```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)

```

```python
def sparsely_connected(processes):
    rand_processes = list(processes)
    random.shuffle(rand_processes)
    for p1 in processes:
        num_neighbours = 0
        for p2 in rand_processes:
            if p1.routerid != p2.routerid and p2.routerid not in p1.get_neighbours():
                make_neighbours(p1, p2)
                num_neighbours += 1
                if num_neighbours >= 1:
                    break

def change_topology(processes):
    processes = list(processes)
    if random.choice([False, True]):
        to_stop = random.sample(processes, len(processes)//2)
        print(f'stopping {len(to_stop)} processes randomly')
        for p in to_stop:
            p.stop()
    else:
        to_start = random.sample(processes, len(processes)//2)
        print(f'starting {len(to_start)} processes randomly')
        for p in to_start:
            p.start()

test1 = Test(fully_connected)

test2 = Test(sparsely_connected)

test3 = Test(fully_connected, change_topology, 5)

test4 = Test(sparsely_connected, change_topology, 10)


class ProcessManager:
    def __init__(self):
        self.processes_dict = {}

    def get_processes(self):
        return self.processes_dict.values()

    def get_alive_processes(self):
        return [p for p in self.processes_dict.values() if p.alive]

    def get_process(self, id):
        return self.processes_dict[id]

    def start_processes(self):
        for p in self.get_processes():
            p.start()

    def stop_processes(self):
        for p in self.get_processes():
            p.stop()

    def new_processes(self):
        self.stop_processes()
        for i in range(1, NUM_ROUTERS+1):
            self.processes_dict[i] = Process(i)

    def setup_test(self, test):
        self.new_processes()
        test.make_neighbours(self.get_processes())
        self.write_configs()
        validate_configs_by_filename([p.filename for p in self.get_processes()])
        self.start_processes()

    def change_test_topology(self, test):
        test.change_topology(self.get_processes())
        for p in self.get_processes():
            p.clear_routing_table()

    def write_configs(self):
        for p in self.get_processes():
            p.write_config()


class Process:
    def __init__(self, routerid):
        self.routerid = routerid
        self.inputs = []
        self.outputs = {}
        self.filename = f'{FOLDER}/autoconfig{self.routerid}.ini'
        self.process = None
        self.alive = False

        self.routing_table = None
        self.routing_table_time = math.inf
        self.have_checked_convergence = False
        self.converged = False


    def __str__(self):
        return str(self.routerid)


    def add_neighbour(self, in_port, out_port, metric, neighbour):
        self.inputs.append(str(in_port))
        self.outputs[neighbour.routerid] = [neighbour, out_port, metric]


    def get_neighbours(self):
        return self.outputs


    def write_config(self):
        config = configparser.ConfigParser()
        config['SETTINGS'] = {
            'router-id': str(self.routerid),
            'input-ports': ','.join(self.inputs),
            'outputs': ','.join(f'{port}-{metric}-{id}' for id, [_, port, metric] in self.outputs.items())
        }
        with open(self.filename, 'w') as file:
            config.write(file)


    def start(self):
        """Start the process and make its stdout non-blocking."""
        if not self.alive:
            self.alive = True
            self.process = Popen(["python", "daemon.py", self.filename, "--autotesting"], stdout=PIPE, stderr=STDOUT)
            fcntl.fcntl(self.process.stdout.fileno(), fcntl.F_SETFL, os.O_NONBLOCK)


    def stop(self):
        self.alive = False
        self.process.kill()


    def get_stdout(self):
        return self.process.stdout


    def read_line(self):
        line = self.process.stdout.readline()
        if line:
            line = line.decode().strip()
            try:
                line = json.loads(line)
            except json.decoder.JSONDecodeError as e:
                print(self, 'decode error', line)
                return
            if type(line) != list:
                print(self, 'received non-list', line)
                return

            if line != self.routing_table:
                self.routing_table = line
                self.routing_table_time = time.time()
                self.have_checked_convergence = False
```

```python
                self.converged = False


    def clear_routing_table(self):
        self.routing_table = None
        self.routing_table_time = math.inf
        self.have_checked_convergence = False
        self.converged = False


    def routing_table_entries(self):
        return {routerid:metric for routerid, _, metric, _ in self.routing_table}


    def check_convergence(self):
        # an offline router is considered converged
        if not self.alive:
            self.converged = True
            return

        # don't check for convergence again if the routing table hasn't changed
        if self.have_checked_convergence:
            return

        # only check if routing table hasn't changed for 10 seconds
        if time.time() - self.routing_table_time < 10:
            return

        self.calculate_convergence()


    def calculate_convergence(self):
        min_costs, parents = dijkstras(self.routerid)
        routing_table_entries = self.routing_table_entries()

        self.converged = True
        for routerid, metric in min_costs.items():
            if metric >= 16 or routerid == self.routerid:
                continue

            if routerid not in routing_table_entries:
                self.converged = False
                print(f'{self} not converged to router {routerid} (not in routing table, cost should be: {metric})')
                print('Dijkstras path:', dijsktras_path(min_costs, parents, self.routerid, routerid))
                print()
                continue

            actual_metric = routing_table_entries[routerid]
            if actual_metric != metric:
                self.converged = False
                print(f'{self} not converged to router {routerid} (current cost: {actual_metric}, should be: {metric})')
                print('Dijkstras path:', dijsktras_path(min_costs, parents, self.routerid, routerid))
                print('Current path:   ', end='')
                print_actual_path(self.routerid, routerid)
                print()

        self.have_checked_convergence = True


def dijkstras(source_id):
    dist = {}
    prev = {}
    queue = []
    for p in processmanager.get_alive_processes():
        id = p.routerid
        dist[id] = math.inf
        prev[id] = None
        queue.append(id)
    assert source_id in dist
    dist[source_id] = 0

    while queue:
        u = None
        min_dist = math.inf
        for v in queue:
           if dist[v] <= min_dist:
               u = v
               min_dist = dist[v]
        queue.remove(u)

        u_neighbours = processmanager.get_process(u).get_neighbours()
        for v, [process, _, metric] in u_neighbours.items():
            if v not in queue:
                continue

            cost = dist[u] + metric
            if cost <= dist[v]:
                dist[v] = cost
                prev[v] = u

    return dist, prev


def dijsktras_path(dist, prev, src, dest):
    current = dest
    path = f'{current} ({dist[current]})'
    while current != src:
        current = prev[current]
```

```
            path = f'{current} ({dist[current]}) --> ' + path
        return path


    def print_actual_path(src, dest, depth=0):
        if depth > 15:
            print('ABORTING')
            return
        if src == dest:
            print(f'{src} (0)')
            return

        src_routing_table = processmanager.get_process(src).routing_table
        if src_routing_table == None:
            print(f'{src} (no route to {dest})')
            return
        for routerid, nexthop, metric, _ in src_routing_table:
            if routerid == dest:
                break
        print(f'{src} ({metric}) --> ', end='')
        print_actual_path(nexthop, dest, depth+1)


    processmanager = ProcessManager()

    def main():
        tests = [test1, test2, test3, test4]
        for i in range(len(tests)):
            test = tests[i]
            processmanager.setup_test(test)
            print(f'test {i} starting')
            run_to_convergence()
            while test.can_change_topology():
                print(f'test {i} changing topology')
                processmanager.change_test_topology(test)
                run_to_convergence()
            print(f'test {i} finished')


    def run_to_convergence():
        selector = selectors.DefaultSelector()
        for p in processmanager.get_processes():
            selector.register(p.get_stdout(), selectors.EVENT_READ, p)

        prev_not_converged = []
        while True:
            events = selector.select(timeout=1)
            for key, _ in events:
                p = key.data
                p.read_line()

```

```python
        all_converged = True
        not_converged = []
        for p in processmanager.get_processes():
            p.check_convergence()
            if not p.converged:
                all_converged = False
                not_converged.append(p.routerid)

        if all_converged:
            print('all routers converged correctly')
            return
        elif not_converged != prev_not_converged:
            prev_not_converged = not_converged
            print(len(not_converged), 'routers not converged.', not_converged[:10])


try:
    main()
except KeyboardInterrupt:
    pass
finally:
    processmanager.stop_processes()
print('exiting')

```