RIP路由协议软件实验报告

实验目的

通过软件实现RIP协议，详细分析距离矢量路由算法，掌握网络协议的构建过程。

实验内容

在软件层面上构建网络拓扑，编写路由器等代码，然后基于网络拓扑实现RIP路由协议，以及相应分析实验结果，如何实现路由表动态更新等。

实验原理

RIP是一种分布式的基于距离向量的路由选择协议，是因特网的协议标准，最大优点是简单。

RIP协议要求网络中每一个路由器都维护**从它自己到其他每一个目的网络的唯一最佳距离记录(即一组距离)**

RIP协议和谁交换？交换什么？多久交换一次？

1.仅和相邻路由器交换信息
2.路由器交换的信息是自己的路由表
3.每隔一个确定的时间交换一次路由信息，然后路由器根据新信息更新路由表。若超过一个确定的时间没收到邻居路由器的通告，则判定邻居没了，并更新自己路由表。

路由器刚开始工作时，只知道直接连接的网络的距离，接着每一个路由器也只和数目非常有限的相邻路由器交换并更新路由信息。
经过若干次更新后，所有路由器最终都会知道到达本自治系统任何一个网络的最短距离和下一跳路由器的地址即“收敛“。

RIP的具体过程：

1.修改相邻路由器发来的RIP报文中所有表项

对地址为X的相邻路由器发来的RIP报文，修改此报文中的所有项目:把“下一跳”字段中的地址改为X，并把所有的“距离”字段+自身到地址为X的相邻路由器的距离。

2.对修改后的RIP报文中的每一个项目，进行以下步骤:
(1)R1路由表中若没有该终点，则把该表项填入R1路由表

(2)R1路由表中若有该终点，则查看下一跳路由器地址:若下一跳是X，则用收到的项目替换源路　　由表中的项目；若下一跳不是X，原来距离比从X走的距离远则更新，否则不作处理。

3.若一个确定的时间还没收到相邻路由器X的更新路由表，则把X记为不可达的路由器。

4.返回

实验过程

编写*myTopo*

```python
import sim

def launch (switch_type = sim.config.default_switch_type, host_type = sim.config.default_host_type):
  """
  Creates a topology with loops.

  It looks like:


  """

  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)
  A.linkTo(B, latency=2)
  A.linkTo(C, latency=7)

  B.linkTo(h2, latency=1)
  B.linkTo(C, latency=1)
  B.linkTo(D, latency=3)

  C.linkTo(h3, latency=1)
  C.linkTo(D, latency=1)

  D.linkTo(h4, latency=1)
```

h1.ping(h4)得到以下输出：

```
>>> h1.ping(h4)
>>> DEBUG:user:h4:rx: <Ping h1->h4 ttl:15> A,B,C,D,h4
DEBUG:user:h1:rx: <Pong <Ping h1->h4 ttl:15>> D,C,B,A,h1

```

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RIP算法设计成功。

Stage 1/10: Static Routes

本阶段实现添加静态路由过程。

对于每个直接连接到路由器的主机，路由器应该记录到该主机的静态路由。

应为这些路由分配 link latency table中记录的latency，并且永远不会过期（expire_time = FOREVER）

在add_static_route(self, host, post) 函数中实现添加静态路由：

```python
    def add_static_route(self, host, port):

        # `port` should have been added to `peer_tables` by `handle_link_up`
        # when the link came up.
        assert port in self.ports.get_all_ports(), "Link should be up, but is not."

        # TODO: fill this in!

        self.table[host] = TableEntry(dst=host, port=port, 
latency=self.ports.get_latency(port), 
expire_time=FOREVER)

```

Stage 2/10: Forwarding

本阶段实现路由转发过程。

每当数据包到达路由器时，就会调用handle_data_packet函数。

如果数据包的目的地没有路由，路由器应该丢弃数据包（什么都不做）。

如果延迟大于或等于INFINITY，也应该丢弃数据包（稍后将进一步调整这种情况）。

```python
    def handle_data_packet(self, packet, in_port):
        """
        Called when a data packet arrives at this router.

        You may want to forward the packet, drop the packet, etc. here.

        :param packet: the packet that arrived.
        :param in_port: the port from which the packet arrived.
        :return: nothing.
        """
        # TODO: fill this in!

        if packet.dst in self.table and self.table[packet.dst][2] < INFINITY:
            self.send(packet, self.table[packet.dst][1])

```

Stage 3/10: Sending Routing Tables Advertisements

本阶段实现广播发送自己的路由表信息过程。

为了让路由器了解彼此的路由，每个路由器都必须向其邻居发送其表中的路由。

路由器应该定期广播路由，以刷新路由并防止它们过期。

通过调用$self.send_routes(force=True)$自动播发表中的所有路由。

send_routes使用的force参数决定是广播所有路由（如果force=True的话）还是只播发自上次播发以来更改的路由（如果force=False的话）。

在本阶段只需实现force=True情况下的send_routes方法。

要发送路由，需要用到RoutePacket类型的数据包，其中包含一个路由。它的构造函数接受一个destination参数（被路由到的主机）和一个latency参数（到目的地的距离）。

```
def send_routes(self, force=False, single_port=None):
    """
    Send route advertisements for all routes in the table.

    :param force: if True, advertises ALL routes in the table;
                  otherwise, advertises only those routes that have
                  changed since the last advertisement.
           single_port: if not None, sends updates only to that port; to
                        be used in conjunction with handle_link_up.
    :return: nothing.
    """
    for port in self.ports.get_all_ports():
        for entry in self.table.values():
            dst = entry[0]
            latency = entry[2]
            neighbor = entry[1]
            packet = RoutePacket(dst, latency)
                    if (force == True):
                        self.send(packet, port)

```

Stage 4/10: Handle Route Advertisements

本阶段实现路由器处理其邻居发送的路由表项过程。

根据对方表项中的主机是否在自己的表中存在，以及对方路由表项的时延是大是小，决定是否更新自己的路由表。

对方路由表项的latency为对方到该主机的latency，还需要加上自己通过端口p发送到对方的latency，即$route_latency + self.ports.get_latency(port)$

需考虑以下几种情况：

- 接收到的路由表项的目的主机不在自己的路由表中，将其添加到自己的路由表
- 接收到的路由表项的目的主机在自己的路由表中：
  - $route_latency + self.ports.get_latency(port) >$之前路由表中的latency，不更新；
  - $route_latency + self.ports.get_latency(port) <$之前路由表中的latency，更新；
  - $route_latency + self.ports.get_latency(port) =$之前路由表中的latency，不更新；
  - 如果新旧表项所指出下一跳的端口相同，代表收到了新的通过该邻居到达该目的地的表项，无论latency，更新（因为新表项为最新的路由信息，若不更新的话可能旧表项的latency为一条故障链路的latency）。

更新表项时，需要重新设置表项的到期时间 $expire_time$为$api.current_time() + self.ROUTE_TTL$。

```python
    def handle_route_advertisement(self, route_dst, route_latency, port):
        """
        Called when the router receives a route advertisement from a neighbor.

        :param route_dst: the destination of the advertised route.
        :param route_latency: latency from the neighbor to the destination.
        :param port: the port that the advertisement arrived on.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                              latency=self.ports.get_latency(port),
expire_time=FOREVER)

        if route_dst in self.table:
            if self.table[route_dst][2] > route_latency + self.ports.get_latency(port):
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
self.ports.get_latency(port), api.current_time() + self.ROUTE_TTL)
            if self.table[route_dst][1] == port:
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                self.ports.get_latency(port),api.current_time() +
self.ROUTE_TTL)
        else:
            self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                        self.ports.get_latency(port),api.current_time() + self.ROUTE_TTL)

```

Stage 5/10: Handling Routing Tables Timeouts

本阶段实现处理超时的路由表项。

删除超时的路由表项即可。

```python
    def expire_routes(self):
        """
        Clears out expired routes from table.
        accordingly.
        """
        # TODO: fill this in!

        """
          A routing table

          You should use a `Table` instance as a `dict` that maps a
          destination host to a `TableEntry` object.
        """

        # usr list in order to keep keys to be deleted  are correct
        for key in list(self.table.keys()):
            if self.table[key].has_expired:
                    del self.table[key]

```

Stage 6/10: Split Horizon (Let’s Get Loopy)

本阶段实现解决相邻两路由器认为对方含有到达某一destination的路由而无限循环更新自身的路由表项问题。

解决方案为如果你使用的是通过邻居N到达目的地D的路线，则不向邻居N发送到达目的地D的这条路由消息。

实现该方法只需要在 $send\_routes()$ 中遍历每个port，在每个port上遍历每个路由表项，如果发现某个路由表项的port 字段与当前遍历到的port相同，那么就不在这个port发送该路由表项。

更新$send_routes$函数如下：

```python
    def send_routes(self, force=False, single_port=None):
        """
        Send route advertisements for all routes in the table.

        :param force: if True, advertises ALL routes in the table;
                      otherwise, advertises only those routes that have
                      changed since the last advertisement.
               single_port: if not None, sends updates only to that port; to
                            be used in conjunction with handle_link_up.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                              latency=self.ports.get_latency(port), expire_time=FOREVER)

        # Split Horizon a->b->c don't advertise destination c to neighbor b

        if single_port == None:
            for port in self.ports.get_all_ports():
                for entry in self.table.values():
                    dst = entry[0]
                    latency = entry[2]
                    neighbor = entry[1]
                    #不发给这个neighbor
                    if self.SPLIT_HORIZON and port == neighbor:
                        pass
                    else:
                        packet = RoutePacket(dst, latency)
                        if (force == True):
                            self.send(packet, port)

```

Stage 7/10: Poison Reverse (Still loopy)

本阶段目的与阶段6相同，但是所用时间更短。

在stage 6，不得不等待无效路线到期才可以终止循环。

在stage 7，当一条路线无效时，通过发送这条链路的latency是INFINITY的路由信息表明该路线已不存在。

更新*send_routes*函数如下：

```
    def send_routes(self, force=False, single_port=None):
        """
        Send route advertisements for all routes in the table.

        :param force: if True, advertises ALL routes in the table;
                      otherwise, advertises only those routes that have
                      changed since the last advertisement.
               single_port: if not None, sends updates only to that port; to
                            be used in conjunction with handle_link_up.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                              latency=self.ports.get_latency(port),
expire_time=FOREVER)

        # Split Horizon a->b->c don't advertise destination c to neighbor b

        if single_port == None:
            for port in self.ports.get_all_ports():
                for entry in self.table.values():
                    dst = entry[0]
                    latency = entry[2]
                    neighbor = entry[1]
                    # 如果要通过port到host，把那个host的路由项设成INDINITY发给这个neighbor
                    if (self.POISON_REVERSE and port == neighbor):
                        packet = RoutePacket(dst, INFINITY)
                        if force == True:
                            self.send(packet, port)
                    #不发给这个neighbor
                    elif self.SPLIT_HORIZON and port == neighbor:
                        pass
                    else:
                        packet = RoutePacket(dst, latency)
                        if (force == True):
                            self.send(packet, port)

```

Stage 8/10: Counting to Infinity

本阶段实现。

stage 6 和 stage 7，只有当循环位于两个相邻路由器之间时，才能检测到它。

通过设立阈值（INFINITY）来解决该问题，不接受任何latency高于我们阈值的路由。

- 如果当前路由器接收到了一个被下了毒的路由信息，并且该路由信息的destination和接收该包的port均与路由表中某个表项匹配，那么将该表项替换为latency为INFINITY的表项，使得“poison”可以在网路中传播；
- 把某一个表项更新为latency为INIFINITY的表项时，不更新该表项的到期时间；
- 如果当前路由器收到了一个被下了毒的路由信息，但是destination和接收该包的port不匹配路由表中的某个表项，那么丢弃该包。

更新*send_routes*函数如下：

```
    def handle_route_advertisement(self, route_dst, route_latency, port):
        """
        Called when the router receives a route advertisement from a neighbor.

        :param route_dst: the destination of the advertised route.
        :param route_latency: latency from the neighbor to the destination.
        :param port: the port that the advertisement arrived on.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                              latency=self.ports.get_latency(port),
expire_time=FOREVER)

        if route_dst in self.table:
            if route_latency >= INFINITY:
                if self.table[route_dst][1] == port and self.table[route_dst][2] < INFINITY:
                    self.table[route_dst] = TableEntry(route_dst, port, INFINITY,
                                                       self.table[route_dst][3])

            else:
                if self.table[route_dst][2] > route_latency + self.ports.get_latency(port):
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                self.ports.get_latency(port), api.current_time() +
self.ROUTE_TTL)
                if self.table[route_dst][1] == port:
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                self.ports.get_latency(port), api.current_time() +
self.ROUTE_TTL)
        else:
            self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                                    self.ports.get_latency(port),
api.current_time() + self.ROUTE_TTL)

```

Stage 9/10: Poisoning Expired Routes

本阶段实现比通过删除过期的路由表项通知整个网络该路径不存在更快的方法，为当一个路由表项过期时，我们将相应路由的latency更新为INFINITY。

同时，为了保证被下了毒的路由表项能够发送出去，重置它的到期时间。

更新$expire_routes$函数如下：

```
    def expire_routes(self):
        """
        Clears out expired routes from table.
        accordingly.
        """
        # TODO: fill this in!

        """
          A routing table

          You should use a `Table` instance as a `dict` that maps a
          destination host to a `TableEntry` object.
        """

        # usr list in order to keep keys to be deleted  are correct
        for key in list(self.table.keys()):
            if self.table[key].has_expired:
                if self.POISON_EXPIRED:
                    self.table[key] = TableEntry(self.table[key][0], self.table[key][1],
                                                 INFINITY, api.current_time() +
self.ROUTE_TTL)
                else:
                    del self.table[key]
```

Stage 10/10: Becoming Eventful

本阶段实现增量更新和触发更新——每次更新（触发）路由器的表项时，路由器都会发送路由表项，并且只发送那些已更改的路由表项（增量）。

具体要实现的内容如下：

- 实现 $send\_routes(self, force, single\_port)$ 中 $force == False$ 的分支，并且在代码中其他函数内根据需要调用 $send\_routes(force = False)$；

- 当路由器有了新连接后，实验环境会调用 $handle\_link\_up$ 函数。此时需要当前路由器单独向新连接的设备发送自己的整个路由表（仅当 $self.SEND\_ON\_LINK\_UP == True$ 时发送），也就是处理 $single\_port! = None$ 的情况；
- 当路由器某个连接断开后，实验环境会调用 $handle\_link\_down$ 函数。此时需要在下线的端口对应的所有路由表项中“下毒”（仅当 $self.POISON\_ON\_LINK\_DOWN == True$ 时下毒）。

为了实现该阶段，维护一个history数据结构，在$__init__$函数中实现如下：

```python
self.history = {}
```

其记录从每个port为每个destination发送的最新路由信息。

为此，添加两个函数如下：

```python
    def update_history(self, port, packet):
        self.history[(port, packet.destination)] = packet.latency

    def has_no_updated(self, port, packet):
        if (port, packet.destination) in self.history.keys() and self.history[(port, packet.destination)] == packet.latency:
            return False
        else:
            return True
```

其中$update_history$函数更新history，$has_no_updated$函数用来检查某个路由信息是否在history字典中，即是否是最新的路由信息。

更新$send_routes$函数如下：

每次发送路由信息时，均更新history，如果force == False，那么调用$has_no_updated$函数来查看该路由信息是否还没有被更新（即不在history字典中，是一个新的路由信息），如果该路由信息是一个新的路由信息的话，那么发该路由信息并更新history。

当$single_port! = None$时，也即$handle_link_up$ 函数需要用到该情况，仅需将$single_port == None$ 时的遍历port改为$port = single_port$ 即可。

```python
    def send_routes(self, force=False, single_port=None):
        """
        Send route advertisements for all routes in the table.

        :param force: if True, advertises ALL routes in the table;
                      otherwise, advertises only those routes that have
                      changed since the last advertisement.
               single_port: if not None, sends updates only to that port; to
                            be used in conjunction with handle_link_up.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                               latency=self.ports.get_latency(port), expire_time=FOREVER)

        # Split Horizon a->b->c don't advertise destination c to neighbor b

        if single_port == None:
            for port in self.ports.get_all_ports():
                for entry in self.table.values():
                    dst = entry[0]
                    latency = entry[2]
                    neighbor = entry[1]
                    # 如果要通过port到host，把那个host的路由项设成INDINITY发给这个neighbor
                    if (self.POISON_REVERSE and port == neighbor):
                        packet = RoutePacket(dst, INFINITY)
                        if force == True:
                            self.send(packet, port)
                            self.update_history(port, packet)
                        elif self.has_no_updated(port, packet):
                            self.send(packet, port)
                            self.update_history(port, packet)
                    #不发给这个neighbor
                    elif self.SPLIT_HORIZON and port == neighbor:
                        pass
                    else:
                        packet = RoutePacket(dst, latency)
                        if (force == True):
                            self.send(packet, port)
                            self.update_history(port, packet)
                        elif self.has_no_updated(port, packet):
                            self.send(packet, port)
                            self.update_history(port, packet)

        else:
            port = single_port
            for entry in self.table.values():
                dst = entry[0]
                latency = entry[2]
                neighbor = entry[1]
                # 如果要通过port到host，把那个host的路由项设成INDINITY发给这个neighbor
                if (self.POISON_REVERSE and port == neighbor):
                    packet = RoutePacket(dst, INFINITY)
                    if force == True:
                        self.send(packet, port)
                        self.update_history(port, packet)
                    elif self.has_no_updated(port, packet):
                        self.send(packet, port)
                        self.update_history(port, packet)
                # 不发给这个neighbor
                elif self.SPLIT_HORIZON and port == neighbor:
                    pass
                else:
                    packet = RoutePacket(dst, latency)
                    if (force == True):
                        self.send(packet, port)
                        self.update_history(port, packet)
                    elif self.has_no_updated(port, packet):
                        self.send(packet, port)
                        self.update_history(port, packet)


```

$handle\_link\_up$ 函数和 $handle\_link\_down$ 函数实现如下：

```python
    def handle_link_up(self, port, latency):
        """
        Called by the framework when a link attached to this router goes up.

        :param port: the port that the link is attached to.
        :param latency: the link latency.
        :returns: nothing.
        """
        self.ports.add_port(port, latency)

        # TODO: fill in the rest!

        if self.SEND_ON_LINK_UP:
            self.send_routes(True, port)

    def handle_link_down(self, port):
        """
        Called by the framework when a link attached to this router does down.

        :param port: the port number used by the link.
        :returns: nothing.
        """
        self.ports.remove_port(port)

        # TODO: fill this in!

        if self.POISON_ON_LINK_DOWN:
            for key in list(self.table.keys()):
                if self.table[key][1] == port:
                    # time should ROUTE_TTL or maintain?
                    self.table[key] = TableEntry(self.table[key][0], self.table[key][1], INFINITY,                                                             api.current_time() + self.ROUTE_TTL)

            self.send_routes()
```

此外，在$handle_route_advertisement$ 函数最后加一行$self.send_routes()$，即：

```python
    def handle_route_advertisement(self, route_dst, route_latency, port):
        """
        Called when the router receives a route advertisement from a neighbor.

        :param route_dst: the destination of the advertised route.
        :param route_latency: latency from the neighbor to the destination.
        :param port: the port that the advertisement arrived on.
        :return: nothing.
        """
        # TODO: fill this in!

        # self.table[host] = TableEntry(dst=host, port=port,
        #                              latency=self.ports.get_latency(port), expire_time=FOREVER)

        if route_dst in self.table:
            if route_latency >= INFINITY:
                if self.table[route_dst][1] == port and self.table[route_dst][2] < INFINITY:
                    self.table[route_dst] = TableEntry(route_dst, port, INFINITY,
                                                       self.table[route_dst][3])

            else:
                if self.table[route_dst][2] > route_latency + self.ports.get_latency(port):
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                                self.ports.get_latency(port), api.current_time() + self.ROUTE_TTL)
                if self.table[route_dst][1] == port:
                    self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                                self.ports.get_latency(port), api.current_time() + self.ROUTE_TTL)
        else:
            self.table[route_dst] = TableEntry(route_dst, port, route_latency +
                                                self.ports.get_latency(port), api.current_time() + self.ROUTE_TTL)

        self.send_routes()
```

注意到在函数*handle_link_up* , *handle_link_down* ,*handle_route_advertisement* 中调用了*send_routes*()函数，那是因为函数*handle_link_up* 此时需要当前路由器单独向新连接的设备发送整个路由表，而函数*handle_link_down* 以及 *handle_route_advertisement* 均有可能改变当前路由器的路由表，故须调用*send_routes*()函数。

实验结果

myTopo部分 实验结果：

h1.ping(h4)得到以下输出：

```
>>> h1.ping(h4)
>>> DEBUG:user:h4:rx: <Ping h1->h4 ttl:15> A,B,C,D,h4
DEBUG:user:h1:rx: <Pong <Ping h1->h4 ttl:15>> D,C,B,A,h1
```

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RIP算法设计成功。

Stage1 - 10 实验结果：

```
PS C:\Users\1120779818\Desktop\计算机网络软件实验2 2\cs168_proj_routing_student-master\simulator> python dv_unit_tests.py 10
********** Stage 0: TestStarterCode **********
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK

```

```
********** Stage 1: TestStaticRoutes **********
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK

********** Stage 2: TestForwarding **********
....
----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK

********** Stage 3: TestAdvertise **********
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK

********** Stage 4: TestHandleAdvertisement **********
........
----------------------------------------------------------------------
Ran 8 tests in 0.005s

OK

********** Stage 5: TestRemoveRoutes **********
....
----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK

********** Stage 6: TestSplitHorizon **********
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK

********** Stage 7: TestPoisonReverse **********
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK

********** Stage 8: TestInfiniteLoops **********
...
----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK
        Stage 4 TestHandleAdvertisement            :  8 / 8  passed
        Stage 5 TestRemoveRoutes                   :  4 / 4  passed
        Stage 6 TestSplitHorizon                   :  1 / 1  passed
        Stage 7 TestPoisonReverse                    :  1 / 1  passed
        Stage 8 TestInfiniteLoops                    :  3 / 3  passed
        Stage 9 TestRoutePoisoning                   :  5 / 5  passed
        Stage 10 TestTriggeredIncrementalUpdates     : 31 / 31 passed

Total score: 100.00 / 100.00

```

实验收获

通过本次实验，进一步掌握了距离向量路由算法以及 RIP 协议的实现，尤其是阶段6-10使我受益匪浅。自己也体会到了计算机网络的魅力所在。