COSC364 Assignment

RIP Routing

Dylan White

29861039

April 24, 2019

Contents

1 Questions

1.1 Contributions

Dylan White completed 100% of the work in this assignment.

1.2 Good Aspects

I believe the file parser is extremely thorough with validity checking of router configuration files.

Breaking the problem down into elements allowed for an object orientated style of coding to be used with classes for each aspect of the protocol. This gave good encapsulation and loose coupling between modules. It was much easier to write a lot of small methods in each class which could then be chained together to make larger processes.

The decision to use JSON for the configuration files complemented the use of Python dictionaries as the structure of the two is very similar. This ethos of using dictionaries for the entire project is also reflected in using Pickle to serialize the data being communicated between the routers so that all elements were Python objects.

Spawning a new thread for every received packet distributes the system load and allowed for multiple packets to be processed at the same time. Using a mutex to lock the table to stop multiple updates trying to occur concurrently also made for a more robust solution.

1.3 Bad Aspects

The most noticeable bad aspect is the amount of nested for loops and nested if statements in the method used to process incoming packets. The way it is currently written has at least $O(n^2)$ time complexity. There are trivial ways to reduce this but due to time constraints the final submission is left in a non optimal form - it should be noted that this did not seem to cause any performance issues for the networks used in testing but, would likely have scalability issues if larger network topologies were used.

The code could potentially have been modularised further with a timer class but, the implementation works for the assignment purposes.

Some of the naming conventions used in the for loops iterating over dictionaries could have more meaningful key and value names for better readability.

Occasionally the program executes in an erroneous way and has to be restarted. This is likely due to hitting a specific combination of branch logic that could not be identified - this is another motivation to rewrite the incoming packet method.

The header and payload could be revised. Currently the header and payload are both dictionaries that update a "packet" dictionary to be serialized before sending. This results in all data jammed into a dictionary that then has to be parsed. A better solution would be to have a dictionary or list with separate header and payload elements.

I do not believe the updates are working exactly as in the RIP specification. An example is when a router becomes unavailable and before it is updated as unreachable and the garbage timer is started in the route table, the router receives a packet with another route to the expired destination. This will then update the next hop and reset the garbage timer to zero, but now on the next update, the initial expiration has propagated through the network and the garbage timer restarts with the new next hop. Overall this only causes a few seconds of advertising the route. This may be due to a bad implementation of split-horizon with reverse poison.

1.4 Atomicity

Atomicity was ensured by using triggering events in a round robin scheduler in the main while loop. The next process would not execute until the other had completed. Incoming packets read on the select() call would spawn a new thread to process the incoming packet. The route table for each router was also protected with a mutex so that no two threads could attempt to modify the route table concurrently.

2 Testing

Small unit testing was completed as the code was being developed and most of this was not documented as most testing was print statements to see at which point the program was breaking or getting stuck at. Testing that failed has not been documented as it should be fixed in the final submission. No test suites were written to automate testing and assert test results, instead all tests were confirmed manually.

2.1 Configuration File Testing

Test	Expected Outcome	Pass/Fail
Bad JSON	Print error and stop	PASS
Missing Router ID	Print error and stop	PASS
Missing inputs	Print error and stop	PASS
Missing outputs	Print error and stop	PASS
Out of range inputs	Print error and stop	PASS
Out of range outputs	Print error and stop	PASS
Input ports found in outputs	Print error and stop	PASS
Output ports found in inputs	Print error and stop	PASS
Multiple occurrence same input port	Print error and stop	PASS
Multiple occurrence same output port	Print error and stop	PASS
Non-numerical Router ID	Print error and stop	PASS
Non-numerical inputs	Print error and stop	PASS
Non-numerical outputs	Print error and stop	PASS

Table 1: Tests and outcomes for JSON configuration files.

Note: All printed errors are unique (not a generic error message) and will identify the configuration file that caused the error and where possible identify which parameter caused the error.

2.2 Program Testing

All testing used the example network given in the assignment specification unless stated otherwise.

Once the configuration files were known to be robust and working the next test was checking if each router populated the initial routing table correctly with the data in the configuration file. This was inspected visually and was correct for all routing tables. Initially there were issues with the next hop and cost values swapped due to an indexing error. The expected outcome was that the printed route table was consistent with the data in the configuration file. It was observed that it was consistent. This purpose of this test was to ensure initial information was correct as if it was wrong at this stage, the program would not converge correctly.

Once asserted that route tables were being correctly populated from configuration files the socket communication between routers was tested. Initially the program was written to just send periodic updates and Netcat was used to observe the incoming packets on the appropriate sockets. It was expected to see the packet in the terminal running Netcat and this was observed. The purpose of this test was to ensure that the correct packets were being sent to the correct peer routers.

This test was then extended into checking that the incoming packet was correctly parsed and any new found routes were added to the table.

Test: Turn on Router 1 and Router 2.

Expected: One the first periodic update, the route table of Router 1 is updated with any new routes learnt from Router 2 and the route table of Router 2 is updated with any new routes learn from Router 1.

Observed: Correct behaviour.

Once new routes were being added successfully, the incoming cost metric then had to be updated and compared to the existing metric. The route table would then need to update the metric or; update the metric and the next hop router. This test was to ensure that the lowest cost routes were converged to.

Test: Turn on Router 1 and Router 2.
Expected: One the first periodic update, the route table of Router 1 is updated with any new routes learnt from Router 2 with correct metrics and the route table of Router 2 is updated with any new routes learn from Router 1 with correct metrics.
Observed: Correct behaviour.

The next test is to assert that the next hop router is updated if a lower cost path is found.

Test: Turn on Routers 1, 4, 7 and wait to converge, then turn on Routers 2 and 3.
Expected: Initially Router 4 will use a next hop of Router 7 to get to Router 1 with a cost of 14. Once Routers 2 and 3 are turned on and the update has propagated through the network, The route table of Router 4 should update the RTE for Router 1 with a new next hop of Router 3 with a new metric of 8.
Observed: Correct behaviour.

Multiple versions of this test were conducted but have not been documented. A set of configuration files were created with all metrics set to 1. This makes it a simple hop count exercise and is easier to visually check the shortest routes between routers.

Testing was carried out to ensure that split-horizon with poison reverse routing was working as expected. To achieve this a set of configuration files were created which represent the network in Figure 1 below.

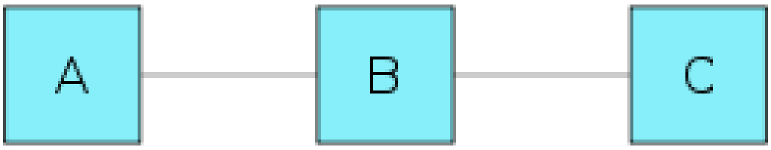

Figure 1: Example network used to test split-horizon with reverse poison.

Test: Turn on Routers A, B and C and then turn off Router C.

Expected: Once Router C expires, Router A will not advertise the route to Router C back to Router B (where it was learnt from).

Observed: Correct behaviour.

Testing was carried out to ensure that bad packets were dropped. To test this, custom configuration files were written that would cause the metric to exceed 16, packets were sent with the wrong version number and packets were sent with a sender ID that was not a peer router. The observed outcomes were the correct behaviour of dropping the packet, printing a message that the packet was dropped and carrying on.

A set of configuration files were written to represent the network given in the assignment specification, Figure 2 below. This was to test that the daemon was working as intended with a larger network. A simple bash script was written to start all routers. A set of tests relating to this network are listed below.

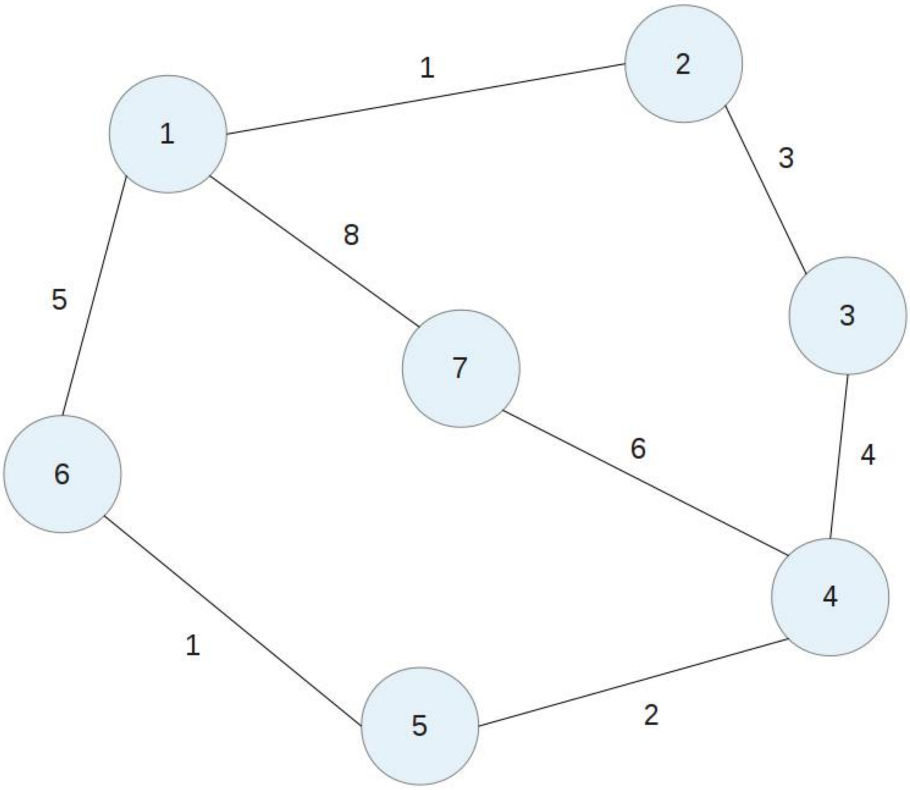

Figure 2: Example network given in assignment specification.

Test: Turn on all 7 routers.

Expected: All route tables converge to shortest cost paths.

Observed: Correct behaviour.

Reason: Prove the route tables are converging to a shortest hop state.

Test: Turn on all 7 routers, let the network converge, turn off Router 1.

Expected: Once each router has not had an update from Router 1 for the specified timeout, mark it as unreachable and start the garbage timer. Once marked as unreachable, a triggered update should be sent (if allowed). The RTE for Router 2 in the route table of Router 7 should update the next hop router to Router 4 and update the metric. All other paths are updated if another past exists through a different router.

Observed: Correct behaviour.

Reason: Prove that garbage timer increments correctly, show triggered updates occur and show that the Router finds a new valid shortest path.

Test: Turn on all 7 routers, let the network converge, turn off Router 1 and wait for garbage-collection timer to expire.
Expected: Once the garbage collection timer has expired for each route table, remove the RTE for Router 1 from the route tables. New shortest paths are found if possible.
Observed: Correct behaviour.
Reason: Check that RTEs are correctly removed from the table once garbage-collected.

Test: Turn on all 7 routers, let the network converge, turn off Router 1 and wait for garbage-collection timer to expire, then turn Router 1 back on.
Expected: The route tables converge to a shortest path state after some activity.
Observed: Correct behaviour.
Reason: Check that the router is correctly added back into the route tables of other routers and the route tables converge to a shortest path state.

Test: Turn on all 7 routers, let the network converge, turn off Router 1, then turn Router 1 back on.
Expected: The garbage and timeout timers are reset in all route tables that have an RTE for Router 1 and the route tables converge to a shortest path state after some activity.
Observed: Correct behaviour.
Reason: Check that the timeout and garbage timers are reset and that the route tables converge back to a shortest path state.

3 Source Code

3.1 main.py

```
#!/usr/bin/python3

from config import *
from router import *
import threading
import sys


def main():
    config = Config(sys.argv[1])
    inputs, outputs, routerId = config.getConfigParams()

    router = Router(inputs, outputs, routerId)
    threading.Timer(1.0, router.run()).start() #start the daemon


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

3.2 config.py

```python
import json
from pprint import pprint
from collections import Counter
import sys


class Config(object):

    def __init__(self, configFile):
        self.configFile = configFile

        self.loadConfig()
        self.checkConfigParams()

    def loadConfig(self):
        try:
            with open(self.configFile) as configParams:
                configParams = json.load(configParams)
        except ValueError as e:
            pprint('Error parsing ' + str(self.configFile) +
                   ". Likely due to malformed JSON.")
            return False
        else:
            self.routerid = configParams['router-id']
            self.inputPorts = configParams['input-ports']
            self.outputs = configParams['outputs']
            return True

    def checkConfigParams(self):
        try:
            # router id checks
            if not self.routerid:
                raise AttributeError(
                    'Missing router ID. Check config file and
                     ↪  restart.')
            if not self.routerid.isdigit():
                raise TypeError(
                    'RouterID Error. Check config file and restart.')

            # input ports checks
            inputPortCount = Counter(self.inputPorts)
            for port, occurences in inputPortCount.items():
                if occurences > 1:
                    raise ValueError(
                        'Multiple occurences of input ports. Check config file and restart')
            for port in self.inputPorts:
                if not str(port).isdigit():
                    raise ValueError(
                        'Input port(s) non-numerical. Check config file and restart.')
                if int(port) < 1024 or int(port) > 64000:
                    raise ValueError(
                        'Input port(s) out of range. Check config file and restart.')

            # output params checks
            outputPorts = []

            if not self.outputs:
                raise AttributeError(
                    'Missing output parameters. Check config file and restart.')
            for output in self.outputs:
                output = output.split('-')
                port = output[0]
                metric = output[1]
                peer = output[2]
                outputPorts.append(int(port))

                if not str(port).isdigit():
                    raise ValueError(
                        'Output port(s) non-numerical. Check config file and restart.')
                if int(port) < 1024 or int(port) > 64000:
                    raise ValueError(
                        'Output port(s) out of range. Check
                         ↪  config file and restart.')
                if not str(metric).isdigit():
                    raise ValueError(
                        'Metric value(s) non-numerical. Check
                         ↪  config file and restart.')
                if not str(peer).isdigit():
                    raise ValueError(
                        'Peer router value(s) non-numerical.
                         ↪  Check config file and restart.')

            outputPortCount = Counter(outputPorts)
            for port, occurences in outputPortCount.items():
                if occurences > 1:
                    raise ValueError(
                        'Multiple occurences of output ports.
                         ↪  Check config file and restart')

            if set(self.inputPorts) & set(outputPorts):
                raise ValueError(
                    'Cannot have same port for input and output.
                     ↪  Check config file and restart.')

        except ValueError as e:
            pprint(str(e))
            sys.exit(0)
            return False
        except AttributeError as e:
            pprint(str(e))
            sys.exit(0)
            return False
        else:
            return True

    def getOutputs(self):
        return self.outputs

    def getInputs(self):
        return self.inputPorts

    def getRouterId(self):
        return self.routerid

    def getConfigParams(self):
        return self.inputPorts, self.outputs, self.routerid
```

3.3 router.py

```python
import json
from socket import *
import random
import time
import select
from pprint import pprint
from threading import *
import threading
import pickle
import copy
import table
from entry import *
from packet import *


HOST = '127.0.0.1'
INFINITY = 16
PERIOD = 30
TIMEOUT = PERIOD * 6
GARBAGE = PERIOD * 8


class Router(object):

    def __init__(self, inputs, outputs, routerId):
        self.sockets = []
        self.inputPorts = inputs
        self.outputs = outputs
        self.routerId = routerId
        self.routeTable = table.RouteTable()
        self.time = 0
        self.randTime = random.randint(PERIOD - 5, PERIOD + 5)
        self.triggeredUpdateTimer = 0
        self.triggeredUpdatePeriod = random.randint(1, 5)
        self.peerRouters = []
        self.tableMutex = Lock()

        self.createSockets()
        self.createInitTable()
        self.getPeerRouters()

    def createSockets(self):
        for port in self.inputPorts:
            try:
                s = socket(AF_INET, SOCK_DGRAM)
                s.bind((HOST, int(port)))
                self.sockets.append(s)
                pprint('Opened socket on port: ' + str(port))
            except:
                pprint('Could not open socket on port: ' +
                ↪   str(port))

    def getPeerRouters(self):
        for output in self.outputs:
            output = output.split('-')
            self.peerRouters.append(output[2])

    def createInitTable(self):
        for output in self.outputs:
            output = output.split('-')
            entry = Entry(output[0], output[2], output[1],
                          self.routerId, output[2])
            entry = entry.createEntry()
            self.routeTable.addEntry(entry)

    def addFoundPeer(self, sourceId):
        for output in self.outputs:
            output = output.split('-')
            if output[2] == sourceId:
                entry = Entry(output[0], output[2],
                              output[1], self.routerId,
                              ↪   output[2])
                entry = entry.createEntry()
                self.routeTable.addEntry(entry)

    def updateGlobalTimers(self):
        self.time += 1
        self.triggeredUpdateTimer += 1

    def updateTimeouts(self):
        # list makes a copy of the keys to avoid RuntimeError
        ↪   (dictionary changing size)
        for k in list(self.routeTable.getTable().keys()):
            garbageUpdated = False
            self.routeTable.updateTimeoutTimer(k)

            if self.routeTable.getRouteMetric(k) == INFINITY:
                self.routeTable.updateGarbageTimer(k)
                garbageUpdated = True

            if self.routeTable.getTimeoutTimer(k) >= TIMEOUT:
                oldMetric = self.routeTable.getRouteMetric(k)

                if oldMetric != INFINITY:
                    self.routeTable.updateRouteMetric(k,
                    ↪   INFINITY)
                    self.routeTable.setRouteChangedFlag(k)

                if not garbageUpdated:
                    self.routeTable.updateGarbageTimer(k)

                if self.routeTable.getGarbageTimer(k) > GARBAGE:
                    self.routeTable.getTable().pop(k)

    def checkUpdateTimers(self):
        if self.time >= self.randTime:
            self.periodicUpdate()
            self.time = 0
            self.randTime = random.randint(PERIOD - 2, PERIOD +
            ↪   2)

        # prevents sending triggered update if periodic is
        ↪   scheduled sooner
        if self.randTime - self.time < self.triggeredUpdateTimer
        ↪   - self.triggeredUpdatePeriod:
            self.triggeredUpdateTimer = 0

        if self.checkForTriggeredUpdates():
            if self.triggeredUpdateTimer >=
            ↪   self.triggeredUpdatePeriod:
                self.triggeredUpdate()
                self.routeTable.resetRouteChangedFlag()
                self.triggeredUpdateTimer = 0
                self.triggeredUpdatePeriod = random.randint(1, 5)

    def printTable(self):
        self.routeTable.printFormattedTable(self.routerId)

    def periodicUpdate(self):
        for output in self.outputs:
            output = output.split('-')
            port = output[0]
            peer = output[2]

            packet = Packet(2, self.routerId, self.routeTable,
                            peer)  # 2 == response (not required)
            header = packet.makeHeader()
            payload = packet.makePayload()
            packetToSend = packet.makePacket(header, payload)
            packet.sendPacket(packetToSend, port, HOST,
            ↪   self.sockets)

    def triggeredUpdate(self):
        for output in self.outputs:
            output = output.split('-')
            port = output[0]
            peer = output[2]

            packet = Packet(2, self.routerId, self.routeTable,
                            peer)  # 2 == response (not required)
            header = packet.makeHeader()
            payload = packet.makeTriggerUpdatePayload()
            packetToSend = packet.makePacket(header, payload)
            packet.sendPacket(packetToSend, port, HOST,
            ↪   self.sockets)
            pprint('--- Sent Triggered Update to: ' + peer + '
            ↪   ---')

    def recv(self):
        readable, _, _ = select.select(self.sockets, [], [], 0)
        for s in readable:
            packet, _ = s.recvfrom(4096)
            loadedPacket = pickle.loads(packet)

            ↪   threading.Thread(target=self.processIncomingPacket(loadedPacket)).start()

    def validateIncomingPacket(self, packet):
        validPacket = True  # assume the packet is valid from the
        ↪   start

        testPacket = copy.deepcopy(packet)

        if testPacket['version'] != 2:
            validPacket = False

        if testPacket['command'] != 2:
            validPacket = False

        if testPacket['routerId'] not in self.peerRouters:
            validPacket = False

        testPacket.pop('version')
        testPacket.pop('command')
        testPacket.pop('routerId')

        for k, v in testPacket.items():
            if int(v['metric']) > INFINITY or int(v['metric']) <
            ↪   1:
                validPacket = False

        if validPacket:
            return packet
        else:
            pprint('Invalid packet. Dropping...')
            return None

    def processIncomingPacket(self, packet):
        packet = self.validateIncomingPacket(packet)

        if packet is None: # Can be None if fails validity check
            return

        sourceId = packet['routerId'] #get the relavent header
        ↪   info then pop it
        packet.pop('version')
        packet.pop('command')
        packet.pop('routerId')

        self.tableMutex.acquire()

        try:
            # reestablishes a to a peer router if timed out
            if sourceId not in self.routeTable.getTable().keys():
                for output in self.outputs:
                    output = output.split('-')
                    if sourceId == output[2]:
                        self.addFoundPeer(sourceId)

            # for all RTEs in the incoming packet...
            for incomingDest, incomingValue in
            ↪   list(packet.items()):

                # for all entries in the current route table...
                for dest, values in
                ↪   self.routeTable.getTable().items():
                    # the next hop for this dest has sent a
                    ↪   packet so reset timers
                    if self.routeTable.getNextHop(dest) ==
                    ↪   sourceId and
                    ↪   self.routeTable.getRouteMetric(dest) !=
                    ↪   INFINITY:
                        self.routeTable.resetTimeoutTimer(dest)
                        self.routeTable.resetGargbageTimer(dest)

                # checking if the dest already exists in table.
                ↪   (Don't add itself to own route table)
                if incomingDest in
                ↪   self.routeTable.getTable().keys() and
                ↪   incomingDest != self.routerId:
                    newMetric = min(int(
                        incomingValue['metric']) +
                        ↪   int(self.routeTable.getRouteMetric(sourceId)),
                        ↪   INFINITY)
```

```python

# if a peer router has timed out and reconnects, reinit the table with config file details
if dest == sourceId and self.routeTable.getGarbageTimer(dest) > 0:
    self.addFoundPeer(sourceId)

for dest, values in self.routeTable.getTable().items():
    if incomingDest == dest:
        # if the packet is from the current next hop, update the metric
        if sourceId == self.routeTable.getNextHop(dest):
            oldMetric = self.routeTable.getRouteMetric(dest)
            self.routeTable.updateRouteMetric(dest, newMetric)
            # if the route has been marked unreachable, set the flag for triggered update
            if newMetric == INFINITY and oldMetric != INFINITY:
                self.routeTable.setRouteChangedFlag(dest)
        else:
            # if the packet is from a different next hop router, only change the next hop if the
            if newMetric < int(self.routeTable.getRouteMetric(dest)):
                # cost is lower.
                self.routeTable.updateNextHop(dest, sourceId)
                self.routeTable.updateRouteMetric(
                                        dest, newMetric)

                                     ↪  self.routeTable.resetTimeoutTimer(dest)

                # not in the table, add a new entry
                elif incomingDest not in
                 ↪  self.routeTable.getTable().keys() and
                 ↪  int(incomingValue['metric']) < INFINITY and
                 ↪  incomingDest != self.routerId:
                    newMetric = min(int(
                        incomingValue['metric']) +
                         ↪  int(self.routeTable.getRouteMetric(sourceId)),
                         ↪  INFINITY)
                    entry = Entry(
                        incomingValue['port'], incomingDest,
                         ↪  newMetric, sourceId, sourceId)
                    entry = entry.createEntry()
                    self.routeTable.addEntry(entry)

                     ↪  self.routeTable.resetTimeoutTimer(incomingDest)

                     ↪  self.routeTable.resetGargbageTimer(incomingDest)

        finally:
            self.tableMutex.release()

    def checkForTriggeredUpdates(self):
        updatesToSend = False
        for k, v in self.routeTable.getTable().items():
            if v['routeChanged'] == True:
                updatesToSend = True
        return updatesToSend

    def run(self):
        while(1):
            time.sleep(1)
            self.updateGlobalTimers()
            self.updateTimeouts()
            self.checkUpdateTimers()
            self.recv()
            self.printTable()
```

3.4 table.py

```python
from pprint import pprint

WIDTH = 13
INFINITY = 16

LINE = ('+' + ''.center(WIDTH, '-') +
        '+' + ''.center(WIDTH, '-') +
        '+' + ''.center(WIDTH, '-') +
        '+' + ''.center(WIDTH, '-') +
        '+' + ''.center(WIDTH, '-') +
        '+' + ''.center(WIDTH, '-') +
        '+')

class RouteTable(object):

    def __init__(self):
        self.table = {}

    def addEntry(self, entry):
        self.table.update(entry)

    def removeEntry(self, dest):
        self.table.pop(dest)

    def getEntry(self, dest):
        return self.table[dest]

    def getTable(self):
        return self.table

    def getRouteMetric(self, dest):
        return self.table[dest]['metric']

    def getTimeoutTimer(self, dest):
        return self.table[dest]['timeout']

    def getGarbageTimer(self, dest):
        return self.table[dest]['garbage']

    def getNextHop(self, dest):
        return self.table[dest]['nextHop']

    def updateRouteMetric(self, dest, metric):
        self.table[dest]['metric'] = metric
        return metric

    def updateNextHop(self, dest, nextHop):
        self.table[dest]['nextHop'] = nextHop

    def updateTimeoutTimer(self, dest):
        timeoutTimer = self.getTimeoutTimer(dest)
        newTime = timeoutTimer + 1
        timeout = {'timeout': newTime}
        self.table[dest].update(timeout)

    def updateGarbageTimer(self, dest):
        garbageTimer = self.getGarbageTimer(dest)
        newTime = garbageTimer + 1
        garbage = {'garbage': newTime}
        self.table[dest].update(garbage)

    def setRouteChangedFlag(self, dest):
        self.table[dest]['routeChanged'] = True

    def resetTimeoutTimer(self, dest):
        self.table[dest]['timeout'] = 0

    def resetGargbageTimer(self, dest):
        self.table[dest]['garbage'] = 0

    def resetRouteChangedFlag(self):
        for k, v in self.table.items():
            if v['routeChanged'] == True:
                v['routeChanged'] = False

    def printTable(self):
        pprint(self.table)

    def printFormattedTable(self, routerId):
        print('Table of Router : {}'.format(routerId).center(80, ' '))
        print(LINE)
        print('|' + 'Dest'.center(WIDTH, ' ') +
              '|' + 'Cost'.center(WIDTH, ' ') +
              '|' + 'Next Hop'.center(WIDTH, ' ') +
              '|' + 'Timeout'.center(WIDTH, ' ') +
              '|' + 'Garbage'.center(WIDTH, ' ') +
              '|' + 'Route Changed'.center(WIDTH, ' ') +
              '|')
        print(LINE)
        for k, v in sorted(self.table.items()):
            print('|' + str(k).center(WIDTH, ' ') +
                  '|' + str(v['metric']).center(WIDTH, ' ') +
                  '|' + str(v['nextHop']).center(WIDTH, ' ') +
                  '|' + str(v['timeout']).center(WIDTH, ' ') +
                  '|' + str(v['garbage']).center(WIDTH, ' ') +
                  '|' + str(v['routeChanged']).center(WIDTH, ' ') +
                  '|')
        print(LINE)
        print(' ')
```

3.5 entry.py

```
from pprint import pprint


class Entry(object):

    def __init__(self, nextPort, dest, metric, parent, nextHop):
        self.nextPort = nextPort
        self.dest = dest
        self.metric = metric
        self.parent = parent
        self.nextHop = nextHop
        self.timeoutTimer = 0
        self.garbageTimer = 0
        self.routeChanged = False
        self.entry = {}

    def createEntry(self):
        self.entry[self.dest] = {
            'metric': self.metric,
            'nextHop': self.nextHop,
            'parent': self.parent,
            'port': self.nextPort,
            'timeout': self.timeoutTimer,
            'garbage': self.garbageTimer,
            'routeChanged': self.routeChanged,
        }
        return self.entry.items()

    def printEntry(self):
        pprint(self.entry)
```

3.6 packet.py

```
import copy
import pickle
from pprint import pprint


INFINITY = 16


class Packet(object):

    def __init__(self, command, routerId, payload, peer):
        self.version = 2  # always response only (no request for
        ↪   this assignment)
        self.command = command
        self.routerId = routerId
        self.payload = payload
        self.peer = peer

    def makeHeader(self):
        return {'version': self.version, 'command': self.command,
        ↪   'routerId': self.routerId}

    def makePayload(self):
        tableCopy = copy.deepcopy(self.payload.getTable())

        for dest, v in tableCopy.items():
            if self.peer == v['nextHop']:  # split-horizon w/
            ↪   reverse poison
                v['metric'] = INFINITY

        return tableCopy

    def makeTriggerUpdatePayload(self):
        tableCopy = copy.deepcopy(self.payload.getTable())

        for k, v in list(tableCopy.items()):
            if v['routeChanged'] == False:
                tableCopy.pop(k)
            if self.peer == v['nextHop']:
                v['metric'] == INFINITY

        return tableCopy

    def makePacket(self, header, payload):
        packet = {}
        packet.update(header)
        packet.update(payload)
        picklePacket = pickle.dumps(packet)
        return picklePacket

    def sendPacket(self, packet, port, host, sockets):
        sockets[0].sendto(packet, (host, int(port)))
```

4 Configuration Files

```
{
        "router-id" : "1",
        "input-ports" : [10013, 10014, 10003],
        "outputs" : ["10000-1-2", "10001-8-7", "10002-5-6"]
}

{
        "router-id" : "2",
        "input-ports" : [10000, 10005],
        "outputs" : ["10003-1-1", "10004-3-3"]
}

{
        "router-id" : "3",
        "input-ports" : [10004, 10007],
        "outputs" : ["10005-3-2", "10006-4-4"]
}

{
        "router-id" : "4",
        "input-ports" : [10006, 10015, 10010],
        "outputs" : ["10007-4-3", "10008-6-7", "10009-2-5"]
}

{
        "router-id" : "5",
        "input-ports" : [10009, 10012],
        "outputs" : ["10010-2-4", "10011-1-6"]
}
```

```
{
        "router-id" : "6",
        "input-ports" : [10002, 10011],
        "outputs" : ["10012-1-5", "10013-5-1"]
}

{
        "router-id" : "7",
        "input-ports" : [10001, 10008],
        "outputs" : ["10014-8-1", "10015-6-4"]
}
```