COSC364 - RIP Assignment

Aljaz Smrekar (asm237 - 92496132)

Noah Ilogon (nil13 – 48063791)

Computer Science and Software Engineering Department

University of Canterbury

COSC364: Internet Technology and Engineering

Barry Wu

April 29, 2025

Contributions

Aljaž Smrekar (50%)

Contributed to 50% of the project, including:

- Implemented half of Reader.py, half of the configuration file logic, and most of Packet.py.
- Developed and set up the initial framework of RoutingTable.py.
- Contributed to RIPv2_router.py by implementing core functionality such as create_sockets().
- Additionally, created approximately 50% of the test cases.

Noah Ilogon (50%)

Contributed to 50% of the project, including:

- Implemented half of Reader.py, half of the configuration file logic, and was the main contributor to both RIPv2_router.py and RoutingTable.py.
- Was the one who mostly designed the overall architecture and interaction between modules.
- Also created approximately 50% of the test cases.

Testing

Test 1 – Basic Convergence Test

Goal – Check and see if the routing tables converge correctly when all the routers have started

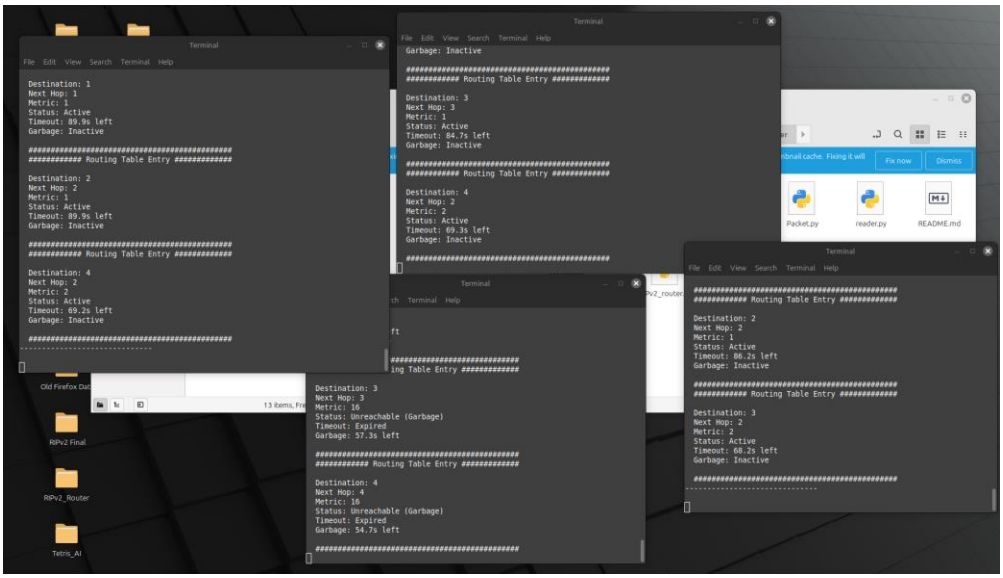
Setup – Launch all the routers which should create a small network. We will also wait a few minutes to see if they will run smoothly without any issues.

Excepted Outcome –

1. Each router should know about all other routers
2. Routing table should converge fully after a few updates
3. Each destination should have a minimum hop cost to each router
4. The network should run smoothly without any issues

Actual Outcome - In this test we waited about 5 minutes and just as expected there were no issues, and the routers did converge quickly once the network came online.

Conclusion - The basic convergence test helped us confirm that we implemented the distance vector algorithm correctly on the RIPv2 routers. All routers successfully exchanged routing information and updated their tables over time. The routing tables also fully converged to reflect the shortest paths to every other router in the network. In the test we also did not observe any packet loss, loops, or instability during the test period.



Test 2 – Header Check

Goal – We tested whether the router correctly detects and rejects a RIP packet with an invalid Version field. According to the RIP v2 specification, the Version field must be exactly 2. If it is not like in this case, then the router must drop the packet.

Setup – We manually send a RIP packet with a wrong version field (e.g., set version to 5 instead of 2) to one of the router's input ports.

Excepted Outcome –

1. The router should return an ERROR saying ("ERROR: Invalid Packet...\n"
"Packet must be either\n" "1 - Request Packet\n" "2 - Response Packet\n"
"Dropping Packet!...") and then return and ERROR saying that header check has failed.
In this test we tested if the header check works correctly by changing the version number to an invalid one.

Actual Outcome - When we sent a packet with Version 5, the router printed the error messages we expected. The routing table was also not updated while also not crashing or creating any unexcepted side effects.

Conclusion – This test shows that the header checking functionality of our RIPv2 is working correctly. This is because it properly validates critical fields like the header before accepting or updating any routes in the table. This in conclusion increases reliability and enhances security by rejecting invalid inputs early before they even get accepted into the system.

```
ERROR: Invalid Version!...
Version must be 2..
Dropping Packet!...
[ERROR] Packet Header Check Failed!
```

Test 3 – Duplicated Input Ports

Goal – Test if the router correctly detects and handles the case where there is a duplicated Input Port.

Setup – We manually create a router that had both Input-Ports = 5002 and

Output-Ports = 5001-1-1, 5003-3-3. Then we created a new router which had a different Router-ID but the same input ports. After they were created, we ran them together.

Excepted Outcome –

1. The router should realise that there is an input port already created and should terminate the router trying to duplicate the port while leaving the other running.

Actual Outcome - When we ran the test it worked out and it gave an Error and told the router the address is already in use. However, it also printed the routing table once before the system was terminated.

Conclusion – This test helps confirm that duplicate port protection is functioning correctly, however, there is a small cosmetic issue. Even though the router terminates on a port error recognition every time, it prints a partial routing table beforehand, which could lead to people getting confused while debugging or simply running the program.

```
Error creating sockts [Errno 98] Address already in use
Seeding directly-connected neighbours into routing table:
  • dst=1, next_hop=1, metric=1
  • dst=3, next_hop=3, metric=3
```

```
  Destination: 3
  Next Hop: 3
  Metric: 3
  Status: Active
  Timeout: 90.0s left
  Garbage: Inactive

  ########################################
[INFO] Refreshing direct routes...
Traceback (most recent call last):
  File "/csse/users/asm237/Desktop/RIPv2 Final/RIPv2_Router/reader.py", line 209, in <module>
    router_instance = RIPv2_Router(ROUTER_ID, ROUTER_INPUTS, ROUTER_OUTPUTS)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/csse/users/asm237/Desktop/RIPv2 Final/RIPv2_Router/RIPv2_router.py", line 101, in __init__
    self.init_periodic_update()
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/csse/users/asm237/Desktop/RIPv2 Final/RIPv2_Router/RIPv2_router.py", line 186, in init_periodic_update
    send_update()
  File "/csse/users/asm237/Desktop/RIPv2 Final/RIPv2_Router/RIPv2_router.py", line 176, in send_update
    self.update_neighbours() # This sends updates based on the *current* table state
    ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/csse/users/asm237/Desktop/RIPv2 Final/RIPv2_Router/RIPv2_router.py", line 223, in update_neighbours
    send_sock = self.sockets[self.inputs[0]]
                ~~~~~~~~~~~~^^^^^^^^^^^^^^^^
TypeError: 'NoneType' object is not subscriptable
[asm237@cs25224cy ~/Desktop/RIPv2 Final/RIPv2_Router]$
```

In this test we tested...

Test 4 – Invalid and Missing Ports

Goal – Test if the router correctly detects and handles the cases where firstly there is an Invalid Port, then if a Port is missing.

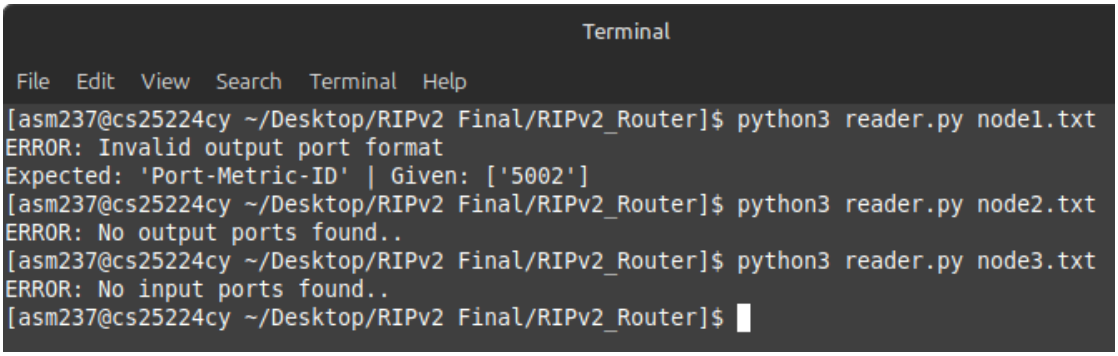
Setup – We manually create a config file that has an Invalid Port, then a Missing Port. We then run these files and see what happens.

Excepted Outcome –

2. The router should instantly recognise the issue and print and ERROR message before terminating the process right after.

Actual Outcome - When we ran the daemon with a config that had missing, or invalid ports, it printed the correct ERROR message and exited cleanly without starting the router.

Conclusion – This shows that the router correctly validates that valid and required Input-Ports and Output-Ports fields exists before proceeding further with the process. This prevents invalid router setups and ensures that necessary socket bindings will not fail further down the line.



Test 5 – Wrong File Name

Goal – We tested whether the config reader correctly detects and rejects an invalid or missing config file. If the config file is not found or is improperly formatted, the program must immediately stop execution and display an error.

Setup – We manually attempted to start the router daemon by providing an invalid config file name (e.g., a file name that does not exist or has a wrong extension) as a command-line argument.

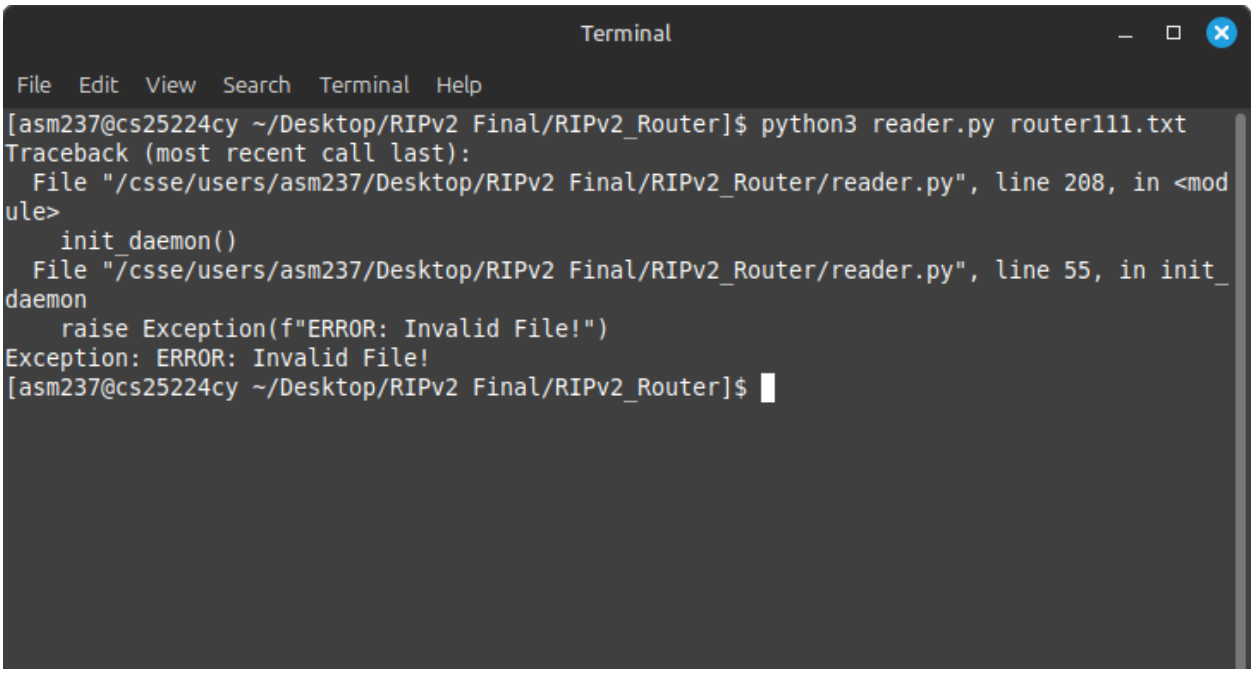
Excepted Outcome –

1. The program recognise and display an ERROR message saying, “Invalid File”.

2. The program should then terminate on the spot and no network, path, or router should be created.

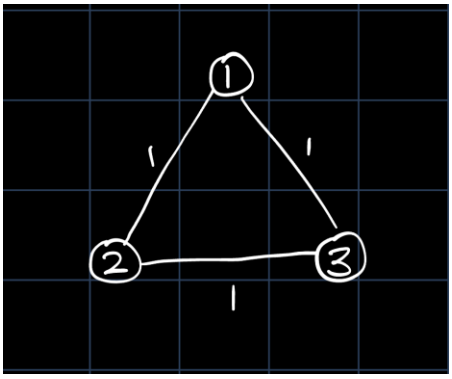
Actual Outcome - When we provided a wrong or invalid config file name, the program correctly printed the ERROR message and terminated the process on the spot. Nothing was also initialised and everything went according to plan.

Conclusion – This test helped us confirm that the config file validation functionality of our router daemon works correctly and is properly implemented. It showed that it properly validates critical startup parameters before proceeding, preventing misconfiguration which in turn helps enhance security by avoiding unintended behavior. This also enhances reliability due to ensuring that routers only start with valid and fully specified configurations.



Test 6 – Configuring a network and testing that the network connects and periodic updates and timeout is implemented.

A network was configured with this layout:



And with the nodes 1 – 4 corresponding to the configured routers 1 – 4. We will wait 5 minutes, and we will expect to see that the network is still connected and sending each other periodic updates, and all routers will have the correct metric and next hop for all other routers and the periodic updates are refreshed after each update received from ite neighbours.

Router 1's Routing table:

```
RoutingTable (Entries: 2):
  ############ Routing Table Entry #############

  Destination: 2
  Next Hop: 2
  Metric: 1
  Status: Active
  Timeout: 90.0s left
  Garbage: Inactive

  ############################################
  ############ Routing Table Entry #############

  Destination: 3
  Next Hop: 3
  Metric: 1
  Status: Active
  Timeout: 85.1s left
  Garbage: Inactive

  ############################################
```

Router 2's Routing Table:

```
--- Periodic Status Update ---
RoutingTable (Entries: 2):
  ############ Routing Table Entry #############

  Destination: 1
  Next Hop: 1
  Metric: 1
  Status: Active
  Timeout: 76.5s left
  Garbage: Inactive

  ############################################
  ############ Routing Table Entry #############

  Destination: 2
  Next Hop: 2
  Metric: 1
  Status: Active
  Timeout: 79.9s left
  Garbage: Inactive

  ############################################
----------------------------
```

Router 3's Routing Table:

```
RoutingTable (Entries: 2):
  ############ Routing Table Entry #############

  Destination: 1
  Next Hop: 1
  Metric: 16
  Status: Unreachable (Garbage)
  Timeout: Expired
  Garbage: Garbage not started

  ############################################
  ############ Routing Table Entry #############

  Destination: 2
  Next Hop: 2
  Metric: 1
  Status: Active
  Timeout: 90.0s left
  Garbage: Inactive

  ############################################
```

After the network has been running for 5 minutes amount of time we can see that all three routers are sending each other periodic updates and refreshing its timeout timer after each update is received.

Test 7 – Periodic Garbage and Pruning dead routes

From the network we have configured above, we will now take down router 2. We should be able to see that Router 1 and Router 3 starts the garbage timer for Router 2 after the 90 second

timeout timer. After the Garbage timer has run out, we should see Router 1 and Router 3 remove the route and entry for Router 2 but still have an active route for each other.

Router 1's Routing Table 90 seconds after router 2 goes offline:

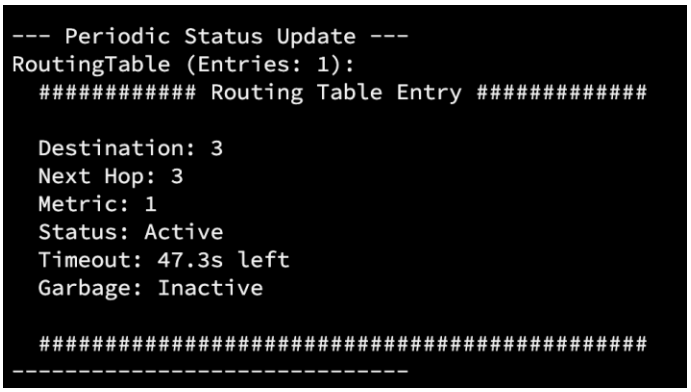
```
 ############ Routing Table Entry #############

 Destination: 2
 Next Hop: 2
 Metric: 16
 Status: Unreachable (Garbage)
 Timeout: Expired
 Garbage: 59.7s left

 ############################################
 ############ Routing Table Entry #############

 Destination: 3
 Next Hop: 3
 Metric: 1
 Status: Active
 Timeout: 77.5s left
 Garbage: Inactive

 ############################################
-----------------------------
```

Router 3's Routing Table 90 seconds after router 2 goes offline:

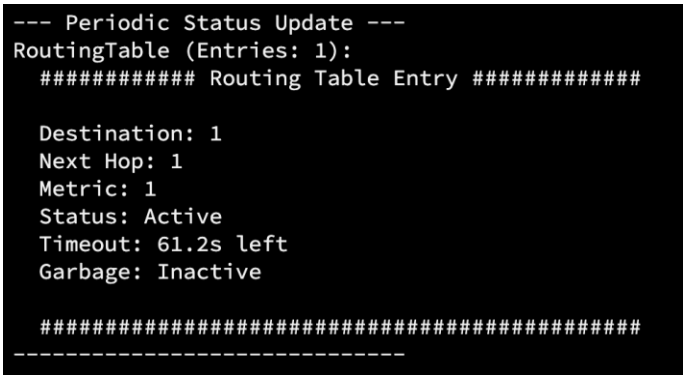
```
--- Periodic Status Update ---
RoutingTable (Entries: 2):
  ############ Routing Table Entry #############

  Destination: 1
  Next Hop: 1
  Metric: 1
  Status: Active
  Timeout: 80.0s left
  Garbage: Inactive

  ############################################
  ############ Routing Table Entry #############

  Destination: 2
  Next Hop: 2
  Metric: 16
  Status: Unreachable (Garbage)
  Timeout: Expired
  Garbage: 54.7s left

  ############################################
-----------------------------
```

We can see that the garbage timer is activated after the timeout expires for Router 2 in Router 1 and 3's routing table. After the garbage timer expires, we should now see that Both Router 1 and Router 3 will prune router 2 from its routing table.

Router 1's Routing table after the garbage timer has run out on Router 2:



Router 3's Routing table after the garbage timer has run out on Router 2:



We can now see that both Routers 1 and 3 now only have 1 entry in its routing table and that Router 2 has now been pruned and removed from the routing table meaning that the network has learnt the Router 2 is no longer a viable path and will no longer be used.

Test 8 – Duplicate Router IDs and Valid Router ID Check

Goal – Test if the router correctly detects and handles cases where two or more routers attempt to use the same Router ID, which must be unique across all routers. We will also test whether the Router ID provided is valid an integer between 1 and 64000 inclusive.

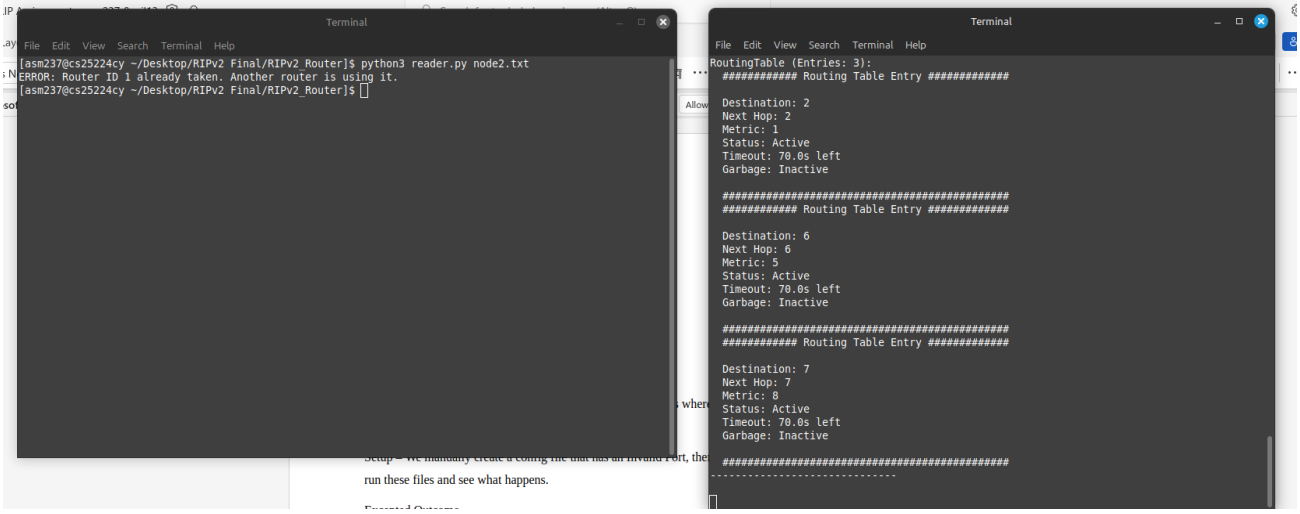
Setup – We manually create a router that has a Router ID of 1 and another router with the same Router ID. After we ran the routers at the same time and watched what happened. We will then also create a router with the Router ID 111111111, and run it.

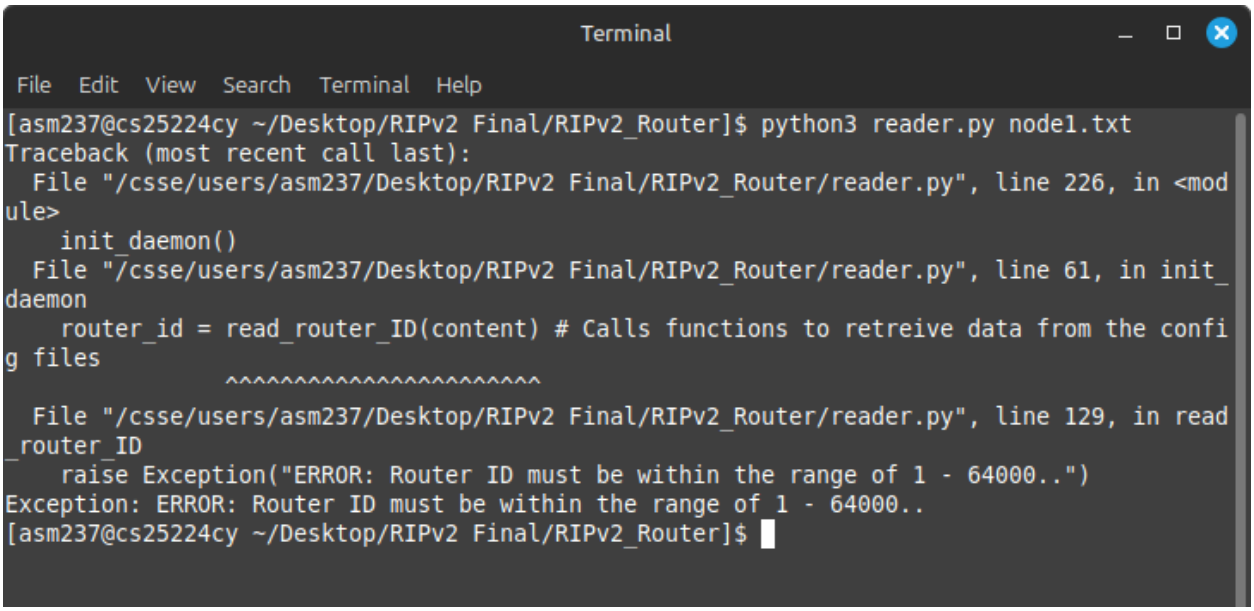
Excepted Outcome – The first router should successfully initialise. When we run the second router, however, the router should recognise that the Router ID is already in use and should print

an ERROR message saying that the Router ID is already in use. When we run the router with a very high ID we expect the router to realise and print an ERROR message and raising an Exception("ERROR: Router ID must be within the range of 1 - 64000..").

Actual Outcome - First router worked perfectly and as soon as we started the second router it returned and ERROR and terminated the process just as we expected. When we tested for an invalid ID number the test also worked perfectly, however, it did print an ERROR message twice which is only a small cosmetic issue.

Conclusion – This test successfully demonstrates that the routers implementation validates the Router ID during startup. It ensures that any Router ID used is both within the valid range and is globally unique across all running routers. If a router attempts to use an invalid ID or a duplicate ID already in use, the program immediately prints a clear error message and terminates cleanly without proceeding. This validation and failsafe at the beginning prevents potential socket binding failures, routing errors, and network instability in the future. Overall, the system's design ensures that only properly configured and uniquely identified routers are allowed to join the network, leading to more reliable and predictable router behavior just as the RIPv2 protocol states.





Test 9 – Empty Routing Table Response Packet

Goal – The goal was to test if the router correctly creates a valid RIPv2 response packet when its routing table is empty. RIPv2 protocol rules state that a router must still respond to requests or periodic updates even if it has no routes, by sending a packet containing only the RIP header and no entries.

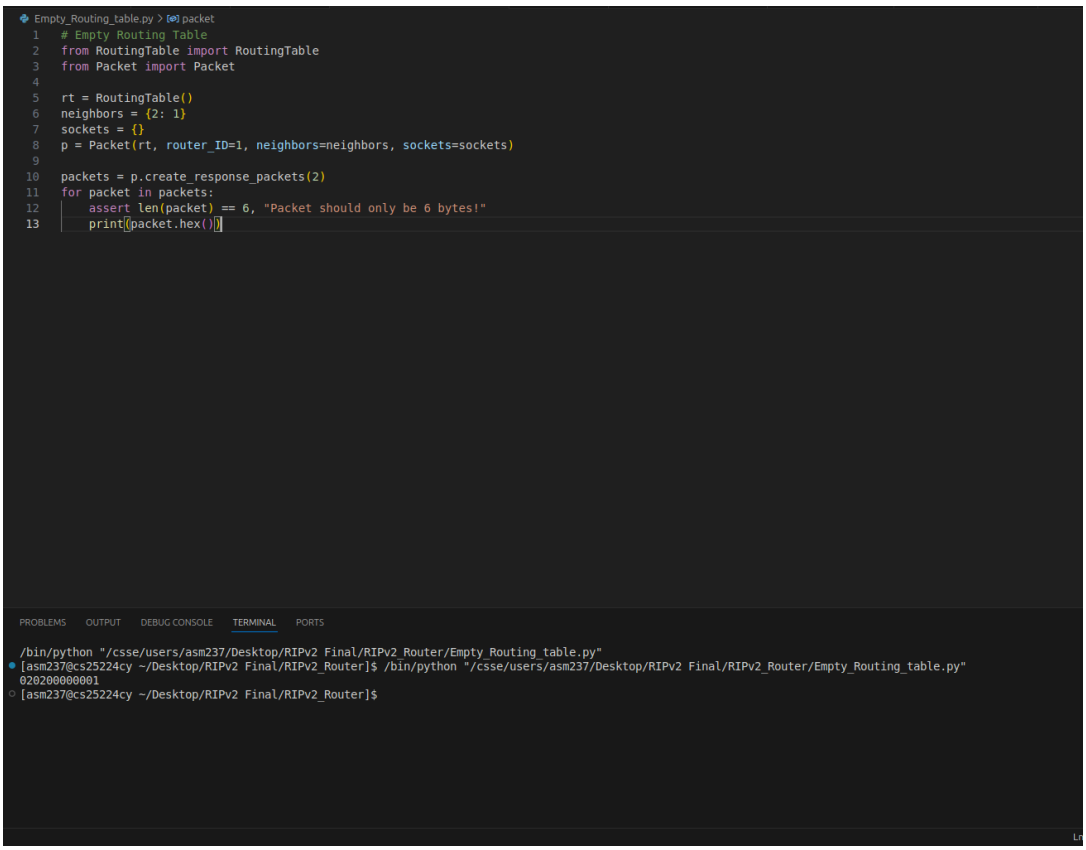
Setup – We manually created an empty RoutingTable object. Then, we instantiated a Packet object using this empty routing table, a Router ID of 1, and a dummy neighbor with ID 2.
We called the create_response_packets() method with the neighbor ID 2 and examined the packet output. We also enforced a test to ensure the packet length was exactly 6 bytes which is the header only, and that the content matched the correct RIPv2 response header format.

Excepted Outcome – The function should return a list with exactly one packet if done correctly and should also contain only 6 bytes corresponding to a valid RIPv2 header

Command = 2 (Response), Version = 2, Reserved = 0, Router ID = 1

Actual Outcome - The router correctly created a packet as we expected which contained only the 6 byte RIPv2 header. The output, which was - 02 02 00 00 00 01 - matched the expected output.

Conclusion – This test successfully demonstrates that the router running our RIPv2 protocol handles the empty routing table case correctly by still generating a valid RIP response packet that consists only of the header as stated in the RFC. This behavior is needed to ensure neighboring routers do not misinterpret a lack of routing information as a communication failure if needed. In conclusion, our Packet.py class implementation based on the RIPv2 protocol RFC handles this special case as expected in accordance with RIPv2.



Test 10 – Empty Routing Table Response Packet

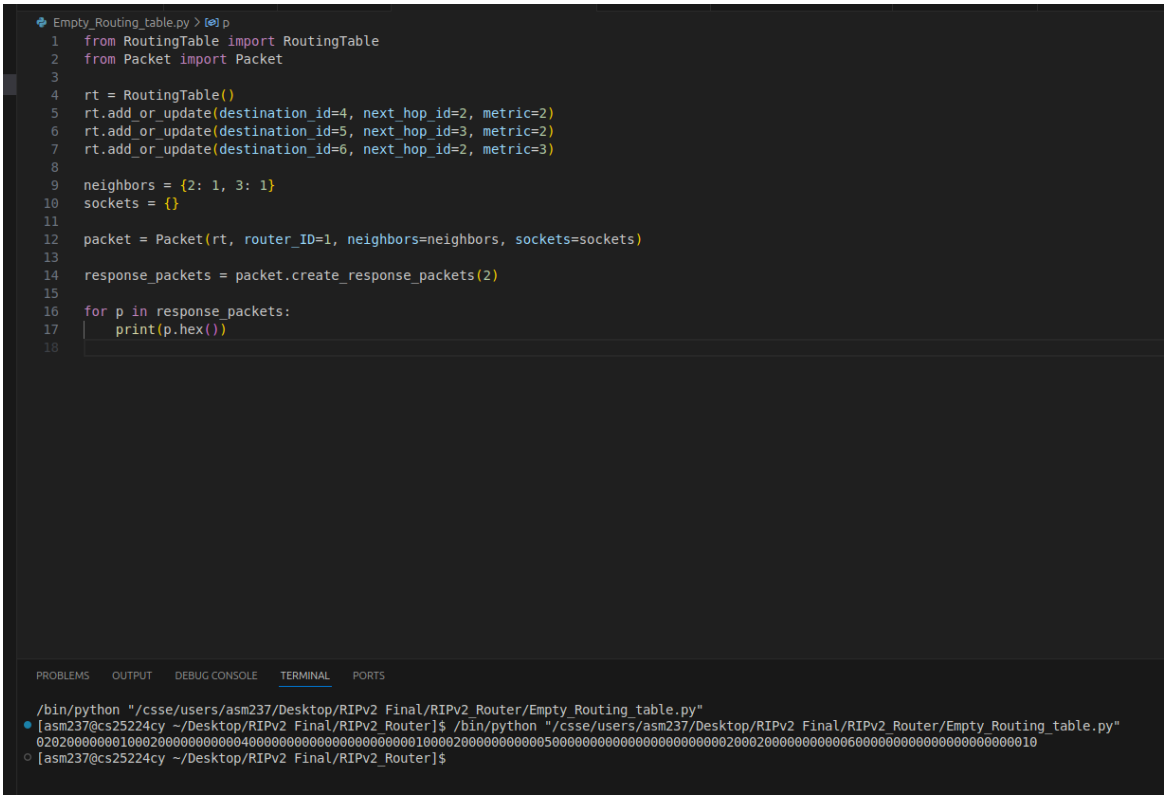
Goal – The goal of this test was to verify that the router correctly implements our Split Horizon with Poison Reverse implementation and ensure that we did not mess it up, when generating response packets for neighbors.

Setup – We manually created a RoutingTable with three different entries whiel setting up 2 neighbors. We then created a Packet object with Router ID = 1 and called the create_response_packets() method which targeted the neighbor 2. We then printed the packet output in hexadecimal to eb able to see the route advertisement behaviors and validate.

Excepted Outcome – The excepted outcome for this test was that routes learned via neighbor 2 in this case destinations 4 and 6 should be advertised to neighbor 2 with a metric of 16. And routes learned via other neighbors should be advertised normally.

Actual Outcome – The output packet that we got was – 02020000000100020000000000040000000000000000000000010000200000000000500000000000000000000000020002000000000060000000000000000000000010 – Which once decoded matches the expected outcome.

Conclusion – The conclusion of this was that the test successfully confirmed that the router correctly applies our implementation of the RIPs Split Horizon with Poison Reverse when constructing RIPv2 response packets. Routes that were learned through a neighbor are poisoned which changes the metric to 16 when sending updates to that neighbor, meaning that it works exactly as the RIPv2 specification specifies. This behavior helps prevent routing loops and count to infinity problems.



Config File Example I.e. router1.txt:

[ConfigFile]

Router-ID = 1

Input-Ports = 5001

Output-Ports = 5002-1-2, 5006-5-6, 5007-8-7

SOURCE CODE

Reader.py Code:

Config File Reader Reader.py:

''' The config file reader will:

- take a txt file and check it for validity
- initialise the router daemon.

{ 'Config-File': { 'Router-ID': '0', 'Input-Ports': '6110, 6201, 7345', 'Output-Ports': '5000-1-1, 5002-5-4', 'Periodic-Time': '', 'Timeout': '' } }

'''

import sys import os import configparser from RIPv2_router import * from collections import Counter

TIME_DEFAULT = 0

LOCAL_HOST = '127.0.0.1'

ROUTER_ID = None

ROUTER_INPUTS = []

ROUTER_OUTPUTS = []

ROUTER_ID_CHECK_PORT_BASE = 7777

def init_daemon(): ''' Initialises the Router Daemon '''

```
if len(sys.argv) < 2:  # Check if a file argument is provided (1st Param)
    raise Exception("ERROR: Please provide config file :)")
    sys.exit()


file = os.path.join("config-files", sys.argv[1])


if not os.path.isfile(file):  # Checks if the file exists if not raise ERROR
    raise Exception(f"ERROR: Invalid File!")
    sys.exit()


content = read_config_file(file) # Returns the content of the config file
router_id = read_router_ID(content) # Calls functions to retreive data from the config files
read_output_ports(content) # Retrieves a list of output ports in the format: "[PORT-METRIC-ID]"
read_input_ports(content)


print("\n######################\n")
print(f">> Router ID: {ROUTER_ID}\n")


print(f">> Output ports: {ROUTER_OUTPUTS}\n")


print(f">> Input ports: {ROUTER_INPUTS}")


print("\nConfiguration File Accepted ")
print("\n######################\n")
```

```
def read_config_file(config_file): ''' This function will read the config file passed in the terminal and will try to parse
it with the config parser '''

try:
    content = configparser.ConfigParser()
    content.read(config_file)


except configparser.ParsingError:
    print("[ERROR] Invalid File\n")
    sys.exit()


return content




def check_router_id_taken(router_id):
'''
Tries to bind a unique socket based on the Router ID.
If bind fails, it means Router ID is already taken.
'''
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
sock.bind((LOCAL_HOST, ROUTER_ID_CHECK_PORT_BASE + router_id))
except OSError:
sys.exit(f"ERROR: Router ID {router_id} already taken. Another router is using it.")
return sock # Keep the socket open to reserve the ID


def read_router_ID(config_data): ''' retreives router ID value from config file and assesses the validity then
returns the router ID '''

try:
    router_id = int(config_data.get("ConfigFile", "Router-ID")) # Finds value under Router-ID field
```

```
except configparser.ParsingError as e:
    print(f"[ERROR] parsing the config file: {e}")
    sys.exit()

except KeyError:
    raise Exception("ERROR: No Router ID found in the config.")

except ValueError:
    raise Exception("ERROR: Router ID is not a valid integer.")

if not (1 <= router_id <= 64000):
    raise Exception("ERROR: Router ID must be within the range of 1 - 64000..")

global ROUTER_ID
ROUTER_ID = router_id
return router_id

def read_input_ports(config_file): ''' retreives the list of input ports in the config file and assess validity and returns
the router input(s) '''

try:
    router_input = config_file['ConfigFile']['Input-Ports'].split(', ') # list of ports

    if router_input == [""]: # Checks if there is no given router input
        print("ERROR: No input ports found..")
        sys.exit()
```

```
    for port in router_input: # Loop through each port, strip whitespaces and convert the ports into an integer
        port = int(port.strip())
        ROUTER_INPUTS.append(port)
        if 1024 < port > 64000: # Checks if the ports are valid
            raise ValueError


    check_ports = set(ROUTER_INPUTS) # Check if there are any duplicates
    if len(check_ports) != len(ROUTER_INPUTS):
        raise Exception("ERROR: Duplciate input ports given | Cannot have a duplicated input port..")


    return ROUTER_INPUTS


except ValueError:
    print("ERROR: Each input port must be a positive integer between 1024-64000 inclusively")
    sys.exit()


except KeyError:
    print('ERROR: "Input-Ports" missing in the Config File.')
    sys.exit()


def read_output_ports(config_file): ''' retreives the list of output ports in the config file
                and assess validity In the format: {'port': 5000, 'metric': 1, 'router_id': 2},
            '''


            duplicate_port_check = []
```

```
try:
    router_outputs = config_file['ConfigFile']['Output-Ports'].split(', ') # list of ports
    if router_outputs == [""]:
        print("ERROR: No output ports found..")
        sys.exit()



    for port in router_outputs: # Loop through each port, strip whitespaces and convert the ports into an
integer
        split = port.split('-')
        if len(split) != 3:
            print("ERROR: Invalid output port format"
            f"\nExpected: 'Port-Metric-ID' | Given: {split}")
            sys.exit()


        duplicate_port_check.append(int(split[0]))


        ROUTER_OUTPUTS.append(split)


    for port in  duplicate_port_check:
        if 1024 < port > 64000: # Checks if the ports are valid
            raise ValueError


    for key, value in Counter(duplicate_port_check).items(): # Check if there are any duplicates
        if value >= 2:
            raise Exception("ERROR: Duplciate output ports found | Cannot have a duplicated output port..")
```

```
            return ROUTER_OUTPUTS


        except ValueError:
            print("ERROR: Each output port must be a positive integer between 1024-64000 inclusively")
            sys.exit()


        except KeyError:
            print('ERROR: "Output-Ports" missing in the Config File.')
            sys.exit()


if name == "main":

        init_daemon()

        id_check_socket = check_router_id_taken(ROUTER_ID)


        router_instance = RIPv2_Router(ROUTER_ID, ROUTER_INPUTS, ROUTER_OUTPUTS)
        router_instance.monitor_RT() # Start listening for updates (INF loop)
```

Packet creator packet.py:

import socket import sys import time from RoutingTable import RoutingTable

LOCAL_HOST = "127.0.0.1" PORT = 520 HDR_SIZE = 6 RT_SIZE = 20 # Routing table size

```
CMD_REQUEST = 1 CMD_RESPONSE = 2

INF = 16

class Packet:

def __init__(self, table, router_ID, neighbors: dict, sockets:dict): # outputs instead of neighbours
self.routing_table = table
    self.router_ID = router_ID
    self.sockets = sockets
    self.neighbours = neighbors



def create_response_packets(self, neighbour_id: int) -> list[bytes]: # Creates an Update packet
    """
    Create the update packet
    """

    packets = []
    entries = list(self.routing_table)  # instance of RT

    if not entries:
        header = bytearray(6)
        header[0] = CMD_RESPONSE
        header[1] = 2
        header[2:4] = (0, 0)
```

```
    header[4] = (self.router_ID >> 8) & 0xFF
    header[5] = self.router_ID & 0xFF

    return [bytes(header)]



for i in range(0, len(entries), 25):
    # Remove entries destined for the neighbor
    entry_chunk = [
        entry for entry in entries[i:i + 25]
        if entry.destination_id != neighbour_id
    ]
    if not entry_chunk:
        continue  # Skip empty chunks

    packet = bytearray(6 + 20 * len(entry_chunk))

    # Header
    packet[0] = CMD_RESPONSE
    packet[1] = 2
    packet[2:4] = (0, 0)
    packet[4] = (self.router_ID >> 8) & 0xFF
    packet[5] = self.router_ID & 0xFF
```

```
        cur_index = 6
        for entry in entry_chunk:
            # AFI + Route Tag
            packet[cur_index] = 0
            packet[cur_index + 1] = 2
            packet[cur_index + 2] = 0
            packet[cur_index + 3] = 0
            cur_index += 4

            # Destination ID (4 bytes)
            packet[cur_index] = (entry.destination_id >> 24) & 0xFF
            packet[cur_index + 1] = (entry.destination_id >> 16) & 0xFF
            packet[cur_index + 2] = (entry.destination_id >> 8) & 0xFF
            packet[cur_index + 3] = entry.destination_id & 0xFF
            cur_index += 4

            # Subnet Mask (8 bytes = 0s)
            for j in range(8):
                packet[cur_index + j] = 0
            cur_index += 8

            # print(f"[DEBUG] Sending route to {entry.destination_id} via {entry.next_hop_id} to
neighbour {neighbour_id}")
```

```
            # Metric with poison reverse
            cost = INF if entry.next_hop_id == neighbour_id else entry.metric

            packet[cur_index] = (cost >> 24) & 0xFF
            packet[cur_index + 1] = (cost >> 16) & 0xFF
            packet[cur_index + 2] = (cost >> 8) & 0xFF
            packet[cur_index + 3] = cost & 0xFF
            cur_index += 4

        packets.append(bytes(packet))

    return packets


def check_header(self, packet: bytes):
    """
        Checks if the RIP header is correct. If it is correct, returns
        the ID of the router it recieved it from
    """
    command = int(packet[0])
    version = int(packet[1])
    reserved = (packet[2], packet[3])  # Reserved bytes must be zero
    router_ID = int.from_bytes(packet[4:6], byteorder='big')
```

```
    if command != 2:
        if command == 1:
            print("ERROR: Packet is a request packet!...")
        else:
            print("ERROR: Invalid Packet...\n"
                  "Packet must be either\n"
                  "1 - Request Packet\n"
                  "2 - Response Packet\n"
                  "Dropping Packet!...")
        return False


    if version != 2:
        print("ERROR: Invalid Version!...\n"
              "Version must be 2..\n"
              "Dropping Packet!...")
        return False


    if reserved != (0, 0):
        print(f"ERROR: Invalid Packet! Reserved bytes must be zero.\nCurrently Reserved =
{reserved}\n"
        "Dropping Packet!...")
        return False
    if not router_ID or (1 > router_ID > 64000):
        print(f"ERROR: Invalid router ID! {router_ID} is out of valid range | Must be between 1-
```

```
64000 inclusive..\n"
                "Dropping Packet!...")
        return False


    return router_ID



def check_entry(self, entry):
    """Checks the entry and insures that the entry and all of its components are valid
        - returns passed
    """


    passed = True



    if len(entry) != 20:
        print(f"[ERROR] Bad entry length: {len(entry)} != 20")
        return False


    family_identifier_first = int(entry[0]) # First Byte
    family_identifier_second = int(entry[1]) # Second Byte
    family_identifier = (family_identifier_first, family_identifier_second) # Family Identifier
    if family_identifier != (0, 2): # Checks Family Identifier
        print("\n[ERROR] Invalid Address Family Identifier!")
```

```
        passed = False
        return passed


    route_tag_first = int(entry[2]) # First Byte
    route_tag_second = int(entry[3]) # Second Byte
    route_tag = (route_tag_first, route_tag_second) # Route Tag
    if route_tag != (0, 0):
        print("\n[ERROR] Invalid Route Tag!") #  distinguish routes learned from other routing
protocols.


        passed = False
        return passed



    IPv4_addy_first = int(entry[4]) # First Byte
    IPv4_addy_second = int(entry[5]) # Second Byte
    IPv4_addy_third = int(entry[6]) # Third Byte
    IPv4_addy_forth = int(entry[7]) # Forth Byte
    IPv4_addy = (IPv4_addy_first, IPv4_addy_second, IPv4_addy_third, IPv4_addy_forth) #
IPv4 Address
    addy = True
    for byte in IPv4_addy:
        if not (0 <= byte <= 0x255):  # Each byte must be between 0 and 255
            print(f"[ERROR] Invalid byte in IPv4 Address: {byte}!")
```

```
            passed = False
            addy = False

    if not addy:
        print(f"[ERROR] Invalid IPv4 Address: {IPv4_addy}!")
        return passed



    subnet_mask_first = int(entry[8])  # First Byte of subnet mask
    subnet_mask_second = int(entry[9])  # Second Byte
    subnet_mask_third = int(entry[10])  # Third Byte
    subnet_mask_forth = int(entry[11])  # Fourth Byte
    subnet_mask = (subnet_mask_first, subnet_mask_second, subnet_mask_third,
subnet_mask_forth)  # Subnet Mask
    mask = True
    for byte in subnet_mask:
        if not (0 <= byte <= 255):  # Each byte must be between 0 and 255
            print(f"[ERROR] Invalid byte in Subnet Mask {byte}! It must be between 0-255
inclusive.")
            mask = False
            passed = False

    if not mask:
        print(f"[ERROR] Bad subnet mask bytes: {subnet_mask}")
```

```
        return passed



    next_hop_first = int(entry[12])  # First Byte of next hop
    next_hop_second = int(entry[13])  # Second Byte
    next_hop_third = int(entry[14])  # Third Byte
    next_hop_forth = int(entry[15])  # Fourth Byte
    next_hop = (next_hop_first, next_hop_second, next_hop_third, next_hop_forth)  # Next hop
address
    hop = True
    for byte in next_hop:
        if not (0 <= byte <= 255):  # Each byte must be between 0 and 255
            print(f"ERROR: Invalid byte in Next Hop: {byte}! It must be between 0-255 inclusive.")
            hop = False
            passed = False

    if not hop:
        print(f"[ERROR] Bad subnet mask bytes: {next_hop}")
        return passed

    metric_first = int(entry[16])  # First byte
    metric_second = int(entry[17])  # Second byte
    metric_third = int(entry[18])  # Third byte
```

```
    metric_fourth = int(entry[19])  # Fourth byte
    received_metric = (metric_first << 24) + (metric_second << 16) + (metric_third << 8) +
metric_fourth  # Combine the 4 bytes into a single metric

    if not (1 <= received_metric <= INF):
        print(f"[ERROR] Invalid metric: {received_metric} (must be between 1 - {INF})")
        passed = False
    return passed




def receive_and_process_packet(self, packet: bytes):
    print("Packet Received Succesfully! \n"
    "Checking for Validity!...")

    received_ID = self.check_header(packet)
    if not received_ID:
        print("[ERROR] Packet Header Check Failed!\n")
        return # Drop the packet if header is invalid

    link_cost_to_sender = self.neighbours.get(received_ID)
    if link_cost_to_sender is None:
        print(f"[ERROR] Received update from non-neighbour {received_ID}. Ignoring packet.")
        return # Ignore updates from non-neighbours
```

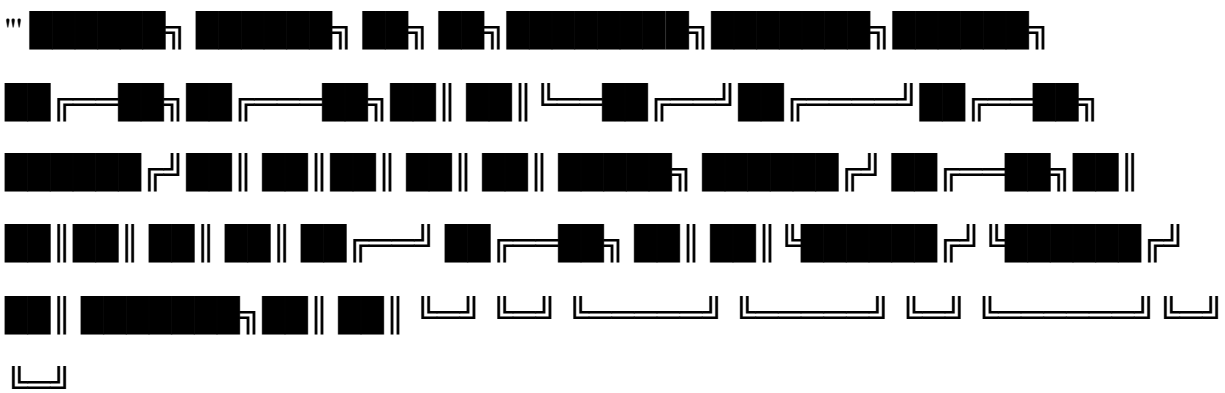
```
    self.routing_table.reset_direct_neighbour_timer(received_ID)

    packet_len = len(packet)
    if packet_len < HDR_SIZE or (packet_len - HDR_SIZE) % RT_SIZE != 0:
        print(f"[ERROR] Invalid packet length: {packet_len}. Must be 6 + N * 20.")
        return

    num_entries = (packet_len - HDR_SIZE) // RT_SIZE

    # Process each route entry in packet
    for entry_index in range(num_entries):
        entry_start_index = HDR_SIZE + entry_index * RT_SIZE
        entry_end_index = entry_start_index + RT_SIZE
        entry_bytes = packet[entry_start_index:entry_end_index]

        if not self.check_entry(entry_bytes):
            # If entry invalid, log the error and skip this entry.
            print(f"[ERROR] Invalid entry at index {entry_index} (starting at byte
{entry_start_index}). Skipping this entry.")
            continue # skip entry

        dest_id = int.from_bytes(entry_bytes[4:8], 'big')
        received_metric  = int.from_bytes(entry_bytes[16:20], 'big')
```

```
        if received_metric >= INF:
            new_metric = INF
        else:
            new_metric = min(received_metric + link_cost_to_sender, INF)

        if new_metric >= INF:
            self.routing_table.mark_unreachable(dest_id)

        else:
            # Otherwise, add or update the route through neighbour
            self.routing_table.add_or_update(dest_id, received_ID, new_metric)

    self.routing_table.prune() # remove dead entries
```

Where the RIP daemon is initiated and the infinite loop for listening for ports occur, RIPv2_router.py:

```
''' Author: Noah & Aljaž Filename: RIPv2_router.py '''

################################### imports from socket import * import select

from threading import Timer from RoutingTable import * from Packet import *
```

import random import time ##################################



'''

LOCAL_HOST = "127.0.0.1" STATUS_PRINT_INTERVAL = 5.0

class RIPv2_Router:

def __init__(self, router_ID, inputs, outputs, timeout=90, garbage_time=60):

    self.router_ID = router_ID

    self.inputs = inputs # Input Ports

    self.outputs = outputs # Output Ports

    # Timeout & Garbage to be done !

    self.timeout = timeout # in seconds

    self.garbage_time = garbage_time

    self.timeout_timer = {} # Dict

    self.garbage_time = {} # Dict

```
# Create input sockets
self.sockets = self.create_sockets()


self.routing_table = RoutingTable(timeout=90, garbage=60)


# Setup to send periodic updates
self.periodic_updates = None
neigh_map = {
   int(link[2]): int(link[1])
   for link in self.outputs
}
self.packet_manager = Packet(self.routing_table,
                        self.router_ID,
                        neigh_map,
                        self.sockets)



for port_s, metric_s, neigh_id_s in self.outputs:
   neigh_id = int(neigh_id_s)
   cost     = int(metric_s)


   self.routing_table.add_or_update(neigh_id,
                              neigh_id,
```

```
                                cost)

    print("Router Daemon Initialised...\n")
    print(f"Router: {self.router_ID}\n")
    print(f"Inputs: {self.inputs}\n")
    print(f"Outputs: {self.outputs}\n")



    # Show the table
    self.routing_table.print_table()


    self.init_periodic_update()



    self._status_timer = None
    self._start_status_timer()



def _start_status_timer(self):
    """Starts the periodic timer for printing the routing table status."""
    # Ensure any existing timer is cancelled before starting a new one
    if self._status_timer:
        self._status_timer.cancel()
```

```
    self._status_timer = Timer(STATUS_PRINT_INTERVAL, self._print_status)
    self._status_timer.daemon = True # Allow program to exit even if this timer is running
    self._status_timer.start()

def _print_status(self):
    """Callback function for the status timer - prints the table and reschedules."""
    print("\n--- Periodic Status Update ---") # periodically prints an update
    self.routing_table.print_table()
    print("-----------------------------\n")

    # Reschedule the timer to run again
    self._start_status_timer()




def receive_packet(self, data):
    # when a packet is received
    self.packet_manager.receive_and_process_packet(data)
    self.routing_table.prune()
    self.routing_table.print_table()




def create_sockets(self):
```

```
    '''
        We initialise the socket so that we can receive packets
    '''
    sockets_list = {}


    try:
        for port in self.inputs:
            sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            sock.setblocking(False) # make it non blocking
            sock.bind((LOCAL_HOST, port))
            sockets_list[port] = sock
        return sockets_list


    except Exception as e:
        print(f"Error creating sockts {e}")



def init_periodic_update(self):
    '''
        Starts the update loop & sends update to neighbours
    '''


    def send_update():
        '''
```

```
            calls helper function to send an update to
            all the neighbour routers and refreshes direct routes
        '''

        self.routing_table.prune()  # Prune dead entries before sending
        self.update_neighbours() # This sends updates based on the *current* table state



        periodic_interval = 30 # Base interval
        plus_minus = random.uniform(-5, 5) # Random offset -5 - 5s
        self.periodic_updates = Timer(periodic_interval + plus_minus, send_update)
        self.periodic_updates.daemon = True # make sure timer doesnt stop exit
        self.periodic_updates.start()

    send_update()



########################## Testing
def update_neighbours(self):
    """
    Send a RIP response to each neighbour. We:
    1) build one or more packets for neighbour
    2) find the correct output port
    3) pick a output socket
```

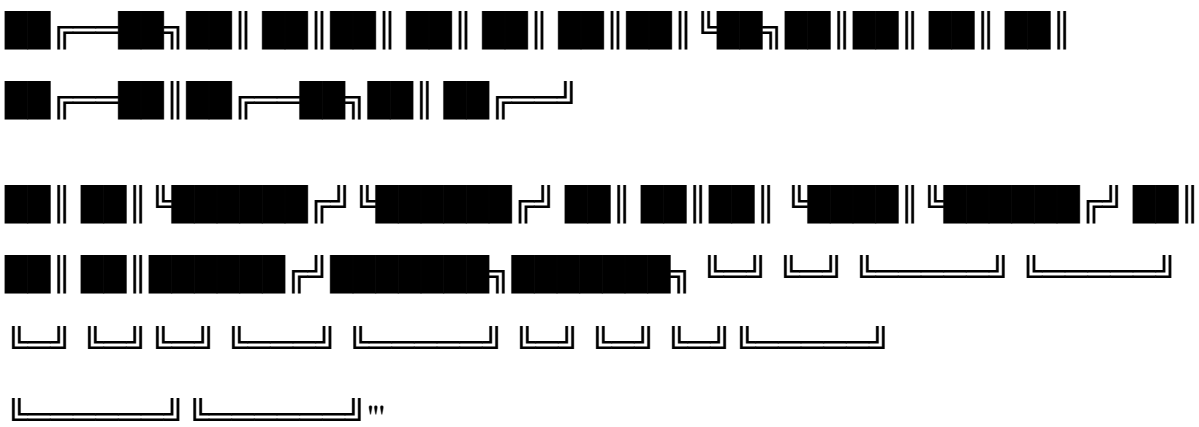
```
4) send each packet.
"""
for neigh_id, link_metric in self.packet_manager.neighbours.items():
    # 1
    packets = self.packet_manager.create_response_packets(neigh_id)
    if not packets:
        print(f"[ERROR] No entries to send to Router {neigh_id}")
        continue

    # 2
    try:
        out_port = next(
            int(entry[0])
            for entry in self.outputs
            if int(entry[2]) == neigh_id
        )
    except StopIteration:
        print(f"[ERROR] No matching output port for neighbour {neigh_id}")
        continue

    # 3
    send_sock = self.sockets[self.inputs[0]]

    # 4
```

```
        for pkt in packets:
            try:
                send_sock.sendto(pkt, (LOCAL_HOST, out_port))

            except Exception as e:
                print(f"[ERROR] Sending to {neigh_id}@{out_port}: {e}")



def monitor_RT(self):
    '''
        This is an infinite loop where we will listen for updates in the sockets and process updates
        and send out updates to other routers.
    '''

    while True:
        readable, _, _ = select.select(list(self.sockets.values()), [], [], 1.0)



        if readable:
            for sock in readable:
                try:
                    data, addr = sock.recvfrom(4096)
```

```
            local_port = sock.getsockname()[1]

            self.packet_manager.receive_and_process_packet(data)

            self.routing_table.prune()

            self.routing_table.print_table()

        except Exception as e:
            print(f"[Error] receiving data: {e}")
```

The Routing table module stores the routing table and prints the updated routing table.
RoutingTable.py:

```
import time import threading

INF = 16

''' ▓▓▓▓▓▓▓╗ ▓▓▓▓▓▓▓╗ ▓▓╗ ▓▓╗▓▓▓▓▓▓▓▓╗▓▓╗▓▓▓▓╗ ▓▓╗ ▓▓▓▓▓▓▓╗
▓▓▓▓▓▓▓▓╗ ▓▓▓▓▓▓╗ ▓▓▓▓▓▓▓╗ ▓▓╗ ▓▓▓▓▓▓▓╗ ▓▓╔═══▓▓╗▓▓╔═══▓▓╗▓▓╗
▓▓╗ ╚═══▓▓╗╔═══▓▓╗▓▓▓▓╗ ▓▓╗▓▓╗╔═══╝
 ╚═══▓▓╗╔═══▓▓╗╔═══▓▓╗╔═══▓▓╗▓▓╗ ▓▓╗╔═══╝ ▓▓▓▓▓▓▓╗╔═══▓▓╗ ▓▓╗▓▓╗
▓▓╗ ▓▓╗ ▓▓╗▓▓╗▓▓▓╗ ▓▓╗▓▓╗▓▓╗ ▓▓▓▓╗ ▓▓╗ ▓▓▓▓▓▓▓▓╗▓▓▓▓▓▓▓╔═══▓▓╗
▓▓▓▓▓▓╗
```



class RTEntry: """ Creates an object for one """

def __init__(self, destination_id: int, next_hop_id: int, metric: int, timeout: float = 90.0, garbage: float = 60.0):

    self.destination_id = destination_id

    self.next_hop_id = next_hop_id

    self.metric = metric

    self._timeout_interval = timeout

    self._garbage_interval = garbage

    self._timeout_timer = None

    self._garbage_timer = None

    self.in_garbage = False

    self._timeout_start_time = None

    self._garbage_start_time = None

```
    # start timeout per entry
    self.reset_timeout()



def reset_timeout(self):
    """
        (Re)start the timeout timer.
    """

    if self._timeout_timer:
        self._timeout_timer.cancel()

    if self._garbage_timer:
        self._garbage_timer.cancel()
        self._garbage_start_time = None

    self._timeout_start_time = time.monotonic() # record start time
    # after timeout, switch to garbage state
    self._timeout_timer = threading.Timer(self._timeout_interval, self._on_timeout)
    self._timeout_timer.daemon = True
    self._timeout_timer.start()

    self.in_garbage = False
```

```
def _on_timeout(self):
    """
        Called when the route times out—mark INF and start garbage.
    """

    if self.metric < INF:
        self.metric =INF


        self.in_garbage = True
        # Start the garbage timer
        self._garbage_start_time = time.monotonic() # Record start time
        self._garbage_timer = threading.Timer(self._garbage_interval, self._on_garbage)
        self._garbage_timer.daemon = True
        self._garbage_timer.start()



def _on_garbage(self):
    """Route has expired and will need to be pruned"""
    pass



def mark_unreachable(self):
    """Mark the route with metric INF and initiate the garbage timer if not already."""
```

```
    if self.metric < INF: # Only if not already INF
        self.metric = INF

        if not self.in_garbage: # Start garbage timer only if not already in garbage
            self.in_garbage = True
            # Start garbage countdown
            # Cancel any cancel timeouts
            if self._timeout_timer:
                self._timeout_timer.cancel()
                self._timeout_start_time = None

            self._garbage_start_time = time.monotonic() # Record start time
            self._garbage_timer = threading.Timer(self._garbage_interval, self._on_garbage)
            self._garbage_timer.daemon = True
            self._garbage_timer.start()


def cancel_timers(self):
    """Cancel both timeout and garbage timers."""
    if self._timeout_timer:
        self._timeout_timer.cancel()
        self._timeout_start_time = None
    if self._garbage_timer:
        self._garbage_timer.cancel()
```

```
        self._garbage_start_time = None


def is_dead(self) -> bool:
    """After garbage_interval has elapsed, this entry is ready for removal."""
    return self.in_garbage and (self._garbage_timer is not None
                                and not self._garbage_timer.is_alive())


def __repr__(self):
    '''
        Prints info about the routing table entry
    '''
    status = "Active"
    timeout_display = "N/A"
    garbage_display = "N/A"

    current_time = time.monotonic()

    if self.metric == INF:
        status = "Unreachable (Garbage)"
        # route marked as inf or timeout is expired
        timeout_display = "Expired"

        if self.in_garbage and self._garbage_timer and self._garbage_start_time is not None:
            # Calculate remaining garbage time
```

```
            elapsed_garbage_time = current_time - self._garbage_start_time
            remaining_garbage_time = max(0.0, self._garbage_interval - elapsed_garbage_time)
            garbage_display = f"{remaining_garbage_time:.1f}s left"

        elif self.in_garbage:
            garbage_display = "Timer not running" # in the garbage

        else:
            garbage_display = "Garbage not started"



    elif self._timeout_timer and self._timeout_start_time is not None:
        status = "Active"
        # Calculate remaining timeout time
        elapsed_timeout_time = current_time - self._timeout_start_time
        remaining_timeout_time = max(0.0, self._timeout_interval - elapsed_timeout_time)
        timeout_display = f"{remaining_timeout_time:.1f}s left"
        garbage_display = "Inactive"



    return (
        f"\n############# Routing Table Entry #############\n"
        f"\nDestination: {self.destination_id}\n"
        f"Next Hop: {self.next_hop_id}\n"
```

```
        f"Metric: {self.metric}\n"
        f"Status: {status}\n"
        f"Timeout: {timeout_display}\n" # show time left
        f"Garbage: {garbage_display}\n" # show time left
        f"\n################################################\n"
        )


class RoutingTable: """ Manages a set of RTEntry instances. key: destination_id """

def __init__(self, timeout: float = 90.0, garbage: float = 60.0): # 90 and 60
    self._entries = {}  # dst_id -> RTEntry
    self._timeout = timeout
    self._garbage = garbage
    self._lock = threading.Lock()



def add_or_update(self, destination_id: int, next_hop_id: int, metric: int):
    """
        Route expired and will need to be pruned

    """



    with self._lock:
```

```
        if destination_id not in self._entries:

            e = RTEntry(destination_id, next_hop_id, metric, timeout=self._timeout,
garbage=self._garbage)

            self._entries[destination_id] = e


        else:
            e = self._entries[destination_id]
            current_metric = e.metric
            is_from_current_next_hop = (e.next_hop_id == next_hop_id)

            if is_from_current_next_hop:

                if metric == current_metric:

                    e.reset_timeout()

                elif metric < INF:
                    e.metric = metric
                    e.reset_timeout()
```

```
            else:
                e.mark_unreachable() # marks route as unreachable

        else:
            if metric < current_metric:
                e.next_hop_id = next_hop_id
                e.metric = metric
                e.reset_timeout()



def mark_unreachable(self, destination_id: int):
    """
        Set the route's metric to INF and let its garbage timer run out
    """

    with self._lock:
        if destination_id in self._entries:
            e = self._entries[destination_id]
            e.mark_unreachable()



def prune(self):
    """
```

```
        deletes entries from routing table when garbage entry is complete
    """

    with self._lock:
        to_delete = [dst for dst, e in self._entries.items() if e.is_dead()]

        for dst in to_delete:
            e = self._entries.pop(dst)
            e.cancel_timers()


def __iter__(self):
    """
        makes routing table iterable; usable with for loops
    """
    with self._lock:

        return iter(list(self._entries.values()))


def __len__(self):
    """
        Allows us to call len() to get the length of the routing table
    """
```

```
    with self._lock:
        return len(self._entries)



def __repr__(self):
    """
        Allows us to represent the objects as a string
    """
    with self._lock:
        lines = [f"RoutingTable (Entries: {len(self._entries)}):"]


        if not self._entries:
            lines.append("  (empty) ")


        else:


            sorted_entries = sorted(self._entries.values(), key=lambda e: e.destination_id)
            for e in sorted_entries:
                entry_lines = repr(e).strip().split('\n')
                for line in entry_lines:
                    lines.append("  " + line)
        return "\n".join(lines)
```

```
def print_table(self):
    """
        call repr() to print the table
    """
    print(self.__repr__())



def reset_direct_neighbour_timer(self, neighbour_id: int):
    """
        check if an entry is connected to a neighbour.
    """

    with self._lock:

        if neighbour_id in self._entries:
            e = self._entries[neighbour_id]

            if e.destination_id == neighbour_id and e.next_hop_id == neighbour_id:
                e.reset_timeout()
```