1 Work

1.1 Simulation Setup

The simulation is designed to work with input parameters to assess the performance of the LRA protocol. The key simulation parameters include a variable **number of nodes**, deployed within a square **area** of chosen dimensions. Nodes are positioned uniformly at random to ensure a realistic network topology.

To introduce mobility, a **random walk model** is employed, enabling nodes to move freely within the designated area. The **speed** attribute of the walk model follows a Pareto distribution to simulate a context in which a few nodes move with a vehicle while the majority move by foot. This setup reflects real-world **disaster recovery** scenarios where emergency responders or supply vehicles travel faster than individuals navigating on foot. At the beginning of the simulation, a **sink node** is designated, serving as the central recipient of network messages.

Each node randomly transmits a fixed number of **echo messages** to the sink exploiting NS3 **UdpEchoServerHelper** and **UdpEchoServerClient** fuctionalities and emulating data communication within the network. The total simulation duration is set from a parameter provided by the user.

The **LRA routing protocol** governs routing decisions, utilizing **link reversal mechanisms** to dynamically adapt to changes in node positions. The success or failure of packet transmissions is monitored, with **packet loss** recorded as a critical metric.

Here is a comprehensive list of the simulation parameters:

- Number of Nodes
- Number of Packets
- Area Side Length
- Simulation Time

```
void LraExample::SetupNodes(int nNodes, double squareSize, double totalTime){
    ns3::NodeContainer nodes(nNodes);
    MobilityHelper mobility;
    mobility.SetPositionAllocator(
      "ns3::RandomRectanglePositionAllocator",
      "X", StringValue("ns3::UniformRandomVariable[Min=0|Max="+squareSize+"]"),
      "Y", StringValue("ns3::UniformRandomVariable[Min=0|Max="+squareSize+"]")
    );

    mobility.SetMobilityModel(
      "ns3::RandomWalk2dMobilityModel",
      "Mode", StringValue("Time"),
      "Time", StringValue(std::to_string(totalTime) + "s"),
      // Pareto: mean = 12, median = 8, min = 6, max = 80
      "Speed", StringValue("ns3::ParetoRandomVariable[Bound=80|Scale=6|Shape=2]"),
      "Bounds", RectangleValue(Rectangle(0, squareSize, 0, squareSize))
    );
    mobility.Install(nodes);
}
```

Listing 1: Simulation Node Initialization

```
void LraExample::SetupNetwork(){
    for (uint32_t i = 0; i < nodes.GetN(); ++i)
    {
        Ptr<Node> node = nodes.Get(i);
        Ptr<LraRoutingProtocol> lraRouting = node->GetObject<LraRoutingProtocol>();
        lraRouting->InitializeNode(m_sinkAddress);
    }

    int echoPort = 9;
    UdpEchoServerHelper echoServer(echoPort);
    ApplicationContainer echoServerApps = echoServer.Install(nodes);
    echoServerApps.Start(Seconds(0));
    echoServerApps.Stop(Seconds(totalTime));

    UdpEchoClientHelper echoClient(Address(m_sinkAddress), echoPort);
    echoClient.SetAttribute("MaxPackets", UintegerValue(n_packets));
    echoClient.SetAttribute("Interval", TimeValue(Seconds(1.0)));
    echoClient.SetAttribute("PacketSize", UintegerValue(32));

    for (uint32_t i = 0; i < nodes.GetN() - 1; ++i)
    {
        auto app = echoClient.Install(nodes.Get(i));
        int randDelay = rand() % 1000;
        app.Start(Seconds(startDelay + randDelay));
        app.Stop(Seconds(totalTime));
    }

    // Trace package send and receive to calculate loss
    Config::Connect(
        "/NodeList/*/ApplicationList/*/$ns3::UdpEchoServer/RxWithAddresses",
        MakeCallback(&LraExample::LogMessageResponse, this)
    );
    Config::Connect(
        "/NodeList/*/ApplicationList/*/$ns3::UdpEchoClient/TxWithAddresses",
        MakeCallback(&LraExample::LogMessageSend, this)
    );
}
```

Listing 2: Application Simulation Initialization

1.2 Network Configuration

The network is established using an **IEEE 802.11ac WiFi standard** with an **ad hoc** configuration. The wireless network is configured as follows:

- The MAC layer is set to **AdhocWifiMac**, ensuring decentralized network operation.
- The physical layer uses the **YansWifiPhy** model, with parameters set for carrier sense threshold and sensitivity at **zero**, optimizing the detection of nearby transmissions.
- The wireless channel follows the **Default YansWifiChannelHelper** configuration.
- The remote station manager utilizes **IdealWifiManager**, optimizing the transmission rate selection.

The Internet stack is deployed as follows:

- Nodes are assigned IP addresses within the **10.0.0.0/8** subnet.
- The sink node is designated as the last node in the list and assigned the highest IP address.

```
void LraExample::SetupNetwork(){
    WifiMacHelper wifiMac("ns3::AdhocWifiMac");

    YansWifiPhyHelper wifiPhy;
    wifiPhy.Set("CcaEdThreshold", DoubleValue (0));
    wifiPhy.Set("CcaSensitivity", DoubleValue (0));

    auto wifiChannel = YansWifiChannelHelper::Default();
    wifiPhy.SetChannel(wifiChannel.Create());

    WifiHelper wifi;
    wifi.SetStandard(WIFI_STANDARD_80211ac);
    wifi.SetRemoteStationManager("ns3::IdealWifiManager");
    wifi.Install(wifiPhy, wifiMac, nodes);

    stack.SetRoutingHelper(LraHelper());
    stack.Install(nodes);

    Ipv4AddressHelper address;
    address.SetBase("10.0.0.0", "255.0.0.0");
    ipv4Interfaces = address.Assign(netDevices);
    m_sinkAddress = ipv4Interfaces.GetAddress(nodes.GetN() - 1);
}
```

Listing 3: Simulation Network Setup

1.3 Routing protocol Implementation

To implement the proposed protocol, a new **NS3** component, **LraRoutingProtocol**, has been developed. This component extends the **Ipv4RoutingProtocol** class provided by the NS3 framework.

The routing table is maintained using a **map data structure**, which establishes a correlation between an **IPv4 address** and its corresponding **LRA Link Status**.

Upon attachment of the routing protocol to a node, the node initiates a **broadcast service packet** known as the **Hello Message**. It subsequently awaits responses from neighboring nodes, which reply with a **Hello Response Message**. This exchange enables the node to **discover its neighbors** and construct its **routing table** accordingly.

When a **non-service packet** is received, the node determines whether it is the intended destination. If the node is the **destination**, the protocol terminates successfully. If the node is **not** the destination or if it is **creating and forwarding a new packet** towards the sink, it searches for the **next hop** within its routing table.

- If a **valid** next **hop** is found, the packet is forwarded to that hop along with an **Ack Request Message**, and a **timer** is started. If no **Ack Response Message** is received before the timer **expires**, the link toward that hop is marked as **"Ingoing"**.
- If **no** valid next **hop** is found, the **Link Reversal** mechanism is triggered, and the search process is repeated. If, after multiple attempts, no next hop is identified, the node **disables itself**, as it is determined to be part of a subgraph that is **disconnected from the sink**.

When a **service packet** is received, the node processes it based on the **packet type**, as follows:

- **Hello Message**: The receiving node updates its routing table by adding the sender with a **link type of "Outgoing"** if the sender's **IP address** is **greater** than that of the recipient; otherwise, the link is classified as **"Ingoing"**.
- **Hello Response Message**: The behavior is identical to that of the **Hello Message**, where the link type is determined based on the comparison of IP addresses.
- **Ack Request Message**: The receiving node responds by transmitting an **Ack Response Message**.
- **Ack Response Message**: Upon reception, the node halts the execution of the **DisableLinkTimer** associated with the sender.
- **Link Reversal Message**: The receiving node updates its routing table by designating the **link towards the sender** as **"Ingoing"**. Additionally, it evaluates whether a **Link Reversal** is necessary.

```
void LraRoutingProtocol::LinkReversal()
{
    // Base case
    if (m_nodeAddress == m_sink)
    {
        return;
    }

    // Actual inversion
    for (auto neighbor : m_neighbors)
    {
        m_linkStatus[neighbor] = 1; //Outgoing
    }
}
```

Listing 4: Link Reversal Method

```
Ipv4Address LraRoutingProtocol::GetNextHop(){
    auto nextHop = GetFirstOutgoingLink();
    if(nextHop != m_broadcastAddress){
        return nextHop;
    }
    else{
        if(nextHop == m_broadcastAddress && m_neighbors.size() > 0){
            LinkReversal();
            auto nextHop = GetFirstOutgoingLink();
            // Notify all nodes that you are now in active state.
            SendReversalMessage(m_broadcastAddress);
            return nextHop;
        }
    }

    return m_broadcastAddress; // fallback address
}
```

Listing 5: Next Hop Find Method

```cpp
void LraRoutingProtocol::RecvLraServicePacket(Ptr<Packet> packet, Ipv4Address origin)
{
    std::string payload = GetPacketPayload(packet);

    // Ack Request Message received
    if (payload == LraRoutingProtocol::LRA_ACK_SEND_MESSAGE)
    {
        if (m_linkStatus[origin] == 1) // Cycle detected
        {
            DisableLinkTo(origin);
            return;
        }
        SendAckResponseMessage(origin);
    }
    // Ack Response received
    else if (payload == LraRoutingProtocol::LRA_ACK_RECV_MESSAGE)
    {
        m_disableLinkToEvent[origin].Cancel(); // link active, avoid disabling it
        m_disableLinkToEvent.erase(origin);
    }
    // Hello Message received
    else if (payload == LraRoutingProtocol::LRA_HELLO_SEND_MESSAGE)
    {
        if (m_nodeAddress < origin)
            EnableLinkTo(origin);
        else
            DisableLinkTo(origin, true);

        SendHelloResponseMessage(origin);
    }
    // Hello Message Response received
    else if (payload == LraRoutingProtocol::LRA_HELLO_RECV_MESSAGE)
    {
        if (m_nodeAddress < origin)
            EnableLinkTo(origin);
        else
            DisableLinkTo(origin, true);
    }
    // Link Reversal Message received
    else if (payload == LraRoutingProtocol::LRA_REVERSAL_SEND_MESSAGE)
    {
        DisableLinkTo(origin);
    }
}
```

Listing 6: Service Packet Received Handling Method