COSC364 RIPv2 Assignment

Jesse Sheehan (53366509)
Will Cowper (81163265)

May 3, 2019

Contents

Plagiarism Declaration

This form needs to accompany your COSC 364 assignment submission.

I understand that plagiarism means taking someone else's work (text, program code, ideas, concepts) and presenting them as my own, without proper attribution. Taking someone else's work can include verbatim copying of text, figures/images, or program code, or it can refer to the extensive use of someone else's original ideas, algorithms or concepts.

I hereby declare that:

- My assignment is my own original work. I have not reproduced or modified code, figures/images, or writings of others without proper attribution. I have not used original ideas and concepts of others and presented them as my own.
- I have not allowed others to copy or modify my own code, figures/images, or writings. I have not allowed others to use original ideas and concepts of mine and present them as their own.
- I accept that plagiarism can lead to consequences, which can include partial or total loss of marks, no grade being awarded and other serious consequences, including notification of the University Proctor.

Name:	Will Cowper	Jesse Sheehan
Student ID:	81163265	53366509
Signature:	[illegible]	[illegible]
Date:	2-5-19	2/5/19

1 Questions

As required, the following questions have been answered:

1.1 Contribution

The contribution toward the entire project was an even split. We both felt as though the work we had contributed was worth 50% each.

1.2 Reflection

Some of the smaller modules in our codebase have been implemented quite well. For example, the Timer and Bencode modules have a very focused purpose and were discrete enough to be able to be doctested. We found that making use of recursion in the Bencode module reduced the complexity that would have otherwise occurred. The Timer module has many features that we didn't end up using but could be useful in the future if we decided to continue developing this project. We also had a clean user-interface that clearly displays the current routing table and some other information about the router. Our protocol adds a CRC32 checksum to the data being sent, this is checked when the packet is received and dropped if it is incorrect. This protects our routers from receiving garbled packets.

The overall system design could be improved. We rewrote some modules several times in order to get it to feel as though it would be easy to work with. We would also spend more time planning the project and understanding the exact steps required to implement the specification. Our current solution only receives at most 4096 bytes of data when reading a packet, this means that we can only send a maximum of 81 full router table entries in a packet. This could become an issue if we had a router with more than 80 directly connected neighbors but could be overcome by sending another packet with the remaining entries.

1.3 Event Processing

Our entire program is based around a main loop that waits for incoming packets and if it doesn't receive any, it will do other things, such as updating the timers, updating the routing table, rendering the screen, etc. We use lists to ensure that our incoming packets are serviced in the order in which they arrive. When packets are processed, they may trigger updates to the routers neighbors. These updates are serviced after the periodic updates have finished being received. Once these updates have been sent to its neighbors, the router simply waits for more information to arrive.

In order to ensure the atomicity of events in our program we have made use of timer-driven functions and their timers in such a way that they do not interrupt other events. Our entire program is single-threaded so we don't need to worry about interruptions from other parts of the program.

1.4 Testing

Many of the smaller functions in the project were discrete enough that we could use doctests on them. We created many configuration files for testing our config module. Some configuration files were well-formed and some were not. This allowed us to make sure that our config module worked as expected.

Once we had most of the program working we had to create test configuration files for entire networks. We found this process to be very tedious and so, wrote a program to generate these files for us (see section 2.3.1). Our testing became much easier after this as we didn't have to manually write these configuration files.

2 Appendices

2.1 Source Code

2.1.1 src/__main__.py

```python
#!/usr/bin/python3
"""

    __main__.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import sys
import os.path

import server
import config


def print_usage():
    """
        Prints the usage of the program.
    """
    print("usage: {0} <config_filename>".format(sys.argv[0]))


def print_filename_error(filename):
    """
        Prints a filename error.
    """
    print("Error: {0} doesn't exist.".format(filename))


def print_config_error():
    """
        Prints a configuration file error.
    """
    print("Error: Couldn't read the configuration file.")


def main():
    """
        The main entry point to the program.
    """

    if len(sys.argv) != 2:
        print_usage()
        return -1

    filename = sys.argv[1]
    file = None
    conf = None

    # accepts config from stdin
    if filename == '--':
        file = sys.stdin

    # or from a file
    else:
        if not os.path.exists(filename):
            print_filename_error(filename)
            return -1
        else:
            file = open(filename, "r")

    try:
        print("Reading configuration file... ", end='')
        conf = config.Config()
        conf.parse_file(file)
        print("done!")

    except:
        print_config_error()
        return -1

    try:
        print("Starting RIP router #" + str(conf.router_id))
        s = server.Server(conf)
        s.start()

    # Ignore KeyboardInterrupts
    except KeyboardInterrupt:
        pass

    # Re-raise other exceptions
    except Exception as err:
        raise err


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

../src/__main__.py

2.1.2 src/bencode.py

```
#!/usr/bin/python3
"""

    generate_network.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

    A bencoding implementation based on the official specification (https
    ://wiki.theory.org/index.php/BitTorrentSpecification#Bencoding)

"""


def bencode(value):
    """
    Test Integer Encoding:
    >>> bencode(42)
    'i42e'
    >>> bencode(0)
    'i0e'
    >>> bencode(-42)
    'i-42e'

    Test String Encoding:
    >>> bencode("spam")
    '4:spam'
    >>> bencode("i")
    '1:i'
    >>> bencode("")
    '0:'
    >>> bencode("COSC364 is the greatest course evarrrr!")
    '39:COSC364 is the greatest course evarrrr!'

    Test List Encoding:
    >>> bencode(["spam", 42])
    'l4:spami42ee'

    Test Dictionary Encoding:
    >>> bencode({"bar": "spam", "foo": 42})
    'd3:bar4:spam3:fooi42ee'

    """

    # integer encoding
    if type(value) is int:
        return "i" + str(value) + "e"
    # string encoding
    if type(value) is str:
        return str(len(value)) + ":" + value

    # list encoding
    if type(value) is list:
        return "l" + "".join(map(bencode, value)) + "e"

    # dictionary encoding
    if type(value) is dict:
        # TODO: keys should be in alphabetical order
        # TODO: check that the key is a string
        return "d" + "".join([bencode(k) + bencode(v) for k, v in value.
    items()]) + "e"

    raise ValueError(str(type(value)) +
                     " must be one of int, str, list or dict")


def bdecode(string, returnLength=False):
    """
    >>> bdecode("i42e")
    42
    >>> bdecode("i0e")
    0
    >>> bdecode("i-42e")
    -42

    >>> bdecode("i42e", True)
    (42, 4)
    >>> bdecode("i0e", True)
    (0, 3)
    >>> bdecode("i-42e", True)
    (-42, 5)

    >>> bdecode("4:spam")
    'spam'
    >>> bdecode("1:i")
    'i'
    >>> bdecode("0:")
    ''
    >>> bdecode("39:COSC364 is the greatest course evarrrr!")
    'COSC364 is the greatest course evarrrr!'

    >>> bdecode("4:spam", True)
    ('spam', 6)
    >>> bdecode("1:i", True)
    ('i', 3)
    >>> bdecode("0:", True)
    ('', 2)
    >>> bdecode("39:COSC364 is the greatest course evarrrr!", True)
    ('COSC364 is the greatest course evarrrr!', 42)
    >>> bdecode("l4:spami42ee")
    ['spam', 42]
    >>> bdecode("l4:spami42el9:more spami-42eee")
    ['spam', 42, ['more spam', -42]]

    >>> bdecode("l4:spami42ee", True)
    (['spam', 42], 12)
    >>> bdecode("l4:spami42el9:more spami-42eee", True)
    (['spam', 42, ['more spam', -42]], 30)

    >>> bdecode("d3:bar4:spam3:fooi42ee")
    {'bar': 'spam', 'foo': 42}
    >>> bdecode("d3:bar4:spam3:fooi42e4:listl4:spami42el9:more spami-42eeee
")
    {'bar': 'spam', 'foo': 42, 'list': ['spam', 42, ['more spam', -42]]}

    >>> bdecode("d3:bar4:spam3:fooi42ee", True)
    ({'bar': 'spam', 'foo': 42}, 22)
    >>> bdecode("d3:bar4:spam3:fooi42e4:listl4:spami42el9:more spami-42eeee
", True)
    ({'bar': 'spam', 'foo': 42, 'list': ['spam', 42, ['more spam', -42]]},
58)

    """

    value = None
    length = 0

    # integer decoding
    if string[0] == 'i':

        # get the end of the integer string
        end = string.find('e')
        if end == -1:
            raise ValueError(string[0:10] + "... is not a bencoded integer"
)

        # get the integer from the string (this may throw a ValueError)
        value = int(string[1:end])

        # update the length to account for the entire integer string
        length = end + 1

    # string decoding
    elif string[0].isnumeric():

        # get the end of the string length
        length_end = string.find(':')
        if length_end == -1:
            raise ValueError(string[0:10] + "... is not a bencoded string")

        # get the string length as an integer
```

```python
        str_length = int(string[0:length_end])

        # get the actual string
        value = string[length_end + 1:length_end + 1 + str_length]

        # update the length to be the length of the string including the string length
        length = length_end + 1 + str_length

    # list decoding
    elif string[0] == 'l':

        # set the offset to 1 to account for the starting 'l'
        offset = 1
        value = []

        while string[offset] != 'e':

            # decode the inner value
            inner_value, inner_length = bdecode(string[offset:], True)
            offset += inner_length

            # update the list
            value.append(inner_value)

        # update the length to account for the closing 'e'
        length = offset + 1

    # dictionary decoding
    elif string[0] == 'd':

        # set the offset to 1 to account for the starting 'd'
        offset = 1

        # in Python >= 3.6, the dictionary implementation remembers the
        # insertion order
        value = {}

        while string[offset] != 'e':

            # decode the key
            inner_key, inner_length = bdecode(string[offset:], True)
            offset += inner_length
            # TODO: inner_key should be a string

            # decode the value
            inner_value, inner_length = bdecode(string[offset:], True)
            offset += inner_length

            # update the dictionary
            value[inner_key] = inner_value

        # TODO: validate that the keys are in alphabetical order

        # update the length to account for the closing 'e'
        length = offset + 1

    # return the length as well if requested
    if returnLength:
        return value, length
    else:
        return value


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

../src/bencode.py

2.1.3 src/config.py

```
#!/usr/bin/python3

"""

    config.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import configparser
import os
import random


class Config:

    """
        Config class used for abstracting the stored config
    """

    def __init__(self):
        self.router_id = 0
        self.input_ports = []
        self.output_ports = []
        self.periodic_update = 0

```

```python
    def parse_file(self, file):
        c = read_config_file(file)
        self.router_id = c["routerId"]
        self.input_ports = c["inputPorts"]
        self.output_ports = [
            OutputPort(o["outputPort"], o["cost"], o["routerId"]) for o in c["outputPorts"]
        ]
        self.periodic_update = c["periodicUpdate"]

    def __str__(self):
        return "Config <id={0}, input_ports={1}, output_ports={2}, periodic_update={3:.3}s>".format(self.router_id, self.input_ports, self.output_ports, self.periodic_update)

    def __repr__(self):
        return self.__str__()


class OutputPort:
    def __init__(self, port, cost, id):
        self.router_id = id
        self.port = port
        self.cost = cost

    def __str__(self):
        return "OutputPort <id={0}, port={1}, cost={2}>".format(self.router_id, self.port, self.cost)

    def __repr__(self):
        return self.__str__()


def read_config_file(file):
    """
        Parses a given file and returns a dict containing the routerID, input ports
        and output ports with their cost and next hop
    """
    #Create an instance of configparser object
    config = configparser.ConfigParser()
    config.read_file(file)
    # dict declartion
    router = {}
    # Reading in each section of the config
    routerId = (config.get('DEFAULT', 'router-id'))
    inputPorts = (config.get('DEFAULT', 'input-ports'))
    outputPorts = (config.get('DEFAULT', 'output-ports'))
    # Checks config file for periodic timer override or defaults to standard
    periodicUpdate = config.get("DEFAULT", "periodic-update", fallback=3.0)

    # Validating all parameters
    router["routerId"] = check_router_id(routerId)
    router["inputPorts"] = check_input_ports(inputPorts)
    router["outputPorts"] = check_output_ports(router, outputPorts)
    router["periodicUpdate"] = check_periodic_update(periodicUpdate)

    return router


def check_periodic_update(periodicUpdate):
    """
        Reduces the chance of collisons and other nasties by implementing a
     random wait to the periodicUpdate
    """
    return periodicUpdate + (random.random() * 2 - 1)


def check_router_id(routerId):
    """
        Takes a routerID string from the config and checks it
        Returns it back as an int if its valid
    """
    try:
        routerId = int(routerId)
    except:
        raise TypeError("RouterID must be an integer")
    if (routerId > 64000 or routerId < 1):
        raise ValueError("RouterID must be between 1 and 64000")
    return routerId


def check_input_ports(inputPorts):
    """
        Takes a string of inputports from the config
        Validates and then returns them as a list
    """
    try:
        inputPorts = [int(port.strip()) for port in inputPorts.split(',')]
    except:
        raise TypeError("Input ports should be comma seperated ints")
    for port in inputPorts:
        if (port > 64000 or port < 1024):
            raise ValueError("Port should be between 1024 and 64000")
    if len(inputPorts) != len(set(inputPorts)):
        raise ValueError("Ports should be unique")
    return inputPorts


def check_output_ports(router, outputPorts):

    """
        Takes an incomplete router dict containing a routerID and input
    ports
        Tests the routerID and input ports against a list of outputPorts
        Returns a list of outputPorts if they are all valid.

    """
    outportPortList = []
    try:
        outputPorts = [port.strip() for port in outputPorts.split(',')]
    except:
        raise TypeError(
            "Outport ports should be comma seperated in the form PORT-COST-
ID")
    for output in outputPorts:
        config = {}
        output = output.split('-')
        output = [int(i) for i in output]
        config["cost"] = output[1]

        if (output[0] > 64000 or output[0] < 1024):
            raise ValueError("Port should be between 1024 and 64000")
        if output[2] == router["routerId"]:
            raise ValueError("Output port routerID matches own routerID")
        if any(d.get('routerId', None) == output[2] for d in
outportPortList):
            raise ValueError("RouterID already exists in output list")
        config["routerId"] = output[2]
        if output[0] in router["inputPorts"]:
            raise ValueError("Outport port is shared with an input port")
        if any(d.get('outputPort', None) == output[0] for d in
outportPortList):
            raise ValueError("OutputPort already in use")
        config["outputPort"] = output[0]
        outportPortList.append(config)

    return outportPortList

def open_config_file(filePath):
    """
        Takes a filepath as argument, validates it and returns a Config
object
    """
    file = open(filePath, 'r')
    if file.mode == 'r':
        config = Config()
        config.parse_file(file)
    else:
        print("Error opening file")
    return config
```

../src/config.py

2.1.4 src/protocol.py

```
#/usr/bin/python3
```

```
"""

    protocol.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import bencode
import binascii

__encoding = "utf-8"


def encode(data):
    """
        Encodes the raw data, including a checksum.
    """
    body = bencode.bencode(data).encode(__encoding)
    crc = binascii.crc32(body)
    return crc.to_bytes(4, "big") + body


def decode(data):
    """
        Decodes raw data, checks the validity and returns the dictionary
    containing the data.
        Returns None if the data is invalid.
    """
    try:
        # get the CRC32 code
        crc = int.from_bytes(data[:4], "big")

        # get the body
        body = data[4:]

        # return None if the checksum is incorrect
        if crc != binascii.crc32(body):
            return None

        # return the decoded data if the checksum is correct
        else:
            return bencode.bdecode(body.decode(__encoding))
    except:
        return None
```

```
class Packet:
    """
        A Packet is used to send and receive updates from other RIP routers
    .
    """

    def __init__(self, link_cost = -1, routes = []):
        """
            Creates a new Packet.
        """
        self.link_cost = link_cost
        self.routes = routes

    def from_data(self, data):
        """
            Sets the packet information from some raw data.
            Returns True if successful.
        """
        d = decode(data)

        if d is not None:
            self.link_cost = d["link-cost"]
            self.routes = d["routes"]
            return True
        else:
            return False

    def to_data(self):
        """
            Returns the raw data to be sent.
        """
        return encode({
            "link-cost": self.link_cost,
            "routes": self.routes
        })
```

../src/protocol.py

2.1.5 src/routing_table_entry.py

```
#!/usr/bin/python3

"""

    routing_table_entry.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)
"""

class RoutingTableEntry:
    """
    A RoutingTableEntry represents a RIP entry that resides in the routing table
    """

    def __init__(self, destination, nextHop, cost):
        self.destination = destination
        self.nextHop = nextHop
        self.cost = cost
        self.age = 0.0
        self.garbage = False

    def __str__(self):
        return "RouteTableEntry <destination={0}, nextHop={1}, cost={2}, age={3}, garbage={4}>".format(self.destination, self.nextHop, self.cost, round(self.age, 2), self.garbage)

    def __repr__(self):
        return self.__str__()
```

../src/routing_table_entry.py

2.1.6 src/routing_table.py

```
#!/usr/bin/python3
"""

    routing_table.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)
"""


import os
from routing_table_entry import RoutingTableEntry
import config


class RoutingTable:
    """
        The RoutingTable represents the list of RoutingTableEntries for a
    router.
    """

    def __init__(self, config, logging_function = None):
        """
            Creates a new RoutingTable based on the Config.
        """
        self.__routes = []
        self.routerID = config.router_id
        self.__logging_function = logging_function

    def add_entry(self, destination, nextHop, totalCost):
        """
            Adds a new RoutingTableEntry to the RoutingTable.
        """
        route = RoutingTableEntry(destination, nextHop, totalCost)
        self.__routes.append(route)

    def set_garbage(self, routerID, isGarbage):
        """
            Sets the garbage flag of the entry.
        """
        index = self.get_index(routerID)
        self.__routes[index].garbage = isGarbage
        self.reset_age(routerID)
        if isGarbage:
            self.set_cost(routerID, 16)

    def log(self, message):
        if self.__logging_function is not None:
            self.__logging_function(message)
        else:
            print(message)

    def reset_age(self, routerID):
        """
            Resets the age of the entry to 0.
        """
        index = self.get_index(routerID)
        self.__routes[index].age = 0.0

    def increment_age(self, time):
        """
            Increments the age of all entries in the RoutingTable.
        """
        for entry in self.__routes:
            if entry.destination != self.routerID:
                entry.age += time

    def delete_entry(self, routerID):
        """
```

```python
            Deletes an entry with the specific routerID from the RoutingTable.
        """
        index = self.get_index(routerID)
        del self.__routes[index]

    def get_index(self, routerID):
        """
            Gets the index of the entry with the routerID. Returns -1 if not found.
        """
        for i, route in enumerate(self.__routes):
            if route.destination == routerID:
                return i
        return -1  # Not found

    def set_cost(self, routerID, cost):
        """
            Sets the cost of the entry.
        """
        index = self.get_index(routerID)
        self.__routes[index].cost = cost

    def set_next_hop(self, routerID, nextHop):
        """
            Sets the next hop of the entry.
        """
        index = self.get_index(routerID)
        self.__routes[index].nextHop = nextHop

    def update(self, triggered_update_callback):
        """
            Performs house-keeping on the entries.
            The `triggered_update_callback` is for performing triggered updates.
        """
        remove_routes = []
        triggered_routes = []

        for route in self.__routes:
            if route.age > 10 and not route.garbage:
                self.log("marked router " + str(route.destination) + " as garbage")
                self.set_garbage(route.destination, True)
                triggered_routes.append(route)

            if route.age > 20 and route.garbage:
                self.log("purged router " + str(route.destination) + " from database")
                remove_routes.append(route)

        if len(triggered_routes) != 0:
            triggered_update_callback(triggered_routes)

        for route in remove_routes:
            self.__routes.remove(route)

    def __getitem__(self, routerId):
        """
            Gets the entry with the given routerId.
        """
        index = self.get_index(routerId)
        if index != -1:
            return self.__routes[index]
        return None

    def __iter__(self):
        """
            Returns the iterator of the routes.
        """
        return iter(self.__routes)

    def __len__(self):
        """
            Returns the number of routes this RoutingTable has.
        """
        return len(self.__routes)

    def __str__(self):
        """
            Returns a human-readable RoutingTable that can be printed to the terminal.
        """
        s = [
            "+------------+------------+------------+------------+------------+",
            "| Dest.      | Next Hop   | Total Cost | Age        | Garbage?   |",
            "+------------+------------+------------+------------+------------+"
        ]
        for route in self.__routes:
            s.append("| {0:<10} | {1:<10} | {2:<10} | {3:<10} | {4:<10} |".format(
                route.destination, route.nextHop, route.cost, round(route.age, 2), route.garbage))
        s.append("+------------+------------+------------+------------+------------+")
        return os.linesep.join(s)


# runs a simple test
if __name__ == "__main__":
    current_directory = os.path.dirname(__file__)
    parent_directory = os.path.split(current_directory)[0]
    file_path = os.path.join(parent_directory, 'configs/good/01.conf')
    config = config.open_config_file(file_path)
    r = RoutingTable(config)
    print(r)
```

../src/routing_table.py

2.1.7 src/server.py

```
#!/usr/bin/python3

"""

    server.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import socket
import select
import time

import timer
import routing_table
import routing_table_entry
import protocol
import utils
import bencode


def create_input_socket(port, host='localhost'):
    """
        Creates a new UDP socket.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((host, port))
    return sock


class Server:

    def __init__(self, config):
        """
            Creates a new server with a configuration.
        """
        self.rt = routing_table.RoutingTable(config, self.log)
        self.config = config
        self.input_ports = []
        self.periodic_timer = None
        self.loglines = []

    def print_display(self):
        """
            Displays useful information for the user.
        """

        # clear the screen
        utils.clear_terminal()

        # print info about this router
        print("RIP Router #" + str(self.config.router_id))
        print("Uptime: {0} seconds".format(
            round(self.periodic_timer.getElapsed())))

        # print the routing table
        print(self.rt)

        # print other info
        print("Press Ctrl+C to quit")

    def process_periodic_update(self, dt):
        """
            Called when the periodic timer is triggered.
        """
        # send destination, next hop and total cost of each routing entry to each input port
        sock = self.input_ports[0]

        for output_port in self.config.output_ports:

            # add self to the routes
            routes = [{
                    "destination": self.config.router_id,
                    "cost": 0,
                    "next-hop": self.config.router_id
                }]

            # if len(self.rt) == 0:
            #     self.log("advertising self to " + str(output_port.router_id))

            for route in self.rt:

                cost = route.cost
                destination = route.destination

                # poison reverse by setting cost to 16 when announcing routes back from where they were learned
                if self.rt[destination].nextHop == output_port.router_id:
                    cost = 16

                routes.append({
                    "destination": destination,
                    "cost": cost,
                    "next-hop": self.config.router_id
                })

            packet = protocol.Packet(output_port.cost, routes)
            sock.sendto(packet.to_data(), ('localhost', output_port.port))

    def log(self, message):
        """
            Writes to the information log (for a maximum of 10 lines).
        """
        self.loglines.append(message)
        while len(self.loglines) > 10:
            self.loglines = self.loglines[1:]

    def process_incoming_data(self, addr, data):
        """
            Called when incoming data is received. The data returned from
this function is sent back through the socket. If None is returned,
nothing will be sent.
        """

        triggered_updates = []
        packet = protocol.Packet()

        if not packet.from_data(data):
            self.log("invalid packet hash")
            return

        for route in packet.routes:

            route_destination = route["destination"]
            route_cost = route["cost"]
            route_next_hop = route["next-hop"]

            destination_entry = self.rt[route_destination]

            # Check route is valid before any processing is done

            # route lists ourself as the destination (useless) or as the
next hop (invalid) and should be dropped
            if route_destination == self.config.router_id or route_next_hop
 == self.config.router_id:
                continue

            # route contains a negative cost and should be dropped
            if (route_cost < 0) or (packet.link_cost < 0):
                continue
            # route is valid and should be processed

            # total cost is the link cost added to the cost contained in the packet
            total_destination_cost = route_cost + packet.link_cost

            is_destination_unreachable = (total_destination_cost >= 16)

            # clamp cost to maximum of 16
            if is_destination_unreachable:
                total_destination_cost = 16

            # is the destination routerID known
            is_destination_in_table = destination_entry is not None

            # New valid route
            if not is_destination_in_table and not is_destination_unreachable:

                # put the destination in the table
                self.rt.add_entry(route_destination, route_next_hop, total_destination_cost)
                self.log("added new router " + str(route_destination) + " via " + str(route_next_hop) + " with a cost of " + str(total_destination_cost))

            # Route already exists in table
            elif is_destination_in_table:

                is_destination_garbage = destination_entry.garbage

                # Check for a better route.
                if total_destination_cost < destination_entry.cost:
                    self.rt.set_cost(route_destination, total_destination_cost)
                    self.rt.set_garbage(route_destination, False)
                    self.rt.set_next_hop(route_destination, route_next_hop)

                    self.log("found new route to " + str(route_destination) + " via " + str(route_next_hop) + " with a cost of " + str(total_destination_cost))

                # Check for worse route from the same hop
                elif route_next_hop == destination_entry.nextHop and total_destination_cost > destination_entry.cost:
                    if is_destination_unreachable:
                        # garbage it if we haven't seen it before, otherwise ignore it
                        if not is_destination_garbage:
                            self.rt.set_garbage(route_destination, True)
                            triggered_updates.append(destination_entry)
                            self.log("processed a triggered update from " + str(packet.routes[0]["next-hop"]) + " marked " + str(route_destination
    ) + " as garbage")

                    # We got a worse route from the samehop but its not
    infinite. As a neighbour we MUST update to the higher cost.
                    else:
                        self.rt.set_cost(route_destination,
    total_destination_cost)
                        self.rt.reset_age(route_destination)

                # Check for worse route from a different hop and ignore it
                elif total_destination_cost > destination_entry.cost:
                    #self.log("Worse route to " + str(route_destination) +
    " ignoring it")
                    continue

                # Check for same route and keep it alive
                elif route_next_hop == destination_entry.nextHop and
    total_destination_cost == destination_entry.cost:
                    # We dont want to keep alive infinite weight routes
                    if not is_destination_garbage:
                        self.rt.reset_age(route_destination)

        if len(triggered_updates) > 0:
            self.log("sending triggered updates")
            self.process_triggered_updates(triggered_updates)

        return None

    def process_triggered_updates(self, routes):
        """
            Processes the triggered updates.
        """
        sock = self.input_ports[0]
        for output_port in self.config.output_ports:

            packet_routes = [{
                    "destination": route.destination,
                    "cost": 16,
                    "next-hop": self.config.router_id
                } for route in routes]

            p = protocol.Packet(output_port.cost, packet_routes)
            sock.sendto(p.to_data(), ('localhost', output_port.port))

    def start(self):
        """
            Starts the server.
        """

        # set up the input ports
        self.input_ports = list(
            map(create_input_socket, self.config.input_ports))
        # start the periodic timer
        self.periodic_timer = timer.Timer(
            self.config.periodic_update, self.process_periodic_update)
        self.periodic_timer.start()
        self.periodic_timer.trigger()

        # only block for a second at a time
        blocking_time = 0.1

        loop_time = time.time()

        while self.input_ports:
            readable, _writable, exceptional = select.select(
                self.input_ports, [], self.input_ports, blocking_time)

            # increment the age
            dt = time.time() - loop_time
            self.rt.increment_age(dt)

            # redisplay the screen
            self.print_display()

            # update the timer, may call process_periodic_update
            self.periodic_timer.update()

            # may call process_triggered_updates
            self.rt.update(self.process_triggered_updates)

            # display the information log
            print("")
            print("Information Log:")
            for line in self.loglines:
                print(" ", line)

            # iterate through all sockets that have data waiting on them
            for sock in readable:
                data, addr = sock.recvfrom(4096)
                resp = self.process_incoming_data(addr, data)

                if resp is not None:
                    sock.sendto(resp, addr)

            # removes a socket from the input list if it raised an error
            for sock in exceptional:
                if sock in self.input_ports:
                    self.input_ports.remove(sock)
                    sock.close()
                raise Exception("A socket raised an error")

            # update the loop time
            loop_time = time.time()

if __name__ == "__main__":
    pass
```

../src/server.py

2.1.8 src/timer.py

```
#!/usr/bin/python3
"""

    timer.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import time

class Timer:

    def __init__(self, period, callback):
        """
            Creates a new Timer with a period and a callback.
        """
        self.__period = period
        self.__callback = callback
        self.__started = False
        self.__startedTime = 0
        self.__paused = False
        self.__pausedTime = 0
        self.__updateTime = 0

    def start(self):
        """
            Starts the timer.
        """
        if not self.__started:
            t = time.time()
            self.__started = True
            self.__startedTime = t
            self.__paused = False
            self.__pausedTime = 0
            self.__updateTime = t

    def stop(self):
        """
```

```
            Stops the timer.
        """
        if self.__started:
            self.__started = False
            self.__startedTime = 0
            self.__paused = False
            self.__pausedTime = 0
            self.__updateTime = 0

    def reset(self):
        """
            Resets the timer.
        """
        if self.__started:
            self.stop()
            self.start()

    def pause(self):
        """
            Pauses the timer.
        """
        if self.__started and not self.__paused:
            self.__paused = True
            self.__pausedTime = time.time() - self.__startedTime
            self.__startedTime = 0

    def resume(self):
        """
            Resumes the timer.
        """
        if self.__started and self.__paused:
            self.__startedTime = time.time() - self.__pausedTime
            self.__paused = False
            self.__pausedTime = 0

    def update(self):
        """
            Updates the timer. May call its callback.
        """
        if self.__started and not self.__paused:
            t = time.time()
            dt = t - self.__updateTime
            if dt > self.__period:
                self.__updateTime = t
                self.__callback(dt)

    def trigger(self):
        """
            Forcefully call the callback.
        """
        if self.__started and not self.__paused:
            t = time.time()
            dt = t - self.__updateTime
            self.__updateTime = t
            self.__callback(dt)

    def getElapsed(self):
        """
            Returns the time elapsed in seconds.
        """
        if self.__started:
            if self.__paused:
                return self.__pausedTime
            else:
                return time.time() - self.__startedTime
        return 0.0

    def isStarted(self):
        """
            Returns True if the timer has been started.

        >>> t = Timer(10, None)
        >>> t.isStarted()
        False
        >>> t.start()
        >>> t.isStarted()
        True
        >>> t.stop()
        >>> t.isStarted()
        False
        """
        return self.__started

    def isPaused(self):
        """
            Returns True if the timer has been paused.

        >>> t = Timer(10, None)
        >>> t.isPaused()
        False
        >>> t.start()
        >>> t.isPaused()
        False
        >>> t.pause()
        >>> t.isPaused()
        True
        >>> t.resume()
        >>> t.isPaused()
        False
        >>> t.stop()
        >>> t.isPaused()
        False
        >>> t.start()
        >>> t.pause()
        >>> t.isPaused()
        True
        >>> t.stop()
        >>> t.isPaused()
        False
        """
        return self.__paused and self.__started

    def __str__(self):
        """
            Returns a string representation of the timer.
        """
        return "Timer <period={0:.3}s, started={1}, paused={2}, elapsed
    ={3:.3}s>".format(self.__period, self.__started, self.__paused, self.
    getElapsed())

    def __repr__(self):
        """
            Returns a string representation of the timer.
        """
        return self.__str__()

# run doctests
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

../src/timer.py

2.1.9 src/utils.py

```
#!/usr/bin/python3

"""

    utils.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import os

def clear_terminal():
    """
        Clears the terminal based on the type of operating system.
    """
    # the terminal clear command for linux
    if os.name == "posix":
        os.system("clear")

    # the console cls command for windows
    elif os.name == "nt":
        os.system("cls")

    # otherwise, just print 25 newlines
    else:
        for _ in range(25):
            print("")
```

../src/utils.py

2.2 Configuration Files

2.2.1 configs/networks/figure1/1.conf

```
; configs/networks/figure1/1.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 1
input-ports = 55501, 55503, 55505
output-ports = 55500-1-2, 55502-5-6, 55504-8-7
```

../configs/networks/figure1/1.conf

2.2.2 configs/networks/figure1/2.conf

```
; configs/networks/figure1/2.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 2
input-ports = 55500, 55507
output-ports = 55501-1-1, 55506-3-3
```

../configs/networks/figure1/2.conf

2.2.3 configs/networks/figure1/3.conf

```
; configs/networks/figure1/3.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 3
input-ports = 55506, 55509
output-ports = 55507-3-2, 55508-4-4
```

../configs/networks/figure1/3.conf

2.2.4 configs/networks/figure1/4.conf

```
; configs/networks/figure1/4.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 4
input-ports = 55508, 55511, 55513
output-ports = 55509-4-3, 55510-2-5, 55512-6-7
```

../configs/networks/figure1/4.conf

2.2.5 configs/networks/figure1/5.conf

```
; configs/networks/figure1/5.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 5
input-ports = 55510, 55515
output-ports = 55511-2-4, 55514-1-6
```

../configs/networks/figure1/5.conf

2.2.6 configs/networks/figure1/6.conf

```
; configs/networks/figure1/6.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 6
input-ports = 55502, 55514
output-ports = 55503-5-1, 55515-1-5
```

../configs/networks/figure1/6.conf

2.2.7 configs/networks/figure1/7.conf

```
; configs/networks/figure1/7.conf
; created with tools/generate_network.py

[DEFAULT]
router-id = 7
input-ports = 55504, 55512
output-ports = 55505-8-1, 55513-6-4
```

../configs/networks/figure1/7.conf

2.3 Other Files

2.3.1 tools/generate_network.py

The following file will interactively prompt the user for information about a network. It will then create all the necessary configuration files for the network to run.

```
#!/usr/bin/python3

"""

    generate_network.py

    COSC364 RIP Assignment

    Date: 02/05/2019

    Written by:
     - Will Cowper (81163265)
     - Jesse Sheehan (53366509)

"""

import os
import sys


def get_network_name():
    network_name = None
    while network_name is None:
        try:
            network_name = input("Enter network name: ")
            network_name = network_name.strip()
            if not network_name.isalnum():
                print("Network name must be alpha-numeric")
                network_name = None
        except:
            print("ASD")
            return None
    return network_name


def get_router_ids():
    router_ids = []
    while len(router_ids) == 0:
        try:
            line = input("Enter router ids seperated by spaces: ")
            router_ids = [id for id in line.strip().split(" ")]
            is_valid = True
            for id in router_ids:
                if not id.isalnum():
                    is_valid = False
                    break
            if not is_valid:
                print("All ids must be alpha-numeric")
                router_ids = None
        except:
            return None
    return router_ids


def get_link_cost(fromId, toId):
    link_cost = None
    while link_cost is None:
        try:
            line = input("Enter link cost between routers '" +
                         str(fromId) + "' and '" + str(toId) + "': ")
            line = line.strip()
            if not line.isnumeric() or int(line) < 0:
                print("Link cost must be a positive integer (or 0 for
    infinity)")
            else:
                link_cost = int(line)
        except Exception as e:
            print(e)
            return None
    return link_cost


def main():
    network_name = get_network_name()
    if network_name is None:
        return

    router_ids = get_router_ids()
    if router_ids is None:
        return

    configs = {}
    port_number_max = 55500
    for index, fromId in enumerate(router_ids):
        for toId in router_ids[index + 1:]:
            link_cost = get_link_cost(fromId, toId)
            if link_cost is None:
                return

            if link_cost == 0:
                continue

            to_port_number = port_number_max
            port_number_max += 1
            from_port_number = port_number_max
            port_number_max += 1

            if fromId not in configs:
                configs[fromId] = {"output-ports": [],
                                    "input-ports": [], "router-id": fromId}
            configs[fromId]["output-ports"].append(
                (to_port_number, link_cost, toId))
            configs[fromId]["input-ports"].append(from_port_number)

            if toId not in configs:
                configs[toId] = {"output-ports": [],
                                 "input-ports": [], "router-id": toId}
            configs[toId]["output-ports"].append(
                (from_port_number, link_cost, fromId))
            configs[toId]["input-ports"].append(to_port_number)

    # assign port numbers
    root_path = os.path.join("configs", "networks", network_name)
    if not os.path.exists(root_path):
        os.mkdir(root_path)
        print("Created directory", root_path)

    for key in configs:
        config = configs[key]
        filename = os.path.join(root_path, config["router-id"] + ".conf")
        with open(filename, "w") as f:
            f.write("; " + filename + "\n")
            f.write("; created with tools/generate_network.py\n")
            f.write("\n")
            f.write("[DEFAULT]\n")
            f.write("router-id = " + str(config["router-id"]) + "\n")
            f.write("input-ports = " + ", ".join([str(x)
                                                  for x in config["input-
    ports"]]) + "\n")
            f.write("output-ports = " + ", ".join([str(x[0]) + "-" + str(x
    [1]) + "-" + str(x[2])
                                                   for x in config["output-
    ports"]]) + "\n")
            f.write("\n")
            print("Created", filename)

    # print("Creating ", network_name, "with", configs)


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

../tools/generate_network.py