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.
4. **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.