COSC364 assignment
RIP ROUTING PROTOCOL IMPLEMENTATION

Mohadesa Sharifi
89853938

Partner contribution

This assignment is done individually due to special consideration. I sent an email to Dr Willig on 17/3/2023 at 10:03 AM that I want to do the assignment without partner contribution.

Questions

There should be a substantial discussion about the testing you have performed (for each test: what was to be tested, what was the expected outcome and what was the actual outcome) and which conclusions these tests allow about the correctness of your design and implementation.

Unit tests

Test config

Configuration files must have three mandatory routing parameters: router-id, input-ports, and outputs. I test read_config(filename) function with configTest1.txt which includes all three parameters as below.

```
router-id 1
input-ports 10012, 10013
outputs 10021-2-2, 10031-7-3
```

The expected outcome is a configs dictionary: configs[“router-id”] equals 1,
configs[“Input-ports”] equal [10012, 10013],
configs[“outputs”] has a list of two dictionaries ->
configs[“outputs”][0] equals {“dest_id”: 2, “metric”: 2, “next_hop”: 10021} and
configs[“outputs”][1] equals {“dest_id”: 3, “metric”: 7, “next_hop”: 10031}
The test passes successfully.

Test utils

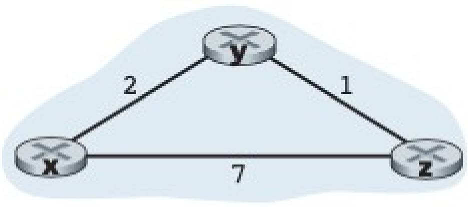
I test Create_rip_packet, procces_rip_packet, create_server_socket methods.
I make a rip packet with next_hop = 2, metric = 3, and dest_id = 7. When proccessing a rip packet I need to retrieve the exact same data. The tests passes succesfully.
Creating a server socket, I test that the socket type eqauls “SOCK_DGRAM”, socket family name equals “AF_INIT” and socket is created on correct values of port numbers.

Test Distance Victor

I test routing table, server sockets, source_router_id, and garbage are correctly initialized.

End to end test

Finding shortest path



At time T0 Router 1 is on. Router 2, and 3 are off. Forwarding table is empty.

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 1
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
-------------------------------------------------------------
```

We turn on router 3 at time T1
at T2 Router 1 updates its forwarding table after hearing from router 3

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 1
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
     3    |     3    |     7    |  1.00     | True
-------------------------------------------------------------
```

At T3 router 3 updates its forwarding table when hears from Router 1

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 3
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
     1    |     1    |     7    |  1.00     | True
-------------------------------------------------------------
```

At T4 router 2 is on. First routing table is empty but after a periodic time Router 2 knows about router 1 and 3.

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 2
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
     1    |     1    |     2    |  1.00     | True
     3    |     3    |     1    |  1.00     | True
-------------------------------------------------------------
```

At T5 router 1 knows that there is a shorter path to router 3 via router 2. it updates its routing table**.**

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 1
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
     2    |     2    |     2    |  1.00     | True
     3    |     2    |     3    |  1.00     | True
-------------------------------------------------------------
```

At T6 router 3 knows there is a shorter path to router 1 via router 2. it updates its routing table.

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 3
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
     1    |     2    |     3    |  1.00     | True
     2    |     2    |     1    |  1.00     | True
-------------------------------------------------------------
```

Now all routers have identical routing information.

At T7 we close router 2.
Router 1, and 3 increase the timer. When the timeout occures Router 1, and 3 make reachable to router 2 false and reroute. By the time garbage timer finishes. Router 2 is deleted from routing table.

Router 1 and router 3 routing tables are as follow after convergence.

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 1
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
    3    |     3    |     7    |  1.00     | True
-------------------------------------------------------------
```

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 3
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
    1    |     1    |     7    |  1.00     | True
-------------------------------------------------------------
```

We turn on router 2 again and routers are able to reroute and find the shortest path again.

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 1
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
    2    |     2    |     2    |  1.00     | True
    3    |     2    |     3    |  1.00     | True
-------------------------------------------------------------
```

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 2
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
    1    |     1    |     2    |  1.00     | True
    3    |     3    |     1    |  1.00     | True
-------------------------------------------------------------
```

```
-------------------------------------------------------------
FORWARDING TABLE OF ROUTER = 3
  Router   | Next-hop  |  Metric   |   Timer    | Reachable
    1    |     2    |     3    |  1.00     | True
    2    |     2    |     1    |  1.00     | True
-------------------------------------------------------------
```

Cofig file

configuration file format
X = router-id, Y = neighbor, M = metric

```
router-id X
imput-ports 100XY
outputs 100YXMY

router-id 1
input-ports 10012, 10016, 10017
outputs 10021-1-2, 10061-5-6, 10071-8-7

router-id 2
input-ports 10021, 10023
outputs 10012-1-1, 10032-3-3

router-id 3
input-ports 10032, 10034
outputs 10023-3-2, 10043-4-4

router-id 4
input-ports 10043, 10045, 10047
outputs 10034-4-3, 10054-2-5, 10074-6-7

router-id 5
input-ports 10054, 10056
outputs 10045-2-4, 10065-1-6

router-id 6
input-ports 10061, 10065
outputs 10016-5-1, 10056-1-5

router-id 7
input-ports 10071, 10074
outputs 10017-8-1, 10047-6-4
```

Demon.py

```python
from pprint import pprint
from random import random
import sys
import os
import config as configuration
import dv
from time import time, sleep
def main():
    """Main function to run the program"""
    try:
        filename = sys.argv[1]
    except IndexError:
        print("Error: no filename given")
        sys.exit()

    if os.path.exists(filename):
        config = configuration.read_config(file_name=filename)
    else:
        print("Error: file does not exist")
        sys.exit()
    distance_vector = dv.DistanceVector(config=config)
    while True:
        distance_vector.timer()
        distance_vector.periodic_event()
        distance_vector.print_routing_table()
        distance_vector.wait_for_packets()
if __name__ == "__main__":
    main()
```

Config.py

```python
import os
import socket
import sys
import time
from pprint import pprint
from select import select
import numpy as np

HOST = "127.0.0.1"
INFINITY = 16
PORTS = []
SOCKETS = []
ROUTER_IDS = []
ROUTES = []
OUTPUTS = []
MY_ROUTER = None
MIN_VALID_PORT = 1024
MAX_VALID_PORT = 64000
MIN_VALID_ROUTER_ID= 1
MAX_VALID_ROUTER_ID= 64000
MIN_VALID_COST=1 #we use cost and metric interchangeably
MAX_VALID_COST=15
INVALID_COST=16

def validate_router_id(id: int):
    """Validate router id
    """
    if id < MIN_VALID_ROUTER_ID or id > MAX_VALID_ROUTER_ID:
        pprint("router-id is not in valid range or router-id is not unique")
        sys.exit()
    return id


def validate_input_ports(input_ports):
    """Validate input ports
    """
    if len(set(input_ports)) != len(input_ports):
        print("One or more input port is repeated.")
        sys.exit()
    for port in input_ports:
        if port < MIN_VALID_PORT or port > MAX_VALID_PORT:
            pprint("port number is not in valid range")
            sys.exit()
    return input_ports


def validate_output(outputs, neighbor_ports):
    """Validate output
    """
    for output in outputs:
        port, metric, router = output.split("-")
        if int(port) < MIN_VALID_PORT or int(port) > MAX_VALID_PORT:
            pprint("port number is not valid range")
            sys.exit()
        if int(router) < MIN_VALID_ROUTER_ID or int(router) > MAX_VALID_ROUTER_ID:
            pprint("port number is not valid range")
            sys.exit()
        if int(metric) < MIN_VALID_COST or int(metric) > MAX_VALID_COST:
            pprint("metric is invalid")
            sys.exit()
        if port in neighbor_ports:
            pprint(f"The peer is listening on port number {port} and can not appear in input ports")
            sys.exit()


def read_config(file_name: str):
    """reads the content of router configuration files and checks"""
    config = {}
    with open(file_name, "r", encoding='utf-8') as f:
        for line in f:
            if line != "\n":
                line = line.strip("\n").split()
                key, remainder = line[0], line[1:]
                match key:
                    case "router-id":
                        router = validate_router_id(int(remainder[0]))
                        config[key] = router
                        assert len(line) == 2, "line must have length 2"

                    case "input-ports":
                        input_ports = validate_input_ports(
                            [int(port_num.strip(",")) for port_num in remainder]
                        )
                        config[key] = input_ports
                    case "outputs":
                        outputs = [_.replace(",","") for _ in remainder]
                        validate_output(
                            outputs, input_ports
                        )
                        for i, output in enumerate(outputs):
                            output_dict = dict()
                            output_dict["next_hop"], output_dict["metric"], output_dict["dest_id"] = [int(_) for _ in output.split("-")]
                            outputs[i] = output_dict
                        config[key] = outputs

    if len(config) < 3:
        pprint("config file must have router-id, input-ports, and outputs")
        sys.exit()
    return config
```

utils.py

```python
import socket
import sys
import config


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


def rip_header(source_router_id: int):
    """Creates the 4 byte header
    command: 2
        See page 6 of Sec 4.2 in rip-assignment.pdf which says
        Do not implement request messages, you should use only the periodic and triggered
        updates.
    """
    command = 2  # as these are responses and not request is implemented
    version = 2
    command = command.to_bytes(1, byteorder="big")
    version = version.to_bytes(1, byteorder="big")
    zeros = source_router_id.to_bytes(2, byteorder="big")
    return command + version + zeros


# def create_rip_packet(next_hop, metric, dest):
def create_rip_packet(dest_id:int, source_router_id: int, routing_table:dict):
    """Create a rip packet"""
    header = rip_header(source_router_id)
    body = bytearray()
    for dest_router, info in routing_table.items():
        next_router, metric = info["next_router"], info["metric"]
        #prevent loop
        if next_router == dest_id:
            metric = 16

        packet = rip_entry(metric, dest_router)
        body += packet
    return header + body


def process_rip_packet(packet: bytearray):
    """A server socket data is processed"""
    header = packet[0:4]
    body = packet[4:]
    len_body = len(body)
    command = int.from_bytes(header[0:1], byteorder="big")
    version = int.from_bytes(header[1:2], byteorder="big")
    source_router_id = int.from_bytes(header[2:4], byteorder="big")
    if command != 2 or version != 2:
        return []
    assert len_body % 20 == 0
    outputs = []
    for _ in range(len_body // 20):
        dic = {}
        entry_body = body[0:20]
        body = body[20:]
        dic["dest_id"] = int.from_bytes(entry_body[4:8], byteorder="big")
        dic["metric"] = int.from_bytes(entry_body[16:20], byteorder="big")
        outputs.append(dic)

    return source_router_id, outputs


def create_server_socket(port):
    """creates and binds sockets
    port are given by config file
    AF_INET refers to the address-family ipv4.
    SOCK_DGRAM refers to connectionless UDP protocols.
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((config.HOST, port))
        print("Socket created on port " + str(port))
        return s
    except socket.error:
        print("Error occurred. Unable to create socket")
        sys.exit()


def create_client_socket():
    """Creates a UDP socket to be used as client"""
    return socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

dv.py

```python
import random
import numpy as np
import select
import utils
import config as configuration
from time import time, sleep
from pprint import pprint
TIMER = 30/30
TIMEOUT = 180/30
GARBAGE_TIMER = (120 +180)/30

class DistanceVector:
    """Distance Vector with Poisoned reverse"""

    def __init__(self, config) -> None:
        self.config = config
        self.router_id = config["router-id"]
        self.output = config["outputs"]
        self.server_sockets = self._create_server_sockets(self.config["input-ports"])
        self.routing_table = {}
        self.intialize_routing_table()
        self.send_to_neighbors()
        self.garbage = []

    def intialize_routing_table(self):
        self.routing_table.update({
            self.router_id: {
                "next_router": "",
                "metric": 0,
                "reachable": True,  # Always True for myself
                "timer": ""
            }
        })
        print(self.print_routing_table())

    def print_routing_table(self):
        """ Prints the forwarding table """
        print("----------------------------------------------------------------")
        print("FORWARDING TABLE OF ROUTER = " + str(self.router_id))
        print("  {:^9} | {:^9} | {:^9} | {:^9}  | {:^9} ".format("Router", "Next-hop", "Metric", "Timer", "Reachable"))
        for router, info in sorted(self.routing_table.items()):
                if router != self.router_id:
                    print("  {:^9} | {:^9} | {:^9} | {:^9.2f}  | {} ".format(router, info["next_router"], info["metric"], info["timer"], info["reachable"]))
        print("----------------------------------------------------------------")
        print('\n')

    def _create_server_sockets(self, input_ports: list):
        """
        configs["input-ports"] = [10012, 10013]
        """
        server_sockets = []
        for port in input_ports:
            soc = utils.create_server_socket(port=port)
            server_sockets.append(soc)
        return server_sockets

    def get_dx_vector(self):
        """Return list of outputs"""
        return self.config["outputs"]
    def parse_output_to_dict(self):
        my_output =  {}
        for output_dict in self.output:
            next_hop = output_dict["next_hop"]
            metric = output_dict["metric"]
            dest_id = output_dict["dest_id"]
            my_output.update({
                dest_id: {
                    "next_hop": next_hop,
                    "metric": metric
                }
            })
        return my_output

    def send_to_neighbors(self):
        """Send distance vector to neighbors
        Example:
        configs["outputs"][0], {"dest_id": 2, "metric": 2, "next_hop": 10021}
        """
        sock = utils.create_client_socket()
        dx_vector = self.get_dx_vector()
        for dx_y in dx_vector:
            port, metric, dest_id = dx_y["next_hop"], dx_y["metric"], dx_y["dest_id"]
            message = utils.create_rip_packet(dest_id, self.router_id, self.routing_table)
            sock.sendto(message, (configuration.HOST, port))

    def periodic_event(self):
        self.send_to_neighbors()

    def handle_upcoming_packet(self, sender_router, packets):
        """1. If there is an existing route, compare the next hop address to the address of the router from which the datagram came.
            - If this datagram is from the same router as the existing route, reinitialize the timeout.
            - Next, compare the metrics.
            a:if the new metric is lower than the old one; do the following actions:
            - Adopt the route from the datagram (i.e., put the new metric in and adjust the next hop address, if necessary).
            - Set the route change flag and signal the output process to trigger an update
            - If the new metric is infinity, start the deletion process otherwise, re-initialize the timeout
            b: If the new metric is the same as the old one, it is simplest to do nothing further (beyond re-initializing the timeout) but heuristic is optional
        2. check to see whether there is already an explicit route for the destination address.
            If there is no such route, add this route to the routing table, unless the metric is infinity (there is no point in adding a route which is unusable)
            Setting the destination address to the destination address in the RTE
        - Setting the metric to the newly calculated metric (as described above)
        - Set the next hop address to be the address of the router from which the datagram came
        - Initialize the timeout for the route.  If the garbage-collection timer is running for this route, stop it
        - Set the route change flag
        - Signal the output process to trigger an update
        """
        my_output = self.parse_output_to_dict()
        for packet in packets:
            dest_id = packet["dest_id"]
            metric = min(packet["metric"] + my_output[sender_router]["metric"], 16)
            #existing path
            if dest_id in self.routing_table:
                if metric < 16:
                    if sender_router == self.routing_table[dest_id]["next_router"] and metric == self.routing_table[dest_id]["metric"]:
                        self.routing_table[dest_id]["timer"] = 0.0
                    if metric < self.routing_table[dest_id]["metric"]:
                        self.routing_table[dest_id]["next_router"] = sender_router
                        self.routing_table[dest_id]["metric"] = metric
                        self.routing_table[dest_id]["reachable"] = True
                        self.routing_table[dest_id]["timer"] = 0.0
                else:
                    #metric can be greater than 16 poisened reverse. if metric is greater than 16 and reachable is false then deletion process begins
                    #by garbage timeout we are going to delete the router
                    #
                    if sender_router == self.routing_table[dest_id]["next_router"]:
                        self.routing_table[dest_id]["metric"] = 16
                        self.routing_table[dest_id]["reachable"] = False
            #new path
            else:
                if metric < 16:
                    self.routing_table.update({
                        packet["dest_id"]: {
                            "next_router": sender_router,
                            "metric": metric,
                            "reachable": True,
                            "timer": 0.0
                        }
                    })

    def timer(self):
        """The 30-second updates are triggered by a clock whose rate is not
        affected by system load or the time required to service the
        previous update timer.

        - 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"""
        time_before_periodic_event = time()
        periodic_event_timer = TIMER
        sleep(periodic_event_timer)
        # increment the routers's timers every 30 seconds. timer might be reinitialized if new event occured. or we increase until timeout occur and we start
        # deleting the route from our forwarding table

        for dest_id, info in self.routing_table.items():
            time_after_event = time()
            if dest_id != self.router_id:
                info["timer"] += time_after_event - time_before_periodic_event

                if info["timer"] >= GARBAGE_TIMER:
                    self.garbage.append(dest_id)
                elif info["timer"] >= TIMEOUT:
                    self.triggered_update(dest_id, info)
                elif info["metric"] == 16 and info["reachable"] == False:
                    self.garbage.append(dest_id)
        self.garbage_collection()


    def triggered_update(self, dest_id, info):
        info["metric"] = 16
        info["reachable"] = False
        #triggered update
        self.send_to_neighbors()

    def garbage_collection(self):
        if len(self.garbage) > 0:
            for garbage_router in self.garbage:
                self.routing_table.pop(garbage_router)
        self.garbage = []

    def wait_for_packets(self):
        """
        rlist, _, _ = select.select(self.server_sockets,[],[], timeout in sec)

        Using Dx(y) = min_v{c(x,v) + Dv(y)}, here v are neighbors of x
        get_dx_vector returns the output/forwarding table
        """
        rlist, _, _ = select.select(self.server_sockets, [], [], 2)
        for sock in rlist:
            dx_vector = self.get_dx_vector()
            data, (_, sender_port) = sock.recvfrom(1024)
            if len(data) > 0:
                sender_router, packet = utils.process_rip_packet(packet=data)
                self.handle_upcoming_packet(sender_router, packet)
```

config_test.py

```python
from pprint import pprint
import sys
import unittest

# from rip import config

import config


class TestConfig(unittest.TestCase):
    def setUp(self) -> None:
        return super().setUp()

    def test_read_config(self):
        """test read_config correctly parses config file"""
        fname = "./data/configTest1.txt"
        configs = config.read_config(file_name=fname)

        # router-id 1
        # input-ports 10012, 10013
        # outputs 10021-2-2, 10031-7-3
        self.assertEqual(configs["router-id"], 1)
        self.assertListEqual(configs["input-ports"], [10012, 10013])
        self.assertDictEqual(
            configs["outputs"][0], {"dest_id": 2, "metric": 2, "next_hop": 10021}
        )
        self.assertDictEqual(
            configs["outputs"][1], {"dest_id": 3, "metric": 7, "next_hop": 10031}
        )

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

dv_test.py

```python
from pprint import pprint
import sys
import os
import unittest
import config
import dv


class TestDV(unittest.TestCase):
    """Distance-Vector (DV) Algorithm as described on page 373 in
    Computer Networking A Top-Down Approach 6th Edition by
    James F. Kurose and Keith W. Ross
    """
    def setUp(self) -> None:
        """
        python server.py 10031
        python server.py 10021
        """
        fname = "./data/configTest1.txt"
        self.config_data = config.read_config(file_name=fname)
        self.distance_vector = dv.DistanceVector(config=self.config_data)
        return super().setUp()

    def test_initialization(self):
        """Test routing table, server sockets, source_router_id, and garbage are correctly initialized. """
        router_id = self.config_data["router-id"]
        self.assertDictEqual(self.distance_vector.routing_table,{
            router_id: {
                "next_router": "",
                "metric": 0,
                "reachable": True,  # Always True for myself
                "timer": ""
            }
        } )
        self.assertListEqual(self.distance_vector.garbage, [])
        self.assertEqual(self.distance_vector.server_sockets[0].getsockname(), ('127.0.0.1', 10012))
        self.assertEqual(self.distance_vector.server_sockets[1].getsockname(), ('127.0.0.1', 10013))
        self.assertEqual(self.distance_vector.router_id, self.config_data["router-id"])

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

utils_test.py

```python
from pprint import pprint
import unittest

import utils


class TestConfig(unittest.TestCase):
    def setUp(self) -> None:
        self.source_id = 1
        self.next_hop = 2
        self.metric = 3
        self.dest = 7
        self.routing_table = {
            self.dest: {
                "next_router": self.next_hop,
                "metric": self.metric,
                "reachable": True,  # Always True for myself
                "timer": 2
            }
        }

        self.rip_packet = utils.create_rip_packet(dest_id=self.dest, source_router_id=self.source_id, routing_table=self.routing_table)
        return super().setUp()

    def test_process_rip_packet(self):
        """test process_rip_packet correctly parses rip packet"""
        source_router_id, outputs = utils.process_rip_packet(self.rip_packet)
        self.assertEqual(source_router_id, self.source_id)
        self.assertListEqual(outputs, [{'dest_id': self.dest, 'metric': self.metric}])


    def test_create_server_socket(self):
        """test create_server_socket creates a UDP socket with correct port"""
        port_num = 1234
        soc = utils.create_server_socket(port=port_num)
        self.assertEqual(soc.type.name, "SOCK_DGRAM")
        self.assertEqual(soc.family.name, "AF_INET")
        self.assertTupleEqual(soc.getsockname(), ("127.0.0.1", port_num))
        soc.close()


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