# COSC364 Assignment 1

Frederik Markwell (fma107) - 51118501

Christopher Stewart (cst141) - 21069553

# Questions

## Contribution percentages

Christopher Stewart: 50%
Frederik Markwell: 50%

## List of contributions

Christopher Stewart (cst141) 21069553

- Basic skeleton program
- Send and receive packet
- Packet validation

Frederik Markwell (fma107) 51118501

- Configuration file validation
- Code cleanup and commenting
- Report editing
- Forwarding table printer

All other major parts of the daemon were pair programmed on the same computer.

## Successes

One aspect of the program that we feel was particularly well done is the overall structure of the program. We went with an object oriented approach, with a main singleton class (Rip_Router) which represents a single router, and a second class that represents an entry in a forwarding table (Row). We particularly like this approach because it allows us to represent all the data and functionality of the router in an easily understandable way.

Another aspect we particularly like is the sending of triggered updates. Originally we simply called the send_all_responses() in update_table() when it saw a cost of 16. We modified this to instead set a flag called triggered_update_waiting, and then the actual update is sent in the main run loop. This ensures that triggered updates contain all the changes made in the routing table, and also allows us to send the updates when a route times out.

## Improvements needed

One of the areas that could have been improved is our event scheduler. Every iteration through the main run loop, the router checks whether any of the timers have expired. This does have a higher CPU load than implementing it so that the program sleeps and is awoken by some mechanism when it is time to process the timer.

We mitigated this issue by giving our select call a timeout of 0.1 instead of 0. The CPU no longer has to run constantly, and in our testing the CPU usage of each process was so low as to round to 0.  This does mean that some timers may fire slightly later than they should, but given that the timeouts are randomised over a much bigger range than 0.1 s the effect is negligible.

## Event processing atomicity

Our program is single threaded - each function call returns before the next one can run. Upon an event (timer firing, receiving packet), the code for processing that event runs, finishes, and only then does the code go back to check for another event. This effectively means that two events will never interfere with each other.

## Weaknesses of RIP

We encountered two issues with RIP in the course of testing our program. The first is the count to infinity problem. As per the specification, we had implemented split-horizon with poison reverse. We tested this on a sample network, and were surprised to see the routers still counting to infinity. It turned out that if there is a larger loop in the network, poisoned reverse doesn't help, as RIP only knows if a route came directly from a neighbouring router. RIP seemingly has no defence against this other than capping infinity at 16.

The second issue with RIP is this capping to 16.  In the example network in the assignment specification, if routers 3 and 6 go down the only route between routers 2 and 5 has a cost of 16. Since 16 is treated as infinity, the routers ignore this route, and treat each other as unreachable, when in reality the route is perfectly available.

# Testing

## Blue Sky Convergence

Our first test was to make sure that in a blue sky scenario the routers converge to a minimal hop network. We set up the routers as described in the example network, and computed the shortest path tree to all nodes from router 1 using an implementation of Dijkstra's algorithm (created for COSC262). Our expected results are in Table 1. We then started all the routers, and examined the table of router 1 (Table 2). The actual routing table, taken from the running program (Table 2) was identical.

Similar results were established comparing the Dijkstra created tables for other routers to their actual tables. This shows that our routing daemon allows routers to successfully acquire new and/or better routes to destinations.

Table 1: Expected Forwarding Table for Router 1

| Address | Next Hop | Cost |
| --- | --- | --- |
| 1 | 1 | 0 |
| 2 | 2 | 1 |
| 3 | 2 | 4 |
| 4 | 2 | 8 |
| 5 | 6 | 6 |
| 6 | 6 | 5 |
| 7 | 7 | 8 |

Table 2: Actual forwarding table for Router 1

```
------------------------------
Forwarding Table for 1
Address | Next Hop | Cost | Timer | Change
---------------------------------------------
   1    |    1     |  0   |  0.00 | False
   2    |    2     |  1   |  0.84 | False
   3    |    2     |  4   |  0.84 | False
   4    |    2     |  8   |  0.84 | False
   5    |    6     |  6   |  0.83 | False
   6    |    6     |  5   |  0.83 | False
   7    |    7     |  8   |  0.85 | False
```

## Poisoned Reverse

Our next test was testing that routers were correctly poisoning routes to neighbouring routers. We set up a network with three routers 5, 6, 7, connected in a triangle. Code was added so that routers print out the messages they receive from other routers. We expect these messages should show a 16 as the metric for any route that goes through that router.

Indeed, this was the case. Below is a message that 5 received from 7. 7's routes to get to 6 and 5 both go through 5, so they are sent with a cost of 16.

```
{7: (cost:0, next_hop:7), 6: (cost:16, next_hop:7), 5: (cost:16, next_hop:7)}
```

We can conclude from this that the poisoned reverse mechanism is working correctly.

## Route Timeouts and Garbage Collection

We next tested that if a route is not updated regularly it times out and has its cost set to 16. After allowing the example network to converge, router 6 was shut down. We expected that the neighbouring routers routes that include router 6 should time out, resulting in a flow on effect. Each route that uses router 6 will have its metric set to 16.

In practice, we watched routers 1 and 5 as their timers ran out and they set the metric to 16, then saw their updates propagate across the network.

This scenario is also a good test of garbage collection. We expect that after some time, all routes to 6 will be deleted from routing tables. We waited 120 s after the routes had their cost set to 16, and they were deleted as expected.

## Triggered Updates

To test that a triggered update is sent and propagated, when a router goes down a scenario is set similar to the above timeout and garbage collection test, but with the response timer for each router set high enough so the triggered update changes can be seen separate from normal updates. As router 6 is shut down, its neighbours 1 and 5 will first send triggered updates and then the neighbours of 1 and 5 will send triggered updates and so on. Router 3 should be the last router to update its cost to reach router 6 to 16 (assuming it doesn't receive a response with a replacement route beforehand).

## Periodic Update Variation

To test that response packets are sent with a random variation (e.g 30 ± 5s), we wait for the network to converge, then analyse the forwarding table. Only entries with the same next hop will have the same timer value. This was true in practice.

## Sending Partial Routing Tables

Triggered update packets should only include the RIP entries for routes that have changed. We test this by printing the length of packets before they are sent. When a single isolated router with 1 neighbour goes down, the length of the triggered update sent by its neighbour (after the route had timed out) is 24 bytes in theory and in practice.

With a network like 1-7-4 and the shutting down of router 1, the payload length (in the triggered update) sent by router 7 is 20 (a single RIP entry) while the table holds three entries so would normally send 60 + 4 bytes. The payload length is printed twice as two messages are sent - one to each neighbour.

Table 3: Router 7 sending a triggered update with a single RIP entry (20 bytes)

```
------------------------------
Forwarding Table for 7
Address | Next Hop | Cost | Timer | Change
---------------------------------------------
   1    |    1     |  16  |  5.01 |  True
   4    |    4     |  6   |  0.42 | False
   7    |    7     |  0   |  0.00 | False
payload len: 20
payload len: 20
Sent a triggered update!
```

## Link Cost Increases

To ensure that an increased link cost propagates across the network, we set the route timeout to a high value. This allowed us to stop two routers, edit the link cost between them, and restart them before any neighbouring routers routes could time out (and trigger an update).

As router 4 uses 5 to reach 6, increasing the cost between 5 and 6 from 1 to 2 should result in router 4 citing a cost to reach 6 from 3 to 4. Router 3 uses 4 to reach 6 so will also change its cost from 7 to 8. This behaviour was witnessed in practice.

Table 4: Router 3 converging correctly after a link cost increase between 5 and 6

```
------------------------------
Forwarding Table for 3
Address | Next Hop | Cost | Timer | Change
--------------------------------------------
-
   1    |    2     |  4   |  0.41 | False
   2    |    2     |  3   |  0.41 | False
   3    |    3     |  0   |  0.00 | False
   4    |    4     |  4   |  0.10 | False
   5    |    4     |  6   |  0.10 | False
   6    |    4     |  8   |  0.10 | False
   7    |    4     |  10  |  0.10 | False
```

## Ignoring Unreachable Routes

To test that an unreachable route is ignored, we used the example network without routers 6 and 3, noting that the cost from 5 to 1 would be 16, and 5 to 2 would be 17. As expected router 5 does not include an address entry to router 1 or 2 as they are unreachable (cost greater than 15).

Table 5: Router 5 correctly not accepting unreachable routes (routes 1 and 2)

```
------------------------------
Forwarding Table for 5
Address | Next Hop | Cost | Timer | Change
-------------------------------------------
   4    |    4     |  2   |  0.82 | False
   5    |    5     |  0   |  0.00 | False
   7    |    4     |  8   |  0.82 | False
```

## Restarting a Router

Our final test was to restart a router after shutting it down. We start all routers in the example network, then stop router 1. We wait for the network to reconverge, then start up router 1 again. It is expected that router 1 can fairly quickly establish routes to the rest of the network, as it just needs to get updates from its neighbours. In practice it took 33 seconds for router 1 to converge to the pre-shutdown value, which makes sense given each neighbour is updating with a period of roughly 30s.

This means we can be confident our network can recover quickly from failures.

Note that it may take longer for other routers to use the new routes available through router 1, as they may have to wait for several update messages.

# Example Configuration File

The below code block shows the configuration file for router 1 in the example network.2.54

**config1.txt**

```
router-id 1
input-ports 10001, 10002, 10003
outputs 2001-1-2, 7001-8-7, 6001-5-6
route-timeout 180
periodic-update-time 30
garbage-time 120
```

# Source Code

Our code is composed of two files, ripd.py and parseutils.py.

**ripd.py**

```
"""
An implementation of the RIP routing protocol for COSC364 Assignment 1
Routes using router-ids instead of network addresses

Christopher Stewart (cst141) 21069553
Frederik Markwell (fma107) 51118501
"""

import socket, os, sys, select, time, random
from parseutils import parse_config_file

# Sets the maximum size packet that the router can receive
MAX_PACKET_SIZE = 4096

# Changes how the router prints out its table. If PRETTY, prints as often as
# possible, clearing the screen. If not, prints only when there is an update
# and does not clear the screen.
PRETTY = True

class Row():
    """
    Entry in the routers forwarding table (dictionary) where the key is the
    destination router_id. A new row is created whenever the route is updated
    """
    def __init__(self, cost, next_hop):
        self.cost = cost
        self.next_hop = next_hop
        self.last_response_time = time.time()
        self.timer = 0

        self.changed = True # Set false when a packet is sent containing this row
    def __str__(self):
        return '(cost:' + str(self.cost) + ', next_hop:' + str(self.next_hop) + ')'
    def __repr__(self):
        return str(self)



class RIP_Router():
    """
    The main router class. Calls self.run on init, which enters an infinite loop.
```

```
    The infinite loop sends an update message every 30 s, and waits for incoming
    messages, which it uses to update its forwarding table
    """
    # Dictionary with key=destination_router_id, value=Row object
    table = {}

    # List of sockets, each bound to one of the input_ports
    input_sockets = None

    # Local computer address
    address = 'localhost'

    # Router-id of running process
    instance_id = None

    # List of info on links to neighbour routers
    # Composed of tuples (output_port, cost, router_id)
    neighbour_info = None

    # Timer to keep track of whether a triggered update has been sent recently
    # Helps prevent network congestion
    triggered_update_timer = 0

    # If the triggered_update_timer reaches 0 and this is True, will send a triggered
update
    triggered_update_waiting = False


    def close(self):
        """
        Closes all sockets and exits the program
        """
        print("Closing")
        if self.input_sockets:
            for input_socket in self.input_sockets:
                input_socket.close()
        sys.exit()


    def __init__(self, filename):
        """
        Parses the provided configuration file and sets all configurable
        variables. Creates sockets, initial forwarding table, and then
        listens in a loop for other RIP daemons
        """
        (self.instance_id,
```

```
        input_ports,
        self.neighbour_info,
        self.timeout,
        self.periodic_update_time,
        self.garbage_time) = parse_config_file(filename)

        self.garbage_time += self.timeout

        self.init_input_ports(input_ports)

        #init table with own entry
        self.table[self.instance_id] = Row(0,self.instance_id)

        self.print_table()

        self.run()
        self.close()


    def init_input_ports(self, input_ports):
        """
        Creates a socket for each input port provided in the configuration file
        and binds them to localhost
        """
        self.input_sockets = []
        for rx_port in input_ports:
            try:
                rx_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
                rx_socket.bind((self.address, rx_port))
                self.input_sockets.append(rx_socket)
            except Exception as e:
                print("failed to create socket.", rx_port, e)
                self.close()

    def print_table(self):
        """
        Prints the forwarding table to the console
        """
        print("\n" + "-" * 30)
        print("Forwarding Table for {}".format(self.instance_id))
        headings = ["Address", "Next Hop", "Cost", "Timer", "Change"]
        print((" | ").join(headings))
        print("-" * sum(len(heading) + 3 for heading in headings))
        for dest, row in sorted(self.table.items(), key=lambda x: x[0]):
            timer = ""
            if dest in self.table.keys():
```

```
                timer = f"{self.table[dest].timer:.2f}"

            print("{} | {} | {} | {} | {}".format(
                str(dest).center(len(headings[0])),
                str(row.next_hop).center(len(headings[1])),
                str(row.cost).center(len(headings[2])),
                str(timer).center(len(headings[3])),
                str(row.changed).center(len(headings[4]))
            ))


    def create_response(self, destination, triggered):
        """
        Creates a RIP response packet for the destination router in
        the below format

        command(1) - version(1) - router_id(2)  #header(4)

        addr_family_id(2) - zero(2)             #each entry (20)
        ipv4_addr(4)
        zero(4)
        zero(4)
        metric(4)
        """
        command = int(2).to_bytes(1, 'big')
        version = int(2).to_bytes(1, 'big')
        router_id = int(self.instance_id).to_bytes(2, 'big')
        zero2 = int(0).to_bytes(2, 'big')
        header = command + version + router_id #header uses router_id instead of 16bit
zero

        payload = bytes()
        for router_id in self.table.keys(): # for each destination router_id
            if not triggered or self.table[router_id].changed:
                # Only send all routes if not triggered update

                addr_family_id = int(2).to_bytes(2, 'big') # 2 = AF_INET
                ipv4_addr = int(router_id).to_bytes(4, 'big') # next_hop is the router
sending the response packet
                zero4 = int(0).to_bytes(4, 'big')


                if self.table[router_id].next_hop == destination:
                    # Route goes through the router we are sending to, should poison
                    metric = int(16).to_bytes(4, 'big')
                else:
```

```
                # No need to poison
                metric = int(self.table[router_id].cost).to_bytes(4, 'big')
            payload += addr_family_id + zero2 + ipv4_addr + zero4 + zero4 + metric

        result = header + payload
        return bytearray(result)

    def send_response(self, addr_id, addr_port, triggered):
        """
        Creates and sends a response / triggered update to a specific router
        """
        packet = self.create_response(addr_id, triggered)
        target = (self.address, addr_port)
        self.input_sockets[0].sendto(bytes(packet),target)

    def send_all_responses(self, triggered=False):
        """
        Iterates all neighbour ports, and sends a response / triggered update to each
        """

        # If we send a normal message, we don't need to send a triggered update later
        self.triggered_update_waiting = False

        for output_port, cost, id in self.neighbour_info:
            self.send_response(id, output_port, triggered)

        # Routes are no longer considered "new" once we have sent them out
        for row in self.table.values():
            row.changed = False

    def read_response(self,data):
        """
        Converts the received packet to a table if it follows the correct format
        returns (packet_valid(bool), router_id(int), table(dict))
        """
        command = data[0]
        version = data[1]
        if command != 2 or version !=2:
            print("invalid command/version",command,version)
            return False, 0, 0 # command or version value is incorrect

        router_id = int.from_bytes(data[2:4], 'big') # router(id) that sent the data

        i = 4 # packet payload (RIP entries) starts after 4 bytes
        if (len(data)-4) % 20 != 0 or len(data) <= 4:
            print("invalid packet length", len(data))
```

```
            return False,0,0 # data length incorrect (should be 4 + 20x) where x > 0

        recvd_table = {}
        while i < len(data):
            try:
                zeros = []#append all expected zero values here to validate packet
                addr_family_id = int.from_bytes(data[i:i+2], 'big')
                i+=2
                zeros.append(int.from_bytes(data[i:i+2], 'big'))#zero2
                i+=2
                ipv4_addr = int.from_bytes(data[i:i+4], 'big')#dest addr from the
router sending
                i+=4
                zeros.append(int.from_bytes(data[i:i+4], 'big'))#zero4
                i+=4
                zeros.append(int.from_bytes(data[i:i+4], 'big'))#zero4
                i+=4
                metric = int.from_bytes(data[i:i+4], 'big')# between 1-15 (inclusive)
or 16 (inf)
                row = Row(metric, router_id)#cost, next_hop
                recvd_table[ipv4_addr] = row
                i+=4

                if min(zeros) != 0 or max(zeros) != 0 or metric < 0 or metric > 16:
                    print("invalid RIP ENTRY format", zeros,metric)
                    return False,0,0#bad RIP entry
            except IndexError:
                print("index error", i, len(data))
                return False,0,0#data length incorrect (should be 4 + 20x)
        return True, router_id, recvd_table

    def cost_to_neighbour(self, router_id):
        """
        Calculates cost to travel to a particular neighbouring router
        """
        neighbour_ids = [x[2] for x in self.neighbour_info]
        cost = self.neighbour_info[neighbour_ids.index(router_id)][1]
        return cost

    def update_table(self, other_router_id, other_table):
        """
        Compares tables with a received table and updates if
            1. A route is better than the current route
                or
            2. The route comes from the router from which the the old route
                came (here called the authority)
```

```
        When a route is updated, the timer on that route is also updated. Timers
        are not updated if the authority repeatedly reports that the cost is 16
        (so that route may timeout)
        """
        cost = self.cost_to_neighbour(other_router_id)
        for dest in other_table.keys():
            other_row = other_table[dest]

            try:
                current_row = self.table[dest]
                from_authority = self.table[dest].next_hop == other_router_id
                cost_changed = current_row.cost != min(16, other_row.cost + cost)

                if from_authority:
                    # Our current route comes from this router, so must take their
value
                    if cost_changed:
                        # Change our table to match the authority
                        self.update_row(dest, cost, other_row, other_router_id)
                        if cost + other_row.cost >= 16:
                            self.triggered_update_waiting = True
                    elif current_row.cost != 16:
                        # Resets the timer for reachable routes (to keep it alive)
                        self.table[dest].last_response_time = time.time()
                        self.table[dest].timer = 0.00
                elif current_row.cost > (other_row.cost + cost):
                    # The current route is less optimal than the jump to the neighbour
+ the neighbours route
                    self.update_row(dest, cost, other_row, other_router_id)

            except KeyError: # We currently do not have a route to this dest
                if other_table[dest].cost + cost < 16: # Ignore routes with cost > 16
                    self.update_row(dest, cost, other_row, other_router_id)

        self.print_table()

    def update_row(self, dest, cost, other_row, other_router_id):
        """
        Replaces an existing route with a route from a row in another table and
        resets the timer on that route
        """

        row = Row(min(16, other_row.cost + cost), other_router_id)
        self.table[dest] = row

        self.table[dest].last_response_time = time.time()
```

```
        self.table[dest].timer = 0.00


    def update_table_timers(self):
        """
        Add time waited to each routes timer, if necessary timeout or delete the route
        """
        routes_to_del = []
        for key in self.table.keys():
            if key != self.instance_id:#don't increase timer of own route
                self.table[key].timer = time.time() - 
self.table[key].last_response_time#update routes timer
                if self.table[key].timer > self.timeout and self.table[key].cost != 
16:#route timed out
                    self.table[key].cost = 16
                    self.table[key].changed = True
                    self.triggered_update_waiting = True
                    self.print_table()
                if self.table[key].timer > self.garbage_time:#route deleted
                    routes_to_del.append(key)

        # We delete all the routes together so that we don't alter the indices while 
looping through
        for route in routes_to_del:
            del self.table[route]#removes entry from table

        if len(routes_to_del) > 0:#if a route is deleted due to garbage collection, 
print updated table
            self.print_table()




    def run(self):
        """
        Enters an infinite loop in which the router reacts to incoming events
        An incoming event is either:
            a routing packet received from a peer
            a timer event
        """

        inputs = [x.fileno() for x in self.input_sockets]

        self.send_all_responses()

        random_range = self.periodic_update_time * 0.4
```

```
        response_timer = self.periodic_update_time

        while True:
            try:

                start = time.time()

                rlist, wlist, xlist = select.select(inputs, [], [], 0.1 if PRETTY else
0.01)


                if response_timer <= 0:
                    response_timer = self.periodic_update_time +
(random.random()*random_range) - random_range / 2
                    self.send_all_responses()
                    self.print_table()


                self.update_table_timers()


                if self.triggered_update_timer == 0 and self.triggered_update_waiting:
                    self.send_all_responses(triggered=True)
                    self.triggered_update_waiting = False
                    self.triggered_update_timer = 1 + random.random() * 4
                    print("Sent a triggered update!")


                '''reads responses (if any) from neighbours and updates tables'''
                for socket_id in rlist:
                    sock = socket.fromfd(socket_id,socket.AF_INET, socket.SOCK_DGRAM)
                    data = sock.recv(MAX_PACKET_SIZE)
                    packet_valid, other_router_id, other_table =
self.read_response(data)
                    print("Received packet from", other_router_id)
                    if packet_valid:
                        self.update_table(other_router_id, other_table)
                    else:
                        print("invalid packet")


                end = time.time()
                delta_time = end - start

                response_timer = max(0, response_timer - delta_time)
```

```
                self.triggered_update_timer = max(0, self.triggered_update_timer -
delta_time)

                if PRETTY:
                    os.system("clear")
                    self.print_table()


            except Exception as e:
                print("An unexpected error occurred [{}]".format(e))
                self.close()
        self.close()


def main():
    arguments = sys.argv[1:]
    if len(arguments) != 1:
        print("Invalid arguments given, must include the directory of a valid
configuration file")
        sys.exit()
    filename = arguments[0]
    router = RIP_Router(filename)



main()
```

**parseutils.py**

```
"""
A collection of functions used in parsing the router config files

Christopher Stewart (cst141) 21069553
Frederik Markwell (fma107) 51118501
"""

import sys

def read_lines_from_file(filename):
    """
    Opens a file and reads the lines, returning a list of strings.
    If the file cannot be found or there is an error, prints a message then calls
    sys.exit
    """
    try:
        with open(filename, 'r') as config_file:
            return config_file.read().splitlines()
    except FileNotFoundError:
        print("Couldn't find", filename)
        sys.exit()
    except OSError:
        print("Error opening file")
        sys.exit()

def is_valid_int(val, min, max, name):
    """
    Checks that the given string is a valid integer between min and max
    If it is not, prints a message using name, and then calls sys.exit
    """

    if val.isdigit():
        val = int(val)
        if val >= min and val <= max:
            return True
        else:
            print(val, "is not a valid {} (must be between {} and {})".format(name,
min, max))
            sys.exit()
    else:
        print(val, "is not a valid {} (non-integer)".format(name))
        sys.exit()

def is_valid_port(port, existing_ports):
    """
```

```
    Checks that the given value is an integer, within the acceptable range,
    and not in the list of existing input_ports. Calls sys.exit if not.
    """
    if is_valid_int(port, 1024, 64000, "port") and int(port) not in existing_ports:
        return True
    print(port, "is a duplicate")
    sys.exit()

def is_valid_link(link, existing_ports):
    """
    Checks if a link (port-metric-id) is valid and formatted correctly. If not,
    calls sys.exit
    """
    try:
        port, cost, id = link.strip().split("-")
    except ValueError:
        print(link, "does not follow the format (port-cost-id)")
        sys.exit()

    # The below functions call sys.exit if not valid
    is_valid_port(port, existing_ports)
    is_valid_int(cost, 1, 16, "cost")
    is_valid_int(id, 1, 64000, "id")

    return True

def parse_config_file(filename):
    """
    Reads a file as described in the assignment description and returns a tuple
    with instance_id, input_ports, neighbour_info, and the timeout values
    """
    lines = read_lines_from_file(filename)


    id_set = False
    inputs_set = False
    outputs_set = False

    input_ports = []
    neighbour_info = []
    timeout = 180
    periodic_update_time = 30
    garbage_time = 120

    for line in lines:
        line = line.strip()
```

```
        line = line.split("#", 1)[0]
        if "router-id" in line:
            id = line.split()[1]
            if is_valid_int(id, 1, 64000, "router_id"):
                instance_id = int(id)
                id_set = True

        elif "input-ports" in line:
            for port in line[len("input-ports"):].split(","):
                port = port.strip()
                if is_valid_port(port, input_ports):
                    input_ports.append(int(port))
                    inputs_set = True

        elif "outputs" in line:
            output_ports = [] # Keeps track so we can check for duplicates
            for link in line[len("outputs "):].split(","):
                if is_valid_link(link, input_ports + output_ports):
                    port, cost, id = [int(x) for x in link.split("-")]

                neighbour_info.append((port, cost, id))
                output_ports.append(port)
            outputs_set = True
        elif "route-timeout" in line:
            if is_valid_int(line.split()[1], 1, float('inf'), "route timeout"):
                timeout = int(line.split()[1])
        elif "periodic-update-time" in line:
            if is_valid_int(line.split()[1], 1, float('inf'), "periodic update time
timeout"):
                periodic_update_time = int(line.split()[1])
        elif "garbage-time" in line:
            if is_valid_int(line.split()[1], 1, float('inf'), "garbage time timeout"):
                garbage_time = int(line.split()[1])
        elif "" == line:
            pass
        else:
            print("Could not process", line)
            sys.exit()



    if not all((id_set, inputs_set, outputs_set)):
        print("Need all of router-id, input-ports, outputs")
        sys.exit()
    return instance_id, input_ports, neighbour_info, timeout, periodic_update_time,
garbage_time
```

# 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 adapted, 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 total loss of marks, no grade being awarded and other serious consequences, including notification of the University Proctor.

Name: Frederik Markwell

Student ID: 51118501

Signature:

Date: 1/05/22

# 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 adapted, 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 total loss of marks, no grade being awarded and other serious consequences, including notification of the University Proctor.

Name: Christopher Stewart

Student ID: 21069553

Signature:

Date: 01/05/22