COSC364 Assignment 1

George Drummond(53243258),
Ryan Cox(64656394)

May 4, 2017

Abstract

The following pertains to Assignment 1 of COSC364 S1 2017. This was a joint work by George Drummond and Ryan Cox in fulfilment of the course requirements of COSC364 S1 2017 and is a result of **equal contribution.**

Contents

1 Particularly well done aspects

One of the initial successes of our design is the decent degree of modulation with which we were able to develop our program. Through the RIP_packet.py and writelog.py programs, we were able to achieve a reasonable level of abstraction around the tasks of packet processing and logging errors. This led to far easier reading and debugging.

Within these modulated programs, the methods of constructing and processing the RIP response packet was a strong point in our design. The packet was represented by a python string of hex values and Pythons type conversion abilities were heavily utilised to allow for swift extraction of the relevant information. Again, this was able to be done under the hood by the helper program RIP_packet.py avoiding much unnecessary indexing by the main program.

These good practices led to a great degree of functionality being achieved by our program. Via the tiered testing method detailed below, we were able to quickly debug and achieve a working (and pretty!) result.

2 Aspects to be improved

One aspect of the program which could have been improved was the way in which we implemented our timing procedures. Though operationally sound, these are not as aesthetically pleasing as they could be, with the precise nature of their operation obscured to the casual observer. This implicit structure would perhaps have been better implemented by an explicit finite automata by defining state variables and functions and working with this as the basis for operation. This would perhaps not only have been nicer to look at but also easier to develop as we could have considered the protocol on a state by state basis rather than a conditional one.

3 Atomicity of event processing

Our program is atomic by nature in that each line is executed one at a time with nothing interrupting the program as it runs through the main round robin (fancy use of terminology) loop. In addition to this, the timers have a small element of randomness to them anyway and its not crucial that we react to them as soon as they tick over so we only react to them when we get back around to checking them. Every event has its time in the loop to do its thing and then waits while we get back around to it. Crucially, no event can interrupt another event that is running

(The blocking and waiting is the longest part of the time waiting of 0.5 seconds and the processing for the other stuff is about 0.0025 seconds so the chance we miss a packet is low, We are not blocking for %0.5 of the time (not doing select for %0.5 on average per program loop)(if we can even miss a packet and if we do we will get it back with the periodic update))

4 discussion

Our primary source for testing during development was our example topology of 7 routers configured as follows.

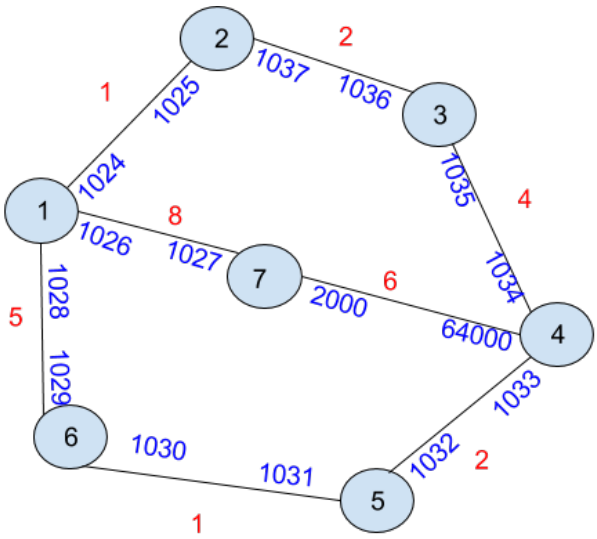

Figure 1: Sample Topology

Where metrics are shown in red and port numbers in blue. This network was implemented by the seven config files of the form router$x$ $x \in \{1, 2, 3, 4, 5, 6, 7\}$ and, in its entirety, is large enough to allow for thorough testing of the RIP routing protocol. This example network is also of particular developmental use as it tests multiple configuration criteria, such as involving both the smallest and largest possible accepted port numbers, checking that the range worked correctly and wasnt accidentally off by one.

During early development, we routinely ran only routing demons 1 and 2 above. This enabled us to quickly check elements of basic functionality (sending/receiving packets, updating routing tables, timeout and garbage collection etc) but obviously did not give us much insight into the correctness of our protocol implementation.

Our second, deeper stage of testing (and the one used for the majority of development) involved a greater number of routing demons and multiple topological aspects. These were namely, the routers 1,2,3 and 6 above. This gave us a far more interesting, though still easily observable network of two transit routers 1 and 2 as well as two stub routers 3 and 6. By taking down either 3 or 6 we were able to test for the correctness of the timeout and garbage collection mechanisms as well as the propagation of link failure information in the form of triggered updates. Likewise, by taking down either 1 or 2, we were able to observe the isolation of a stub router (6 or 3 respectively) from the rest of the network. This stage of testing however, did stop short of the whole picture as it did not test for non-trivial path updates.

Finally a complete run of the network was used as our last stage of testing. Bringing up all routers allowed us to observe convergence times (MAYBE SOME DATA ON CONVERGENCE?) and also to check the validity of the shortest path information of each router. We could also, at this point, test the networks adaptation to extreme topological change. A favourite such change was to bring down routers 4 and 6. This had the joint effect of isolating 5 (i.e making router 5 unreachable from all other routers) and also changing a plethora of shortest path information (for example D(7,3) changes from 10 to 11). It was with this particular test, that a great deal of valuable debugging was achieved.

A particularly useful tool for debugging was the runlog_x $x \in \{1, 2, 3, 4, 5, 6, 7\}$files generated by the routing demon instance Which after being initialised by the routing demon we could write the states and any errors in the routing demon so we could look back at what happened and when. Each routing demon makes its own separate log using its ID and since the log file object was part of the routing class it was easy to write to the log from the code without having to pass around the file object.

5 Main Program

```
#!/usr/bin/python
import sys
import select
import socket
import random
from RIP_packet import *
from writelog import *
import time

MAX_BUFF = 600
MAX_DATA = 512
INF = 16
HOST_ID = '127.0.0.1'

# Router STATES: 0 -> Waiting for input with periodic updates
#                1 -> Needs to send a triggered update

def valid_portn(portn):
    return int(portn) in range(1024,64001)

def valid_ID(routerID):
    return int(routerID) in range(1,64001)

def valid_metric(metric):
    return int(metric) in range(0,INF+1)



class RIProuter:
    def __init__(self,configFile):
        self.periodic = 0
        self.updateFlag = 0
        self.configFile = configFile
        self.parse_config()
        self.socket_setup()
        self.routingTable = RoutingTable(self.timers[1],self.timers[2]) #
    timeout and garbage considered
        self.log = init_log(self.routerID)

        print('routerID =',self.routerID)
        print('inport numbers =',self.inPort_numbers)

        print('peerInfo =',self.peerInfo)
        print('timers =',self.timers)
        print('table=\n',self.routingTable)


    def socket_setup(self):
        self.inPorts = []
```

```
49          for portn in self.inPort_numbers:
50               newSocket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
51
52               #newSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR
    , 1)
53
54               newSocket.bind((HOST_ID, portn))
55               self.inPorts += [newSocket]
56
57
58     def close_sockets(self):
59          ''' Close all sockets'''
60          for port in self.inPorts:
61               port.close()
62
63
64     def parse_config(self):
65          ''' Parse the supplied config file'''
66          lines = self.configFile.readlines()
67          for line in lines:
68               entries = line.split(',')
69               #print(entries)
70               lineType = entries[0]
71               tail = entries[1:]
72               if lineType == 'router-id':
73                    self.set_ID(tail)
74
75               elif lineType == 'input-ports':
76                    self.set_InPort_numbers(tail)
77
78               elif lineType == 'outputs':
79                    self.set_peerInfo(tail)
80
81               elif lineType == 'timers':
82                    self.set_timers(tail)
83
84
85     def set_ID(self,tail):
86          ''' Checks and stores routerID'''
87          myID = int(tail[0])
88          if valid_ID(myID):
89               self.routerID = int(tail[0])
90
91          else:
92               raise(IndexError('Router ID not valid'))
93
94
95     def set_InPort_numbers(self,tail):
96          ''' Checks and stores all supplied inport numbers'''
97          self.inPort_numbers = []
98          for portstring in tail:
99               port = int(portstring)
```

```
100             if (port not in self.inPort_numbers) and valid_portn(port):
101                 self.inPort_numbers += [port]
102             else:
103                 print("invalid inport port {} supplied".format(port))
104
105
106     def set_peerInfo(self,tail):
107         ''' Stores info relevent to immediate neighbours '''
108         self.peerInfo = dict()
109         for triplet in tail:
110             portN,metric,peerID = triplet.split('-')
111             if valid_portn(portN) and valid_metric(metric) and valid_ID(
    peerID):
112                 self.peerInfo[int(peerID)] = (int(portN),int(metric))
113             else:
114                 print("invalid peer info for peer {}".format(peerID))
115
116
117     def set_timers(self,tail):
118         ''' Stores supplied timer info (i.e. periodic, timeout, garbage)
    '''
119         self.timers = []
120         for entry in tail:
121             self.timers += [int(entry)]
122
123
124
125     def send_updates(self):
126         ''' Sends an update message to each neighbour'''
127         i = 0
128         for peerID in self.peerInfo.keys():
129             print("update sent to {}".format(peerID))
130             write_to_log(self.log,
131                          "Sent update to {}".format(peerID))
132             OutSock = self.inPorts[i] # use a different socket to send
    each
133
134             peerPort = self.peerInfo[peerID][0]
135             response = self.response_packet(peerID)
136
137             OutSock.sendto(response.encode('UTF-8'),(HOST_ID,peerPort))
138
139             i += 1
140
141
142     def response_packet(self, peerID):
143         ''' Construct a response packet destined to a neighboring router.
144             Suitable for a periodic or triggered update'''
145
146         packet = ""
147         packet += rip_header(self.routerID)
148         for Entry in self.routingTable:
```

```
149                 # Implement split horizon with poisson reverse
150                 if (Entry.nextHop == peerID) or (Entry.garbageFlag == 1):
151                       print("split horizon entry sent to {}".format(peerID))
152                       packet += RTE(TableEntry(Entry.dest, INF, Entry.nextHop
    )) # set metric to INF
153                 else:
154                       packet += RTE(Entry)
155
156           return packet
157
158       def process_rip_packet(self, packet):
159           ''' Processes a RIP response packet'''
160
161           (peerID,RTEs) = rip_packet_info(packet)
162
163
164           if not valid_ID(peerID):
165                 print("[Error] peerID {} out of range".format(peerID))
166                 write_to_log(self.log, "[Error] peerID {} out of range".
    format(peerID))
167                 #need to do something here
168
169           print("processing packet from {}".format(peerID))
170           cost = self.peerInfo[peerID][1]
171
172           # Consider direct link to peer Router
173           incomingEntry = self.routingTable.get_entry(peerID)
174           if incomingEntry is None:
175                 print("added directlink entry to router {}".format(peerID))
176                 self.routingTable.add_entry(peerID, cost, peerID)
177           else:
178                 incomingEntry.metric = cost
179                 incomingEntry.timeout = 0 # Reinitialise timeout for this
    link
180                 incomingEntry.garbageFlag = 0
181                 incomingEntry.garbage = 0
182
183
184
185           for RTE in RTEs:
186                 self.processRTE(RTE, peerID, cost)
187
188
189       def processRTE(self, RTE, peerID, cost):
190           '''processes an RTE of a RIP responce packet from a peer router
    '''
191           (dest, metric) = RTE
192
193           new_metric = min(metric + cost, INF) # update metric
194
195           """check metric here?"""
196           currentEntry = self.routingTable.get_entry(dest)
```

```
197
198
199
200         if new_metric >= INF:
201             print("Path ({},{}) from {} not processed as unreachable".
    format(dest, metric,peerID))
202             write_to_log(self.log,
203                 "Path ({},{}) from {} not processed as unreachable".
    format(dest, metric,peerID))
204                 #do something here
205
206         if (currentEntry is None):
207             print("current route not in table")
208             if (new_metric < INF): # Add a new entry
209                 NewEntry = TableEntry(dest, new_metric, peerID)
210                 print('new Entry {}'.format(NewEntry))
211                 write_to_log(self.log,
212                     "New route added from {} to {} with Metric {}"
213                     .format(self.routerID, NewEntry, new_metric))
214
215
216                 self.routingTable.add_entry(dest, new_metric, peerID)
217
218         else: # Compare to existing entry
219
220             print("Existing entry for {}".format(dest))
221             if (currentEntry.nextHop == peerID): # Same router as
    existing route
222
223                 currentEntry.timeout = 0 # Reinitialise timeout
224                 currentEntry.garbageFlag = 0
225                 currentEntry.garbage = 0
226
227                 if (new_metric != currentEntry.metric):
228                     self.existing_route_update(currentEntry,
    new_metric, peerID)
229
230
231
232             elif (new_metric < currentEntry.metric):
233                 print("update route to {}".format(dest))
234                 write_to_log(self.log,
235                         "Route from {} to {} updated with new Metric
    {}"
236                         .format(self.routerID, NewEntry, new_metric))
237                 self.existing_route_update(currentEntry, new_metric,
    peerID)
238
239
240
241
242
```

```
243
244
245
246     def existing_route_update(self, currentEntry, new_metric, peerID):
247         ''' updates an existing routing table entry with a new metric'''
248         currentEntry.metric = new_metric
249         print("route to {} updated to metric = {}".format(currentEntry.
    dest,new_metric))
250         currentEntry.nextHop = peerID
251
252         if (new_metric >= INF):
253             print("Triggered update flag set")
254             self.updateFlag = 1 #Set some update flag
255             currentEntry.garbageFlag = 1
256
257
258
259
260 class RoutingTable:
261     def __init__(self, timeoutMax, garbageMax):
262         self.table = []
263         self.timeoutMax = timeoutMax
264         self.garbageMax = garbageMax
265
266     def __iter__(self):
267         i = 0
268         while i < len(self.table):
269             yield(self.table[i])
270             i += 1
271
272     def __repr__(self):
273         blank = "-" * 54
274         print(blank + "\n| dest | metric | nextHop | flag | timeout |
    garbage |")
275         for Entry in self.table:
276             print("|{:>5} |{:>7} |{:>8} |{:>5} |{:>8.3f} |{:>8.3f} |".
    format(
277                 Entry.dest, Entry.metric, Entry.nextHop, Entry.
    garbageFlag,
278                 Entry.timeout, Entry.garbage))
279
280         return blank
281
282     def add_entry(self,dest, metric, nextHop):
283         self.table += [TableEntry(dest, metric, nextHop)]
284
285     def remove_entry(self, Entry):
286         print("Entry {} removed".format(Entry))
287         self.table.remove(Entry)
288
289     def get_entry(self, dest):
290         ''' returns required table entry if already present'''
```

```
291             i = 0
292             while i < len(self.table):
293                  Entry = self.table[i]
294                  if Entry.dest == dest:
295                       return Entry
296
297                  i += 1
298
299             return None
300
301
302
303 class TableEntry:
304      def __init__(self,dest, metric, nextHop):
305           self.dest = dest
306           self.metric = metric
307           self.nextHop = nextHop
308           self.garbageFlag = 0
309           self.timeout = 0
310           self.garbage = 0
311
312      def __repr__(self):
313           return str((self.dest, self.metric,
314                       self.nextHop, self.garbageFlag, self.timeout, self.
    garbage))
315
316
317
318 def main():
319      configFile = open(sys.argv[1])
320      #configFile = open("router1.conf") # Just for developement
321      router = RIProuter(configFile)
322      selecttimeout = 0.5
323      periodicWaitTime = router.timers[0]
324
325      starttime = time.time() #Gets the start time before processing
326
327      while(1):
328           try:
329                # Wait for at least one of the sockets to be ready for
    processing
330                print("table reads\n",router.routingTable)
331                readable, writable, exceptional = select.select(router.
    inPorts, [], router.inPorts, selecttimeout) #block for incoming packets
    for half a second
332
333                # Send triggered updates at this stage
334                if (router.updateFlag == 1):
335                     router.send_updates()
336                     router.updateFlag = 0
337
338                for sock in readable:
```

```
339
340                     packet = sock.recv(MAX_BUFF).decode('UTF-8')
341                     router.process_rip_packet(packet)
342
343                 timeInc = (time.time() - starttime) #finds the time taken on
    processing
344                 #print("proc time = {}".format(timeInc))
345                 starttime = time.time()
346                 router.periodic += timeInc
347
348                 if (router.periodic >= periodicWaitTime): # Periodic update
349                     router.send_updates()
350                     #Recalculate new random wait time in [0.8*periodic,
    1.2*periodic]
351                     periodicWaitTime = random.uniform(0.8*router.timers
    [0],1.2*router.timers[0])
352
353                     router.periodic = 0 # Reset periodic timer
354                     print("Periodic update")
355
356                 for Entry in router.routingTable:
357
358                     if (Entry.garbageFlag == 1):
359                         Entry.garbage += timeInc
360                         if (Entry.garbage >= router.timers[2]): # Garbage
    collection
361                             print('Removed {}'.format(Entry))
362                             write_to_log(router.log,
363                                          "[Warning] Route from {} to {}
    has been removed"
364                                          .format(router.routerID, Entry.
    dest))
365                             router.routingTable.remove_entry(Entry)
366
367                     else:
368                         Entry.timeout += timeInc
369                         if (Entry.timeout >= router.timers[1]): # timeout/
    delete event
370                             print('Timeout')
371                             write_to_log(router.log,
372                                          "[Warning] Route from {} to {}
    has timed out"
373                                          .format(router.routerID, Entry.
    dest))
374                             Entry.metric = INF
375                             router.updateFlag = 1 # require triggered
    update
376                             Entry.garbageFlag = 1 # Set garbage flag
377
378         except KeyboardInterrupt: # 'Taking down' router
379             print("Exiting program")
380             close_log(router.log)
```

```
381                 router.close_sockets()
382                 break
383
384
385 main()
```

6 RIPpacket

```
#!/usr/bin/python

# REMEMBER bytes.hex() and bytes.fromhex()

def bytes_to_int(byte_string):
    return int.from_bytes(byte_string, byteorder='big')

#def int_to_bytes(myint,size):
    #''' converts integer to 'size' number of bytes'''
    #return (myint).to_bytes(size, byteorder='big').hex()


def int_to_bytes(myint, size):
    suffix = hex(myint)[2:]
    prefix = '0'*(2*size-len(suffix))
    return (prefix + suffix)

def rip_header(routerID):
    header = '0201' + int_to_bytes(routerID,2)
    return header

def RTE(Entry):
    zero_row =  int_to_bytes(0,4)
    s = zero_row
    s+= int_to_bytes(Entry.dest,4)
    s+= zero_row
    s+= zero_row
    s+= int_to_bytes(Entry.metric,4)
    return(s)


def rip_packet_info(packet):
    ''' Extracts relevent info from a RIP response packet'''

    RTEs = []

    peerID = int(packet[4:8],16)

    i = 8 # Start of first RTE
    while i < len(packet):

        dest = int(packet[i+8:i+16],16) # Read dest from RTE
        metric = int(packet[i+32:i+40],16) # Read metric from RTE
        RTEs += [(dest, metric)]

        i += (8*5) # Proceed to next RTE

    return(peerID,RTEs)
```

7 writelog

```
from time import *
import inspect

def init_log(ID):
    """Initialises the log with an ID for the file"""
    filename = "runlog_" + str(ID) + ".log"
    program_log = open(filename, 'w')
    program_log.write("Log File for {} in program {}\n{}\n"
                      .format(ID, inspect.stack()[1][1], "/" * 200))
    write_to_log(program_log, "Log Started")
    return program_log

def write_to_log(log, string):
    """takes a log object and writes the given string and timestamps it"""
    logtime = strftime("[%H:%M:%S %d/%m/%Y] ", gmtime())
    log.write(logtime + string + '\n')

def close_log(log):
    """closes the file"""
    write_to_log(log, "Log Ended")
    log.close()

'''#Test case
log = init_log(0)
write_to_log(log, "Error 1")
write_to_log(log, "Error 2")
write_to_log(log, "Error 3")
write_to_log(log, "Error 4")
close_log(log)'''
```