NS3 Source Code Update for 802.11bd

Refer to the below code changes to use 802.11bd (blind re-transmissions)

src/wave/helper/wave-helper.cc →

```
// Aniket Sukhija (Added wifi Standard code for 802.11bd)
NetDeviceContainer
WaveHelper::Install (const WifiPhyHelper &phyHelper,  const
WifiMacHelper &macHelper, NodeContainer c, const WifiStandard
wifiStandard) const
{
 try
   {
     [[maybe_unused]] const QosWaveMacHelper& qosMac =
dynamic_cast<const QosWaveMacHelper&> (macHelper);
   }
 catch (const std::bad_cast &)
   {
     NS_FATAL_ERROR ("WifiMacHelper should be the class or subclass
of QosWaveMacHelper");
   }

 NetDeviceContainer devices;
 for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
   {
     Ptr<Node> node = *i;
     Ptr<WaveNetDevice> device = CreateObject<WaveNetDevice> ();

     device->SetChannelManager (CreateObject<ChannelManager> ());
     device->SetChannelCoordinator (CreateObject<ChannelCoordinator>
());
     device->SetVsaManager (CreateObject<VsaManager> ());
     device->SetChannelScheduler
(m_channelScheduler.Create<ChannelScheduler> ());

     for (uint32_t j = 0; j != m_physNumber; ++j)
       {
         Ptr<WifiPhy> phy = phyHelper.Create (node, device);
         phy->ConfigureStandard (wifiStandard);
```

```
          phy->SetOperatingChannel (WifiPhy::ChannelTuple
{ChannelManager::GetCch (), 0,

WIFI_PHY_BAND_5GHZ, 0});
          device->AddPhy (phy);
        }

      for (std::vector<uint32_t>::const_iterator k =
m_macsForChannelNumber.begin ();
           k != m_macsForChannelNumber.end (); ++k)
        {
          Ptr<WifiMac> wifiMac = macHelper.Create (device,
wifiStandard);
          Ptr<OcbWifiMac> ocbMac = DynamicCast<OcbWifiMac> (wifiMac);
          ocbMac->SetWifiRemoteStationManager
(m_stationManager.Create<WifiRemoteStationManager> ());
          ocbMac->EnableForWave (device);
          device->AddMac (*k, ocbMac);
        }

      device->SetAddress (Mac48Address::Allocate ());

      node->AddDevice (device);
      devices.Add (device);
    }
 return devices;
}
// Aniket Sukhija
```

src/wave/model/wave-net-device.cc →

```
// packet->AddHeader (llc); (Aniket Sukhija, llc header removed)
```

src/wifi/model/wifi-phy.h →

```
/**
  * Configure WifiPhy with appropriate channel frequency and
  * supported rates for 802.11bd standard. (Aniket Sukhija)
  */
 void Configure80211bd (void);
```

src/wifi/model/frame-exchange-manager.cc →

```
bool
FrameExchangeManager::StartTransmission (Ptr<Txop> dcf)
{
 NS_LOG_FUNCTION (this << dcf);

 NS_ASSERT (m_mpdu == 0);
 if (m_txTimer.IsRunning ())
   {
     m_txTimer.Cancel ();
   }
 m_dcf = dcf;

 Ptr<WifiMacQueue> queue = dcf->GetWifiMacQueue ();

 // Even though channel access is requested when the queue is not
empty, at
 // the time channel access is granted the lifetime of the packet
might be
 // expired and the queue might be empty.
 if (queue->IsEmpty ())
   {
     NS_LOG_DEBUG ("Queue empty");
     m_dcf->NotifyChannelReleased ();
     m_dcf = 0;
     return false;
   }

 m_dcf->NotifyChannelAccessed ();
 Ptr<WifiMacQueueItem> mpdu = queue->Peek ()->GetItem ();
 // NS_LOG_UNCOND("packet uid is: "<<mpdu->GetPacket()->GetUid()<<"
Queue is: "<<mpdu->GetQueueAc());
 NS_ASSERT (mpdu != 0);
 NS_ASSERT (mpdu->GetHeader ().IsData () || mpdu->GetHeader ().IsMgt
());

 // assign a sequence number if this is not a fragment nor a
retransmission
 if (!mpdu->IsFragment () && !mpdu->GetHeader ().IsRetry ())
   {
```

```
     uint16_t sequence = m_txMiddle->GetNextSequenceNumberFor
(&mpdu->GetHeader ());
     mpdu->GetHeader ().SetSequenceNumber (sequence);
   }

 NS_LOG_DEBUG ("MPDU payload size=" << mpdu->GetPacketSize () <<
               ", to=" << mpdu->GetHeader ().GetAddr1 () <<
               ", seq=" << mpdu->GetHeader ().GetSequenceControl
());

 // check if the MSDU needs to be fragmented
 // mpdu = GetFirstFragmentIfNeeded (mpdu); // (Aniket Sukhija,
edited here)
  NS_ASSERT (m_protectionManager != 0);
 NS_ASSERT (m_ackManager != 0);
 // (Aniket Sukhija, too much changes)
 if (mpdu->GetHeader ().IsMoreFragments ())
 {
   if (m_fragmentedPacket == 0) { // the error is that
m_fragmentedPacket is not empty
     // NS_LOG_UNCOND("First Fragment:
"<<mpdu->GetPacket()->GetUid()<<" QueueAC: "<<mpdu->GetQueueAc());
     m_fragmentedPacket = mpdu->GetPacket ()->Copy ();
     Ptr<Packet> fragment = m_fragmentedPacket->CreateFragment (0,
m_mac->GetWifiRemoteStationManager ()->GetFragmentSize (mpdu, 0));
     Ptr<WifiMacQueueItem> item = Create<WifiMacQueueItem>
(fragment, mpdu->GetHeader (), mpdu->GetTimeStamp ());
     item->GetHeader().SetMoreFragments();
     m_mac->GetTxopQueue(mpdu->GetQueueAc())->Replace(mpdu, item);
     mpdu = item;
     m_mpdu = mpdu;
   } else {
     m_mpdu = mpdu; // this is the old mpdu
     Ptr<WifiMacQueueItem> item = GetNextFragment();
     m_mac->GetTxopQueue(mpdu->GetQueueAc())->Replace (mpdu, item);
     mpdu = item;
     // NS_LOG_UNCOND("kth Fragment:
"<<mpdu->GetPacket()->GetUid()<<" QueueAC: "<<mpdu->GetQueueAc());
     m_mpdu = mpdu; // this is the new mpdu
   }
```

```
   if (m_mpdu->GetHeader().IsMoreFragments()) m_moreFragments =
true;
   WifiTxParameters txParams;
   txParams.m_txVector = m_mac->GetWifiRemoteStationManager
()->GetDataTxVector (mpdu->GetHeader ());
   txParams.m_protection = m_protectionManager->TryAddMpdu (mpdu,
txParams);
   txParams.m_acknowledgment = m_ackManager->TryAddMpdu (mpdu,
txParams);
   txParams.AddMpdu (mpdu);
   UpdateTxDuration (mpdu->GetHeader ().GetAddr1 (), txParams);
   m_txParams = std::move (txParams);
   Time txDuration = m_phy->CalculateTxDuration (GetPsduSize
(m_mpdu, txParams.m_txVector),
                                                 txParams.m_txVector,
m_phy->GetPhyBand ());
   Simulator::Schedule (txDuration,
&FrameExchangeManager::TransmissionSucceeded, this);
   // transmit the MPDU
   ForwardMpduDown (m_mpdu, m_txParams.m_txVector);
   if (m_txParams.m_acknowledgment->method ==
WifiAcknowledgment::NONE)
   {
     // we are done with frames that do not require acknowledgment
     m_mpdu = 0;
   }
   if (!m_moreFragments) {
     // NS_LOG_UNCOND("Dequeuing the mpdu:
"<<mpdu->GetPacket()->GetUid()<<" QueueAC: "<<mpdu->GetQueueAc());
     // DequeueMpdu(mpdu);
     Simulator::Schedule(txDuration + m_phy->GetSifs(),
&FrameExchangeManager::DequeueMpdu, this, mpdu);
     m_fragmentedPacket = 0;
   }
   // NS_LOG_UNCOND("finally done: "<<mpdu->GetPacket()->GetUid());
 }
 else {
   // NS_LOG_UNCOND("single Fragment:
"<<mpdu->GetPacket()->GetUid()<<" QueueAC: "<<mpdu->GetQueueAc());
   WifiTxParameters txParams;
```

```
   txParams.m_txVector = m_mac->GetWifiRemoteStationManager
()->GetDataTxVector (mpdu->GetHeader ());
   txParams.m_protection = m_protectionManager->TryAddMpdu (mpdu,
txParams);
   txParams.m_acknowledgment = m_ackManager->TryAddMpdu (mpdu,
txParams);
   txParams.AddMpdu (mpdu);
   UpdateTxDuration (mpdu->GetHeader ().GetAddr1 (), txParams);
   SendMpduWithProtection (mpdu, txParams);
 }
 // (Aniket Sukhija, too many changes)


 return true;
}
```

```
Ptr<WifiMacQueueItem>
FrameExchangeManager::GetFirstFragmentIfNeeded
(Ptr<WifiMacQueueItem> mpdu)
{
 NS_LOG_FUNCTION (this << *mpdu);

 if (mpdu->IsFragment ())
   {
     // a fragment cannot be further fragmented
     NS_ASSERT (m_fragmentedPacket != 0);
   }
 else if (m_mac->GetWifiRemoteStationManager ()->NeedFragmentation
(mpdu))
   {
     NS_LOG_DEBUG ("Fragmenting the MSDU");
     m_fragmentedPacket = mpdu->GetPacket ()->Copy ();
     // create the first fragment
     Ptr<Packet> fragment = m_fragmentedPacket->CreateFragment (0,
m_mac->GetWifiRemoteStationManager ()->GetFragmentSize (mpdu, 0));
     // enqueue the first fragment
     Ptr<WifiMacQueueItem> item = Create<WifiMacQueueItem>
(fragment, mpdu->GetHeader (), mpdu->GetTimeStamp ());
     item->GetHeader ().SetMoreFragments ();
```

```
      m_mac->GetTxopQueue (mpdu->GetQueueAc ())->Replace (mpdu,
item); // (Aniket Sukhija, fucking with code)
      return item;
    }
 return mpdu;
}
```

src/wifi/model/wifi-phy.cc →

```
case WIFI_STANDARD_80211bd: // Aniket Sukhija (added 80211bd
function)
      Configure80211bd ();
      break;
```

```
// (Aniket Sukhija) TODO: more work to be done
void WifiPhy::Configure80211bd (void)
{
 NS_LOG_FUNCTION (this);
 if (GetChannelWidth () == 20)
    {
      AddPhyEntity (WIFI_MOD_CLASS_OFDM, Create<OfdmPhy>
(OFDM_PHY_DEFAULT));

      // See Table 17-21 "OFDM PHY characteristics" of 802.11-2016
      SetSifs (MicroSeconds (32));
      SetSlot (MicroSeconds (13));
      SetPifs (GetSifs () + GetSlot ());
      m_ackTxTime = MicroSeconds (88);
    }
 else if (GetChannelWidth () == 10)
    {
      AddPhyEntity (WIFI_MOD_CLASS_OFDM, Create<OfdmPhy>
(OFDM_PHY_10_MHZ));

      // See Table 17-21 "OFDM PHY characteristics" of 802.11-2016
      SetSifs (MicroSeconds (64));
      SetSlot (MicroSeconds (21));
      SetPifs (GetSifs () + GetSlot ());
      m_ackTxTime = MicroSeconds (176);
    }
```

```
 else
   {
     NS_FATAL_ERROR ("802.11bd configured with a wrong channel
width!");
   }
}
```

src/wifi/model/wifi-standards.h →

```
enum WifiStandard
{
 WIFI_STANDARD_UNSPECIFIED,
 WIFI_STANDARD_80211a,
 WIFI_STANDARD_80211b,
 WIFI_STANDARD_80211g,
 WIFI_STANDARD_80211p,
 WIFI_STANDARD_80211bd, // added 80211bd as new wifi standard
(Aniket Sukhija)
 WIFI_STANDARD_80211n,
 WIFI_STANDARD_80211ac,
 WIFI_STANDARD_80211ax
};
```

```
inline std::ostream& operator<< (std::ostream& os, WifiStandard
standard)
{
 switch (standard)
   {
   case WIFI_STANDARD_80211a:
     return (os << "802.11a");
   case WIFI_STANDARD_80211b:
     return (os << "802.11b");
   case WIFI_STANDARD_80211g:
     return (os << "802.11g");
   case WIFI_STANDARD_80211p:
     return (os << "802.11p");
   case WIFI_STANDARD_80211n:
     return (os << "802.11n");
   case WIFI_STANDARD_80211ac:
     return (os << "802.11ac");
```

```
    case WIFI_STANDARD_80211ax:
      return (os << "802.11ax");
    case WIFI_STANDARD_80211bd:
      return (os << "802.11bd"); // (Aniket Sukhija)
    default:
      return (os << "UNSPECIFIED");
    }
}
const std::map<WifiStandard, std::list<WifiPhyBand>> wifiStandards =
{
 { WIFI_STANDARD_80211a, { WIFI_PHY_BAND_5GHZ } },
 { WIFI_STANDARD_80211b, { WIFI_PHY_BAND_2_4GHZ } },
 { WIFI_STANDARD_80211g, { WIFI_PHY_BAND_2_4GHZ } },
 { WIFI_STANDARD_80211p, { WIFI_PHY_BAND_5GHZ } },
 { WIFI_STANDARD_80211bd, { WIFI_PHY_BAND_5GHZ } }, // (Aniket
Sukhija)
 { WIFI_STANDARD_80211n, { WIFI_PHY_BAND_2_4GHZ, WIFI_PHY_BAND_5GHZ
} },
 { WIFI_STANDARD_80211ac, { WIFI_PHY_BAND_5GHZ } },
 { WIFI_STANDARD_80211ax, { WIFI_PHY_BAND_2_4GHZ,
WIFI_PHY_BAND_5GHZ, WIFI_PHY_BAND_6GHZ } }
};
```

```
enum FrequencyChannelType : uint8_t
{
 WIFI_PHY_DSSS_CHANNEL = 0,
 WIFI_PHY_OFDM_CHANNEL,
 WIFI_PHY_80211p_CHANNEL,
 WIFI_PHY_80211bd_CHANNEL // (Aniket Sukhija, yet to edit here)
};
```

```
inline FrequencyChannelType GetFrequencyChannelType (WifiStandard
standard)
{
 switch (standard)
    {
      case WIFI_STANDARD_80211b:
        return WIFI_PHY_DSSS_CHANNEL;
      case WIFI_STANDARD_80211p:
```

```
      return WIFI_PHY_80211p_CHANNEL; // (Aniket Sukhija, yet to
edit here)
    case WIFI_STANDARD_80211bd:
      return WIFI_PHY_80211bd_CHANNEL; // (Aniket Sukhija, yet to
edit here)
    default:
      return WIFI_PHY_OFDM_CHANNEL;
  }
}
```

```
inline uint16_t GetMaximumChannelWidth (WifiStandard standard)
{
 switch (standard)
   {
    case WIFI_STANDARD_80211b:
      return 22;
    case WIFI_STANDARD_80211p:
      return 10;
    case WIFI_STANDARD_80211a:
    case WIFI_STANDARD_80211g:
      return 20;
    case WIFI_STANDARD_80211n:
    case WIFI_STANDARD_80211bd: // (Aniket Sukhija)
      return 40;
    case WIFI_STANDARD_80211ac:
    case WIFI_STANDARD_80211ax:
      return 160;
    default:
      NS_ABORT_MSG ("Unknown standard: " << standard);
      return 0;
   }
}
```

```
inline uint16_t GetDefaultChannelWidth (WifiStandard standard,
WifiPhyBand band)
{
 switch (standard)
   {
   case WIFI_STANDARD_80211b:
     return 22;
```

```
    case WIFI_STANDARD_80211p:
      return 10;
    case WIFI_STANDARD_80211bd: // (Aniket Sukhija)
      return 20;
    case WIFI_STANDARD_80211ac:
      return 80;
    case WIFI_STANDARD_80211ax:
      return (band == WIFI_PHY_BAND_2_4GHZ ? 20 : 80);
    default:
      return 20;
    }
}
```

```
inline WifiPhyBand GetDefaultPhyBand (WifiStandard standard)
{
 switch (standard)
    {
    case WIFI_STANDARD_80211p:
    case WIFI_STANDARD_80211bd: // (Aniket Sukhija)
    case WIFI_STANDARD_80211a:
    case WIFI_STANDARD_80211ac:
    case WIFI_STANDARD_80211ax:
      return WIFI_PHY_BAND_5GHZ;
    default:
      return WIFI_PHY_BAND_2_4GHZ;
    }
}
```

src/wifi/model/wifi-phy-operating-channel.cc →

```
// 802.11bd 10 MHz channels at the 5.855-5.925 band (for
simplification, we consider the same center frequencies as the 10
MHz channels)
 { std::make_tuple (171, 5860, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 { std::make_tuple (173, 5870, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 { std::make_tuple (175, 5880, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 { std::make_tuple (177, 5890, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
```

```
 { std::make_tuple (179, 5900, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 { std::make_tuple (181, 5910, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 { std::make_tuple (183, 5920, 10, WIFI_PHY_80211bd_CHANNEL,
WIFI_PHY_BAND_5GHZ) },
 // (Aniket Sukhija)
```

src/wifi/model/wifi-remote-station-manager.cc →

```
bool
WifiRemoteStationManager::NeedFragmentation (Ptr<const
WifiMacQueueItem> mpdu)
{
 NS_LOG_FUNCTION (this << *mpdu);
 // if (mpdu->GetHeader ().GetAddr1 ().IsGroup ()) // (Aniket
Sukhija, unable to fragment because IsGroup comes to be true)
 //   {
 //     return false;
 //   }
 bool normally = mpdu->GetSize () > GetFragmentationThreshold ();
 NS_LOG_DEBUG ("WifiRemoteStationManager::NeedFragmentation result:
" << std::boolalpha << normally);
 return DoNeedFragmentation (Lookup (mpdu->GetHeader ().GetAddr1
()), mpdu->GetPacket (), normally);
}
```

```
void
WifiRemoteStationManager::DoSetFragmentationThreshold (uint32_t
threshold)
{
 NS_LOG_FUNCTION (this << threshold);
 uint32_t thres = 32;
 if (threshold < thres) // (Edited from 256 to 32)
   {
     /*
      * ASN.1 encoding of the MAC and PHY MIB (256 ... 8000)
      */
     NS_LOG_WARN ("Fragmentation threshold should be larger than
128. Setting to 128.");
     m_fragmentationThreshold = thres;
```

```
   }
 else
   {
     /*
      * The length of each fragment shall be an even number of
octets, except for the last fragment if an MSDU or
      * MMPDU, which may be either an even or an odd number of
octets.
      */
     if (threshold & 1) // (Aniket Sukhija)
       {
         NS_LOG_WARN ("Fragmentation threshold should be an even
number. Setting to " << threshold - 1);
         m_fragmentationThreshold = threshold - 1;
       }
     else
       {
         m_fragmentationThreshold = threshold;
       }
   }
}
```

src/wifi/model/wifi-default-ack-manager.cc →

```
if (receiver.IsGroup ()) // (Aniket Sukhija, here comes the error)
   {
     NS_ABORT_MSG_IF (txParams.GetSize (receiver) > 0,
                      "Unicast frames only can be aggregated");
     WifiNoAck* acknowledgment = new WifiNoAck;
     if (hdr.IsQosData ())
       {
         acknowledgment->SetQosAckPolicy (receiver, hdr.GetQosTid
(),
                                          WifiMacHeader::NO_ACK);
       }
     return std::unique_ptr<WifiAcknowledgment> (acknowledgment);
   }
```

src/wave/model/ocb-wifi-mac.cc →

```
if (GetQosSupported ())
   {
```

```
      // Sanity check that the TID is valid
      NS_ASSERT (tid < 8);
      Ptr<WifiRemoteStationManager> wifiRemoteStationManager =
GetWifiRemoteStationManager();
      uint32_t fragmentationThreshold =
wifiRemoteStationManager->GetFragmentationThreshold();
      if (packet->GetSize() > fragmentationThreshold)
hdr.SetMoreFragments();  // Aniket Sukhija
      // NS_LOG_UNCOND("packet Uid: "<<packet->GetUid()<<" QueueAC:
"<<QosUtilsMapTidToAc (tid));
      GetQosTxop (tid)->Queue (packet, hdr);
    }
  else
    {
      GetTxop ()->Queue (packet, hdr);
    }
```