COSC 364
RIP Assignment Report

29 April 2025

Jordan Chubb 62646119
Wenlong Zong 42718910

Percentage contribution

- Jordan Chubb - 60%
- Wenlong Zong - 40%

List of contributions

Jordan Chubb

- Argument parsing
- Configuration file processing
- Socket setup
- RipManager and RoutingTableEntry classes
- Automatic testing

Wenlong Zong

- Timers
- Packet building
- Manual testing
- Dijkstra's algorithm (automatic testing)

Test discussion

Configuration Files

Correctness of configuration files is checked thoroughly in a specific module. Functions within this module are tested using the doctest module. This module can also test the validity of multiple configuration files at once, ensuring that they are all compatible with each other. The test checks that router-id's are unique and are between 1 and 64000, all port numbers are between 1024 and 64000, and metrics are between 1 and 16. Each output of every configuration file is checked that it matches up to exactly one other configuration file, and that the metrics between them are the same. For automatic testing, all the automatically generated configuration files are checked to be valid. The expected outcome of invalid configuration files is that the program will fail to launch and that an appropriate error message will be displayed, which is what was found during testing.

Manual Testing

Configuration files were generated for the example network in the assignment specification. It was expected that the routing tables would converge to the exact topology in the provided example. The outcome was that they converged correctly.

Different routers were then stopped with the expectation that routers who had the stopped-router as a next-hop would then timeout this route and set the metric to 16. This metric of 16 would then propagate to the other routes, and all routers who had the metric at 16 would start the deletion process before deleting the route from their table. The outcome was that this was exactly the case.

Automatic Testing

The automatic testing program works by launching multiple router daemons as individual processes which are then monitored. The network topology is randomly generated by the program, configuration files are made and saved based on this, and then the daemons are started. The

program monitors the daemon processes through their stdout, which uses a json-encoded list to communicate their entire routing table. For each router daemon, a minimum spanning tree is generated based on the previously generated network topology using Dijkstra's algorithm. The minimum spanning tree is compared against the current routers routing table to ensure that the routing table costs are equal to the minimum costs from Dijkstra's algorithm.
The automatic tests include:

- 100 routers, fully-connected (each router is connected to every other router)
- 100 routers, sparsely-connected (each router connects to one other router)

For each of these tests, the program waits until the network has converged correctly, and then once it has, it changes the topology by stopping (crashing) or starting 50 random routers, and then waits for convergence again. It repeats this stopping/starting step several times, waiting for convergence after each change. For these tests we expect every router to eventually converge. Our test results found that the network would converge every time for all tests. This showed that our daemon was correctly converging after topology changes, even for very large networks with groups of routers being disconnected from other groups of routers.

During the convergence process of automatic testing it was discovered that convergence would take over 1 minute after stopping/starting random routers and that the CPU usage was 100% for a 30 second period. It was considered that this may have been caused by the routers waiting for their routes to timeout before updating them. In section 3.9.2 of RIP RFC specification, there is an optional heuristic that switches to a new route if the metric is the same and it is at least halfway to expiry. This is a potential method to improve the long convergence time. This optional heuristic was implemented, but it did not improve the problem.

The problem was then considered to be caused by a large number of triggered updates being sent by routers, which in turn caused other routers to also send triggered updates, making the problem worse. Section 3.10.1 of RIP RFC specification was then implemented, and the outcome was that the CPU usage would no longer go to 100% during the convergence process. This showed that the triggered updates were the source of this problem.

Packet Testing
Malformed packets are randomly generated to check that all packets fit the message format. The RIP packets are expected with command number 2, version number 2, must be zero section for 2 bytes and followed with payload of 1 to 25 RIP entries. Each RIP entry should have address family identifier AF_INET(2), must be zero section for 2 bytes, destination IPv4 address, 2 must be zero sections and the current metric for the destination. All malformed packets are expected to be dropped and that is achieved by the program.

Example configuration file

File: /csse/users/jch442/COSC364/co...onfiguration_files/router1.ini Page 1 of 1

```
[SETTINGS]
router-id = 1
input-ports = 1024, 1025, 1026
outputs = 2001-1-2, 7001-8-7, 6001-5-6
```

Source code

```python
'''COSC364 RIP Assignment
Jordan Chubb
Vincent Zong
29/04/2025
'''

import argparse
from functools import partial
import json
import selectors
import socket
import time

from configmanager import read_config_file
from ripmanager import RipManager


MAX_PACKET_SIZE = 4 + 20 * 25 # header + rip entry * max number of rip entries


parser = argparse.ArgumentParser()
parser.add_argument("config", help="filename of the configuration file")
parser.add_argument("-d", "--debug", help="print debugging information", action="store_true")
parser.add_argument("--autotesting", help="for automatic testing", action="store_true")
args = parser.parse_args()


if args.autotesting:
    """Force all print calls to flush immediately. Required for
    automatic testing's reading of stdout.
    """
    print = partial(print, flush=True)


def main():
    config = read_config_file(args.config)
    debug(config)

    sockets = get_sockets(config)
    selector = selectors.DefaultSelector()
    for sock in sockets:
        selector.register(sock, selectors.EVENT_READ)

    rip = RipManager(debug, config, sockets[0])

    next_print_time = time.time()
    while True:
        next_print = max(0, next_print_time - time.time())
        next_timeout = min(next_print, rip.next_timeout())
```

```
50          events = selector.select(timeout=next_timeout)
51
52          for key, _ in events:
53              sock = key.fileobj
54              message = sock.recv(MAX_PACKET_SIZE)
55              rip.incoming_message(message)
56          rip.send_any_updates()
57
58          if time.time() >= next_print_time:
59              next_print_time = time.time() + 1
60              if args.autotesting:
61                  print(json.dumps(rip.table_list()))
62              else:
63                  print(rip)
64
65
66  def debug(line):
67      if args.debug:
68          print(line)
69
70
71  def get_sockets(config):
72      """Return a socket for each input port."""
73      sockets = []
74      for port in config.input_ports:
75          sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
76          sock.bind(('127.0.0.1', port))
77          sockets.append(sock)
78      return sockets
79
80
81  if __name__ == "__main__":
82      main()
83
```

```python
import math
import random
import time

from configmanager import routerid_is_valid, metric_is_valid


TIME_MULTIPLIER = 6

PERIODIC_UPDATE_DELAY =     30 / TIME_MULTIPLIER
TRIGGERED_UPDATE_DELAY =    5 / TIME_MULTIPLIER
ENTRY_TIMEOUT_DELAY =       180 / TIME_MULTIPLIER
GARBAGE_COLLECTION_DELAY =  120 / TIME_MULTIPLIER


INFINITE_METRIC = 16

POISONED_REVERSE = True


class RipManager:
    """This class manages the Routing Information Protocol. The routing
    table is a dictionary where the key is the destination and the value
    is a RoutingTableEntry. e.g. {destination: RoutingTableEntry}
    A RoutingTableEntry contains info about the next_hop, metric, and timeouts.
    """

    def __init__(self, debug_func, config, output_socket):
        global debug
        debug = debug_func
        self.our_routerid = config.router_id
        self.output_routers = config.outputs
        self.socket = output_socket

        self.routing_table = {}
        self.next_periodic_update = time.time()
        self.triggered_update_pending = False
        self.next_triggered_update = 0


    def __str__(self):
        lines = f'''Router {self.our_routerid:<16} Routing Table
+-------------+----------+--------+------------+--------------+
| destination | next hop | metric | update due | deletion due |
+-------------+----------+--------+------------+--------------+
'''
        for dest, entry in sorted(self.routing_table.items()):
            deletion_due = entry.deletion_due_in()
            if deletion_due == math.inf:
                deletion_due = ''
            else:
```

```
52                  deletion_due = int(deletion_due)
53              lines += f'| {dest:>11} | {entry.next_hop:>8} | {entry.metric:>6} '
54              lines += f'| {entry.update_due_in():>10.0f} | {deletion_due:>12} |\n'
55          lines += '+-------------+---------+--------+-----------+--------------+\n'
56          return lines
57
58
59      def table_list(self):
60          """Return a list of routing table entries. Does not include
61          timeout times, but does include a deletion process flag.
62          Used for automatic testing and to detect routing table changes.
63          """
64          return [[d, e.next_hop, e.metric, e.deletion_process_underway()] for d,e in ↵
             sorted(self.routing_table.items())]
65
66
67      def next_timeout(self):
68          """Return the time in seconds (as a float) until the next timeout.
69          Timeouts include periodic update messages (every 30 seconds), and
70          triggered updates related to routing table entries. Triggered
71          updates occur if a routing table entry hasn't been updated for
72          180 seconds, or if a routing table entry has been garbage
73          collected for 120 seconds.
74          """
75          next_periodic_update_in = self.next_periodic_update - time.time()
76          next_periodic_update_in = max(0, next_periodic_update_in)
77
78          timeouts = [next_periodic_update_in]
79          for entry in self.routing_table.values():
80              timeouts.append(entry.next_timeout())
81
82          if self.triggered_update_pending:
83              next_triggered_update_in = self.next_triggered_update - time.time()
84              timeouts.append(next_triggered_update_in)
85
86          smallest_timeout = min(timeouts)
87          return max(0, smallest_timeout)
88
89
90      def incoming_message(self, message):
91          """Process an incoming UDP packet."""
92          try:
93              rip_packet = RipPacket(message)
94          except AssertionError as e:
95              debug(f"Received invalid packet: {e}")
96              return
97
98          next_hop = rip_packet.routerid
99          if next_hop not in self.output_routers:
100             debug(f'Received packet from unknown router {next_hop}')
101             return
```

```
102             _, metric_to_next_hop = self.output_routers[next_hop]
103             self.add_to_table(next_hop, next_hop, metric_to_next_hop) # add sender to
                routing table
104
105             for rip_entry in rip_packet.entries:
106                 metric = min(metric_to_next_hop + rip_entry.metric, INFINITE_METRIC)
107                 self.add_to_table(rip_entry.routerid, next_hop, metric)
108
109
110         def add_to_table(self, destination, next_hop, metric):
111             """Update or add a table entry.
112             Only add a new entry if the metric isn't infinity.
113             The RIP assignment says to not send a triggered message for
114             metric updates or new routes.
115             """
116             if destination == self.our_routerid:
117                 return # don't add ourself to our routing table
118             if destination in self.routing_table.keys():
119                 reason = self.routing_table[destination].update_entry(next_hop, metric)
120                 if reason:
121                     debug(f'{self.our_routerid} updating routing table entry for
                    destination {destination}:')
122                     debug(f'    {reason}')
123             elif metric < INFINITE_METRIC:
124                 debug(f'{self.our_routerid} added a new route to destination {
                    destination} next-hop {next_hop} metric {metric}')
125                 self.routing_table[destination] = RoutingTableEntry(next_hop, metric)
126
127
128         def send_any_updates(self):
129             """Check if a periodic or triggered update should be sent.
130             Triggered updates only for when routes become invalid (route
131             deleted or metric set to 16), not for new/updated routes.
132             After sending a triggered update, don't send future triggered
133             updates for 1 to 5 seconds.
134             """
135             to_delete = []
136             for destination, entry in self.routing_table.items():
137                 if entry.should_delete():
138                     to_delete.append(destination)
139                     self.triggered_update_pending = True
140                 elif entry.should_begin_deletion():
141                     debug(f'Starting deletion process for destination {destination}')
142                     entry.begin_deletion()
143                     self.triggered_update_pending = True
144
145             for dest in to_delete: # since you cant delete entries while iterating
                over them
146                 debug(f'Deleting destination {dest}')
147                 del self.routing_table[dest]
148
```

```
149             periodic_update = time.time() >= self.next_periodic_update
150             triggered_update = self.triggered_update_pending and time.time() >= self.
                next_triggered_update
151             if periodic_update or triggered_update:
152                 self.send_response_messages()
153
154
155         def send_response_messages(self):
156             """Send a periodic/triggered update message.
157             Send a response message to all neighbours
158             containing the complete routing table (as set by assignment
159             specifications) utilising split-horizon with poisoned-reverse.
160             The next periodic update message should be sent in
161             30 seconds +/- up to 5 seconds (1/6th of 30s) randomly.
162             The next triggered update message should be sent in
163             1 (1/5th of 5 seconds) to 5 seconds randomly.
164             """
165             for router_id, [port, metric] in self.output_routers.items():
166                 packets = self.build_packets(router_id)
167                 for p in packets:
168                     try:
169                         RipPacket(p)
170                     except AssertionError as e:
171                         debug(f'Sending invalid packet: {e}')
172                     self.socket.sendto(p, ('127.0.0.1', port))
173
174             self.next_periodic_update = (time.time() +
175                 PERIODIC_UPDATE_DELAY +
176                 random.uniform(-PERIODIC_UPDATE_DELAY/6, PERIODIC_UPDATE_DELAY/6))
177             self.triggered_update_pending = False
178             self.next_triggered_update = (time.time() +
179                 random.uniform(TRIGGERED_UPDATE_DELAY/5, TRIGGERED_UPDATE_DELAY))
180
181
182         def build_packets(self, destination_router_id):
183             """Return response message packets to be sent to the defined
184             router. Utilises split-horizon with optional poisoned-reverse.
185             """
186             packets = []
187
188             packet = self.empty_rip_packet()
189             packet += rip_entry(destination_router_id, INFINITE_METRIC) # always add
                the receiver as a rip entry with inf metric
190
191             for destination, entry in self.routing_table.items():
192                 metric = entry.metric
193                 if entry.next_hop == destination_router_id:
194                     if POISONED_REVERSE:
195                         metric = INFINITE_METRIC
196                     else:
197                         continue # don't add the entry
```

```
198
199             if len(packet) >= (4 + 20*25): # if 25 entries
200                 packets.append(packet)
201                 packet = self.empty_rip_packet()
202
203             packet += rip_entry(destination, metric)
204
205         packets.append(packet)
206         return packets
207
208
209     def empty_rip_packet(self):
210         """Return an empty rip packet (headers only).
211         RFC all-zeros field is used for the routerid by assignment specs.
212         """
213         packet = bytearray(4)
214         packet[0] = 2 # command
215         packet[1] = 2 # version
216         packet[2:4] = self.our_routerid.to_bytes(2)
217         return packet
218
219
220 def rip_entry(destination, metric):
221     """Return a rip entry for use in a rip packet."""
222     entry = bytearray(20)
223     entry[0:2] = (2).to_bytes(2) # address family identifier
224     entry[4:8] = destination.to_bytes(4)
225     entry[16:20] = metric.to_bytes(4)
226     return entry
227
228
229 class RoutingTableEntry:
230     """A single entry for use in the routing table.
231     The RFC's 'garbage-collection' is called 'deletion' here.
232     Route change flags are not used due to us not sending triggered
233     updates for route metric changes according to the RIP assignment.
234     """
235
236     def __init__(self, next_hop, metric):
237         self.next_hop = next_hop
238         self.metric = metric
239         self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
240         self.time_deletion_due = None
241
242
243     def deletion_process_underway(self):
244         return self.time_deletion_due != None
245
246
247     def over_halfway_to_update_due(self):
248         due_in = self.time_update_due - time.time()
```

```
249             return due_in <= ENTRY_TIMEOUT_DELAY/2
250
251
252         def update_due_in(self):
253             """Time in seconds until an update is due."""
254             due_in = self.time_update_due - time.time()
255             return max(0, due_in)
256
257
258         def deletion_due_in(self):
259             """Time in seconds until deletion is due."""
260             due_in = math.inf
261             if self.deletion_process_underway():
262                 due_in = self.time_deletion_due - time.time()
263             return max(0, due_in)
264
265
266         def next_timeout(self):
267             """Return the time in seconds (as a float) until the next timeout."""
268             smallest_time = min(self.update_due_in(), self.deletion_due_in())
269             return max(0, smallest_time)
270
271
272         def update_entry(self, next_hop, new_metric):
273             """If the deletion process is underway for a route, replace it.
274             If the new metric is 16 then don't add it (no better than current).
275             Return a string describing the reason for change.
276             """
277             reason = None
278             update_timeouts = False
279
280             if next_hop == self.next_hop:
281                 update_timeouts = True
282                 if self.metric != new_metric:
283                     reason = f'updated next-hop {self.next_hop} metric from {self.
                        metric} to {new_metric} (update is from next-hop)'
284                     self.metric = new_metric
285
286             elif new_metric < self.metric:
287                 reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
                    next_hop} ({new_metric}) (better metric)'
288                 update_timeouts = True
289                 self.next_hop = next_hop
290                 self.metric = new_metric
291
292             # RFC section 3.9.2 heuristic
293             elif (new_metric != INFINITE_METRIC and
294                   new_metric == self.metric and
295                   self.over_halfway_to_update_due()):
296                 update_timeouts = True
297                 reason = f'updated next-hop from {self.next_hop} ({self.metric}) to {
```

```
                next_hop} ({new_metric}) (over halfway to update due)'
298             self.next_hop = next_hop
299             self.metric = new_metric
300
301         if update_timeouts:
302             self.time_update_due = time.time() + ENTRY_TIMEOUT_DELAY
303             if self.metric < INFINITE_METRIC:
304                 self.time_deletion_due = None
305
306         return reason
307
308
309     def should_begin_deletion(self):
310         """Return True if the deletion process should be started.
311         Deletion process should not be started if it is already underway.
312         """
313         if not self.deletion_process_underway():
314             return (self.metric >= INFINITE_METRIC or
315                     time.time() >= self.time_update_due)
316         return False
317
318
319     def begin_deletion(self):
320         assert self.deletion_process_underway() is False
321         self.metric = INFINITE_METRIC
322         self.time_deletion_due = time.time() + GARBAGE_COLLECTION_DELAY
323
324
325     def should_delete(self):
326         """Return True if this entry should be deleted immediately."""
327         if self.deletion_process_underway():
328             return time.time() >= self.time_deletion_due
329         return False
330
331
332 class RipPacket:
333     """This class represents a validated RIP request packet.
334     If a RIP packet entry is invalid, ignore it.
335     1 byte - command (must be 2)
336     1 byte - version (must be 2)
337     2 bytes - routerid (all-zeros in RIP RFC)
338     20 bytes - rip entry (1 to 25 lots of these)
339     """
340     def __init__(self, packet):
341         self.validate_rip_packet(packet)
342         self.routerid = int.from_bytes(packet[2:4])
343         self.entries = []
344         for i in range(4, len(packet), 20):
345             try:
346                 self.entries.append(RipEntry(packet[i: i+20]))
347             except AssertionError as e:
```

```
348                 debug(f'RIP packet entry error: {e}')
349
350     def __str__(self):
351         lines = f'''packet:
352     Source: {self.routerid}'''
353         for entry in self.entries:
354             lines += f"""
355         {entry}"""
356         if not self.entries:
357             lines += f"""
358         <EMPTY PACKET>"""
359         return lines + '\n'
360
361     def validate_rip_packet(self, packet):
362         """Raise an AssertionError if the packet is invalid.
363         Does not check the validity of the contained rip entries.
364         """
365         assert len(packet) >= 4+20, f"packet length invalid: {len(packet)}"
366         assert len(packet) <= 4+20*25, f"packet length invalid: {len(packet)}"
367         assert (len(packet) - 4) % 20 == 0, f"packet length invalid: {len(packet)}"
368         assert packet[0] == 2, "command field not 2"
369         assert packet[1] == 2, "version field not 2"
370         routerid = int.from_bytes(packet[2:4])
371         assert routerid_is_valid(routerid), f"router-id invalid {routerid}"
372
373
374 class RipEntry:
375     """This class represents a validated RIP entry from a RIP packet.
376     2 bytes - address family (ignore)
377     2 bytes - all zeros
378     4 bytes - routerid (IPv4 in RIP RFC)
379     8 bytes - all zeros
380     4 bytes - metric
381     """
382     def __init__(self, entry):
383         self.validate_rip_entry(entry)
384         self.routerid = int.from_bytes(entry[4:8])
385         self.metric = int.from_bytes(entry[16:20])
386
387     def __str__(self):
388         return f'router-id: {self.routerid} metric: {self.metric}'
389
390     def validate_rip_entry(self, entry):
391         """Raise an AssertionError if the rip entry is invalid."""
392         assert len(entry) == 20, "RIP entry length not 20"
393         assert int.from_bytes(entry[0:2]) == 2, "address family must be 2"
394         assert int.from_bytes(entry[2:4]) == 0, "field must be all zeros"
395         routerid = int.from_bytes(entry[4:8])
396         assert routerid_is_valid(routerid), f"router-id invalid {routerid}"
397         assert int.from_bytes(entry[8:16]) == 0, "field must be all zeros"
398         metric = int.from_bytes(entry[16:20])
```

```
399             assert metric_is_valid(metric), f"metric invalid {metric}"
400
```

```
import configparser
from itertools import combinations


class Config:
    def __init__(self, router_id, input_ports, outputs):
        self.router_id = router_id
        self.input_ports = input_ports
        self.outputs = outputs

    def __str__(self):
        lines = f"""CONFIG:
    router id: {self.router_id}
    input ports: {self.input_ports}
    outputs:"""
        for routerid, [port, metric] in self.outputs.items():
            lines += f"""
        router-id: {routerid} port: {port} metric: {metric}"""
        return lines + '\n'


def read_config_file(filename):
    config = configparser.ConfigParser()
    config.read(filename)
    try:
        return get_config(config)
    except ValueError as e:
        raise ValueError(f'CONFIG {filename} ERROR: {e}')


def get_config(config):
    """
    >>> config = configparser.ConfigParser()
    >>> config['SETTINGS'] =
    {'router-id':'2','input-ports':'2000','outputs':'3000-1-3'}
    >>> c1 = get_config(config)
    >>> print(c1)
    CONFIG:
        router id: 2
        input ports: [2000]
        outputs:
            router-id: 3 port: 3000 metric: 1
    <BLANKLINE>

    """
    router_id, input_ports, outputs = validate_config(config)
    return Config(router_id, input_ports, outputs)


def validate_configs_by_filename(filenames):
    configs = [read_config_file(filename) for filename in filenames]
```

```
51      validate_configs(configs)
52
53  def validate_configs(configs):
54      """For all the provided configs:
55      ensures that all router-ids are unique,
56      sending/receiving router-ids match between neighbours,
57      and that metrics between neighbours are the same.
58
59      >>> config1 = configparser.ConfigParser()
60      >>> config1['SETTINGS'] =
        {'router-id':'2','input-ports':'2000','outputs':'3000-1-3'}
61      >>> config2 = configparser.ConfigParser()
62      >>> config3 = configparser.ConfigParser()
63
64      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3000','outputs':'2000-1-2'}
65      >>> validate_configs([get_config(config1), get_config(config2)])
66
67      >>> config2['SETTINGS'] =
        {'router-id':'2','input-ports':'3000','outputs':'2000-1-2'}
68      >>> validate_configs([get_config(config1), get_config(config2)])
69      Traceback (most recent call last):
70      AssertionError: same router-id: 2
71
72      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3333','outputs':'3000-1-2'}
73      >>> validate_configs([get_config(config1), get_config(config2)])
74      Traceback (most recent call last):
75      AssertionError: port 3000 is already an output to router 3
76
77      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3000','outputs':'2000-1-3'}
78      >>> validate_configs([get_config(config1), get_config(config2)])
79      Traceback (most recent call last):
80      AssertionError: router-id mismatch between routers 2 and 3 on port 2000
81
82      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3000','outputs':'2222-1-2'}
83      >>> validate_configs([get_config(config1), get_config(config2)])
84      Traceback (most recent call last):
85      AssertionError: router 2 listening on port 2000 but no sender
86
87      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3333','outputs':'2000-1-2'}
88      >>> validate_configs([get_config(config1), get_config(config2)])
89      Traceback (most recent call last):
90      AssertionError: sending to router 3 on port 3000 but no receiver
91
92      >>> config2['SETTINGS'] =
        {'router-id':'3','input-ports':'3000','outputs':'2000-2-2'}
93      >>> validate_configs([get_config(config1), get_config(config2)])
```

```
94        Traceback (most recent call last):
95        AssertionError: metric mismatch between routers 2 and 3
96
97        >>> config1['SETTINGS'] = {'router-id':'2','input-ports':'2000,2001','outputs':'3000-1-3,4000-2-4'}
98        >>> config2['SETTINGS'] = {'router-id':'3','input-ports':'3000,3001','outputs':'2000-1-2,4001-3-4'}
99        >>> config3['SETTINGS'] = {'router-id':'4','input-ports':'4000,4001','outputs':'2001-2-2,3001-3-3'}
100       >>> validate_configs([get_config(config1), get_config(config2), get_config(config3)])
101       """
102       for c1, c2 in combinations(configs, 2):
103           assert c1.router_id != c2.router_id, f'same router-id: {c1.router_id}'
104
105       port_ids = {} # {port: [input_id, output_id]}
106       metrics = {}  # {(router1_id, router2_id), metric]} # where router1_id < router2_id
107       for config in configs:
108           for port in config.input_ports:
109               current_ids = port_ids.get(port, [None, None])
110               assert current_ids[0] is None, f'port {port} already an input for router {current_ids[0]}'
111               current_ids[0] = config.router_id
112               port_ids[port] = current_ids
113
114           for router_id, [port, metric] in config.outputs.items():
115               current_ids = port_ids.get(port, [None, None])
116               assert current_ids[1] is None, f'port {port} is already an output to router {current_ids[1]}'
117               current_ids[1] = router_id
118               port_ids[port] = current_ids
119
120               lower_id, upper_id = sorted([config.router_id, router_id])
121               current_metric = metrics.get((lower_id, upper_id), None)
122               if current_metric is not None:
123                   assert current_metric == metric, f'metric mismatch between routers {lower_id} and {upper_id}'
124               metrics[(lower_id, upper_id)] = metric
125
126       for port, [in_id, out_id] in port_ids.items():
127           assert in_id != None, f'sending to router {out_id} on port {port} but no receiver'
128           assert out_id != None, f'router {in_id} listening on port {port} but no sender'
129           assert in_id == out_id, f'router-id mismatch between routers {in_id} and {out_id} on port {port}'
130
131
132   def routerid_is_valid(routerid):
133       return  1 <= routerid <= 64000
```

```
134
135 def validate_router_id(routerid):
136     """
137     >>> validate_router_id('1')
138     1
139     >>> validate_router_id('64000')
140     64000
141     >>> validate_router_id('0')
142     Traceback (most recent call last):
143     ValueError: router-id must be a number between 1 and 64000. Got "0"
144     >>> validate_router_id('64001')
145     Traceback (most recent call last):
146     ValueError: router-id must be a number between 1 and 64000. Got "64001"
147     """
148     routerid = routerid.strip()
149     if routerid.isdigit() and routerid_is_valid(int(routerid)):
150         return int(routerid)
151     else:
152         raise ValueError(f'router-id must be a number between 1 and 64000. Got "{
        routerid}"')
153
154
155 def port_is_valid(port):
156     return 1024 <= port <= 64000
157
158 def validate_port(port):
159     """
160     >>> validate_port('1024')
161     1024
162     >>> validate_port('64000')
163     64000
164     >>> validate_port('1023')
165     Traceback (most recent call last):
166     ValueError: port must be a number between 1024 and 64000. Got "1023"
167     >>> validate_port('64001')
168     Traceback (most recent call last):
169     ValueError: port must be a number between 1024 and 64000. Got "64001"
170     """
171     port = port.strip()
172     if port.isdigit() and port_is_valid(int(port)):
173         return int(port)
174     else:
175         raise ValueError(f'port must be a number between 1024 and 64000. Got "{port
        }"')
176
177
178 def metric_is_valid(metric):
179     return 1 <= metric <= 16
180
181 def validate_metric(metric):
182     """
```

```
183         >>> validate_metric('1')
184         1
185         >>> validate_metric('16')
186         16
187         >>> validate_metric('0')
188         Traceback (most recent call last):
189         ValueError: metric must be a number between 1 and 16. Got "0"
190         >>> validate_metric('17')
191         Traceback (most recent call last):
192         ValueError: metric must be a number between 1 and 16. Got "17"
193         """
194         metric = metric.strip()
195         if metric.isdigit() and metric_is_valid(int(metric)):
196             return int(metric)
197         else:
198             raise ValueError(f'metric must be a number between 1 and 16. Got "{metric
                }"')
199
200
201     def validate_config(config):
202         """
203         >>> config = configparser.ConfigParser()
204         >>> validate_config(config)
205         Traceback (most recent call last):
206         ValueError: SETTINGS header not found
207
208         >>> config['SETTINGS'] = {'input-ports':'1024','outputs':'64000-0-1'}
209         >>> validate_config(config)
210         Traceback (most recent call last):
211         ValueError: "router-id" parameter not found
212
213         >>> config['SETTINGS'] = {'router-id':'1','outputs':'64000-0-1'}
214         >>> validate_config(config)
215         Traceback (most recent call last):
216         ValueError: "input-ports" parameter not found
217
218         >>> config['SETTINGS'] = {'router-id':'1','input-ports':'1024'}
219         >>> validate_config(config)
220         Traceback (most recent call last):
221         ValueError: "outputs" parameter not found
222
223         >>> config['SETTINGS'] =
            {'router-id':'1','input-ports':'2000,2000','outputs':'5000-15-1'}
224         >>> validate_config(config)
225         Traceback (most recent call last):
226         ValueError: "2000" is a duplicate port number
227
228         >>> config['SETTINGS'] =
            {'router-id':'1','input-ports':'2000','outputs':'2000-15-1'}
229         >>> validate_config(config)
230         Traceback (most recent call last):
```

```
231       ValueError: "2000" is already defined as an input port
232
233       >>> config['SETTINGS'] =                                                    ↵
          {'router-id':'1','input-ports':'1024','outputs':'64000-1-1'}
234       >>> validate_config(config)
235       (1, [1024], {1: [64000, 1]})
236
237       >>> config['SETTINGS'] = {'router-id':' 01 ','input-ports':' 01024 ,        ↵
          01025','outputs':' 064000 - 011 - 01 , 05000 - 012 - 02'}
238       >>> validate_config(config)
239       (1, [1024, 1025], {1: [64000, 11], 2: [5000, 12]})
240
241       >>> config['SETTINGS'] =                                                    ↵
          {'router-id':'1','input-ports':'2000,2001,2002','outputs':'5000-14-2,5001-15-64 ↵
          000'}
242       >>> validate_config(config)
243       (1, [2000, 2001, 2002], {2: [5000, 14], 64000: [5001, 15]})
244
245       """
246       if not 'SETTINGS' in config:
247           raise ValueError('SETTINGS header not found')
248       for param in ['router-id', 'input-ports', 'outputs']:
249           if not param in config['SETTINGS']:
250               raise ValueError(f'"{param}" parameter not found')
251
252       router_id = config['SETTINGS']['router-id']
253       router_id = validate_router_id(router_id)
254
255       input_ports_str = config['SETTINGS']['input-ports'].split(',')
256       input_ports = []
257       for port in input_ports_str:
258           port = validate_port(port)
259           if port in input_ports:
260               raise ValueError(f'"{port}" is a duplicate port number')
261           else:
262               input_ports.append(port)
263
264       outputs_str = config['SETTINGS']['outputs'].split(',')
265       outputs = {}
266       for output in outputs_str:
267           port, metric, out_routerid = output.strip().split('-')
268
269           port = validate_port(port)
270           if port in input_ports:
271               raise ValueError(f'"{port}" is already defined as an input port')
272           metric = validate_metric(metric)
273           out_routerid = validate_router_id(out_routerid)
274
275           outputs[out_routerid] = [port, metric]
276
277       if input_ports == []:
```

```
278             raise ValueError(f'There must be at least one input port')
279         if outputs == []:
280             raise ValueError(f'There must be at least one output')
281 
282         return router_id, input_ports, outputs
283 
284 
285     if __name__ == '__main__':
286         import doctest
287         results = doctest.testmod()
288         print(results)
289 
```

```python
import configparser
import fcntl
from itertools import combinations
import json
import math
import os
import random
import selectors
from subprocess import Popen, PIPE, STDOUT
import time

from configmanager import validate_configs_by_filename


NUM_ROUTERS = 100


FOLDER = 'test_configs'
os.makedirs(FOLDER, exist_ok=True)


class Test:
    def __init__(self, neighbour_func, change_topology=None, topology_changes=1):
        self.make_neighbours_func = neighbour_func
        self.topology_change_func = change_topology
        self.topology_changes_remaining = topology_changes

    def make_neighbours(self, processes):
        self.make_neighbours_func(processes)

    def can_change_topology(self):
        return self.topology_change_func != None and self.
        topology_changes_remaining > 0

    def change_topology(self, processes):
        self.topology_changes_remaining -= 1
        self.topology_change_func(processes)


ports = iter(range(10000, 64000))
def make_neighbours(p1, p2):
    port1 = next(ports)
    port2 = next(ports)
    metric = random.randint(1, 15)
    p1.add_neighbour(port1, port2, metric, p2)
    p2.add_neighbour(port2, port1, metric, p1)

def fully_connected(processes):
    for p1, p2 in combinations(processes, 2):
        make_neighbours(p1, p2)

```

```
51  def sparsely_connected(processes):
52      rand_processes = list(processes)
53      random.shuffle(rand_processes)
54      for p1 in processes:
55          num_neighbours = 0
56          for p2 in rand_processes:
57              if p1.routerid != p2.routerid and p2.routerid not in p1.get_neighbours ↵
                ():
58                  make_neighbours(p1, p2)
59                  num_neighbours += 1
60                  if num_neighbours >= 1:
61                      break
62
63  def change_topology(processes):
64      processes = list(processes)
65      if random.choice([False, True]):
66          to_stop = random.sample(processes, len(processes)//2)
67          print(f'stopping {len(to_stop)} processes randomly')
68          for p in to_stop:
69              p.stop()
70      else:
71          to_start = random.sample(processes, len(processes)//2)
72          print(f'starting {len(to_start)} processes randomly')
73          for p in to_start:
74              p.start()
75
76  test1 = Test(fully_connected)
77
78  test2 = Test(sparsely_connected)
79
80  test3 = Test(fully_connected, change_topology, 5)
81
82  test4 = Test(sparsely_connected, change_topology, 10)
83
84
85  class ProcessManager:
86      def __init__(self):
87          self.processes_dict = {}
88
89      def get_processes(self):
90          return self.processes_dict.values()
91
92      def get_alive_processes(self):
93          return [p for p in self.processes_dict.values() if p.alive]
94
95      def get_process(self, id):
96          return self.processes_dict[id]
97
98      def start_processes(self):
99          for p in self.get_processes():
100             p.start()
```

```
101
102     def stop_processes(self):
103         for p in self.get_processes():
104             p.stop()
105
106     def new_processes(self):
107         self.stop_processes()
108         for i in range(1, NUM_ROUTERS+1):
109             self.processes_dict[i] = Process(i)
110
111     def setup_test(self, test):
112         self.new_processes()
113         test.make_neighbours(self.get_processes())
114         self.write_configs()
115         validate_configs_by_filename([p.filename for p in self.get_processes()])
116         self.start_processes()
117
118     def change_test_topology(self, test):
119         test.change_topology(self.get_processes())
120         for p in self.get_processes():
121             p.clear_routing_table()
122
123     def write_configs(self):
124         for p in self.get_processes():
125             p.write_config()
126
127
128 class Process:
129     def __init__(self, routerid):
130         self.routerid = routerid
131         self.inputs = []
132         self.outputs = {}
133         self.filename = f'{FOLDER}/autoconfig{self.routerid}.ini'
134         self.process = None
135         self.alive = False
136
137         self.routing_table = None
138         self.routing_table_time = math.inf
139         self.have_checked_convergence = False
140         self.converged = False
141
142
143     def __str__(self):
144         return str(self.routerid)
145
146
147     def add_neighbour(self, in_port, out_port, metric, neighbour):
148         self.inputs.append(str(in_port))
149         self.outputs[neighbour.routerid] = [neighbour, out_port, metric]
150
151
```

```
152     def get_neighbours(self):
153         return self.outputs
154 
155 
156     def write_config(self):
157         config = configparser.ConfigParser()
158         config['SETTINGS'] = {
159             'router-id': str(self.routerid),
160             'input-ports': ','.join(self.inputs),
161             'outputs': ','.join(f'{port}-{metric}-{id}' for id, [_, port, metric]
                in self.outputs.items())
162         }
163         with open(self.filename, 'w') as file:
164             config.write(file)
165 
166 
167     def start(self):
168         """Start the process and make its stdout non-blocking."""
169         if not self.alive:
170             self.alive = True
171             self.process = Popen(["python", "daemon.py", self.filename,
                "--autotesting"], stdout=PIPE, stderr=STDOUT)
172             fcntl.fcntl(self.process.stdout.fileno(), fcntl.F_SETFL, os.O_NONBLOCK)
173 
174 
175     def stop(self):
176         self.alive = False
177         self.process.kill()
178 
179 
180     def get_stdout(self):
181         return self.process.stdout
182 
183 
184     def read_line(self):
185         line = self.process.stdout.readline()
186         if line:
187             line = line.decode().strip()
188             try:
189                 line = json.loads(line)
190             except json.decoder.JSONDecodeError as e:
191                 print(self, 'decode error', line)
192                 return
193             if type(line) != list:
194                 print(self, 'received non-list', line)
195                 return
196 
197             if line != self.routing_table:
198                 self.routing_table = line
199                 self.routing_table_time = time.time()
200                 self.have_checked_convergence = False
```

```
201                 self.converged = False
202
203
204     def clear_routing_table(self):
205         self.routing_table = None
206         self.routing_table_time = math.inf
207         self.have_checked_convergence = False
208         self.converged = False
209
210
211     def routing_table_entries(self):
212         return {routerid:metric for routerid, _, metric, _ in self.routing_table}
213
214
215     def check_convergence(self):
216         # an offline router is considered converged
217         if not self.alive:
218             self.converged = True
219             return
220
221         # don't check for convergence again if the routing table hasn't changed
222         if self.have_checked_convergence:
223             return
224
225         # only check if routing table hasn't changed for 10 seconds
226         if time.time() - self.routing_table_time < 10:
227             return
228
229         self.calculate_convergence()
230
231
232     def calculate_convergence(self):
233         min_costs, parents = dijkstras(self.routerid)
234         routing_table_entries = self.routing_table_entries()
235
236         self.converged = True
237         for routerid, metric in min_costs.items():
238             if metric >= 16 or routerid == self.routerid:
239                 continue
240
241             if routerid not in routing_table_entries:
242                 self.converged = False
243                 print(f'{self} not converged to router {routerid} (not in routing table, cost should be: {metric})')
244                 print('Dijkstras path:', dijsktras_path(min_costs, parents, self.routerid, routerid))
245                 print()
246                 continue
247
248             actual_metric = routing_table_entries[routerid]
249             if actual_metric != metric:
```

```
250                     self.converged = False
251                     print(f'{self} not converged to router {routerid} (current cost: {
                        actual_metric}, should be: {metric})')
252                     print('Dijkstras path:', dijsktras_path(min_costs, parents, self.
                        routerid, routerid))
253                     print('Current path:   ', end='')
254                     print_actual_path(self.routerid, routerid)
255                     print()
256
257             self.have_checked_convergence = True
258
259
260     def dijkstras(source_id):
261         dist = {}
262         prev = {}
263         queue = []
264         for p in processmanager.get_alive_processes():
265             id = p.routerid
266             dist[id] = math.inf
267             prev[id] = None
268             queue.append(id)
269         assert source_id in dist
270         dist[source_id] = 0
271
272         while queue:
273             u = None
274             min_dist = math.inf
275             for v in queue:
276                if dist[v] <= min_dist:
277                     u = v
278                     min_dist = dist[v]
279             queue.remove(u)
280
281             u_neighbours = processmanager.get_process(u).get_neighbours()
282             for v, [process, _, metric] in u_neighbours.items():
283                 if v not in queue:
284                     continue
285
286                 cost = dist[u] + metric
287                 if cost <= dist[v]:
288                     dist[v] = cost
289                     prev[v] = u
290
291         return dist, prev
292
293
294     def dijsktras_path(dist, prev, src, dest):
295         current = dest
296         path = f'{current} ({dist[current]})'
297         while current != src:
298             current = prev[current]
```

```
299             path = f'{current} ({dist[current]}) --> ' + path
300         return path
301 
302 
303     def print_actual_path(src, dest, depth=0):
304         if depth > 15:
305             print('ABORTING')
306             return
307         if src == dest:
308             print(f'{src} (0)')
309             return
310 
311         src_routing_table = processmanager.get_process(src).routing_table
312         if src_routing_table == None:
313             print(f'{src} (no route to {dest})')
314             return
315         for routerid, nexthop, metric, _ in src_routing_table:
316             if routerid == dest:
317                 break
318         print(f'{src} ({metric}) --> ', end='')
319         print_actual_path(nexthop, dest, depth+1)
320 
321 
322     processmanager = ProcessManager()
323 
324     def main():
325         tests = [test1, test2, test3, test4]
326         for i in range(len(tests)):
327             test = tests[i]
328             processmanager.setup_test(test)
329             print(f'test {i} starting')
330             run_to_convergence()
331             while test.can_change_topology():
332                 print(f'test {i} changing topology')
333                 processmanager.change_test_topology(test)
334                 run_to_convergence()
335             print(f'test {i} finished')
336 
337 
338     def run_to_convergence():
339         selector = selectors.DefaultSelector()
340         for p in processmanager.get_processes():
341             selector.register(p.get_stdout(), selectors.EVENT_READ, p)
342 
343         prev_not_converged = []
344         while True:
345             events = selector.select(timeout=1)
346             for key, _ in events:
347                 p = key.data
348                 p.read_line()
349 
```

```
350             all_converged = True
351             not_converged = []
352             for p in processmanager.get_processes():
353                 p.check_convergence()
354                 if not p.converged:
355                     all_converged = False
356                     not_converged.append(p.routerid)
357 
358             if all_converged:
359                 print('all routers converged correctly')
360                 return
361             elif not_converged != prev_not_converged:
362                 prev_not_converged = not_converged
363                 print(len(not_converged), 'routers not converged.', not_converged[:10])
364 
365 
366     try:
367         main()
368     except KeyboardInterrupt:
369         pass
370     finally:
371         processmanager.stop_processes()
372     print('exiting')
373 
```