**Java Implementation of SAMC Algorithm for Floodlight Multi-Controller**

|  |
| --- |
| package net.floodlightcontroller.smc;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import java.security.SecureRandom;  import java.util.HashMap;  import java.util.Map;  import java.util.Set;  public class SAMCController {  private final Logger log = LoggerFactory.getLogger(SAMCController.class);  private final Map<String, String> controllerKeys = new HashMap<>();  private final SecureRandom random = new SecureRandom();  // Initialize the SAMC algorithm  public void initialize(Set<String> controllers) {  log.info("Initializing Controllers...");  for (String controller : controllers) {  String key = generateCryptographicKey();  controllerKeys.put(controller, key);  log.info("Controller {} initialized with key {}", controller, key);  authenticateController(controller);  }  }  // Generate a cryptographic key for each controller  private String generateCryptographicKey() {  byte[] keyBytes = new byte[32];  random.nextBytes(keyBytes);  return bytesToHex(keyBytes);  }  // Authenticate the controller using its cryptographic key  private void authenticateController(String controller) {  String key = controllerKeys.get(controller);  if (key != null && !key.isEmpty()) {  log.info("Authenticating Controller {}...", controller);  // Simulate successful authentication  log.info("Controller {} authenticated successfully", controller);  } else {  log.warn("Authentication failed for Controller {}", controller);  }  }  // Establish a Trusted Communication Channel (TCC) between controllers  public void establishTCC(Set<String> controllers) {  log.info("Establishing Trusted Communication Channels (TCC)...");  for (String controller : controllers) {  log.info("TCC established for Controller {}", controller);  // Placeholder for secure communication setup  }  }  // Monitor the network state  public void monitorNetwork(Set<String> controllers) {  log.info("Monitoring Network State...");  for (String controller : controllers) {  Map<String, Double> metrics = gatherNetworkMetrics(controller);  log.info("Controller {} metrics: {}", controller, metrics);  }  }  // Gather network metrics for each controller  private Map<String, Double> gatherNetworkMetrics(String controller) {  Map<String, Double> metrics = new HashMap<>();  metrics.put("traffic\_volume", random.nextDouble());  metrics.put("delay", random.nextDouble());  metrics.put("link\_reliability", random.nextDouble());  metrics.put("topology\_adjustments", random.nextDouble());  return metrics;  }  // Evaluate each controller's state  public void evaluateControllers(Set<String> controllers) {  log.info("Evaluating Controllers...");  for (String controller : controllers) {  Map<String, Boolean> state = assessControllerState(controller);  log.info("Controller {} state: {}", controller, state);  }  }  // Assess controller state (e.g., overloaded, underperforming)  private Map<String, Boolean> assessControllerState(String controller) {  Map<String, Boolean> state = new HashMap<>();  state.put("overloaded", random.nextBoolean());  state.put("underperforming", random.nextBoolean());  return state;  }  // Trigger dynamic adaptation if necessary  public void triggerDynamicAdaptation(Set<String> controllers) {  log.info("Triggering Dynamic Adaptation if necessary...");  for (String controller : controllers) {  if (random.nextBoolean()) { // Simulate condition check  log.info("Dynamic adaptation initiated for Controller {}", controller);  adjustControlInterfaces(controller);  }  }  }  // Adjust control interfaces and protocols dynamically  private void adjustControlInterfaces(String controller) {  log.info("Adjusting control interfaces and protocols for Controller {}...", controller);  // Placeholder for dynamic control adjustments  }  // Helper method to convert bytes to hex string  private String bytesToHex(byte[] bytes) {  StringBuilder hexString = new StringBuilder();  for (byte b : bytes) {  String hex = Integer.toHexString(0xff & b);  if (hex.length() == 1) hexString.append('0');  hexString.append(hex);  }  return hexString.toString();  }  } |