**Python Implementation of SAMC Algorithm for Ryu Multi-Controller**

|  |
| --- |
| from ryu.base import app\_manager  from ryu.controller import ofp\_event  from ryu.controller.handler import MAIN\_DISPATCHER, DEAD\_DISPATCHER  from ryu.controller.handler import set\_ev\_cls  from ryu.lib import hub  import os  import random  import binascii  class SAMCController(app\_manager.RyuApp):  def \_\_init\_\_(self, \*args, \*\*kwargs):  super(SAMCController, self).\_\_init\_\_(\*args, \*\*kwargs)  self.controllers = {}  self.monitor\_thread = hub.spawn(self.\_monitor)  def initialize(self, controllers):  self.logger.info("Initializing Controllers...")  for controller in controllers:  key = self.generate\_cryptographic\_key()  self.controllers[controller] = {'key': key, 'state': 'active'}  self.logger.info("Controller %s initialized with key %s", controller, key)  self.authenticate\_controller(controller)  def generate\_cryptographic\_key(self):  return binascii.hexlify(os.urandom(32)).decode()  def authenticate\_controller(self, controller):  key = self.controllers[controller]['key']  if key:  self.logger.info("Authenticating Controller %s...", controller)  # Simulate successful authentication  self.logger.info("Controller %s authenticated successfully", controller)  else:  self.logger.warning("Authentication failed for Controller %s", controller)  def establish\_tcc(self, controllers):  self.logger.info("Establishing Trusted Communication Channels (TCC)...")  for controller in controllers:  self.logger.info("TCC established for Controller %s", controller)  # Placeholder for secure communication setup  def \_monitor(self):  while True:  self.monitor\_network(self.controllers.keys())  hub.sleep(10)  def monitor\_network(self, controllers):  self.logger.info("Monitoring Network State...")  for controller in controllers:  metrics = self.gather\_network\_metrics(controller)  self.logger.info("Controller %s metrics: %s", controller, metrics)  def gather\_network\_metrics(self, controller):  return {  "traffic\_volume": random.random(),  "delay": random.random(),  "link\_reliability": random.random(),  "topology\_adjustments": random.random()  }  def evaluate\_controllers(self):  self.logger.info("Evaluating Controllers...")  for controller in self.controllers.keys():  state = self.assess\_controller\_state(controller)  self.logger.info("Controller %s state: %s", controller, state)  def assess\_controller\_state(self, controller):  return {  "overloaded": bool(random.getrandbits(1)),  "underperforming": bool(random.getrandbits(1))  }  def trigger\_dynamic\_adaptation(self):  self.logger.info("Triggering Dynamic Adaptation if necessary...")  for controller in self.controllers.keys():  if bool(random.getrandbits(1)): # Simulate condition check  self.logger.info("Dynamic adaptation initiated for Controller %s", controller)  self.adjust\_control\_interfaces(controller)  def adjust\_control\_interfaces(self, controller):  self.logger.info("Adjusting control interfaces and protocols for Controller %s...", controller)  # Placeholder for dynamic control adjustments |