```python
import math
import random
import time

from configmanager import routerid_is_valid, metric_is_valid


TIME_MULTIPLIER = 6

PERIODIC_UPDATE_DELAY =     30 / TIME_MULTIPLIER
TRIGGERED_UPDATE_DELAY =    5 / TIME_MULTIPLIER
ENTRY_TIMEOUT_DELAY =       180 / TIME_MULTIPLIER
GARBAGE_COLLECTION_DELAY =  120 / TIME_MULTIPLIER


INFINITE_METRIC = 16

POISONED_REVERSE = True


class RipManager:
    """This class manages the Routing Information Protocol. The routing
    table is a dictionary where the key is the destination and the value
    is a RoutingTableEntry. e.g. {destination: RoutingTableEntry}
    A RoutingTableEntry contains info about the next_hop, metric, and timeouts.
    """

    def __init__(self, debug_func, config, output_socket):
        global debug
        debug = debug_func
        self.our_routerid = config.router_id
        self.output_routers = config.outputs
        self.socket = output_socket

        self.routing_table = {}
        self.next_periodic_update = time.time()
        self.triggered_update_pending = False
        self.next_triggered_update = 0


    def __str__(self):
        lines = f'''Router {self.our_routerid:<16} Routing Table
+-------------+----------+--------+------------+--------------+
| destination | next hop | metric | update due | deletion due |
+-------------+----------+--------+------------+--------------+
'''
        for dest, entry in sorted(self.routing_table.items()):
            deletion_due = entry.deletion_due_in()
            if deletion_due == math.inf:
                deletion_due = ''
            else:
                    deletion_due = int(deletion_due)
                lines += f'| {dest:>11} | {entry.next_hop:>8} | {entry.metric:>6} '
                lines += f'| {entry.update_due_in():>10.0f} | {deletion_due:>12} |\n'
            lines += '+------------+---------+--------+-----------+-------------+\n'
            return lines


    def table_list(self):
        """Return a list of routing table entries. Does not include
        timeout times, but does include a deletion process flag.
        Used for automatic testing and to detect routing table changes.
        """
        return [[d, e.next_hop, e.metric, e.deletion_process_underway()] for d,e in ⏎
         sorted(self.routing_table.items())]


    def next_timeout(self):
        """Return the time in seconds (as a float) until the next timeout.
        Timeouts include periodic update messages (every 30 seconds), and
        triggered updates related to routing table entries. Triggered
        updates occur if a routing table entry hasn't been updated for
        180 seconds, or if a routing table entry has been garbage
        collected for 120 seconds.
        """
        next_periodic_update_in = self.next_periodic_update - time.time()
        next_periodic_update_in = max(0, next_periodic_update_in)

        timeouts = [next_periodic_update_in]
        for entry in self.routing_table.values():
            timeouts.append(entry.next_timeout())

        if self.triggered_update_pending:
            next_triggered_update_in = self.next_triggered_update - time.time()
            timeouts.append(next_triggered_update_in)

        smallest_timeout = min(timeouts)
        return max(0, smallest_timeout)


    def incoming_message(self, message):
        """Process an incoming UDP packet."""
        try:
            rip_packet = RipPacket(message)
        except AssertionError as e:
            debug(f"Received invalid packet: {e}")
            return

        next_hop = rip_packet.routerid
        if next_hop not in self.output_routers:
            debug(f'Received packet from unknown router {next_hop}')
            return
        _, metric_to_next_hop = self.output_routers[next_hop]
        self.add_to_table(next_hop, next_hop, metric_to_next_hop) # add sender to routing table

        for rip_entry in rip_packet.entries:
            metric = min(metric_to_next_hop + rip_entry.metric, INFINITE_METRIC)
            self.add_to_table(rip_entry.routerid, next_hop, metric)


    def add_to_table(self, destination, next_hop, metric):
        """Update or add a table entry.
        Only add a new entry if the metric isn't infinity.
        The RIP assignment says to not send a triggered message for
        metric updates or new routes.
        """
        if destination == self.our_routerid:
            return # don't add ourself to our routing table
        if destination in self.routing_table.keys():
            reason = self.routing_table[destination].update_entry(next_hop, metric)
            if reason:
                debug(f'{self.our_routerid} updating routing table entry for destination {destination}:')
                debug(f'    {reason}')
        elif metric < INFINITE_METRIC:
            debug(f'{self.our_routerid} added a new route to destination {destination} next-hop {next_hop} metric {metric}')
            self.routing_table[destination] = RoutingTableEntry(next_hop, metric)


    def send_any_updates(self):
        """Check if a periodic or triggered update should be sent.
        Triggered updates only for when routes become invalid (route
        deleted or metric set to 16), not for new/updated routes.
        After sending a triggered update, don't send future triggered
        updates for 1 to 5 seconds.
        """
        to_delete = []
        for destination, entry in self.routing_table.items():
            if entry.should_delete():
                to_delete.append(destination)
                self.triggered_update_pending = True
            elif entry.should_begin_deletion():
                debug(f'Starting deletion process for destination {destination}')
                entry.begin_deletion()
                self.triggered_update_pending = True

        for dest in to_delete: # since you cant delete entries while iterating over them
            debug(f'Deleting destination {dest}')
            del self.routing_table[dest]

        periodic_update = time.time() >= self.next_periodic_update
        triggered_update = self.triggered_update_pending and time.time() >= self.next_triggered_update
        if periodic_update or triggered_update:
            self.send_response_messages()


    def send_response_messages(self):
        """Send a periodic/triggered update message.
        Send a response message to all neighbours
        containing the complete routing table (as set by assignment
        specifications) utilising split-horizon with poisoned-reverse.
        The next periodic update message should be sent in
        30 seconds +/- up to 5 seconds (1/6th of 30s) randomly.
        The next triggered update message should be sent in
        1 (1/5th of 5 seconds) to 5 seconds randomly.
        """
        for router_id, [port, metric] in self.output_routers.items():
            packets = self.build_packets(router_id)
            for p in packets:
                try:
                    RipPacket(p)
                except AssertionError as e:
                    debug(f'Sending invalid packet: {e}')
                self.socket.sendto(p, ('127.0.0.1', port))

        self.next_periodic_update = (time.time() +
            PERIODIC_UPDATE_DELAY +
            random.uniform(-PERIODIC_UPDATE_DELAY/6, PERIODIC_UPDATE_DELAY/6))
        self.triggered_update_pending = False
        self.next_triggered_update = (time.time() +
            random.uniform(TRIGGERED_UPDATE_DELAY/5, TRIGGERED_UPDATE_DELAY))


    def build_packets(self, destination_router_id):
        """Return response message packets to be sent to the defined
        router. Utilises split-horizon with optional poisoned-reverse.
        """
        packets = []

        packet = self.empty_rip_packet()
        packet += rip_entry(destination_router_id, INFINITE_METRIC) # always add the receiver as a rip entry with inf metric

        for destination, entry in self.routing_table.items():
            metric = entry.metric
            if entry.next_hop == destination_router_id:
                if POISONED_REVERSE:
                    metric = INFINITE_METRIC
                else:
                    continue # don't add the entry

            if len(packet) >= (4 + 20*25): # if 25 entries
                packets.append(packet)
                packet = self.empty_rip_packet()

            packet += rip_entry(destination, metric)

        packets.append(packet)
        return packets


    def empty_rip_packet(self):
        """Return an empty rip packet (headers only).
        RFC all-zeros field is used for the routerid by assignment specs.
        """
        packet = bytearray(4)
        packet[0] = 2 # command
        packet[1] = 2 # version
        packet[2:4] = self.our_routerid.to_bytes(2)
        return packet


def rip_entry(destination, metric):
    """Return a rip entry for use in a rip packet."""
    entry = bytearray(20)
    entry[0:2] = (2).to_bytes(2) # address family identifier
    entry[4:8] = destination.to_bytes(4)
    entry[16:20] = metric.to_bytes(4)
    return entry


class RoutingTableEntry:
    """A single entry for use in the routing table.
    The RFC's 'garbage-collection' is called 'deletion' here.
    Route change flags are not used due to us not sending triggered
    updates for route metric changes according to the RIP assignment.
    """

    def __init__(self, next_hop, metric):
        self.next_hop = next_hop
        self.metric = metric
        self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
        self.time_deletion_due = None


    def deletion_process_underway(self):
        return self.time_deletion_due != None


    def over_halfway_to_update_due(self):
        due_in = self.time_update_due - time.time()
        return due_in <= ENTRY_TIMEOUT_DELAY/2


    def update_due_in(self):
        """Time in seconds until an update is due."""
        due_in = self.time_update_due - time.time()
        return max(0, due_in)


    def deletion_due_in(self):
        """Time in seconds until deletion is due."""
        due_in = math.inf
        if self.deletion_process_underway():
            due_in = self.time_deletion_due - time.time()
        return max(0, due_in)


    def next_timeout(self):
        """Return the time in seconds (as a float) until the next timeout."""
        smallest_time = min(self.update_due_in(), self.deletion_due_in())
        return max(0, smallest_time)


    def update_entry(self, next_hop, new_metric):
        """If the deletion process is underway for a route, replace it.
        If the new metric is 16 then don't add it (no better than current).
        Return a string describing the reason for change.
        """
        reason = None
        update_timeouts = False

        if next_hop == self.next_hop:
            update_timeouts = True
            if self.metric != new_metric:
                reason = f'updated next-hop {self.next_hop} metric from {self.
                metric} to {new_metric} (update is from next-hop)'
                self.metric = new_metric

        elif new_metric < self.metric:
            reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
            next_hop} ({new_metric}) (better metric)'
            update_timeouts = True
            self.next_hop = next_hop
            self.metric = new_metric

        # RFC section 3.9.2 heuristic
        elif (new_metric != INFINITE_METRIC and
              new_metric == self.metric and
              self.over_halfway_to_update_due()):
            update_timeouts = True
            reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
```

```python
                next_hop} ({new_metric}) (over halfway to update due)'
                self.next_hop = next_hop
                self.metric = new_metric

            if update_timeouts:
                self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
                if self.metric < INFINITE_METRIC:
                    self.time_deletion_due = None

            return reason


        def should_begin_deletion(self):
            """Return True if the deletion process should be started.
            Deletion process should not be started if it is already underway.
            """
            if not self.deletion_process_underway():
                return (self.metric >= INFINITE_METRIC or
                        time.time() >= self.time_update_due)
            return False


        def begin_deletion(self):
            assert self.deletion_process_underway() is False
            self.metric = INFINITE_METRIC
            self.time_deletion_due = time.time() + GARBAGE_COLLECTION_DELAY


        def should_delete(self):
            """Return True if this entry should be deleted immediately."""
            if self.deletion_process_underway():
                return time.time() >= self.time_deletion_due
            return False


    class RipPacket:
        """This class represents a validated RIP request packet.
        If a RIP packet entry is invalid, ignore it.
        1 byte - command (must be 2)
        1 byte - version (must be 2)
        2 bytes - routerid (all-zeros in RIP RFC)
        20 bytes - rip entry (1 to 25 lots of these)
        """
        def __init__(self, packet):
            self.validate_rip_packet(packet)
            self.routerid = int.from_bytes(packet[2:4])
            self.entries = []
            for i in range(4, len(packet), 20):
                try:
                    self.entries.append(RipEntry(packet[i: i+20]))
                except AssertionError as e:
                    debug(f'RIP packet entry error: {e}')

        def __str__(self):
            lines = f'''packet:
        Source: {self.routerid}'''
            for entry in self.entries:
                lines += f"""
            {entry}"""
            if not self.entries:
                lines += f"""
            <EMPTY PACKET>"""
            return lines + '\n'

        def validate_rip_packet(self, packet):
            """Raise an AssertionError if the packet is invalid.
            Does not check the validity of the contained rip entries.
            """
            assert len(packet) >= 4+20, f"packet length invalid: {len(packet)}"
            assert len(packet) <= 4+20*25, f"packet length invalid: {len(packet)}"
            assert (len(packet) - 4) % 20 == 0, f"packet length invalid: {len(packet)}"
            assert packet[0] == 2, "command field not 2"
            assert packet[1] == 2, "version field not 2"
            routerid = int.from_bytes(packet[2:4])
            assert routerid_is_valid(routerid), f"router-id invalid {routerid}"


    class RipEntry:
        """This class represents a validated RIP entry from a RIP packet.
        2 bytes - address family (ignore)
        2 bytes - all zeros
        4 bytes - routerid (IPv4 in RIP RFC)
        8 bytes - all zeros
        4 bytes - metric
        """
        def __init__(self, entry):
            self.validate_rip_entry(entry)
            self.routerid = int.from_bytes(entry[4:8])
            self.metric = int.from_bytes(entry[16:20])

        def __str__(self):
            return f'router-id: {self.routerid} metric: {self.metric}'

        def validate_rip_entry(self, entry):
            """Raise an AssertionError if the rip entry is invalid."""
            assert len(entry) == 20, "RIP entry length not 20"
            assert int.from_bytes(entry[0:2]) == 2, "address family must be 2"
            assert int.from_bytes(entry[2:4]) == 0, "field must be all zeros"
            routerid = int.from_bytes(entry[4:8])
            assert routerid_is_valid(routerid), f"router-id invalid {routerid}"
            assert int.from_bytes(entry[8:16]) == 0, "field must be all zeros"
            metric = int.from_bytes(entry[16:20])
            assert metric_is_valid(metric), f"metric invalid {metric}"

```