19/04/23

19/04/23

COSC364 RIP Assignment

Tim Lindbom (tli89) 46808065

& Ben Ireland (bir16) 93524067

Contributions:

Ben 60%

• Bellman-Ford algorithm: BellmanFordAlgorithm.py is used throughout the project to compute the

shortest path to each router, add new routers to a routing table, and update routing tables if any changes need to be made.

• Response handler: The ResponseHandler.py file is used to Generate packets and packet headers, read, and unpack messages received from neighbours, and send responses.

• RIP demon: RIPDemon.py is the main file of the program. This file calls the helper functions and initiates the main infinite loop of the program.

• Timer: Timer.py contains the helper functions for all the timers used in the program, such as the timer

initialiser, timer checks, and the garbage collection timer.

Tim 40%

• Router: the router.py file is where the router class is defined. After the txt file is parsed the output is passed into a Router class constructor.

• Router Configuration: The routerConfig.py file parses the input file in which the router descriptions are

contained and returns a list containing the id, input, output ports, and timers.

• Error Handling: The ErrorHandler.py file is where most of the error check functions for the Id, input and

output ports, timers, and packet generation are held.

Testing:

When we first begin testing one big issue we ran into while testing was how we were going to run the

demon if we had all the routers in one text file and only one instance of the program running. The code we had at that time was expected to parse the single txt file and create a router for each line in the file. Once the routers had been initialised and the ports had been bound, we expected the code to call the select method on each router and send a packet to each router in its output list. However, when we ran this, we got stuck in what seemed to be an infinite loop since we had forgotten to account for the fact that select blocks the CPU until it receives something from a port or reaches a timeout. Because of this, no router ended up sending a packet and the program halted. To combat this issue, we separated the routers into separate txt files and ran multiple instances of the program, which resulted in the routers being able to communicate with each other.

Some of the testing that was done throughout the production of the program can be found as commented out code in the routerTesting.py file. Each of the tests conducted within this file were designed to test the functionality of each module.

These tests included:

* Instantiating a router object and computing the routing algorithm, then printing the parameters of the router checking that the routing table had updated correctly.
* Testing the response handler by manually generating the response packet for a router, sending this to a neighbour router, then reading the response packet to confirm it was being received correctly.
* The functionality of the triggered updates, done by setting the metric of an entry in a routers table to sixteen and receiving the triggered update from another router.
* The timing handler was tested by setting up a router and entering an infinite loop checking and
* altering the values of the timers to check they were responding as expected.

Once these tests, and variations of them, were completed, testing of the main command line program begun. Most of the debugging was completed using a simple three router network connected in a triangular

configuration. This allowed any errors within the demon’s process to be directly identified and remedied. Testing the demon on the example configuration as well as other configurations was then completed,

checking that the converged routing tables matched what was to be expected to ascertain the demon was performing correctly.

Example Configuration:

Each router in the topology is defined in a separate txt file and formatted like

“router\_id 1, inputs 1106 1107 1102, outputs 1201-1-2 1701-8-7 1601-5-6”

(Actual configuration of router1 in the example network shown in figure 3)

Where the router ID is unique to each router, the input port numbers are only used as inputs for one router. The outputs are formatted so that the first number is the outputting router, the next number is the cost to the router, and the third number is the id of the receiving router.

To execute the program in the terminal, the program would be called with the line “python3 RIPDemon.py <filename>” as shown in figure 1. This would have to be called in a new terminal window/tab for each file in the configuration. Once the line has been called the program will first output the router's ID, input, and

outputs defined in the txt file. Next, it will print the routing table of its router, which at this point in the

program will only contain itself. Throughout the lifetime of the program, the routing tables will update after a

predetermined amount of time given in a list in the source code.

A picture containing text

Description automatically generated

Figure 1

Calendar

Description automatically generated

Figure 2

Figure 2 shows that output for router 7 once all the routers have been turned on and the routers a have converged and found the shortest paths to each other router based the topology given in Figure 3.

A picture containing map, different

Description automatically generated

Figure 3

Router.py

#Router class used for RIP protocol

#Tim Lindbom & Ben Ireland

#8/3/23

import socket

import time

class Router:

    def \_\_init\_\_(self, parameters):

        self.id = parameters[0]

        self.inputs = parameters[1]

        self.outputs = parameters[2]

        self.timers = parameters[3]

        self.routingTable = {}

        self.localIP = "127.0.0.1"

        self.sockets = []

        self.routingTable.update({self.id: [0, self.id, 0, [None, None]]})      # Layout follows [Entry ID : Metric, Next-Hop, RouteChangeFlag, [TimeOutTimer, GarbageTimer]]

    def PrintParams(self):

        """used for testing"""

        print( "id: ", self.id,"\n",\

            "inputs: ", self.inputs,"\n",\

            "outputs", self.outputs, "\n",\

            "routing table: ", self.routingTable, "\n", \

            "sockets: ", self.sockets, "\n")

        return

    def OpenSockets(self):

        """iterates over the input list, creates and binds sockets for each port"""

        for port in self.inputs:

            sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

            sock.bind(('', port))

            self.sockets.append(sock)

    def PrintTable(self):

        """Prints a formatted version of the routing table"""

        sysTime = time.time()

        table = f"ROUTER ID: {self.id}\n| DESTINATION | COST | NEXT-HOP | TIMEOUT | GARBAGE COLLECTION |\n"

        for entry, route in self.routingTable.items():

            if route[3][0] == None:

                timeoutTimer = f"{'-':^9}"

            else:

                timeoutTimer = f"{(route[3][0] - sysTime):^9.2f}"

            if route[3][1] == None:

                garbTimer = f"{'-':^20}"

            else:

                garbTimer = f"{(route[3][1] - sysTime):^20.2f}"

            table = table + f"|{entry:^13}|{route[0]:^6}|{route[1]:^10}|{timeoutTimer}|{garbTimer}|\n"

        print(table)

RIPDemon.py

#Main RIP protocol file

#Tim Lindbom & Ben Ireland

#8/3/23

import random

import sys

import time

from select import select

import routerConfig

from BellmanFordAlgorithm import \*

from ResponseHandler import \*

from Router import \*

def GetResponseTime(router):

    """Initialise the response timer for a specified router"""

    responseVal = router.timers[0]

    interval = random.uniform(responseVal \* 0.8, responseVal\*1.2)     # Uses the user specified timer value adding +- randomness as specified in RIP spec

    return interval

def main():

    # Reads router config file and creates a router object

    routerFile = routerConfig.readFile(sys.argv[1])

    routerInfo = routerConfig.getInfo(routerFile)

    router = Router(routerInfo)

    # Opens the  all Sockets

    router.OpenSockets()

    router.PrintTable()

    # Set up response timer

    responsePeriod = GetResponseTime(router)

    sysTime = time.time()

    responseTimer = responsePeriod  + sysTime

    # Enter the main program loop

    while 1:

        # Check if its time to send a response

        sysTime = time.time()

        if sysTime >= responseTimer:

            responsePeriod = GetResponseTime(router)

            responseTimer = responsePeriod  + sysTime

            SendResponses(router)

        else:

            # Wait for response messages to arrive

            printTable = False

            readSockets, writeSockets, null = select(router.sockets,[],[], router.timers[0])

            if len(readSockets) != 0 :

                for soc in readSockets:

                    try:

                        response, addr = soc.recvfrom(504)

                    except ConnectionResetError:

                        continue

                    headerInfo, peerRouterEntries = ReadResponse(response)

                    peerRouterID = headerInfo[2]

                    ComputeRoutingAlgorithm(router, peerRouterID, peerRouterEntries)

                    # Check if there has been a change to the routes

                    for route in router.routingTable.values():

                        routeChangeFlag = route[2]

                        if routeChangeFlag == 1:

                            printTable = True

                    CheckTimers(router)

            if printTable == True:

                router.PrintTable()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

routerConfig.py

#Router configuration file

#Tim Lindbom & Ben Ireland

#8/3/23

import sys

import ErrorHandler

def readFile(name):

    """takes the config file name as a string and

    splits the file by line into a list of strings"""

    file = open(name, 'r')

    config = file.read().splitlines()

    file.close()

    return config

def getInfo(info):

    print(info)

    routerInfo = info[0].split(", ")

    processedInputs = []

    processedOutputs = []

    processedTimers = []

    try:

        #id represents a unique identifier for each router

        id = (routerInfo[0].split(" ")[1])

    except IndexError:

        print("invalid router ID \n Please make sure the router ID configuartion follows the specified configuarion")

        sys.exit()

    try:

        #inputs is a list of ports that a router can receive data from

        inputs = routerInfo[1].split(" ")[1:]

    except IndexError:

        print("ivalid input routers \n Please make sure the router inputs configuartion follows the specified configuarion")

        sys.exit()

    try:

        #outputs is a nested list each list in outputs represents a port that the

        #router can output data. Each inner list is of the form [x, y, z] where

        #x is the port number, y is the weight/distance and z is the recieving router

        outputs = routerInfo[2].split(" ")[1:]

    except IndexError:

        print("Invalid Output Router configuration \n Please make sure the router outputs configuartion follows the specified configuarion")

        sys.exit()

    # Allows for Timers to be set.

    timers = []

    if len(routerInfo) == 4:

        tempTimers = (routerInfo[3].split(" ")[1:])

        for timer in tempTimers:

            timers.append(int(timer))

    else:

        timers = [30, 180, 120]  #[T, 6\*T, 4\*T]

#--------------------------------------------------------

#------------------ERROR-CHECKS--------------------------

#--------------------------------------------------------

    #runs error checks on the id value

    #if all the tests pass, id is converted to an int

    id = ErrorHandler.RouterIdCheck(id)

    #error checks the inputs values. If all tests passes

    #processed inputs is assigned to a list of ints of the port numbers

    processedInputs = ErrorHandler.PortNumberChecks(inputs)

    #processedOutputs conatins all the output port inormation

    #outputPortNumbers is just the port numbers

    processedOutputs, outputPortNumbers = ErrorHandler.RouterOutputCheck(outputs)

    #comapres the input/output port number to confirm that none appear in both lists

    CompareInputsOutputs(processedInputs, outputPortNumbers)

    #makes sure the timer values follow the correct formatting

    processedTimers = ErrorHandler.TimerChecks(timers)

    return [id, processedInputs, processedOutputs, processedTimers]

BellmanFordAlgorithm.py

#Bellman Ford Algorithm for use in a RIP routing protocol

#Tim Lindbom & Benjamin Ireland

#23/2/23

from Router import \*

from Timer import \*

def ComputeRoutingAlgorithm(hostRouter, peerRouterID, peerRouterEntries):

    """Computes RIP routing algorithm and updates the routing table"""

    for entry in peerRouterEntries:

        # Match the Peer Router ID to one of the IDs in the output list to get the cost of the link

        for output in hostRouter.outputs:

            outputRouterID = output[2]

            outputCost = output[1]

            if outputRouterID == peerRouterID:

                linkCost = outputCost

        entryMetric = entry[1]

        metric = min(entryMetric + linkCost, 16)

        # Checks if router is in the table. If not add the router to table. Otherwise we update the route.

        entryID = entry[0]

        if entryID not in hostRouter.routingTable:

            AddNewRoute(hostRouter, peerRouterID, entryID, metric)

        else:

            UpdateRoute(hostRouter, peerRouterID, entryID, metric)

def AddNewRoute(hostRouter, peerRouterID, entryID, metric):

    """Adds a new route to the routing table if the entry does not exist"""

    if metric < 16:

        # Adds the new router to the routing table and initialises the route timeout

        hostRouter.routingTable[entryID] = [metric, peerRouterID, 1, [None, None]]

        InitTimeout(hostRouter, entryID)

def UpdateRoute(hostRouter, peerRouterID, entryID, newMetric):

    """Updates the current route if it has been changed and resets timers."""

    currentNextHopID = hostRouter.routingTable.get(entryID)[1]

    currentMetric = hostRouter.routingTable.get(entryID)[0]

    # Check if the router the update is coming from is the same router as in routing table

    if currentNextHopID == peerRouterID:

        # If the metric is greater infinity we start the Timeout process

        if newMetric >= 16:

            if currentMetric != newMetric:

                Timeout(hostRouter, entryID)

                SendResponses(hostRouter, True)

        elif newMetric != currentMetric:

            hostRouter.routingTable[entryID] = [newMetric, currentNextHopID, 1, [None, None]]

            InitTimeout(hostRouter, entryID)

        elif newMetric == currentMetric:

            hostRouter.routingTable[entryID][2] = 0

            InitTimeout(hostRouter, entryID)

    else:

        if newMetric < currentMetric:

            hostRouter.routingTable[entryID] = [newMetric, peerRouterID, 1, [None, None]]

            InitTimeout(hostRouter, entryID)

ResponseHandler.py

#Generates and processes RIP Response packets

#Tim Lindbom & Benjamin Ireland

#23/2/23

import socket

import sys

import ErrorHandler

import Router

# TEST IMPORTS

# from BellmanFordAlgorithm import \*

def GenerateResponse(router, recieverID, triggered=False):

    """Generates response packet to be sent to other routers"""

     ### Checks if the response is a triggered response send only the invalid routes

    if triggered == True:

        routingTable = GetInvalidRoutes(router)

        if len(routingTable) == 0:

            return None

    else:

        routingTable = router.routingTable

    response = bytearray(4)

    response[0] = 2 # Indicating response message

    response[1] = 2 # Version 2

    response[2] = router.id >> 8

    response[3] = router.id & 0xFF #Router ID

    for entryID, route in routingTable.items():

        learnedID = route[1]

        RTE = bytearray(20)

        RTE[6] = entryID >> 8          # Add Router ID

        RTE[7] = entryID & 0xFF

        if recieverID == learnedID:

            RTE[18] = 16 & 0xFF00       # If the route was learned from the router it sets the metric to INF

            RTE[19] = 16 & 0xFF

        else:

            RTE[18] = route[0] & 0xFF00 # Add Metric to router

            RTE[19] = route[0] & 0xFF

        response =  response + RTE      # Add RTE onto the end of response message

    return response

def SendResponses(router, triggered=False):

    """Used to send a response message to a specified Port"""

    i = 0

    while i < len(router.outputs):   # Enter a loop cycling through the output list and sending each peer router their designated message

        port = router.outputs[i][0]

        recieverID = router.outputs[i][2]

        soc = router.sockets[i]

        response = GenerateResponse(router, recieverID, triggered)

        if response != None:

            soc.sendto(response, (router.localIP, port))

        i += 1

def ReadResponse(response):

    """Used to unpack recieved response message to use in the Bellman Ford algorithm"""

    i = 0

    peerRouterEntries = []

    responseHeader = response[:4]

    entries = response[4:]

    messageType = responseHeader[0]         # Gets the message, version, and router ID

    versionType = responseHeader[1]

    peerRouterID = responseHeader[2] << 8 | response[3]

#---------------------ERROR-CHECKS-----------------------------------

    if messageType != 2:

        print("message type must be '2'")

        sys.exit()

    if versionType != 2:

        print("version type must be '2'")

        sys.exit()

#--------------------------------------------------------------------

    while i < len(entries) - 1:             # Adds each router entry into a list

        peerRouterEntries.append([entries[i + 6] << 8 | entries[i + 7], entries[i + 18] << 8 | entries[i + 19]])

        i += 20

    return [messageType, versionType, peerRouterID], peerRouterEntries

def GetInvalidRoutes(router):

    """When called will iterate over the routing table entries, if the route invalid flag is True then it will add the route to a temporary dictionary"""

    invalidRoutes = {}

    for entryID, route in router.routingTable.items():

        routeChangeFlag = route[2]

        metric = route[0]

        if routeChangeFlag == 1 and metric == 16:

            invalidRoutes[entryID] = route

    return invalidRoutes

Timer.py

# Timer file, handles all timer related events.

#Tim Lindbom & Benjamin Ireland

#23/2/23

import random

import time

from ResponseHandler import \*

from Router import \*

def CheckTimers(router):

    """Used to check route timers, if there is a timeout the grabage collector will be started for that route.

    If the Garbage Collector expires the route is deleted"""

    # Keep track of all routes that need to be deleted

    garbageRoutes = []

    for entry, route in router.routingTable.items():

        systemTime = time.time()

        timeoutTime = route[3][0]

        garbageColTime = route[3][1]

        # If the timeout is reset while the garbage colector is running we clear the garbage timer.

        if timeoutTime != None and garbageColTime != None:

            route[3][1] = None

        # Check if a timeout has occured

        elif timeoutTime != None and garbageColTime == None:

            if systemTime >=  timeoutTime:

                Timeout(router, entry)

                SendResponses(router, True)

        # Track any routes that have expired

        elif garbageColTime != None and timeoutTime == None:

            if systemTime >= garbageColTime:

               garbageRoutes.append(entry)

    # Delete expired routes

    for garbageEntry in garbageRoutes:

        del router.routingTable[garbageEntry]

def InitTimeout(router, entryID):

    """Initialse the timeout timer"""

    timeoutVal = router.timers[1]

    timeoutTime = time.time() + timeoutVal

    router.routingTable[entryID][3][0] = timeoutTime

def Timeout(router, entryID):

    """Initialises the garbage collector timer and processes the timeout"""

    garbageCollectionVal = router.timers[2]

    garbageColTime = time.time() + garbageCollectionVal

    router.routingTable[entryID][3][1] = garbageColTime

    router.routingTable[entryID][3][0] = None

    router.routingTable[entryID][0] = 16

    router.routingTable[entryID][2] = 1

ErrorHandler.py

# Error checks for RIP protocol

# Tim Lindbom & Ben Ireland

# 18/4/23

import sys

def RouterIdCheck(id):

    """runs Error checks on the ID to make

    sure the given value is valid"""

    numericCheck(id)

    id = int(id)

    if not (1 <= id <= 64000):

        print(f"{id} is not a valid ID number. \

              \n please give an id value between 1 and 64000")

        sys.exit()

    return id

def RouterOutputCheck(outputs):

    """performs error checks on the output lists.

    Returns list of all the output port info

    as well as a list of just output port numbers"""

    processedOutputs = []

    portNumbers = []

    for output in outputs:

        outputData = output.split("-")

        tempOutputs = []

        index = 0

        for value in outputData:

            numericCheck(value)

            tempOutputs.append(int(value))

            if index == 0:

                portNumbers.append(value)

            index += 1

        processedOutputs.append(tempOutputs)

    PortNumberChecks(portNumbers)

    return (processedOutputs, portNumbers)

def PortNumberChecks(portNumbers):

    """perfroms error checks on each

    input port number in the inputs list

    and returns a list of processed inputs"""

    processedPortNumbers = []

    for portNum in portNumbers:

        numericCheck(portNum)

        portNum = int(portNum)

        if not (1024 <= portNum <= 64000):

            print(f"input port number {portNum} is not a valid port number. \n \

                  Please make sure all input numbers are within 1024-64000")

            sys.exit()

        if portNum in processedPortNumbers:

            print(f"input port number {portNum} has been used more than once. \n \

                  Please make sure all input numbers are unique")

            sys.exit()

        processedPortNumbers.append(portNum)

    return processedPortNumbers

def CompareInputsOutputs(inputs, outputs):

    """comapres the input port and output port lists to

    make sure that no ports appear in both lists"""

    for inputPort in inputs:

        if inputPort in outputs:

            print(f"port number {inputPort} appears in both inputs and outputs")

            sys.exit()

    for outputPort in outputs:

        if outputPort in inputs:

            print(f"port number {outputPort} appears in both inputs and outputs")

            sys.exit()

def numericCheck(value):

    """checks that an excpected integer

    is the correct the correct type  """

    if value.isnumeric() == False:

        print(f"'{value}' is not a numeric value, please input a numeric value")

        sys.exit()

def TimerChecks(timers):

    """runs error check on the timer values"""

    firstTimer = timers[0]

    if (timers[1] == 6\*firstTimer) & (timers[2] == 4\*firstTimer):

        return timers

    else:

        print(f"timer values do not follow the correct format \n \

              please make sure the timers follow the [T, 6\*T, 4\*T] formatting")

        sys.exit()

def HeaderChecks():

    """run error checks on the packet haeder before sending

    before sending a response"""

RipTesting.py

#   ResponseHandler

# ---- TESTING BASE FUNCTIONALITY ----

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [30, 180, 240]])

# ComputeRoutingAlgorithm(router1, 1, [[1, 0], [3, 3]])

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# router1.PrintParams()

# response = GenerateResponse(router1)

# print(ReadResponse(response))

# ---- TESTING ALGORITHM FUNCTIONALITY ----

# router\_id 0, inputs 701 702 777, outputs 5000-1-1 5002-5-4

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [30, 180, 240]])

# ComputeRoutingAlgorithm(router1, 1, [[1, 0], [3, 3]])

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# UpdateRoute(router1, 4, 3, 1)

# router1.PrintParams()

# ---- TESTING TRIGGERED FUNCTIONALITY ----

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [30, 180, 240]])

# router1.OpenSockets()

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 16]])

# SendResponses(router1, True)

# ---- TESTING TIMER FUNCTIONALITY ----

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [3, 4, 3]])

# ComputeRoutingAlgorithm(router1, 1, [[1, 0], [3, 3]])

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# i = 0

# while(1):

#     CheckTimers(router1)

#     i += 1

#     if i == 1000000:

#         i = 0

#         router1.PrintParams()

# ---- TESTING RESPONSE GENERATION AND READING ----

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [30, 180, 240]])

# ComputeRoutingAlgorithm(router1, 1, [[1, 0], [3, 3]])

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# router1.PrintParams()

# response = GenerateResponse(router1, 1)

# print(ReadResponse(response))

# # ---- TESTING SENDING FUNCTIONALITY ----

# router1 = Router([0, [701, 702, 777], [[5000, 1, 1], [5002, 5, 4]], [6, 180, 240]])

# router1.OpenSockets()

# ComputeRoutingAlgorithm(router1, 1, [[1, 0], [3, 3]])

# router1.PrintParams()

# ResponseTimer(router1)

# ComputeRoutingAlgorithm(router1, 4, [[4, 0], [3, 2]])

# router1.PrintParams()

# UpdateRoute(router1, 4, 3, 1)

# router1.PrintParams()

# response = GenerateResponse(router1)

# print(ReadResponse(response))