```
'''
                COSC364 (RIPv2 Routing Protocol)
          Authors: Haider Saeed (msa280), Drogo Shi (msh217)
                    Date: 07/03/2022
                 Filename: RIPv2_Router.py

Program Definition: Contains the functionality of a router which includes
                the creation and checking of packets. It also includes
                route and metric calculation, timers, split hoirzon with
                poisoned reverse. This class handles the functions of a
                RIP Version 2 Router.
'''


import sys
import configparser     # ConfigParser class which implements a basic configuration language
import time
import socket
import select
import threading    # use timer here so sys load will not affect the time
import random
from threading import Timer



LOCAL_HOST = '127.0.0.1'



class RIPv2_Router():
    """ This is the Router class for the RIPv2 Protocol."""


    def __init__(self, router_id, neighbours, sending_socket):
        """ Initialises the router. """
        self.table = {}
        self.self_id = router_id
        self.neighbours = neighbours
        self.sending_socket = sending_socket
        self.timeout = 30
        self.garbage_time = 30
        self.timeout_timer_dict = {} #diction for record every entry's start time of timeout timer
        self.garbage_timer_dict = {} #diction for record every entry's start time of garbage timer




    def init_timer(self, dst_id):
        """ Initializes the timer. """
        (flag, timeout_timer, garbage_timer) = (self.table[dst_id][2], self.table[dst_id][3], self.table[dst_id][4])
        self.table[dst_id][2] = False
        self.table[dst_id][3].cancel()
        self.table[dst_id][4].cancel()
        self.table[dst_id][3] = self.start_timeout(dst_id)
        self.table[dst_id][4] = threading.Timer(self.garbage_time, self.delete_router, (dst_id,))
        if self.garbage_timer_dict.get(dst_id):
            self.garbage_timer_dict.pop(dst_id) #so no garbage timer started




    def start_timeout(self, router_id):
        """Starts a timeout timer for an entry in the routing table"""
        # Remember when deleting from the table, cancel this timer first
        threading_time = threading.Timer(self.timeout, self.end_timeout,(router_id,)) #for every 30 will call not_reciving func
        threading_time.start()
```

```
    self.timeout_timer_dict[router_id] = time.time()
    return threading_time



def end_timeout(self, router_id):
    """ Updates the routing table after a router has timed out. """
    # After timeout, the router changes the metric of that entry and triggers updates

    self.give_update("Router {} has timed out!\n".format(router_id))
    entry = self.table.get(router_id)
    if entry[2] == False:
        entry[0] = 16  # Change metric to 16
        entry[2] = True # Change Flag
        self.send_packet() # Triggers updates when metric first becomes 16
    self.start_garbage_timer(router_id) # Start a garbage timer
    entry[3].cancel() # Closes the timeout timer



def start_garbage_timer(self, router_id):
    '''Starts a garbage timer. '''
    self.table.get(router_id)[4].start()
    self.garbage_timer_dict[router_id] = time.time()



def delete_router(self, router_id):
    '''Upon completion of garbage timer, it pops the router from the table.'''
    timeout_timer, garbage_timer = (self.table[router_id][3], self.table[router_id][4])
    #cancel timer for the timeout and garbage
    timeout_timer.cancel()
    garbage_timer.cancel()
    popped_router = self.table.pop(router_id, 0)
    self.give_update("Router {} has been deleted from the routing table.".format(router_id))



def create_packet(self, send_to_neighbour_id):
    """ Creates a RIP Packet from the routing table to send to one of the neighbours. """

    packet = bytearray()

    # RIP Common Header
    packet.append(2) # Command
    packet.append(2) # Version
    packet.append(self.self_id &0xFF) # Router id
    packet.append((self.self_id >> 8) &0xFF) #R outeprint_routing_table ID

    # RIP Entries
    if (len(self.table) == 0): #If no entry in table, only header will be sent to the table
        return packet

    for dst_router_id, values in self.table.items():
        if dst_router_id == send_to_neighbour_id:
            continue

        # Address family identifier
        packet.append(0)
        packet.append(0)

        # Route tag
```

```
        packet.append(0)
        packet.append(0)

        # Router ID/IP
        packet.append(0)
        packet.append(0)
        packet.append(dst_router_id &0xFF)
        packet.append((dst_router_id >> 8) &0xFF)
        # Subnet mask
        packet.append(0)
        packet.append(0)
        packet.append(0)
        packet.append(0)
        # Next Hop
        packet.append(0)
        packet.append(0)
        packet.append(0)
        packet.append(0)


        metric = values[0]
        next_hop = values[1]

        # Implementing split horzion with posioned reverse. Updating metric accordingly.
        # Metric
        if ((send_to_neighbour_id != dst_router_id) and (next_hop == send_to_neighbour_id)):
            packet.append(0)
            packet.append(0)
            packet.append(0)
            packet.append(16 &0xFF)
        else:
            packet.append(0)
            packet.append(0)
            packet.append(0)
            packet.append(metric &0xFF)

    return packet




def check_rip_header(self, header):
    """ Checks if the RIP header is correct. If it is correct, it returns
    the id of the router it received from."""

    command = int(header[0])
    version = int(header[1])
    received_from_router_id = (int(header[2] & 0xFF)) + int((header[3] << 8))

    if ((command != 2 or version != 2) or (received_from_router_id < 1 or received_from_router_id > 6400)) :
        print_msg("Wrong Packet Received!\n")
        return False
    else:
        return received_from_router_id




def check_rip_entry(self, entry):
    """  Checks the incoming packet for correctness and returns True if
    check passed. """

    address_family_id = (int(entry[0]), int(entry[1]))
    route_tag = (int(entry[2]), int(entry[3]))
    router_id = (int(entry[4]), int(entry[5]))
    subnet_mask = (int(entry[8]), int(entry[9]), int(entry[10]), int(entry[11]))
```

```
        next_hop = (int(entry[12]), int(entry[13]), int(entry[14]), int(entry[15]))
        metric_zero_part = (int(entry[16]), int(entry[17]), int(entry[18]))
        metric = (int(entry[19]))

        entry_check_passed = False

        # Check address family identifier
        if (address_family_id != (0,0)):
            print_msg("\nIncorrect Packet. Wrong address family identifier value.")
        # Check route tag
        elif (route_tag != (0,0)):
            print_msg("\nIncorrect Packet. Wrong route tag value.")
        # Check router id
        elif (router_id != (0,0)):
            print_msg("\nIncorrect Packet. Wrong router id value.")
        # Check subnet mask
        elif (subnet_mask != (0,0,0,0)):
            print_msg("\nIncorrect Packet. Wrong subnet mask value.")
        # Check next_hop
        elif (next_hop != (0,0,0,0)):
            print_msg("\nIncorrect Packet. Wrong next hop value.")
        # Check metric
        elif (metric_zero_part != (0,0,0)):
            print_msg("\nIncorrect Packet. Wrong metric value.")
        elif (metric > 16 or metric < 0):
            print_msg("\nIncorrect Packet. Metric value out of range.")
        else:
            entry_check_passed = True

        return entry_check_passed




    def receive_packet(self, packet):
        """ Process a received packet. """
        self.give_update("Packet received.")

        # Check header and entries
        len_packet = len(packet)
        neb_id = self.check_rip_header(packet)

        # If header check fails, then the packet is dropped.
        if (neb_id == False):
            print_msg("Incorrect packet header. Packet dropped!")
            return

        # Packet checking
        for entry_start_index in range(4, len_packet, 20): #every entry has 20byts
            if not (self.check_rip_entry(packet[entry_start_index:entry_start_index+20])):
                index_entry = (len_packet - 4) // 20
                print_msg(f"Wrong entry for index_entry: {index_entry}")
                print_msg("Dropped the packet!")
                return

        # If table doesn't have neighbour
        if (self.table.get(neb_id) == None):
            cost,_ = self.neighbours.get(neb_id)
            self.table[neb_id] = [cost, neb_id, False, self.start_timeout(neb_id), threading.Timer(self.garbage_time,
self.delete_router, (neb_id,))]
        # Else, reinitialize the timer and cost
        else:
            cost,_ = self.neighbours.get(neb_id)
            self.init_timer(neb_id)
            self.table[neb_id][0] = cost
```

```
        # Stops furthur processing if only the header is received.
        if (len_packet == 4):
            return
        # End of neighbour processing

        # Start for non-neighbour processing
        for entry_start_index in range(4,len_packet,20):
            index_entry = (len_packet - 4) // 20
            self.process_entry(packet[entry_start_index:entry_start_index+20],neb_id)

        # Prints routing table after receiving and processing packet.
        #self.print_routing_table()




    def process_entry(self, entry, neb_id):
        """Processes one entry so the table might be changed."""
        router_id = (int(entry[6] & 0xFF)) + int((entry[7] << 8))
        entry_metric = int(entry[19])
        total_cost = min(16, self.neighbours.get(neb_id)[0] + entry_metric)

        # Change metric of the table
        if self.table.get(router_id):

            # Next hop
            metric, next_hop = self.table.get(router_id)[0], self.table.get(router_id)[1]

            #next hop is packet from
            if (next_hop == neb_id):

                # If cost is different, reinitialze the timer and change cost
                if (total_cost != metric):
                    self.init_timer(router_id)
                    self.table.get(router_id)[0] = total_cost

                    # Trigger event if metric changes to 16
                    if (total_cost == 16):
                        # Flag changes to True
                        self.table.get(router_id)[2] = True
                        # Timeout timer gets cancelled
                        self.table[router_id][3].cancel()
                        # Packet is then sent to neighbour and garbage timer is started
                        self.periodically_send_packets()
                        self.start_garbage_timer(router_id)
                else:
                    # If cost doesn't change and is not 16, reset router timers
                    if (total_cost != 16):
                        self.init_timer(router_id)

            # If next hop != neb_id  and it cost less
            else:
                if (total_cost < metric):
                    # Cancel both the timers
                    self.table[router_id][3].cancel()
                    self.table[router_id][4].cancel()
                    self.table[router_id] = [total_cost, neb_id, False, self.start_timeout(router_id), threading.Timer(self.garbage_time,
self.delete_router,(router_id,))]

        elif self.table.get(router_id) == None and total_cost != 16:
            self.table[router_id] = [total_cost, neb_id, False, self.start_timeout(router_id), threading.Timer(self.garbage_time,
self.delete_router, (router_id,))]

        else:
            return
```

```
def print_routing_table(self):
    """ Prints the routing table's of the router."""

    #self.wait(0.4)
    print("\n")
    print(" _____________________[Router {}]_____________________".format(self.self_id))
    print("|_____________________________________________________________|")
    print("| Router ID | Next Hop | Cost |    Timeout   |  Garbage Timer  |")
    print("|===========|==========|======|==============|=================|")

    router_id_list = list(self.table.keys())
    router_id_list.sort()
    for router_id in router_id_list:
        metric, next_hop, flag, timeout, garbage = self.table.get(router_id)
        timeout_time = time.time() - self.timeout_timer_dict[router_id]

        if (self.garbage_timer_dict.get(router_id)):
            garbage_timer_time = time.time() - self.garbage_timer_dict[router_id]
            timeout_time = 0
        else:
            garbage_timer_time = 0

        print("|{:^11}|{:^10}|{:^6}|{:^14.3f}|{:^17.3f}|".format(router_id, next_hop, metric, timeout_time, garbage_timer_time))

    print("|___________|__________|______|______________|_________________|")
    print("\n")



def send_packet(self):
    """ Creates a packet for each neigbour and sends it to the neighbour. """
    self.give_update("Sending Packet.")
    #self.wait(0.4)
    for neighbor_id, values in self.neighbours.items():
        packet = self.create_packet(neighbor_id)
        neb_port_num = values[1]
        self.sending_socket.sendto(packet, (LOCAL_HOST, neb_port_num))



def periodically_send_packets(self):
    """ Sends packets to neigbours periodically. Done when a certain
    random amount of time has passed. """
    self.send_packet()
    t = threading.Timer(5 + 0.2 * (random.randrange(0, 6) * random.randrange(-1, 1)), self.periodically_send_packets)
    t.start()
    self.print_routing_table()



def give_update(self, message):
    """ Gives the update message and the time at which it was sent. """
    current_time = time.strftime("%Hh:%Mm:%Ss")
    print("[" + current_time + "]: " + message)
```