364 Assignment

<u>Kahu Griffin</u>

kgr78

28226702

<u>Ismail Sarwari</u>

Isa50

737126737

Contributions

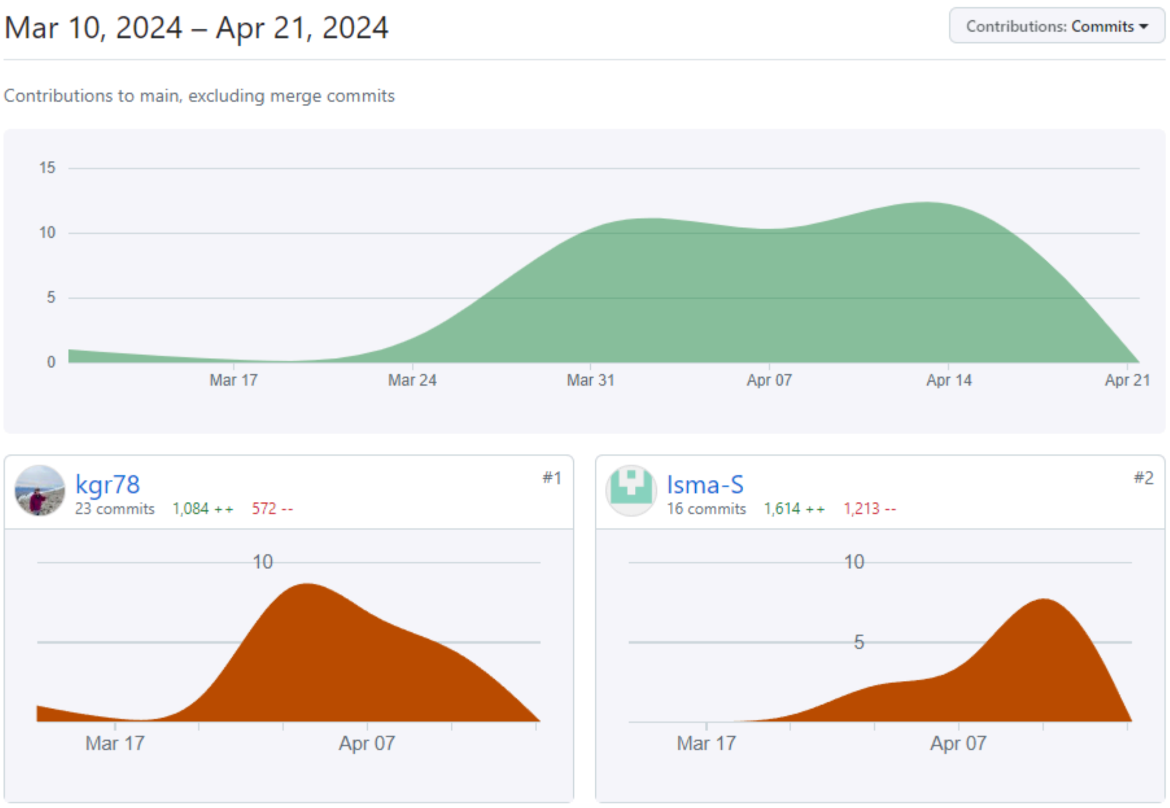
Partner	Kahu Griffin	Ismail
Contributions	45%	55%

Kahu Griffin Contributions -

- Config file set up and creation.
- Github creation
- Configfile Reader
- Report
- Rip Protocol
- Testing

Ismail Sarwari Contributions -

- Config file set up and creation.
- Configfile Reader
- Report
- Rip Protocol
- Testing
- Router
- Route
- Print Table

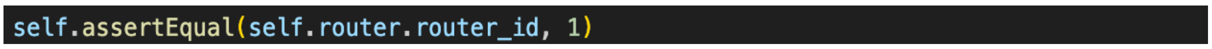


Test:

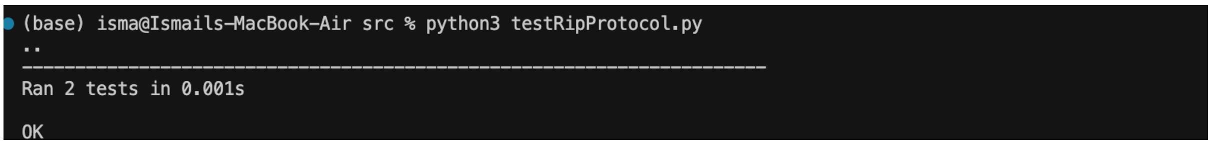
Test is a crucial part of programming as it helps with correctness and reliability of the program.For the Rip Assignment three test classes (testConfigParser.py, testRipProtocol.py, testRoutingTable.py) to analyse different aspect of Rip program. For each test class the unittest has been used to assert the expected value. If the assertion does not match then an error would occur. Also an error handler has been implemented in RipProtocol.py to catch and print any errors that occur.

Condition for router 1

Successful test

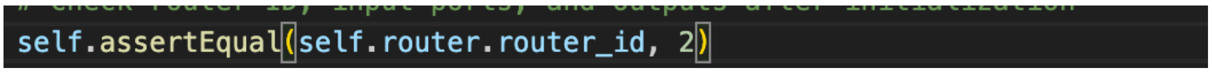


```
self.assertEqual(self.router.router_id, 1)
```

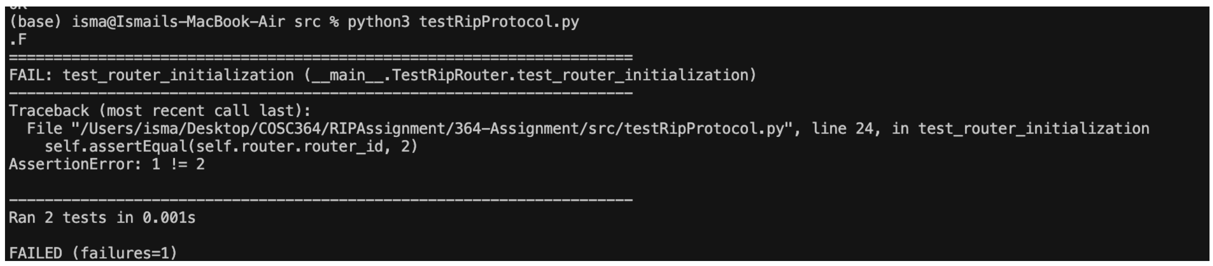


```
(base) isma@Ismails-MacBook-Air src % python3 testRipProtocol.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Failed test



```
self.assertEqual(self.router.router_id, 2)
```



```
(base) isma@Ismails-MacBook-Air src % python3 testRipProtocol.py
.F
======================================================================
FAIL: test_router_initialization (__main__.TestRipRouter.test_router_initialization)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/isma/Desktop/COSC364/RIPAssignment/364-Assignment/src/testRipProtocol.py", line 24, in test_router_initialization
    self.assertEqual(self.router.router_id, 2)
AssertionError: 1 != 2

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

The test covers the critical region, however the implementation is limited to using the unittest. This could be further improved with the unittest tools (magic mock etc), however due to limited knowledge, only able to use the asserted tool, and this also limits how much can be tested due to object oriented programs. The test also covers around an estimated 40% of the program due to limited time as well. However the test does cover the critical regions.

ConfigParser test:

The configparser class has been testing with a given file in a list,(rather than reading the file) and then checking the expected Value. This ensures that the class initialises the Router object and then uses the setters and getters function to get the expected value.

RipProtocol test:

The RipProtocol class has been tested with initialising a router object with the given config files, then checking each function to behave as expected. The test ensures that when create_rip_packet() calls ,the return value (message) is a bytearray and that the array is also not empty, the unit test will pick up if it doesn't meet the requirement or if an error has occurred. The router object has also been tested in this class, and ensures that the router attribute has the expected value. Altho this may not be the most optimal way of testing, this demonstrates that the class functions behave as intended.

RoutingTable test:

Simple class that checks that the known and unknown route has been added to the routing table successfully, and verifies route object attributes. This class can be further improved with implementation of times.

testRipProtocol.py

```python
import unittest
from ConfigParser import ConfigParser
from RipProtocol import RipRouter


class TestRipRouter(unittest.TestCase):


   def setUp(self):
       # Set up router with a sample configuration file
       self.router = RipRouter("config1.txt")
```

```
    def tearDown(self):
        # Close sockets to avoid ResourceWarning
        self.router.close_input_sockets()

    def test_create_rip_packet(self):
        # Test create_rip_packet method
        message = self.router.create_rip_packet()
        self.assertIsInstance(message, bytearray)
        self.assertNotEqual(len(message), 0)

    def test_router_initialization(self):
        # Check router ID, input ports, and outputs after initialization
        self.assertEqual(self.router.router_id, 1)
        self.assertListEqual(self.router.input_ports, [5000, 5001, 5002])
        self.assertListEqual(self.router.output_ports, [5003, 5004, 5005])
        self.assertEqual(len(self.router.routing_table.routes), 0)

if __name__ == '__main__':
    unittest.main()
```

testConfigParser.py

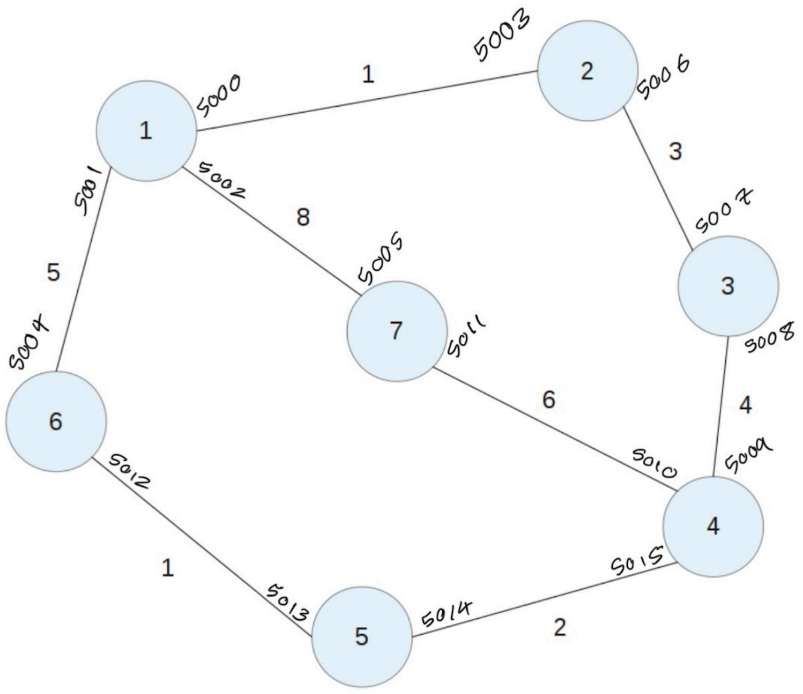
```
import unittest
from ConfigParser import ConfigParser

class TestConfigParser(unittest.TestCase):
    def test_valid_config(self):
        print("Testing valid configuration...")
        config_parser = ConfigParser()
        router = config_parser.read_config_data([
            "router-id, 1",
            "input-ports, 5000, 5001, 5002",
            "outputs, 5003-1-2, 5004-5-6, 5005-8-7"
        ])
        self.assertEqual(router.get_router_id(), 1)
        self.assertEqual(router.get_input_ports(), [5000, 5001,
5002])
        self.assertEqual(router.get_outputs(), [5003, 5004, 5005])
        print("Valid configuration test passed.")
```

```python
    def test_invalid_config(self):
        print("Testing invalid configuration...")
        invalid_config_data = [
            "router-id, 1",
            "input-ports, 5000, 5001, 5002",
            "outputs, 5003-1-2, 5004-5-6, 5005-8-7",
            "invalid-section, data"
        ]

        config_parser = ConfigParser()
        with self.assertRaises(ValueError):
            router =
config_parser.read_config_data(invalid_config_data)

        print("Invalid configuration test passed.")

    def test_duplicate_entries(self):
        print("Testing duplicate entries...")
        duplicate_config_data = [
            "router-id, 1",
            "input-ports, 5000, 5001, 5002",
            "outputs, 5003-1-2, 5004-5-6, 5005-8-7",
            "outputs, 5006-4-3, 5007-7-9"
        ]

        config_parser = ConfigParser()
        with self.assertRaises(ValueError):
            router =
config_parser.read_config_data(duplicate_config_data)
        print("Duplicate entries test passed.")

if __name__ == '__main__':
    unittest.main()
```

Config1.txt

```
router-id, 1
input-ports, 5000, 5001, 5002
outputs, 5003-1-2, 5004-5-6, 5005-8-7
```

Network Topology:



Source code

RipProtocol.py

```
import random
import select
import socket
import sys
import datetime
from time import sleep
from RoutingTable import *
from ConfigParser import *

LOCAL_HOST = "127.0.0.1"
```

```
class ErrorHandler:
    """
    ErrorHandler class logs error and prints error messages if any error occurs.
Note: must set the print_logs flag to True.
    """
    def __init__(self, print_logs):
        self.print_logs = print_logs
    def log(self, message):
        if self.print_logs:
            print(message)
class RipRouter:
    """
    The RipRouter class represents a router implementing the RIP (Routing
Information Protocol) protocol v2.
    It handles the initialization of the router with configuration data, sending and
receiving RIP packets,
    updating the routing table based on received packets, and managing periodic
updates and route timers.
    """
    def __init__(self, config_filename):
        """
        Initializes the RipRouter object with the given configuration file.
        Parameters:
            config_filename (str): The name of the configuration file.
        """
        self.error_handler = ErrorHandler(print_logs=False)
        config_parser = ConfigParser()
        self.router = config_parser.read_config_file(config_filename)
        self.router_id = self.router.get_router_id()
        self.input_ports = self.router.get_input_ports()
        self.output_ports = self.router.get_outputs()
        self.input_sockets = self.setup_input_sockets()
        self.routing_table = RoutingTable()
        self.routing_table.set_router_id(self.router_id)
        self.periodic_update_timer = datetime.datetime.now()

    def get_outputs(self):
        """
        Returns:
            output_ports (list): The list of output ports.
        """
        return self.output_ports
    def setup_input_sockets(self):
        """
        Sets up the input sockets for the RIP protocol.
        Returns:
            sockets (list): A list of input sockets.
        """
        sockets = []
        for port in self.input_ports:
            sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            sock.bind((LOCAL_HOST, port))
            sockets.append(sock)
        return sockets

    def create_rip_packet(self):
        """
        Creates a RIP packet from the routing table.
        Returns:
            packet (bytearray): The created RIP packet.
        """

        packet = bytearray(4)
        header_command = 0x02
        header_version = 0x02
        packet[0] = header_command
        packet[1] = header_version

        packet[2] = self.router_id >> 8
        packet[3] = (self.router_id & 0x00FF)

        for route in self.routing_table.routes:
            rip_entry = bytearray(20)

            # Check if the destination is valid
            if route.destination < 1 or route.destination > 64000:
                self.error_handler.log(f"Error: Invalid destination value, 
destination: {route.destination}")
            else:
                rip_entry[4] = route.destination >> 24
                rip_entry[5] = (route.destination & 0x00FF0000) >> 16
                rip_entry[6] = (route.destination & 0x0000FF00) >> 8
                rip_entry[7] = (route.destination & 0x000000FF)

            # Check if the metric is valid
            if route.metric < 1 or route.metric > 16:
                self.error_handler.log(f"Error: Invalid metric value, metric: 
{route.metric}")
            else:
                rip_entry[16] = route.metric >> 24
                rip_entry[17] = (route.metric & 0x00FF0000) >> 16
                rip_entry[18] = (route.metric & 0x0000FF00) >> 8
                rip_entry[19] = (route.metric & 0x000000FF)

            packet.extend(rip_entry)

        return packet



    def send_rip_packets(self):
        """
        Sends RIP packets to all the output ports.
        """
        message = self.create_rip_packet()
        send_socket = self.input_sockets[0]
        for port in self.router.get_outputs():
            try:
                send_socket.sendto(message, (LOCAL_HOST, port))
            except socket.error as e:
                self.error_handler.log(f"Error occurred while sending RIP packet to
port {port}: {e}")
    def close_input_sockets(self):
        """
        Closes all the input sockets. Used for testing, else it will cause an
resource allocation error.
        """
        for socket in self.input_sockets:
            socket.close()
        self.input_sockets.clear()

    def receive_packets(self):
        """
        Receives RIP packets from all the input ports.
        """
        timeout = 1
        readable_sockets, _, _ = select.select(self.input_sockets, [], [], timeout)

        for socket in readable_sockets:
            data, _ = socket.recvfrom(1024)
            self.process_received_packet(data)

    def process_received_packet(self, data):
        """
        Processes a received RIP packet and updates the routing table if necessary.
If updated than sends a new RIP packet.
        Parameters:
            data (bytearray): The received RIP packet.

        """
        routing_table_updated = False

        rip_header = data[:4]
        rip_data = data[4:]
        command = rip_header[0]
        version = rip_header[1]
        next_hop_router_id = int.from_bytes(rip_header[2:], "big")

        if command != 2:
            self.error_handler.log(f"Error: Command is invalid, command:{command}")
            return
        if version != 2:
            self.error_handler.log(f"Error: Version is invalid, version:{version}")
            return
        if not (0 < next_hop_router_id < 64001):
            self.error_handler.log(f"Error: Router id is invalid, router
id:{next_hop_router_id}")
            return
        if not self.router.is_router_in_outputs(next_hop_router_id):
            self.error_handler.log(f"Dropping packet: Router id not in outputs,
router id:{next_hop_router_id}")
            return

        output = self.router.get_output_by_router_id(next_hop_router_id)
        if not self.routing_table.is_route_known(next_hop_router_id):
            self.routing_table.add_route(next_hop_router_id, next_hop_router_id,
output["metric"])
            self.error_handler.log("is_route_known == False")
            routing_table_updated = True
        else:
            route = self.routing_table.get_route_id_by_id(next_hop_router_id)
            if output["metric"] < route.metric:
                route.update_route(next_hop_router_id, next_hop_router_id,
output["metric"])
                routing_table_updated = True
            if route.next_hop == next_hop_router_id:
                route.reset_timers()

        routes = []
        for i in range(0, int(len(rip_data)), 20):
            routes.append(rip_data[i:i + 20])

        for route in routes:
            afi = int.from_bytes(route[:2], 'big')
            if afi != 0:
                self.error_handler.log(f"Error: afi is invalid, afi:{afi}")
            else:
                router_id = int.from_bytes(route[4:8], 'big')
                self.error_handler.log(f"Route router id: {router_id}")
                if 0 < router_id < 64001:
                    if router_id != self.router_id:
                        output =
self.router.get_output_by_router_id(next_hop_router_id)
                        metric = int.from_bytes(route[16:], 'big')
                        route_object =
self.routing_table.get_route_id_by_id(router_id)

                        if (0 < (metric + output["metric"]) < 17) or metric == 16:
                            if route_object:
                                if route_object.next_hop == next_hop_router_id:
                                    if metric == 16:
                                        if route_object.garbage_timer is None:
                                            route_object.mark_for_deletion()
                                            routing_table_updated = True
                                    elif route_object.metric != metric +
output["metric"]:
                                        routing_table_updated = True
                                    else:
                                        route_object.reset_timers()
                                elif metric + output["metric"] <
route_object.metric:

route_object.update_route(route_object.destination, next_hop_router_id,
                                                            metric +
output["metric"])
                                    routing_table_updated = True
                            elif metric < 16:
                                self.routing_table.add_route(router_id,
next_hop_router_id,
                                                            metric +
output["metric"])
                                routing_table_updated = True
                        else:
                            self.error_handler.log(f"Error: metric out of bound,
metric:{metric}")

        if routing_table_updated:
            self.send_rip_packets()





    def get_update_timer_duration(self):
        """
        Calculates the duration since the last periodic update
        """
        return (datetime.datetime.now() - self.periodic_update_timer).seconds

    def reset_periodic_update_timer(self):
        """
        Resets the periodic update timer
        """
        self.periodic_update_timer = datetime.datetime.now()

    def check_timeout_entries_periodically(self):
        """
        Periodically checks for timeout entries in the routing table and sends RIP
packets if needed.
        The 30-second timer is offset by a small random time (+/- 0 to 5 seconds)
each time it is set.
        (Implementors may wish to consider even larger variation in the light of
recent research results [10])
        """
        random_offset_period = 11 + random.randrange(-5, 5)
        if self.get_update_timer_duration() > random_offset_period:

            self.send_rip_packets()
            self.reset_periodic_update_timer()

    def check_route_timers(self):
        """
        Checks for timeout entries in the routing table and sends RIP packets if
needed.
        """
        if self.routing_table.check_route_timers():
            self.send_rip_packets()
    def rip_protocol(self):
        """
        The main RIP protocol loop.
        """

        self.send_rip_packets()

        while(1):
            self.routing_table.print_table()
            self.receive_packets()
            self.check_timeout_entries_periodically()
            self.check_route_timers()
            sleep(1)

def main(config_filename):
    try:
        router = RipRouter(config_filename)
        router.rip_protocol()
    except Exception as exception:
            print(exception)

if __name__=="__main__":
    if len(sys.argv) != 2:
        print("Error: Invalid number of arguments, usage: python3 RipProtocol.py
<config_filename>")
    else:
        config_filename = sys.argv[1]
        main(config_filename)
```

Router.py

```
from Router import Router

class ConfigParser:
   def __init__(self):
       self.router = Router()

   def validate_config(self, config_data):
       if len(config_data) != 3:
           raise ValueError("Config format is invalid. Each of 'router-id',
'input-ports', and 'outputs' must be specified on separate lines.")

       headers = [item[0][0] for item in config_data]
       if len(set(headers)) != len(headers):
           raise ValueError("Duplicate config headers found. Each header must be
unique.")

   def validate_router_id(self, router_id_data, line_num):
       if len(router_id_data) != 2:
           raise ValueError(f"At line {line_num}, the router ID format is
incorrect. Use 'router-id, {{integer between 1 and 64000}}'")

       try:
           router_id = int(router_id_data[1])
           if not (1 <= router_id <= 64000):
               raise ValueError(f"At line {line_num}, the router ID is invalid. It
must be an integer between 1 and 64000.")
       except ValueError:
           raise ValueError(f"At line {line_num}, the router ID is invalid. It must
be an integer.")
       self.router.set_router_id(router_id)
   def validate_input_ports(self, input_ports_data, line_num):
       try:
           for port_index, port_str in enumerate(input_ports_data[1:], start=1):
               port = int(port_str)
               if not (1024 <= port <= 64000):
                   raise ValueError(f"At line {line_num}, the input port number is
invalid. Port {port_index} must be between 1024 and 64000.")
               if port in self.router.get_input_ports():
                   raise ValueError(f"At line {line_num}, a duplicate input port
number is found. Port {port_index} is repeated.")
               self.router.add_input_port(port)
        except ValueError:
           raise ValueError(f"At line {line_num}, the input port number is invalid.
Port {port_index} must be an integer.")

   def validate_output_links(self, outputs_data, line_num):
       for port_index, port_str in enumerate(outputs_data[1:], start=1):
           try:
               port, metric, router_id = map(int, port_str.split("-"))
           except ValueError:
               raise ValueError(f"At line {line_num}, the output format is invalid.
Expected 'port-metric-router_id'.")

           if not (1024 <= port <= 64000):
               raise ValueError(f"At line {line_num}, the output port number is
invalid. Port {port_index} must be between 1024 and 64000.")
           if router_id == self.router.router_id:
               raise ValueError(f"At line {line_num}, the output router ID is
invalid. Router ID for port {port_index} must not be the same as the host router
ID.")
           if port in self.router.get_outputs():
               raise ValueError(f"At line {line_num}, a duplicate output port
number is found. Port {port_index} is repeated.")
           if port in self.router.input_ports:
               raise ValueError(f"At line {line_num}, the output port number is
also used as an input port.")

           self.router.add_output(port, metric, router_id)

   def read_config_file(self, file_name):
       try:
           with open(file_name, 'r') as config_file:
               config = config_file.readlines()
       except FileNotFoundError:
           print(f"Error: The configuration file '{file_name}' was not found.")
           return
       parse_config = []
       for line_num, line in enumerate(config, start=1):
           line = line.strip()  # Remove leading/trailing spaces
           if line:
               parse_config.append((line.split(', '), line_num))

       self.validate_config(parse_config)

       for header, line_num in parse_config:
           if header[0] == 'router-id':
               self.validate_router_id(header, line_num)
           elif header[0] == 'input-ports':
               self.validate_input_ports(header, line_num)
           elif header[0] == 'outputs':
               self.validate_output_links(header, line_num)
           else:
               raise ValueError(f"At line {line_num}, the header '{header[0]}' is
invalid.")

       return self.router
   def read_config_data(self, config_data):
       parse_config = []
       for line_num, line in enumerate(config_data, start=1):
           line = line.strip()
           if line:
               parse_config.append((line.split(', '), line_num))
       self.validate_config(parse_config)

       for header, line_num in parse_config:
           if header[0] == 'router-id':
               self.validate_router_id(header, line_num)
           elif header[0] == 'input-ports':
               self.validate_input_ports(header, line_num)
           elif header[0] == 'outputs':
               self.validate_output_links(header, line_num)
           else:
               raise ValueError(f"At line {line_num}, the header '{header[0]}' is invalid.")

       return self.router
```

ConfigParser.py

```
class Router:

    def __init__(self):
        self.outputs = []
        self.router_id = None
        self.input_ports = []
    def add_input_port(self, port):
        self.input_ports.append(port)
    def get_input_ports(self):
        return self.input_ports
    def set_router_id(self, router_id):
        self.router_id = router_id
    def get_router_id(self):
        return self.router_id
    def create_output(self, port, metric, router_id):
        return {
            'port': port,
            'metric': metric,
            'router_id': router_id
        }

    def add_output(self, port, metric, router_id):

        self.outputs.append(self.create_output(port, metric, router_id))

    def get_output_by_router_id(self, router_id):
        for output in self.outputs:
            if output['router_id'] == router_id:
                return output

    def get_outputs(self):
        return [output['port'] for output in self.outputs]

    def is_router_in_outputs(self, router_id):

        return router_id in [output['router_id'] for output in self.outputs]

    def print_outputs(self):

        for output in self.outputs:
            print(output)
```

Route.py

```
import datetime
from gc import garbage
import time



class Route:
    def __init__(self, destination, next_hop, metric):
        self.destination = destination
        self.next_hop = next_hop
        self.metric = metric
        self.deletion_timer = datetime.datetime.now()
        self.garbage_timer = None
        self.timer_limit = 30
        self.router_id = None

    def set_router_id(self, router_id):
        self.router_id = router_id

    def get_deletion_timer(self):

        if self.deletion_timer:
            return (datetime.datetime.now() - self.deletion_timer).seconds
        return 0

    def get_garbage_timer(self):
        if self.garbage_timer:
            return (datetime.datetime.now() - self.garbage_timer).seconds
        return 0

    def reset_timers(self):

        self.deletion_timer = datetime.datetime.now()
        self.garbage_timer = None

    def mark_for_deletion(self):

        self.deletion_timer = None
        self.garbage_timer = datetime.datetime.now()
        self.metric = 16


    def check_timers(self):

        if self.get_garbage_timer() > self.timer_limit:
            return 0
        if self.get_deletion_timer() > self.timer_limit:
            self.mark_for_deletion()
            return 1
        return 2

    def update_route(self, destination, next_hop, metric):

        self.destination = destination
        self.next_hop = next_hop
        self.metric = metric
        self.reset_timers()
```

RoutingTable.py

```
Import os
from Route import *

class RoutingTable:
    def __init__(self):
        self.routes = []
        self.router_id = None
```

```
   def print_table(self):

       table = [
           "                                           Router Id: {}
".format(self.router_id),

"+-----------------+-----------------+-----------------+-----------------+---------
--------+-----------------+",
           "| Destination     | Next Hop        | Metric          | Deletion Timer
| Garbage Timer   | State           |",

"+-----------------+-----------------+-----------------+-----------------+---------
--------+-----------------+"
       ]

       for route in self.routes:
           if route.get_garbage_timer() > 2:
               state = "Unreachable"
           else:
               state = "Active"
           table.append("| {0:<15} | {1:<15} | {2:<15} | {3:<15} | {4:<15} |
{5:<15} |".format(route.destination, route.next_hop, route.metric,
route.get_deletion_timer(), route.get_garbage_timer(), state))

table.append("+-----------------+-----------------+-----------------+--------------
---+-----------------+-----------------+")
       if os.name == 'nt':  # Windows
           os.system('cls')
       else:  # Linux or macOS
           os.system('clear')
       print("\n".join(table))

   def set_router_id(self, router_id):
       self.router_id = router_id

   def is_route_known(self, router_id):

       return router_id in [route.destination for route in self.routes]

   def add_route(self, destination, next_hop, metric):

       self.routes.append(Route(destination, next_hop, metric))
       self.routes = sorted(self.routes, key=lambda x: x.destination)
   def get_route_id_by_id(self, router_id):

       for route in self.routes:
           if route.destination == router_id:
               return route

   def check_route_timers(self):

       state = False
       routes_to_remove = set()

       for route in self.routes:
           timer_check_result = route.check_timers()

           if timer_check_result == 0:
               routes_to_remove.add(route)
               routes_to_remove.update([x for x in self.routes if x.next_hop ==
route.destination])
           elif timer_check_result in [0, 1]:
               state = True

       if routes_to_remove:
           updated_routes = []
           for route in self.routes:
               if route not in routes_to_remove:
                   updated_routes.append(route)
           self.routes = updated_routes
       return state
```