```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:
```

```
52                  deletion_due = int(deletion_due)
53              lines += f'| {dest:>11} | {entry.next_hop:>8} | {entry.metric:>6} '
54              lines += f'| {entry.update_due_in():>10.0f} | {deletion_due:>12} |\n'
55          lines += '+-------------+---------+--------+-----------+--------------+\n'
56          return lines
57
58
59      def table_list(self):
60          """Return a list of routing table entries. Does not include
61          timeout times, but does include a deletion process flag.
62          Used for automatic testing and to detect routing table changes.
63          """
64          return [[d, e.next_hop, e.metric, e.deletion_process_underway()] for d,e in ↵
             sorted(self.routing_table.items())]
65
66
67      def next_timeout(self):
68          """Return the time in seconds (as a float) until the next timeout.
69          Timeouts include periodic update messages (every 30 seconds), and
70          triggered updates related to routing table entries. Triggered
71          updates occur if a routing table entry hasn't been updated for
72          180 seconds, or if a routing table entry has been garbage
73          collected for 120 seconds.
74          """
75          next_periodic_update_in = self.next_periodic_update - time.time()
76          next_periodic_update_in = max(0, next_periodic_update_in)
77
78          timeouts = [next_periodic_update_in]
79          for entry in self.routing_table.values():
80              timeouts.append(entry.next_timeout())
81
82          if self.triggered_update_pending:
83              next_triggered_update_in = self.next_triggered_update - time.time()
84              timeouts.append(next_triggered_update_in)
85
86          smallest_timeout = min(timeouts)
87          return max(0, smallest_timeout)
88
89
90      def incoming_message(self, message):
91          """Process an incoming UDP packet."""
92          try:
93              rip_packet = RipPacket(message)
94          except AssertionError as e:
95              debug(f"Received invalid packet: {e}")
96              return
97
98          next_hop = rip_packet.routerid
99          if next_hop not in self.output_routers:
100             debug(f'Received packet from unknown router {next_hop}')
101             return
```

```
102             _, metric_to_next_hop = self.output_routers[next_hop]
103             self.add_to_table(next_hop, next_hop, metric_to_next_hop) # add sender to
                routing table
104
105             for rip_entry in rip_packet.entries:
106                 metric = min(metric_to_next_hop + rip_entry.metric, INFINITE_METRIC)
107                 self.add_to_table(rip_entry.routerid, next_hop, metric)
108
109
110         def add_to_table(self, destination, next_hop, metric):
111             """Update or add a table entry.
112             Only add a new entry if the metric isn't infinity.
113             The RIP assignment says to not send a triggered message for
114             metric updates or new routes.
115             """
116             if destination == self.our_routerid:
117                 return # don't add ourself to our routing table
118             if destination in self.routing_table.keys():
119                 reason = self.routing_table[destination].update_entry(next_hop, metric)
120                 if reason:
121                     debug(f'{self.our_routerid} updating routing table entry for
                    destination {destination}:')
122                     debug(f'    {reason}')
123             elif metric < INFINITE_METRIC:
124                 debug(f'{self.our_routerid} added a new route to destination {
                    destination} next-hop {next_hop} metric {metric}')
125                 self.routing_table[destination] = RoutingTableEntry(next_hop, metric)
126
127
128         def send_any_updates(self):
129             """Check if a periodic or triggered update should be sent.
130             Triggered updates only for when routes become invalid (route
131             deleted or metric set to 16), not for new/updated routes.
132             After sending a triggered update, don't send future triggered
133             updates for 1 to 5 seconds.
134             """
135             to_delete = []
136             for destination, entry in self.routing_table.items():
137                 if entry.should_delete():
138                     to_delete.append(destination)
139                     self.triggered_update_pending = True
140                 elif entry.should_begin_deletion():
141                     debug(f'Starting deletion process for destination {destination}')
142                     entry.begin_deletion()
143                     self.triggered_update_pending = True
144
145             for dest in to_delete: # since you cant delete entries while iterating
                over them
146                 debug(f'Deleting destination {dest}')
147                 del self.routing_table[dest]
148
```

```
149             periodic_update = time.time() >= self.next_periodic_update
150             triggered_update = self.triggered_update_pending and time.time() >= self.
                next_triggered_update
151             if periodic_update or triggered_update:
152                 self.send_response_messages()
153
154
155         def send_response_messages(self):
156             """Send a periodic/triggered update message.
157             Send a response message to all neighbours
158             containing the complete routing table (as set by assignment
159             specifications) utilising split-horizon with poisoned-reverse.
160             The next periodic update message should be sent in
161             30 seconds +/- up to 5 seconds (1/6th of 30s) randomly.
162             The next triggered update message should be sent in
163             1 (1/5th of 5 seconds) to 5 seconds randomly.
164             """
165             for router_id, [port, metric] in self.output_routers.items():
166                 packets = self.build_packets(router_id)
167                 for p in packets:
168                     try:
169                         RipPacket(p)
170                     except AssertionError as e:
171                         debug(f'Sending invalid packet: {e}')
172                     self.socket.sendto(p, ('127.0.0.1', port))
173
174             self.next_periodic_update = (time.time() +
175                 PERIODIC_UPDATE_DELAY +
176                 random.uniform(-PERIODIC_UPDATE_DELAY/6, PERIODIC_UPDATE_DELAY/6))
177             self.triggered_update_pending = False
178             self.next_triggered_update = (time.time() +
179                 random.uniform(TRIGGERED_UPDATE_DELAY/5, TRIGGERED_UPDATE_DELAY))
180
181
182         def build_packets(self, destination_router_id):
183             """Return response message packets to be sent to the defined
184             router. Utilises split-horizon with optional poisoned-reverse.
185             """
186             packets = []
187
188             packet = self.empty_rip_packet()
189             packet += rip_entry(destination_router_id, INFINITE_METRIC) # always add
                the receiver as a rip entry with inf metric
190
191             for destination, entry in self.routing_table.items():
192                 metric = entry.metric
193                 if entry.next_hop == destination_router_id:
194                     if POISONED_REVERSE:
195                         metric = INFINITE_METRIC
196                     else:
197                         continue # don't add the entry
```

```
198
199             if len(packet) >= (4 + 20*25): # if 25 entries
200                 packets.append(packet)
201                 packet = self.empty_rip_packet()
202
203             packet += rip_entry(destination, metric)
204
205         packets.append(packet)
206         return packets
207
208
209     def empty_rip_packet(self):
210         """Return an empty rip packet (headers only).
211         RFC all-zeros field is used for the routerid by assignment specs.
212         """
213         packet = bytearray(4)
214         packet[0] = 2 # command
215         packet[1] = 2 # version
216         packet[2:4] = self.our_routerid.to_bytes(2)
217         return packet
218
219
220 def rip_entry(destination, metric):
221     """Return a rip entry for use in a rip packet."""
222     entry = bytearray(20)
223     entry[0:2] = (2).to_bytes(2) # address family identifier
224     entry[4:8] = destination.to_bytes(4)
225     entry[16:20] = metric.to_bytes(4)
226     return entry
227
228
229 class RoutingTableEntry:
230     """A single entry for use in the routing table.
231     The RFC's 'garbage-collection' is called 'deletion' here.
232     Route change flags are not used due to us not sending triggered
233     updates for route metric changes according to the RIP assignment.
234     """
235
236     def __init__(self, next_hop, metric):
237         self.next_hop = next_hop
238         self.metric = metric
239         self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
240         self.time_deletion_due = None
241
242
243     def deletion_process_underway(self):
244         return self.time_deletion_due != None
245
246
247     def over_halfway_to_update_due(self):
248         due_in = self.time_update_due - time.time()
```

```
249             return due_in <= ENTRY_TIMEOUT_DELAY/2
250
251
252         def update_due_in(self):
253             """Time in seconds until an update is due."""
254             due_in = self.time_update_due - time.time()
255             return max(0, due_in)
256
257
258         def deletion_due_in(self):
259             """Time in seconds until deletion is due."""
260             due_in = math.inf
261             if self.deletion_process_underway():
262                 due_in = self.time_deletion_due - time.time()
263             return max(0, due_in)
264
265
266         def next_timeout(self):
267             """Return the time in seconds (as a float) until the next timeout."""
268             smallest_time = min(self.update_due_in(), self.deletion_due_in())
269             return max(0, smallest_time)
270
271
272         def update_entry(self, next_hop, new_metric):
273             """If the deletion process is underway for a route, replace it.
274             If the new metric is 16 then don't add it (no better than current).
275             Return a string describing the reason for change.
276             """
277             reason = None
278             update_timeouts = False
279
280             if next_hop == self.next_hop:
281                 update_timeouts = True
282                 if self.metric != new_metric:
283                     reason = f'updated next-hop {self.next_hop} metric from {self.
                        metric} to {new_metric} (update is from next-hop)'
284                     self.metric = new_metric
285
286             elif new_metric < self.metric:
287                 reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
                    next_hop} ({new_metric}) (better metric)'
288                 update_timeouts = True
289                 self.next_hop = next_hop
290                 self.metric = new_metric
291
292             # RFC section 3.9.2 heuristic
293             elif (new_metric != INFINITE_METRIC and
294                   new_metric == self.metric and
295                   self.over_halfway_to_update_due()):
296                 update_timeouts = True
297                 reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
```

```
                next_hop} ({new_metric}) (over halfway to update due)'
298             self.next_hop = next_hop
299             self.metric = new_metric
300
301         if update_timeouts:
302             self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
303             if self.metric < INFINITE_METRIC:
304                 self.time_deletion_due = None
305
306         return reason
307
308
309     def should_begin_deletion(self):
310         """Return True if the deletion process should be started.
311         Deletion process should not be started if it is already underway.
312         """
313         if not self.deletion_process_underway():
314             return (self.metric >= INFINITE_METRIC or
315                     time.time() >= self.time_update_due)
316         return False
317
318
319     def begin_deletion(self):
320         assert self.deletion_process_underway() is False
321         self.metric = INFINITE_METRIC
322         self.time_deletion_due = time.time() + GARBAGE_COLLECTION_DELAY
323
324
325     def should_delete(self):
326         """Return True if this entry should be deleted immediately."""
327         if self.deletion_process_underway():
328             return time.time() >= self.time_deletion_due
329         return False
330
331
332 class RipPacket:
333     """This class represents a validated RIP request packet.
334     If a RIP packet entry is invalid, ignore it.
335     1 byte - command (must be 2)
336     1 byte - version (must be 2)
337     2 bytes - routerid (all-zeros in RIP RFC)
338     20 bytes - rip entry (1 to 25 lots of these)
339     """
340     def __init__(self, packet):
341         self.validate_rip_packet(packet)
342         self.routerid = int.from_bytes(packet[2:4])
343         self.entries = []
344         for i in range(4, len(packet), 20):
345             try:
346                 self.entries.append(RipEntry(packet[i: i+20]))
347             except AssertionError as e:
```

```
348                 debug(f'RIP packet entry error: {e}')
349
350     def __str__(self):
351         lines = f'''packet:
352     Source: {self.routerid}'''
353         for entry in self.entries:
354             lines += f"""
355         {entry}"""
356         if not self.entries:
357             lines += f"""
358         <EMPTY PACKET>"""
359         return lines + '\n'
360
361     def validate_rip_packet(self, packet):
362         """Raise an AssertionError if the packet is invalid.
363         Does not check the validity of the contained rip entries.
364         """
365         assert len(packet) >= 4+20, f"packet length invalid: {len(packet)}"
366         assert len(packet) <= 4+20*25, f"packet length invalid: {len(packet)}"
367         assert (len(packet) - 4) % 20 == 0, f"packet length invalid: {len(packet)}"
368         assert packet[0] == 2, "command field not 2"
369         assert packet[1] == 2, "version field not 2"
370         routerid = int.from_bytes(packet[2:4])
371         assert routerid_is_valid(routerid), f"router-id invalid {routerid}"
372
373
374 class RipEntry:
375     """This class represents a validated RIP entry from a RIP packet.
376     2 bytes - address family (ignore)
377     2 bytes - all zeros
378     4 bytes - routerid (IPv4 in RIP RFC)
379     8 bytes - all zeros
380     4 bytes - metric
381     """
382     def __init__(self, entry):
383         self.validate_rip_entry(entry)
384         self.routerid = int.from_bytes(entry[4:8])
385         self.metric = int.from_bytes(entry[16:20])
386
387     def __str__(self):
388         return f'router-id: {self.routerid} metric: {self.metric}'
389
390     def validate_rip_entry(self, entry):
391         """Raise an AssertionError if the rip entry is invalid."""
392         assert len(entry) == 20, "RIP entry length not 20"
393         assert int.from_bytes(entry[0:2]) == 2, "address family must be 2"
394         assert int.from_bytes(entry[2:4]) == 0, "field must be all zeros"
395         routerid = int.from_bytes(entry[4:8])
396         assert routerid_is_valid(routerid), f"router-id invalid {routerid}"
397         assert int.from_bytes(entry[8:16]) == 0, "field must be all zeros"
398         metric = int.from_bytes(entry[16:20])
```

```
399             assert metric_is_valid(metric), f"metric invalid {metric}"
400
```