Java Implementation of LDMC Algorithm for Beacon Multi-Controller

```
// LDMC Algorithm Implementation for Beacon Multi-Controller (Java)

import java.io.*;
import java.net.*;
import java.security.KeyStore;
import javax.net.ssl.*;
import java.security.SecureRandom;
import java.util.*;
import java.util.concurrent.*;
import java.util.logging.Logger;
import com.google.gson.Gson;

public class LDMC {

    private static final Logger LOGGER = Logger.getLogger(LDMC.class.getName());

    // Configuration (Replace with your actual values)
    private static final String LC_ADDRESS = "127.0.0.1";
    private static final int LC_PORT = 8081;
    private static final String[] PC_ADDRESSES = {"127.0.0.1:8082", "127.0.0.1:8083"};
    private static final String CERT_FILE = "path/to/your/certificate.jks";
    private static final String KEYSTORE_PASSWORD = "your_keystore_password";
    private static final String TRUSTSTORE_FILE = "path/to/your/truststore.jks";
    private static final String TRUSTSTORE_PASSWORD = "your_truststore_password";
    private static final int MONITORING_INTERVAL = 1000; // Milliseconds
    private static final int GLOBAL_MITIGATION_SEVERITY = 3;
    private static final int REGIONAL_MITIGATION_SEVERITY = 2;

    private static final Gson gson = new Gson();

    // Data Structures
    static class Alert {
        String srcCtrlId;
        String tgtNetSeg;
        int atkSev;
        long ts;
        List<String> extFeats;

        public String toJson() {
            return gson.toJson(this);
        }

        public static Alert fromJson(String json) {
            return gson.fromJson(json, Alert.class);
        }
    }

    // Security Context Setup
    private static SSLContext createSSLContext() throws Exception {
        KeyStore keyStore = KeyStore.getInstance("JKS");
        try (FileInputStream fis = new FileInputStream(CERT_FILE)) {
            keyStore.load(fis, KEYSTORE_PASSWORD.toCharArray());
        }

        KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
        keyManagerFactory.init(keyStore, KEYSTORE_PASSWORD.toCharArray());

        KeyStore trustStore = KeyStore.getInstance("JKS");
```

```
        try (FileInputStream fis = new FileInputStream(TRUSTSTORE_FILE)) {
            trustStore.load(fis, TRUSTSTORE_PASSWORD.toCharArray());
        }

        TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
        trustManagerFactory.init(trustStore);

        SSLContext sslContext = SSLContext.getInstance("TLS");
        sslContext.init(keyManagerFactory.getKeyManagers(), trustManagerFactory.getTrustManagers(), new
SecureRandom());
        return sslContext;
    }

    // Communication Functions
    private static void sendSecure(SSLSocket socket, String message) throws IOException {
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        out.println(message);
    }

    private static String receiveSecure(SSLSocket socket) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        return in.readLine();
    }

   private static boolean anomalyDetected(String switchId) {
                return false;
    }

    private static int detectSeverity(Object anomaly) {
                return 1;
    }

    private static List<String> extractFeatures(String switchId) {
        return new ArrayList<>();     }

    private static List<String> generateGlobalMitigationRules(Alert alert) {
        return Arrays.asList("global_rule
    }

    private static List<String> generateRegionalMitigationRules(Alert alert) {
        return Arrays.asList("regional_rule");
    }

    private static List<String> generateLocalMitigationRules(Alert alert) {
        return Arrays.asList("local_rule");
    }

    private static List<String> findControllers(String targetSegment) {
        return Arrays.asList("127.0.0.1:8082");
    }

    private static String findController(String controllerId) {
        return "127.0.0.1:8082";
    }

    private static void applyMitigation(String controllerAddress, List<String> rules) {
        LOGGER.info("Applying mitigation rules " + rules + " to " + controllerAddress);
    }

    private static void updateGlobalView(String data) {
        LOGGER.info("Updating global view with data: " + data);
```

```
    }

    // Physical Controller (PC) Thread
    private static class PCThread implements Runnable {
        private final String pcAddress;
        private final List<String> managedSwitches;
        private final SSLContext sslContext;

        public PCThread(String pcAddress, List<String> managedSwitches, SSLContext sslContext) {
            this.pcAddress = pcAddress;
            this.managedSwitches = managedSwitches;
            this.sslContext = sslContext;
        }

        @Override
        public void run() {
            try {
                String pcId = UUID.randomUUID().toString();
                String[] parts = pcAddress.split(":");
                String host = parts[0];
                int port = Integer.parseInt(parts[1]);

                SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();
                try (SSLSocket socket = (SSLSocket) sslSocketFactory.createSocket(LC_ADDRESS, LC_PORT))
{
                    for (String switchId : managedSwitches) {
                        if (anomalyDetected(switchId)) {
                            int severity = detectSeverity(null); //replace null with anomaly
                            List<String> features = extractFeatures(switchId);
                            Alert alert = new Alert();
                            alert.srcCtrlId = pcId;
                            alert.tgtNetSeg = "network_segment";
                            alert.atkSev = severity;
                            alert.ts = System.currentTimeMillis();
                            alert.extFeats = features;
                            sendSecure(socket, alert.toJson());
                            sendSecure(socket, "local_data"); // Simulate sending local data
                            Thread.sleep(MONITORING_INTERVAL);
                        }
                    }
                }
            } catch (Exception e) {
                LOGGER.severe("PC thread error: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }

    // Logical Controller (LC) Thread
    private static class LCThread implements Runnable {
        private final SSLContext sslContext;
        private final BlockingQueue<Alert> alertQueue = new LinkedBlockingQueue<>();
        private final BlockingQueue<String> dataQueue = new LinkedBlockingQueue<>();

        public LCThread(SSLContext sslContext) {
            this.sslContext = sslContext;
        }

        @Override
        public void run() {
            try {
                SSLServerSocketFactory sslServerSocketFactory = sslContext.getServerSocketFactory();
```

```
                try (SSLServerSocket serverSocket = (SSLServerSocket)
sslServerSocketFactory.createServerSocket(LC_PORT)) {
                    while (true) {
                        SSLSocket clientSocket = (SSLSocket) serverSocket.accept();
                        new Thread(() -> handleClient(clientSocket)).start();
                    }
                }
            } catch (Exception e) {
                LOGGER.severe("LC thread error: " + e.getMessage());
                e.printStackTrace();
            }
        }

        private void handleClient(SSLSocket clientSocket) {
            try {
                while (true) {
                    String data = receiveSecure(clientSocket);
                    if (data != null) {
                        try {
                            Alert alert = Alert.fromJson(data);
                            alertQueue.put(alert);
                        } catch (Exception e) {
                            dataQueue.put(data);
                        }
                    } else {
                        break;
                    }
                }
            } catch (IOException e) {
                LOGGER.severe("LC client handler error: " + e.getMessage());
                e.printStackTrace();
            }
        }

        private void processAlerts() {
            while (true) {
               try {
                    Alert alert = alertQueue.take();
                    if (alert.atkSev == GLOBAL_MITIGATION_SEVERITY) {
                        List<String> rules = generateGlobalMitigationRules(alert);
                        List<String> controllers = findControllers("Net");
                        for (String controller : controllers) {
                            applyMitigation(controller, rules);
                        }
                    } else if (alert.atkSev == REGIONAL_MITIGATION_SEVERITY) {
                        List<String> rules = generateRegionalMitigationRules(alert);
                        List<String> controllers = findControllers(alert.tgtNetSeg);
                        for (String controller : controllers) {
                            applyMitigation(controller, rules);
                        }
                    } else {
                        String controllerAddress = findController(alert.srcCtrlId);
                        List<String> rules = generateLocalMitigationRules(alert);
                        applyMitigation(controllerAddress, rules);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }

        private void processData() {
```

```
            while (true) {
                try {
                    String data = dataQueue.take();
                    updateGlobalView(data);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }

        public void startProcessing() {
            new Thread(this::processAlerts).start();
            new Thread(this::processData).start();
        }
    }

    // Main Execution
    public static void main(String[] args) throws Exception {
        SSLContext sslContext = createSSLContext();

        // Start LC Thread
        LCThread lcThread = new LCThread(sslContext);
        new Thread(lcThread).start();
        lcThread.startProcessing();

        // Start PC Threads
        for (String pcAddress : PC_ADDRESSES) {
            List<String> managedSwitches = Arrays.asList("switch1", "switch2"); // Replace with actual
switches
            PCThread pcThread = new PCThread(pcAddress, managedSwitches, sslContext);
            new Thread(pcThread).start();
        }
    }
}
```

Explanation of the Java Code:

1. **Dependencies:**
    - The code uses standard Java libraries for networking, security (SSL), and threading.
    - It also uses the Gson library for JSON serialization/deserialization. You'll need to add this dependency to your project (e.g., using Maven or Gradle).
2. **Configuration:**
    - The Java code sets up addresses, ports, certificate paths, passwords, and other configuration parameters.
3. **Data Structures:**
    - The Alert class is defined as a Java class with fields corresponding to the alert message structure.
    - It includes methods for converting the object to and from JSON using the Gson library.
4. **Security Context:**
    - The createSSLContext() method sets up the SSL context using keystore and truststore files.
    - This ensures secure communication between the LC and PCs.
5. **Communication Functions:**
    - sendSecure() and receiveSecure() handle sending and receiving messages over SSL sockets.

6. **Physical Controller (PC) Thread:**
    - The PCThread class implements the logic for the physical controllers.
    - It connects to the LC, monitors switches, detects anomalies, generates alerts, and sends them to the LC.
7. **Logical Controller (LC) Thread:**
    - The LCThread class implements the logic for the logical controller.
    - It listens for connections from PCs, receives alerts and data, and processes them.
    - It uses a BlockingQueue to handle alerts and data asynchronously.
8. **Main Execution:**
    - The main() method sets up the SSL context, starts the LC thread, and starts the PC threads.
    - This initializes the LDMC system.

<u>Deployment Instructions (Beacon Multi-Controller):</u>

1. **Prerequisites:**
    - Java Development Kit (JDK): Ensure you have a compatible JDK installed.
    - Beacon Controller: Download and install the Beacon controller.
    - SSL Certificates: Generate or obtain SSL certificates for secure communication.
    - *Gson* Library: Add the *Gson* library to your Beacon project.
2. **Code Integration:**
    - Create a new Java class (e.g., *LDMC.java*) in your Beacon project and copy the provided code into it.
    - Update the configuration parameters with your environment's settings.
    - Place your certificate and *truststore* files in the specified paths.
3. **Build and Run:**
    - Build the Java project using your IDE or build tool (e.g., *Maven, Gradle*).
    - Start the Beacon controller.
    - Run the *LDMC.java* application.
4. **Mininet-WiFi Deployment (Emulation):**
    - Mininet-WiFi Setup: Set up a Mininet-WiFi topology with P4-enabled switches.
    - Beacon Integration: Configure the Mininet-WiFi switches to connect to the Beacon controller.
    - LDMC Deployment: Run the LDMC.java application.
    - Testing: Simulate SYN flood attacks in Mininet-WiFi to test the LDMC framework's detection and mitigation capabilities.
5. **Real-World Deployment:**
    - Hardware Setup: Deploy P4-enabled switches and servers in your physical network.
    - Beacon Installation: Install the Beacon controller on a server.
    - LDMC Deployment: Run the LDMC.java application on the same server or a separate server.
    - Network Configuration: Configure the P4 switches to connect to the Beacon controller.
    - Monitoring and Testing: Monitor the network for SYN flood attacks and test the LDMC framework's effectiveness.