COSC 364

RIPv2

Contents

[Implementation 3](#_Toc419495487)

[Improvement 3](#_Toc419495488)

[Atomicity of Event Processing 3](#_Toc419495489)

[Tests and Bash Scripts 4](#_Toc419495490)

[Source Code 5](#_Toc419495491)

[Configuration Files 15](#_Toc419495492)

# 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

1. **import** configparser
2. **import** re
3. **import** sys
5. """
6. Author: Carl Kenny
7. Date:   10/03/2015
8. Prog:   Parses router config (.ini) and provides basic error checking
9. """

12. **def** parse\_config(config):
13. """
14. :param config: ConfigParser() object
15. :return dict: Contains: router-id, input-ports and outputs
16. """
17. config\_dict = {}
19. router\_id = int(config['ROUTER']['router-id'])
21. **if** 1 <= router\_id < 64000:
22. config\_dict['router-id'] = router\_id
23. **else**:
24. **return** None
26. ports = set()
27. ports\_split = config['ROUTER']['input-ports'].split(',')
28. port\_count = len(ports\_split)
30. **for** port **in** ports\_split:
31. port = int(port)
32. **if** 1024 <= port <= 64000:
33. ports.add(port)
34. **else**:
35. **return** None
37. config\_dict['input-ports'] = ports
38. output\_split = config['ROUTER']['outputs'].split(',')
39. outputs = []
41. **for** output **in** output\_split:
42. re\_result = re.search("(.\*)-(.)-(.)", output).groups()
43. output\_port, metric, router\_id  = [int(i) **for** i **in** re\_result]
44. **if** 1024 <= output\_port <= 64000 **and** output\_port **not** **in** config\_dict['input-ports']:
45. outputs.append([output\_port, metric, router\_id])
46. **else**:
47. **return** None
49. config\_dict['outputs'] = outputs
51. **return** config\_dict **if** port\_count == len(ports) **else** None

packet.py

1. **class** ByteArray(object):
2. **def** \_\_init\_\_(self, data=None):
3. **if** (data **is** None):
4. self.data = []
5. **else**:
6. self.data = data
7. self.pointer = 0
9. **def** is\_empty(self):
10. **return** (len(self.data) - self.pointer) <= 0
12. **def** size(self):
13. **return** (len(self.data) - self.pointer)
15. **def** \_\_str\_\_(self):
16. **return** "".join(format(x, '02x') **for** x **in** self.data)
18. **def** set\_pointer(self, pointer):
19. self.pointer = pointer
21. **def** get\_pointer(self):
22. **return** self.pointer
24. **def** set\_data(self, data):
25. self.data = data
27. **def** get\_data(self):
28. **return** self.data
30. **def** insert\_byte(self, byte):
31. **assert** byte >= 0 **and** byte < 256
32. bytes = byte.to\_bytes(1, byteorder='big')
33. self.data.extend(bytes)
35. **def** insert\_word(self, word):
36. **assert** word >= 0 **and** word < 65536
37. bytes = word.to\_bytes(2, byteorder='big')
38. self.data.extend(bytes)
40. **def** insert\_dword(self, dword):
41. **assert** dword >= 0 **and** dword < 4294967296
42. bytes = dword.to\_bytes(4, byteorder='big')
43. self.data.extend(bytes)
45. **def** peek\_byte(self, offset=0):
46. byte = self.data[self.pointer + offset]
47. bytes = [byte]
48. **return** int.from\_bytes(bytes, byteorder='big')
50. **def** peek\_word(self, offset=0):
51. index = self.pointer + offset
52. bytes = self.data[index:index + 2]
53. **return** int.from\_bytes(bytes, byteorder='big')
55. **def** peek\_dword(self, offset=0):
56. index = self.pointer + offset
57. bytes = self.data[index:index + 4]
58. **return** int.from\_bytes(bytes, byteorder='big')
60. **def** get\_byte(self):
61. byte = self.peek\_byte()
62. self.pointer += 1
63. **return** byte
65. **def** get\_word(self):
66. word = self.peek\_word()
67. self.pointer += 2
68. **return** word
70. **def** get\_dword(self):
71. dword = self.peek\_dword()
72. self.pointer += 4
73. **return** dword

router.py

1. **from** threading **import** Timer, Thread
2. **import** time, struct, socket, random
3. **import** socketserver
4. **import** packet
6. **class** ThreadedUDPRequestHandler(socketserver.BaseRequestHandler):
7. **def** handle(self):
8. data = self.request[0]
9. self.server.router.incoming\_update(data)
11. **class** ThreadedUDPServer(socketserver.ThreadingMixIn, socketserver.UDPServer):
12. """ Overrides original handler """
13. **pass**
15. **class** Route(object):
16. MIN\_HOPS = 0
17. MAX\_HOPS = 16
19. **def** \_\_init\_\_(self, address=None, next\_address=None, hops=0, afi=2):
20. """
21. Initialize the Route class. Default paramaters are provided, if not set.
22. :param address
23. :param next\_address
24. :param hops int
25. :param afi int
26. """
27. self.address = address            *# Destination address*
28. self.next\_address = next\_address  *# The next address (usually who we recieved info from)*
29. self.hops = hops                  *# Distance to get to Destination address from Next address*
30. self.afi = afi                    *# AFI*
31. self.next\_cost = 1                *# Cost to get to the next address*
32. self.marked = False
33. self.marked\_time = 0
35. **def** mark(self):
36. self.marked = True
37. self.marked\_time = time.time()
39. **def** set\_next\_cost(self, cost):
40. self.next\_cost = cost
42. **def** cost(self):
43. c = self.hops + self.next\_cost
44. **if** (c > 16): c = 16
45. **return** c
47. **def** \_\_repr\_\_(self):
48. """
49. Provides the string representation of the class, to be used if directly
50. printed, e.g. print(Route).
51. """
52. marked\_token = ""
53. **if** (self.marked):
54. marked\_token = " [MARKED FOR DELETION at " + time.strftime("%X", time.localtime(self.marked\_time)) + "]"
55. **return** "Route(dest:" + str(self.address) + ", next: " + str(self.next\_address) + " (cost: " + str(self.cost()) + "))" + marked\_token

58. **def** if\_failed(condition, message):
59. """
60. :param condition  bool
61. :param message    str
62. """
64. **if** (**not** bool(condition)):
65. **print**("[WARNING] Check Failed: " + str(message))
66. **return** True
67. **return** False
69. **class** Router(object):
70. TIMER\_TICK = 5.0
71. NEIGHBOR\_TIMEOUT = 30.0
72. DELETION\_TIMEOUT = 7.5
74. **def** \_\_init\_\_(self, config):
75. """ Initialize the Router class. Config param required """
76. self.routes = dict()
77. self.config = config
79. *# load configuration*
80. **print**(self.config)
82. self.neighbors = dict()
83. outputs = self.config["outputs"]
85. **for** output **in** outputs:
86. port = output[0]
87. metric = output[1]
88. router\_id = output[2]
89. last\_updated = time.time() *# Current time (in seconds)*
90. self.neighbors[router\_id] = [port, metric, last\_updated]
92. **for** port **in** self.config["input-ports"]:
93. server = ThreadedUDPServer(("localhost", port), ThreadedUDPRequestHandler)  *# 127.0.0.1*
94. server.router = self
95. server\_thread = Thread(target=server.serve\_forever)
96. server\_thread.daemon = True
97. server\_thread.start()
98. **print**("Listening on port " + str(port) + " for datagrams...")
100. *# Load entry for self and initialize metric to 0*
101. router\_id = self.config["router-id"]
102. self.router\_id = self.config["router-id"]
103. route = Route(router\_id, router\_id, 0)
104. route.next\_cost = 0
105. self.id = router\_id
106. self.routes[router\_id] = route
108. *# That's all, now we start!*
109. self.print\_table()
110. self.\_start\_timer()
112. **def** get\_neighbor\_port(self, router\_id):
113. **return** self.neighbors[router\_id][0] *# port*
115. **def** get\_neighbor\_metric(self, router\_id):
116. **return** self.neighbors[router\_id][1] *# metric*
118. **def** get\_neighbor\_updated(self, router\_id):
119. **return** self.neighbors[router\_id][2] *# last\_updated*
121. **def** set\_neighbor\_updated(self, router\_id):
122. """
123. Resets metric to known value if update and metric >= 16
124. :param router\_id int
125. :return void
126. """
127. self.neighbors[router\_id][2] = time.time()
129. **if** (self.get\_neighbor\_metric(router\_id) >= 16):
130. outputs = self.config["outputs"]
131. **for** output **in** outputs:
132. **if** (output[2] == router\_id):
133. self.neighbors[router\_id][1] = output[1]
134. **break**
136. **def** incoming\_route(self, route):
137. """
138. Adjust routes depending on cost of incoming route and whether we already know the route
139. :param route
140. :return void
141. """
142. cost = self.get\_neighbor\_metric(route.next\_address)
143. route.set\_next\_cost(cost)
145. *# Check if we have this route already*
146. **if** (route.address **in** self.routes.keys()):
147. *# We have that route in the table already*
148. old\_route = self.routes[route.address]
150. **if** old\_route.next\_address == route.next\_address:
151. *# Sanity check, if it's marked we already know it's unreachable*
152. **if** (route.cost() != old\_route.cost()):
153. self.routes[route.address] = route
155. **elif** route.cost() < old\_route.cost():
156. *# Store new value*
157. self.routes[route.address] = route
158. **else**:
159. **return**
161. **elif** route.cost() < 16:
162. *# The route is new, and under 16*
163. self.routes[route.address] = route
165. **else**:
166. *# The route is new but is marked for deletion.*
167. *# Seems odd, our neighbors should get this anyway and if not then they won't have this route*
168. *# anyway, since we don't have it either*
169. **return**
171. **if** route.cost() >= 16 **and** **not** route.marked **and** \
172. route.next\_address != self.id **and** route.address != self.id:
173. route.mark()
174. *# Force an update of all routers*
175. self.update()
177. **def** incoming\_update(self, raw\_data):
178. """
179. Provides checks of packet length, version, command and non-neighbors
180. :param raw\_data data received (bytes)
181. """
182. data = packet.ByteArray(raw\_data)
183. **if** if\_failed(data.size() >= 4, "Recieved invalid packet of length " + str(data.size())):
184. **return**
186. command = data.get\_byte()
187. **if** if\_failed(command == 2, "Recieved RIPv2 request (expected only responses)"):
188. **return**
189. version = data.get\_byte()
190. **if** if\_failed(version == 2, "Recieved RIPv1 or other message (expected RIPv2)"):
191. **return**
192. from\_id = data.get\_word()
194. **if** if\_failed(from\_id **in** self.neighbors, "Recieved update from non-neighbor router"):
195. **return**
197. self.set\_neighbor\_updated(from\_id)
199. **while** (**not** data.is\_empty()):
200. afi = data.get\_word()       *# AF\_INET (2)*
201. **if** if\_failed(afi == 2, "Recieved unknown AF\_INET (expected 2)"):
202. **return**
203. data.get\_word()             *# (BLANK) should we check for this being 0? Probably not important.*
204. address = data.get\_dword()  *# Router-ID*
205. data.get\_dword()            *# (BLANK)*
206. data.get\_dword()            *# (BLANK)*
207. hops = data.get\_dword()     *# 1-15 inclusive, or 16 (infinity)*
208. **if** if\_failed(0 <= hops <= 16, "Recieved invalid hop count (setting to 16)"):
209. hops = 16
211. r = Route(address, from\_id, hops, afi)
212. self.incoming\_route(r)

215. **def** \_start\_timer(self):
216. delay = Router.TIMER\_TICK \* random.uniform(0.8, 1.2)
217. self.timer = Timer(delay, Router.tick, args=[self])
218. self.timer.start()
220. **def** print\_table(self):
221. """
222. Prints a human readable representation of the routes in routing table
223. """
224. now = time.strftime("%X")
225. **print**("[{}] Routing Table for {}".format(str(now), self.router\_id))
226. **for** route\_id **in** self.routes.keys():
227. route = self.routes[route\_id]
228. **print**("**\t\t**", route)
230. **def** send\_updates(self):
231. **for** neighbor **in** self.neighbors.keys():
232. **if** (self.get\_neighbor\_metric(neighbor) < 16):
233. self.send\_update\_to\_router(neighbor)

236. **def** send\_update\_to\_router(self, router\_id):
237. *# Sends a specialized update message to a neighbor using split-horizon poisoning*
238. *# Skip sending updates to self*
239. **if** (router\_id == self.id):
240. **return**
241. entries = []
242. **for** route\_id **in** self.routes.keys():
243. route = self.routes[route\_id]
244. new\_metric = route.cost()
245. **if** (route.address == router\_id **or** route.next\_address == router\_id):
246. *# Split-horizon poisoning*
247. new\_metric = 16
249. *# Skip marked messages from contacting the router who sent us the deletion message*
250. **if** (route.marked): **continue**
252. entries.append({"afi": 2, "address": route.address, "metric": new\_metric})
254. data = self.build\_packet(self.id, entries).get\_data()
255. port = self.get\_neighbor\_port(router\_id)
256. sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)
257. sock.sendto(bytes(data), ("localhost", port))
259. **def** update(self):
260. """
261. Check for non-responsive neighbors
262. :return void
263. """
265. now = time.time()
266. **for** router\_id **in** self.neighbors.keys():
267. last = self.get\_neighbor\_updated(router\_id)
268. **if** last + Router.NEIGHBOR\_TIMEOUT < now **and** self.get\_neighbor\_metric(router\_id) < 16:
269. **print**("Router #" + str(router\_id) + " has not responded. Setting metric to 16...")
270. self.neighbors[router\_id][1] = 16
272. *# Check for marked routes ready for deletion*
273. *# Update inaccessible routes with 16*
274. **for** route\_id **in** self.routes.keys():
275. route = self.routes[route\_id]
276. **if** (route.address **in** self.neighbors **and** self.get\_neighbor\_metric(route.address) >= 16 **or** \
277. route.next\_address **in** self.neighbors **and** self.get\_neighbor\_metric(route.next\_address) >= 16):
278. *# If it's a neighbor, check to see if our internal metric is 16.*
279. *# If it is, replace our "route" metrics with 16*
280. route.next\_cost = 16
281. *# Mark for deletion in a little bit...*
282. **if** (**not** route.marked):
283. **print**("Marking route to " + str(route.address) + " for deletion (neighbor metric is 16)")
284. route.mark()
286. **if** (route.marked **and** route.marked\_time + Router.DELETION\_TIMEOUT < now):
287. **print**("Deleting marked route " + str(route) + " as inaccessible")
288. self.routes[route\_id] = None
290. self.routes = { k : v **for** k,v **in** self.routes.items() **if** v **is** **not** None }
292. *# If it's time, send an update ourselves (this handles a metric of 16 for the above!)*
293. self.send\_updates()
294. self.print\_table()
296. **def** tick(self):
297. self.\_start\_timer()
298. self.update()
300. **def** build\_packet(self, sender\_id, entries, command=2, version=2):
301. *# (Byte) Command*
302. *# (Byte) Version*
303. *# (Word) Padding 0x0 (or Sender-ID in COSC364's case)*
304. *# (Void) Entries (20 bytes each, 1-25 entries)*
305. datagram = packet.ByteArray()
306. datagram.insert\_byte(command)
307. datagram.insert\_byte(version)
309. *# COSC364 special: Use the sending router-id here*
310. datagram.insert\_word(sender\_id)
312. *# Limited entries to 25?*
313. **if** if\_failed(len(entries) <= 25, "Recieved invalid entries count (larger than 25)"):
314. **return** datagram
316. **for** i, item **in** enumerate(entries):
317. datagram.insert\_word(item["afi"]) *# AF\_INET (2)*
318. datagram.insert\_word(0)
319. datagram.insert\_dword(item["address"]) *# IPv4*
320. datagram.insert\_dword(0)
321. datagram.insert\_dword(0)
322. datagram.insert\_dword(item["metric"]) *# 1-15 inclusive, or 16 (infinity)*
324. **return** datagram

run.py

1. **import** config\_parser
2. **import** configparser
3. **import** sys
4. **from** router **import** Router

7. **def** main():
8. **if** len(sys.argv) < 2:
9. sys.exit("No file given")
11. *# closes file when done, file like object*
12. **with** open(sys.argv[1]) **as** fp:
13. config = configparser.ConfigParser()
14. config.readfp(fp)
15. config\_dict = config\_parser.parse\_config(config)
17. **if** **not** config\_dict:
18. sys.exit("Invalid config file. Exiting...")
19. **print**("Loaded config file. Starting router...")
20. x = Router(config\_dict)
22. **if** \_\_name\_\_ == "\_\_main\_\_":
23. 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