**Source Code Explanation and Deployment Instructions**

1. **ADMC Source Code Explanation and Deployment Instructions**
2. **DMECP Source Code Explanation and Deployment Instructions**
3. **P4-ATM Source Code Explanation and Deployment Instructions**
4. **P4-MCAM Source Code Explanation and Deployment Instructions**
5. **ADMC Source Code Explanation and Deployment Instructions**

Below is a detailed implementation for each of the controllers (Beacon, ONOS, Floodlight, OpenDaylight, POX, and Ryu). Each implementation covers all phases of the ADMC algorithm, with step-by-step deployment instructions.

* 1. **Java Implementation of ADMC Algorithm for ONOS Multi-Controller**

|  |
| --- |
| import org.apache.felix.scr.annotations.Component;  import org.apache.felix.scr.annotations.Activate;  import org.onosproject.core.ApplicationId;  import org.onosproject.core.CoreService;  import org.onosproject.net.DeviceId;  import org.onosproject.net.device.DeviceService;  import org.onosproject.net.device.Device;  @Component(immediate = true)  public class ADMCApp {  private ApplicationId appId;  private CoreService coreService;  private DeviceService deviceService;  @Activate  protected void activate() {  appId = coreService.registerApplication("org.onosproject.admc");  authenticateControllers();  monitorNetworkConditions();  adaptiveControlLoop();  }  **// Phase 1: Identity Verification and Authentication Procedure**  private void authenticateControllers() {  Iterable<Device> devices = deviceService.getDevices();  for (Device device : devices) {  String id = generateUniqueIdentifier(device);  String cryptographicKey = generateCryptographicKey(device);  if (validateDevice(device, cryptographicKey)) {  establishSecurePropagationChannel(device);  }  }  }  private String generateUniqueIdentifier(Device device) {  // Generate a unique identifier for the device  return "ID\_" + device.id().toString();  }  private String generateCryptographicKey(Device device) {  // Generate a cryptographic key for the device  return "CK\_" + device.id().toString();  }  private boolean validateDevice(Device device, String cryptographicKey) {  // Validate the device using the cryptographic key  return device.annotations().value("cryptographicKey").equals(cryptographicKey);  }  private void establishSecurePropagationChannel(Device device) {  // Establish a secure communication channel with the device  // Example: Use ONOS intents to ensure secure paths  }  **// Phase 2: Monitor Network Conditions**  private void monitorNetworkConditions() {  // Continuously monitor the network conditions in the ONOS network  while (true) {  // Get current network status  Iterable<Device> devices = deviceService.getDevices();  for (Device device : devices) {  // Process network conditions based on device status  // Example: Use ONOS metrics service  }  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  **// Phase 3: Assess Controller States**  private void assessControllerStates() {  Iterable<Device> devices = deviceService.getDevices();  for (Device device : devices) {  if (isOverloaded(device)) {  triggerDynamicAdaptation(device);  }  }  }  private boolean isOverloaded(Device device) {  // Check if the device is overloaded  // Example: Use device load annotations  return false;  }  **// Phase 4: Conditions Triggering Dynamic Adaptation**  private void triggerDynamicAdaptation(Device device) {  if (networkTrafficIsHigh() || isOverloaded(device)) {  adjustControlInterface(device);  adjustCommunicationProtocol(device);  enhanceCollaboration(device);  }  }  private boolean networkTrafficIsHigh() {  // Check if network traffic is high  // Example: Use ONOS traffic statistics  return false;  }  **// Phase 5: Control Interface Adjustment**  private void adjustControlInterface(Device device) {  // Adjust control interfaces on the device  }  **// Phase 6: Communication Protocol Adjustment**  private void adjustCommunicationProtocol(Device device) {  if (needMoreEfficientProtocol()) {  switchToEfficientProtocol(device);  }  }  private boolean needMoreEfficientProtocol() {  // Determine if a more efficient protocol is needed  return false;  }  private void switchToEfficientProtocol(Device device) {  // Switch to a more efficient communication protocol  }  **// Phase 7: Collaboration Enhancement**  private void enhanceCollaboration(Device device) {  // Enhance collaboration between devices  // Example: Use ONOS intents for device-to-device collaboration  }  **// Phase 8: Adaptive Control Loop (ACL)**  private void adaptiveControlLoop() {  while (true) {  gatherFeedback();  fineTuneControlInterfaces();  fineTuneCommunicationProtocols();  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  private void gatherFeedback() {  // Gather feedback from devices  }  private void fineTuneControlInterfaces() {  // Fine-tune control interfaces  }  private void fineTuneCommunicationProtocols() {  // Fine-tune communication protocols  }  } |

* 1. **Java Implementation of ADMC Algorithm for Beacon Multi-Controller**

|  |
| --- |
| import net.beaconcontroller.core.IBeaconProvider;  import net.beaconcontroller.core.IBeaconModule;  public class ADMCModule implements IBeaconModule {  private IBeaconProvider beaconProvider;  // Phase 1: Identity Verification and Authentication Procedure  private void authenticateControllers() {  for (Controller controller : getAllControllers()) {  String id = generateUniqueIdentifier(controller);  String cryptographicKey = generateCryptographicKey(controller);  if (validateController(controller, cryptographicKey)) {  establishSecurePropagationChannel(controller);  }  }  }  private String generateUniqueIdentifier(Controller controller) {  // Generate a unique identifier for the controller  return "ID\_" + controller.getName();  }  private String generateCryptographicKey(Controller controller) {  // Generate a cryptographic key for the controller  return "CK\_" + controller.getName();  }  private boolean validateController(Controller controller, String cryptographicKey) {  // Validate the controller using the cryptographic key  return controller.getCryptographicKey().equals(cryptographicKey);  }  private void establishSecurePropagationChannel(Controller controller) {  // Establish a secure communication channel with the controller  controller.setSecureChannel(true);  }  // Phase 2: Monitor Network Conditions  private void monitorNetworkConditions() {  while (true) {  NetworkConditions conditions = getNetworkConditions();  processNetworkConditions(conditions);  sleep(1000); // Monitor every second  }  }  private NetworkConditions getNetworkConditions() {  // Gather network conditions  return new NetworkConditions();  }  private void processNetworkConditions(NetworkConditions conditions) {  // Process network conditions  // Adjust controller settings if needed  }  // Phase 3: Assess Controller States  private void assessControllerStates() {  for (Controller controller : getAllControllers()) {  ControllerState state = getControllerState(controller);  if (isOverloaded(controller, state)) {  triggerDynamicAdaptation(controller);  }  }  }  private ControllerState getControllerState(Controller controller) {  // Get the state of the controller  return controller.getState();  }  private boolean isOverloaded(Controller controller, ControllerState state) {  // Check if the controller is overloaded  return state.getCpuUsage() > 80 || state.getMemoryUsage() > 80;  }  // Phase 4: Conditions Triggering Dynamic Adaptation  private void triggerDynamicAdaptation(Controller controller) {  if (networkTrafficIsHigh() || controller.isOverloaded()) {  adjustControlInterface(controller);  adjustCommunicationProtocol(controller);  enhanceCollaboration(controller);  }  }  private boolean networkTrafficIsHigh() {  // Check if the network traffic is high  return getNetworkConditions().getTrafficLoad() > 80;  }  // Phase 5: Control Interface Adjustment  private void adjustControlInterface(Controller controller) {  controller.modifyMessageIntervals();  controller.changeMessagePriorities();  controller.changeControlMessageTypes();  }  // Phase 6: Communication Protocol Adjustment  private void adjustCommunicationProtocol(Controller controller) {  if (needMoreEfficientProtocol()) {  switchToEfficientProtocol(controller);  }  }  private boolean needMoreEfficientProtocol() {  // Determine if a more efficient protocol is needed  return getNetworkConditions().getLatency() > 100;  }  private void switchToEfficientProtocol(Controller controller) {  // Switch to a more efficient protocol  controller.setCommunicationProtocol("EfficientProtocol");  }  // Phase 7: Collaboration Enhancement  private void enhanceCollaboration(Controller controller) {  shareControlInfo(controller);  }  private void shareControlInfo(Controller controller) {  // Share control information with other controllers  for (Controller otherController : getAllControllers()) {  if (!otherController.equals(controller)) {  controller.sendControlInfo(otherController);  }  }  }  // Phase 8: Adaptive Control Loop (ACL)  private void adaptiveControlLoop() {  while (true) {  gatherFeedback();  fineTuneControlInterfaces();  fineTuneCommunicationProtocols();  sleep(1000);  }  }  private void gatherFeedback() {  // Gather feedback from controllers  }  private void fineTuneControlInterfaces() {  // Fine-tune control interfaces  }  private void fineTuneCommunicationProtocols() {  // Fine-tune communication protocols  }  @Override  public void startUp() {  authenticateControllers();  monitorNetworkConditions();  adaptiveControlLoop();  }  } |

* 1. **Java Implementation of ADMC Algorithm for Floodlight Multi-Controller**

|  |
| --- |
| import net.floodlightcontroller.core.IFloodlightProviderService;  import net.floodlightcontroller.core.module.IFloodlightModule;  import net.floodlightcontroller.core.module.IFloodlightModuleContext;  import net.floodlightcontroller.core.module.IFloodlightService;  import java.util.Collection;  import java.util.Map;  public class ADMCModule implements IFloodlightModule {  private IFloodlightProviderService floodlightProvider;  @Override  public void init(IFloodlightModuleContext context) {  floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);  }  @Override  public void startUp(IFloodlightModuleContext context) {  authenticateControllers();  monitorNetworkConditions();  adaptiveControlLoop();  }  // Phase 1: Identity Verification and Authentication Procedure  private void authenticateControllers() {  for (Controller controller : getAllControllers()) {  String id = generateUniqueIdentifier(controller);  String cryptographicKey = generateCryptographicKey(controller);  if (validateController(controller, cryptographicKey)) {  establishSecurePropagationChannel(controller);  }  }  }  private String generateUniqueIdentifier(Controller controller) {  // Generate a unique identifier for the controller  return "ID\_" + controller.getId();  }  private String generateCryptographicKey(Controller controller) {  // Generate a cryptographic key for the controller  return "CK\_" + controller.getId();  }  private boolean validateController(Controller controller, String cryptographicKey) {  // Validate the controller using the cryptographic key  return controller.getCryptographicKey().equals(cryptographicKey);  }  private void establishSecurePropagationChannel(Controller controller) {  // Establish a secure communication channel with the controller  // Example: Use Floodlight messaging API  }  // Phase 2: Monitor Network Conditions  private void monitorNetworkConditions() {  while (true) {  NetworkConditions conditions = getNetworkConditions();  processNetworkConditions(conditions);  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  private NetworkConditions getNetworkConditions() {  // Gather network conditions  return new NetworkConditions();  }  private void processNetworkConditions(NetworkConditions conditions) {  // Process network conditions and adjust settings if necessary  }  // Phase 3: Assess Controller States  private void assessControllerStates() {  for (Controller controller : getAllControllers()) {  ControllerState state = getControllerState(controller);  if (isOverloaded(controller, state)) {  triggerDynamicAdaptation(controller);  }  }  }  private ControllerState getControllerState(Controller controller) {  // Get the state of the controller  return controller.getState();  }  private boolean isOverloaded(Controller controller, ControllerState state) {  // Check if the controller is overloaded  return state.getCpuUsage() > 80 || state.getMemoryUsage() > 80;  }  // Phase 4: Conditions Triggering Dynamic Adaptation  private void triggerDynamicAdaptation(Controller controller) {  if (networkTrafficIsHigh() || controller.isOverloaded()) {  adjustControlInterface(controller);  adjustCommunicationProtocol(controller);  enhanceCollaboration(controller);  }  }  private boolean networkTrafficIsHigh() {  // Check if the network traffic is high  return getNetworkConditions().getTrafficLoad() > 80;  }  // Phase 5: Control Interface Adjustment  private void adjustControlInterface(Controller controller) {  controller.modifyMessageIntervals();  controller.changeMessagePriorities();  controller.changeControlMessageTypes();  }  // Phase 6: Communication Protocol Adjustment  private void adjustCommunicationProtocol(Controller controller) {  if (needMoreEfficientProtocol()) {  switchToEfficientProtocol(controller);  }  }  private boolean needMoreEfficientProtocol() {  // Determine if a more efficient protocol is needed  return getNetworkConditions().getLatency() > 100;  }  private void switchToEfficientProtocol(Controller controller) {  // Switch to a more efficient protocol  controller.setCommunicationProtocol("EfficientProtocol");  }  // Phase 7: Collaboration Enhancement  private void enhanceCollaboration(Controller controller) {  shareControlInfo(controller);  }  private void shareControlInfo(Controller controller) {  // Share control information with other controllers  for (Controller otherController : getAllControllers()) {  if (!otherController.equals(controller)) {  controller.sendControlInfo(otherController);  }  }  }  // Phase 8: Adaptive Control Loop (ACL)  private void adaptiveControlLoop() {  while (true) {  gatherFeedback();  fineTuneControlInterfaces();  fineTuneCommunicationProtocols();  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  private void gatherFeedback() {  // Gather feedback from controllers  }  private void fineTuneControlInterfaces() {  // Fine-tune control interfaces  }  private void fineTuneCommunicationProtocols() {  // Fine-tune communication protocols  }  } |

* 1. **Java Implementation of ADMC Algorithm for OpenDaylight Multi-Controller**

|  |
| --- |
| import org.opendaylight.controller.sal.core.ComponentActivatorAbstractBase;  import org.opendaylight.controller.sal.core.Component;  public class ADMCApp extends ComponentActivatorAbstractBase {  @Override  public void start() {  authenticateControllers();  monitorNetworkConditions();  adaptiveControlLoop();  }  // Phase 1: Identity Verification and Authentication Procedure  private void authenticateControllers() {  for (Controller controller : getAllControllers()) {  String id = generateUniqueIdentifier(controller);  String cryptographicKey = generateCryptographicKey(controller);  if (validateController(controller, cryptographicKey)) {  establishSecurePropagationChannel(controller);  }  }  }  private String generateUniqueIdentifier(Controller controller) {  // Generate a unique identifier for the controller  return "ID\_" + controller.getId();  }  private String generateCryptographicKey(Controller controller) {  // Generate a cryptographic key for the controller  return "CK\_" + controller.getId();  }  private boolean validateController(Controller controller, String cryptographicKey) {  // Validate the controller using the cryptographic key  return controller.getCryptographicKey().equals(cryptographicKey);  }  private void establishSecurePropagationChannel(Controller controller) {  // Establish a secure communication channel with the controller  // Example: Use OpenDaylight's messaging service  }  // Phase 2: Monitor Network Conditions  private void monitorNetworkConditions() {  while (true) {  NetworkConditions conditions = getNetworkConditions();  processNetworkConditions(conditions);  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  private NetworkConditions getNetworkConditions() {  // Gather network conditions  return new NetworkConditions();  }  private void processNetworkConditions(NetworkConditions conditions) {  // Process network conditions and adjust settings if necessary  }  // Phase 3: Assess Controller States  private void assessControllerStates() {  for (Controller controller : getAllControllers()) {  ControllerState state = getControllerState(controller);  if (isOverloaded(controller, state)) {  triggerDynamicAdaptation(controller);  }  }  }  private ControllerState getControllerState(Controller controller) {  // Get the state of the controller  return controller.getState();  }  private boolean isOverloaded(Controller controller, ControllerState state) {  // Check if the controller is overloaded  return state.getCpuUsage() > 80 || state.getMemoryUsage() > 80;  }  // Phase 4: Conditions Triggering Dynamic Adaptation  private void triggerDynamicAdaptation(Controller controller) {  if (networkTrafficIsHigh() || controller.isOverloaded()) {  adjustControlInterface(controller);  adjustCommunicationProtocol(controller);  enhanceCollaboration(controller);  }  }  private boolean networkTrafficIsHigh() {  // Check if the network traffic is high  return getNetworkConditions().getTrafficLoad() > 80;  }  // Phase 5: Control Interface Adjustment  private void adjustControlInterface(Controller controller) {  controller.modifyMessageIntervals();  controller.changeMessagePriorities();  controller.changeControlMessageTypes();  }  // Phase 6: Communication Protocol Adjustment  private void adjustCommunicationProtocol(Controller controller) {  if (needMoreEfficientProtocol()) {  switchToEfficientProtocol(controller);  }  }  private boolean needMoreEfficientProtocol() {  // Determine if a more efficient protocol is needed  return getNetworkConditions().getLatency() > 100;  }  private void switchToEfficientProtocol(Controller controller) {  // Switch to a more efficient protocol  controller.setCommunicationProtocol("EfficientProtocol");  }  // Phase 7: Collaboration Enhancement  private void enhanceCollaboration(Controller controller) {  shareControlInfo(controller);  }  private void shareControlInfo(Controller controller) {  // Share control information with other controllers  for (Controller otherController : getAllControllers()) {  if (!otherController.equals(controller)) {  controller.sendControlInfo(otherController);  }  }  }  // Phase 8: Adaptive Control Loop (ACL)  private void adaptiveControlLoop() {  while (true) {  gatherFeedback();  fineTuneControlInterfaces();  fineTuneCommunicationProtocols();  try {  Thread.sleep(1000); // Monitor every second  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  private void gatherFeedback() {  // Gather feedback from controllers  }  private void fineTuneControlInterfaces() {  // Fine-tune control interfaces  }  private void fineTuneCommunicationProtocols() {  // Fine-tune communication protocols  }  } |

**Deployment Instructions:**

**Step 1: Environment Setup**

1. **Install JDK**: Ensure that JDK 8 or later is installed on the system.
2. **Install Maven**: Required for building Java-based controllers.
3. **Install the SDN Controllers**: Download and install Beacon, ONOS, Floodlight, and OpenDaylight controllers from their respective repositories.
4. **Clone the Project Repository**: Clone the source code for each controller into their respective environments.

**Step 2: Build and Deploy the ADMC Module**

1. **Navigate to the Project Directory**:

*cd /path/to/controller*

1. **Build the Project**:

*mvn clean install*

1. **Deploy the Compiled Jar**:
   * For **Beacon**, place the *JAR* file in the modules directory.
   * For **ONOS**, place the *OSGi* bundle in the apps directory.
   * For **Floodlight**, add the module to the *floodlightdefault.properties* configuration file.
   * For **OpenDaylight**, deploy the *OSGi* bundle to the *Karaf* container.

**Step 3: Start the Controllers**

1. **Start Each Controller**:
   * **Beacon**: *java -jar beacon.jar*
   * **ONOS**: *onos-service start*
   * **Floodlight**: *java -jar floodlight.jar*
   * **OpenDaylight**: *bin/karaf*
2. **Verify Deployment**: Ensure that the ADMC module is correctly loaded and operational by checking the controller logs.

**Step 4: Configure the Network**

1. **Set Up Network Devices**: Use Mininet or a similar network emulator to create a simulated network environment.
2. **Connect Devices to the Controllers**: Ensure that the devices are correctly linked to the SDN controllers.
3. **Start Network Simulation**: Run the network simulation to initiate the ADMC algorithm across controllers.
   1. **Python Implementation of ADMC Algorithm for Ryu Multi-Controller**

|  |
| --- |
| from ryu.base import app\_manager  from ryu.controller import ofp\_event  from ryu.controller.handler import MAIN\_DISPATCHER, set\_ev\_cls  from ryu.ofproto import ofproto\_v1\_3  class ADMCApp(app\_manager.RyuApp):  OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]  def \_\_init\_\_(self, \*args, \*\*kwargs):  super(ADMCApp, self).\_\_init\_\_(\*args, \*\*kwargs)  self.controllers = {} # Store controller information  @set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, MAIN\_DISPATCHER)  def switch\_features\_handler(self, ev):  controller\_id = ev.msg.datapath.id  self.controllers[controller\_id] = {  'cryptographic\_key': self.generate\_cryptographic\_key(controller\_id),  'authenticated': False  }  self.authenticate\_controllers(controller\_id)  self.monitor\_network\_conditions()  self.adaptive\_control\_loop()  # Phase 1: Identity Verification and Authentication Procedure  def authenticate\_controllers(self, controller\_id):  if self.validate\_controller(controller\_id):  self.establish\_secure\_propagation\_channel(controller\_id)  self.controllers[controller\_id]['authenticated'] = True  def generate\_cryptographic\_key(self, controller\_id):  # Generate a cryptographic key for the controller  return f"CK\_{controller\_id}"  def validate\_controller(self, controller\_id):  # Validate the controller using the cryptographic key  return self.controllers[controller\_id]['cryptographic\_key'] == f"CK\_{controller\_id}"  def establish\_secure\_propagation\_channel(self, controller\_id):  # Establish a secure communication channel with the controller  self.logger.info(f"Secure Propagation Channel established for controller {controller\_id}")  # Phase 2: Monitor Network Conditions  def monitor\_network\_conditions(self):  self.logger.info("Monitoring network conditions...")  # This would involve periodically gathering network data and making adjustments  # Phase 3: Assess Controller States  def assess\_controller\_states(self):  for controller\_id, controller\_info in self.controllers.items():  if self.is\_overloaded(controller\_id):  self.trigger\_dynamic\_adaptation(controller\_id)  def is\_overloaded(self, controller\_id):  # Check if the controller is overloaded  # This is a placeholder; actual logic would involve monitoring load metrics  return False  # Phase 4: Conditions Triggering Dynamic Adaptation  def trigger\_dynamic\_adaptation(self, controller\_id):  if self.network\_traffic\_is\_high() or self.is\_overloaded(controller\_id):  self.adjust\_control\_interface(controller\_id)  self.adjust\_communication\_protocol(controller\_id)  self.enhance\_collaboration(controller\_id)  def network\_traffic\_is\_high(self):  # Placeholder for checking high network traffic  return False  # Phase 5: Control Interface Adjustment  def adjust\_control\_interface(self, controller\_id):  self.logger.info(f"Adjusting control interface for controller {controller\_id}")  # Phase 6: Communication Protocol Adjustment  def adjust\_communication\_protocol(self, controller\_id):  self.logger.info(f"Adjusting communication protocol for controller {controller\_id}")  # Phase 7: Collaboration Enhancement  def enhance\_collaboration(self, controller\_id):  self.logger.info(f"Enhancing collaboration for controller {controller\_id}")  for other\_id in self.controllers:  if other\_id != controller\_id:  self.logger.info(f"Sharing control info between controller {controller\_id} and controller {other\_id}")  # Phase 8: Adaptive Control Loop (ACL)  def adaptive\_control\_loop(self):  self.logger.info("Starting adaptive control loop...")  # Continuously monitor and adapt based on the network conditions and controller states |

* 1. **Python Implementation of ADMC Algorithm for POX Multi-Controller**

|  |
| --- |
| from pox.core import core  import pox.openflow.libopenflow\_01 as of  log = core.getLogger()  class ADMCApp(object):  def \_\_init\_\_(self):  core.openflow.addListeners(self)  self.controllers = {} # Store controller information  def \_handle\_ConnectionUp(self, event):  controller\_id = event.dpid  self.controllers[controller\_id] = {  'cryptographic\_key': self.generate\_cryptographic\_key(controller\_id),  'authenticated': False  }  self.authenticate\_controllers(controller\_id)  self.monitor\_network\_conditions()  self.adaptive\_control\_loop()  # Phase 1: Identity Verification and Authentication Procedure  def authenticate\_controllers(self, controller\_id):  if self.validate\_controller(controller\_id):  self.establish\_secure\_propagation\_channel(controller\_id)  self.controllers[controller\_id]['authenticated'] = True  def generate\_cryptographic\_key(self, controller\_id):  # Generate a cryptographic key for the controller  return f"CK\_{controller\_id}"  def validate\_controller(self, controller\_id):  # Validate the controller using the cryptographic key  return self.controllers[controller\_id]['cryptographic\_key'] == f"CK\_{controller\_id}"  def establish\_secure\_propagation\_channel(self, controller\_id):  # Establish a secure communication channel with the controller  log.info(f"Secure Propagation Channel established for controller {controller\_id}")  # Phase 2: Monitor Network Conditions  def monitor\_network\_conditions(self):  log.info("Monitoring network conditions...")  # This would involve periodically gathering network data and making adjustments  # Phase 3: Assess Controller States  def assess\_controller\_states(self):  for controller\_id, controller\_info in self.controllers.items():  if self.is\_overloaded(controller\_id):  self.trigger\_dynamic\_adaptation(controller\_id)  def is\_overloaded(self, controller\_id):  # Check if the controller is overloaded  # This is a placeholder; actual logic would involve monitoring load metrics  return False  # Phase 4: Conditions Triggering Dynamic Adaptation  def trigger\_dynamic\_adaptation(self, controller\_id):  if self.network\_traffic\_is\_high() or self.is\_overloaded(controller\_id):  self.adjust\_control\_interface(controller\_id)  self.adjust\_communication\_protocol(controller\_id)  self.enhance\_collaboration(controller\_id)  def network\_traffic\_is\_high(self):  # Placeholder for checking high network traffic  return False  # Phase 5: Control Interface Adjustment  def adjust\_control\_interface(self, controller\_id):  log.info(f"Adjusting control interface for controller {controller\_id}")  # Phase 6: Communication Protocol Adjustment  def adjust\_communication\_protocol(self, controller\_id):  log.info(f"Adjusting communication protocol for controller {controller\_id}")  # Phase 7: Collaboration Enhancement  def enhance\_collaboration(self, controller\_id):  log.info(f"Enhancing collaboration for controller {controller\_id}")  for other\_id in self.controllers:  if other\_id != controller\_id:  log.info(f"Sharing control info between controller {controller\_id} and controller {other\_id}")  # Phase 8: Adaptive Control Loop (ACL)  def adaptive\_control\_loop(self):  log.info("Starting adaptive control loop...")  # Continuously monitor and adapt based on the network conditions and controller states  def launch():  core.registerNew(ADMCApp) |

**Deployment Instructions:**

1. **Ryu Controller**

* **Install Ryu**: Install Ryu using pip: *(pip install ryu).*
* Place the *ADMCApp.py* file in your working directory.
* **Run the ADMCApp**: *(ryu-manager ADMCApp.py).*
* **Simulate Network**: Use Mininet to create a network and connect it to the Ryu controller.
* **Monitor and Adapt**: Use Ryu’s logging features to monitor the execution of the ADMC algorithm and observe the adaptive behaviors in real-time.

1. **POX Controller**

* **Install POX**: Clone the POX repository from GitHub: *(git clone http://github.com/noxrepo/pox).*
* cd pox
* Install dependencies using pip: *(pip install -r requirements.txt).*
* Place the *ADMCApp.py* file in the pox/ext directory.
* **Run the ADMCApp**: *(./pox.py ext.ADMCApp).*
* **Simulate Network**: Use Mininet to create a network and connect it to the POX controller.

1. **DMECP Source Code Explanation and Deployment Instructions**

Here is a P4 program that implements the DMECP (Dynamic Meta-Ensemble with Confidence-Based Prioritization) algorithm as a detection module deployed at the P4 switch:

**Implementation of the DMECP Algorithm as a P4 Application**

|  |
| --- |
| **// Define headers**  **header ethernet\_t {**  **macAddr\_t dstAddr;**  **macAddr\_t srcAddr;**  **bit<16> etherType;**  **}**  **header ipv4\_t {**  **bit<4> version;**  **bit<4> ihl;**  **bit<8> diffserv;**  **bit<16> totalLen;**  **bit<16> identification;**  **bit<3> flags;**  **bit<13> fragOffset;**  **bit<8> ttl;**  **bit<8> protocol;**  **bit<16> hdrChecksum;**  **ipv4Addr\_t srcAddr;**  **ipv4Addr\_t dstAddr;**  **}**  **header tcp\_t {**  **bit<16> srcPort;**  **bit<16> dstPort;**  **bit<32> seqNo;**  **bit<32> ackNo;**  **bit<4> dataOffset;**  **bit<4> reserved;**  **bit<8> flags;**  **bit<16> window;**  **bit<16> checksum;**  **bit<16> urgentPtr;**  **}**  **// Define metadata**  **struct metadata\_t {**  **bit<32> classifier\_score[6]; // Scores from 6 classifiers: RF, DT, kNN, GNB, SVM, BLR**  **bit<8> classifier\_confidence[6]; // Confidence scores for each classifier**  **bit<32> meta\_learner\_output; // Final output from meta-learner**  **bit<1> is\_attack; // Final decision**  **}**  **// Define parser**  **parser MyParser(packet\_in packet,**  **out headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **state start {**  **packet.extract(hdr.ethernet);**  **transition select(hdr.ethernet.etherType) {**  **0x0800: parse\_ipv4;**  **default: accept;**  **}**  **}**  **state parse\_ipv4 {**  **packet.extract(hdr.ipv4);**  **transition select(hdr.ipv4.protocol) {**  **6: parse\_tcp;**  **default: accept;**  **}**  **}**  **state parse\_tcp {**  **packet.extract(hdr.tcp);**  **transition accept;**  **}**  **}**  **// Define tables**  **table feature\_selection\_table {**  **actions = {**  **select\_relevant\_features;**  **\_nop;**  **}**  **size = 1024;**  **default\_action = \_nop();**  **}**  **table classifier\_prediction\_table {**  **actions = {**  **rf\_predict;**  **dt\_predict;**  **knn\_predict;**  **gnb\_predict;**  **svm\_predict;**  **blr\_predict;**  **\_nop;**  **}**  **size = 6; // One entry for each classifier**  **default\_action = \_nop();**  **}**  **table meta\_learner\_table {**  **actions = {**  **combine\_predictions;**  **\_nop;**  **}**  **size = 1;**  **default\_action = \_nop();**  **}**  **// Define actions**  **action select\_relevant\_features() {**  **// Placeholder for feature selection logic**  **// Example: Extract specific features from the packet and store them in metadata**  **}**  **action rf\_predict() {**  **// Placeholder for RF prediction logic**  **meta.classifier\_score[0] = <calculated\_score>;**  **meta.classifier\_confidence[0] = <calculated\_confidence>;**  **}**  **action dt\_predict() {**  **// Placeholder for DT prediction logic**  **meta.classifier\_score[1] = <calculated\_score>;**  **meta.classifier\_confidence[1] = <calculated\_confidence>;**  **}**  **action knn\_predict() {**  **// Placeholder for kNN prediction logic**  **meta.classifier\_score[2] = <calculated\_score>;**  **meta.classifier\_confidence[2] = <calculated\_confidence>;**  **}**  **action gnb\_predict() {**  **// Placeholder for GNB prediction logic**  **meta.classifier\_score[3] = <calculated\_score>;**  **meta.classifier\_confidence[3] = <calculated\_confidence>;**  **}**  **action svm\_predict() {**  **// Placeholder for SVM prediction logic**  **meta.classifier\_score[4] = <calculated\_score>;**  **meta.classifier\_confidence[4] = <calculated\_confidence>;**  **}**  **action blr\_predict() {**  **// Placeholder for BLR prediction logic**  **meta.classifier\_score[5] = <calculated\_score>;**  **meta.classifier\_confidence[5] = <calculated\_confidence>;**  **}**  **action combine\_predictions() {**  **// Combine the predictions using the meta-learner logic**  **bit<32> weighted\_sum = 0;**  **for (int i = 0; i < 6; i++) {**  **weighted\_sum += meta.classifier\_score[i] \* meta.classifier\_confidence[i];**  **}**  **meta.meta\_learner\_output = weighted\_sum / 6; // Example weighted average**  **}**  **action make\_final\_decision() {**  **if (meta.meta\_learner\_output >= <threshold>) {**  **meta.is\_attack = 1;**  **} else {**  **meta.is\_attack = 0;**  **}**  **}**  **// Define control logic**  **control MyIngress(inout headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **apply(feature\_selection\_table);**  **apply(classifier\_prediction\_table);**  **apply(meta\_learner\_table);**  **make\_final\_decision();**  **}**  **// Define deparser**  **control MyDeparser(packet\_out packet,**  **in headers\_t hdr) {**  **apply {**  **packet.emit(hdr.ethernet);**  **packet.emit(hdr.ipv4);**  **packet.emit(hdr.tcp);**  **}**  **}**  **// Switch pipeline**  **control MyVerifyChecksum(inout headers\_t hdr,**  **inout metadata\_t meta) { ... }**  **control MyComputeChecksum(inout headers\_t hdr,**  **inout metadata\_t meta) { ... }**  **V1Switch(MyParser(),**  **MyVerifyChecksum(),**  **MyIngress(),**  **MyEgress(),**  **MyComputeChecksum(),**  **MyDeparser()) main;** |

**Explanation of the P4 Program:**

1. **Header Definitions**:
   * **Ethernet, IPv4, and TCP Headers**: These headers are used to extract the necessary information from each packet for further processing.
2. **Metadata Structure**:
   * **Classifier Scores and Confidence**: Metadata fields store the predictions and confidence scores from the six classifiers (RF, DT, kNN, GNB, SVM, and BLR).
   * **Meta-Learner Output**: Stores the combined output from the meta-learner.
   * **Final Decision**: Stores the final decision on whether the traffic is an attack or normal.
3. **Parser**:
   * The parser extracts the Ethernet, IPv4, and TCP headers from incoming packets.
4. **Tables**:
   * **Feature Selection Table**: This table is used to apply the feature selection process on the packet data.
   * **Classifier Prediction Table**: This table applies each classifier's prediction logic to the packet data.
   * **Meta-Learner Table**: This table combines the predictions from the classifiers based on their confidence scores to generate a final decision.
5. **Actions**:
   * **select\_relevant\_features**: Placeholder for logic that selects relevant features from the packet.
   * **rf\_predict, dt\_predict, etc.**: Placeholders for the prediction logic of each classifier.
   * **combine\_predictions**: Combines predictions using a weighted average or other meta-learner logic.
   * **make\_final\_decision**: Determines whether the traffic is an attack based on the meta-learner's output.
6. **Control Logic**:
   * **MyIngress**: Applies feature selection, classifier predictions, and meta-learner combination in sequence, followed by the final decision-making process.
7. **Deparser**:
   * The deparser reassembles the packet headers and emits the packet after processing.

**Deployment Instructions:**

1. **Compile the P4 Program**: Compile the program using the P4 compiler (*p4c*) to generate the *JSON* file.
2. **Run on P4-Enabled Switch**: Use the behavioral model (*bmv2*) or a hardware P4 switch to deploy the program.
3. **Test and Monitor**: Verify the detection and adaptation capabilities of the DMECP algorithm using live network traffic.
4. **P4-ATM Source Code Explanation and Deployment Instructions**

The following implementation outlines the P4-Enabled Adaptive Traffic Monitoring (P4-ATM) module. This P4 program is designed to run on a P4-enabled switch, where it will monitor various aspects of IoT traffic by capturing and analyzing packet headers, maintaining state information, and dynamically responding to observed traffic patterns.

**Implementation of the Traffic Monitoring Module as a P4 Application**

|  |
| --- |
| **// Define headers**  **header ethernet\_t {**  **macAddr\_t dstAddr;**  **macAddr\_t srcAddr;**  **bit<16> etherType;**  **}**  **header ipv4\_t {**  **bit<4> version;**  **bit<4> ihl;**  **bit<8> diffserv;**  **bit<16> totalLen;**  **bit<16> identification;**  **bit<3> flags;**  **bit<13> fragOffset;**  **bit<8> ttl;**  **bit<8> protocol;**  **bit<16> hdrChecksum;**  **ipv4Addr\_t srcAddr;**  **ipv4Addr\_t dstAddr;**  **}**  **header tcp\_t {**  **bit<16> srcPort;**  **bit<16> dstPort;**  **bit<32> seqNo;**  **bit<32> ackNo;**  **bit<4> dataOffset;**  **bit<4> reserved;**  **bit<8> flags;**  **bit<16> window;**  **bit<16> checksum;**  **bit<16> urgentPtr;**  **}**  **// Define metadata**  **struct metadata\_t {**  **bit<32> packet\_count;**  **bit<64> byte\_count;**  **bit<32> flow\_duration;**  **bit<32> src\_ip\_entropy;**  **bit<32> dst\_ip\_entropy;**  **// Add more fields as required**  **}**  **// Define parser**  **parser MyParser(packet\_in packet,**  **out headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **state start {**  **packet.extract(hdr.ethernet);**  **transition select(hdr.ethernet.etherType) {**  **0x0800: parse\_ipv4;**  **default: accept;**  **}**  **}**  **state parse\_ipv4 {**  **packet.extract(hdr.ipv4);**  **transition select(hdr.ipv4.protocol) {**  **6: parse\_tcp;**  **default: accept;**  **}**  **}**  **state parse\_tcp {**  **packet.extract(hdr.tcp);**  **transition accept;**  **}**  **}**  **// Define tables**  **table packet\_count\_table {**  **actions = {**  **update\_packet\_count;**  **\_nop;**  **}**  **size = 1024;**  **default\_action = \_nop();**  **}**  **table byte\_count\_table {**  **actions = {**  **update\_byte\_count;**  **\_nop;**  **}**  **size = 1024;**  **default\_action = \_nop();**  **}**  **// Define actions**  **action update\_packet\_count() {**  **meta.packet\_count = meta.packet\_count + 1;**  **}**  **action update\_byte\_count() {**  **meta.byte\_count = meta.byte\_count + standard\_metadata.packet\_length;**  **}**  **// Define control logic**  **control MyIngress(inout headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **apply(packet\_count\_table);**  **apply(byte\_count\_table);**  **}**  **// Define deparser**  **control MyDeparser(packet\_out packet,**  **in headers\_t hdr) {**  **apply {**  **packet.emit(hdr.ethernet);**  **packet.emit(hdr.ipv4);**  **packet.emit(hdr.tcp);**  **}**  **}**  **// Switch pipeline**  **control MyVerifyChecksum(inout headers\_t hdr,**  **inout metadata\_t meta) { ... }**  **control MyComputeChecksum(inout headers\_t hdr,**  **inout metadata\_t meta) { ... }**  **V1Switch(MyParser(),**  **MyVerifyChecksum(),**  **MyIngress(),**  **MyEgress(),**  **MyComputeChecksum(),**  **MyDeparser()) main;** |

**Explanation of the Source Code**

1. **Header Definitions**:
   * **Ethernet, IPv4, and TCP Headers**: These headers are defined to capture essential information from each packet, including source and destination MAC addresses, IP addresses, and TCP port numbers.
2. **Metadata Structure**:
   * **Metadata Fields**: The metadata structure contains fields such as *packet\_count, byte\_count, flow\_duration*, and entropy values for source and destination IP addresses. These fields store important statistics and state information used for traffic monitoring.
3. **Parser**:
   * The parser extracts the necessary headers from incoming packets. It starts with the Ethernet header, then processes the IPv4 header, and finally extracts the TCP header if the protocol is TCP.
4. **Tables**:
   * **Packet Count and Byte Count Tables**: These tables keep track of the number of packets and bytes processed. The tables are configured with *update\_packet\_count* and *update\_byte\_count* actions to increment the counters.
5. **Actions**:
   * **Update Actions**: The *update\_packet\_count* action increments the packet count, and the *update\_byte\_count* action adds the packet's length to the byte count.
6. **Control Logic**:
   * **MyIngress**: The main control logic applies the packet and byte count tables to every incoming packet, updating the respective counters.
7. **Deparser**:
   * **MyDeparser**: After processing, the deparser reassembles the packet headers and emits the packet out of the switch.

**Step-by-Step Deployment Instructions**

1. **Install P4 Development Environment**:
   * Install the P4 development environment on your machine. This includes the P4 compiler (*p4c*), behavioral model (*bmv2*), and Mininet (for emulating the network).
2. **Compile the P4 Program**:
   * Compile the P4 program using the P4 compiler: *p4c --target bmv2 --arch v1model -o p4\_atm.json p4\_atm.p4*
   * This command generates a JSON file *(p4\_atm.json)* that describes the P4 pipeline for the switch.
3. **Set Up the Switch**:
   * Use the behavioral model (bmv2) to run a software switch with the compiled P4 program: *sudo simple\_switch --log-console -i 1@<interface> p4\_atm.json*
   * Replace <*interface*> with the appropriate network interface (e.g., eth0).
4. **Test the Program**:
   * Use Mininet to create a network topology with the P4 switch: *sudo mn --custom p4\_mininet\_script.py --topo p4\_topo --controller=remote*
   * Ensure that the traffic monitoring module is working correctly by generating traffic and observing the counters.
5. **Deploy in a Real Network**:
   * Once tested in Mininet, the program can be deployed on hardware P4 switches for monitoring real SD-IoT traffic. Ensure that the switches are configured to run the compiled P4 program and connected to your network.
6. **P4-MCAM Source Code Explanation and Deployment Instructions**

Here's a complete P4 program for deploying the P4-Enabled Multi-Control Adaptive Mitigation (P4-MCAM) algorithm. This implementation represents the mitigation module, which is deployed at the P4 switch.

**Implementation of the P4-MCAM Algorithm as a P4 Application**

|  |
| --- |
| **// Define the headers**  **header ethernet\_t {**  **macAddr\_t dstAddr;**  **macAddr\_t srcAddr;**  **bit<16> ethType;**  **}**  **header ipv4\_t {**  **bit<4> version;**  **bit<4> ihl;**  **bit<8> diffserv;**  **bit<16> totalLen;**  **bit<16> identification;**  **bit<3> flags;**  **bit<13> fragOffset;**  **bit<8> ttl;**  **bit<8> protocol;**  **bit<16> hdrChecksum;**  **ip4Addr\_t srcAddr;**  **ip4Addr\_t dstAddr;**  **}**  **header tcp\_t {**  **bit<16> srcPort;**  **bit<16> dstPort;**  **bit<32> seqNo;**  **bit<32> ackNo;**  **bit<4> dataOffset;**  **bit<6> reserved;**  **bit<6> flags;**  **bit<16> window;**  **bit<16> checksum;**  **bit<16> urgentPtr;**  **}**  **// Define the metadata**  **struct metadata\_t {**  **bit<1> is\_attack;**  **}**  **// Define the parser**  **parser MyParser(packet\_in packet,**  **out ethernet\_t eth\_hdr,**  **out ipv4\_t ip\_hdr,**  **out tcp\_t tcp\_hdr,**  **inout metadata\_t meta) {**  **state start {**  **packet.extract(eth\_hdr);**  **transition select(eth\_hdr.ethType) {**  **0x0800: parse\_ipv4;**  **default: accept;**  **}**  **}**  **state parse\_ipv4 {**  **packet.extract(ip\_hdr);**  **transition select(ip\_hdr.protocol) {**  **6: parse\_tcp;**  **default: accept;**  **}**  **}**  **state parse\_tcp {**  **packet.extract(tcp\_hdr);**  **transition accept;**  **}**  **}**  **// Define tables**  **table attack\_detection {**  **key = {**  **ip\_hdr.srcAddr: exact;**  **ip\_hdr.dstAddr: exact;**  **tcp\_hdr.srcPort: exact;**  **tcp\_hdr.dstPort: exact;**  **}**  **actions = {**  **set\_attack\_flag;**  **no\_op;**  **}**  **size = 1024;**  **default\_action = no\_op();**  **}**  **table mitigation\_actions {**  **key = {**  **meta.is\_attack: exact;**  **}**  **actions = {**  **drop\_packet;**  **forward\_packet;**  **}**  **size = 2;**  **default\_action = forward\_packet();**  **}**  **// Define actions**  **action set\_attack\_flag() {**  **meta.is\_attack = 1;**  **}**  **action no\_op() {**  **// Do nothing**  **}**  **action drop\_packet() {**  **mark\_to\_drop();**  **}**  **action forward\_packet() {**  **// Forward packet to the next hop (default action)**  **}**  **// Define the control block**  **control MyIngress(inout ethernet\_t eth\_hdr,**  **inout ipv4\_t ip\_hdr,**  **inout tcp\_t tcp\_hdr,**  **inout metadata\_t meta) {**  **apply {**  **attack\_detection.apply();**  **mitigation\_actions.apply();**  **}**  **}**  **// Define the deparser**  **control MyDeparser(packet\_out packet,**  **in ethernet\_t eth\_hdr,**  **in ipv4\_t ip\_hdr,**  **in tcp\_t tcp\_hdr) {**  **apply {**  **packet.emit(eth\_hdr);**  **packet.emit(ip\_hdr);**  **packet.emit(tcp\_hdr);**  **}**  **}**  **// Define the pipeline**  **control MyPipeline {**  **MyParser() parser;**  **MyIngress() ingress;**  **MyDeparser() deparser;**  **apply {**  **parser.apply();**  **ingress.apply();**  **deparser.apply();**  **}**  **}**  **MyPipeline() main;** |

**Explanation of the P4 Program:**

1. **Headers Definition**: The program starts by defining the Ethernet, IPv4, and TCP headers, which will be extracted from the incoming packets.
2. **Metadata Definition**: A metadata structure ***metadata\_t*** is defined, which contains a single bit ***is\_attack*** to flag whether a packet is considered part of an attack.
3. **Parser**: The ***MyParser*** function extracts the Ethernet, IPv4, and TCP headers from incoming packets. It starts by extracting the Ethernet header and then transitions based on the ***ethType*** to either extract IPv4 and TCP headers or accept the packet.
4. **Tables**:

* **attack\_detection**: This table uses the source IP, destination IP, source port, and destination port as keys to identify known attack patterns. If an attack is detected, the ***set\_attack\_flag*** action is applied.
* **mitigation\_actions**: This table checks the ***is\_attack*** flag in the metadata. If the flag is set, the packet is dropped; otherwise, it is forwarded as usual.

1. **Actions**:

* **set\_attack\_flag**: Sets the ***is\_attack*** metadata field to 1, indicating that the packet is part of an attack.
* **no\_op**: Does nothing, used as a default action.
* **drop\_packet**: Marks the packet to be dropped.
* **forward\_packet**: Forwards the packet to the next hop, the default action for non-attack packets.

1. **Control Block (MyIngress)**: This control block applies the ***attack\_detection*** and ***mitigation\_actions*** tables. Based on the outcomes, the packet is either forwarded or dropped.
2. **Deparser**: The ***MyDeparser*** function reassembles the packet from the Ethernet, IPv4, and TCP headers before sending it out.
3. **Pipeline**: The ***MyPipeline*** control connects the parser, ingress, and ***deparser***, forming the packet processing pipeline.

**Step-by-Step Deployment Instructions:**

1. **Setup P4 Development Environment**: Ensure that we have the P4 development environment set up, including tools like P4C (P4 compiler) and BMv2 (Behavioral Model v2).
2. **Write the P4 Program**: Copy the P4 code provided above into a ***.p4*** file (***e.g., p4\_mcam.p4***).
3. **Compile the P4 Program**: Use the P4C compiler to compile the P4 program: ***p4c --target bmv2 --arch v1model p4\_mcam.p4 -o p4\_mcam.json***
4. **Deploy on a P4 Switch**: Load the compiled JSON file onto your P4-enabled switch or a BMv2 software switch: ***simple\_switch --log-console --json p4\_mcam.json***
5. **Configure the Switch**: Use a control plane application or a simple runtime CLI to populate the ***attack\_detection*** and ***mitigation\_actions*** tables with appropriate entries.
6. **Monitor and Adapt**: Continuously monitor the network using the deployed P4-MCAM module. Adjust the control plane logic and table entries as new attack patterns are identified or as the network environment changes.
7. **Logging and Reporting**: Ensure that logs are generated for all mitigation actions, including packet drops and forwarding decisions, and store them for future analysis.