ns-3 training: Wi-Fi

Sébastien Deronne, June 2019

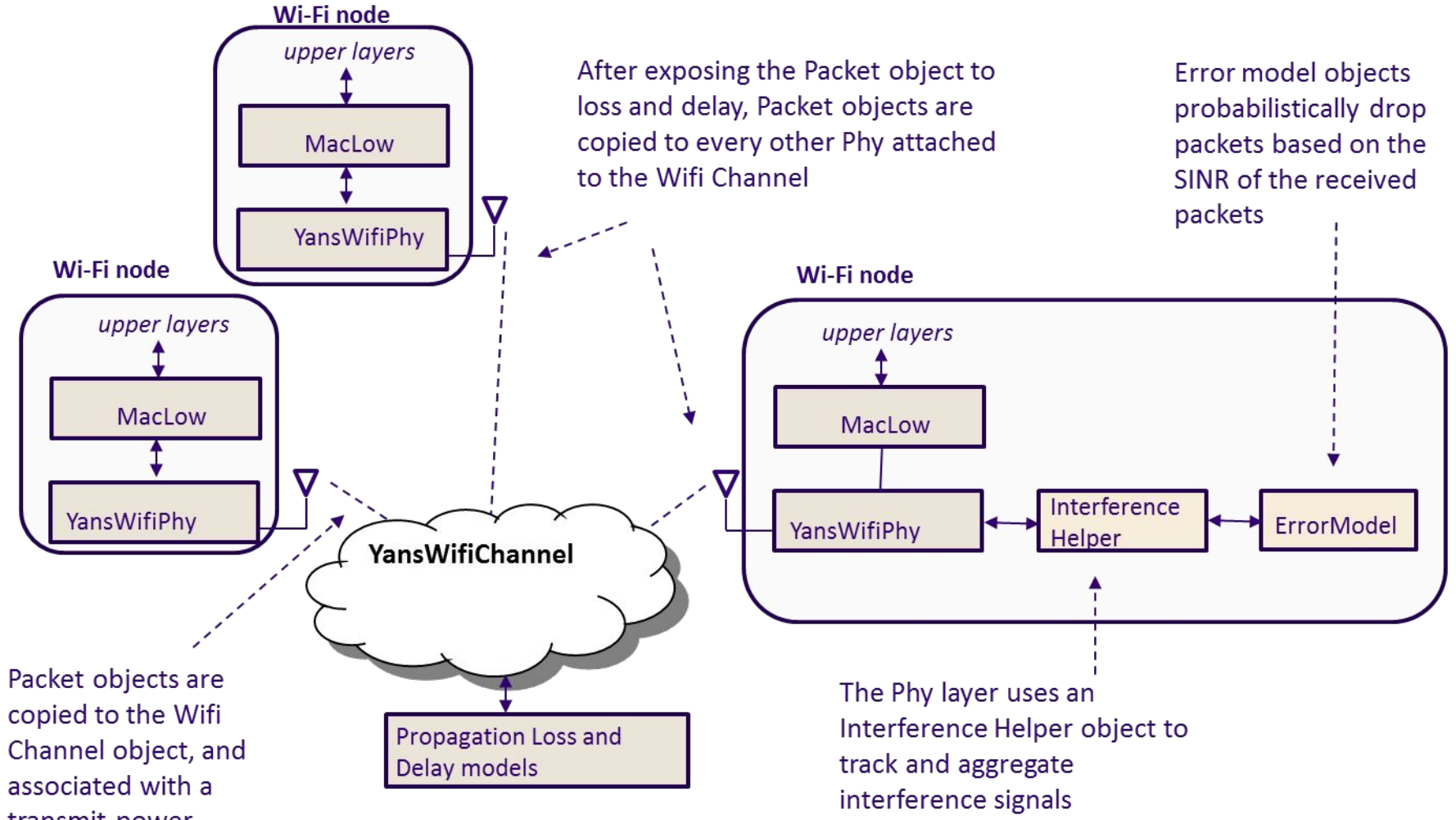
Outline

- **Wi-Fi module overview:**
  - Abstraction elements
  - MAC layer
  - PHY layer
  - Related models: propagation, mobility & energy
- **Usage:**
  - Configuration helpers
  - Simulation outputs: logging, traces and PCAP files
  - DCF and EDCA examples
  - 11n/ac/ax simulations
  - Spectrum examples

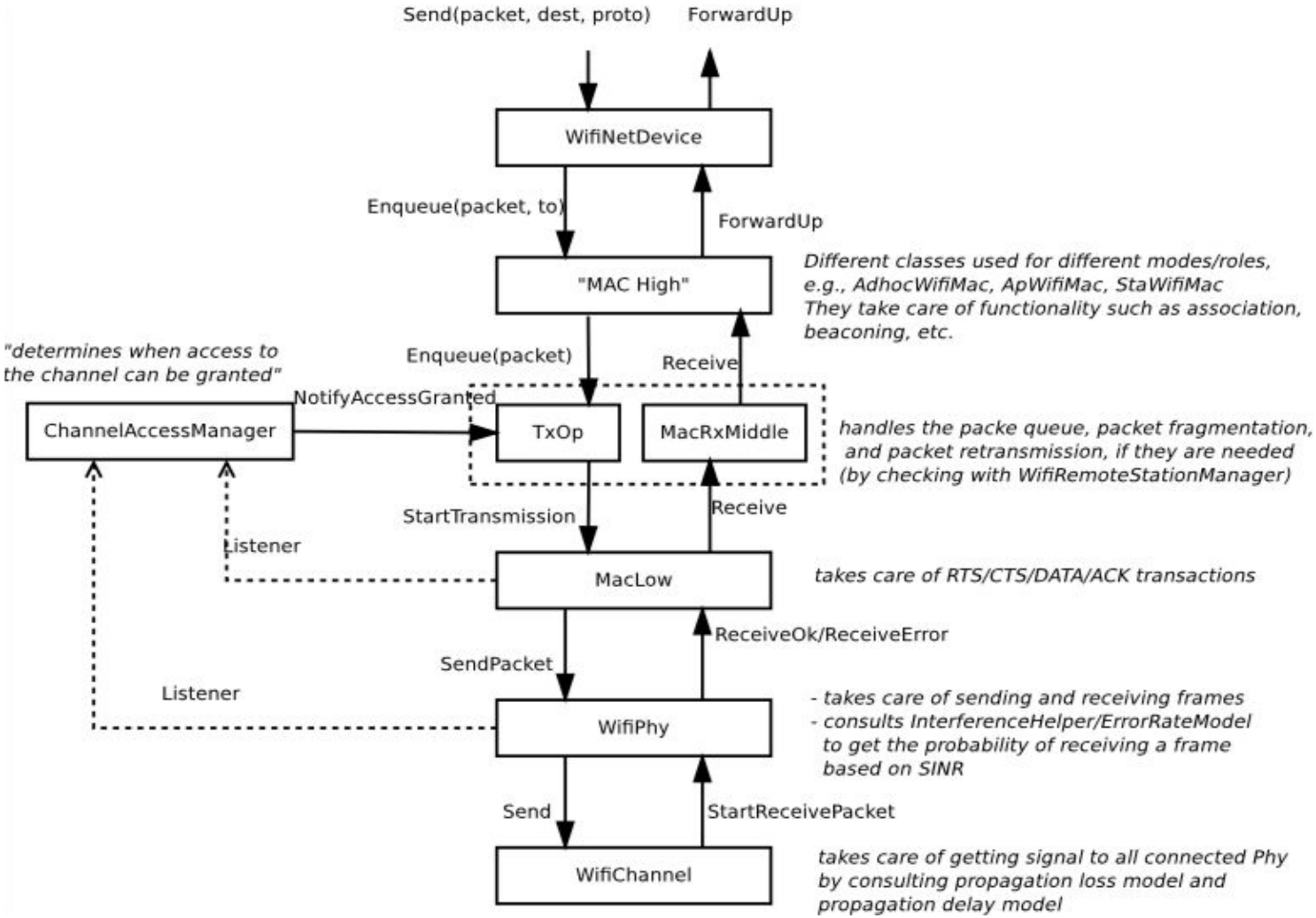
Wi-Fi Overview

- Supported features:
  - 802.11 a/b/g/n/ac/ax (2.4 & 5 GHz) PHYs
  - DCF implementation (Basic + RTS/CTS)
  - PCF implementation
  - QoS support (EDCA)
  - MSDU/MPDU aggregation and block ACK
  - Infrastructure and ad-hoc modes
  - Many rate adaptation algorithms
  - AWGN-based error models
- Unsupported features:
  - 11ac advanced features (TX beamforming, MU-MIMO, …)
  - 11ax features (OFDMA, …)
  - Frequency selective fading models

Wi-Fi abstraction elements



Wi-Fi architecture



MAC layer

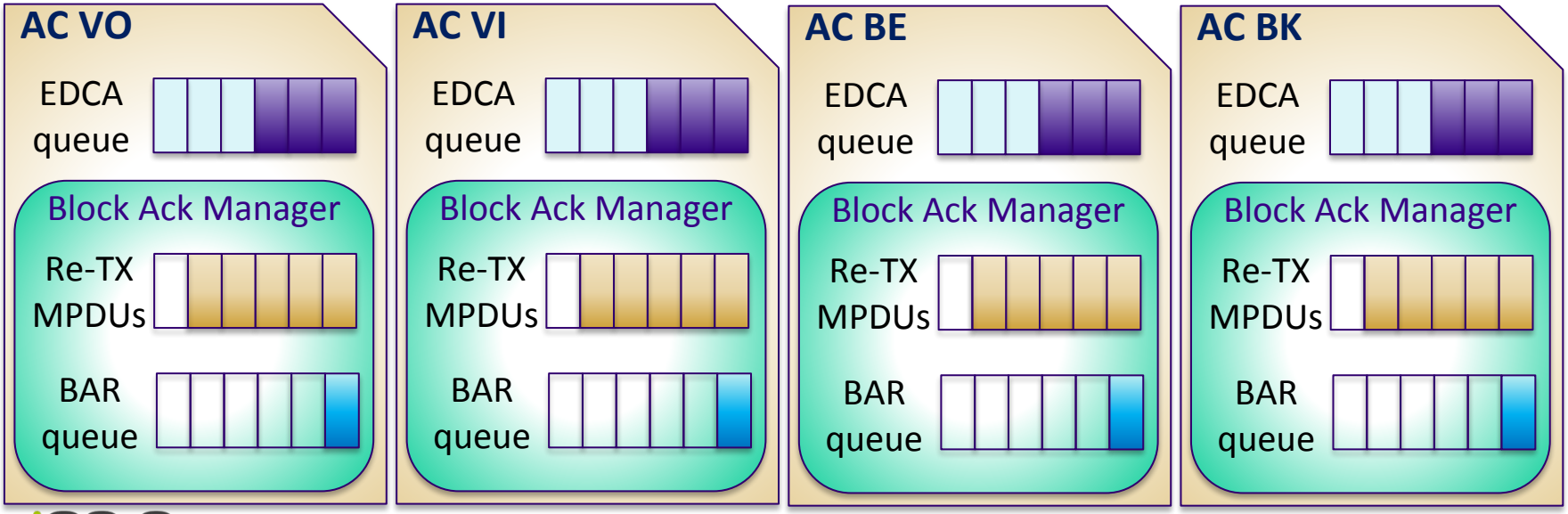
- 3 MAC (high) models:
  - **AdhocWifiMac**:
    simplest one, for adhoc mode.
  - **ApWifiMac**:
    access point in infrastructure mode, sends beacons and handles associations requests.
  - **StaWifiMac**:
    station in infrastructure mode, keeps associated to the access point.

MAC layer (cont.)

- **MacTxMiddle/MaxRxMiddle:**
  - Attaches sequence numbers;
  - Reassembles fragmented packets;
  - Discards duplicated frames by verifying the received sequence numbers.
- **Txop/QosTxop:**
  - Queue data and management packets;
  - Fragmentation/Retransmissions;
  - MSDU aggregation and block ACK sessions;
  - 1 Txop, 4 QosTxop (one per TID).
- **ChannelAccessManager**:
  - Determine when a transmission can start based on listeners (physical and virtual).
  - Txop/QosTxop typically request access, ChannelAccessManager grants access.
- **MacLow:**
  - RTS/CTS/DATA/ACK transactions;
  - MPDU Aggregation and block ACKs.

QoS support

- QosTxop:
  - Correspond to a given access class (AC): voice, video, best-effort and background;
  - Maintain several queues.
- An AC that gains channel access looks for frames in the following order:
  - Block Ack Requests (BARs) queue
  - Re-transmission queue
  - EDCA queue (from upper layer)



Frame aggregation

- 11n/ac/ax feature
- A-MSDU is attempted if the MSDU is dequeued from the EDCA queue (handled in QosTxop)
- A-MPDU is attempted if there exists a Block Ack agreement with the receiver (handled in MacLow)
- Constraints for A-MSDU and A-MPDU aggregation:
  - Maximum A-MSDU size;
  - Maximum A-MPDU size;
  - Maximum PPDU TX duration;
  - Additional constraints on TX duration (remaining TXOP duration, …)

Rate control

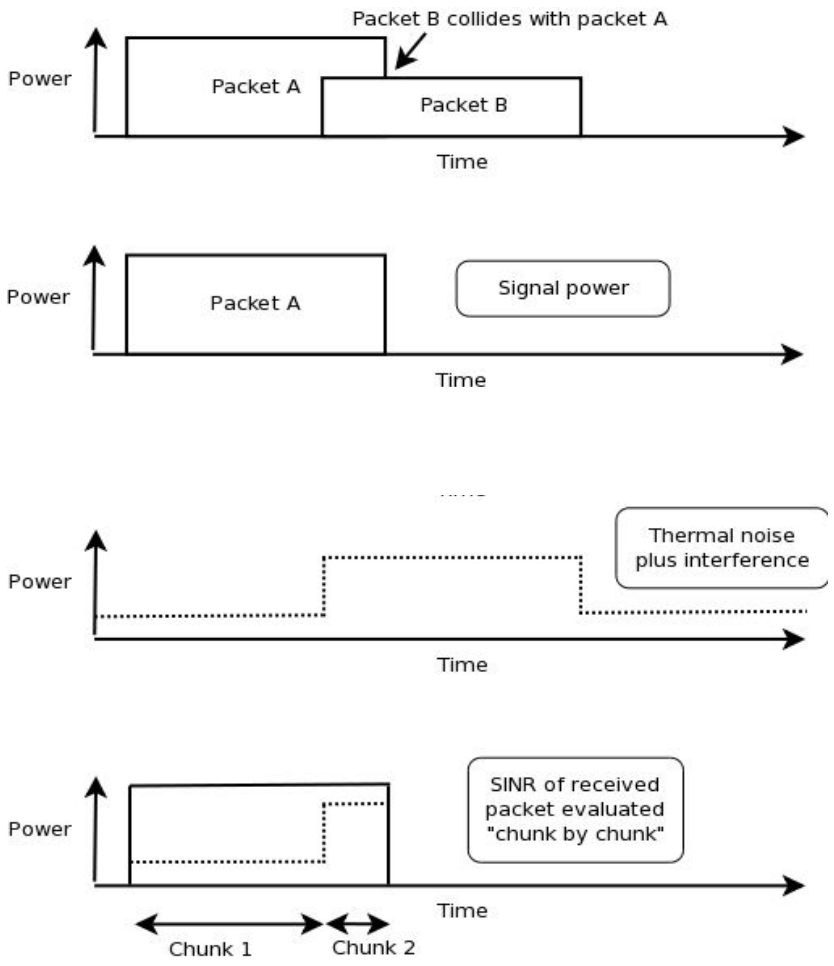
- Rate control algorithm is used to determine at which rate a frame should be sent by the PHY.
- Supported rate control algorithms:
  - Algorithms found in real devices:
    ArfWifiManager (default for WifiHelper), OnoeWifiManager, ConstantRateWifiManager, MinstrelWifiManager, MinstrelHtWifiManager (for 802.11n/ac).
  - Algorithms found in literature:
    IdealWifiManager, AarfWifiManager, AmrrWifiManager, CaraWifiManager, RraaWifiManager, AarfcdWifiManager, ParfWifiManager, AparfWifiManager.
- 802.11n/ac can use Constant, Ideal or MinstrelHT, 802.11ax is currently limited to be used with Constant and Ideal.

PHY layer

- Provides a packet-level abstraction.
- 2 models: YANS & Spectrum
- Most of the (common) logic implemented in WifiPhy.
- Handle packet transmission and reception:
  - each packet received is probabilistically evaluated for successful or failed reception (random number versus PER);
  - an object exists to track (InterferenceHelper) all received signals so that the correct interference power for each packet can be computed when a reception decision has to be made;
  - error models corresponding to the modulation and standard are used to look up probability of successful reception

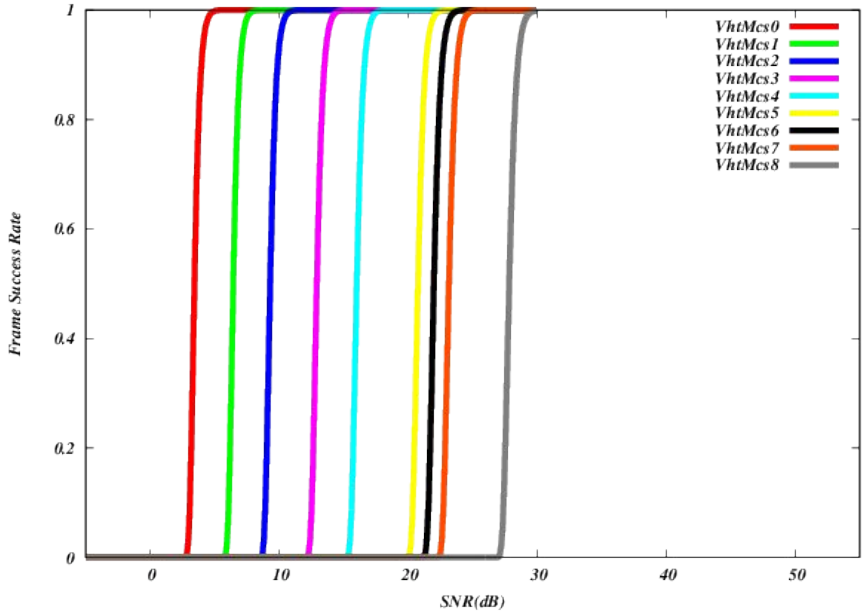
Interference helper

- Object that tracks power of all incoming packets
- Calculates probability of error values for packet being received:
  - Called for various portions in the packet (PHY headers and PHY payload)
  - SINR is evaluated on chunk-by-chunk basis.
  - Call the error rate model to determine PER of each chunk.
  - PER = product of all the PERs.

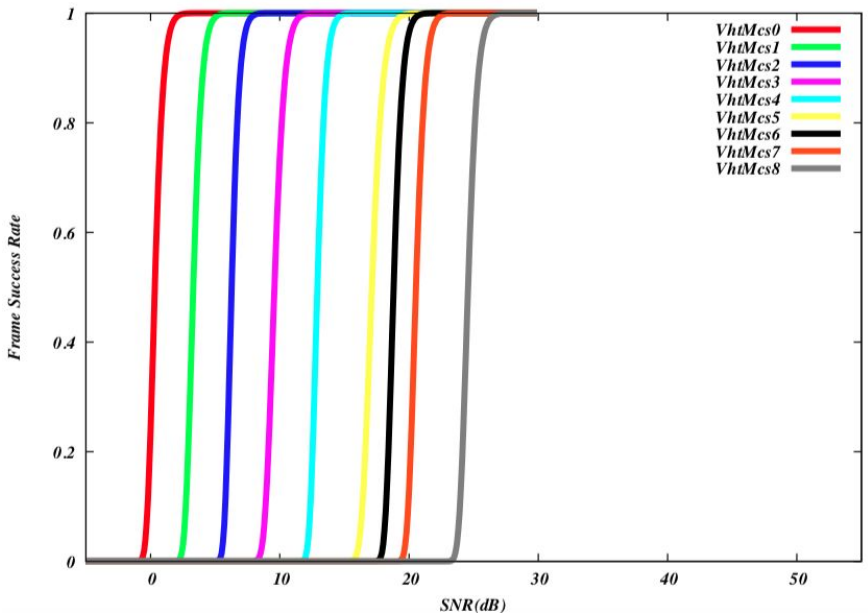


Error models

- Computes a probability of error for given SINR and modulation.
- Based on analytical models with error bounds.
- Supports additive white gaussian noise channels (AWGN) only; any potential fast fading effects are not modeled.
- Three implementations with different bounds: YansErrorRateModel, NistErrorRateModel (default), DsssErrorRateModel (802.11b).



NistErrorRateModel (802.11ac)



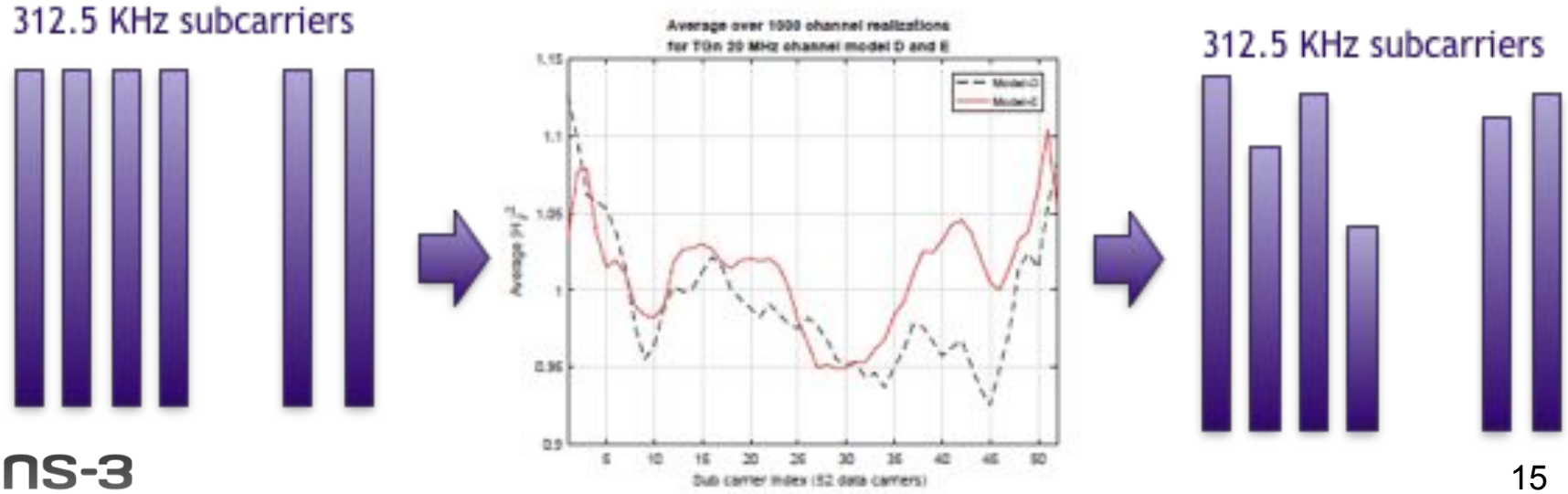
YansErrorRateModel (802.11ac)

PHY state machine

- WifiPhyStateHelper manages the state machine of the PHY layer.
- It allows other objects to hook as listeners to monitor PHY state (MAC layer, energy module, …).
- The PHY layer can be in one of seven states:
  - **TX**: the PHY is currently transmitting a signal on behalf of its associated MAC
  - **RX**: the PHY is synchronized on a signal and is waiting until it has received its last bit to forward it to the MAC.
  - **CCA Busy**: the PHY is not in TX or RX state but the measured energy is higher than the energy detection threshold.
  - **IDLE**: the PHY is not in the TX, RX, or CCA BUSY states.
  - **SWITCHING**: the PHY is switching channels.
  - **SLEEP**: the PHY is in a power save mode and cannot send nor receive frames.
  - **OFF**: the PHY is switched off and cannot send nor receive frames (used only if energy model)

YansWifiPhy vs SpectrumWifiPhy (cont.)

- **YansWifiPhy**:
  - Legacy implementation;
  - Works together with YansWifiChannel, which determines when a packet should be copied to a YansWifiPhy object attached to the channel (using propagation delay model) and what should be its received power (using propagation loss models).
- **SpectrumWifiPhy**:
  - Decompose Wi-Fi signals into subcarrier representation;
  - Can work together with other wireless technology attached to the Spectrum channel.



YansWifiPhy vs SpectrumWifiPhy (cont.)

	YansWifiPhy	SpectrumWifiPhy
Signals	Wi-Fi only	Permits to include multiple technologies coexisting on the same channel
Frequency-level decomposition	No	Yes
Simulation time	Fast	Slow

Which one to choose?

- Simulations involving mixed technologies (Wi-Fi, LTE, …) => **Spectrum**
- Simulation involving frequency dependent effects => **Spectrum**
- Simulation involving other Wi-Fi networks working in adjacent channels or using different channel bonding configurations => **Spectrum**
- Otherwise => **Yans (faster) or Spectrum**

Propagation Models

- Propagation module defines:
  - Propagation delay models:
    Calculate the time for signals to travel from the TX antennas to RX antennas.
  - Propagation loss models:
    Calculate the Rx signal power considering the Tx signal power and the mutual Rx and Tx antennas positions.

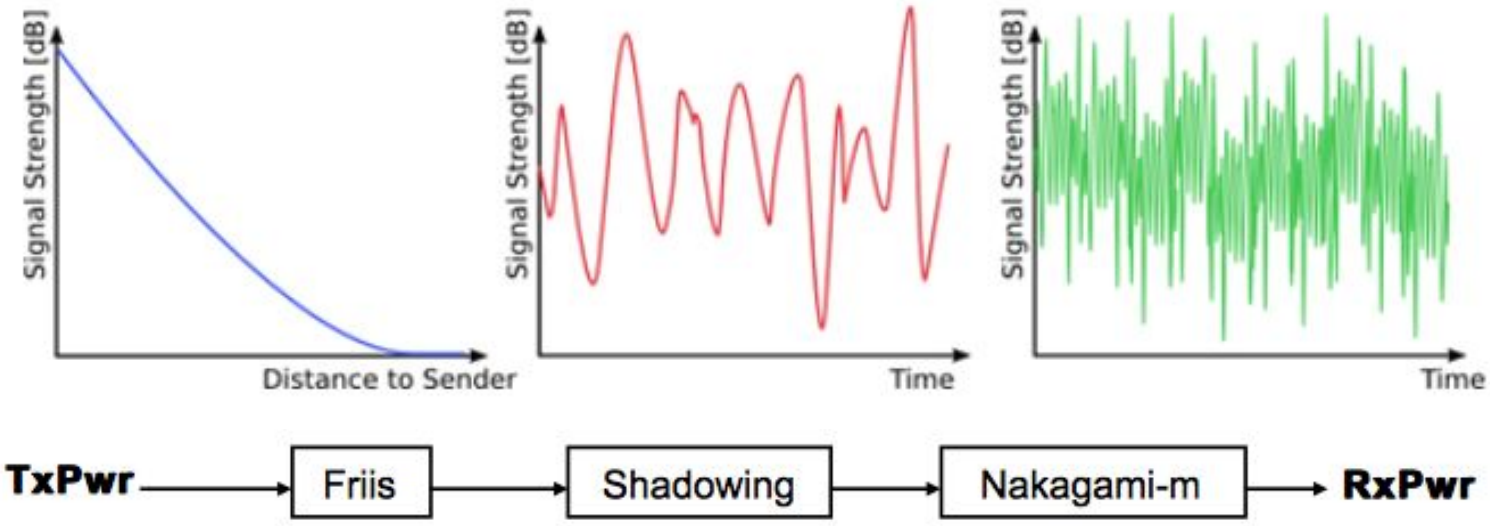
- Propagation delay model should always be set to:
  - ConstantSpeedPropagationDelayModel: In this model, the signal travels with constant speed. The delay is calculated according with the transmitter and receiver positions.

Propagation Models (cont.)

- Propagation loss models:
  - Many propagation loss models are implemented:
    - ✔ Abstract propagation loss models:
      FixedRss, Range, Random, Matrix, …
    - ✔ Deterministic path loss models:
      Friis, LogDistance, ThreeLogDistance, TwoRayGround, …
    - ✔ Stochastic fading models:
      Nakagami, Jakes, …
  - Extension for Spectrum:
    Some of the models have been extended for Spectrum in order to provide a different RX power per sub-band.

Propagation Models (3)

– A propagation loss model can be “chained” to another one, making a list. The final Rx power takes into account all the chained models.
Example: path loss model + shadowing model + fading model



(!) Be careful when using YansWifiChannelHelper::Default(),
the default LogDistance propagation model is added.
Calling AddPropagationLoss() again will add a second propagation loss model.

MobilityHelper: mobility and position

- The MobilityHelper combines a mobility model and position allocator.
- Position Allocators setup initial position of nodes (only used when simulation starts):
  - **List**: allocate positions from a deterministic list specified by the user;
  - **Grid**: allocate positions on a rectangular 2D grid (row first or column first);
  - **Random position allocators**: allocate random positions within a selected form (rectangle, circle, …).
- Mobility models specify how nodes will move during the simulation:
  - **Constant**: position, velocity or acceleration;
  - **Waypoint**: specify the location for a given time (time-position pairs);
  - **Trace-file based**: parse files and convert into ns-3 mobility events, support mobility tools such as SUMO, BonnMotion (using NS2 format) , TraNS, …

Position Allocation Examples

- **List**

```
MobilityHelper mobility;
// place two nodes at specific positions (100,0) and (0,100)
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (100, 0, 0));
positionAlloc->Add (Vector (0, 100, 0));
mobility.SetPositionAllocator(positionAlloc);
```

- **Grid Position**

```
MobilityHelper mobility;
// setup the grid itself: nodes are laid out started from (-100,-100) with 20 per row, the x
// interval between each object is 5 meters and the y interval between each object is 20 meters
mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", DoubleValue (-100.0), "MinY", DoubleValue (-100.0),
                               "DeltaX", DoubleValue (5.0), "DeltaY", DoubleValue (20.0),
                               "GridWidth", UintegerValue (20), "LayoutType", StringValue ("RowFirst"));
```

- **Random Rectangle Position**

```
// place nodes uniformly on a straight line from (0, 100)
MobilityHelper mobility;
Ptr<RandomRectanglePositionAllocator> positionAloc = CreateObject<RandomRectanglePositionAllocator>();
positionAloc->SetAttribute("X", StringValue("ns3::UniformRandomVariable[Min=0.0|Max=100.0]"));
positionAloc->SetAttribute("Y", StringValue("ns3::ConstantRandomVariable[Constant=50.0]"));
mobility.SetPositionAllocator(positionAloc);
```

Mobility Model Example

- **Constant Position**

```
MobilityHelper mobility;
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (nodes);
```

- **Constant Speed**

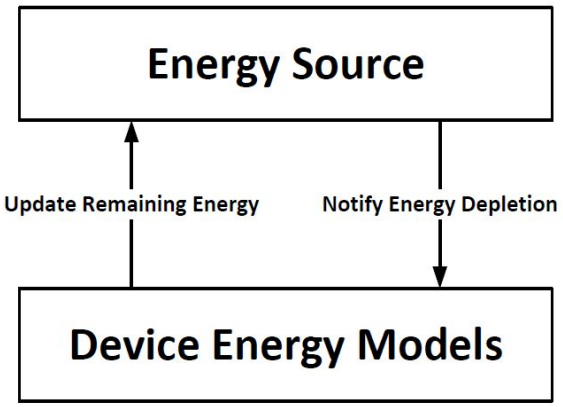
```
MobilityHelper mobility;
mobility.SetMobilityModel ("ns3::ConstantVelocityMobilityModel");
mobility.Install (nodes);
Ptr<UniformRandomVariable> rvar = CreateObject<UniformRandomVariable>();
for (NodeContainer::Iterator i = nodes.Begin (); i != nodes.End (); ++i) {
  Ptr<Node> node = (*i);
  double speed = rvar->GetValue(15, 25); //random speed between 15 and 25 m/s
  node->GetObject<ConstantVelocityMobilityModel>()->SetVelocity(Vector(speed,0,0));
}
```

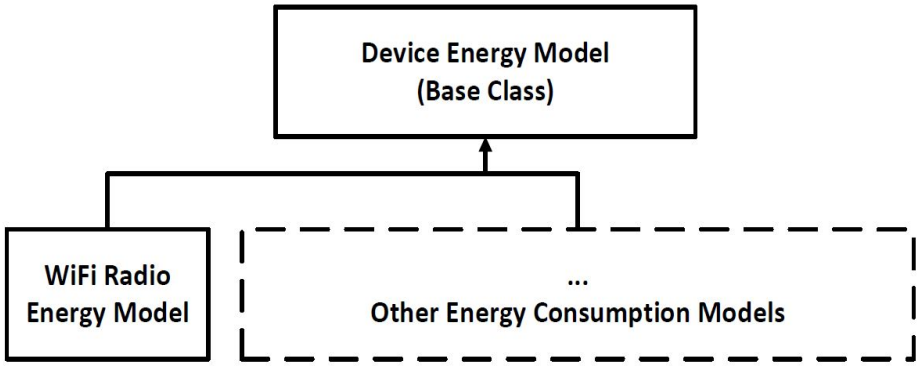
- **Trace-file based**

```
// Create Ns2MobilityHelper with the specified trace log file as parameter
Ns2MobilityHelper ns2 = Ns2MobilityHelper (“mobility_trace.txt”);
ns2.Install (); // configure movements for each node, while reading trace file
```

Wifi Radio Energy Model (1)

- Wi-Fi devices have different energy consumption based on state
  ⇒ current (A) per state configurable via WifiRadioEnergyModelHelper.
- Wi-Fi handles energy depletion and recharge:
  - Energy depletion callback will move wifi state to OFF;
  - Energy recharged callback move wifi state to IDLE and restart channel access manager;
  - When state change, compute time the device can stay ON and re-schedule OFF state.
- Model architecture:





Wifi Radio Energy Model (2)

- Usage:

```
/* energy source */
BasicEnergySourceHelper basicSourceHelper;

// configure energy source
basicSourceHelper.Set ("BasicEnergySourceInitialEnergyJ", DoubleValue (0.1));

// install source
EnergySourceContainer sources = basicSourceHelper.Install (nodes);

/* device energy model */
WifiRadioEnergyModelHelper radioEnergyHelper;

// configure radio energy model
radioEnergyHelper.Set ("TxCurrentA", DoubleValue (0.0174));

// install device model
DeviceEnergyModelContainer deviceModels;
deviceModels = radioEnergyHelper.Install (devices, sources);
```

Wi-Fi configuration via helpers

- Wi-Fi helpers are available for users to create Wi-Fi devices and channels with only a few lines of code.

- To create a Wi-Fi network, users need to follow these steps:
  - Decide if SpectrumWifiPhy or YansWifiPhy (different helpers).
  - Configure the wireless Channel: add propagation loss model(s) and set propagation delay model.
  - Configure PHY: error rate model and other PHY attributes (channel width, frequency, …) can be set.
  - Configure MAC: architecture (ad-hoc, infrastructure, station or access point), select rate control algorithm, ...
  - Create devices: install MAC and PHY on nodes.
  - Configure mobility: specify initial locations and how nodes will move during simulation.

Wi-Fi configuration via helpers (cont.)

- Different ways to configure some parameters (e.g. channel number):
  - Via global configuration default:
    ```
    Config::SetDefault ("ns3::WifiPhy::ChannelNumber", UintegerValue (3));
    ```
  - Via the helper:
    ```
    yansWifiPhyHelper.Set ("ChannelNumber", UintegerValue (3));
    ```
  - By selecting the standard;
  - By performing post-installation configuration:
    ```
    Config::Set("/NodeList/0/DeviceList/*/$ns3::WifiNetDevice/Phy/$ns3::WifiPhy
    /ChannelNumber", Uinteger (3));
    ```
- But be carefull:
  - Some parameters are NOT INDEPENDANT:
    channel number, frequency, channel width and standard.
  - The SetStandard method set default parameters (mainly timing parameters) as defined in the selected standard, OVERWRITING values that may have been configured. In order to change those parameters, one should use post-installation method.

=> Always make sure you correctly set your settings!

Typical configuration

```
// Create 1 access point and 2 stations
NodeContainer ap, stas;
ap.Create (1);
stas.Create (2);

// Select 802.11b standard (802.11a is default)
WifiHelper wifi; //used to install 802.11 PHY and MAC on the nodes
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

// Setup PHY layer (YANS with default NIST error rate model)
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
// If one wants to use YANS:
//wifiPhy.SetErrorRateModel ("ns3::YansErrorRateModel");
// ns-3 supports RadioTap and Prism tracing extensions for 802.11
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel; //no call to default here, since we set
propagation delay and propagation loss models ourselves

// reference loss must be changed since 802.11b is operating at 2.4GHz!
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel",
                                "ReferenceLoss", DoubleValue (40.0459));

wifiPhy.SetChannel (wifiChannel.Create ());
```

Typical configuration (cont.)

```
// Setup a non-QoS MAC layer, with rate control disabled (11 Mbit/s
constant rate for data frames, 1 Mbit/s constant rate for RTS frames)
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode", StringValue (“DsssRate11Mbps”),
                              "ControlMode",StringValue
(“DsssRate1Mbps”));

NetDeviceContainer devices;
// Setup access point
WifiMacHelper wifiMac;
wifiMac.SetType ("ns3::ApWifiMac",
                 "Ssid", SsidValue (“wifi-ns-3”));
NetDeviceContainer apDevice = wifi.Install (wifiPhy, wifiMac, ap);
devices.Add (apDevice);

// Setup stations
wifiMac.SetType ("ns3::StaWifiMac",
                 "Ssid", SsidValue (“wifi-ns-3”));
NetDeviceContainer staDevices = wifi.Install (wifiPhy, wifiMac, stas);
devices.Add (staDevices);
```

Typical configuration (cont.)

```
// Configure mobility
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc =
CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (0.0, 0.0, 0.0)); //AP
positionAlloc->Add (Vector (5.0, 0.0, 0.0)); //STA 1
positionAlloc->Add (Vector (0.0, 5.0, 0.0)); //STA 2
mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (ap);
mobility.Install (sta);

// other set up (e.g. Internet stack, Application, …)
```

Logging

- Enable all debug logging:
  WifiHelper wifiHelper;
  wifiHelper.EnableLogComponents ();

- Enable a subset of debug logging:
  - *LogComponentEnable ("MacLow", LOG_LEVEL_ALL);*
  - *NS_LOG="MacLow" ./waf --run third*

```
+2.032165053s 0 [mac=00:00:00:00:00:0a] MacLow:ForwardDown(): [DEBUG] send DATA,
to=00:00:00:00:00:09, size=1088, mode=OfdmRate6Mbps, preamble=0, duration=+60000.0ns, seq=0x1a0
+2.033641079s 5 [mac=00:00:00:00:00:07] MacLow:DeaggregateAmpduAndReceive(0x2052310)
+2.033641079s 5 [mac=00:00:00:00:00:07] MacLow:ReceiveOk(0x2052310, 0x20d1b40, 4500.46,
OfdmRate6Mbps, 0)
+2.033641079s 5 [mac=00:00:00:00:00:07] MacLow:ReceiveOk(): [DEBUG] duration/id=+60000.0ns
+2.033641079s 5 [mac=00:00:00:00:00:07] MacLow:ReceiveOk(): [DEBUG] rx not for me
from=00:00:00:00:00:0a
+2.033641085s 6 [mac=00:00:00:00:00:08] MacLow:DeaggregateAmpduAndReceive(0x20575b0)
+2.033641085s 6 [mac=00:00:00:00:00:08] MacLow:ReceiveOk(0x20575b0, 0x20f2290, 2265.83,
OfdmRate6Mbps, 0)
+2.033641085s 6 [mac=00:00:00:00:00:08] MacLow:ReceiveOk(): [DEBUG] duration/id=+60000.0ns
+2.033641085s 6 [mac=00:00:00:00:00:08] MacLow:ReceiveOk(): [DEBUG] rx not for me
from=00:00:00:00:00:0a
+2.033641095s 7 [mac=00:00:00:00:00:09] MacLow:DeaggregateAmpduAndReceive(0x205c740)
+2.033641095s 7 [mac=00:00:00:00:00:09] MacLow:ReceiveOk(0x205c740, 0x20cb050, 1000.5, OfdmRate6Mbps,
0)
+2.033641095s 7 [mac=00:00:00:00:00:09] MacLow:ReceiveOk(): [DEBUG] duration/id=+60000.0ns
+2.033641095s 7 [mac=00:00:00:00:00:09] MacLow:ReceiveOk(): [DEBUG] rx unicast/sendAck
from=00:00:00:00:00:0a
```

Tracing

- Hook to existing wifi trace sources:
  - Example: trace TX PHY events (from wifi-ap.cc)

```
void
PhyTxTrace (std::string context, Ptr<const Packet> packet, WifiMode
mode, WifiPreamble preamble, uint8_t txPower)
{
  std::cout << "PHYTX mode=" << mode << " " << *packet << std::endl;
}
…
Config::Connect ("/NodeList/*/DeviceList/*/Phy/State/Tx",
MakeCallback (&PhyTxTrace));
```

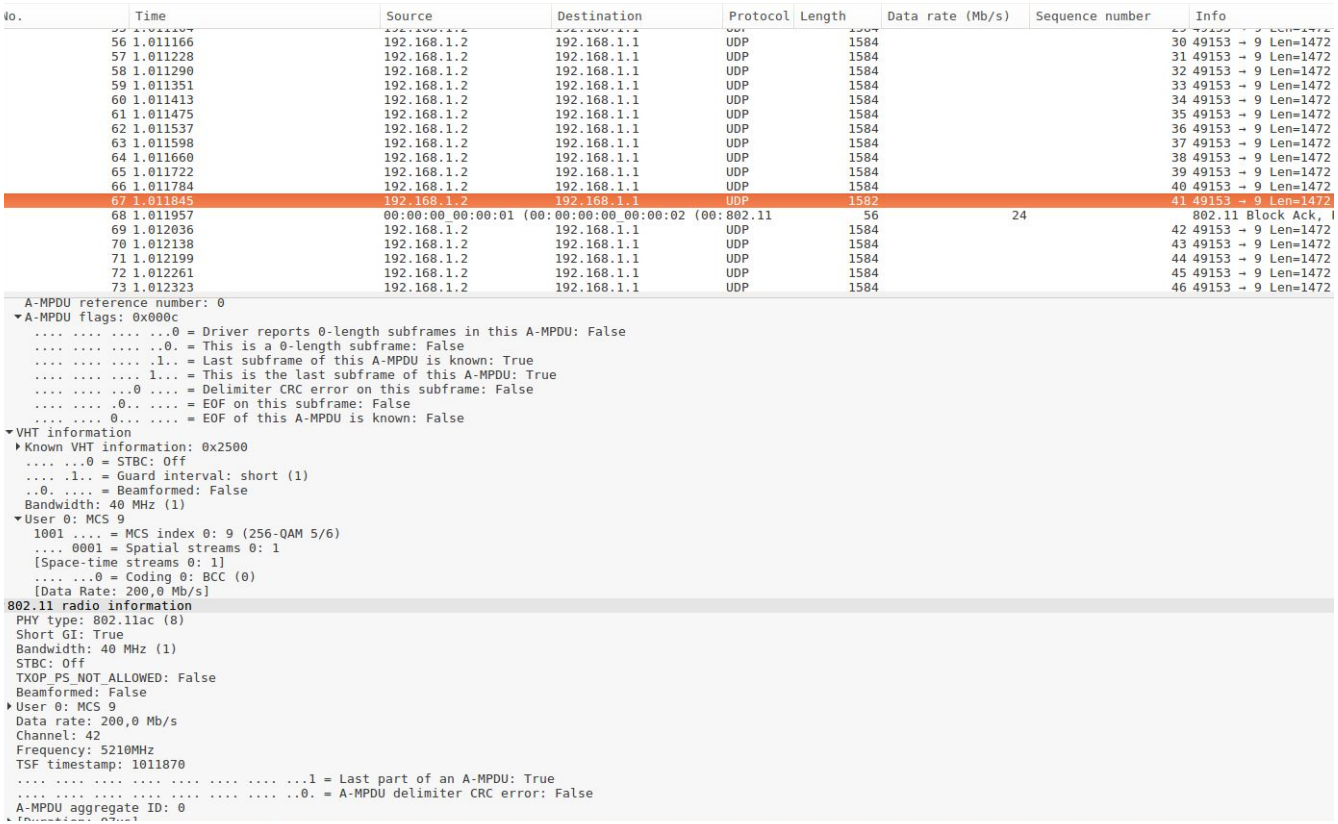
  - Terminal output:

```
…
PHYTX mode=OfdmRate36Mbps ns3::WifiMacHeader (DATA ToDS=0, FromDS=1,
MoreFrag=0, Retry=1, MoreData=0 Duration/ID=44us,
DA=00:00:00:00:00:02, SA=00:00:00:00:00:01, BSSID=00:00:00:00:00:03,
FragNumber=0, SeqNumber=1000) ns3::LlcSnapHeader (type 0x1) Payload
(size=512) ns3::WifiMacTrailer ()
…
```

PCAP

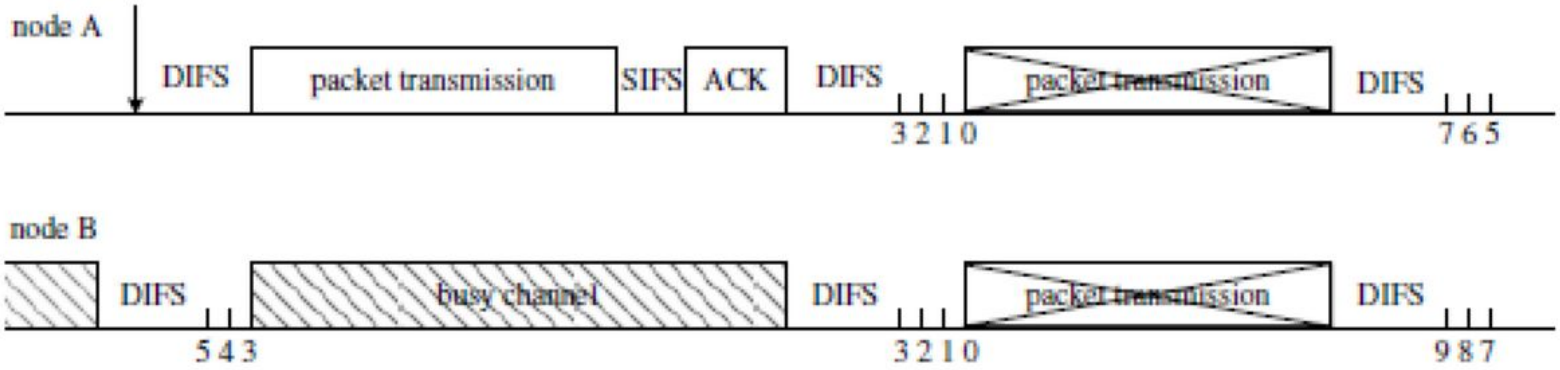
- Enable PCAP file(s) generation during simulation run:

```
YansWifiPhyHelper phy;
phy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
wifiPhy.EnablePcap ("AccessPoint", apDevice);
```



DCF example

- **802.11 DCF illustration** (without RTS/CTS)**:**

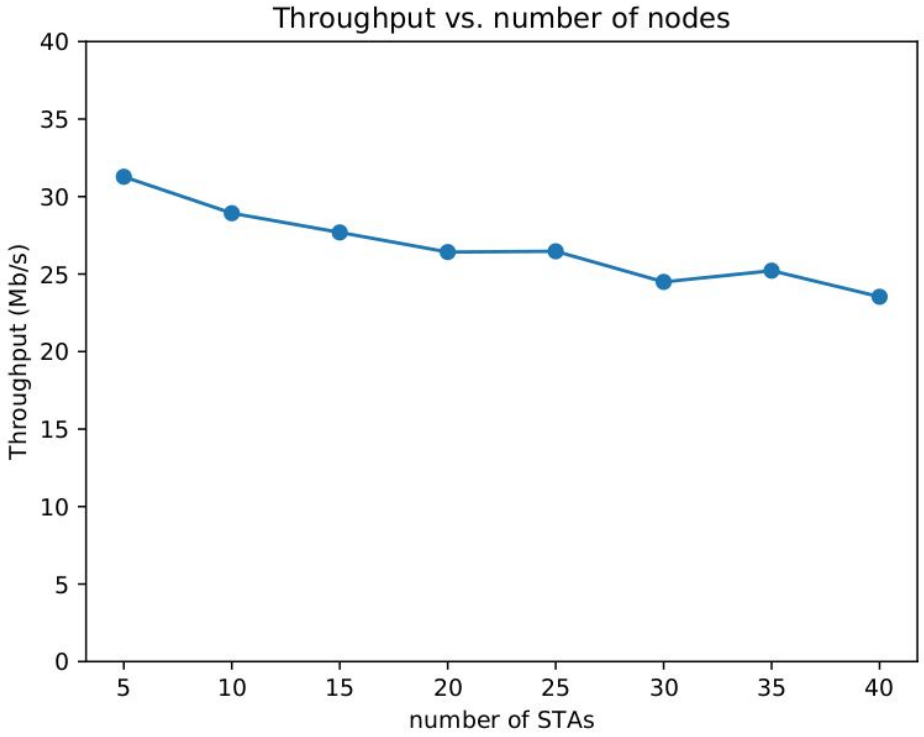


- **wifi-dcf.cc** program can be used to test DCF implementation and has the following main parameters to control the simulation:
  - numStas: number of contending stations;
  - packetArrivalRate: packet arrival rate per second;
  - cwMin: CWmin parameter of DCF;
  - cwMax: CWmax parameter of DCF;
  - rtsThreshold: used to enable/disable RTS/CTS;
  - radius: control distance between AP and STAs (m).

DCF example (cont.)

- **DCF experiment: effect of contention on total throughput**

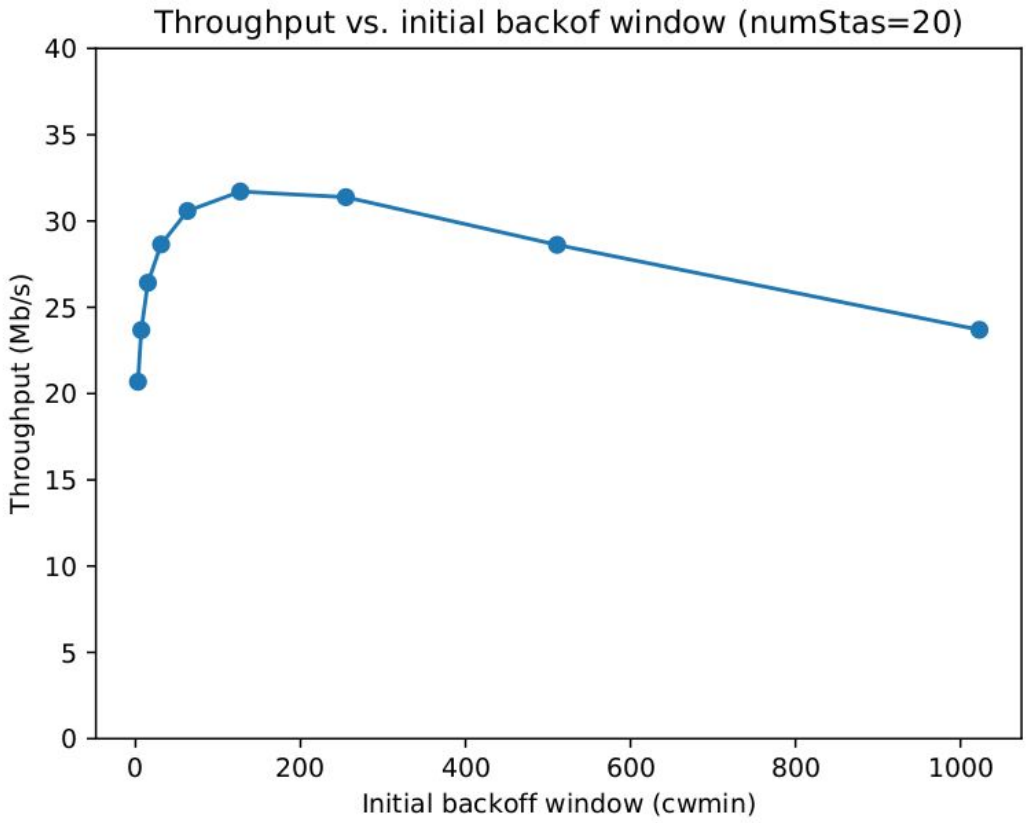
```
./waf --run 'wifi-dcf --numStas=5 --packetArrivalRate=800'
Throughput observed at AP: 31.277 Mb/s
…
./waf --run 'wifi-dcf --numStas=40 --packetArrivalRate=100'
Throughput observed at AP: 23.5342 Mb/s
```



DCF example (cont.)

- **DCF experiment: effect of DCF parameters on total throughput**

```
./waf --run 'wifi-dcf --numStas=20 --packetArrivalRate=200 --cwMin=3 --cwMax=255'
Throughput observed at AP: 20.6842 Mb/s
...
```



DCF example (cont.)

- Differentiate collisions & PHY decoding errors (caused by a too low SNR):
  - Trace RX errors (on **AP** node):

```
Config::Connect ("/NodeList/0/DeviceList/*/Phy/State/RxError",
MakeCallback (&PhyRxErrorTrace));
```

    ⇒ #RX errors = #collisions + PHY decoding errors due to low SNR

  - Trace TX timestamps (on **all nodes**):

```
Config::Connect ("/NodeList/*/DeviceList/*/Phy/State/Tx",
MakeCallback (&PhyTxTrace));
```

    ⇒ #collisions => nodes with similar TX timestamps

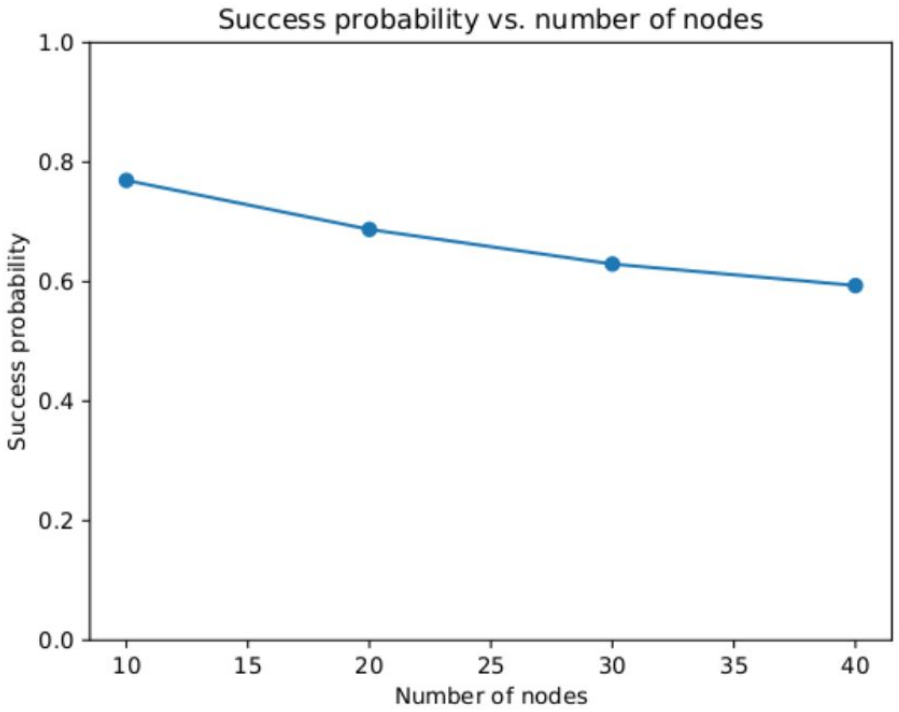
- Traces showing collision:

```
+1.060932798s 0 WifiPhy:EndReceive(0x12fa5d0, 0x12d3540, 0x12ee950)
+1.060932798s 0 WifiPhy:EndReceive(): [DEBUG] mode=54000000, snr(dB)=38.5939, per=1, size=1936
+1.060932798s 0 [mac=00:00:00:00:00:15] MacLow:ReceiveError(0x12f5f20, 0x12d3540, 7234.19)
+1.060932798s 0 [mac=00:00:00:00:00:15] MacLow:ReceiveError(): [DEBUG] rx failed

+1.061007741s 19 [mac=00:00:00:00:00:13] MacLow:NormalAckTimeout(0x12ebc40)
+1.061007741s 19 [mac=00:00:00:00:00:13] MacLow:NormalAckTimeout(): [DEBUG] normal ack timeout
+1.061007775s 4 [mac=00:00:00:00:00:04] MacLow:NormalAckTimeout(0x1207cb0)
+1.061007775s 4 [mac=00:00:00:00:00:04] MacLow:NormalAckTimeout(): [DEBUG] normal ack timeout
```

DCF example (cont.)

- **DCF experiment: effect of contention on average success probability**
  - Default standard CWmin/CWmax, various number of nodes
  - #success transmission ⇒ RxOk trace
  - #failed transmissions (no interference, so only collisions): ⇒ RxError trace

Node	successes	failures	success prob.
0	19128	0	1.000000
1	2091	1236	0.628494
2	1958	1152	0.629582
3	1983	1178	0.627333
4	1867	1120	0.625042
5	2086	1169	0.640860
6	1943	1170	0.624157
7	1872	1147	0.620073
8	1795	1085	0.623264
9	1994	1170	0.630215
10	1661	1067	0.60887



QoS configuration (EDCA)

- Select QoS or non-Qos for 802.11a/b/g, QoS is always enabled for 802.11n/ac/ax.

```
wifiMacHelper wifiMac;
wifiMac.SetType ("ns3::StaWifiMac",
                 "Ssid", SsidValue (“wifi-ns-3”),
                 ”QosSupported", BooleanValue (true));
NetDeviceContainer staDevices = wifi.Install (wifiPhy, wifiMac,
stas);
```

- Example programs:
  - examples/wireless/80211e-txop.cc
  - examples/wireless/wifi-blockack.cc
  - examples/wireless/wifi-multi-tos.cc

QoS examples

- TXOP example:
  - 4 networks with different TXOP limit settings

Throughput for AC_BE with default TXOP limit (0ms): 28.5968 Mbit/s
Throughput for AC_BE with non-default TXOP limit (3.008ms): 36.389 Mbit/s
Throughput for AC_VI with default TXOP limit (3.008ms): 36.9425 Mbit/s
Throughput for AC_VI with non-default TXOP limit (0ms): 32.1756 Mbit/s

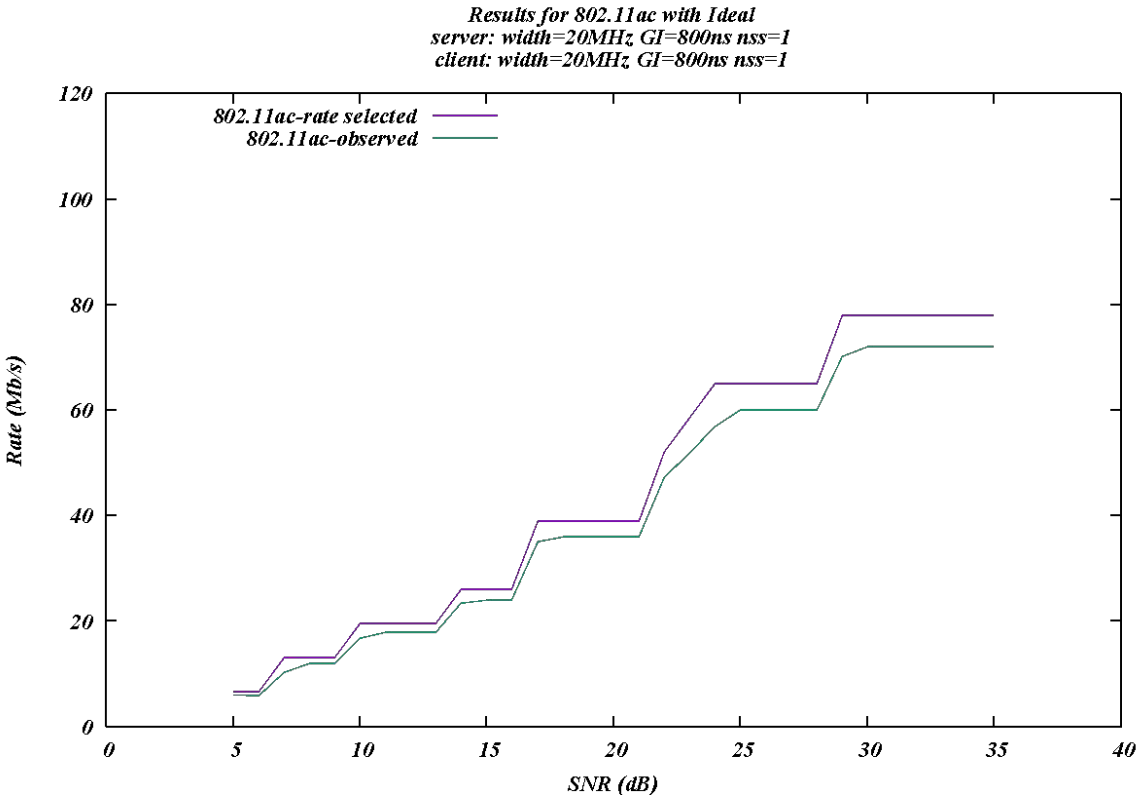
- Multi-TOS example:
  - BSS made of 4 stations
  - each STA has 4 applications with different TOS (type of service, set in IP header) sending saturated traffic to the access point.

Aggregated throughput for AC_BE: 8.53053 Mbit/s
Aggregated throughput for AC_BK: 6.445 Mbit/s
Aggregated throughput for AC_VI: 9.18999 Mbit/s
Aggregated throughput for AC_VO: 9.91539 Mbit/s

Rate managers example

- Example script to test any rate control algorithms:
  src/wifi/examples/wifi-manager-example.cc

./waf --run 'wifi-manager-example --wifiManager=Ideal --standard=802.11ac --serverChannelWidth=20 --clientChannelWidth=20 --serverShortGuardInterval=800 --clientShortGuardInterval=800 --serverNss=1 --clientNss=1'



Configuring 802.11n/ac/ax

- Example programs:
  - examples/wireless/ht-wifi-network.cc
  - examples/wireless/vht-wifi-network.cc
  - examples/wireless/he-wifi-network.cc

- Setting the WifiPhyStandard will set most defaults reasonably (e.g. 802.11ac):

```
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211ac);
```

- Specific configurations to be attached:
  HtConfiguration, VhtConfiguration and HeConfiguration.

Configuring 802.11n/ac/ax (cont.)

- 11n/ac/ax configuration objects:
  - Created once the ns3::WifiHelper::Install has been called (depending on the selected standard).
  - The configuration objects are used to store and manage specific attributes.

- **HtConfiguration:**
  - 802.11n specific configuration: guard interval, …
- **VhtConfiguration:**
  - 802.11ac specific configuration: none (currently)
- **HeConfiguration:**
  - 802.11ax specific configuration: guard interval duration, BSS color, …

802.11ax example

- Example: 802.11ax simulation (he-wifi-network.cc)
  - BSS made of 1 STA;
  - Constant rate (MCS 0 - 11);
  - Various channel bonding and GI settings.

MCS value	Channel width	GI	Throughput
0	20 MHz	3200 ns	6.35904 Mbit/s
0	20 MHz	1600 ns	7.0656 Mbit/s
0	20 MHz	800 ns	7.53664 Mbit/s
0	40 MHz	3200 ns	12.8123 Mbit/s
0	40 MHz	1600 ns	14.508 Mbit/s
0	40 MHz	800 ns	15.3559 Mbit/s
0	80 MHz	3200 ns	27.6029 Mbit/s
0	80 MHz	1600 ns	30.2879 Mbit/s
0	80 MHz	800 ns	31.6539 Mbit/s
0	160 MHz	3200 ns	54.1225 Mbit/s
0	160 MHz	1600 ns	59.351 Mbit/s
0	160 MHz	800 ns	63.9201 Mbit/s
...			

Setting frame aggregation

- Configure aggregation for AC_BE queue:

```
WifiMacHelper mac;
mac.SetType ("ns3::StaWifiMac",
             "Ssid", SsidValue (ssid),
             "BE_MaxAmpduSize", UintegerValue(32768),
             "BE_MaxAmsduSize", UintegerValue (3839));
```

- Example program:
  - wifi-aggregation.cc
  - 4 networks with different aggregation settings

Throughput with default configuration (A-MPDU aggregation enabled, 65kB): 59.5465 Mbit/s
Throughput with aggregation disabled: 30.3585 Mbit/s
Throughput with A-MPDU disabled and A-MSDU enabled (8kB): 51.5259 Mbit/s
Throughput with A-MPDU enabled (32kB) and A-MSDU enabled (4kB): 58.9624 Mbit/s

MIMO configurations

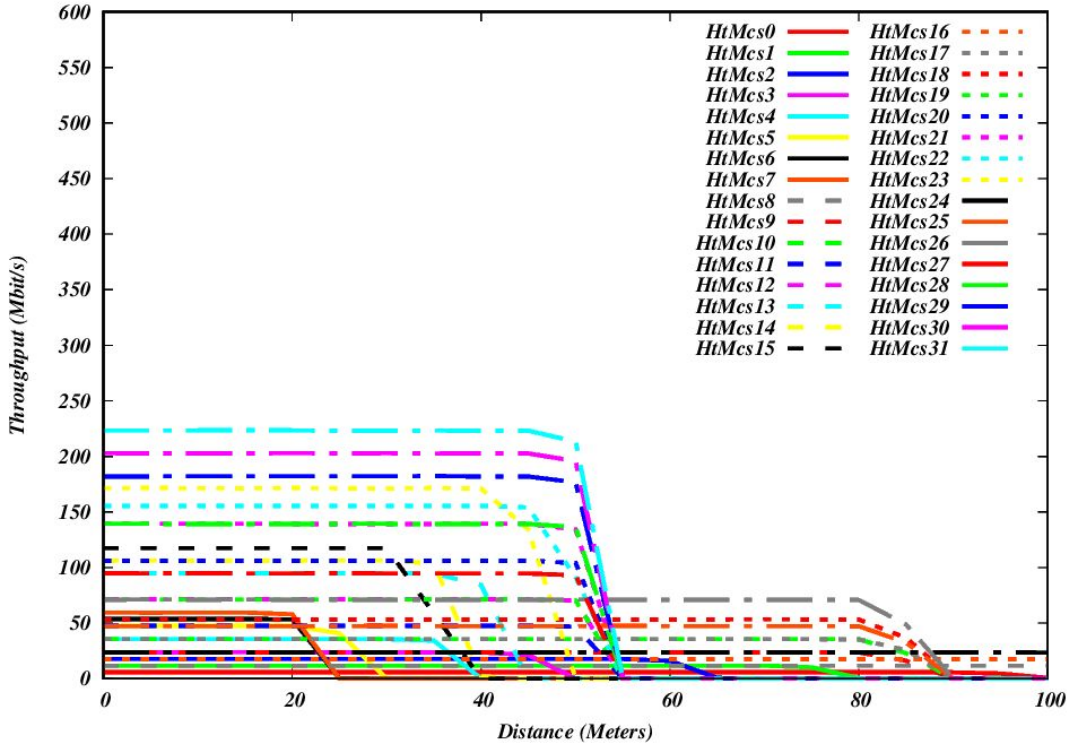
- Relevant attributes (WifiPhy):
  - Antennas: specify the number of physical antennas;
  - MaxSupportedTxSpatialStreams: maximum number of spatial streams that can be transmitted by the device;
  - MaxSupportedRxSpatialStreams: maximum number of spatial streams that can be received by the device.
- Different configurations possible:
  - Symmetrical: 2 x 2 : 2
  - Asymmetrical with more antennas at TX side: 3 x 2 : 2
  - Asymmetrical with more antennas at RX side: 2 x 3 : 2
  - Limit number of streams: 3 x 3 : 2
- If more antennas the number of streams:
  => wireless link more robust (approximation: + 3dB SNR gain per additional antenna)

MIMO configurations (cont.)

- Usage:

```
YansWifiPhyHelper phy;
phy.Set ("Antennas", UintegerValue (3));
phy.Set ("MaxSupportedTxSpatialStreams", UintegerValue (2));
phy.Set ("MaxSupportedRxSpatialStreams", UintegerValue (3));
```

- Example program: 80211n-mimo.cc



Use of helpers for Spectrum model

```
…

// Create Spectrum channel and PHY
Ptr<MultiModelSpectrumChannel> channel =
CreateObject<MultiModelSpectrumChannel> ();
Ptr<FriisPropagationLossModel> lossModel =
CreateObject<FriisPropagationLossModel> ();
channel->AddPropagationLossModel (lossModel);
Ptr<ConstantSpeedPropagationDelayModel> delayModel =
CreateObject<ConstantSpeedPropagationDelayModel> ();
channel->SetPropagationDelayModel (delayModel);

SpectrumWifiPhyHelper phy = SpectrumWifiPhyHelper::Default ();
phy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
phy.SetChannel (channel);

…
```

Spectrum for non-wifi interference

- Wifi Spectrum model can be used to co-exist with non-wifi signals.
  - Example program: wifi-spectrum-per-example
  - non-Wifi interferer: control its power spread over the wifi 20 MHz band

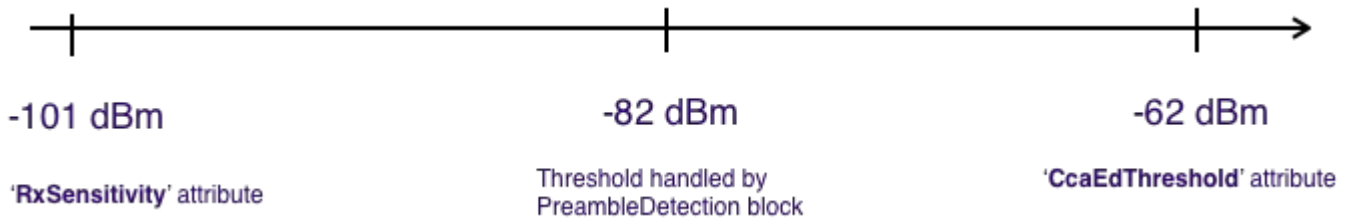
```
./waf --run 'wifi-spectrum-per-interference --simulationTime=1 --waveformPower=  0'
wifiType: ns3::SpectrumWifiPhy distance: 50m; time: 1; TxPower: 16 dBm (40 mW)
index   MCS  Rate (Mb/s) Tput (Mb/s) Received Signal (dBm)Noi+Inf(dBm) SNR (dB)
    0     0       6.50        5.72     735      -64.72       -93.97       29.24
    1     1      13.00       11.49    1478      -64.72       -93.97       29.24
    2     2      19.50       17.30    2225      -64.72       -93.97       29.24
    3     3      26.00       23.09    2970      -64.72       -93.97       29.24
    4     4      39.00       34.79    4474      -64.72       -93.97       29.24
    5     5      52.00       46.41    5968      -64.72       -93.97       29.24
…

./waf --run 'wifi-spectrum-per-interference --simulationTime=1 --waveformPower=  0.0001'
wifiType: ns3::SpectrumWifiPhy distance: 50m; time: 1; TxPower: 16 dBm (40 mW)
index   MCS  Rate (Mb/s) Tput (Mb/s) Received Signal (dBm)Noi+Inf(dBm) SNR (dB)
    0     0       6.50        5.72     735      -64.72       -80.08       15.35
    1     1      13.00       11.49    1478      -64.72       -80.08       15.35
    2     2      19.50       17.30    2225      -64.72       -80.08       15.35
    3     3      26.00       23.09    2970      -64.72       -80.08       15.35
    4     4      39.00        0.19      25      -64.72       -80.08       15.35
    5     5      52.00        0.00       0         N/A          N/A         N/A
…
```

Q&A

Backup slides

PHY thresholds

- **RX sensitivity:** (Default: -101 dBm)
  - Filter out all signals with received power lower than this threshold;
  - Lower signals are not tracked by InterferenceHelper;
  - Used to speed up simulations;
- **CCA preamble detection (PD) threshold:**
  - Modelled by the preamble detection model (see next slide).
- **CCA Energy Detection (ED) threshold**: (Default: -62 dBm)
  - Used for received non-wifi signals, or wifi signals that did not pass preamble detection;
  - If the total energy (i.e. sum of all signals tracked by InterferenceHelper) is above this threshold, PHY state is declared as CCA_BUSY.

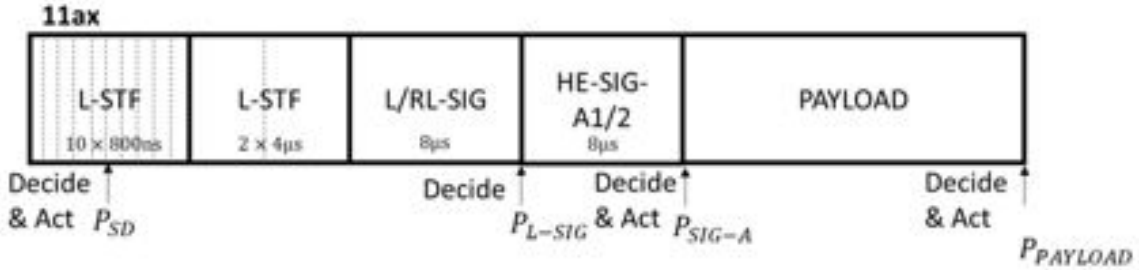


CCA Preamble Detection model

- Model to determine whether a received preamble can be detected successfully, based on its RSSI, its SNR and the channel width.
- Currently only one model (enabled by default): ThresholdPreambleDetectionModel
- ThresholdPreambleDetectionModel::IsPreambleDetected returns true if:
  - the RSSI is above a target RSSI (default: -82 dBm);
  - the SNR is above a given threshold (default: 4 dB).
- Custom CCA-PD model can be used by implementing a new class that inherits from PreambleDetectionModel and implements IsPreambleDetected method (which is called by the PHY).

PPDU reception

- Multi-stage reception model:
  - CCA-SD is run 4 μs after a reception event has been received, to determine whether the preamble is successfully detected and whether the PHY state should move to RX.
  - After the L-SIG duration, the PHY checks whether the L-SIG can be successfully received. If L-SIG fails, the PHY goes back to IDLE (or CCA_BUSY if energy above CCA-ED).
  - For non-legacy signals, it checks whether SIG-A is successfully received. Only when the L-SIG and SIG-A are received successfully, the receiver proceeds with the payload.
  - After the PPDU duration, it checks whether the payload is successfully received and whether it should be forwarded up to MacLow.
  - CCA-ED runs continuously to determine whether the PHY state should move to CCA_BUSY.



- Even if a PPDU is dropped during the reception process, it is tracked by the InterferenceHelper object (unless its power is below RxSensitivity threshold) for SINR computations and making CCA decisions.

Frame capture model

- Model to determine whether reception should switch a newly incoming PPDU while PHY is already in RX state.
- Currently only one model (disabled by default): SimpleFrameCaptureModel
- SimpleFrameCaptureModel::CaptureNewFrame returns true if:
  - the RSSI difference between the current PPDU and the new PPDU is above a configured margin (default: 5 dB);
- Custom frame capture model can be used by implementing a new class that inherits from FrameCaptureModel and implements CaptureNewFrame method (which is called by the PHY).
- Important: even if no frame capture model is attached to the PHY, reception moves to the reception of any incoming PPDU with a stronger power during preamble detection.

802.11ax reception: spatial reuse

- There is an additional stage in the reception process when incoming signal is an HE PPDU.
- Event at the end of HE-SIG (before payload starts): read HE-SIG header from the PPDU (BSS color info) and trigger OBSS_PD spatial-reuse algorithm (currently: ConstantObssPdAlgorithm).
- ConstantObssPdAlgorithm:
  - Once a HE-SIG has been received, algorithm checks whether this is an OBSS frame by comparing its BSS color with the BSS color of the received preamble.
  - If this is an OBSS frame, it compares the received RSSI with its configured OBSS_PD threshold value.
  - Resets PHY to IDLE state if RSSI is lower than that constant OBSS_PD threshold value, and is informed about a TX power restrictions to be applied.
- Custom OBSS_PD spatial-reuse algorithms can easily be designed.