # 获取当前时间
    current_time = api.current_time()

    #遍历当前路由表中的所有条目
    for host,entry in self.table.items():

        #检查每个条目的过期时间是否仍然大于当前时间
        #如果该条目未过期，则将其添加到新的路由表中
        if entry.expire_time > current_time:
            renew_table[host] = entry

            #如果启用了过期路由的“中毒逆转”，并且条目的延迟不是无穷大
            #为这个过期路由设置一个新的过期时间
            #然后在新路由表中添加一个更新后的条目，将延迟设置为无穷大
            #表示路由现在不可用
        elif self.POISON_EXPIRED and entry.latency < INFINITY:
            new_expire_time = current_time + self.ROUTE_TTL
            renew_table[host] =TableEntry(
                dst=entry.dst,
                port=entry.port,
                latency=INFINITY,
                expire_time=new_expire_time)

            #使用新的路由表替换旧的路由表
            self.table = renew_table
```

2️⃣运行demo：

```
$ python3 simulator.py --start --default-switch-type=dv_router topos.candy
......
>>> s4.unlinkTo(s5)
>>> h1a.ping(h2a)
```

1. 使用 `topos.candy` 拓扑结构启动网络模拟器
2. 断开路由器 s4 和 s5 之间的连接
3. 等15s后让路由表自动更新
4. 之后再验证是否能正确转发，按理来说说要能正确转发的

6. 验收

```
$ python simulator.py --start --default-switch-type=dv_1 router topos.myTopo
....
>>> h1.ping(h4)
```

最后网络中没有出现泛洪现象，同时包走的是最短路径(A -> B -> C -> D)，说明我们的RIP算法设计成功