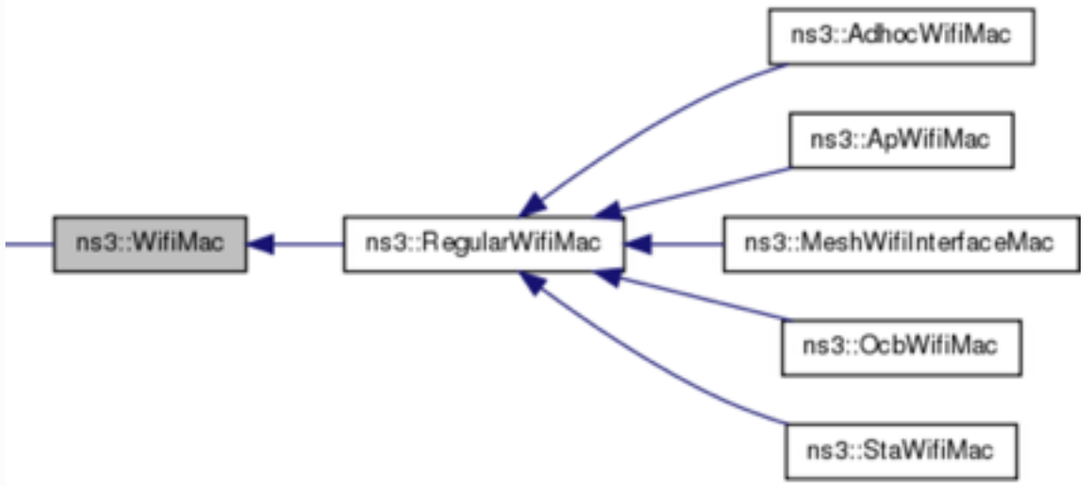
WIFI NET DEVICE

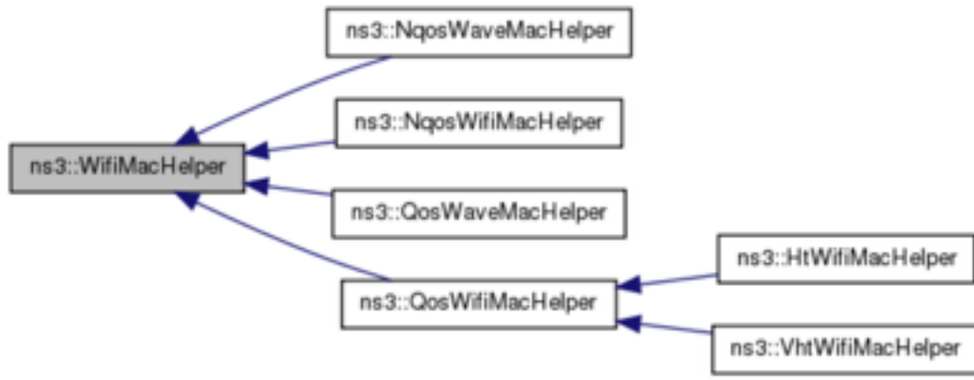
```
class WifiRemoteStationManager;
class WifiChannel;
class WifiPhy;
class WifiMac;

class WifiNetDevice : public NetDevice
{
    Ptr<Node> m_node;
    Ptr<WifiPhy> m_phy;
    Ptr<WifiMac> m_mac;
    Ptr<WifiRemoteStationManager> m_stationManager;
```

```
WifiNetDevice::Send (Ptr<Packet> packet, const Address& dest, uint16_t protocolNumber)
{
  NS_LOG_FUNCTION (this << packet << dest << protocolNumber);
  NS_ASSERT (Mac48Address::IsMatchingType (dest));

  Mac48Address realTo = Mac48Address::ConvertFrom (dest);

  LlcSnapHeader llc;
  llc.SetType (protocolNumber);
  packet->AddHeader (llc);

  m_mac->NotifyTx (packet);
  m_mac->Enqueue (packet, realTo);
  return true;
}
```

```
Ptr<WifiMac> m_mac;
```

```
void
WifiMac::NotifyTx (Ptr<const Packet> packet)
{
  m_macTxTrace (packet);
}
```






```
namespace ns3 {

class WifiMac;

class WifiMacHelper
{
public:
  WifiMacHelper ();
  virtual ~WifiMacHelper ();

  virtual void SetType (std::string type,
                        std::string n0 = "", const AttributeValue &v0 = EmptyAttributeValue (),
                        std::string n1 = "", const AttributeValue &v1 = EmptyAttributeValue (),
                        std::string n2 = "", const AttributeValue &v2 = EmptyAttributeValue (),
                        std::string n3 = "", const AttributeValue &v3 = EmptyAttributeValue (),
                        std::string n4 = "", const AttributeValue &v4 = EmptyAttributeValue (),
                        std::string n5 = "", const AttributeValue &v5 = EmptyAttributeValue (),
                        std::string n6 = "", const AttributeValue &v6 = EmptyAttributeValue (),
                        std::string n7 = "", const AttributeValue &v7 = EmptyAttributeValue (),
                        std::string n8 = "", const AttributeValue &v8 = EmptyAttributeValue (),
                        std::string n9 = "", const AttributeValue &v9 = EmptyAttributeValue (),
                        std::string n10 = "", const AttributeValue &v10 = EmptyAttributeValue (

  virtual Ptr<WifiMac> Create (void) const;

protected:
  ObjectFactory m_mac;
};
```

```
WifiMacHelper::~WifiMacHelper ()
{
}

void
WifiMacHelper::SetType (std::string type,
                        std::string n0, const AttributeValue &v0,
                        std::string n1, const AttributeValue &v1,
                        std::string n2, const AttributeValue &v2,
                        std::string n3, const AttributeValue &v3,
                        std::string n4, const AttributeValue &v4,
                        std::string n5, const AttributeValue &v5,
                        std::string n6, const AttributeValue &v6,
                        std::string n7, const AttributeValue &v7,
                        std::string n8, const AttributeValue &v8,
                        std::string n9, const AttributeValue &v9,
                        std::string n10, const AttributeValue &v10)
{
  m_mac.SetTypeId (type);
  m_mac.Set (n0, v0);
  m_mac.Set (n1, v1);
  m_mac.Set (n2, v2);
  m_mac.Set (n3, v3);
  m_mac.Set (n4, v4);
  m_mac.Set (n5, v5);
  m_mac.Set (n6, v6);
  m_mac.Set (n7, v7);
  m_mac.Set (n8, v8);
  m_mac.Set (n9, v9);
  m_mac.Set (n10, v10);
}
```

```
Ptr<WifiMac>
WifiMacHelper::Create (void) const
{
  Ptr<WifiMac> mac = m_mac.Create<WifiMac> ();
  return mac;
}
```

```
// MAC layer configuration
WifiMacHelper wifiMac;
wifiMac.SetType ("ns3::AdhocWifiMac","QosSupported", BooleanValue (false));
```

```
void
WifiMac::NotifyTx (Ptr<const Packet> packet)
{
  m_macTxTrace (packet);
}
```

```
TracedCallback<Ptr<const Packet> > m_macTxTrace;
```

```
virtual void Enqueue (Ptr<const Packet> packet, Mac48Address to, Mac48Address from) = 0;
```

```
virtual void Enqueue (Ptr<const Packet> packet, Mac48Address to) = 0;
```

```
virtual void Enqueue (Ptr<const Packet> packet, Mac48Address to) = 0;
```

```
void
AdhocWifiMac::Enqueue (Ptr<const Packet> packet, Mac48Address to)
{
  NS_LOG_FUNCTION (this << packet << to);
  if (m_stationManager->IsBrandNew (to))
    {
      //In ad hoc mode, we assume that every destination supports all
      //the rates we support.
      if (m_htSupported || m_vhtSupported)
        {
          m_stationManager->AddAllSupportedMcs (to);
          m_stationManager->AddStationHtCapabilities (to, GetHtCapabilities());
        }
      if (m_vhtSupported)
        {
          m_stationManager->AddStationVhtCapabilities (to, GetVhtCapabilities());
        }
      m_stationManager->AddAllSupportedModes (to);
      m_stationManager->RecordDisassociated (to);
    }

  WifiMacHeader hdr;

  //If we are not a QoS STA then we definitely want to use AC_BE to
  //transmit the packet. A TID of zero will map to AC_BE (through \c
  //QosUtilsMapTidToAc()), so we use that as our default here.
  uint8_t tid = 0;

  //For now, a STA that supports QoS does not support non-QoS
  //associations, and vice versa. In future the STA model should fall
  //back to non-QoS if talking to a peer that is also non-QoS. At
  //that point there will need to be per-station QoS state maintained
  //by the association state machine, and consulted here.
  if (m_qosSupported)
    {
      hdr.SetType (WIFI_MAC_QOSDATA);
      hdr.SetQosAckPolicy (WifiMacHeader::NORMAL_ACK);
      hdr.SetQosNoEosp ();
```

```
hdr.SetQosNoAmsdu ();
//Transmission of multiple frames in the same TXOP is not
//supported for now
hdr.SetQosTxopLimit (0);

//Fill in the QoS control field in the MAC header
tid = QosUtilsGetTidForPacket (packet);
//Any value greater than 7 is invalid and likely indicates that
//the packet had no QoS tag, so we revert to zero, which will
//mean that AC_BE is used.
if (tid > 7)
  {
    tid = 0;
  }
hdr.SetQosTid (tid);
  }
else
  {
    hdr.SetTypeData ();
  }

if (m_htSupported || m_vhtSupported)
  {
    hdr.SetNoOrder ();
  }
hdr.SetAddr1 (to);
hdr.SetAddr2 (m_low->GetAddress ());
hdr.SetAddr3 (GetBssid ());
hdr.SetDsNotFrom ();
hdr.SetDsNotTo ();
```

```
  if (m_htSupported || m_vhtSupported)
    {
      hdr.SetNoOrder ();
    }
  hdr.SetAddr1 (to);
  hdr.SetAddr2 (m_low->GetAddress ());
  hdr.SetAddr3 (GetBssid ());
  hdr.SetDsNotFrom ();
  hdr.SetDsNotTo ();

  if (m_qosSupported)
    {
      //Sanity check that the TID is valid
      NS_ASSERT (tid < 8);
      m_edca[QosUtilsMapTidToAc (tid)]->Queue (packet, hdr);
    }
  else
    {
      m_dca->Queue (packet, hdr);
    }
}
```

```
/** This holds a pointer to the DCF instance for this WifiMac - used
for transmission of frames to non-QoS peers. */
Ptr<DcaTxop> m_dca;
```

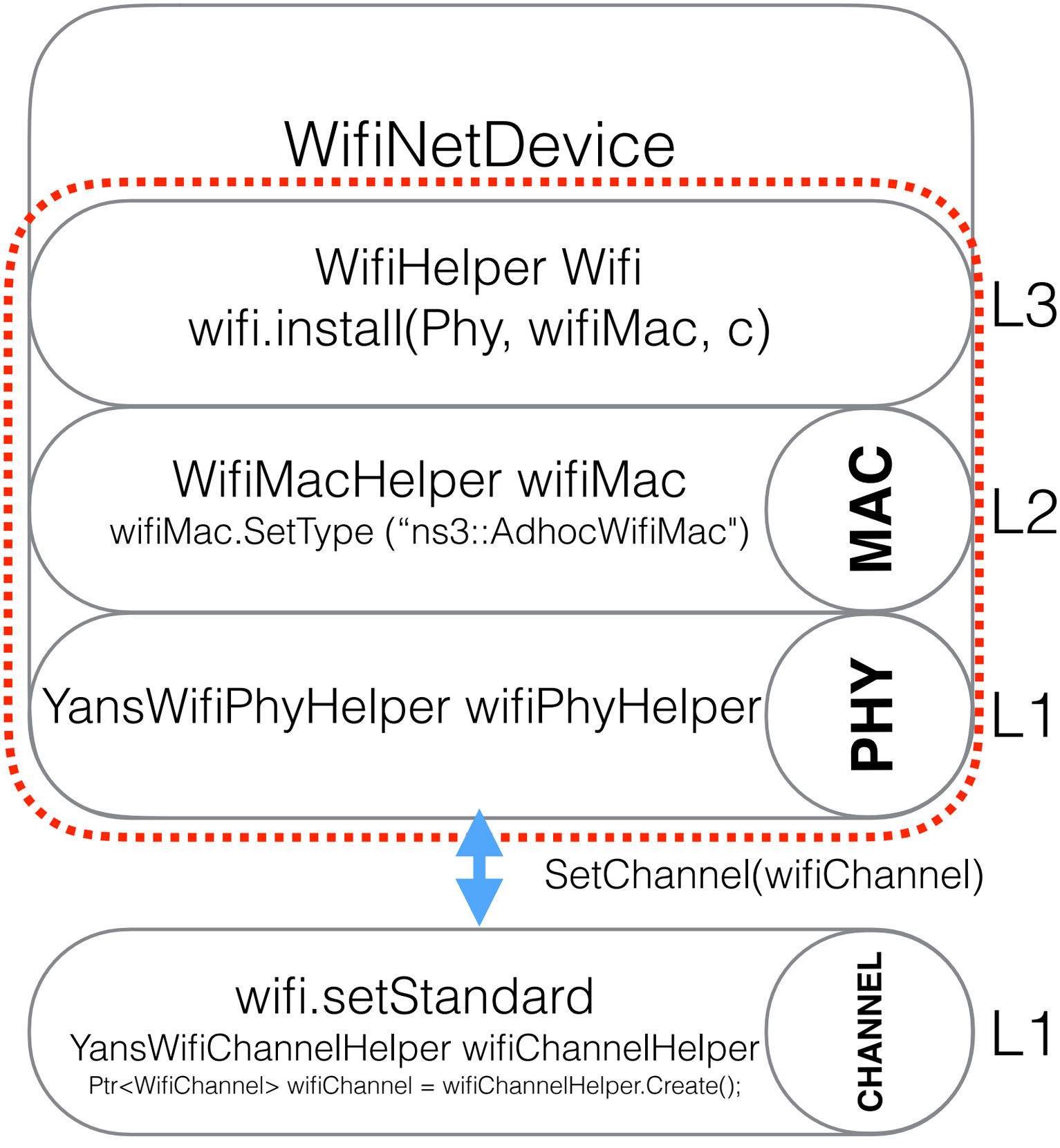
```
void
DcaTxop::Queue (Ptr<const Packet> packet, const WifiMacHeader &hdr)
{
  NS_LOG_FUNCTION (this << packet << &hdr);
  WifiMacTrailer fcs;
  m_stationManager->PrepareForQueue (hdr.GetAddr1 (), &hdr, packet);
  m_queue->Enqueue (packet, hdr);
  StartAccessIfNeeded ();
}
```

```
Ptr<WifiMacQueue> m_queue;
```

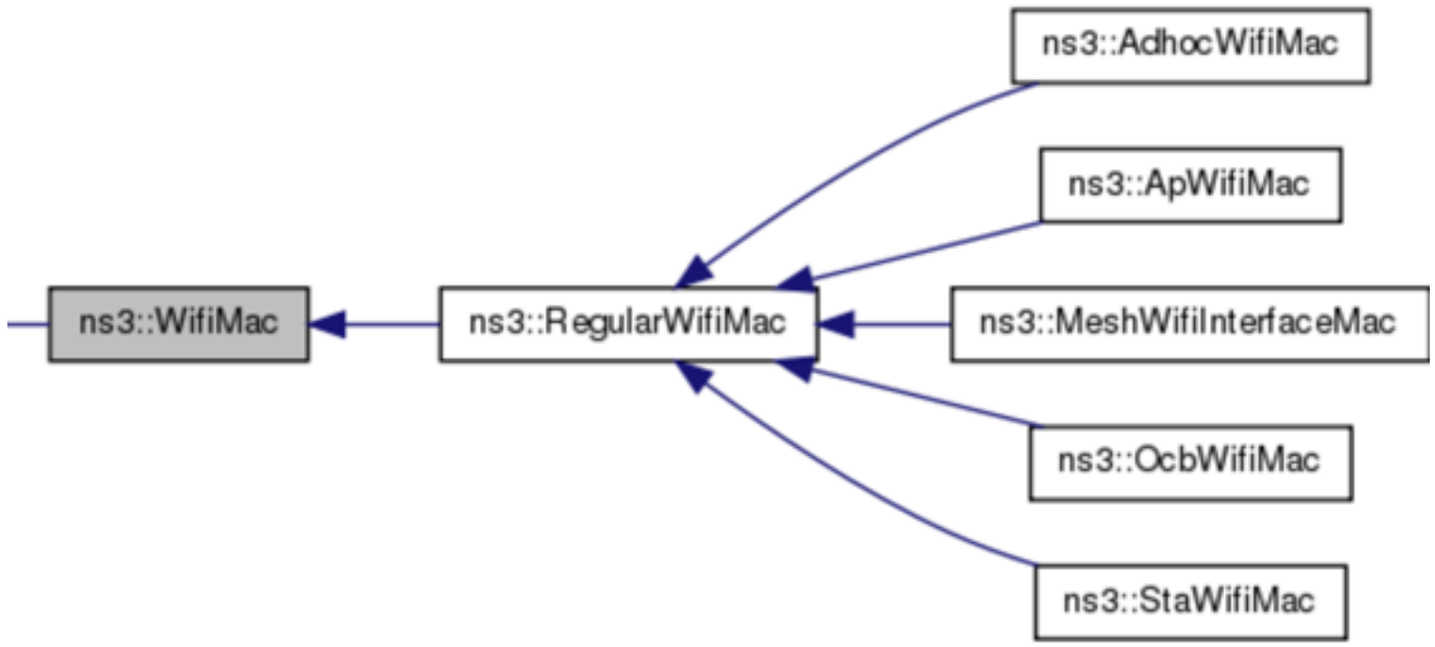
```
void
WifiMacQueue::Enqueue (Ptr<const Packet> packet, const WifiMacHeader &hdr)
{
  Cleanup ();
  if (m_size == m_maxSize)
    {
      return;
    }
  Time now = Simulator::Now ();
  m_queue.push_back (Item (packet, hdr, now));
  m_size++;
}
```

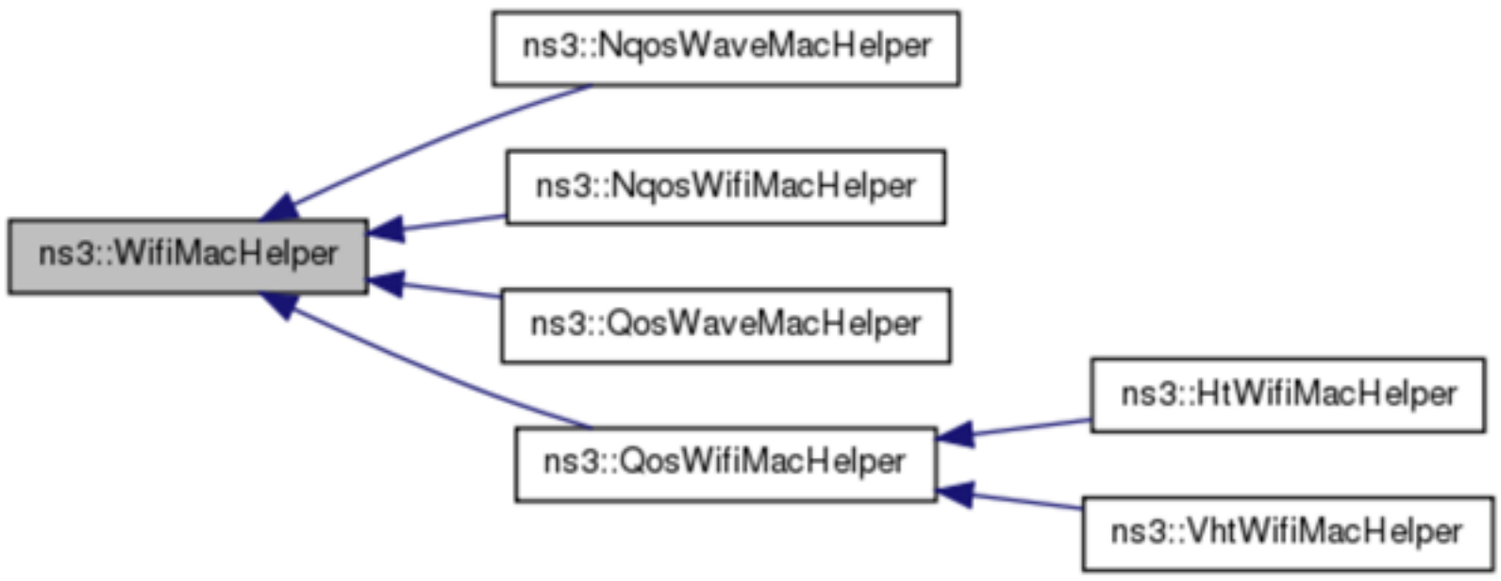
```
/**
 * typedef for packet (struct Item) queue.
 */
typedef std::list<struct Item> PacketQueue;
/**
```

```
PacketQueue m_queue;
```


WifiNetDevice
WifiHelper Wifi
wifi.install(Phy, wifiMac, c)
L3
WifiMacHelper wifiMac
wifiMac.SetType (“ns3::AdhocWifiMac")
MAC
L2
YansWifiPhyHelper wifiPhyHelper
PHY
L1
SetChannel(wifiChannel)
wifi.setStandard
YansWifiChannelHelper wifiChannelHelper
Ptr<WifiChannel> wifiChannel = wifiChannelHelper.Create();
CHANNEL
L1

MAC


ns3::AdhocWifiMac
ns3::ApWifiMac
ns3::WifiMac
ns3::RegularWifiMac
ns3::MeshWifiInterfaceMac
ns3::OcbWifiMac
ns3::StaWifiMac


ns3::NqosWaveMacHelper
ns3::NqosWifiMacHelper
ns3::WifiMacHelper
ns3::QosWaveMacHelper
ns3::HtWifiMacHelper
ns3::QosWifiMacHelper
ns3::VhtWifiMacHelper

[legend]

WIFI

```
NetDeviceContainer
WifiHelper::Install (const WifiPhyHelper &phyHelper,
                     const WifiMacHelper &macHelper, NodeContainer c) const

  NetDeviceContainer devices;
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
      Ptr<Node> node = *i;
      Ptr<WifiNetDevice> device = CreateObject<WifiNetDevice> ();
      Ptr<WifiRemoteStationManager> manager = m_stationManager.Create<WifiRemoteStationManager> ();
      Ptr<WifiMac> mac = macHelper.Create ();
      Ptr<WifiPhy> phy = phyHelper.Create (node, device);
      mac->SetAddress (Mac48Address::Allocate ());
      mac->ConfigureStandard (m_standard);
      phy->ConfigureStandard (m_standard);
      device->SetMac (mac);
      device->SetPhy (phy);
      device->SetRemoteStationManager (manager);
      node->AddDevice (device);
      devices.Add (device);
      NS_LOG_DEBUG ("node=" << node << ", mob=" << node->GetObject<MobilityModel> ());
    }
  return devices;
}
```

```
WifiMac::ConfigureStandard (enum WifiPhyStandard standard)
{
  switch (standard)
    {
    case WIFI_PHY_STANDARD_80211a:
      Configure80211a ();
      break;
    case WIFI_PHY_STANDARD_80211b:
      Configure80211b ();
      break;
    case WIFI_PHY_STANDARD_80211g:
      Configure80211g ();
      break;
    case WIFI_PHY_STANDARD_80211_10MHZ:
      Configure80211_10Mhz ();
      break;
    case WIFI_PHY_STANDARD_80211_5MHZ:
      Configure80211_5Mhz ();
      break;
    case WIFI_PHY_STANDARD_holland:
      Configure80211a ();
      break;
    case WIFI_PHY_STANDARD_80211n_2_4GHZ:
      Configure80211n_2_4Ghz ();
      break;
    case WIFI_PHY_STANDARD_80211n_5GHZ:
      Configure80211n_5Ghz ();
      break;
    case WIFI_PHY_STANDARD_80211ac:
      Configure80211ac ();
      break;
    default:
      NS_ASSERT (false);
      break;
    }
  FinishConfigureStandard (standard);
```

```
WifiHelper::WifiHelper ()
  : m_standard (WIFI_PHY_STANDARD_80211a)
{
  SetRemoteStationManager ("ns3::ArfWifiManager");
}
```

```
WifiMacHelper::WifiMacHelper ()
{
  //By default, we create an AdHoc MAC layer without QoS.
  SetType ("ns3::AdhocWifiMac",
           "QosSupported", BooleanValue (false));
}
```

```
Ptr<WifiMac>
WifiMacHelper::Create (void) const
{
  Ptr<WifiMac> mac = m_mac.Create<WifiMac> ();
  return mac;
}
```

AODV

```
#include "ns3/core-module.h"
```

```
#include "abort.h"
#include "assert.h"
#include "attribute-accessor-helper.h"
#include "attribute-construction-list.h"
#include "attribute-helper.h"
#include "attribute.h"
#include "boolean.h"
#include "breakpoint.h"
#include "build-profile.h"
#include "calendar-scheduler.h"
#include "callback.h"
#include "command-line.h"
#include "config.h"
#include "default-deleter.h"
#include "default-simulator-impl.h"
#include "deprecated.h"
#include "double.h"
#include "empty.h"
#include "enum.h"
#include "event-garbage-collector.h"
#include "event-id.h"
#include "event-impl.h"
#include "fatal-error.h"
#include "fatal-impl.h"
#include "global-value.h"
#include "hash-fnv.h"
#include "hash-function.h"
#include "hash-murmur3.h"
#include "hash.h"
#include "heap-scheduler.h"
#include "int-to-type.h"
#include "int64x64-128.h"
#include "int64x64-double.h"
#include "int64x64.h"
#include "integer.h"
#include "list-scheduler.h"
#include "log-macros-disabled.h"
#include "log-macros-enabled.h"
#include "log.h"
#include "make-event.h"
#include "map-scheduler.h"
#include "math.h"
#include "names.h"
#include "non-copyable.h"
#include "nstime.h"
#include "object-base.h"
#include "object-factory.h"
#include "object-map.h"
#include "object-ptr-container.h"
#include "object-vector.h"
#include "object.h"
#include "pointer.h"
#include "ptr.h"
#include "random-variable-stream-helper.h"
#include "random-variable-stream.h"
#include "realtime-simulator-impl.h"
#include "ref-count-base.h"
#include "rng-seed-manager.h"
#include "rng-stream.h"
#include "scheduler.h"
#include "simple-ref-count.h"
#include "simulation-singleton.h"
#include "simulator-impl.h"
#include "simulator.h"
#include "singleton.h"
#include "string.h"
#include "synchronizer.h"
#include "system-condition.h"
#include "system-mutex.h"
#include "system-path.h"
#include "system-thread.h"
#include "system-wall-clock-ms.h"
#include "test.h"
#include "timer-impl.h"
#include "timer.h"
#include "trace-source-accessor.h"
#include "traced-callback.h"
#include "traced-value.h"
#include "type-id.h"
#include "type-name.h"
#include "type-traits.h"
#include "uinteger.h"
#include "unix-fd-reader.h"
#include "unused.h"
#include "valgrind.h"
#include "vector.h"
#include "wall-clock-synchronizer.h"
#include "watchdog.h"
#endif
```

/src/aodv/helper/aodv-helper.cc

```cpp
#include "aodv-helper.h"
#include "ns3/aodv-routing-protocol.h"
#include "ns3/node-list.h"
#include "ns3/names.h"
#include "ns3/ptr.h"
#include "ns3/ipv4-list-routing.h"
namespace ns3
{

AodvHelper::AodvHelper() :
  Ipv4RoutingHelper ()
{
  m_agentFactory.SetTypeId ("ns3::aodv::RoutingProtocol");
}

AodvHelper*
AodvHelper::Copy (void) const
{
  return new AodvHelper (*this);
}

Ptr<Ipv4RoutingProtocol>
AodvHelper::Create (Ptr<Node> node) const
{
  Ptr<aodv::RoutingProtocol> agent = m_agentFactory.Create<aodv::RoutingProtocol> ();
  node->AggregateObject (agent);
  return agent;
}
```

```cpp
private:
  /** the factory to create AODV routing object */
  ObjectFactory m_agentFactory;
};
```

/src/core/model/object-factory.cc

```cpp
Ptr<Object>
ObjectFactory::Create (void) const
{
  NS_LOG_FUNCTION (this);
  Callback<ObjectBase *> cb = m_tid.GetConstructor ();
  ObjectBase *base = cb ();
  Object *derived = dynamic_cast<Object *> (base);
  NS_ASSERT (derived != 0);
  derived->SetTypeId (m_tid);
  derived->Construct (m_parameters);
  Ptr<Object> object = Ptr<Object> (derived, false);
  return object;
}
```

```cpp
#include "ns3/aodv-module.h"

// Module headers:
#include "aodv-dpd.h"
#include "aodv-helper.h"
#include "aodv-id-cache.h"
#include "aodv-neighbor.h"
#include "aodv-packet.h"
#include "aodv-routing-protocol.h"
#include "aodv-rqueue.h"
#include "aodv-rtable.h"
#endif
```

/src/aodv/mode/aodv-routing-protocol.h

```cpp
#include "aodv-rtable.h"
#include "aodv-rqueue.h"
#include "aodv-packet.h"
#include "aodv-neighbor.h"
#include "aodv-dpd.h"
#include "ns3/node.h"
#include "ns3/random-variable-stream.h"
#include "ns3/output-stream-wrapper.h"
#include "ns3/ipv4-routing-protocol.h"
#include "ns3/ipv4-interface.h"
#include "ns3/ipv4-l3-protocol.h"
#include <map>

namespace ns3
{
namespace aodv
{
/**
 * \ingroup aodv
 *
 * \brief AODV routing protocol
 */
class RoutingProtocol : public Ipv4RoutingProtocol
{
public:
  static TypeId GetTypeId (void);
  static const uint32_t AODV_PORT;

  /// c-tor
  RoutingProtocol ();
  virtual ~RoutingProtocol();
  virtual void DoDispose ();
```

/src/aodv/mode/aodv-routing-protocol.h

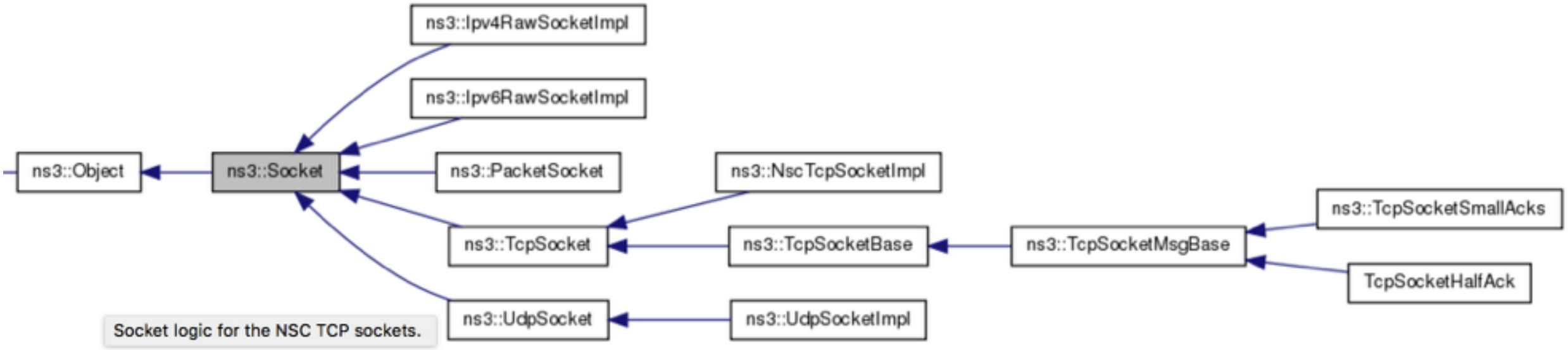
```
#include "aodv-rtable.h"
#include "aodv-rqueue.h"
#include "aodv-packet.h"
#include "aodv-neighbor.h"
#include "aodv-dpd.h"
#include "ns3/node.h"
#include "ns3/random-variable-stream.h"
#include "ns3/output-stream-wrapper.h"
#include "ns3/ipv4-routing-protocol.h"
#include "ns3/ipv4-interface.h"
#include "ns3/ipv4-l3-protocol.h"
#include <map>

namespace ns3
{
namespace aodv
{
/**
 * \ingroup aodv
 *
 * \brief AODV routing protocol
 */
class RoutingProtocol : public Ipv4RoutingProtocol
{
public:
  static TypeId GetTypeId (void);
  static const uint32_t AODV_PORT;

  /// c-tor
  RoutingProtocol ();
  virtual ~RoutingProtocol();
  virtual void DoDispose ();
```

/src/internet/mode/ipv4-raw-socket-impl.h

```
namespace ns3 {

class NetDevice;
class Node;

/**
 * \class Ipv4RawSocketImpl
 * \brief IPv4 raw socket.
 * \ingroup socket
 *
 * A RAW Socket typically is used to ac
 * available through L4 sockets, e.g., 
 * particular care to define the Ipv4Ra
 * particular the Protocol attribute.
 */
class Ipv4RawSocketImpl : public Socket
{

 virtual int SendTo (Ptr<Packet> p, uint32_t flags,
                     const Address &toAddress);
```

```
TypeId
Ipv4RawSocketImpl::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::Ipv4RawSocketImpl")
    .SetParent<Socket> ()
    .SetGroupName ("Internet")
    .AddAttribute ("Protocol", "Protocol number to match.",
                   UintegerValue (0),
```

SOCKET


ns3::Ipv4RawSocketImpl
ns3::Ipv6RawSocketImpl
ns3::Object
ns3::Socket
ns3::PacketSocket
ns3::NscTcpSocketImpl
ns3::TcpSocketSmallAcks
ns3::TcpSocket
ns3::TcpSocketBase
ns3::TcpSocketMsgBase
TcpSocketHalfAck
ns3::UdpSocket
ns3::UdpSocketImpl
Socket logic for the NSC TCP sockets.

src/aodv/model/aodv-routing-protocol.cc

```
void
RoutingProtocol::SendTo (Ptr<Socket> socket, Ptr<Packet> packet, Ipv4Address destination)
{
    socket->SendTo (packet, 0, InetSocketAddress (destination, AODV_PORT));

}
```

src/network/model/socket.h

```
int SendTo (const uint8_t* buf, uint32_t size, uint32_t flags,
            const Address &address);

virtual int SendTo (Ptr<Packet> p, uint32_t flags,
                    const Address &toAddress) = 0;
```

```
namespace ns3 {

class NetDevice;
class Node;

/**
 * \class Ipv4RawSocketImpl
 * \brief IPv4 raw socket.
 * \ingroup socket
 *
 * A RAW Socket typically is used to ac
 * available through L4 sockets, e.g.,
 * particular care to define the Ipv4Ra
 * particular the Protocol attribute.
 */
class Ipv4RawSocketImpl : public Socket
{
```

Internet Stack

src/internet/helper/internet-stack-helper.cc

```
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);
    }
```

```
void
InternetStackHelper::SetRoutingHelper (const Ipv4RoutingHelper &routing)
{
  delete m_routing;
  m_routing = routing.Copy ();
}
```

```
void
InternetStackHelper::Install (NodeContainer c) const
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
      Install (*i);
    }
}
```

Continues…

/src/aodv/mode/aodv-routing-protocol.h

```
namespace ns3
{
namespace aodv
{
/**
 * \ingroup aodv
 *
 * \brief AODV routing protocol
 */
class RoutingProtocol : public Ipv4RoutingProtocol
{
public:
  static TypeId GetTypeId (void);
  static const uint32_t AODV_PORT;

  /// c-tor
  RoutingProtocol ();
  virtual ~RoutingProtocol();
  virtual void DoDispose ();
```

src/internet/helper/internet-stack-helper.cc

```
void
InternetStackHelper::Install (Ptr<Node> node) const
{
  if (m_ipv4Enabled)
    {
      if (node->GetObject<Ipv4> () != 0)
        {
          NS_FATAL_ERROR ("InternetStackHelper::Install (): Aggregating "
                          "an InternetStack to a node with an existing Ipv4 object");
          return;
        }

      CreateAndAggregateObjectFromTypeId (node, "ns3::ArpL3Protocol");
      CreateAndAggregateObjectFromTypeId (node, "ns3::Ipv4L3Protocol");
      CreateAndAggregateObjectFromTypeId (node, "ns3::Icmpv4L4Protocol");
      if (m_ipv4ArpJitterEnabled == false)
        {
          Ptr<ArpL3Protocol> arp = node->GetObject<ArpL3Protocol> ();
          NS_ASSERT (arp);
          arp->SetAttribute ("RequestJitter", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));
        }
      // Set routing
      Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
      Ptr<Ipv4RoutingProtocol> ipv4Routing = m_routing->Create (node);
      ipv4->SetRoutingProtocol (ipv4Routing);
    }

  if (m_ipv4Enabled || m_ipv6Enabled)
    {
      CreateAndAggregateObjectFromTypeId (node, "ns3::TrafficControlLayer");
      CreateAndAggregateObjectFromTypeId (node, "ns3::UdpL4Protocol");
      node->AggregateObject (m_tcpFactory.Create<Object> ());
      Ptr<PacketSocketFactory> factory = CreateObject<PacketSocketFactory> ();
      node->AggregateObject (factory);
    }
}
```

```
InternetStackHelper::InternetStackHelper ()
  : m_routing (0),
    m_routingv6 (0),
    m_ipv4Enabled (true),
    m_ipv6Enabled (true),
    m_ipv4ArpJitterEnabled (true),
    m_ipv6NsRsJitterEnabled (true)
```

```
class Ipv4L3Protocol : public Ipv4
{
public:
  /**
   * \brief Get the type ID.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  static const uint16_t PROT_NUMBER; //!< Protocol number (0x0800)

  Ipv4L3Protocol();
  virtual ~Ipv4L3Protocol ();

  /**
   * \enum DropReason
   * \brief Reason why a packet has been dropped.
   */
  enum DropReason
  {
    DROP_TTL_EXPIRED = 1,    /**< Packet TTL has expired */
    DROP_NO_ROUTE,    /**< No route to host */
    DROP_BAD_CHECKSUM,    /**< Bad checksum */
    DROP_INTERFACE_DOWN,    /**< Interface is down so can not send packet */
    DROP_ROUTE_ERROR,    /**< Route error */
    DROP_FRAGMENT_TIMEOUT /**< Fragment timeout exceeded */
  };

  /**
   * \brief Set node associated with this stack.
   * \param node node to set
   */
  void SetNode (Ptr<Node> node);

  // functions defined in base class Ipv4

  void SetRoutingProtocol (Ptr<Ipv4RoutingProtocol> routingProtocol);
  Ptr<Ipv4RoutingProtocol> GetRoutingProtocol (void) const;

  Ptr<Socket> CreateRawSocket (void);
  void DeleteRawSocket (Ptr<Socket> socket);
```

src/internet/model/ipv4-l3-protocol.cc

```
Ptr<Socket>
Ipv4L3Protocol::CreateRawSocket (void)
{
  NS_LOG_FUNCTION (this);
  Ptr<Ipv4RawSocketImpl> socket = CreateObject<Ipv4RawSocketImpl> ();
  socket->SetNode (m_node);
  m_sockets.push_back (socket);
  return socket;
}
```

src/internet/model/ipv4-raw-socket-impl.cc

```
int
Ipv4RawSocketImpl::SendTo (Ptr<Packet> p, uint32_t flags,
                           const Address &toAddress)
{
  NS_LOG_FUNCTION (this << p << flags << toAddress);
  if (!InetSocketAddress::IsMatchingType (toAddress))
    {
      m_err = Socket::ERROR_INVAL;
      return -1;
    }
  if (m_shutdownSend)
    {
      return 0;
    }
  InetSocketAddress ad = InetSocketAddress::ConvertFrom (toAddress);
  Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4> ();
  Ipv4Address dst = ad.GetIpv4 ();
  Ipv4Address src = m_src;
  if (ipv4->GetRoutingProtocol ())
    {
      Ipv4Header header;
      if (!m_iphdrincl)
        {
          header.SetDestination (dst);
          header.SetProtocol (m_protocol);
        }
      else
        {
          p->RemoveHeader (header);
          dst = header.GetDestination ();
          src = header.GetSource ();
        }
      SocketErrno errno_ = ERROR_NOTERROR; //do not use errno as it
      Ptr<Ipv4Route> route;
      Ptr<NetDevice> oif = m_boundnetdevice; //specify non-zero if
      if (!oif && src != Ipv4Address::GetAny ())
        {
          int32_t index = ipv4->GetInterfaceForAddress (src);
```

src/internet/model/ipv4.h

```
virtual void Send (Ptr<Packet> packet, Ipv4Address source,
                   Ipv4Address destination, uint8_t protocol, Ptr<Ipv4Route> route) = 0;
virtual void SendWithHeader (Ptr<Packet> packet, Ipv4Header ipHeader, Ptr<Ipv4Route> route) = 0;
```

Therefore the method will in the Ipv4L3Protocol

```
class Ipv4L3Protocol : public Ipv4
{
```

src/internet/model/ipv4-l3-protocol.cc

```
void
Ipv4L3Protocol::SendWithHeader (Ptr<Packet> packet,
                                Ipv4Header ipHeader,
                                Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this << packet << ipHeader << route);
  if (Node::ChecksumEnabled ())
    {
      ipHeader.EnableChecksum ();
    }
  SendRealOut (route, packet, ipHeader);
}
```

```
          NS_ASSERT (index >= 0);
          oif = ipv4->GetNetDevice (index);
          NS_LOG_LOGIC ("Set index " << oif << "from source " << src);
        }

      // TBD-- we could cache the route and just check its validity
      route = ipv4->GetRoutingProtocol ()->RouteOutput (p, header, oif, errno_);
      if (route != 0)
        {
          NS_LOG_LOGIC ("Route exists");
          uint32_t pktSize = p->GetSize ();
          if (!m_iphdrincl)
            {
              ipv4->Send (p, route->GetSource (), dst, m_protocol, route);
            }
          else
            {
              pktSize += header.GetSerializedSize ();
              ipv4->SendWithHeader (p, header, route);
            }
          NotifyDataSent (pktSize);
          NotifySend (GetTxAvailable ());
          return pktSize;
        }
      else
        {
          NS_LOG_DEBUG ("dropped because no outgoing route.");
          return -1;
        }
    }
  return 0;
```

src/internet/model/ipv4-l3-protocol.cc

```
void
Ipv4L3Protocol::Send (Ptr<Packet> packet,
                      Ipv4Address source,
                      Ipv4Address destination,
                      uint8_t protocol,
                      Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this << packet << source << destination << uint32_t (protocol) << route);

  Ipv4Header ipHeader;
  bool mayFragment = true;
  uint8_t ttl = m_defaultTtl;
  SocketIpTtlTag tag;
  bool found = packet->RemovePacketTag (tag);
  if (found)
    {
      ttl = tag.GetTtl ();
    }

  uint8_t tos = m_defaultTos;
  SocketIpTosTag ipTosTag;
  found = packet->RemovePacketTag (ipTosTag);
  if (found)
    {
      tos = ipTosTag.GetTos ();
    }

  // Handle a few cases:
  // 1) packet is destined to limited broadcast address
  // 2) packet is destined to a subnet-directed broadcast address
  // 3) packet is not broadcast, and is passed in with a route entry
  // 4) packet is not broadcast, and is passed in with a route entry but route->GetGateway is not set (e.g., on-demand)
  // 5) packet is not broadcast, and route is NULL (e.g., a raw socket call, or ICMP)
```

```
// 1) packet is destined to limited broadcast address or link-local multicast address
if (destination.IsBroadcast () || destination.IsLocalMulticast ())
  {
    NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 1:  limited broadcast");
    ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
    uint32_t ifaceIndex = 0;
    for (Ipv4InterfaceList::iterator ifaceIter = m_interfaces.begin ();
         ifaceIter != m_interfaces.end (); ifaceIter++, ifaceIndex++)
      {
        Ptr<Ipv4Interface> outInterface = *ifaceIter;
        bool sendIt = false;
        if (source == Ipv4Address::GetAny ())
          {
            sendIt = true;
          }
        for (uint32_t index = 0; index < outInterface->GetNAddresses (); index++)
          {
            if (outInterface->GetAddress (index).GetLocal () == source)
              {
                sendIt = true;
              }
          }
        if (sendIt)
          {
            Ptr<Packet> packetCopy = packet->Copy ();

            NS_ASSERT (packetCopy->GetSize () <= outInterface->GetDevice ()->GetMtu ());

            m_sendOutgoingTrace (ipHeader, packetCopy, ifaceIndex);
            CallTxTrace (ipHeader, packetCopy, m_node->GetObject<Ipv4> (), ifaceIndex);
            outInterface->Send (packetCopy, ipHeader, destination);
          }
      }
    return;
  }
```

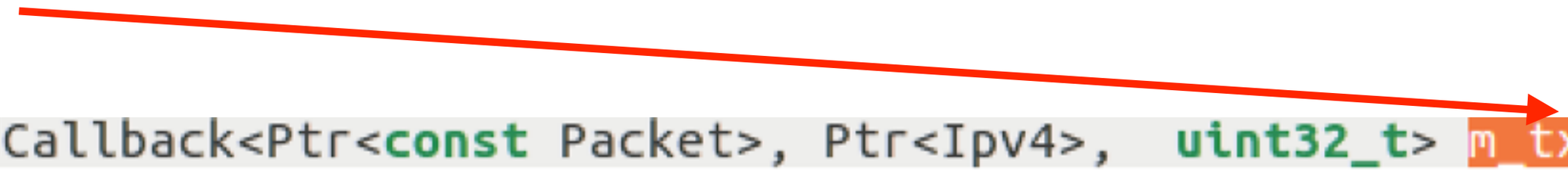
```
// 2) check: packet is destined to a subnet-directed broadcast address
uint32_t ifaceIndex = 0;
for (Ipv4InterfaceList::iterator ifaceIter = m_interfaces.begin ();
     ifaceIter != m_interfaces.end (); ifaceIter++, ifaceIndex++)
  {
    Ptr<Ipv4Interface> outInterface = *ifaceIter;
    for (uint32_t j = 0; j < GetNAddresses (ifaceIndex); j++)
      {
        Ipv4InterfaceAddress ifAddr = GetAddress (ifaceIndex, j);
        NS_LOG_LOGIC ("Testing address " << ifAddr.GetLocal () << " with mask " << ifAddr.GetMask ());
        if (destination.IsSubnetDirectedBroadcast (ifAddr.GetMask ()) &&
            destination.CombineMask (ifAddr.GetMask ()) == ifAddr.GetLocal ().CombineMask (ifAddr.GetMask ())   )
          {
            NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 2:  subnet directed bcast to " << ifAddr.GetLocal ());
            ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
            Ptr<Packet> packetCopy = packet->Copy ();
            m_sendOutgoingTrace (ipHeader, packetCopy, ifaceIndex);
            CallTxTrace (ipHeader, packetCopy, m_node->GetObject<Ipv4> (), ifaceIndex);
            outInterface->Send (packetCopy, ipHeader, destination);
            return;
          }
      }
  }

// 3) packet is not broadcast, and is passed in with a route entry
//    with a valid Ipv4Address as the gateway
if (route && route->GetGateway () != Ipv4Address ())
  {
    NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 3:  passed in with route");
    ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
    int32_t interface = GetInterfaceForDevice (route->GetOutputDevice ());
    m_sendOutgoingTrace (ipHeader, packet, interface);
    SendRealOut (route, packet->Copy (), ipHeader);
    return;
  }
```

```
// 4) packet is not broadcast, and is passed in with a route entry but route->GetGateway is not set (e.g., on-demand)
if (route && route->GetGateway () == Ipv4Address ())
  {
    // This could arise because the synchronous RouteOutput() call
    // returned to the transport protocol with a source address but
    // there was no next hop available yet (since a route may need
    // to be queried).
    NS_FATAL_ERROR ("Ipv4L3Protocol::Send case 4: This case not yet implemented");
  }
// 5) packet is not broadcast, and route is NULL (e.g., a raw socket call)
NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 5:  passed in with no route " << destination);
Socket::SocketErrno errno_;
Ptr<NetDevice> oif (0); // unused for now
ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
Ptr<Ipv4Route> newRoute;
if (m_routingProtocol != 0)
  {
    newRoute = m_routingProtocol->RouteOutput (packet, ipHeader, oif, errno_);
  }
else
  {
    NS_LOG_ERROR ("Ipv4L3Protocol::Send: m_routingProtocol == 0");
  }
if (newRoute)
  {
    int32_t interface = GetInterfaceForDevice (newRoute->GetOutputDevice ());
    m_sendOutgoingTrace (ipHeader, packet, interface);
    SendRealOut (newRoute, packet->Copy (), ipHeader);
  }
else
  {
    NS_LOG_WARN ("No route to host.  Drop.");
    m_dropTrace (ipHeader, packet, DROP_NO_ROUTE, m_node->GetObject<Ipv4> (), 0);
  }
}
```

```
void
Ipv4L3Protocol::SendRealOut (Ptr<Ipv4Route> route,
                             Ptr<Packet> packet,
                             Ipv4Header const &ipHeader)
{
  NS_LOG_FUNCTION (this << route << packet << &ipHeader);
  if (route == 0)
    {
      NS_LOG_WARN ("No route to host.  Drop.");
      m_dropTrace (ipHeader, packet, DROP_NO_ROUTE, m_node->GetObject<Ipv4> (), 0);
      return;
    }
  Ptr<NetDevice> outDev = route->GetOutputDevice ();
  int32_t interface = GetInterfaceForDevice (outDev);
  NS_ASSERT (interface >= 0);
  Ptr<Ipv4Interface> outInterface = GetInterface (interface);
  NS_LOG_LOGIC ("Send via NetDevice ifIndex " << outDev->GetIfIndex () << " ipv4InterfaceIndex " << interface);

  if (!route->GetGateway ().IsEqual (Ipv4Address ("0.0.0.0")))
    {
      if (outInterface->IsUp ())
        {
          NS_LOG_LOGIC ("Send to gateway " << route->GetGateway ());
          if ( packet->GetSize () + ipHeader.GetSerializedSize () > outInterface->GetDevice ()->GetMtu () )
    {
      std::list<Ipv4PayloadHeaderPair> listFragments;
      DoFragmentation (packet, ipHeader, outInterface->GetDevice ()->GetMtu (), listFragments);
      for ( std::list<Ipv4PayloadHeaderPair>::iterator it = listFragments.begin (); it != listFragments.end (); it++ )
        {
          CallTxTrace (it->second, it->first, m_node->GetObject<Ipv4> (), interface);
          outInterface->Send (it->first, it->second, route->GetGateway ());
        }
    }
  else
    {
      CallTxTrace (ipHeader, packet, m_node->GetObject<Ipv4> (), interface);
      outInterface->Send (packet, ipHeader, route->GetGateway ());
    }
```

```
            }
        }
      else
        {
          NS_LOG_LOGIC ("Dropping -- outgoing interface is down: " << route->GetGateway ());
          m_dropTrace (ipHeader, packet, DROP_INTERFACE_DOWN, m_node->GetObject<Ipv4> (), interface);
        }
    }
  else
    {
      if (outInterface->IsUp ())
        {
          NS_LOG_LOGIC ("Send to destination " << ipHeader.GetDestination ());
          if ( packet->GetSize () + ipHeader.GetSerializedSize () > outInterface->GetDevice ()->GetMtu () )
            {
              std::list<Ipv4PayloadHeaderPair> listFragments;
              DoFragmentation (packet, ipHeader, outInterface->GetDevice ()->GetMtu (), listFragments);
              for ( std::list<Ipv4PayloadHeaderPair>::iterator it = listFragments.begin (); it != listFragments.end (); it++ )
                {
                  NS_LOG_LOGIC ("Sending fragment " << *(it->first) );
                  CallTxTrace (it->second, it->first, m_node->GetObject<Ipv4> (), interface);
                  outInterface->Send (it->first, it->second, ipHeader.GetDestination ());
                }
            }
          else
            {
              CallTxTrace (ipHeader, packet, m_node->GetObject<Ipv4> (), interface);
              outInterface->Send (packet, ipHeader, ipHeader.GetDestination ());
            }
        }
      else
        {
          NS_LOG_LOGIC ("Dropping -- outgoing interface is down: " << ipHeader.GetDestination ());
          m_dropTrace (ipHeader, packet, DROP_INTERFACE_DOWN, m_node->GetObject<Ipv4> (), interface);
        }
    }
}
```

```
void
Ipv4L3Protocol::CallTxTrace (const Ipv4Header & ipHeader, Ptr<Packet> packet,
                             Ptr<Ipv4> ipv4, uint32_t interface)
{
  Ptr<Packet> packetCopy = packet->Copy ();
  packetCopy->AddHeader (ipHeader);
  m_txTrace (packetCopy, ipv4, interface);
}
```



```
TracedCallback<Ptr<const Packet>, Ptr<Ipv4>,  uint32_t> m_txTrace;
```

```
  Ptr<Ipv4Interface> outInterface = *ifaceIter;
outInterface->Send (packetCopy, ipHeader, destination);
```

```
void
Ipv4Interface::Send (Ptr<Packet> p, const Ipv4Header & hdr, Ipv4Address dest)
{
  NS_LOG_FUNCTION (this << *p << dest);
  if (!IsUp ())
    {
      return;
    }

  // Check for a loopback device, if it's the case we don't pass through
  // traffic control layer
  if (DynamicCast<LoopbackNetDevice> (m_device))
    {
      /// \todo additional checks needed here (such as whether multicast
      /// goes to loopback)?
      p->AddHeader (hdr);
      m_device->Send (p, m_device->GetBroadcast (), Ipv4L3Protocol::PROT_NUMBER);
      return;
    }

  NS_ASSERT (m_tc != 0);

  // is this packet aimed at a local interface ?
  for (Ipv4InterfaceAddressListCI i = m_ifaddrs.begin (); i != m_ifaddrs.end (); ++i)
    {
      if (dest == (*i).GetLocal ())
        {
          p->AddHeader (hdr);
          m_tc->Receive (m_device, p, Ipv4L3Protocol::PROT_NUMBER,
                         m_device->GetBroadcast (),
                         m_device->GetBroadcast (),
                         NetDevice::PACKET_HOST);
          return;
        }
    }
```

```
  if (m_device->NeedsArp ())
    {
      NS_LOG_LOGIC ("Needs ARP" << " " << dest);
      Ptr<ArpL3Protocol> arp = m_node->GetObject<ArpL3Protocol> ();
      Address hardwareDestination;
      bool found = false;
      if (dest.IsBroadcast ())
        {
          NS_LOG_LOGIC ("All-network Broadcast");
          hardwareDestination = m_device->GetBroadcast ();
          found = true;
        }
      else if (dest.IsMulticast ())
        {
          NS_LOG_LOGIC ("IsMulticast");
          NS_ASSERT_MSG (m_device->IsMulticast (),
                         "ArpIpv4Interface::SendTo (): Sending multicast packet over "
                         "non-multicast device");

          hardwareDestination = m_device->GetMulticast (dest);
          found = true;
        }
      else
        {
          for (Ipv4InterfaceAddressListCI i = m_ifaddrs.begin (); i != m_ifaddrs.end (); ++i)
            {
              if (dest.IsSubnetDirectedBroadcast ((*i).GetMask ()))
                {
                  NS_LOG_LOGIC ("Subnetwork Broadcast");
                  hardwareDestination = m_device->GetBroadcast ();
                  found = true;
                  break;
                }
            }
          if (!found)
            {
              NS_LOG_LOGIC ("ARP Lookup");
              found = arp->Lookup (p, hdr, dest, m_device, m_cache, &hardwareDestination);
```

```
        }
    }

  if (found)
    {
      NS_LOG_LOGIC ("Address Resolved.  Send.");
      m_tc->Send (m_device, Create<Ipv4QueueDiscItem> (p, hardwareDestination, Ipv4L3Protocol::PROT_NUMBER, hdr));
    }
  }
else
  {
    NS_LOG_LOGIC ("Doesn't need ARP");
    m_tc->Send (m_device, Create<Ipv4QueueDiscItem> (p, m_device->GetBroadcast (), Ipv4L3Protocol::PROT_NUMBER, hdr));
  }
}
```

```
Ptr<NetDevice> m_device; //!< The associated NetDevice
Ptr<TrafficControlLayer> m_tc; //!< The associated TrafficControlLayer
```

```
void
TrafficControlLayer::Send (Ptr<NetDevice> device, Ptr<QueueDiscItem> item)
{
  NS_LOG_FUNCTION (this << device << item);

  NS_LOG_DEBUG ("Send packet to device " << device << " protocol number " <<
                item->GetProtocol ());

std::map<Ptr<NetDevice>, NetDeviceInfo>::iterator qdMap = m_netDeviceQueueToQueueDiscMap.find (device);
NS_ASSERT (qdMap != m_netDeviceQueueToQueueDiscMap.end ());
Ptr<NetDeviceQueueInterface> devQueueIface = qdMap->second.first;
NS_ASSERT (devQueueIface);

// determine the transmission queue of the device where the packet will be enqueued
uint8_t txq = devQueueIface->GetSelectedQueue (item);
NS_ASSERT (txq < devQueueIface->GetTxQueuesN ());

if (qdMap->second.second.empty ())
  {
    // The device has no attached queue disc, thus add the header to the packet and
    // send it directly to the device if the selected queue is not stopped
    if (!devQueueIface->GetTxQueue (txq)->IsStopped ())
      {
        item->AddHeader ();
        device->Send (item->GetPacket (), item->GetAddress (), item->GetProtocol ());
      }
  }
else
  {
    // Enqueue the packet in the queue disc associated with the netdevice queue
    // selected for the packet and try to dequeue packets from such queue disc
    item->SetTxQueueIndex (txq);

    Ptr<QueueDisc> qDisc = qdMap->second.second[txq];
    NS_ASSERT (qDisc);
    qDisc->Enqueue (item);
    qDisc->Run ();
  }
```

net device

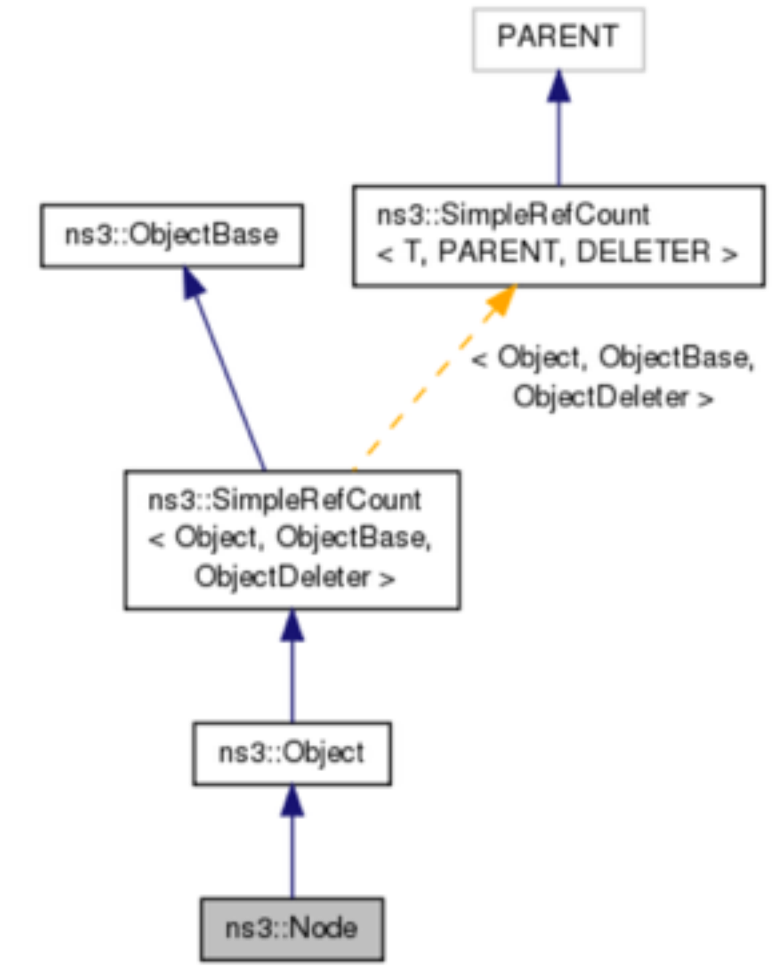
```
virtual bool Send (Ptr<Packet> packet, const Address& dest, uint16_t protocolNumber) =
```

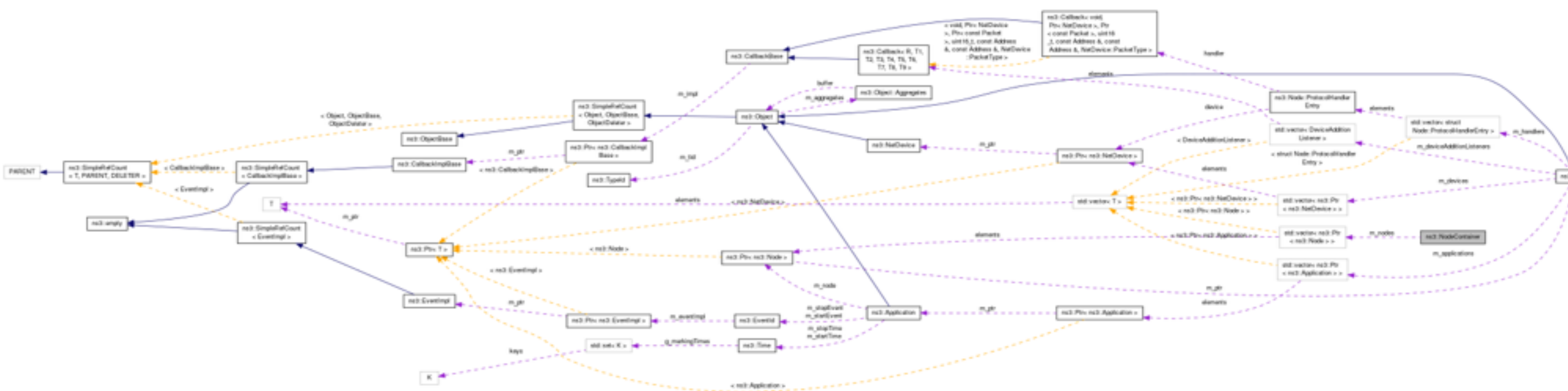
NODE

Node Container

/ns-allinone-3.25/ns-3.25/src/network/helper$

```
NodeContainer::Create (uint32_t n)
{
  for (uint32_t i = 0; i < n; i++)
    {
      m_nodes.push_back (CreateObject<Node> ());
    }
}
void

private:
  std::vector<Ptr<Node> > m_nodes; //!< Nodes smart pointers
};
```


PARENT
ns3::ObjectBase
ns3::SimpleRefCount
< T, PARENT, DELETER >
< Object, ObjectBase,
ObjectDeleter >
ns3::SimpleRefCount
< Object, ObjectBase,
ObjectDeleter >
ns3::Object
ns3::Node



AODV EXAMPLE

```
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);
    }
}
```

Summary

- **Ipv4RoutingHelper** uses **Ipv4RoutingProtocol** functions

/src/aodv/helper/aodv-helper.cc

Aodv

```
namespace ns3
{

AodvHelper::AodvHelper() :
  Ipv4RoutingHelper ()
{
  m_agentFactory.SetTypeId ("ns3::aodv::RoutingProtocol");
}
```

```
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);
    }
}
```

```
class AodvHelper : public Ipv4RoutingHelper
```

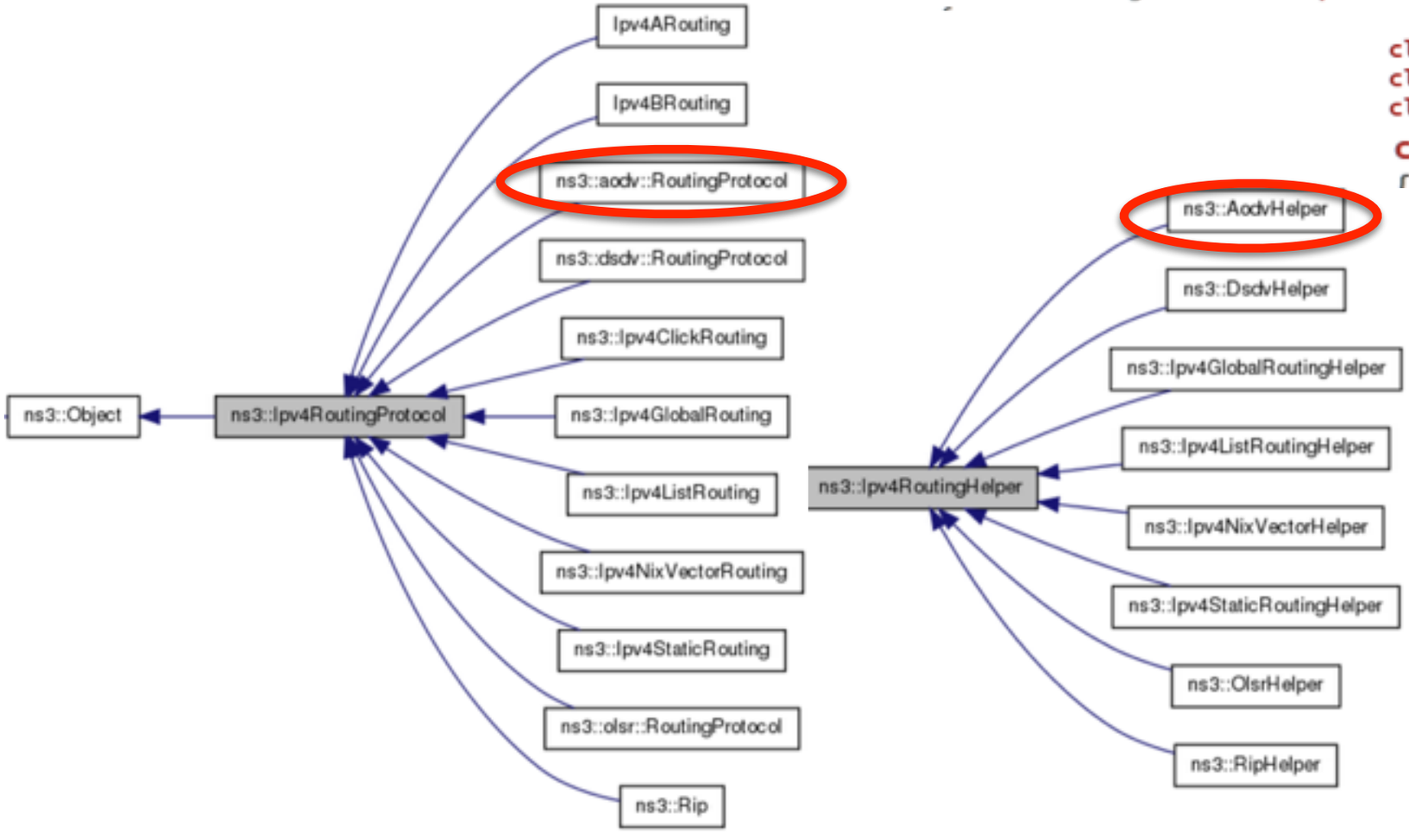
```
class Ipv4RoutingProtocol;
class Node;
```

class Ipv4RoutingHelper

```
namespace aodv
class RoutingProtocol : public Ipv4RoutingProtocol
```

```
class Ipv4MulticastRoute;
class Ipv4Route;
class NetDevice;
```

class Ipv4RoutingProtocol : public Object



Summary

```
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);
```

src/internet/helper/internet-stack-helper.cc

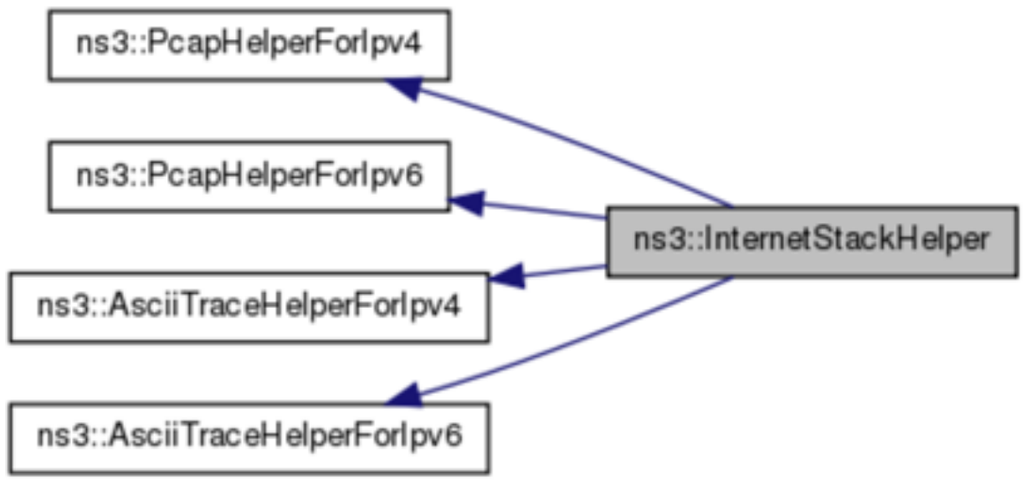
InternetStackHelper

```
class Node;
class Ipv4RoutingHelper;
class Ipv6RoutingHelper;
class InternetStackHelper : public PcapHelperForIpv4, public PcapHelperForIpv6,
                            public AsciiTraceHelperForIpv4, public AsciiTraceHelperForIpv6
```

```
InternetStackHelper::InternetStackHelper ()
  : m_routing (0),
    m_routingv6 (0),
    m_ipv4Enabled (true),
    m_ipv6Enabled (true),
    m_ipv4ArpJitterEnabled (true),
    m_ipv6NsRsJitterEnabled (true)

{
  Initialize ();
}
```

```
    /**
     * \brief IPv4 install state (enabled/disabled) ?
     */
    bool m_ipv4Enabled;
    /**
     * \brief IPv4 routing helper.
     */
    const Ipv4RoutingHelper *m_routing;
```

```
// private method called by both constructor and Reset ()
void
InternetStackHelper::Initialize ()
{
  SetTcp ("ns3::TcpL4Protocol");
  Ipv4StaticRoutingHelper staticRouting;
  Ipv4GlobalRoutingHelper globalRouting;
  Ipv4ListRoutingHelper listRouting;
  Ipv6StaticRoutingHelper staticRoutingv6;
  listRouting.Add (staticRouting, 0);
  listRouting.Add (globalRouting, -10);
  SetRoutingHelper (listRouting);
  SetRoutingHelper (staticRoutingv6);
}
```



Summary

```
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);
    }
}
```

```
class Node;
class Ipv4RoutingHelper;
class Ipv6RoutingHelper;
class InternetStackHelper : public PcapHelperForIpv4, public PcapHelperForIpv6,
                            public AsciiTraceHelperForIpv4, public AsciiTraceHelperForIpv6
{
  /**
   * \brief IPv4 install state (enabled/disabled) ?
   */
  bool m_ipv4Enabled;
  /**
   * \brief IPv4 routing helper.
   */
  const Ipv4RoutingHelper *m_routing;
```

src/internet/helper/internet-stack-helper.cc

InternetStackHelper

```
void
InternetStackHelper::SetRoutingHelper (const Ipv4RoutingHelper &routing)
{
  delete m_routing;
  m_routing = routing.Copy ();
}
```

```cpp
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);
    }
}
```

```cpp
void
InternetStackHelper::Install (NodeContainer c) const
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
      Install (*i);
    }
}
```

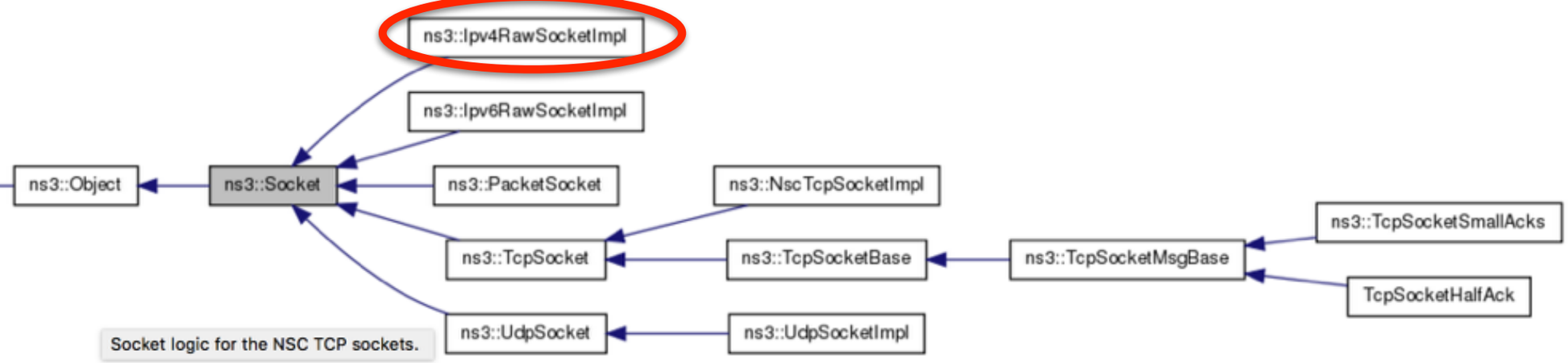
```cpp
void
InternetStackHelper::Install (Ptr<Node> node) const
{
  if (m_ipv4Enabled)
    {
      if (node->GetObject<Ipv4> () != 0)
        {
          NS_FATAL_ERROR ("InternetStackHelper::Install (): Aggregating "
                          "an InternetStack to a node with an existing Ipv4 object");
          return;
        }

      CreateAndAggregateObjectFromTypeId (node, "ns3::ArpL3Protocol");
      CreateAndAggregateObjectFromTypeId (node, "ns3::Ipv4L3Protocol");
      CreateAndAggregateObjectFromTypeId (node, "ns3::Icmpv4L4Protocol");
      if (m_ipv4ArpJitterEnabled == false)
        {
          Ptr<ArpL3Protocol> arp = node->GetObject<ArpL3Protocol> ();
          NS_ASSERT (arp);
          arp->SetAttribute ("RequestJitter", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));
        }
      // Set routing
      Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
      Ptr<Ipv4RoutingProtocol> ipv4Routing = m_routing->Create (node);
      ipv4->SetRoutingProtocol (ipv4Routing);
    }
```

```cpp
InternetStackHelper::InternetStackHelper ()
  : m_routing (0),
    m_routingv6 (0),
    m_ipv4Enabled (true),
    m_ipv6Enabled (true),
    m_ipv4ArpJitterEnabled (true),
    m_ipv6NsRsJitterEnabled (true)
```

```cpp
class Packet;
class NetDevice;
class Ipv4Interface;
class Ipv4Address;
class Ipv4Header;
class Ipv4RoutingTableEntry;
class Ipv4Route;
class Node;
class Socket;
class Ipv4RawSocketImpl;
class IpL4Protocol;
class Icmpv4L4Protocol;
```

```cpp
class Ipv4L3Protocol : public Ipv4
{
public:
  /**
   * \brief Get the type ID.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  static const uint16_t PROT_NUMBER; //!< Protocol number (0x0800)

  Ipv4L3Protocol();
  virtual ~Ipv4L3Protocol ();

  /**
   * \enum DropReason
   * \brief Reason why a packet has been dropped.
   */
  enum DropReason
  {
    DROP_TTL_EXPIRED = 1,   /**< Packet TTL has expired */
    DROP_NO_ROUTE,    /**< No route to host */
    DROP_BAD_CHECKSUM,    /**< Bad checksum */
    DROP_INTERFACE_DOWN,    /**< Interface is down so can not send packet */
    DROP_ROUTE_ERROR,   /**< Route error */
    DROP_FRAGMENT_TIMEOUT /**< Fragment timeout exceeded */
  };

  /**
   * \brief Set node associated with this stack.
   * \param node node to set
   */
  void SetNode (Ptr<Node> node);

  // functions defined in base class Ipv4

  void SetRoutingProtocol (Ptr<Ipv4RoutingProtocol> routingProtocol);
  Ptr<Ipv4RoutingProtocol> GetRoutingProtocol (void) const;

  Ptr<Socket> CreateRawSocket (void);
  void DeleteRawSocket (Ptr<Socket> socket);
```

```cpp
Ptr<Socket>
Ipv4L3Protocol::CreateRawSocket (void)
{
  NS_LOG_FUNCTION (this);
  Ptr<Ipv4RawSocketImpl> socket = CreateObject<Ipv4RawSocketImpl> ();
  socket->SetNode (m_node);
  m_sockets.push_back (socket);
  return socket;
```




```
void
RoutingProtocol::SendTo (Ptr<Socket> socket, Ptr<Packet> packet, Ipv4Address destination)
{
    socket->SendTo (packet, 0, InetSocketAddress (destination, AODV_PORT));
```

The Ipv4L3protocol created a Ipv4RawSocketImpl socket

```
namespace ns3 {

class NetDevice;
class Node;

/**
 * \class Ipv4RawSocketImpl
 * \brief IPv4 raw socket.
 * \ingroup socket
 *
 * A RAW Socket typically is used to access specific IP layers not usually
 * available through L4 sockets, e.g., ICMP. The implementer should take
 * particular care to define the Ipv4RawSocketImpl Attributes, and in
 * particular the Protocol attribute.
 */
class Ipv4RawSocketImpl : public Socket
{
public:
  /**
   Ipv4RawSocketImpl::Ipv4RawSocketImpl ()
   {
     NS_LOG_FUNCTION (this);
     m_err = Socket::ERROR_NOTERROR;
     m_node = 0;
     m_src = Ipv4Address::GetAny ();
     m_dst = Ipv4Address::GetAny ();
     m_protocol = 0;
     m_shutdownSend = false;
     m_shutdownRecv = false;
   }
```

```
int
Ipv4RawSocketImpl::SendTo (Ptr<Packet> p, uint32_t flags,
                           const Address &toAddress)
{
  NS_LOG_FUNCTION (this << p << flags << toAddress);
  if (!InetSocketAddress::IsMatchingType (toAddress))
    {
      m_err = Socket::ERROR_INVAL;
      return -1;
    }
  if (m_shutdownSend)
    {
      return 0;
    }
  InetSocketAddress ad = InetSocketAddress::ConvertFrom (toAddress);
  Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4> ();
  Ipv4Address dst = ad.GetIpv4 ();
  Ipv4Address src = m_src;
  if (ipv4->GetRoutingProtocol ())
    {
      Ipv4Header header;
      if (!m_iphdrincl)
        {
          header.SetDestination (dst);
          header.SetProtocol (m_protocol);
        }
      else
        {
          p->RemoveHeader (header);
          dst = header.GetDestination ();
          src = header.GetSource ();
        }
      SocketErrno errno_ = ERROR_NOTERROR; //do not use errno as it is the standard C last error number
      Ptr<Ipv4Route> route;
      Ptr<NetDevice> oif = m_boundnetdevice; //specify non-zero if bound to a source address
      if (!oif && src != Ipv4Address::GetAny ())
        {
          int32_t index = ipv4->GetInterfaceForAddress (src);
          NS_ASSERT (index >= 0);
          oif = ipv4->GetNetDevice (index);
          NS_LOG_LOGIC ("Set index " << oif << "from source " << src);
        }

      // TBD-- we could cache the route and just check its validity
      route = ipv4->GetRoutingProtocol ()->RouteOutput (p, header, oif, errno_);
      if (route != 0)
        {
          NS_LOG_LOGIC ("Route exists");
          uint32_t pktSize = p->GetSize ();
          if (!m_iphdrincl)
            {
              ipv4->Send (p, route->GetSource (), dst, m_protocol, route);
            }
          else
            {
              pktSize += header.GetSerializedSize ();
              ipv4->SendWithHeader (p, header, route);
            }
          NotifyDataSent (pktSize);
          NotifySend (GetTxAvailable ());
          return pktSize;
        }
      else
        {
          NS_LOG_DEBUG ("dropped because no outgoing route.");
          return -1;
        }
    }
  return 0;
```

src/internet/model/ipv4.h

```
virtual void Send (Ptr<Packet> packet, Ipv4Address source,
                   Ipv4Address destination, uint8_t protocol, Ptr<Ipv4Route> route) = 0;

virtual void SendWithHeader (Ptr<Packet> packet, Ipv4Header ipHeader, Ptr<Ipv4Route> route) = 0;
```

ipv4 defined the send and sendwithheader functions as virtual, remembering that the Ipv4L3protocol create the socket and therefore implement the functions

```
class Ipv4L3Protocol : public Ipv4
```

```
void
Ipv4L3Protocol::SendWithHeader (Ptr<Packet> packet,
                                Ipv4Header ipHeader,
                                Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this << packet << ipHeader << route);
  if (Node::ChecksumEnabled ())
    {
      ipHeader.EnableChecksum ();
    }
  SendRealOut (route, packet, ipHeader);
}
```

the send function is shown below since is quite extensive

src/internet/model/ipv4-l3-protocol.cc

```
void
Ipv4L3Protocol::Send (Ptr<Packet> packet,
                      Ipv4Address source,
                      Ipv4Address destination,
                      uint8_t protocol,
                      Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this << packet << source << destination << uint32_t (protocol) << route);

  Ipv4Header ipHeader;
  bool mayFragment = true;
  uint8_t ttl = m_defaultTtl;
  SocketIpTtlTag tag;
  bool found = packet->RemovePacketTag (tag);
  if (found)
    {
      ttl = tag.GetTtl ();
    }

  uint8_t tos = m_defaultTos;
  SocketIpTosTag ipTosTag;
  found = packet->RemovePacketTag (ipTosTag);
  if (found)
    {
      tos = ipTosTag.GetTos ();
    }

  // Handle a few cases:
  // 1) packet is destined to limited broadcast address
  // 2) packet is destined to a subnet-directed broadcast address
  // 3) packet is not broadcast, and is passed in with a route entry
  // 4) packet is not broadcast, and is passed in with a route entry but route->GetGateway is not set (e.g., on-demand)
  // 5) packet is not broadcast, and route is NULL (e.g., a raw socket call, or ICMP)
```

```
// 1) packet is destined to limited broadcast address or link-local multicast address
if (destination.IsBroadcast () || destination.IsLocalMulticast ())
  {
    NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 1:  limited broadcast");
    ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
    uint32_t ifaceIndex = 0;
    for (Ipv4InterfaceList::iterator ifaceIter = m_interfaces.begin ();
         ifaceIter != m_interfaces.end (); ifaceIter++, ifaceIndex++)
      {
        Ptr<Ipv4Interface> outInterface = *ifaceIter;
        bool sendIt = false;
        if (source == Ipv4Address::GetAny ())
          {
            sendIt = true;
          }
        for (uint32_t index = 0; index < outInterface->GetNAddresses (); index++)
          {
            if (outInterface->GetAddress (index).GetLocal () == source)
              {
                sendIt = true;
              }
          }
        if (sendIt)
          {
            Ptr<Packet> packetCopy = packet->Copy ();

            NS_ASSERT (packetCopy->GetSize () <= outInterface->GetDevice ()->GetMtu ());

            m_sendOutgoingTrace (ipHeader, packetCopy, ifaceIndex);
            CallTxTrace (ipHeader, packetCopy, m_node->GetObject<Ipv4> (), ifaceIndex);
            outInterface->Send (packetCopy, ipHeader, destination);
          }
      }
    return;
  }
```

```
// 2) check: packet is destined to a subnet-directed broadcast address
uint32_t ifaceIndex = 0;
for (Ipv4InterfaceList::iterator ifaceIter = m_interfaces.begin ();
     ifaceIter != m_interfaces.end (); ifaceIter++, ifaceIndex++)
  {
    Ptr<Ipv4Interface> outInterface = *ifaceIter;
    for (uint32_t j = 0; j < GetNAddresses (ifaceIndex); j++)
      {
        Ipv4InterfaceAddress ifAddr = GetAddress (ifaceIndex, j);
        NS_LOG_LOGIC ("Testing address " << ifAddr.GetLocal () << " with mask " << ifAddr.GetMask ());
        if (destination.IsSubnetDirectedBroadcast (ifAddr.GetMask ()) &&
            destination.CombineMask (ifAddr.GetMask ()) == ifAddr.GetLocal ().CombineMask (ifAddr.GetMask ())   )
          {
            NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 2:  subnet directed bcast to " << ifAddr.GetLocal ());
            ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
            Ptr<Packet> packetCopy = packet->Copy ();
            m_sendOutgoingTrace (ipHeader, packetCopy, ifaceIndex);
            CallTxTrace (ipHeader, packetCopy, m_node->GetObject<Ipv4> (), ifaceIndex);
            outInterface->Send (packetCopy, ipHeader, destination);
            return;
          }
      }
  }

// 3) packet is not broadcast, and is passed in with a route entry
//    with a valid Ipv4Address as the gateway
if (route && route->GetGateway () != Ipv4Address ())
  {
    NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 3:  passed in with route");
    ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
    int32_t interface = GetInterfaceForDevice (route->GetOutputDevice ());
    m_sendOutgoingTrace (ipHeader, packet, interface);
    SendRealOut (route, packet->Copy (), ipHeader);
    return;
  }
```

```
// 4) packet is not broadcast, and is passed in with a route entry but route->GetGateway is not set (e.g., on-demand)
if (route && route->GetGateway () == Ipv4Address ())
  {
    // This could arise because the synchronous RouteOutput() call
    // returned to the transport protocol with a source address but
    // there was no next hop available yet (since a route may need
    // to be queried).
    NS_FATAL_ERROR ("Ipv4L3Protocol::Send case 4: This case not yet implemented");
  }
// 5) packet is not broadcast, and route is NULL (e.g., a raw socket call)
NS_LOG_LOGIC ("Ipv4L3Protocol::Send case 5:  passed in with no route " << destination);
Socket::SocketErrno errno_;
Ptr<NetDevice> oif (0); // unused for now
ipHeader = BuildHeader (source, destination, protocol, packet->GetSize (), ttl, tos, mayFragment);
Ptr<Ipv4Route> newRoute;
if (m_routingProtocol != 0)
  {
    newRoute = m_routingProtocol->RouteOutput (packet, ipHeader, oif, errno_);
  }
else
  {
    NS_LOG_ERROR ("Ipv4L3Protocol::Send: m_routingProtocol == 0");
  }
if (newRoute)
  {
    int32_t interface = GetInterfaceForDevice (newRoute->GetOutputDevice ());
    m_sendOutgoingTrace (ipHeader, packet, interface);
    SendRealOut (newRoute, packet->Copy (), ipHeader);
  }
else
  {
    NS_LOG_WARN ("No route to host.  Drop.");
    m_dropTrace (ipHeader, packet, DROP_NO_ROUTE, m_node->GetObject<Ipv4> (), 0);
  }
}
```

sendrealout

```
void
Ipv4L3Protocol::SendRealOut (Ptr<Ipv4Route> route,
                             Ptr<Packet> packet,
                             Ipv4Header const &ipHeader)
{
  NS_LOG_FUNCTION (this << route << packet << &ipHeader);
  if (route == 0)
    {
      NS_LOG_WARN ("No route to host.  Drop.");
      m_dropTrace (ipHeader, packet, DROP_NO_ROUTE, m_node->GetObject<Ipv4> (), 0);
      return;
    }
  Ptr<NetDevice> outDev = route->GetOutputDevice ();
  int32_t interface = GetInterfaceForDevice (outDev);
  NS_ASSERT (interface >= 0);
  Ptr<Ipv4Interface> outInterface = GetInterface (interface);
  NS_LOG_LOGIC ("Send via NetDevice ifIndex " << outDev->GetIfIndex () << " ipv4InterfaceIndex " << interface);

  if (!route->GetGateway ().IsEqual (Ipv4Address ("0.0.0.0")))
    {
      if (outInterface->IsUp ())
        {
          NS_LOG_LOGIC ("Send to gateway " << route->GetGateway ());
          if ( packet->GetSize () + ipHeader.GetSerializedSize () > outInterface->GetDevice ()->GetMtu () )
            {
              std::list<Ipv4PayloadHeaderPair> listFragments;
              DoFragmentation (packet, ipHeader, outInterface->GetDevice ()->GetMtu (), listFragments);
              for ( std::list<Ipv4PayloadHeaderPair>::iterator it = listFragments.begin (); it != listFragments.end (); it++ )
                {
                  CallTxTrace (it->second, it->first, m_node->GetObject<Ipv4> (), interface);
                  outInterface->Send (it->first, it->second, route->GetGateway ());
                }
            }
          else
            {
              CallTxTrace (ipHeader, packet, m_node->GetObject<Ipv4> (), interface);
              outInterface->Send (packet, ipHeader, route->GetGateway ());
            }
        }
      else
        {
          NS_LOG_LOGIC ("Dropping -- outgoing interface is down: " << route->GetGateway ());
          m_dropTrace (ipHeader, packet, DROP_INTERFACE_DOWN, m_node->GetObject<Ipv4> (), interface);
        }
    }
  else
    {
      if (outInterface->IsUp ())
        {
          NS_LOG_LOGIC ("Send to destination " << ipHeader.GetDestination ());
          if ( packet->GetSize () + ipHeader.GetSerializedSize () > outInterface->GetDevice ()->GetMtu () )
            {
              std::list<Ipv4PayloadHeaderPair> listFragments;
              DoFragmentation (packet, ipHeader, outInterface->GetDevice ()->GetMtu (), listFragments);
              for ( std::list<Ipv4PayloadHeaderPair>::iterator it = listFragments.begin (); it != listFragments.end (); it++ )
                {
                  NS_LOG_LOGIC ("Sending fragment " << *(it->first) );
                  CallTxTrace (it->second, it->first, m_node->GetObject<Ipv4> (), interface);
                  outInterface->Send (it->first, it->second, ipHeader.GetDestination ());
                }
            }
          else
            {
              CallTxTrace (ipHeader, packet, m_node->GetObject<Ipv4> (), interface);
              outInterface->Send (packet, ipHeader, ipHeader.GetDestination ());
            }
        }
      else
        {
          NS_LOG_LOGIC ("Dropping -- outgoing interface is down: " << ipHeader.GetDestination ());
          m_dropTrace (ipHeader, packet, DROP_INTERFACE_DOWN, m_node->GetObject<Ipv4> (), interface);
        }
    }
}
```

```
void
Ipv4L3Protocol::CallTxTrace (const Ipv4Header & ipHeader, Ptr<Packet> packet,
                             Ptr<Ipv4> ipv4, uint32_t interface)
{
  Ptr<Packet> packetCopy = packet->Copy ();
  packetCopy->AddHeader (ipHeader);
  m_txTrace (packetCopy, ipv4, interface);
}
```

```
TracedCallback<Ptr<const Packet>, Ptr<Ipv4>,  uint32_t> m_txTrace;
```

For the outInterface send method

```
Ptr<Ipv4Interface> outInterface = *ifaceIter;
outInterface->Send (packetCopy, ipHeader, destination);
```

```
void
Ipv4Interface::Send (Ptr<Packet> p, const Ipv4Header & hdr, Ipv4Address dest)
{
  NS_LOG_FUNCTION (this << *p << dest);
  if (!IsUp ())
    {
      return;
    }

  // Check for a loopback device, if it's the case we don't pass through
  // traffic control layer
  if (DynamicCast<LoopbackNetDevice> (m_device))
    {
      /// \todo additional checks needed here (such as whether multicast
      /// goes to loopback)?
      p->AddHeader (hdr);
      m_device->Send (p, m_device->GetBroadcast (), Ipv4L3Protocol::PROT_NUMBER);
      return;
    }

  NS_ASSERT (m_tc != 0);

  // is this packet aimed at a local interface ?
  for (Ipv4InterfaceAddressListCI i = m_ifaddrs.begin (); i != m_ifaddrs.end (); ++i)
    {
      if (dest == (*i).GetLocal ())
        {
          p->AddHeader (hdr);
          m_tc->Receive (m_device, p, Ipv4L3Protocol::PROT_NUMBER,
                         m_device->GetBroadcast (),
                         m_device->GetBroadcast (),
                         NetDevice::PACKET_HOST);
          return;
        }
    }
```

```
  if (m_device->NeedsArp ())
    {
      NS_LOG_LOGIC ("Needs ARP" << " " << dest);
      Ptr<ArpL3Protocol> arp = m_node->GetObject<ArpL3Protocol> ();
      Address hardwareDestination;
      bool found = false;
      if (dest.IsBroadcast ())
        {
          NS_LOG_LOGIC ("All-network Broadcast");
          hardwareDestination = m_device->GetBroadcast ();
          found = true;
        }
      else if (dest.IsMulticast ())
        {
          NS_LOG_LOGIC ("IsMulticast");
          NS_ASSERT_MSG (m_device->IsMulticast (),
                         "ArpIpv4Interface::SendTo (): Sending multicast packet over "
                         "non-multicast device");

          hardwareDestination = m_device->GetMulticast (dest);
          found = true;
        }
      else
        {
          for (Ipv4InterfaceAddressListCI i = m_ifaddrs.begin (); i != m_ifaddrs.end (); ++i)
            {
              if (dest.IsSubnetDirectedBroadcast ((*i).GetMask ()))
                {
                  NS_LOG_LOGIC ("Subnetwork Broadcast");
                  hardwareDestination = m_device->GetBroadcast ();
                  found = true;
                  break;
                }
            }
          if (!found)
            {
              NS_LOG_LOGIC ("ARP Lookup");
              found = arp->Lookup (p, hdr, dest, m_device, m_cache, &hardwareDestination);
```

```
            }
        }

      if (found)
        {
          NS_LOG_LOGIC ("Address Resolved.  Send.");
          m_tc->Send (m_device, Create<Ipv4QueueDiscItem> (p, hardwareDestination, Ipv4L3Protocol::PROT_NUMBER, hdr));
        }
    }
  else
    {
      NS_LOG_LOGIC ("Doesn't need ARP");
      m_tc->Send (m_device, Create<Ipv4QueueDiscItem> (p, m_device->GetBroadcast (), Ipv4L3Protocol::PROT_NUMBER, hdr));
    }
}
```

```
Ptr<NetDevice> m_device; //!< The associated NetDevice
Ptr<TrafficControlLayer> m_tc; //!< The associated TrafficControlLayer
```

```
void
TrafficControlLayer::Send (Ptr<NetDevice> device, Ptr<QueueDiscItem> item)
{
  NS_LOG_FUNCTION (this << device << item);

  NS_LOG_DEBUG ("Send packet to device " << device << " protocol number " <<
                item->GetProtocol ());

  std::map<Ptr<NetDevice>, NetDeviceInfo>::iterator qdMap = m_netDeviceQueueToQueueDiscMap.find (device);
  NS_ASSERT (qdMap != m_netDeviceQueueToQueueDiscMap.end ());
  Ptr<NetDeviceQueueInterface> devQueueIface = qdMap->second.first;
  NS_ASSERT (devQueueIface);

  // determine the transmission queue of the device where the packet will be enqueued
  uint8_t txq = devQueueIface->GetSelectedQueue (item);
  NS_ASSERT (txq < devQueueIface->GetTxQueuesN ());

  if (qdMap->second.second.empty ())
    {
      // The device has no attached queue disc, thus add the header to the packet and
      // send it directly to the device if the selected queue is not stopped
      if (!devQueueIface->GetTxQueue (txq)->IsStopped ())
        {
          item->AddHeader ();
          device->Send (item->GetPacket (), item->GetAddress (), item->GetProtocol ());
        }
    }
  else
    {
      // Enqueue the packet in the queue disc associated with the netdevice queue
      // selected for the packet and try to dequeue packets from such queue disc
      item->SetTxQueueIndex (txq);

      Ptr<QueueDisc> qDisc = qdMap->second.second[txq];
      NS_ASSERT (qDisc);
      qDisc->Enqueue (item);
      qDisc->Run ();
    }
```

net device has his send method as virtual therefore the net device in this case a wifi net device will perform the send function

```
virtual bool Send (Ptr<Packet> packet, const Address& dest, uint16_t protocolNumber) = 0;
```

```cpp
WifiNetDevice::Send (Ptr<Packet> packet, const Address& dest, uint16_t protocolNumber)
{
  NS_LOG_FUNCTION (this << packet << dest << protocolNumber);
  NS_ASSERT (Mac48Address::IsMatchingType (dest));

  Mac48Address realTo = Mac48Address::ConvertFrom (dest);

  LlcSnapHeader llc;
  llc.SetType (protocolNumber);
  packet->AddHeader (llc);

  m_mac->NotifyTx (packet);
  m_mac->Enqueue (packet, realTo);
  return true;
}
```

```cpp
class WifiRemoteStationManager;
class WifiChannel;
class WifiPhy;
class WifiMac;

class WifiNetDevice : public NetDevice
```

```cpp
Ptr<WifiMac> m_mac;
```

```cpp
void
WifiMac::NotifyTx (Ptr<const Packet> packet)
{
  m_macTxTrace (packet);
}
```

```cpp
/**
 * The trace source fired when packets come into the "top" of the device
 * at the L3/L2 transition, before being queued for transmission.
 *
 * \see class CallBackTraceSource
 */
TracedCallback<Ptr<const Packet> > m_macTxTrace;
```

WifiMac have virtual the function enqueue, depends which mac user chose will be the implementation, in this case adhocmac

```
virtual void Enqueue (Ptr<const Packet> packet, Mac48Address to) = 0;
```

```
void
AdhocWifiMac::Enqueue (Ptr<const Packet> packet, Mac48Address to)
{
  NS_LOG_FUNCTION (this << packet << to);
  if (m_stationManager->IsBrandNew (to))
    {
      //In ad hoc mode, we assume that every destination supports all
      //the rates we support.
      if (m_htSupported || m_vhtSupported)
        {
          m_stationManager->AddAllSupportedMcs (to);
          m_stationManager->AddStationHtCapabilities (to, GetHtCapabilities());
        }
      if (m_vhtSupported)
        {
          m_stationManager->AddStationVhtCapabilities (to, GetVhtCapabilities());
        }
      m_stationManager->AddAllSupportedModes (to);
      m_stationManager->RecordDisassociated (to);
    }

  WifiMacHeader hdr;

  //If we are not a QoS STA then we definitely want to use AC_BE to
  //transmit the packet. A TID of zero will map to AC_BE (through \c
  //QosUtilsMapTidToAc()), so we use that as our default here.
  uint8_t tid = 0;

  //For now, a STA that supports QoS does not support non-QoS
  //associations, and vice versa. In future the STA model should fall
  //back to non-QoS if talking to a peer that is also non-QoS. At
  //that point there will need to be per-station QoS state maintained
  //by the association state machine, and consulted here.
  if (m_qosSupported)
    {
      hdr.SetType (WIFI_MAC_QOSDATA);
      hdr.SetQosAckPolicy (WifiMacHeader::NORMAL_ACK);
      hdr.SetQosNoEosp ();
      hdr.SetQosNoAmsdu ();
      //Transmission of multiple frames in the same TXOP is not
      //supported for now
      hdr.SetQosTxopLimit (0);

      //Fill in the QoS control field in the MAC header
      tid = QosUtilsGetTidForPacket (packet);
      //Any value greater than 7 is invalid and likely indicates that
      //the packet had no QoS tag, so we revert to zero, which will
      //mean that AC_BE is used.
      if (tid > 7)
        {
          tid = 0;
        }
      hdr.SetQosTid (tid);
    }
  else
    {
      hdr.SetTypeData ();
    }

  if (m_htSupported || m_vhtSupported)
    {
      hdr.SetNoOrder ();
    }
  hdr.SetAddr1 (to);
  hdr.SetAddr2 (m_low->GetAddress ());
  hdr.SetAddr3 (GetBssid ());
  hdr.SetDsNotFrom ();
  hdr.SetDsNotTo ();

  if (m_qosSupported)
    {
      //Sanity check that the TID is valid
      NS_ASSERT (tid < 8);
      m_edca[QosUtilsMapTidToAc (tid)]->Queue (packet, hdr);
    }
  else
    {
      m_dca->Queue (packet, hdr);
    }
}
```

```
/** This holds a pointer to the DCF instance for this WifiMac - used
for transmission of frames to non-QoS peers. */
Ptr<DcaTxop> m_dca;
```

```
void
DcaTxop::Queue (Ptr<const Packet> packet, const WifiMacHeader &hdr)
{
  NS_LOG_FUNCTION (this << packet << &hdr);
  WifiMacTrailer fcs;
  m_stationManager->PrepareForQueue (hdr.GetAddr1 (), &hdr, packet);
  m_queue->Enqueue (packet, hdr);
  StartAccessIfNeeded ();
}
```

```
Ptr<WifiMacQueue> m_queue;
```

```
void
WifiMacQueue::Enqueue (Ptr<const Packet> packet, const WifiMacHeader &hdr)
{
  Cleanup ();
  if (m_size == m_maxSize)
    {
      return;
    }
  Time now = Simulator::Now ();
  m_queue.push_back (Item (packet, hdr, now));
  m_size++;
}
```

```
/**
 * typedef for packet (struct Item) queue.
 */
typedef std::list<struct Item> PacketQueue;
/**
```

```
PacketQueue m_queue;
```