Resume :

To transform the Ad-hoc On-Demand Distance Vector (AODV) routing protocol into the Adaptive Routing Protocol for Mobile Edge Computing-based IoT Networks (ARPMEC) described in the provided document, a detailed and systematic approach is required. ARPMEC introduces clustering, link quality prediction, and adaptive routing tailored for IoT networks with Mobile Edge Computing (MEC) infrastructure, which differs significantly from AODV's reactive, distance-vector-based routing. Below, I outline a comprehensive, step-by-step process to modify AODV to implement ARPMEC, including the specific files and components of AODV to modify, the algorithms from the ARPMEC paper to integrate, and the detailed logic to achieve the desired functionality. Since AODV is typically implemented in network simulators like ns-2, ns-3, or real-world systems like Linux kernel modules, I’ll assume an ns-3 implementation for specificity, but the concepts apply broadly.

---

### Overview of AODV and ARPMEC Differences

\*\*AODV\*\*:

- \*\*Type\*\*: Reactive (on-demand) routing protocol for Mobile Ad-hoc Networks (MANETs).

- \*\*Operation\*\*: Uses Route Request (RREQ) and Route Reply (RREP) messages to discover paths only when needed. Maintains routes via sequence numbers and handles route breaks with Route Error (RERR) messages.

- \*\*Key Features\*\*: No clustering, no link quality prediction, no MEC integration, and assumes homogeneous nodes with basic mobility support.

- \*\*Files in ns-3\*\*: Typically found in `src/aodv/` (e.g., `aodv-routing-protocol.cc`, `aodv-rtable.cc`, `aodv-packet.h`).

\*\*ARPMEC\*\*:

- \*\*Type\*\*: Adaptive routing protocol for IoT networks with MEC infrastructure.

- \*\*Operation\*\*: Consists of two phases:

1. \*\*Clustering Phase\*\*: Uses link quality prediction (LQE) based on RSSI and PDR to form clusters with Cluster Heads (CHs) and Cluster Members (CMs).

2. \*\*Adaptive Routing Phase\*\*: Routes data through clusters, considering node mobility and energy efficiency, with MEC servers (Gateways) facilitating inter-cluster communication.

- \*\*Key Features\*\*: Clustering, LQE using machine learning, energy-aware routing, mobility adaptation, and MEC integration.

- \*\*Algorithms\*\*: Algorithm 1 (main protocol), Algorithm 2 (clustering), Algorithm 3 (adaptive routing).

\*\*Challenges in Modification\*\*:

- AODV lacks clustering and LQE mechanisms.

- AODV does not integrate MEC servers or handle energy-aware routing.

- ARPMEC’s adaptive routing requires dynamic topology updates based on mobility and link quality, unlike AODV’s static route maintenance.

- Implementing ARPMEC’s machine learning-based LQE and clustering logic in a simulator like ns-3 requires additional modules.

---

### Step-by-Step Process to Modify AODV into ARPMEC

#### Step 1: Understand AODV Implementation in ns-3

- \*\*Directory\*\*: `src/aodv/`

- \*\*Key Files\*\*:

- `aodv-routing-protocol.cc/h`: Main routing logic, handles RREQ, RREP, RERR.

- `aodv-rtable.cc/h`: Routing table management.

- `aodv-packet.cc/h`: Packet formats for AODV messages.

- `aodv-neighbor.cc/h`: Neighbor management.

- \*\*Functionality to Modify\*\*:

- Route discovery (replace with clustering and adaptive routing).

- Neighbor management (add LQE and clustering logic).

- Packet formats (add new message types for ARPMEC).

- Add energy and mobility models.

#### Step 2: Define ARPMEC’s Requirements

- \*\*Clustering\*\*: Implement Algorithm 2 to form clusters using LQE.

- \*\*Link Quality Prediction\*\*: Integrate RSSI and PDR metrics with a machine learning model (e.g., Random Forest as suggested in the paper).

- \*\*Adaptive Routing\*\*: Implement Algorithm 3 for data routing within and across clusters.

- \*\*MEC Integration\*\*: Model MEC servers as Gateways (GWs) for inter-cluster communication.

- \*\*Energy Model\*\*: Use the energy model from Equation 8 (Page 10) to track node energy consumption.

- \*\*Mobility Support\*\*: Use GPS-based mobility tracking (assumed in ARPMEC).

- \*\*Time-Slotted Channel Hopping (TSCH)\*\*: Implement a TSCH-based MAC layer for channel management.

#### Step 3: Modify AODV Packet Formats

ARPMEC requires new message types for clustering and routing. Modify `aodv-packet.h/cc` to include ARPMEC-specific packets.

- \*\*New Packet Types\*\* (based on Algorithm 2 and 3):

- \*\*HELLO\*\*: For LQE, contains node ID and channel info.

- \*\*JOIN\*\*: Sent by nodes to join a cluster, includes node ID and chosen CH ID.

- \*\*CH\_NOTIFICATION\*\*: Sent by CH to GW to announce cluster members.

- \*\*CLUSTER\_LIST\*\*: Sent by GW to broadcast cleaned cluster assignments.

- \*\*DATA\*\*: For adaptive data routing, includes source, destination, and cluster info.

- \*\*ABDICATE\*\*: Sent by CH to resign due to low energy.

- \*\*Modification in `aodv-packet.h`\*\*:

```x-c++hdr

enum ArpmecPacketType {

ARPMEC\_HELLO = 1,

ARPMEC\_JOIN = 2,

ARPMEC\_CH\_NOTIFICATION = 3,

ARPMEC\_CLUSTER\_LIST = 4,

ARPMEC\_DATA = 5,

ARPMEC\_ABDICATE = 6

};

struct ArpmecHelloHeader {

uint32\_t nodeId;

uint8\_t channelId;

};

struct ArpmecJoinHeader {

uint32\_t nodeId;

uint32\_t chId;

};

struct ArpmecChNotificationHeader {

uint32\_t chId;

std::vector<uint32\_t> clusterMembers;

};

struct ArpmecClusterListHeader {

std::vector<std::pair<uint32\_t, std::vector<uint32\_t>>> clusters; // CH ID and member IDs

std::vector<Vector> clusterCoordinates; // Geographical coordinates

};

struct ArpmecDataHeader {

uint32\_t sourceId;

uint32\_t destId;

uint32\_t chId; // Current cluster head

};

struct ArpmecAbdicateHeader {

uint32\_t chId;

};

```

- \*\*Modification in `aodv-packet.cc`\*\*:

- Add serialization/deserialization methods for each new header.

- Update `AodvPacket::AddHeader` to support ARPMEC headers.

#### Step 4: Implement Link Quality Prediction Module

ARPMEC uses LQE based on RSSI and PDR, with a Random Forest (RF) model for prediction (Page 4). Create a new module for LQE.

- \*\*New File\*\*: `arpmec-lqe.cc/h` in `src/aodv/`

- \*\*Functionality\*\*:

- Collect RSSI and PDR for each neighbor link.

- Use a simplified RF model (or pre-trained weights for simulation) to predict link quality score (PDR, RSSI pair).

- Store link quality scores for clustering decisions.

- \*\*Code Outline\*\*:

```x-c++hdr

#ifndef ARPMEC\_LQE\_H

#define ARPMEC\_LQE\_H

#include "ns3/object.h"

#include "ns3/mac48-address.h"

#include <vector>

#include <map>

namespace ns3 {

struct LinkQuality {

double pdr; // Packet Delivery Ratio

double rssi; // Received Signal Strength Indicator

double score; // Predicted link quality

};

class ArpmecLqe : public Object {

public:

static TypeId GetTypeId();

ArpmecLqe();

void UpdateLinkQuality(Mac48Address neighbor, double rssi, bool packetReceived);

LinkQuality GetLinkQuality(Mac48Address neighbor);

Mac48Address GetBestNeighbor();

private:

std::map<Mac48Address, LinkQuality> m\_linkQualities;

uint32\_t m\_helloCount; // Track HELLO messages for PDR

uint32\_t m\_expectedHello; // R from paper

double PredictScore(double pdr, double rssi); // Simplified RF model

};

}

#endif

```

```x-c++src

#include "arpmec-lqe.h"

#include "ns3/log.h"

#include <cmath>

namespace ns3 {

NS\_LOG\_COMPONENT\_DEFINE("ArpmecLqe");

NS\_OBJECT\_ENSURE\_REGISTERED(ArpmecLqe);

TypeId ArpmecLqe::GetTypeId() {

static TypeId tid = TypeId("ns3::ArpmecLqe")

.SetParent<Object>()

.AddConstructor<ArpmecLqe>();

return tid;

}

ArpmecLqe::ArpmecLqe() : m\_helloCount(0), m\_expectedHello(100) {} // R=100 from paper

void ArpmecLqe::UpdateLinkQuality(Mac48Address neighbor, double rssi, bool packetReceived) {

auto& lq = m\_linkQualities[neighbor];

lq.rssi = (lq.rssi \* m\_helloCount + rssi) / (m\_helloCount + 1); // Average RSSI

if (packetReceived) {

lq.pdr = (lq.pdr \* m\_helloCount + 1.0) / (m\_helloCount + 1);

} else {

lq.pdr = (lq.pdr \* m\_helloCount + 0.0) / (m\_helloCount + 1);

}

m\_helloCount++;

if (m\_helloCount >= m\_expectedHello) {

lq.score = PredictScore(lq.pdr, lq.rssi);

m\_helloCount = 0; // Reset for next round

}

}

double ArpmecLqe::PredictScore(double pdr, double rssi) {

// Simplified RF model: weighted sum (replace with actual RF if available)

return 0.6 \* pdr + 0.4 \* (rssi / -100.0); // Normalize RSSI

}

LinkQuality ArpmecLqe::GetLinkQuality(Mac48Address neighbor) {

return m\_linkQualities[neighbor];

}

Mac48Address ArpmecLqe::GetBestNeighbor() {

Mac48Address best;

double maxScore = -1.0;

for (const auto& pair : m\_linkQualities) {

if (pair.second.score > maxScore) {

maxScore = pair.second.score;

best = pair.first;

}

}

return best;

}

}

```

#### Step 5: Implement Clustering (Algorithm 2)

Modify `aodv-routing-protocol.cc/h` to include clustering logic based on Algorithm 2 (Page 6-7).

- \*\*Changes in `aodv-routing-protocol.h`\*\*:

- Add cluster state (CH or CM), energy level, and LQE module.

- Add methods for clustering and CH election.

```x-c++hdr

#include "arpmec-lqe.h"

#include "ns3/mobility-model.h"

#include "ns3/energy-source.h"

namespace ns3 {

class ArpmecRoutingProtocol : public AodvRoutingProtocol {

public:

static TypeId GetTypeId();

ArpmecRoutingProtocol();

void StartClustering();

void HandleHello(Ptr<Packet> packet, Ipv4Address src);

void HandleJoin(Ptr<Packet> packet, Ipv4Address src);

void HandleAbdicate(Ptr<Packet> packet);

void HandleChNotification(Ptr<Packet> packet);

void HandleClusterList(Ptr<Packet> packet);

private:

enum NodeRole { CM, CH };

NodeRole m\_role;

uint32\_t m\_nodeId;

uint32\_t m\_chId;

std::vector<uint32\_t> m\_clusterMembers; // For CH

Ptr<ArpmecLqe> m\_lqe;

Ptr<EnergySource> m\_energySource;

Ptr<MobilityModel> m\_mobility; // For GPS

uint32\_t m\_channelCount; // C from paper

uint32\_t m\_maxHub; // HUB\_max

double m\_energyThreshold; // zeta

void ElectNewCh();

void BroadcastHello();

void SendJoin(uint32\_t chId);

void SendChNotification();

void SendAbdicate();

};

}

```

- \*\*Changes in `aodv-routing-protocol.cc`\*\*:

- Implement Algorithm 2 logic:

1. Broadcast HELLO messages on each channel (Line 4).

2. Record RSSI and PDR for LQE (Lines 8-10).

3. Predict link quality and select best neighbor (Lines 11-12).

4. Send JOIN message (Line 13).

5. CH election and notification (Lines 19-20).

6. Cluster cleaning via GW (Line 21).

```x-c++src

#include "aodv-routing-protocol.h"

#include "ns3/simulator.h"

#include "ns3/log.h"

namespace ns3 {

NS\_LOG\_COMPONENT\_DEFINE("ArpmecRoutingProtocol");

NS\_OBJECT\_ENSURE\_REGISTERED(ArpmecRoutingProtocol);

TypeId ArpmecRoutingProtocol::GetTypeId() {

static TypeId tid = TypeId("ns3::ArpmecRoutingProtocol")

.SetParent<AodvRoutingProtocol>()

.AddConstructor<ArpmecRoutingProtocol>();

return tid;

}

ArpmecRoutingProtocol::ArpmecRoutingProtocol()

: m\_role(CM), m\_nodeId(0), m\_chId(0), m\_channelCount(16), m\_maxHub(50), m\_energyThreshold(0.5) {

m\_lqe = CreateObject<ArpmecLqe>();

m\_energySource = GetObject<EnergySource>();

m\_mobility = GetObject<MobilityModel>();

}

void ArpmecRoutingProtocol::StartClustering() {

m\_nodeId = GetNode()->GetId() + 1; // 1-based IDs

Simulator::Schedule(Seconds(0), &ArpmecRoutingProtocol::BroadcastHello, this);

}

void ArpmecRoutingProtocol::BroadcastHello() {

for (uint32\_t c = 1; c <= m\_channelCount; c++) {

for (uint32\_t r = 1; r <= 100; r++) { // R=100

ArpmecHelloHeader hello;

hello.nodeId = m\_nodeId;

hello.channelId = c;

Ptr<Packet> packet = Create<Packet>();

packet->AddHeader(hello);

SendPacket(packet, ARPMEC\_HELLO);

Simulator::Schedule(MilliSeconds(1), &ArpmecRoutingProtocol::BroadcastHello, this);

}

}

Simulator::Schedule(Seconds(m\_channelCount \* 100 \* GetNode()->GetNDevices()),

&ArpmecRoutingProtocol::SendJoin, this, m\_lqe->GetBestNeighbor().Get());

}

void ArpmecRoutingProtocol::HandleHello(Ptr<Packet> packet, Ipv4Address src) {

ArpmecHelloHeader hello;

packet->RemoveHeader(hello);

double rssi = CalculateRssi(packet); // Implement based on PHY layer

m\_lqe->UpdateLinkQuality(Mac48Address::ConvertFrom(src), rssi, true);

}

void ArpmecRoutingProtocol::SendJoin(uint32\_t chId) {

ArpmecJoinHeader join;

join.nodeId = m\_nodeId;

join.chId = chId;

Ptr<Packet> packet = Create<Packet>();

packet->AddHeader(join);

SendPacket(packet, ARPMEC\_JOIN);

}

void ArpmecRoutingProtocol::HandleJoin(Ptr<Packet> packet, Ipv4Address src) {

ArpmecJoinHeader join;

packet->RemoveHeader(join);

if (join.chId == m\_nodeId) {

m\_clusterMembers.push\_back(join.nodeId);

if (m\_role == CM) {

m\_role = CH;

Simulator::Schedule(Seconds(1), &ArpmecRoutingProtocol::SendChNotification, this);

}

}

}

void ArpmecRoutingProtocol::SendChNotification() {

ArpmecChNotificationHeader notif;

notif.chId = m\_nodeId;

notif.clusterMembers = m\_clusterMembers;

Ptr<Packet> packet = Create<Packet>();

packet->AddHeader(notif);

SendPacket(packet, ARPMEC\_CH\_NOTIFICATION);

}

void ArpmecRoutingProtocol::HandleChNotification(Ptr<Packet> packet) {

// GW logic: forward to cloud server for cluster cleaning

// Cloud server sends CLUSTER\_LIST back

}

void ArpmecRoutingProtocol::HandleClusterList(Ptr<Packet> packet) {

ArpmecClusterListHeader list;

packet->RemoveHeader(list);

for (const auto& cluster : list.clusters) {

if (std::find(cluster.second.begin(), cluster.second.end(), m\_nodeId) != cluster.second.end()) {

m\_chId = cluster.first;

break;

}

}

}

void ArpmecRoutingProtocol::SendAbdicate() {

if (m\_role == CH && m\_energySource->GetRemainingEnergy() < m\_energyThreshold) {

ArpmecAbdicateHeader abdicate;

abdicate.chId = m\_nodeId;

Ptr<Packet> packet = Create<Packet>();

packet->AddHeader(abdicate);

SendPacket(packet, ARPMEC\_ABDICATE);

m\_role = CM;

m\_clusterMembers.clear();

}

}

void ArpmecRoutingProtocol::HandleAbdicate(Ptr<Packet> packet) {

ArpmecAbdicateHeader abdicate;

packet->RemoveHeader(abdicate);

if (m\_chId == abdicate.chId) {

Simulator::Schedule(Seconds(1), &ArpmecRoutingProtocol::ElectNewCh, this);

}

}

void ArpmecRoutingProtocol::ElectNewCh() {

// Broadcast energy and ID, select highest energy/ID as CH (Lines 15-22)

// Simplified: assume local election within cluster

}

```

#### Step 6: Implement Adaptive Routing (Algorithm 3)

Replace AODV’s route discovery with ARPMEC’s adaptive routing logic (Page 8-9).

- \*\*Changes in `aodv-routing-protocol.cc`\*\*:

- Implement Algorithm 3:

1. CH checks energy and listens for items (Lines 4-7).

2. CH forwards non-cluster items to GW (Line 9).

3. CM listens for CH abdication and triggers CH election (Lines 13-22).

4. Nodes broadcast data items (Line 23).

- Modify routing table to store cluster-based routes (CH and GW paths).

```x-c++src

void ArpmecRoutingProtocol::RouteData(Ptr<Packet> packet, Ipv4Address dest) {

ArpmecDataHeader data;

packet->RemoveHeader(data);

if (m\_role == CH) {

if (m\_energySource->GetRemainingEnergy() < m\_energyThreshold) {

SendAbdicate();

return;

}

// Check if dest is in cluster

if (std::find(m\_clusterMembers.begin(), m\_clusterMembers.end(), dest.Get()) != m\_clusterMembers.end()) {

SendPacket(packet, ARPMEC\_DATA); // Direct to CM

} else {

// Forward to GW

SendToGateway(packet);

}

} else {

// CM forwards to CH

SendToCh(packet);

}

}

void ArpmecRoutingProtocol::SendToCh(Ptr<Packet> packet) {

ArpmecDataHeader data;

packet->RemoveHeader(data);

data.chId = m\_chId;

packet->AddHeader(data);

SendPacket(packet, ARPMEC\_DATA);

}

void ArpmecRoutingProtocol::SendToGateway(Ptr<Packet> packet) {

// Forward to MEC server (GW)

// Assume GW is reachable via known route

}

```

#### Step 7: Integrate Energy Model

Use Equation 8 (Page 10) to track energy consumption.

- \*\*New File\*\*: `arpmec-energy-model.cc/h`

- \*\*Logic\*\*:

- Calculate energy for transmission, reception, and amplification.

- Update energy source on each packet send/receive.

```x-c++hdr

#ifndef ARPMEC\_ENERGY\_MODEL\_H

#define ARPMEC\_ENERGY\_MODEL\_H

#include "ns3/energy-source.h"

namespace ns3 {

class ArpmecEnergyModel : public Object {

public:

static TypeId GetTypeId();

ArpmecEnergyModel();

void UpdateEnergy(Ptr<Packet> packet, bool isTx, double distance);

double GetTotalEnergyConsumed() const;

private:

double m\_transmitEnergy; // e\_t = 0.03J

double m\_receiveEnergy; // e\_r = 0.02J

double m\_amplifyEnergy; // e\_amp = 0.01J

double m\_totalEnergy;

double m\_energyParameter; // Q

};

}

#endif

```

```x-c++src

#include "arpmec-energy-model.h"

#include "ns3/log.h"

namespace ns3 {

NS\_LOG\_COMPONENT\_DEFINE("ArpmecEnergyModel");

NS\_OBJECT\_ENSURE\_REGISTERED(ArpmecEnergyModel);

TypeId ArpmecEnergyModel::GetTypeId() {

static TypeId tid = TypeId("ns3::ArpmecEnergyModel")

.SetParent<Object>()

.AddConstructor<ArpmecEnergyModel>();

return tid;

}

ArpmecEnergyModel::ArpmecEnergyModel()

: m\_transmitEnergy(0.03), m\_receiveEnergy(0.02), m\_amplifyEnergy(0.01), m\_totalEnergy(0.0), m\_energyParameter(1.0) {}

void ArpmecEnergyModel::UpdateEnergy(Ptr<Packet> packet, bool isTx, double distance) {

uint32\_t n = packet->GetSize(); // Approximate items

double energy = m\_energyParameter \* n \* (m\_transmitEnergy + m\_amplifyEnergy + distance \* distance) + m\_receiveEnergy \* n;

if (!isTx) {

energy = m\_receiveEnergy \* n; // Only reception for Rx

}

m\_totalEnergy += energy;

Ptr<EnergySource> source = GetObject<EnergySource>();

source->DecreaseRemainingEnergy(energy);

}

double ArpmecEnergyModel::GetTotalEnergyConsumed() const {

return m\_totalEnergy;

}

```

#### Step 8: Implement TSCH MAC Layer

ARPMEC assumes a TSCH-based MAC layer (Page 6). Modify or replace AODV’s default MAC (e.g., IEEE 802.11) with a TSCH implementation.

- \*\*Approach\*\*:

- Use ns-3’s `sixlowpan` module or create a custom TSCH MAC.

- Assign slots for HELLO, JOIN, and DATA messages.

- Support multiple channels (C=16).

#### Step 9: Integrate MEC Servers (Gateways)

- \*\*Model GWs\*\*:

- Create a new node type for MEC servers in ns-3.

- Assign static IP addresses and high energy capacity.

- \*\*Logic\*\*:

- GWs receive CH\_NOTIFICATION and forward to a cloud server (simulated as another node).

- Cloud server cleans clusters and sends CLUSTER\_LIST via GWs.

#### Step 10: Update Simulation Setup

- \*\*Simulation Script\*\*:

- Configure nodes with ARPMEC protocol, energy model, and TSCH MAC.

- Set parameters from Table 3 (Page 10): N=500, C=16, R=100, Δ=10,000, T=200.

- Use mobility model (e.g., RandomWaypointMobilityModel) with GPS support.

```x-c++src

#include "ns3/core-module.h"

#include "ns3/network-module.h"

#include "ns3/mobility-module.h"

#include "ns3/energy-module.h"

#include "ns3/aodv-module.h"

using namespace ns3;

int main(int argc, char \*argv[]) {

NodeContainer nodes;

nodes.Create(500); // N=500

NodeContainer gateways;

gateways.Create(5); // K=5 MEC servers

// Install mobility

MobilityHelper mobility;

mobility.SetMobilityModel("ns3::RandomWaypointMobilityModel");

mobility.Install(nodes);

// Install energy source

BasicEnergySourceHelper energySourceHelper;

energySourceHelper.Set("BasicEnergySourceInitialEnergy", DoubleValue(100.0));

energySourceHelper.Install(nodes);

// Install ARPMEC protocol

AodvHelper aodv;

aodv.Set("Protocol", StringValue("ns3::ArpmecRoutingProtocol"));

InternetStackHelper stack;

stack.SetRoutingHelper(aodv);

stack.Install(nodes);

stack.Install(gateways);

// Configure TSCH MAC

// Custom implementation required

// Schedule clustering

for (uint32\_t i = 0; i < nodes.GetN(); i++) {

Ptr<ArpmecRoutingProtocol> proto = nodes.Get(i)->GetObject<ArpmecRoutingProtocol>();

Simulator::Schedule(Seconds(0), &ArpmecRoutingProtocol::StartClustering, proto);

}

Simulator::Stop(Seconds(2000)); // Run for 200 rounds

Simulator::Run();

Simulator::Destroy();

return 0;

}

```

#### Step 11: Test and Validate

- \*\*Metrics\*\* (Page 10):

- Energy consumption (compare with ICP, ISCP, NESEPRIN, ABBPWHN).

- Network delay, throughput, success rate.

- \*\*Scenarios\*\*:

- Vary R (25, 50, 75, 100) and C (1, 4, 8, 16).

- Test with 125, 250, 375, 500 nodes.

- \*\*Validation\*\*:

- Ensure clustering forms as per Figure 2.

- Verify ARPMEC’s energy efficiency (Figure 6).

---

### Summary of Modified Files

1. \*\*aodv-packet.h/cc\*\*: Add ARPMEC packet types (HELLO, JOIN, etc.).

2. \*\*arpmec-lqe.h/cc\*\*: New module for LQE.

3. \*\*aodv-routing-protocol.h/cc\*\*: Main ARPMEC logic (clustering, routing).

4. \*\*arpmec-energy-model.h/cc\*\*: Energy consumption tracking.

5. \*\*arpmec-simulation.cc\*\*: Simulation setup.

6. \*\*MAC Layer\*\*: Custom TSCH implementation (optional, depending on ns-3 support).

---

### Feasibility

Yes, it is possible to modify AODV to implement ARPMEC. The process involves significant changes to AODV’s core components but leverages ns-3’s modular design. Key challenges include:

- Implementing a realistic RF model for LQE (simplified in this example).

- Integrating TSCH, which may require a custom MAC layer.

- Simulating MEC servers and cloud interactions in ns-3.

By following the detailed steps above, you can achieve a complete ARPMEC implementation, ensuring all aspects of the protocol (clustering, LQE, adaptive routing, energy efficiency) are integrated into AODV’s framework.