Rory Varcoe	83048685
Carl Kenny	93678486

COSC 364

RIPv2

Rory Varcoe 83048685
Carl Kenny 93678486

Contents

Implementation

We believe that the ByteArray class was exceptionally well executed and actually represents bytes, rather than the easier option (Strings). Even though we implemented RIPv2 using the Python programming language, this allowed us to achieve a reasonably low level via sockets and bytes. This the ByteArray class being setup as it is, it allowed us to efficiently propagate updates from the network, with minimal words sent.

The config parser, utilizes Python's configparser library and tidily processes the .ini format config files. We specifically choose this format, as it plays well with both *nix and Windows.

Improvement

Although the program is functional, it required use of both Socket and SocketServer Python libraries. As this assignment didn't have strict space requirements, it isn't too inferior but it would have been preferred to only use one. We spent a lot of time attempting to communicate solely through the socket interface but weren't successful. SocketServer is quite a large library, however it did provide use with a useful multi-thread interface.

Atomicity of Event Processing

Atomicity was achieved by simple careful consideration of the read, update and write steps. The SocketServer framework allows four classes to process requests asynchronously, so that no data is accidentally overwritten or unexpected race conditions. This framework also offers the ability to designate a specific thread for each server.

Tests and Bash Scripts

The following bash script allows all the routers to be initialized, in-order to check for basic correctness (each router sending routing tables of reachable neighbours etc.)

```
#!/bin/sh
xterm -title "Router 1" -e "python3 run.py config_1.ini" &
xterm -title "Router 2" -e "python3 run.py config_2.ini" &
xterm -title "Router 3" -e "python3 run.py config_3.ini" &
xterm -title "Router 4" -e "python3 run.py config_4.ini" &
xterm -title "Router 5" -e "python3 run.py config_5.ini" &
xterm -title "Router 6" -e "python3 run.py config_6.ini" &
xterm -title "Router 7" -e "python3 run.py config_7.ini"
```

While this is handy for initially setting up all the routers, manual testing of Split Horizon and dying/reviving had to be done. We originally tried to use the following *nix commands, to locate the process id and kill it, however this approach was annoyingly unsuccessful.

```
$ps -ef | grep "python3 run.py config_1.ini" | awk '{print $2}' | xargs kill
```

Upon expiration of the timeout, the entry for that route stays in the routing table for a brief moment, while the neighbours are notified. This was manually inspected via Python print statements, to ensure that neighbours receive update before the route is dropped from the corresponding dictionary.

Syntactically incorrect config files while cause the Python script to sys.exit() and abort with a relevant error message. Abrupt sys.exit() were also used during the development and testing stages, to ensure that that routes weren't learned from one neighbour when sending updates to that neighbour. When we directly found cases of this, we were able to implement Split Horizon with poisoned reserve i.e. setting their metrics to 16 (infinity).

During testing on a windows machine, we found that port 5004 was occupied by Windows Media Player network and caused some serious strife when setting up the routes, only to find router 5 was non-reachable. In the usual 'it works on my computer' philosophy, one of us was unable to replicate this result, resulting in a frantic debugging VoIP call. “Port used by Real Time Streaming Protocol (RTSP) for Microsoft Windows Media streaming services.” was the delight of many as we quickly reconfigured the ports.

Source Code

config_parser.py

```python
import configparser
import re
import sys

"""
Author: Carl Kenny
Date:   10/03/2015
Prog:   Parses router config (.ini) and provides basic error checking
"""


def parse_config(config):
    """
   :param config: ConfigParser() object
   :return dict: Contains: router-id, input-ports and outputs
   """
    config_dict = {}

    router_id = int(config['ROUTER']['router-id'])

    if 1 <= router_id < 64000:
        config_dict['router-id'] = router_id
    else:
        return None

    ports = set()
    ports_split = config['ROUTER']['input-ports'].split(',')
    port_count = len(ports_split)

    for port in ports_split:
        port = int(port)
        if 1024 <= port <= 64000:
            ports.add(port)
        else:
            return None

    config_dict['input-ports'] = ports
    output_split = config['ROUTER']['outputs'].split(',')
    outputs = []

    for output in output_split:
        re_result = re.search("(.*)-(.)-(.)", output).groups()
        output_port, metric, router_id  = [int(i) for i in re_result]
        if 1024 <= output_port <= 64000 and output_port not in config_dict['input-
ports']:
            outputs.append([output_port, metric, router_id])
        else:
            return None

    config_dict['outputs'] = outputs

    return config_dict if port_count == len(ports) else None
```

packet.py

```
class ByteArray(object):
  def __init__(self, data=None):
    if (data is None):
      self.data = []
    else:
      self.data = data
    self.pointer = 0

  def is_empty(self):
    return (len(self.data) - self.pointer) <= 0

  def size(self):
    return (len(self.data) - self.pointer)

  def __str__(self):
    return "".join(format(x, '02x') for x in self.data)

  def set_pointer(self, pointer):
    self.pointer = pointer

  def get_pointer(self):
    return self.pointer

  def set_data(self, data):
    self.data = data

  def get_data(self):
    return self.data

  def insert_byte(self, byte):
    assert byte >= 0 and byte < 256
    bytes = byte.to_bytes(1, byteorder='big')
    self.data.extend(bytes)

  def insert_word(self, word):
    assert word >= 0 and word < 65536
    bytes = word.to_bytes(2, byteorder='big')
    self.data.extend(bytes)

  def insert_dword(self, dword):
    assert dword >= 0 and dword < 4294967296
    bytes = dword.to_bytes(4, byteorder='big')
    self.data.extend(bytes)

  def peek_byte(self, offset=0):
    byte = self.data[self.pointer + offset]
    bytes = [byte]
    return int.from_bytes(bytes, byteorder='big')

  def peek_word(self, offset=0):
    index = self.pointer + offset
    bytes = self.data[index:index + 2]
    return int.from_bytes(bytes, byteorder='big')

  def peek_dword(self, offset=0):
    index = self.pointer + offset
    bytes = self.data[index:index + 4]
    return int.from_bytes(bytes, byteorder='big')

  def get_byte(self):
    byte = self.peek_byte()
    self.pointer += 1
    return byte
```

```

 def get_word(self):
   word = self.peek_word()
   self.pointer += 2
   return word

 def get_dword(self):
   dword = self.peek_dword()
   self.pointer += 4
   return dword
```

router.py

```python
from threading import Timer, Thread
import time, struct, socket, random
import socketserver
import packet

class ThreadedUDPRequestHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request[0]
        self.server.router.incoming_update(data)

class ThreadedUDPServer(socketserver.ThreadingMixIn, socketserver.UDPServer):
    """ Overrides original handler """
    pass

class Route(object):
  MIN_HOPS = 0
  MAX_HOPS = 16

  def __init__(self, address=None, next_address=None, hops=0, afi=2):
    """
   Initialize the Route class. Default paramaters are provided, if not set.
   :param address
   :param next_address
   :param hops int
   :param afi int
   """
    self.address = address            # Destination address
    self.next_address = next_address  # The next address (usually who we recieved info from)
    self.hops = hops                  # Distance to get to Destination address from Next address
    self.afi = afi                    # AFI
    self.next_cost = 1                # Cost to get to the next address
    self.marked = False
    self.marked_time = 0

  def mark(self):
    self.marked = True
    self.marked_time = time.time()

  def set_next_cost(self, cost):
    self.next_cost = cost

  def cost(self):
    c = self.hops + self.next_cost
    if (c > 16): c = 16
    return c

  def __repr__(self):
    """
   Provides the string representation of the class, to be used if directly
   printed, e.g. print(Route).
   """
    marked_token = ""
    if (self.marked):
      marked_token = " [MARKED FOR DELETION at " + time.strftime("%X", time.localtime(self.marked_time)) + "]"
    return "Route(dest:" + str(self.address) + ", next: " + str(self.next_address) + " (cost: " + str(self.cost()) + "))" + marked_token


def if_failed(condition, message):
  """
```

```
 :param condition  bool
 :param message    str
 """

  if (not bool(condition)):
    print("[WARNING] Check Failed: " + str(message))
    return True
  return False

class Router(object):
  TIMER_TICK = 5.0
  NEIGHBOR_TIMEOUT = 30.0
  DELETION_TIMEOUT = 7.5

  def __init__(self, config):
    """ Initialize the Router class. Config param required """
    self.routes = dict()
    self.config = config

    # Load configuration
    print(self.config)

    self.neighbors = dict()
    outputs = self.config["outputs"]

    for output in outputs:
      port = output[0]
      metric = output[1]
      router_id = output[2]
      last_updated = time.time() # Current time (in seconds)
      self.neighbors[router_id] = [port, metric, last_updated]

    for port in self.config["input-ports"]:
      server = ThreadedUDPServer(("localhost", port), ThreadedUDPRequestHandler)  #
127.0.0.1
      server.router = self
      server_thread = Thread(target=server.serve_forever)
      server_thread.daemon = True
      server_thread.start()
      print("Listening on port " + str(port) + " for datagrams...")

    # Load entry for self and initialize metric to 0
    router_id = self.config["router-id"]
    self.router_id = self.config["router-id"]
    route = Route(router_id, router_id, 0)
    route.next_cost = 0
    self.id = router_id
    self.routes[router_id] = route

    # That's all, now we start!
    self.print_table()
    self._start_timer()

  def get_neighbor_port(self, router_id):
    return self.neighbors[router_id][0] # port

  def get_neighbor_metric(self, router_id):
    return self.neighbors[router_id][1] # metric

  def get_neighbor_updated(self, router_id):
    return self.neighbors[router_id][2] # last_updated

  def set_neighbor_updated(self, router_id):
    """
   Resets metric to known value if update and metric >= 16
   :param router_id int
```

```python
   :return void
   """
    self.neighbors[router_id][2] = time.time()

    if (self.get_neighbor_metric(router_id) >= 16):
      outputs = self.config["outputs"]
      for output in outputs:
        if (output[2] == router_id):
          self.neighbors[router_id][1] = output[1]
          break

  def incoming_route(self, route):
    """
   Adjust routes depending on cost of incoming route and whether we already know the route
   :param route
   :return void
   """
    cost = self.get_neighbor_metric(route.next_address)
    route.set_next_cost(cost)

    # Check if we have this route already
    if (route.address in self.routes.keys()):
      # We have that route in the table already
      old_route = self.routes[route.address]

      if old_route.next_address == route.next_address:
        # Sanity check, if it's marked we already know it's unreachable
        if (route.cost() != old_route.cost()):
          self.routes[route.address] = route

      elif route.cost() < old_route.cost():
        # Store new value
        self.routes[route.address] = route
      else:
        return

    elif route.cost() < 16:
      # The route is new, and under 16
      self.routes[route.address] = route

    else:
      # The route is new but is marked for deletion.
      # Seems odd, our neighbors should get this anyway and if not then they won't have this route
      # anyway, since we don't have it either
      return

    if route.cost() >= 16 and not route.marked and \
         route.next_address != self.id and route.address != self.id:
        route.mark()
        # Force an update of all routers
        self.update()

  def incoming_update(self, raw_data):
    """
   Provides checks of packet length, version, command and non-neighbors
   :param raw_data data received (bytes)
   """
    data = packet.ByteArray(raw_data)
    if if_failed(data.size() >= 4, "Recieved invalid packet of length " + str(data.size())):
      return

    command = data.get_byte()
    if if_failed(command == 2, "Recieved RIPv2 request (expected only responses)"):
      return
    version = data.get_byte()
    if if_failed(version == 2, "Recieved RIPv1 or other message (expected RIPv2)"):
      return
    from_id = data.get_word()

    if if_failed(from_id in self.neighbors, "Recieved update from non-neighbor router"):
      return

    self.set_neighbor_updated(from_id)

    while (not data.is_empty()):
      afi = data.get_word()       # AF_INET (2)
      if if_failed(afi == 2, "Recieved unknown AF_INET (expected 2)"):
        return
      data.get_word()             # (BLANK) should we check for this being 0? Probably not important.
      address = data.get_dword()  # Router-ID
      data.get_dword()            # (BLANK)
      data.get_dword()            # (BLANK)
      hops = data.get_dword()     # 1-15 inclusive, or 16 (infinity)
      if if_failed(0 <= hops <= 16, "Recieved invalid hop count (setting to 16)"):
        hops = 16

      r = Route(address, from_id, hops, afi)
      self.incoming_route(r)


  def _start_timer(self):
    delay = Router.TIMER_TICK * random.uniform(0.8, 1.2)
    self.timer = Timer(delay, Router.tick, args=[self])
    self.timer.start()

  def print_table(self):
    """
   Prints a human readable representation of the routes in routing table
   """
    now = time.strftime("%X")
    print("[{}] Routing Table for {}".format(str(now), self.router_id))
    for route_id in self.routes.keys():
      route = self.routes[route_id]
      print("\t\t", route)

  def send_updates(self):
    for neighbor in self.neighbors.keys():
        if (self.get_neighbor_metric(neighbor) < 16):
          self.send_update_to_router(neighbor)


  def send_update_to_router(self, router_id):
    # Sends a specialized update message to a neighbor using split-horizon poisoning
    # Skip sending updates to self
    if (router_id == self.id):
      return
    entries = []
    for route_id in self.routes.keys():
      route = self.routes[route_id]
      new_metric = route.cost()
      if (route.address == router_id or route.next_address == router_id):
        # Split-horizon poisoning
        new_metric = 16

        # Skip marked messages from contacting the router who sent us the deletion message
        if (route.marked): continue

      entries.append({"afi": 2, "address": route.address, "metric": new_metric})

    data = self.build_packet(self.id, entries).get_data()
    port = self.get_neighbor_port(router_id)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(bytes(data), ("localhost", port))

  def update(self):
    """
    Check for non-responsive neighbors
    :return void
    """

    now = time.time()
    for router_id in self.neighbors.keys():
      last = self.get_neighbor_updated(router_id)
      if last + Router.NEIGHBOR_TIMEOUT < now and self.get_neighbor_metric(router_id) < 16:
        print("Router #" + str(router_id) + " has not responded. Setting metric to 16...")
        self.neighbors[router_id][1] = 16

    # Check for marked routes ready for deletion
    # Update inaccessible routes with 16
    for route_id in self.routes.keys():
      route = self.routes[route_id]
      if (route.address in self.neighbors and self.get_neighbor_metric(route.address) >= 16 or \
          route.next_address in self.neighbors and self.get_neighbor_metric(route.next_address) >= 16):
        # If it's a neighbor, check to see if our internal metric is 16.
        # If it is, replace our "route" metrics with 16
        route.next_cost = 16
        # Mark for deletion in a little bit...
        if (not route.marked):
          print("Marking route to " + str(route.address) + " for deletion (neighbor metric is 16)")
          route.mark()

      if (route.marked and route.marked_time + Router.DELETION_TIMEOUT < now):
        print("Deleting marked route " + str(route) + " as inaccessible")
        self.routes[route_id] = None

    self.routes = { k : v for k,v in self.routes.items() if v is not None }

    # If it's time, send an update ourselves (this handles a metric of 16 for the above!)
    self.send_updates()
    self.print_table()

  def tick(self):
    self._start_timer()
    self.update()

  def build_packet(self, sender_id, entries, command=2, version=2):
    # (Byte) Command
    # (Byte) Version
    # (Word) Padding 0x0 (or Sender-ID in COSC364's case)
    # (Void) Entries (20 bytes each, 1-25 entries)
    datagram = packet.ByteArray()
    datagram.insert_byte(command)
    datagram.insert_byte(version)

    # COSC364 special: Use the sending router-id here
    datagram.insert_word(sender_id)
    # Limited entries to 25?
    if if_failed(len(entries) <= 25, "Recieved invalid entries count (larger than 25)"):
      return datagram

    for i, item in enumerate(entries):
      datagram.insert_word(item["afi"]) # AF_INET (2)
      datagram.insert_word(0)
      datagram.insert_dword(item["address"]) # IPv4
      datagram.insert_dword(0)
      datagram.insert_dword(0)
      datagram.insert_dword(item["metric"]) # 1-15 inclusive, or 16 (infinity)

    return datagram
```

run.py

```
import config_parser
import configparser
import sys
from router import Router


def main():
    if len(sys.argv) < 2:
        sys.exit("No file given")

    # closes file when done, file like object
    with open(sys.argv[1]) as fp:
        config = configparser.ConfigParser()
        config.readfp(fp)
        config_dict = config_parser.parse_config(config)

        if not config_dict:
            sys.exit("Invalid config file. Exiting...")
        print("Loaded config file. Starting router...")
        x = Router(config_dict)

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

Configuration Files

```
# config_1.ini
[ROUTER]
router-id = 1
input-ports =  1102, 1103, 1104, 1105, 1106, 1107
outputs = 2001-1-2, 6001-5-6, 7001-8-7

# config_2.ini
[ROUTER]
router-id = 2
input-ports = 2001, 2003, 2004, 2005, 2006, 2007
outputs = 1102-1-1, 3002-3-3

# config_3.ini
[ROUTER]
router-id = 3
input-ports = 3001, 3002, 3004, 3005, 3006, 3007
outputs = 2003-3-2, 4003-4-4

# config_4.ini
[ROUTER]
router-id = 4
input-ports = 4001, 4002, 4003, 4005, 4006, 4007
outputs = 3004-4-3, 5004-2-5, 7004-6-7

# config_5.ini
[ROUTER]
router-id = 5
input-ports = 5001, 5002, 5003, 5004, 5006, 5007
outputs = 4005-2-4, 6005-1-6

# config_6.ini
[ROUTER]
router-id = 6
input-ports = 6001, 6002, 6003, 6004, 6005, 6007
outputs = 1106-5-1, 5006-1-5

# config_7.ini
[ROUTER]
router-id =  7
input-ports =  7001, 7002, 7003, 7004, 7005, 7006
outputs =  1107-8-1, 4007-6-4
```