```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_router.py
"""

# Import Modules
import time
import random
from datetime import datetime
from network_interface import Interface
from forwarding_route import Route
from rip_packet import RipPacket, RipEntry
from IO_formatter import routing_table_formatter


# Router Class
class Router:
    """
    An object that simulates a router with rip protocol
    """

    # Class attributes
    INFINITY = 16
    REGULAR_TIMER_OFFSET = 1.0

    def __init__(self, router_id,
                 inputs, outputs,
                 period, timeout):
        """
        the __* attributes are private attributes which can only be
        accessed by getter outside of class.

        Parameters:
        router_id: an integer, i.e. 1, 2, 3, etc
        inputs: a list of integers, i.e. [5001, 5002, 5003]
        outputs: a dictionary of dictionaries, i.e.
        {6010(port): {'metric': 1, 'router_id': 1},
         6030(port): {'metric': 2, 'router_id': 3},
               ... : {...}
        }
        period: an integer
        timout: an integer
        """
        # Instance attributes
        self.__router_id = router_id
        self.__split_horizon_poison_reverse = True
        self.__input_ports = inputs
        self.__output_ports = outputs
        self.__regular_advertise_timer = time.time()
        self.__default_period = period
        self.__period = period
        self.__trigger_advertise_timer = time.time()
        self.__default_triggered_updates_period = period / 6
        self.__triggered_updates_period = period / 6
        self.__timeout_check_timer = time.time()
        self.__timeout = timeout
        self.__garbage_collection_time = period * 4
        self.__interface = None
        self.__routing_table = {}
        # Initialisation
        self.init_interface(inputs)
        self.init_routing_table()
        self.random_offset_period()


    def get_router_id(self):
        """
        router_id getter
        """
        return self.__router_id


    def set_router_id(self, new_id):
        """
        router_id setter
        """
        self.__router_id = new_id


    def get_input_ports(self):
        """
        router input_ports getter
        """
        return self.__input_ports


    def set_input_ports(self, new_inputs):
        """
        router input_ports setter
        """
        self.__input_ports = new_inputs
        self.init_interface(new_inputs)


    def get_output_ports(self):
        """
        router output_ports getter
        """
        return self.__output_ports


    def set_output_ports(self, new_outputs):
        """
        router output_ports setter
        """
        self.__output_ports = new_outputs


    def get_period(self):
        """
        router period getter
        """
        return self.__period


    def set_period(self, new_period):
        """
        router period setter
        """
        self.__period = new_period


    def get_timeout(self):
        """
        router timeout getter
        """
        return self.__timeout


    def set_timeout(self, new_timeout):
        """
        router timout setter
        """
        self.__timeout = new_timeout


    def get_interface(self):
        """
        router interface getter
        """
        return self.__interface


    def get_routing_table(self):
        """
        router routing_table getter
        """
        return self.__routing_table


    def print_routing_table(self):
        """
        Print the current self.__routing_table
        """
        print(routing_table_formatter(self.__router_id,
                                      self.__routing_table))


    def random_offset_period(self):
        """
        randomize self.__period +- TIMER_OFFSET
        """
        self.__period = self.__default_period +\
            random.uniform(-self.REGULAR_TIMER_OFFSET, \
                           +self.REGULAR_TIMER_OFFSET)
        print("Set Router regular update period to " + \
              f"{self.__period:.2f}")


    def random_triggered_updates_period(self):
        """
        randomize self.__triggered_updates_period
        """
        self.__triggered_updates_period = \
            self.__default_triggered_updates_period -\
            random.uniform(0, 0.4)
        print("Set Router triggered update period to " + \
              f"{self.__triggered_updates_period:.2f}")


    def init_interface(self, ports):
        """
        Create a new Interface object and set it as the default
        interface for the current Router object
        """
        self.__interface = Interface(ports)


    def init_routing_table(self):
        """
        Initialise the __routing_table attribute

        Route object format:
        route.next_hop: 2,
        route.metric: 1,
        route.timeout: 1234,
        route.garbage_collect_time: None(default)
        state: 'active'(default)
        """
        # Create a new Route object to router itself
        self_route = Route('-', 0, None)
        self.__routing_table[self.__router_id] = self_route


#------------------------------------------------------
# Above is the init implementation
#------------------------------------------------------


    def advertise_all_routes_periodically(self):
        """
        Call advertise_all_routes() periodcally by self.__period

        Use random.random() to calculate offset for self.__period
        in order to avoid synchronized update messages which can lead
        to unnecessary collisions on broadcast networks.
        """
        now = time.time()
        if now - self.__regular_advertise_timer >= self.__period:
            self.advertise_routes('all')
            self.print_routing_table()
            self.__regular_advertise_timer = now
            self.random_offset_period()


    def advertise_updated_routes(self):
        """
        advertise the updated routes to all neighbours
        """
        now = time.time()
        if now - self.__trigger_advertise_timer >= \
           self.__triggered_updates_period:
            self.advertise_routes('update')
            self.print_routing_table()
            self.__trigger_advertise_timer = now
            self.random_triggered_updates_period()


    def advertise_routes(self, mode):
        """
        parameter:
        mode: a string 'all' / 'update'
        get the latest advertising rip packet from
        update_packet() & triggered_packet() methods and
        advertise the packet to all the neighbours (ouput ports)

        need to add a parameter for updata_packet/triggered_packet
        """
        try:
            ports_num = len(self.__output_ports)
            if ports_num < 1:
                raise ValueError("No output port/socket available")
            for dest_port, metric_id in self.__output_ports.items():
                packet = self.update_packet(metric_id['router_id'], mode)
                if packet is None:
                    print("A packet without entry. Stop Sending")
                    return
                self.__interface.send(packet, dest_port)
                current_time = datetime.now().strftime('%H:%M:%S.%f')[:-4]
                if mode == 'all':
                    message = "Sends all routes to Router"
                else:
                    message = 'Sends triggred update to Router'
                print(message +
                     f"{metric_id['router_id']} " +
                     f"[{dest_port}] at {current_time}")
            # clear flags of "update"
            for route in self.__routing_table.values():
                if mode == 'update' and route.state == 'updated':
                    route.state = 'active'
        except ValueError as error:
            print(error)


    def update_packet(self, receiver_id, mode):
        """
        parameter:
        receiver_port

        Process the current routing table data and convert it into
        a rip format packet for advertise_all_routes() method
        """
        # Create RipEntries for all the routes
        entries = []
        for dest, route in self.__routing_table.items():

            if mode == "update" and route.state == "active":
                continue
            metric = route.metric
            # split_horizon_poison_reverse
            if self.__split_horizon_poison_reverse and\
               route.next_hop == receiver_id:
                metric = self.INFINITY
            entry = RipEntry(dest, metric)
            entries.append(entry)

        # Create RipPacket
        packet = RipPacket(entries, self.__router_id)
        packet_bytes = packet.packet_bytes()
        return packet_bytes


#----------------------------------------
# Above is sender implementation
#----------------------------------------


    def receive_routes(self):
        """
        Receive the routes update from neighbours (input ports)

        The implementation is in a while loop and should be called with
        a separate thread from the main thread
        """
        # The __interface only listen to the input ports
        # print(f"Listening to ports at {time.ctime()}")
        packets_list = self.__interface.receive()
        for raw_packet in packets_list:
            self.process_received_packet(raw_packet)


    def process_received_packet(self, raw_packet):
        """
        Process the received packet and call update_routing_table()
        if necessary

        Parameter: packet
        an array of bytes
        """
        # Check if raw_packet valid in RipPacket and RipEntry classes
        # Process the raw_packet if valid,
        # and return (True, RipPacket object)
        # otherwise, return (False, router_id)
        is_valid, rip_packet = RipPacket.decode_packet(raw_packet)
        if is_valid:
            # update routing_table if incoming packet is valid
            print(f'Received update from Router {rip_packet.router_id}')
            self.update_routing_table(rip_packet)
        else:
            # drop the packet if incoming packet is invalid
            print(f'Drop invalid packet from Router {rip_packet}')


    def update_routing_table(self, rip_packet):
        """
        check all the entries in rip_packet object, and update current
        routing table if necessary

        Parameter:
        rip_packet: a valid rip_packet object

        Reture: boolean
        return True if new route added, otherwise False
        """
        # get metric from sender
        sender_id = rip_packet.router_id
        metric_to_sender = None
        for neighbour in self.__output_ports.values():
            if neighbour['router_id'] == sender_id:
                metric_to_sender = neighbour['metric']
        for entry in rip_packet.entries:
            # update the metric for each entry
            # by adding the metric to sender
            # metric = min(metric + metric_to_sender, 16(infinity))
            updated_metric = min(entry.metric + metric_to_sender,
                                 self.INFINITY)
            #if route to dest is unavailable in __routing_table
            if updated_metric != self.INFINITY and\
               not entry.dest in self.__routing_table.keys():
                self.__routing_table[entry.dest] = \
                    Route(sender_id, updated_metric, time.time())
                # Triggered update for new route
                # self.__routing_table[entry.dest].state = 'updated'
                # print("Triggerd update for new route")
                # self.advertise_updated_routes()
            elif entry.dest in self.__routing_table:
                self.update_availabe_route(entry,
                                           updated_metric,
                                           sender_id)


    def update_availabe_route(self, entry, updated_metric, sender_id):
        """
        Parameters:
        entry: a RipEntry object
        sender_id: the router id from which the entry is sent
        """
        # if route to dest is available in __routing_table

        # 1. if packet is from the same router as
        # existing router, reinitialize the timeout anyway
        from_same_router = sender_id == \
            self.__routing_table[entry.dest].next_hop
        is_timeout = not \
            self.__routing_table[entry.dest].garbage_collect_time is \
            None
        if from_same_router:
            self.__routing_table[entry.dest].timeout = time.time()


        # 2. compare metrics
        new_metric = updated_metric
        old_metric = self.__routing_table[entry.dest].metric
        have_differnt_metrics = new_metric != old_metric
        is_lower_new_metric = new_metric < old_metric
        is_almost_timeout = \
            not self.__routing_table[entry.dest].timeout is None and \
            not is_timeout and \
            (time.time() - self.__routing_table[entry.dest].timeout) \
            >= self.__timeout / 2

        if from_same_router and have_differnt_metrics:
            self.__routing_table[entry.dest].metric = new_metric
            if not is_timeout and new_metric == self.INFINITY:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = time.time()
                # Triggered update for invalid route
                self.__routing_table[entry.dest].state = 'dying'
                print("triggered update for invalid route")
                self.advertise_updated_routes()
            elif is_timeout:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = None
                self.__routing_table[entry.dest].state = 'active'

        elif is_lower_new_metric:
            self.__routing_table[entry.dest].metric = new_metric
            self.__routing_table[entry.dest].next_hop = sender_id
            self.__routing_table[entry.dest].timeout = time.time()
            if is_timeout:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = None
                self.__routing_table[entry.dest].state = 'active'
            # Triggered update
            # self.__routing_table[entry.dest].state = 'updated'
            # print("triggered updated route from different router with lower metric")
            # self.advertise_updated_routes()
        elif not from_same_router and \
             not have_differnt_metrics and \
             not is_timeout and is_almost_timeout:
            self.__routing_table[entry.dest].next_hop = sender_id
            self.__routing_table[entry.dest].timeout = time.time()


#----------------------------------------
# Above is receiver implementation
#----------------------------------------

    def check_timeout_entries_periodically(self):
        """
        call check_timeout_entries() every default_period
        """
        now = time.time()
        if now - self.__timeout_check_timer >= self.__default_period:
            self.check_timeout_entries()
            self.__timeout_check_timer = now


    def check_timeout_entries(self):
        """
        Check the timeout of each entry in __routing_table

        if an entry is timeout, start its garbage_collect_time
        """
        current_time = datetime.now().strftime('%H:%M:%S.%f')[:-4]
        print(f"Checking timeout entries at {current_time}")

        entries_to_remove = []
        for dest_id, entry in self.__routing_table.items():
            if not entry.timeout is None and \
               entry.garbage_collect_time is None and \
               time.time() - entry.timeout >= self.__timeout:
                entry.garbage_collect_time = time.time()
                entry.metric = self.INFINITY
                entry.state = 'dying'
                # Triggered update
                print("Triggered update for invalid route")
                self.advertise_updated_routes()

            if not entry.garbage_collect_time is None and \
                (time.time() - entry.garbage_collect_time) \
                >= self.__garbage_collection_time:
                entries_to_remove.append(dest_id)

        for dest_id in entries_to_remove:
            self.__routing_table.pop(dest_id)
            print(f"Removed timeout route to {dest_id}")
            self.print_routing_table()


#-------------------------------------------------------
# Above is timeout and garbage_collection implementation
#-------------------------------------------------------

    def __str__(self):
        return ("Router: {0}\n"
                "Input Ports: {1}\n"
                "Output Ports: {2}\n"
                "Period: {3}\n"
                "Timeout: {4}").format(self.__router_id,
                                       self.__input_ports,
                                       self.__output_ports,
                                       self.__period,
                                       self.__timeout)
```