File: RIP_daemon.py

```
import sys
import time
import select
import socket as s

SOCKETS = []
PERIODIC_UPDATE_INTERVAL = 5       # seconds
ROUTE_TIMEOUT = 30                 # 6 Ã— periodic
GARBAGE_COLLECTION_INTERVAL = 20   # 4 Ã— periodic
ROUTING_TABLE_PRINT_INTERVAL = 15  # seconds

class Router:

    def __init__(self, id, input_ports, output_ports):
        self.id = id
        self.input_ports = input_ports
        self.output_ports = output_ports
        self.routing_table = {}
        self.send_socket = s.socket(s.AF_INET, s.SOCK_DGRAM)
        self.check_constraints()
        self.instantiate_ports()
        self.convert_output_ports()
        self.neighbors = {output_port[2]: output_port[1] for output_port in self.output_ports}
        self.initialise_routing_table()
        self.periodic_update_timer = time.time()
        self.routing_table_timer = time.time()
        self.route_timers = {}
        self.garbage_timers = {}

    def __str__(self):
        return f'Router ID: {self.id}\n' \
            f'Input Ports: {self.input_ports}\n' \
            f'Output Ports: {self.output_ports}\n' \
            f'Neighbors: {self.neighbors}\n' \

    def display_routing_table(self):
        print('--------------------------------')
        print(f'Router: {self.id}')
        for entry in self.routing_table.keys():
            print(
                f'Destination: {entry}, '
                f'Cost: {self.routing_table[entry][0]}, '
                f'Next hop: {self.routing_table[entry][1][0]}, '
                f'Is valid: {self.routing_table[entry][2]}'
            )
        print('--------------------------------')

    def instantiate_ports(self):
        for port in self.input_ports:
            try:
                sock = s.socket(s.AF_INET, s.SOCK_DGRAM)
                sock.setsockopt(s.SOL_SOCKET, s.SO_REUSEADDR, 1)  # Allow address reuse
                sock.bind(('', port))
                SOCKETS.append(sock)
                print(f"Successfully bound to input port {port}")
            except Exception as e:
                print(f"Failed to bind to input port {port}: {e}")

    def check_constraints(self):
        if self.id < 1 or self.id > 64000:
            raise Exception("Router-id must be between 1 and 64000 (inclusive).")
        for input in self.input_ports:
            if input < 1024 or input > 64000:
                raise Exception("Port numbers must be between 1024 and 64000 (inclusive).")
        for output in self.output_ports:
            output = output.split('-')
            port = int(output[0])
            if port < 1024 or port > 64000:
                raise Exception("Port numbers must be between 1024 and 64000 (inclusive).")

    def convert_output_ports(self):
        self.output_ports = [tuple(map(int, output.split('-'))) for output in self.output_ports]

    def initialise_routing_table(self):
        # Routing table layout - Destination: cost, (next hop, port), is_valid
```

```
    for output in self.output_ports:
        self.routing_table[output[2]] = (output[1], (output[2], output[0]), True)

def construct_packet(self, neighbor_id):
    packet = bytearray()

    packet.append(2) #COMMAND: reponse
    packet.append(2) #Version: 2
    packet += self.id.to_bytes(2, 'big') #Router-ID in place of zero-byte field

    for entry in self.routing_table.keys():
        dest_id = entry
        cost, (next_hop, _), is_valid = self.routing_table[entry]

        if next_hop == neighbor_id or not is_valid: #split-horizon with poison reverse
            cost = 16

        packet += (2).to_bytes(2, 'big') #address family identifier (AF_INET)
        packet += (0).to_bytes(2, 'big') #Route tag (set to 0)
        packet += dest_id.to_bytes(4, 'big')
        packet += bytes(4) #Subnet mask not used
        packet += next_hop.to_bytes(4, 'big')
        packet += cost.to_bytes(4, 'big')

    return packet

def decode_packet(self, packet):
    """
    Decodes a received RIP packet and processes its routes.
    """
    if packet[0] != 2:  # Check Command field
        print('Invalid packet header, Command incorrect. Packet dropped')
        return
    if packet[1] != 2:  # Check version field
        print('Invalid packet header, version is not 2. Packet dropped')
        return

    sender_id = int.from_bytes(packet[2:4], 'big')  # Extract sender ID

    self.route_timers[sender_id] = time.time()  # Reset the timer for this sender
    index = 4  # Start reading RIP entries
    routes = []

    while index < len(packet):
        if int.from_bytes(packet[index:index+2], 'big') != 2:
            print("Invalid RIPv2 entry with incorrect Addr Family. Packet dropped.")
            return
        index += 2

        if int.from_bytes(packet[index:index+2], 'big') != 0:
            print("Invalid RIPv2 entry with Route Tag. Packet dropped.")
            return
        index += 2

        dest_id = int.from_bytes(packet[index:index+4], 'big')
        index += 4

        if int.from_bytes(packet[index:index+4], 'big') != 0:
            print('Invalid RIPv2 entry, Subnet mask should be 0. Packet dropped.')
            return
        index += 4

        next_hop = int.from_bytes(packet[index:index+4], 'big')
        index += 4

        cost = int.from_bytes(packet[index:index+4], 'big')
        index += 4

        try:
            # Check received constraints
            self.validate_route_entry(sender_id, dest_id, next_hop, cost)
            routes.append((sender_id, dest_id, next_hop, cost))
        except ValueError as e:
            print(e)

    # Process the routes (update the routing table)
```

```
    self.calculate_routes(routes)

def send_packets(self):
    for output in self.output_ports:
        neighbor_port = output[0]
        neighbor_id = output[2]

        # If the neighbor is not in the routing table, we don't send a packet
        if neighbor_id not in self.routing_table:
            continue

        packet = self.construct_packet(neighbor_id)
        self.send_socket.sendto(packet, ('localhost', neighbor_port))

def update_timers(self):
    # Periodic updates
    if time.time() - self.periodic_update_timer >= PERIODIC_UPDATE_INTERVAL:
        self.send_packets()
        #print("Periodic update: Packets sent.")
        self.periodic_update_timer = time.time()

    # Print routing table
    if time.time() - self.routing_table_timer >= ROUTING_TABLE_PRINT_INTERVAL:
        self.display_routing_table()
        self.routing_table_timer = time.time()

    triggered_update_needed = False  # Flag to track if a triggered update is needed

    for entry in list(self.routing_table.keys()):  # Update the route timers
        if entry not in self.route_timers:
            self.route_timers[entry] = time.time()
        else:
            if self.routing_table[entry][2] == True:  # Only consider valid routes
                if time.time() - self.route_timers[entry] >= ROUTE_TIMEOUT:
                    print(f"Route to {entry} has timed out and is now invalid.")
                    # Set the route to invalid
                    self.routing_table[entry] = (16, self.routing_table[entry][1], False)
                    del self.route_timers[entry]
                    self.garbage_timers[entry] = time.time()  # Add garbage timer
                    triggered_update_needed = True  # Mark that a triggered update is needed

    if triggered_update_needed:
        self.send_packets()  # Send a triggered update immediately
        print("Triggered update: Packets sent.")

    # Update the garbage collection timers and delete the routes if necessary
    for entry in list(self.garbage_timers.keys()):
        if time.time() - self.garbage_timers[entry] >= GARBAGE_COLLECTION_INTERVAL:
            print(f"Route to {entry} has been garbage collected.")
            del self.route_timers[entry]
            del self.routing_table[entry]
            del self.garbage_timers[entry]

def find_output_port(self, neighbor_id):
    """
    Given a neighbor router ID, find the correct output port number
    to reach that neighbor. Returns None if not found.
    """
    for output_port in self.output_ports:
        if output_port[2] == neighbor_id:
            return output_port[0]
    return None


def calculate_routes(self, routes):
    """
    Updates the routing table based on received routes.
    Ensures no loops, ignores unreachable routes (cost=16),
    and updates the table with the cheapest path.
    Sends triggered updates when routes become invalid.
    """
    for sender_id, dest_id, next_hop, cost in routes:

        # Corresponding output port for sender_id
        correct_port = self.find_output_port(sender_id)
```

```
        # Routes with cost 16 are poisoned updates and need to be addressed
        if cost == 16:
            if dest_id in self.routing_table:
                # Only if it's still valid and the next hop was the sender
                if self.routing_table[dest_id][0] != 16 \
                and self.routing_table[dest_id][1][0] == sender_id:
                    print(f"Invalidating route to {dest_id} via {sender_id}")
                    self.routing_table[dest_id] = (16, (sender_id, correct_port), False)
                    self.garbage_timers[dest_id] = time.time()
            continue

        # This occurs when a router is revived, also check if its revived after timeout
        # but before garbage collection
        if sender_id not in self.routing_table and sender_id in self.neighbors \
        or sender_id in self.garbage_timers and sender_id in self.neighbors:
            # cost is from neighbors dictionary
            self.routing_table[sender_id] = (self.neighbors[sender_id], (sender_id, correct_port), True)
            print(f"Router {sender_id} has come online")
            self.route_timers[sender_id] = time.time()
            if dest_id in self.garbage_timers:
                    del self.garbage_timers[dest_id]  # Remove from garbage timers

        # Reset the route timer for the destination if the sender is the next hop
        if dest_id in self.routing_table and self.routing_table[dest_id][1][0] == sender_id:
            self.route_timers[dest_id] = time.time()
            if dest_id in self.garbage_timers:
                del self.garbage_timers[dest_id]  # Reset garbage timer

        # Reset the route timer for the sender
        if sender_id in self.route_timers:
            self.route_timers[sender_id] = time.time()
            if sender_id in self.garbage_timers:
                del self.garbage_timers[sender_id]

        # Calculate the total cost
        # Only if sender is valid
        if sender_id in self.routing_table and self.routing_table[sender_id][2]:
            total_cost = cost + self.routing_table[sender_id][0]
        else:
            total_cost = cost  # Default to the received cost if sender is not in the table

        # Ignore if total cost exceeds the maximum allowed cost
        if total_cost > 16:
            continue

        # Check for loops: avoid adding a route back to the sender
        if dest_id == self.id:
            continue
        if next_hop == self.id:
            continue

        # Update the routing table if:
        # 1. The destination is not in the table, or
        # 2. The new route has a lower cost, or
        # 3. The next hop for the destination is the sender (to refresh the route)
        if (dest_id not in self.routing_table or
            total_cost < self.routing_table[dest_id][0] or
            (self.routing_table[dest_id][1][0] == sender_id and total_cost < self.routing_table[dest_id][0])):

            self.routing_table[dest_id] = (total_cost, (sender_id, correct_port), True)
            self.route_timers[dest_id] = time.time()  # Reset the timer for this route
            if dest_id in self.garbage_timers:
                del self.garbage_timers[dest_id]  # Remove from garbage timers

    def validate_route_entry(self, sender_id, dest_id, next_hop, cost):
        """
        Validates a single route entry.
        Ensures that sender_id, dest_id, next_hop, and cost are within valid ranges.
        """
        if sender_id < 1 or sender_id > 64000:
            raise ValueError(f"Invalid sender ID {sender_id}. Must be between 1 and 64000.")
        if dest_id < 1 or dest_id > 64000:
            raise ValueError(f"Invalid destination ID {dest_id}. Must be between 1 and 64000.")
        if next_hop < 1 or next_hop > 64000:
            raise ValueError(f"Invalid next hop {next_hop}. Must be between 1 and 64000.")
        if cost < 1 or cost > 16:
```

```
            raise ValueError(f"Invalid cost {cost}. Must be between 1 and 16.")
        return True

def read_config_file(filename):
    #Reads a config file for a single router and returns a dictionary with the configuration.

    with open(filename, 'r') as file:
        lines = [line.strip() for line in file if line.strip()]

    if len(lines) != 3:
        raise Exception(f"Config file '{filename}' must contain exactly 3 non-empty lines.")

    config = {}

    #router-id
    parts = lines[0].split()
    if parts[0].lower() != 'router-id' or len(parts) != 2:
        raise Exception(f"Line 1 in '{filename}' must be: router-id <id>")
    try:
        config['router-id'] = int(parts[1])
        if config['router-id'] < 1 or config['router-id'] > 64000:
            raise ValueError("Router ID must be between 1 and 64000 (inclusive).")
    except ValueError:
        raise ValueError("Router ID must be an integer.")

    #input-ports
    parts = lines[1].split()
    if parts[0].lower() != 'input-ports' or len(parts) < 2:
        raise Exception(f"Line 2 in '{filename}' must be: input-ports <port1> <port2> ...")
    try:
        config['input-ports'] = list(map(int, parts[1:]))
        for port in config['input-ports']:
            if port < 1024 or port > 64000:
                raise ValueError("Port numbers must be between 1024 and 64000 (inclusive).")
    except ValueError:
        raise ValueError("Input ports must be integers.")

    #output-ports
    parts = lines[2].split()
    if parts[0].lower() != 'output-ports' or len(parts) < 2:
        raise ValueError(f"Line 3 in '{filename}' must be: output-ports <port-cost-id> ...")
    config['output-ports'] = parts[1:]
    for output in config['output-ports']:
        parts = output.split('-')
        if len(parts) != 3:
            raise Exception("Output ports must be in the format <port-cost-id>.")
        try:
            port = int(parts[0])
            cost = int(parts[1])
            id = int(parts[2])
            if port < 1024 or port > 64000:
                raise ValueError("Port numbers must be between 1024 and 64000 (inclusive).")
            if cost < 1 or cost > 16:
                raise ValueError("Cost must be between 1 and 16 (inclusive).")
            if id < 1 or id > 64000:
                raise ValueError("Router ID must be between 1 and 64000 (inclusive).")
        except ValueError:
            raise ValueError("Output ports must be integers.")

    return config


def routing_loop():
    ROUTER.send_packets()

    while True:
        readable, _, _ = select.select(SOCKETS, [], [], 1)
        if readable:
            for sock in readable:
                try:
                    data, _ = sock.recvfrom(4096)
                    ROUTER.decode_packet(data)
                except Exception as e:
                    print(f"Error receiving from socket {sock.getsockname()}: {e}")

        # Always check timers
```

```
        ROUTER.update_timers()


def main():
    if len(sys.argv) != 2:
        raise Exception("Invalid command line arguments.")
        sys.exit(1)

    config_file = sys.argv[1]
    config = read_config_file(config_file)

    router_id = config.get('router-id')
    input_ports = config.get('input-ports', [])
    output_ports = config.get('output-ports', [])

    global ROUTER
    ROUTER = Router(router_id, input_ports, output_ports)

    print(ROUTER)
    ROUTER.display_routing_table()

    routing_loop()


if __name__ == "__main__":
    main()
```