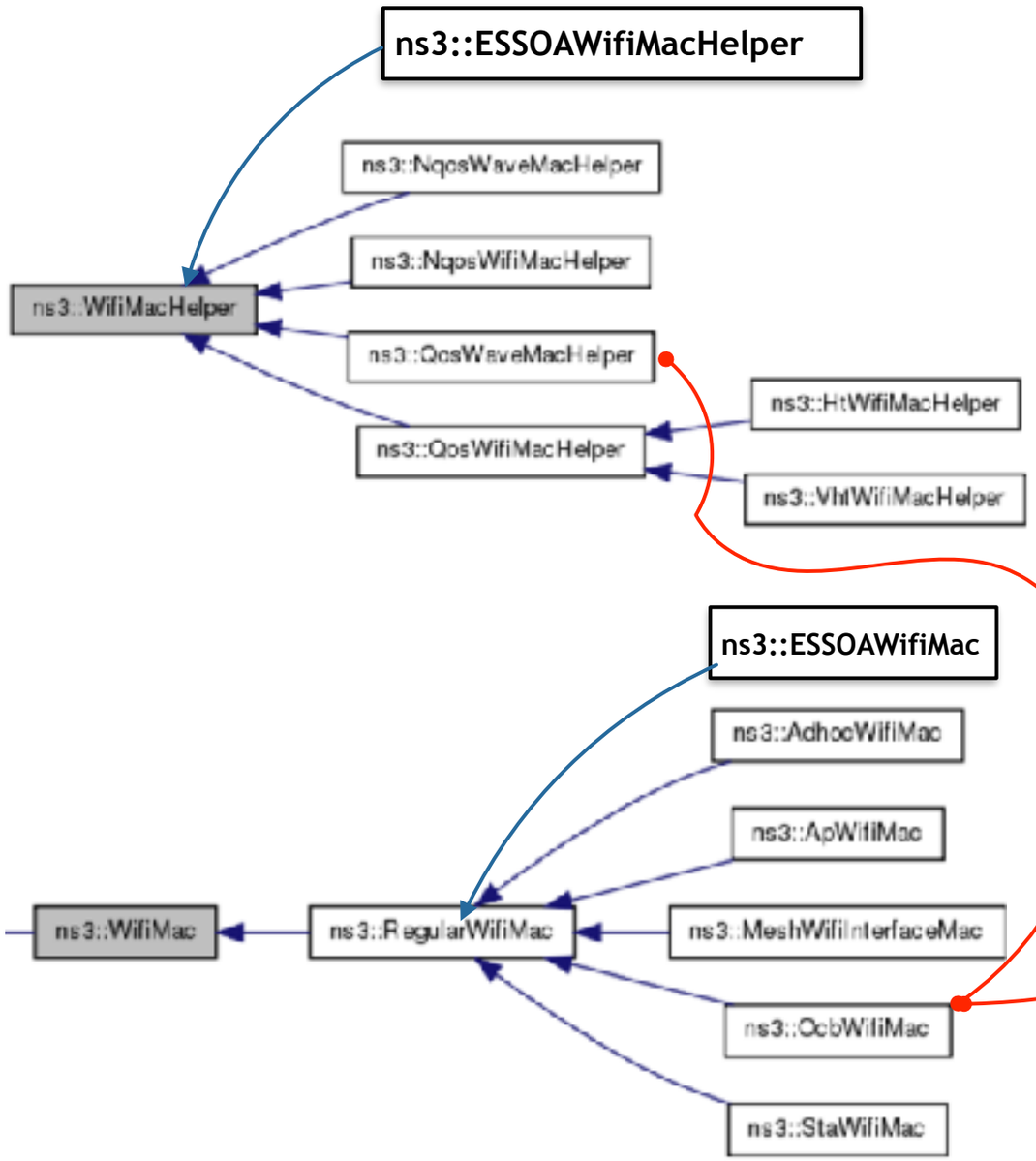
New MAC model on Wifi

M.S. Francisco Eduardo Balart Sanchez




```
QosWaveMacHelper waveMac = QosWaveMacHelper::Default ();
```

```
devices = waveHelper.Install (wavePhy, waveMac, nodes);
```

```
QosWaveMacHelper
QosWaveMacHelper::Default (void)
{
  QosWaveMacHelper helper;

  // We're making QoS-enabled Wi-Fi MACs here, so we set the necessary
  // attribute. I've carefully positioned this here so that someone
  // who knows what they're doing can override with explicit
  // attributes.
helper.SetType ("ns3::OcbWifiMac", "QosSupported", BooleanValue (true));
```

WifiMAC

Go to:

https://github.com/balart40/big_data_iteso_phd_public/wiki/NS3_with_MANET#creating-a-new-wifi-mac-model

AODV

AODV

```
void
AodvExample::Run ()
{
//  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThresho
  CreateNodes ();
  CreateDevices ();
  InstallInternetStack ();
  InstallApplications ();

  std::cout << "Starting simulation for " << totalTime << " s ...\n";

  Simulator::Stop (Seconds (totalTime));
  Simulator::Run ();
  Simulator::Destroy ();
}
```

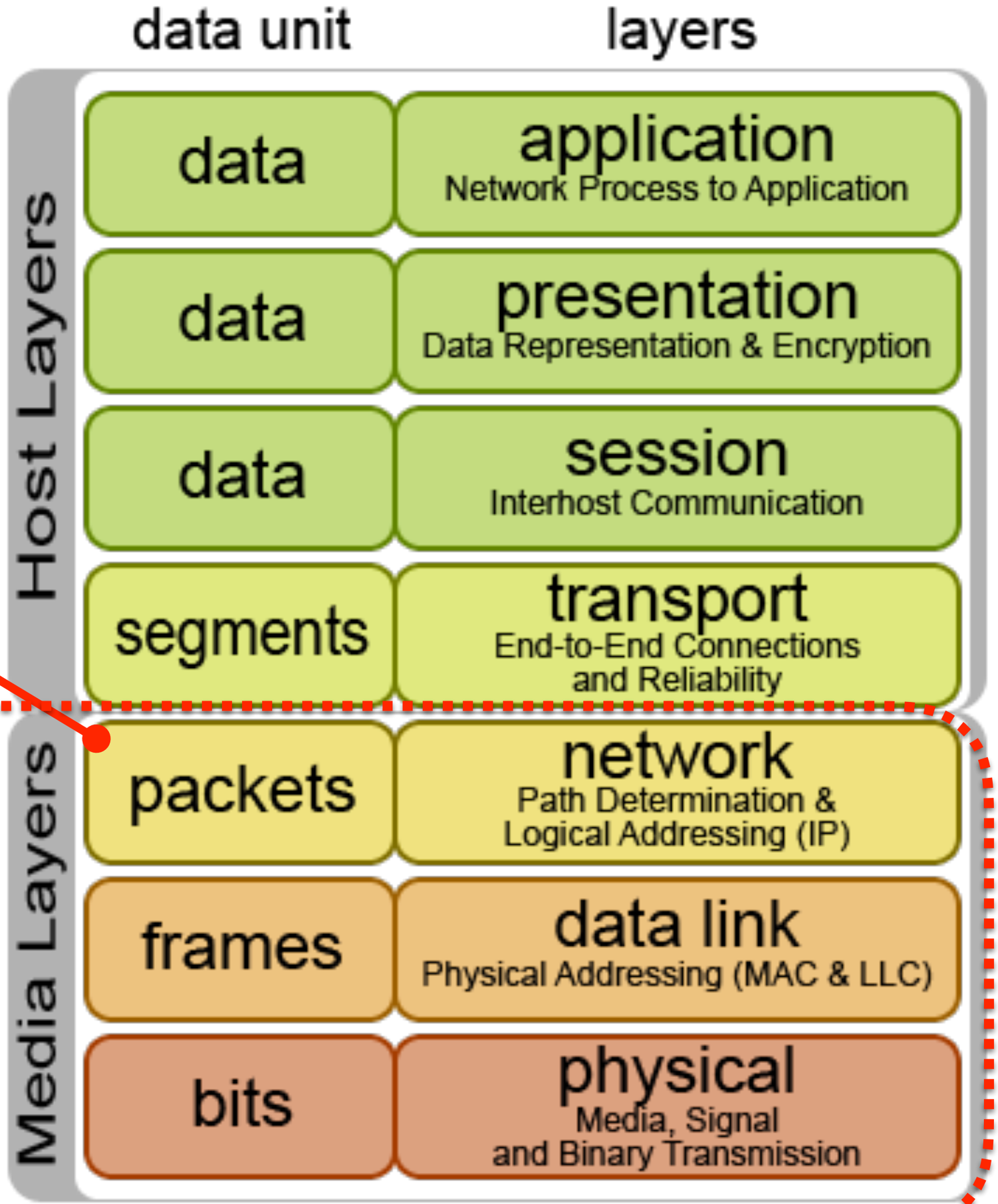
WifiNetDevices

```
void
AodvExample::CreateNodes ()
{
  std::cout << "Creating " << (unsigned)size << " nodes " << step << " m apart.\n";
  nodes.Create (size);
  // Name nodes
  for (uint32_t i = 0; i < size; ++i)
    {
      std::ostringstream os;
      os << "node-" << i;
      Names::Add (os.str (), nodes.Get (i));
    }
  // Create static grid
  MobilityHelper mobility;
  mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                 "MinX", DoubleValue (0.0),
                                 "MinY", DoubleValue (0.0),
                                 "DeltaX", DoubleValue (step),
                                 "DeltaY", DoubleValue (0),
                                 "GridWidth", UintegerValue (size),
                                 "LayoutType", StringValue ("RowFirst"));
  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (nodes);
}
```

```
void
AodvExample::CreateDevices ()
{
  WifiMacHelper wifiMac;
  wifiMac.SetType ("ns3::AdhocWifiMac");
  YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
  YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default ();
  wifiPhy.SetChannel (wifiChannel.Create ());
  WifiHelper wifi;
  wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode", StringValue ("OfdmRate6Mbps"), "RtsCtsThreshold", UintegerValue (0));
  devices = wifi.Install (wifiPhy, wifiMac, nodes);

  if (pcap)
    {
      wifiPhy.EnablePcapAll (std::string ("aodv"));
    }
}
```

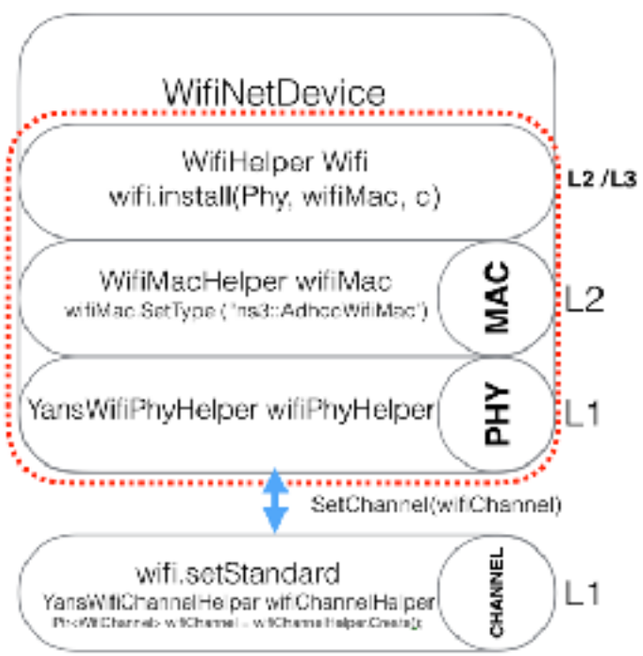
```
void
AodvExample::InstallInternetStack ()
{
  AodvHelper aodv;
  // you can configure AODV attributes here using aodv.Set(name, value)
  InternetStackHelper stack;
  stack.SetRoutingHelper (aodv); // has effect on the next Install ()
  stack.Install (nodes);
  Ipv4AddressHelper address;
  address.SetBase ("10.0.0.0", "255.0.0.0");
  interfaces = address.Assign (devices);

  if (printRoutes)
    {
      Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("aodv.routes", std::ios::out);
      aodv.PrintRoutingTableAllAt (Seconds (8), routingStream);
    }
}
```

OSI



Routing

InternetStack



https://gende.sscc.uchile.cl/w/images/2/2b/Osi-model.png

AODV routing aodv-routing-protocol.h

```
namespace ns3
{
namespace aodv
{
/**
 * \ingroup aodv
 *
 * \brief AODV routing protocol
 */
class RoutingProtocol : public Ipv4RoutingProtocol
private:
  Neighbors m_nb;
```

AODV routing

aodv-routing-protocol.cc

```
RoutingProtocol::RoutingProtocol () :
  m_nb (HelloInterval),
  {
    m_nb.SetCallback (MakeCallback (&RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this));
  }

void
RoutingProtocol::Start ()
  if (EnableHello)
    {
      m_nb.ScheduleTimer ();
    }

bool
RoutingProtocol::RouteInput (Ptr<const Packet> p, const Ipv4Header &header,
                             Ptr<const NetDevice> idev, UnicastForwardCallback ucb,
                             MulticastForwardCallback mcb, LocalDeliverCallback lcb, ErrorCallback ecb)
    // Unicast local delivery
    if (m_ipv4->IsDestinationAddress (dst, iif))
      {
        UpdateRouteLifeTime (origin, ActiveRouteTimeout);
        RoutingTableEntry toOrigin;
        if (m_routingTable.LookupValidRoute (origin, toOrigin))
          {
            UpdateRouteLifeTime (toOrigin.GetNextHop (), ActiveRouteTimeout);
            m_nb.Update (toOrigin.GetNextHop (), ActiveRouteTimeout);
          }

 bool
 RoutingProtocol::Forwarding (Ptr<const Packet> p, const Ipv4Header & header,
                              UnicastForwardCallback ucb, ErrorCallback ecb)
       m_nb.Update (route->GetGateway (), ActiveRouteTimeout);
       m_nb.Update (toOrigin.GetNextHop (), ActiveRouteTimeout);

void
RoutingProtocol::NotifyInterfaceUp (uint32_t i)
       if (l3->GetInterface (i)->GetArpCache ())
         {
           m_nb.AddArpCache (l3->GetInterface (i)->GetArpCache ());
         ]
        mac->TraceConnectWithoutContext ("TxErrHeader", m_nb.GetTxErrorCallback ());
```

AODV routing

aodv-routing-protocol.cc

```
void
RoutingProtocol::NotifyInterfaceDown (uint32_t i)
    if (wifi != 0)
      {
        Ptr<WifiMac> mac = wifi->GetMac ()->GetObject<AdhocWifiMac> ();
        if (mac != 0)
          {
            mac->TraceDisconnectWithoutContext ("TxErrHeader",
                                                m_nb.GetTxErrorCallback ());
            m_nb.DelArpCache (l3->GetInterface (i)->GetArpCache ());
          }
      }
     if (m_socketAddresses.empty ())
       {
         NS_LOG_LOGIC ("No aodv interfaces");
         m_htimer.Cancel ();
         m_nb.Clear ();
         m_routingTable.Clear ();
         return;
       }
```

```
 void
 RoutingProtocol::NotifyRemoveAddress (uint32_t i, Ipv4InterfaceAddress address)
     if (m_socketAddresses.empty ())
       {
         NS_LOG_LOGIC ("No aodv interfaces");
         m_htimer.Cancel ();
         m_nb.Clear ();
         m_routingTable.Clear ();
         return;
       }
 void
 RoutingProtocol::RecvRequest (Ptr<Packet> p, Ipv4Address receiver, Ipv4Address src)
 {
   m_nb.Update (src, Time (AllowedHelloLoss * HelloInterval));

void
RoutingProtocol::ProcessHello (RrepHeader const & rrepHeader, Ipv4Address receiver )
{
     if (EnableHello)
       {
         m_nb.Update (rrepHeader.GetDst (), Time (AllowedHelloLoss * HelloInterval));
       }
```

aodv-neighbor class

```
balart40@balart40-VirtualBox:~/Desktop/balart40/ns/ns-allinone-3.25/ns-3.25/src/aodv$ ls
bindings  doc  examples  helper  model  test  wscript
balart40@balart40-VirtualBox:~/Desktop/balart40/ns/ns-allinone-3.25/ns-3.25/src/aodv$ cd model/
balart40@balart40-VirtualBox:~/Desktop/balart40/ns/ns-allinone-3.25/ns-3.25/src/aodv/model$ ls
aodv-dpd.cc  aodv-id-cache.cc  aodv-neighbor.cc  aodv-packet.cc  aodv-routing-protocol.cc  aodv-rqueue.cc  aodv-rtable.cc
aodv-dpd.h   aodv-id-cache.h   aodv-neighbor.h   aodv-packet.h   aodv-routing-protocol.h   aodv-rqueue.h   aodv-rtable.h
balart40@balart40-VirtualBox:~/Desktop/balart40/ns/ns-allinone-3.25/ns-3.25/src/aodv/model$
```

*.h

```
#ifndef AODVNEIGHBOR_H
#define AODVNEIGHBOR_H

#include "ns3/simulator.h"
#include "ns3/timer.h"
#include "ns3/ipv4-address.h"
#include "ns3/callback.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/arp-cache.h"
#include <vector>

namespace ns3
{
namespace aodv
{
class RoutingProtocol;
/**
 * \ingroup aodv
 * \brief maintain list of active neighbors
 */
class Neighbors
{
public:
  /// c-tor
  Neighbors (Time delay);
  /// Neighbor description
  struct Neighbor
  {
    Ipv4Address m_neighborAddress;
    Mac48Address m_hardwareAddress;
    Time m_expireTime;
    bool close;

    Neighbor (Ipv4Address ip, Mac48Address mac, Time t) :
      m_neighborAddress (ip), m_hardwareAddress (mac), m_expireTime (t),
      close (false)
    {
    }
  };
  /// Return expire time for neighbor node with address addr, if exists, else return 0.
  Time GetExpireTime (Ipv4Address addr);
  /// Check that node with address addr  is neighbor
  bool IsNeighbor (Ipv4Address addr);
  /// Update expire time for entry with address addr, if it exists, else add new entry
  void Update (Ipv4Address addr, Time expire);
  /// Remove all expired entries
  void Purge ();
  /// Schedule m_ntimer.
  void ScheduleTimer ();
  /// Remove all entries
  void Clear () { m_nb.clear (); }

  /// Add ARP cache to be used to allow layer 2 notifications processing
  void AddArpCache (Ptr<ArpCache>);
  /// Don't use given ARP cache any more (interface is down)
  void DelArpCache (Ptr<ArpCache>);
  /// Get callback to ProcessTxError
  Callback<void, WifiMacHeader const &> GetTxErrorCallback () const { return m_txErrorCallback; }

  /// Handle link failure callback
  void SetCallback (Callback<void, Ipv4Address> cb) { m_handleLinkFailure = cb; }
  /// Handle link failure callback
  Callback<void, Ipv4Address> GetCallback () const { return m_handleLinkFailure; }

private:
  /// link failure callback
  Callback<void, Ipv4Address> m_handleLinkFailure;
  /// TX error callback
  Callback<void, WifiMacHeader const &> m_txErrorCallback;
  /// Timer for neighbor's list. Schedule Purge().
  Timer m_ntimer;
  /// vector of entries
  std::vector<Neighbor> m_nb;
  /// list of ARP cached to be used for layer 2 notifications processing
  std::vector<Ptr<ArpCache> > m_arp;

  /// Find MAC address by IP using list of ARP caches
  Mac48Address LookupMacAddress (Ipv4Address);
  /// Process layer 2 TX error notification
  void ProcessTxError (WifiMacHeader const &);
};

}
}

#endif /* AODVNEIGHBOR_H */
```

aodv-neighbor class

```
#include "aodv-neighbor.h"
#include "ns3/log.h"
#include <algorithm>

namespace ns3
{
   NS_LOG_COMPONENT_DEFINE ("AodvNeighbors");
namespace aodv
{
Neighbors::Neighbors (Time delay) :
  m_ntimer (Timer::CANCEL_ON_DESTROY)
{
  m_ntimer.SetDelay (delay);
  m_ntimer.SetFunction (&Neighbors::Purge, this);
  m_txErrorCallback = MakeCallback (&Neighbors::ProcessTxError, this);
}
bool Neighbors::IsNeighbor (Ipv4Address addr)
{
  Purge ();
  for (std::vector<Neighbor>::const_iterator i = m_nb.begin ();
       i != m_nb.end (); ++i)
    {
      if (i->m_neighborAddress == addr)
        return true;
    }
  return false;
}
Time Neighbors::GetExpireTime (Ipv4Address addr)
{
  Purge ();
  for (std::vector<Neighbor>::const_iterator i = m_nb.begin (); i
       != m_nb.end (); ++i)
    {
      if (i->m_neighborAddress == addr)
        return (i->m_expireTime - Simulator::Now ());
    }
  return Seconds (0);
}
void Neighbors::Update (Ipv4Address addr, Time expire)
{
  for (std::vector<Neighbor>::iterator i = m_nb.begin (); i != m_nb.end (); ++i)
    if (i->m_neighborAddress == addr)
      {
        i->m_expireTime
          = std::max (expire + Simulator::Now (), i->m_expireTime);
        if (i->m_hardwareAddress == Mac48Address ())
          i->m_hardwareAddress = LookupMacAddress (i->m_neighborAddress);
        return;
      }
  NS_LOG_LOGIC ("Open link to " << addr);
  Neighbor neighbor (addr, LookupMacAddress (addr), expire + Simulator::Now ());
  m_nb.push_back (neighbor);
  Purge ();
}
struct CloseNeighbor
{
  bool operator() (const Neighbors::Neighbor & nb) const
  {
    return ((nb.m_expireTime < Simulator::Now ()) || nb.close);
  }
};

void Neighbors::Purge ()
{
  if (m_nb.empty ())
    return;

  CloseNeighbor pred;
  if (!m_handleLinkFailure.IsNull ())
    {
      for (std::vector<Neighbor>::iterator j = m_nb.begin (); j != m_nb.end (); ++j)
        {
          if (pred (*j))
            {
              NS_LOG_LOGIC ("Close link to " << j->m_neighborAddress);
              m_handleLinkFailure (j->m_neighborAddress);
            }
        }
    }
  m_nb.erase (std::remove_if (m_nb.begin (), m_nb.end (), pred), m_nb.end ());
  m_ntimer.Cancel ();
  m_ntimer.Schedule ();
}
void Neighbors::ScheduleTimer ()
{
  m_ntimer.Cancel ();
  m_ntimer.Schedule ();
}
void Neighbors::AddArpCache (Ptr<ArpCache> a)
{
  m_arp.push_back (a);
}
void Neighbors::DelArpCache (Ptr<ArpCache> a)
{
  m_arp.erase (std::remove (m_arp.begin (), m_arp.end (), a), m_arp.end ());
}
Mac48Address Neighbors::LookupMacAddress (Ipv4Address addr)
{
  Mac48Address hwaddr;
  for (std::vector<Ptr<ArpCache> >::const_iterator i = m_arp.begin ();
       i != m_arp.end (); ++i)
    {
      ArpCache::Entry * entry = (*i)->Lookup (addr);
      if (entry != 0 && (entry->IsAlive () || entry->IsPermanent ()) && !entry->IsExpired ())
        {
          hwaddr = Mac48Address::ConvertFrom (entry->GetMacAddress ());
          break;
        }
    }
  return hwaddr;
}
void Neighbors::ProcessTxError (WifiMacHeader const & hdr)
{
  Mac48Address addr = hdr.GetAddr1 ();

  for (std::vector<Neighbor>::iterator i = m_nb.begin (); i != m_nb.end (); ++i)
    {
      if (i->m_hardwareAddress == addr)
        i->close = true;
    }
  Purge ();
}}}
```