Java Implementation of LDMC Algorithm for OpenDaylight Multi-Controller

```java
// LDMC Algorithm Implementation for OpenDaylight Controller (Java)

import org.opendaylight.controller.sal.binding.api.BindingAwareBroker;
import org.opendaylight.controller.sal.binding.api.RpcProviderRegistry;
import
org.opendaylight.yang.gen.v1.urn.opendaylight.params.xml.ns.yang.controller.config.rev130405.ServiceHelp
er;
import org.osgi.framework.BundleContext;
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Reference;
import org.osgi.service.component.annotations.ReferenceCardinality;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

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 com.google.gson.Gson;

/**
 * LDMC Application for OpenDaylight Controller.
 */
@Component(service = LDMC.class, immediate = true)
public class LDMC {

    private static final Logger LOG = LoggerFactory.getLogger(LDMC.class);

    @Reference(cardinality = ReferenceCardinality.MANDATORY)
    private BindingAwareBroker bindingAwareBroker;

    @Reference(cardinality = ReferenceCardinality.MANDATORY)
    private RpcProviderRegistry rpcProviderRegistry;

    private BundleContext bundleContext;

    // 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) {
        LOG.info("Applying mitigation rules " + rules + " to " + controllerAddress);
    }

    private static void updateGlobalView(String data) {
        LOG.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) {
                LOG.error("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) {
                LOG.error("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) {
                LOG.error("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();
        }
    }

    // OpenDaylight Component Lifecycle Methods
    @Activate
    protected void activate(BundleContext bundleContext) {
        this.bundleContext = bundleContext;
        try {
            SSLContext sslContext = createSSLContext();
            LCThread lcThread = new LCThread(sslContext);
            new Thread(lcThread).start();
            lcThread.startProcessing();

            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();
            }

            LOG.info("Started LDMC Application");
        } catch (Exception e) {
            LOG.error("Error activating LDMC application", e);
        }
    }

    @Deactivate
    protected void deactivate() {
        LOG.info("Stopped LDMC Application");
    }
}
```

Explanation of the OpenDaylight Code:

1. **OpenDaylight Component:**
    - The LDMC class is annotated with @Component(service = LDMC.class, immediate = true) to register it as an OSGi component.
2. **References:**
    - The bindingAwareBroker and rpcProviderRegistry are injected using @Reference annotations. These are core OpenDaylight services.
3. **Bundle Context:**
    - The BundleContext is stored for later use if needed.
4. **Configuration, Data Structures, Security, Communication, and Placeholder Functions:**
    - These parts are identical to the ONOS code and serve the same purpose.
5. **Physical Controller (PC) and Logical Controller (LC) Threads:**
    - These threads are also identical to the ONOS code.
6. **OpenDaylight Component Lifecycle Methods:**
    - @Activate is used for initialization. The application registers itself and starts the LC and PC threads.
    - @Deactivate is used for cleanup.
7. **Logging:**
    - The slf4j logger is used for logging messages.

Deployment Instructions (OpenDaylight Controller):

1. **Prerequisites:**
    - Java Development Kit (JDK): Ensure you have a compatible JDK installed.
    - OpenDaylight Controller: Download and install the OpenDaylight controller.
    - SSL Certificates: Generate or obtain SSL certificates for secure communication.
    - Gson Library: Add the Gson library to your OpenDaylight project.
2. **Code Integration:**
    - Create a new Java class (e.g., LDMC.java) in your OpenDaylight project.
    - 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 Deploy:**
    - Build the OpenDaylight project using Maven.
    - Deploy the built bundle to OpenDaylight using the Karaf console (e.g., osgi:install file:/path/to/your/bundle.jar; osgi:start <bundle-id>).
4. **Mininet-WiFi Deployment (Emulation):**
    - Mininet-WiFi Setup: Set up a Mininet-WiFi topology with P4-enabled switches.
    - OpenDaylight Integration: Configure the Mininet-WiFi switches to connect to the OpenDaylight controller.

- LDMC Deployment: The LDMC application will start automatically when the bundle is deployed and started.
- 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.
    - OpenDaylight Installation: Install the OpenDaylight controller on a server.
    - LDMC Deployment: Deploy the LDMC bundle to OpenDaylight.
    - Network Configuration: Configure the P4 switches to connect to the OpenDaylight controller.
    - Monitoring and Testing: Monitor the network for SYN flood attacks and test the LDMC framework's effectiveness.