```python
"""
Assignment 1: RIP protocol
Team: Bach Vu (25082165), Charlie Hunter (27380476)
Router main program/Daemon.py
"""
########## Header ##########
from daemon_sup import *
import socket, time, select
import sys, random # must use
import traceback # optional features
from router import Router, RTimer

LocalHost = "127.0.0.1"
ROUTER = None # Router Obj
SOCKETS = {} # Enabled Interfaces

########## Body ##########
def createSocket():
    for port in ROUTER.INPUT_PORTS:
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind((LocalHost, port))
        SOCKETS[port] = sock

def send(mode):
    """ Send forwarding table to neighbour routers """
    for destID, link in ROUTER.OUTPUT_PORTS.items():
        table = ROUTER.get_routing_table(destID, mode)
        if len(table) == 0:
            # entry may got updated while delay & prepare package
            continue
        message = create_rip_packet(table)
        dest = (LocalHost, link[0])
        SOCKETS[link[2]].sendto(message, dest)

    print(f"Routing Table ({mode}) sent to neighbours at {strCurrTime()}.\n")
    if mode == ROUTER.REGULAR_UPDATE:
        ROUTER.reset_timer(RTimer.PERIODIC_TIMEOUT)


def send_periodic():
    mode = "None"
    if ROUTER.is_expired(RTimer.PERIODIC_TIMEOUT, getTime()):
        # Regular update
        mode = ROUTER.REGULAR_UPDATE
    elif ROUTER.has_expired_entry(getTime()):
        # Triggered update. Known NoResponse links don't trigger this
        mode = ROUTER.EXPIRED_UPDATE
    else:
        return

    send(mode)

def receive(timeout = 0.025):
    """ Return True if some data received """
    readable, _, _ = select.select(SOCKETS.values(), [], [], timeout)
    for sock in readable:
        receiver = sock.getsockname()
        data, sender = sock.recvfrom(1024)
```

```
59          if not ROUTER.is_expected_sender(sender, receiver):
60              print(f"Droped message on {sender} → {receiver} link!")
61              continue
62          routes = process_rip_packet(data)
63          print(f"Received ROUTES {str(routes)} at {strCurrTime(getTime())} from {sender}"
    )
64          triggered_update = ROUTER.update_route_table(routes, getTime())
65          if triggered_update:
66              # Next time, the record become Reset Timer
67              send(Router.FAST_ROUTE_UPDATE)
68
69  def garbage_collection():
70      ROUTER.garbage_collection(getTime())
71
72  ########## Program ##########
73  def init_router():
74      global ROUTER # include this if modifying global variable
75      filename = sys.argv[1]
76      rID, inputs, outputs, timeout = read_config(filename)
77
78      #  Router instance with default routing table
79      ROUTER = Router(rID, inputs, outputs, getTime(True), timeout)
80      ROUTER.print_hello()
81
82      # First time notice to neighbours
83      createSocket()
84
85  if __name__ == "__main__":
86      try:
87          init_router()
88          while True:
89              ROUTER.print_route_table(getTime())
90              send_periodic()
91              garbage_collection()
92              receive()
93
94      except IndexError:
95          print("Error: Config file is not provided!")
96      except FileNotFoundError:
97          print("Error: given Config file not found!")
98      except ValueError as v_err:
99          print("Warning:", v_err)
100     except socket.error as s_err:
101         print("Error:", s_err)
102     except KeyboardInterrupt:
103         print("\n******** Daemon exit successfully! Router shuting down... ********")
104     except Exception as e:
105         traceback.print_exc() # Traceback unknown error
106         print("Program exited unexpectedly.\n")
107     finally:
108         print()
109         sys.exit()
```

```python
"""
Assignment 1: RIP protocol
Team: Bach Vu (25082165), Charlie Hunter (27380476)
Router support function/daemon_sup.py
"""
import os, sys
import numpy as np
from datetime import datetime

FILE_EXTENSION = ".txt"

def read_config(filename):
    rID, inputs, outputs, timeout = None, None, None, None
    if filename.endswith(FILE_EXTENSION):
        config_file = open(filename)
    else:
        config_file = open(filename + FILE_EXTENSION)
    config_data = config_file.readlines()

    for line in config_data:
        head, data = line.split(':')
        if head == "router-id":
            rID = int(data)
            if not 0 < rID or rID > 64000:
                raise ValueError("Router ID must be between 1 and 64000.")
        elif head == "input-ports":
            inputs = [int(port) for port in data.rstrip().split(',')]
            if not is_valid_ports(inputs):
                raise ValueError("Invalid input port(s) in config data.\nPorts must be between 1024 and 64000.")
        elif head == "outputs":
            outputs = [port.strip() for port in data.rstrip().split(',')]
            ports = [int(output.split('-')[1]) for output in outputs]
            if not is_valid_ports(ports):
                raise ValueError("Invalid output port(s) in config data.\nPorts must be between 1024 and 64000.")
        elif head == "timer":
            timeout = int(data)
            if not 0 < timeout or timeout > 30:
                raise ValueError("Timeout must be between 1 and 30.")
    return rID, inputs, outputs, timeout

def is_valid_ports(ports):
    ports = np.array(ports)
    return np.all((ports ⩾ 1024) & (ports ⩽ 64000))

def create_rip_packet(table):
    header = create_rip_head()
    body = bytearray()
    for entry in table:
        new_entry = create_rip_entry(entry)
        body += new_entry
    return header + body

def create_rip_head(TTL=0):
    "Creates the 4 byte header"
    command = 1
    verison = 2
```

```
    command = command.to_bytes(1, byteorder='big')
    verison = verison.to_bytes(1, byteorder='big')
    reserve = (TTL+1).to_bytes(2, byteorder='big')
    return command + verison + reserve

def create_rip_entry(entry):
    "Creates the 20 byte body of packet"
    address_fam, zero = 0, 0
    afi         = address_fam.to_bytes(2, byteorder='big')
    route_tag   =        zero.to_bytes(2, byteorder='big')
    dest        =    entry[0].to_bytes(4, byteorder='big') # routerID
    subnet      =        zero.to_bytes(4, byteorder='big')
    next_hop    =    entry[1].to_bytes(4, byteorder='big')
    metric      =    entry[2].to_bytes(4, byteorder='big')
    return afi + route_tag + dest + subnet + next_hop + metric

def process_rip_packet(packet):
    command = int.from_bytes(packet[0:1], byteorder='big')
    version = int.from_bytes(packet[1:2], byteorder='big')
    if command =≠ 1 or version =≠ 2:
        return []

    routes = []
    entry_count = (len(packet)-4)//20
    for i in range(entry_count):
        si = i*20 + 4 # entry_start_index
        dest_id  = int.from_bytes(packet[si+4:si+8], byteorder='big')
        next_hop = int.from_bytes(packet[si+12:si+16], byteorder='big')
        metric   = int.from_bytes(packet[si+16:si+20], byteorder='big')
        routes.append((dest_id, next_hop, metric))

    return routes


def strCurrTime(time=None):
    if time is None:
        return datetime.now().strftime('%H:%M:%S')
    else:
        return time.strftime('%H:%M:%S')

def getTime(as_float=False):
    """ Get current time as float or object """
    if as_float:
        return datetime.now().timestamp()
    else:
        return datetime.now()
```

```python
"""
Assignment 1: RIP protocol
Team: Bach Vu (25082165), Charlie Hunter (27380476)
Router main program/router.py
"""
from timer import RTimer
from daemon_sup import strCurrTime

class Router:
    EXPIRED_UPDATE = "expired"
    REGULAR_UPDATE = "periodic"
    FAST_ROUTE_UPDATE = "Poison enhance"
    def __init__(self, rID, inputs, outputs, startTime, timeout):
        _timeout = timeout if timeout is not None else 5
        self.timer = RTimer(_timeout)
        self._garbages = {} # (dest, time since expired)

        self.ROUTER_ID = rID
        self.INPUT_PORTS = inputs

        self._ROUTING_TABLE = {}  # {Dest: nxt Hop, metric, time, note}
        self._ROUTING_TABLE[rID] = ["-", 0, startTime, "Time Active"]

        self.OUTPUT_PORTS = {} # (dest, cost, port_to_send)
        for output in outputs:
            from_port, to_port, cost, dest = output.split('-')
            from_port, to_port, cost, dest = int(from_port), int(to_port), int(cost), int
(dest)
            self.OUTPUT_PORTS[dest] = (to_port, cost, from_port)

    def get_routing_table(self, dest, mode):
        entries = []
        for key, val in self._ROUTING_TABLE.items():
            if dest == val[0] or dest == key:
                # don't re-advertise info from a hop (Split horizon)
                # dest == key not needed, but can reduce packet size
                continue
            if mode == "expired" and val[1] != 16:
                # triggered update, contain expired entries only (pg29)
                continue
            if mode == "Poison enhance" and val[3] != "Shorter route":
                continue

            new_metric = val[1] + self.OUTPUT_PORTS[dest][1]
            if new_metric > 15 and val[1] != 16:
                continue
            new_metric = min(new_metric, 16)
            entries.append((key, self.ROUTER_ID, new_metric))
        return entries

    def update_route_table(self, routes, utime):
        update_flag = False
        for route in routes:
            dest, nxtHop, metric = route
            new_entry = [nxtHop, metric, utime.timestamp(), ""]
            exist_entry = self._ROUTING_TABLE.get(dest, None)

            if not self._need_update(new_entry, exist_entry):
```

```
58                 continue
59
60             self._ROUTING_TABLE[dest] = new_entry
61             if new_entry[3] == "Shorter route":
62                 # trigger update with small delay. Not needed for small delay network.
63                 update_flag = True
64
65             # updated dest entry could be in garbage collecting
66             self._garbages.pop(dest, None)
67         return update_flag
68
69     def _need_update(self, new_entry, exist_entry):
70         """ For fancy purpose of taking note when update an entry
71             return True if new entry is valid to be updated
72         """
73         if exist_entry is None:
74             if new_entry[1] == 16:
75                 # Don't worry about dead link of unknown dest
76                 return False
77             new_entry[3] = "New dest."
78         else:
79             if new_entry[1] < exist_entry[1]:
80                 new_entry[3] = "Shorter route"
81
82             elif new_entry[1] == 16:
83                 if exist_entry[1] == 16:
84                     # already receive this link dead
85                     return False
86                 elif exist_entry[0] != new_entry[0]:
87                     # link dead is not currently in route table
88                     return False
89                 # 1st time known dest (metric < 16) has dead link
90                 new_entry[3] = "Link dead."
91
92             elif new_entry[1] == exist_entry[1]:
93                 new_entry[3] = "Reset timer"
94                 if new_entry[0] != exist_entry[0]:
95                     # New route, reset timer still
96                     new_entry[3] = "Same cost"
97
98             else:
99                 # ["Slower route."], not update
100                return False
101
102        return True
103
104    def garbage_collection(self, gtime):
105        if not self.timer.is_expired(RTimer.GARBAGES_TIMEOUT, gtime):
106            return False
107
108        for item, time in self._garbages.copy().items():
109            if self.timer.is_expired(RTimer.GARBAGE_TIMEOUT, gtime, time):
110                self._ROUTING_TABLE.pop(item, None)
111                self._garbages.pop(item)
112                print(f"Removed dead link to {item} at {strCurrTime(gtime)}")
113        self.timer.reset_timer(RTimer.GARBAGES_TIMEOUT)
114
115    def has_expired_entry(self, etime):
```

```
116         if not self.timer.is_expired(RTimer.ENTRIES_TIMEOUT, etime):
117             """ Trigger once if multilink die in short period """
118             return False
119 
120         garbage_found = 0
121         for dest,entry in self._ROUTING_TABLE.items():
122             if dest == self.ROUTER_ID:
123                 continue
124 
125             _, metric, ttl, _ = entry
126             if metric == 16:
127                 if dest in self._garbages.keys():
128                     # Waiting to be removed, skip to avoid sending same info to network
129                     continue
130                 self._garbages[dest] = etime.timestamp()
131                 garbage_found += 1
132 
133             elif self.timer.is_expired(RTimer.ENTRY_TIMEOUT, etime, ttl):
134                 entry[1], entry[3] = 16, "No response."
135                 self._ROUTING_TABLE[dest][1] = 16 # set to infinity
136                 self._garbages[dest] = etime.timestamp()
137                 print(f"Found expired link to {dest} at {strCurrTime(etime)}")
138                 garbage_found += 1
139 
140         self.timer.reset_timer(RTimer.ENTRIES_TIMEOUT)
141         # print(garbage_found)
142         return garbage_found > 0
143 
144     def is_expected_sender(self, sender, receiver):
145         """ Avoid unwanted broadcast/malicious pecket """
146         for link in self.OUTPUT_PORTS.values():
147             if sender[1] == link[0] and receiver[1] == link[2]:
148                 return True
149         return False
150 
151     def print_hello(self):
152         print("-"*66)
153         print(f"Router {self.ROUTER_ID} is running ...")
154         print("Input ports:", self.INPUT_PORTS)
155         print("Output ports:")
156         for dest, link in self.OUTPUT_PORTS.items():
157             print(f"    {link} to Router ID {dest}")
158         print("-"*66)
159         print("Use Ctrl+C or Del to shutdown.")
160         print()
161 
162     def print_route_table(self, ptime):
163         if not self.timer.is_expired(RTimer.PRINT_TIMEOUT, ptime):
164             return
165 
166         print("="*66)
167         print("|{:16}--{} [{}]--{:16}|".format(" ", "ROUTING TABLE", strCurrTime(ptime
    ), " "))
168         print("|{:^10}|{:^10}|{:^10}|{:^10}|{:^20}|".format(
169             "Dest.", "Next Hop", "Metric", "Time (s)", "Notes"))
170         print("|" + "-"*64 + "|")
171         for dest, record in self._ROUTING_TABLE.items():
172             hop, cost, log_time, note = record
```

```
            duration = ptime.timestamp() - log_time
            print("|{:^10}|{:^10}|{:^10}|{:^10.3f}|{:^20}|".format(
                dest, hop, cost, duration, str(note)))
        print("="*66)
        self.timer.reset_timer(RTimer.PRINT_TIMEOUT)

    def reset_timer(self, mode):
        self.timer.reset_timer(mode)

    def is_expired(self, mode, curr_time):
        return self.timer.is_expired(mode, curr_time)


```

```python
"""
Assignment 1: RIP protocol
Team: Bach Vu (25082165), Charlie Hunter (27380476)
Timer main program/timer.py
"""
import random, time

class RTimer:
    PRINT_TIMEOUT = 0
    PERIODIC_TIMEOUT = 1
    ENTRY_TIMEOUT = 2
    GARBAGE_TIMEOUT = 3
    ENTRIES_TIMEOUT = 4
    GARBAGES_TIMEOUT = 5
    def __init__(self, base):
        self._timeout = base
        self._time_logs = [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0]

    def get_print_timeout(self):
        """ How often to print routing table """
        return self._timeout * 1/2

    def get_periodic_timeout(self):
        """ From config, Ripv2 value 30 +- (0,5) """
        return self._timeout * (1-random.uniform(-1/5, 1/5))

    def get_entry_timeout(self):
        """ Expiry of a routing entry. Ripv2 value 180 """
        return self._timeout * 6

    def get_garbage_timeout(self):
        """ Delete expired entry delay. Ripv2 value 120 """
        return self._timeout * 4

    def get_entry_check_timeout(self):
        """ Router periodic check expired entries """
        return 1 #self._timeout / 5

    def get_garbage_check_timeout(self):
        """ Router periodic check expired garbage """
        return 1 # self._timeout / 5

    def reset_timer(self, mode):
        self._time_logs[mode] = time.time()

    def is_expired(self, mode, curr_time, ttl=None):
        """ Check time log """
        curr_time = curr_time.timestamp()
        if ttl is not None:
            self._time_logs[mode] = ttl
        if self._time_logs[mode] == -1:
            return True

        timeout_value = [self.get_print_timeout, self.get_periodic_timeout,
            self.get_entry_timeout, self.get_garbage_timeout,
            self.get_entry_check_timeout, self.get_garbage_check_timeout]
        time_elapsed = curr_time - self._time_logs[mode]
        return time_elapsed >= timeout_value[mode]()
```