The ns-3 Wi-Fi Module Documentation

Release ns-3-dev

ns-3 project

January 12, 2016

CHAPTER

ONE

DESIGN DOCUMENTATION

ns-3 nodes can contain a collection of NetDevice objects, much like an actual computer contains separate interface cards for Ethernet, Wifi, Bluetooth, etc. This chapter describes the *ns-3* WifiNetDevice and related models. By adding WifiNetDevice objects to *ns-3* nodes, one can create models of 802.11-based infrastructure and ad hoc networks.

1.1 Overview of the model

The WifiNetDevice models a wireless network interface controller based on the IEEE 802.11 standard [ieee80211]. We will go into more detail below but in brief, *ns-3* provides models for these aspects of 802.11:

- basic 802.11 DCF with **infrastructure** and **adhoc** modes
- **802.11a**, **802.11b**, **802.11g**, **802.11n** (both 2.4 and 5 GHz bands) and **802.11ac** physical layers
- **MSDU aggregation** and **MPDU aggregation** extensions of 802.11n, and both can be combined together (two-level aggregation)
- QoS-based EDCA and queueing extensions of **802.11e**
- the ability to use different propagation loss models and propagation delay models, please see the chapter on *Propagation* for more detail
- various rate control algorithms including **Aarf, Arf, Cara, Onoe, Rraa, ConstantRate, and Minstrel**
- 802.11s (mesh), described in another chapter
- 802.11p and WAVE (vehicular), described in another chapter

The set of 802.11 models provided in *ns-3* attempts to provide an accurate MAC-level implementation of the 802.11 specification and to provide a packet-level abstraction of the PHY-level for different PHYs, corresponding to 802.11a/b/e/g/n/ac specifications.

In *ns-3*, nodes can have multiple WifiNetDevices on separate channels, and the WifiNetDevice can coexist with other device types; this removes an architectural limitation found in *ns-2*. Presently, however, there is no model for cross-channel interference or coupling between channels.

The source code for the WifiNetDevice and its models lives in the directory `src/wifi`.

The implementation is modular and provides roughly three sublayers of models:

- the **PHY layer models**
- the so-called **MAC low models**: they model functions such as medium access (DCF and EDCA), RTS/CTS and ACK. In *ns-3*, the lower-level MAC is further subdivided into a **MAC low** and **MAC middle** sublayering, with channel access grouped into the **MAC middle**.

- the so-called **MAC high models**: they implement non-time-critical processes in Wifi such as the MAC-level beacon generation, probing, and association state machines, and a set of **Rate control algorithms**. In the literature, this sublayer is sometimes called the **upper MAC** and consists of more software-oriented implementations vs. time-critical hardware implementations.

Next, we provide an design overview of each layer, shown in Figure *WifiNetDevice architecture.*.

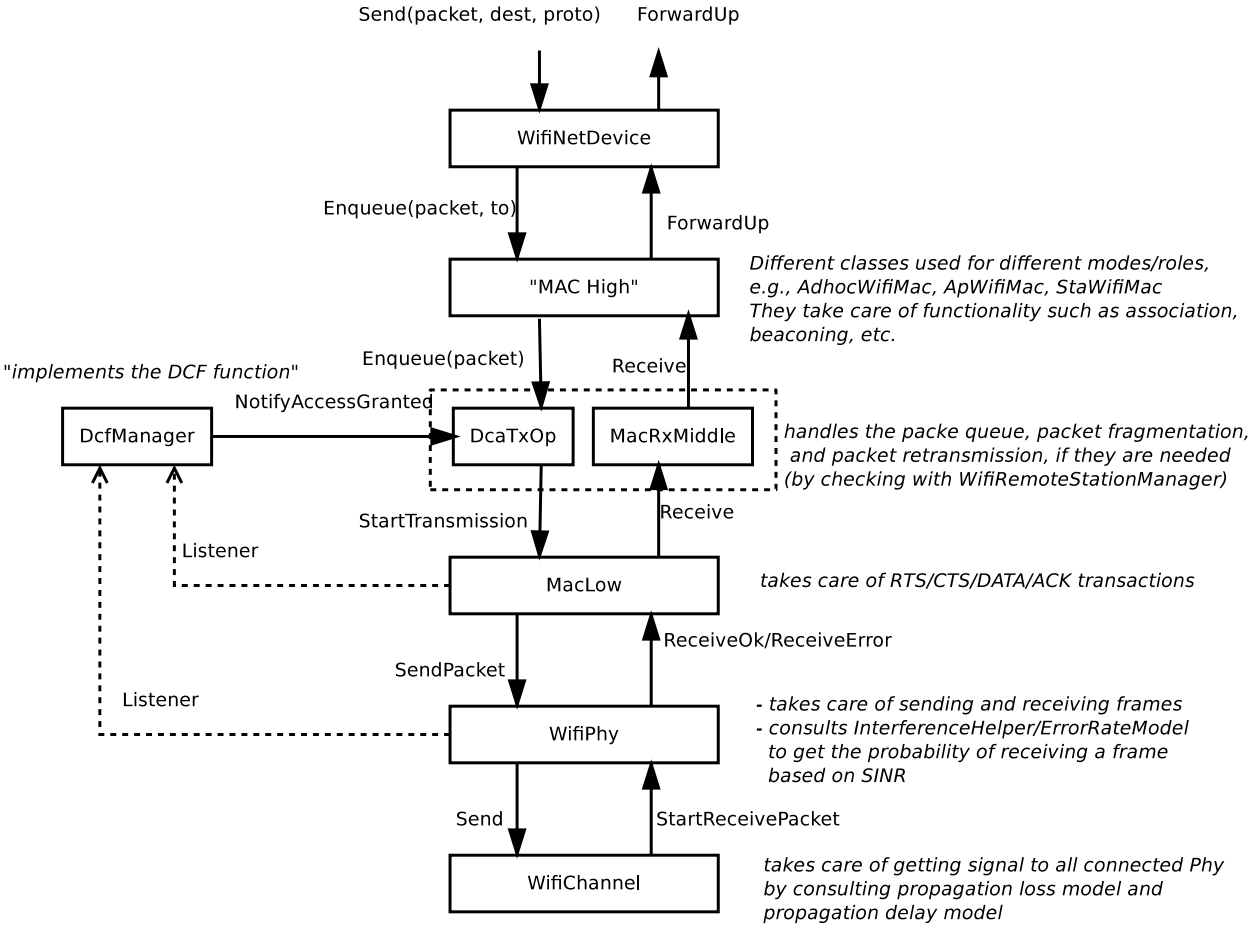

Figure 1.1: WifiNetDevice architecture.

1.1.1 MAC high models

There are presently three **MAC high models** that provide for the three (non-mesh; the mesh equivalent, which is a sibling of these with common parent `ns3::RegularWifiMac`, is not discussed here) Wi-Fi topological elements - Access Point (AP) (`ns3::ApWifiMac`), non-AP Station (STA) (`ns3::StaWifiMac`), and STA in an Independent Basic Service Set (IBSS - also commonly referred to as an ad hoc network (`ns3::AdhocWifiMac`).

The simplest of these is `ns3::AdhocWifiMac`, which implements a Wi-Fi MAC that does not perform any kind of beacon generation, probing, or association. The `ns3::StaWifiMac` class implements an active probing and association state machine that handles automatic re-association whenever too many beacons are missed. Finally, `ns3::ApWifiMac` implements an AP that generates periodic beacons, and that accepts every attempt to associate.

These three MAC high models share a common parent in `ns3::RegularWifiMac`, which exposes, among other MAC configuration, an attribute `QosSupported` that allows configuration of 802.11e/WMM-style QoS sup-

port, an attribute `HtSupported` that allows configuration of 802.11n High Throughput style support an attribute `VhtSupported` that allows configuration of 802.11ac Very High Throughput style support.

There are also several **rate control algorithms** that can be used by the MAC low layer. A complete list of available rate control algorithms is provided in a separate section.

1.1.2 MAC low layer

The **MAC low layer** is split into three main components:

1. `ns3::MacLow` which takes care of RTS/CTS/DATA/ACK transactions.
2. `ns3::DcfManager` and `ns3::DcfState` which implements the DCF and EDCAF functions.
3. `ns3::DcaTxop` and `ns3::EdcaTxopN` which handle the packet queue, packet fragmentation, and packet retransmissions if they are needed. The `ns3::DcaTxop` object is used high MACs that are not QoS-enabled, and for transmission of frames (e.g., of type Management) that the standard says should access the medium using the DCF. `ns3::EdcaTxopN` is is used by QoS-enabled high MACs and also performs 802.11n-style MSDU aggregation.

1.1.3 PHY layer models

The PHY layer implements a single model in the `ns3::WifiPhy` class: the physical layer model implemented there is described in a paper entitled Yet Another Network Simulator The acronym *Yans* derives from this paper title.

In short, the physical layer models are mainly responsible for modeling the reception of packets and for tracking energy consumption. There are typically three main components to this:

- each packet received is probabilistically evaluated for successful or failed reception. The probability depends on the modulation, on the signal to noise (and interference) ratio for the packet, and on the state of the physical layer (e.g. reception is not possible while transmission or sleeping is taking place);
- an object exists to track (bookkeeping) all received signals so that the correct interference power for each packet can be computed when a reception decision has to be made; and
- one or more error models corresponding to the modulation and standard are used to look up probability of successful reception.

1.2 Scope and Limitations

The IEEE 802.11 standard [ieee80211] is a large specification, and not all aspects are covered by *ns-3*; the documentation of *ns-3*'s conformance by itself would lead to a very long document. This section attempts to summarize compliance with the standard and with behavior found in practice.

The physical layer and channel models operate on a per-packet basis, with no frequency-selective propagation or interference effects. Detailed link simulations are not performed, nor are frequency-selective fading or interference models available. Directional antennas and MIMO are also not supported at this time. For additive white gaussian noise (AWGN) scenarios, or wideband interference scenarios, performance is governed by the application of analytical models (based on modulation and factors such as channel width) to the received signal-to-noise ratio, where noise combines the effect of thermal noise and of interference from other Wi-Fi packets. Moreover, interference from other technologies is not modeled. The following details pertain to the physical layer and channel models:

- 802.11n MIMO is not supported
- 802.11n/ac MIMO is not supported
- 802.11n/ac beamforming is not supported

- PLCP preamble reception is not modeled
- PHY_RXSTART is not supported

At the MAC layer, most of the main functions found in deployed Wi-Fi equipment for 802.11a/b/e/g are implemented, but there are scattered instances where some limitations in the models exist. Most notably, 802.11n/ac configurations are not supported by adaptive rate controls; only the so-called `ConstantRateWifiManager` can be used by those standards at this time. Support for 802.11n and ac is evolving. Some additional details are as follows:

- 802.11g does not support 9 microseconds slot
- 802.11e TXOP is not supported
- BSSBasicRateSet for 802.11b has been assumed to be 1-2 Mbit/s
- BSSBasicRateSet for 802.11a/g has been assumed to be 6-12-24 Mbit/s
- cases where RTS/CTS and ACK are transmitted using HT formats are not supported

1.3 Design Details

The remainder of this section is devoted to more in-depth design descriptions of some of the Wi-Fi models. Users interested in skipping to the section on usage of the wifi module (User Documentation) may do so at this point. We organize these more detailed sections from the bottom-up, in terms of layering, by describing the channel and PHY models first, followed by the MAC models.

1.3.1 WifiChannel

`ns3::WifiChannel` is an abstract base class that allows different channel implementations to be connected. At present, there is only one such channel (the `ns3::YansWifiChannel`). The class works in tandem with the `ns3::WifiPhy` class; if you want to provide a new physical layer model, you must subclass both `ns3::WifiChannel` and `ns3::WifiPhy`.

The WifiChannel model exists to interconnect WifiPhy objects so that packets sent by one Phy are received by some or all other Phys on the channel.

YansWifiChannel

This is the only channel model presently in the *ns-3* wifi module. The `ns3::YansWifiChannel` implementation uses the propagation loss and delay models provided within the *ns-3 Propagation* module. In particular, a number of propagation models can be added (chained together, if multiple loss models are added) to the channel object, and a propagation delay model also added. Packets sent from a `ns3::YansWifiPhy` object onto the channel with a particular signal power, are copied to all of the other `ns3::YansWifiPhy` objects after the signal power is reduced due to the propagation loss model(s), and after a delay corresponding to transmission (serialization) delay and propagation delay due any channel propagation delay model (typically due to speed-of-light delay between the positions of the devices).

Only objects of `ns3::YansWifiPhy` may be attached to a `ns3::YansWifiChannel`; therefore, objects modeling other (interfering) technologies such as LTE are not allowed. Furthermore, packets from different channels do not interact; if a channel is logically configured for e.g. channels 5 and 6, the packets do not cause adjacent channel interference (even if their channel numbers overlap).

1.3.2 WifiPhy and related models

The `ns3::WifiPhy` is an abstract base class representing the 802.11 physical layer functions. Packets passed to this object (via a `SendPacket()` method are sent over the `WifiChannel` object, and upon reception, the receiving PHY object decides (based on signal power and interference) whether the packet was successful or not. This class also provides a number of callbacks for notifications of physical layer events, exposes a notion of a state machine that can be monitored for MAC-level processes such as carrier sense, and handles sleep/wake models and energy consumption. The `ns3::WifiPhy` hooks to the `ns3::MacLow` object in the WifiNetDevice.

There is currently one implementation of the `WifiPhy`, which is the `ns3::YansWifiPhy`. It works in conjunction with three other objects:

- **WifiPhyStateHelper**: Maintains the PHY state machine
- **InterferenceHelper**: Tracks all packets observed on the channel
- **ErrorModel**: Computes a probability of error for a given SNR

YansWifiPhy and WifiPhyStateHelper

Class `ns3::YansWifiPhy` is responsible for taking packets passed to it from the MAC (the `ns3::MacLow` object) and sending them onto the `ns3::YansWifiChannel` to which it is attached. It is also responsible to receive packets from that channel, and, if reception is deemed to have been successful, to pass them up to the MAC.

Class `ns3::WifiPhyStateHelper` manages the state machine of the PHY layer, and allows other objects to hook as *listeners* to monitor PHY state. The main use of listeners is for the MAC layer to know when the PHY is busy or not (for transmission and collision avoidance).

The PHY layer can be in one of six states:

1. TX: the PHY is currently transmitting a signal on behalf of its associated MAC
2. RX: the PHY is synchronized on a signal and is waiting until it has received its last bit to forward it to the MAC.
3. IDLE: the PHY is not in the TX, RX, or CCA BUSY states.
4. CCA Busy: the PHY is not in TX or RX state but the measured energy is higher than the energy detection threshold.
5. SWITCHING: the PHY is switching channels.
6. SLEEP: the PHY is in a power save mode and cannot send nor receive frames.

Packet reception works as follows. The `YansWifiPhy` attribute CcaMode1Threshold corresponds to what the standard calls the "ED threshold" for CCA Mode 1. In section 16.4.8.5: "CCA Mode 1: Energy above threshold. CCA shall report a busy medium upon detection of any energy above the ED threshold."

There is a "noise ED threshold" in the standard for non-Wi-Fi signals, and this is usually set to 20 dB greater than the "carrier sense ED threshold". However, the model doesn't support this, because there are no 'foreign' signals in the YansWifi model– everything is a Wi-Fi signal.

In the standard, there is also what is called the "minimum modulation and coding rate sensitivity" in section 18.3.10.6 CCA requirements. This is the -82 dBm requirement for 20 MHz channels. This is analogous to the EnergyDetectionThreshold attribute in `YansWifiPhy`. CCA busy state is not raised in this model when this threshold is exceeded but instead RX state is immediately reached, since it is assumed that PLCP sync always succeeds in this model. Even if the PLCP header reception fails, the channel state is still held in RX until YansWifiPhy::EndReceive().

In ns-3, the values of these attributes are set to small default values (-96 dBm for EnergyDetectionThreshold and -99 dBm for CcaMode1Threshold). So, if a signal comes in at > -96 dBm and the state is IDLE or CCA BUSY, this model will lock onto it for the signal duration and raise RX state. If it comes in at <= -96 dBm but >= -99 dBm, it will definitely raise CCA BUSY but not RX state. If it comes in < -99 dBm, it gets added to the interference tracker and,

by itself, it will not raise CCA BUSY, but maybe a later signal will contribute more power so that the threshold of -99 dBm is reached at a later time.

The energy of the signal intended to be received is calculated from the transmission power and adjusted based on the Tx gain of the transmitter, Rx gain of the receiver, and any path loss propagation model in effect.

The packet reception occurs in two stages. First, an event is scheduled for when the PLCP header has been received. PLCP header is often transmitted at a lower modulation rate than is the payload. The portion of the packet corresponding to the PLCP header is evaluated for probability of error based on the observed SNR. The InterferenceHelper object returns a value for "probability of error (PER)" for this header based on the SNR that has been tracked by the InterferenceHelper. The `YansWifiPhy` then draws a random number from a uniform distribution and compares it against the PER and decides success or failure. The process is again repeated after the payload has been received (possibly with a different error model applied for the different modulation). If both the header and payload are successfully received, the packet is passed up to the `MacLow` object.

Even if packet objects received by the PHY are not part of the reception process, they are remembered by the InterferenceHelper object for purposes of SINR computation and making clear channel assessment decisions.

InterferenceHelper

The InterferenceHelper is an object that tracks all incoming packets and calculates probability of error values for packets being received, and also evaluates whether energy on the channel rises above the CCA threshold.

The basic operation of probability of error calculations is shown in Figure *SNIR function over time.*. Packets are represented as bits (not symbols) in the *ns-3* model, and the InterferenceHelper breaks the packet into one or more "chunks" each with a different signal to noise (and interference) ratio (SNIR). Each chunk is separately evaluated by asking for the probability of error for a given number of bits from the error model in use. The InterferenceHelper builds an aggregate "probability of error" value based on these chunks and their duration, and returns this back to the `YansWifiPhy` for a reception decision.

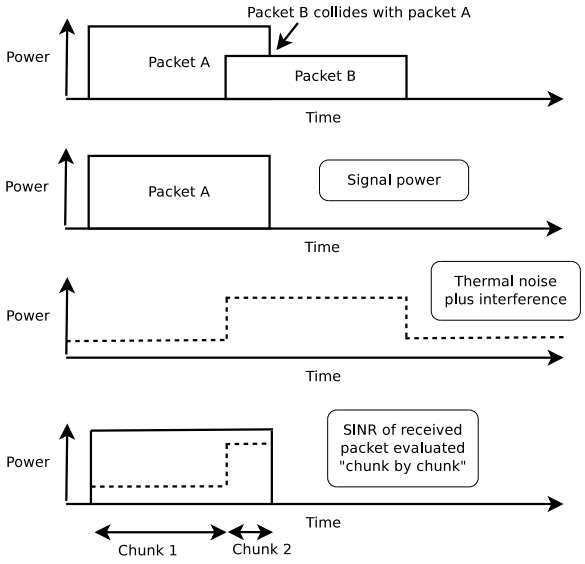

Figure 1.2: *SNIR function over time.*

From the SNIR function we can derive the Bit Error Rate (BER) and Packet Error Rate (PER) for the modulation and coding scheme being used for the transmission.

ErrorModel

The error models are described in more detail in outside references. Please refer to [pei80211ofdm], [pei80211b], [lacage2006yans], [Haccoun] and [Frenger] for a detailed description of the available BER/PER models.

The current *ns-3* error rate models are for additive white gaussian noise channels (AWGN) only; any potential fast fading effects are not modeled.

The original error rate model was called the `ns3::YansErrorRateModel` and was based on analytical results. For 802.11b modulations, the 1 Mbps mode is based on DBPSK. BER is from equation 5.2-69 from [proakis2001]. The 2 Mbps model is based on DQPSK. Equation 8 of [ferrari2004]. More details are provided in [lacage2006yans].

The `ns3::NistErrorRateModel` was later added and became the *ns-3* default. The model was largely aligned with the previous `ns3::YansErrorRateModel` for DSSS modulations 1 Mbps and 2 Mbps, but the 5.5 Mbps and 11 Mbps models were re-based on equations (17) and (18) from [pursley2009]. For OFDM modulations, newer results were obtained based on work previously done at NIST [miller2003]. The results were also compared against the CMU wireless network emulator, and details of the validation are provided in [pei80211ofdm]. Since OFDM modes use hard-decision of punctured codes, the coded BER is calculated using Chernoff bounds.

The 802.11b model was split from the OFDM model when the NIST error rate model was added, into a new model called DsssErrorRateModel. The current behavior is that users may

Furthermore, the 5.5 Mbps and 11 Mbps models for 802.11b rely on library methods implemented in the GNU Scientific Library (GSL). The Waf build system tries to detect whether the host platform has GSL installed; if so, it compiles in the newer models from [pursley2009] for 5.5 Mbps and 11 Mbps; if not, it uses a backup model derived from Matlab simulations.

As a result, there are three error models:

1. `ns3::DsssErrorRateModel`: contains models for 802.11b modes. The 802.11b 1 Mbps and 2 Mbps error models are based on classical modulation analysis. If GNU GSL is installed, the 5.5 Mbps and 11 Mbps from [pursley2009] are used; otherwise, a backup Matlab model is used.
2. `ns3::NistErrorRateModel`: is the default for OFDM modes and reuses

   `ns3::DsssErrorRateModel` for 802.11b modes.

1. `ns3::YansErrorRateModel`: is the legacy for OFDM modes and reuses

   `ns3::DsssErrorRateModel` for 802.11b modes.

Users should select either Nist or Yans models for OFDM (Nist is default), and Dsss will be used in either case for 802.11b.

1.3.3 The MAC model

The 802.11 Distributed Coordination Function is used to calculate when to grant access to the transmission medium. While implementing the DCF would have been particularly easy if we had used a recurring timer that expired every slot, we chose to use the method described in [ji2004sslswn] where the backoff timer duration is lazily calculated whenever needed since it is claimed to have much better performance than the simpler recurring timer solution.

The backoff procedure of DCF is described in section 9.2.5.2 of [ieee80211].

- "The backoff procedure shall be invoked for a STA to transfer a frame when finding the medium busy as indicated by either the physical or virtual CS mechanism."
- "A backoff procedure shall be performed immediately after the end of every transmission with the More Fragments bit set to 0 of an MPDU of type Data, Management, or Control with subtype PS-Poll, even if no additional transmissions are currently queued."

Thus, if the queue is empty, a newly arrived packet should be transmitted immediately after channel is sensed idle for DIFS. If queue is not empty and after a successful MPDU that has no more fragments, a node should also start the backoff timer.

Some users have observed that the 802.11 MAC with an empty queue on an idle channel will transmit the first frame arriving to the model immediately without waiting for DIFS or backoff, and wonder whether this is compliant. According to the standard, "The backoff procedure shall be invoked for a STA to transfer a frame when finding the medium busy as indicated by either the physical or virtual CS mechanism." So in this case, the medium is not found to be busy in recent past and the station can transmit immediately.

The higher-level MAC functions are implemented in a set of other C++ classes and deal with:

- packet fragmentation and defragmentation,
- use of the RTS/CTS protocol,
- rate control algorithm,
- connection and disconnection to and from an Access Point,
- the MAC transmission queue,
- beacon generation,
- MSDU aggregation,
- etc.

Multiple rate control algorithms are available in *ns-3*. Some rate control algorithms are modeled after real algorithms used in real devices; others are found in literature. The following rate control algorithms can be used by the MAC low layer:

Algorithms found in real devices:

- `ArfWifiManager` (default for `WifiHelper`)
- `OnoeWifiManager`
- `ConstantRateWifiManager`
- `MinstrelWifiManager`

Algorithms in literature:

- `IdealWifiManager`
- `AarfWifiManager` [lacage2004aarfamrr]
- `AmrrWifiManager` [lacage2004aarfamrr]
- `CaraWifiManager` [kim2006cara]
- `RraaWifiManager` [wong2006rraa]
- `AarfcdWifiManager` [maguolo2008aarfcd]
- `ParfWifiManager` [akella2007parf]
- `AparfWifiManager` [chevillat2005aparf]

ConstantRateWifiManager

The constant rate control algorithm always uses the same transmission mode for every packet. Users can set a desired 'DataMode' for all 'unicast' packets and 'ControlMode' for all 'request' control packets (e.g. RTS).

To specify different data mode for non-unicast packets, users must set the 'NonUnicastMode' attribute of the WifiRemoteStationManager. Otherwise, WifiRemoteStationManager will use a mode with the lowest rate for non-unicast packets.

The 802.11 standard is quite clear on the rules for selection of transmission parameters for control response frames (e.g. CTS and ACK). *ns-3* follows the standard and selects the rate of control response frames from the set of basic rates or mandatory rates. This means that control response frames may be sent using different rate even though the ConstantRateWifiManager is used. The ControlMode attribute of the ConstantRateWifiManager is used for RTS frames only. The rate of CTS and ACK frames are selected according to the 802.11 standard. However, users can still manually add WifiMode to the basic rate set that will allow control response frames to be sent at other rates. Please consult the project wiki on how to do this.

Available attributes:

- DataMode (default WifiMode::OfdmRate6Mbps): specify a mode for all non-unicast packets
- ControlMode (default WifiMode::OfdmRate6Mbps): specify a mode for all 'request' control packets

IdealWifiManager

The ideal rate control algorithm selects the best mode according to the SNR of the previous packet sent. Consider node *A* sending a unicast packet to node *B*. When *B* successfully receives the packet sent from *A*, *B* records the SNR of the received packet into a `ns3::SnrTag` and adds the tag to an ACK back to *A*. By doing this, *A* is able to learn the SNR of the packet sent to *B* using an out-of-band mechanism (thus the name 'ideal'). *A* then uses the SNR to select a transmission mode based on a set of SNR thresholds, which was built from a target BER and mode-specific SNR/BER curves.

Available attribute:

- BerThreshold (default 10e-6): The maximum Bit Error Rate that is used to calculate the SNR threshold for each mode.

MinstrelWifiManager

The minstrel rate control algorithm is a rate control algorithm originated from madwifi project. It is currently the default rate control algorithm of the Linux kernel.

Minstrel keeps track of the probability of successfully sending a frame of each available rate. Minstrel then calculates the expected throughput by multiplying the probability with the rate. This approach is chosen to make sure that lower rates are not selected in favor of the higher rates (since lower rates are more likely to have higher probability).

In minstrel, roughly 10 percent of transmissions are sent at the so-called lookaround rate. The goal of the lookaround rate is to force minstrel to try higher rate than the currently used rate.

For a more detailed information about minstrel, see [linuxminstrel].

Modifying Wifi model

Modifying the default wifi model is one of the common tasks when performing research. We provide an overview of how to make changes to the default wifi model in this section. Depending on your goal, the common tasks are (in no particular order):

- Creating or modifying the default Wi-Fi frames/headers by making changes to `wifi-mac-header.*`.
- MAC low modification. For example, handling new/modified control frames (think RTS/CTS/ACK/Block ACK), making changes to two-way transaction/four-way transaction. Users usually make changes to `mac-low.*` to accomplish this. Handling of control frames is performed in `MacLow::ReceiveOk`.

- MAC high modification. For example, handling new management frames (think beacon/probe), beacon/probe generation. Users usually make changes to `regular-wifi-mac.*`, `sta-wifi-mac.*`, `ap-wifi-mac.*`, or `adhoc-wifi-mac.*` to accomplish this.
- Wi-Fi queue management. The files `dca-txop.*` and `edca-txop-n.*` are of interested for this task.
- Channel access management. Users should modify the files `dcf-manager.*`, which grant access to `DcaTxop` and `EdcaTxopN`.
- Fragmentation and RTS threholds are handled by Wi-Fi remote station manager. Note that Wi-Fi remote station manager simply indicates if fragmentation and RTS are needed. Fragmentation is handled by `DcaTxop` or `EdcaTxopN` while RTS/CTS transaction is hanled by `MacLow`.
- Modifying or creating new rate control algorithms can be done by creating a new child class of Wi-Fi remote station manager or modifying the existing ones.

CHAPTER

TWO

USER DOCUMENTATION

2.1 Using the WifiNetDevice

The modularity provided by the implementation makes low-level configuration of the WifiNetDevice powerful but complex. For this reason, we provide some helper classes to perform common operations in a simple matter, and leverage the *ns-3* attribute system to allow users to control the parametrization of the underlying models.

Users who use the low-level *ns-3* API and who wish to add a WifiNetDevice to their node must create an instance of a WifiNetDevice, plus a number of constituent objects, and bind them together appropriately (the WifiNetDevice is very modular in this regard, for future extensibility). At the low-level API, this can be done with about 20 lines of code (see `ns3::WifiHelper::Install`, and `ns3::YansWifiPhyHelper::Create`). They also must create, at some point, a WifiChannel, which also contains a number of constituent objects (see `ns3::YansWifiChannelHelper::Create`).

However, a few helpers are available for users to add these devices and channels with only a few lines of code, if they are willing to use defaults, and the helpers provide additional API to allow the passing of attribute values to change default values. Commonly used attribute values are listed in the Attributes section. The scripts in `examples/wireless` can be browsed to see how this is done. Next, we describe the common steps to create a WifiNetDevice from the bottom layer (WifiChannel) up to the device layer (WifiNetDevice).

To create a WifiNetDevice, users need to configure mainly five steps:

- Configure the WifiChannel: WifiChannel takes care of getting signal from one device to other devices on the same wifi channel. The main configurations of WifiChannel are propagation loss model and propagation delay model.
- Configure the WifiPhy: WifiPhy takes care of actually sending and receiving wireless signal from WifiChannel. Here, WifiPhy decides whether each frame will be successfully decoded or not depending on the received signal strength and noise. Thus, the main configuration of WifiPhy is the error rate model, which is the one that actually calculates the probability of successfully decoding the frame based on the signal.
- Configure WifiMac: this step is more on related to the architecture and device level. The users configure the wifi architecture (i.e. ad-hoc or ap-sta) and whether QoS (802.11e), HT (802.11n) and/or VHT (802.11ac) features are supported or not.
- Create WifiDevice: at this step, users configure the desired wifi standard (e.g. **802.11b**, **802.11g**, **802.11a**, **802.11n** or **802.11ac**) and rate control algorithm
- Configure mobility: finally, mobility model is (usually) required before WifiNetDevice can be used.

2.1.1 YansWifiChannelHelper

The YansWifiChannelHelper has an unusual name. Readers may wonder why it is named this way. The reference is to the yans simulator from which this model is taken. The helper can be used to create a WifiChannel with a default

PropagationLoss and PropagationDelay model.

Users will typically type code such as:

```
YansWifiChannelHelper wifiChannelHelper = YansWifiChannelHelper::Default ();
Ptr<WifiChannel> wifiChannel = wifiChannelHelper.Create ();
```

to get the defaults. Specifically, the default is a channel model with a propagation delay equal to a constant, the speed of light (`ns3::ConstantSpeedPropagationDelayModel`), and a propagation loss based on a default log distance model (`ns3::LogDistancePropagationLossModel`)), using a default exponent of 3. Please note that the default log distance model is configured with a reference loss of 46.6777 dB at reference distance of 1m. The reference loss of 46.6777 dB was calculated using Friis propagation loss model at 5.15 GHz. The reference loss must be changed if **802.11b**, **802.11g** or **802.11n** (at 2.4 GHz) are used since they operate at 2.4 Ghz.

Note the distinction above in creating a helper object vs. an actual simulation object. In *ns-3*, helper objects (used at the helper API only) are created on the stack (they could also be created with operator new and later deleted). However, the actual *ns-3* objects typically inherit from `class ns3::Object` and are assigned to a smart pointer. See the chapter in the *ns-3* manual for a discussion of the *ns-3* object model, if you are not familiar with it.

The following two methods are useful when configuring YansWifiChannelHelper:

- `YansWifiChannelHelper::AddPropagationLoss` adds a PropagationLossModel to a chain of PropagationLossModel
- `YansWifiChannelHelper::SetPropagationDelay` sets a PropagationDelayModel

2.1.2 YansWifiPhyHelper

Physical devices (base class `ns3::WifiPhy`) connect to `ns3::WifiChannel` models in *ns-3*. We need to create WifiPhy objects appropriate for the YansWifiChannel; here the `YansWifiPhyHelper` will do the work.

The YansWifiPhyHelper class configures an object factory to create instances of a `YansWifiPhy` and adds some other objects to it, including possibly a supplemental ErrorRateModel and a pointer to a MobilityModel. The user code is typically:

```
YansWifiPhyHelper wifiPhyHelper = YansWifiPhyHelper::Default ();
wifiPhyHelper.SetChannel (wifiChannel);
```

The default YansWifiPhyHelper is configured with NistErrorRateModel (`ns3::NistErrorRateModel`). You can change the error rate model by calling the `YansWifiPhyHelper::SetErrorRateModel` method.

Optionally, if pcap tracing is needed, a user may use the following command to enable pcap tracing:

```
YansWifiPhyHelper::SetPcapDataLinkType (enum SupportedPcapDataLinkTypes dlt)
```

ns-3 supports RadioTap and Prism tracing extensions for 802.11.

Note that we haven't actually created any WifiPhy objects yet; we've just prepared the YansWifiPhyHelper by telling it which channel it is connected to. The Phy objects are created in the next step.

802.11n/ac PHY layer can use either either long (800 ns) or short (400 ns) OFDM guard intervals. To configure this parameter, the following line of code could be used (in this example, it enables the support of a short guard interval):

```
wifiPhyHelper.Set ("ShortGuardEnabled", BooleanValue(true));
```

Furthermore, 802.11n provides an optional mode (GreenField mode) to reduce preamble durations and which is only compatible with 802.11n devices. This mode is enabled as follows:

```
wifiPhyHelper.Set ("GreenfieldEnabled",BooleanValue(true));
```

802.11n PHY layer can support both 20 (default) or 40 MHz channel width, and 802.11ac PHY layer can use either 20, 40, 80 (default) or 160 MHz channel width. Since the channel width value is overwritten by `WifiHelper::SetStandard`, this should be done post-install using `Config::Set`:

```
WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211ac);
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode", StringValue("VHtMcs9"), "Co
VhtWifiMacHelper mac = VhtWifiMacHelper::Default ();

//Install PHY and MAC
Ssid ssid = Ssid ("ns3-wifi");
mac.SetType ("ns3::StaWifiMac",
"Ssid", SsidValue (ssid),
"ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevice;
staDevice = wifi.Install (phy, mac, wifiStaNode);

mac.SetType ("ns3::ApWifiMac",
"Ssid", SsidValue (ssid));

NetDeviceContainer apDevice;
apDevice = wifi.Install (phy, mac, wifiApNode);

//Once install is done, we overwrite the channel width value
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/ChannelWidth", UintegerValue (160));
```

2.1.3 WifiMacHelper

The next step is to configure the MAC model. We use WifiMacHelper to accomplish this. WifiMacHelper takes care of both the MAC low model and MAC high model. A user must decide if 802.11/WMM-style QoS and/or 802.11n-style High throughput (HT) and/or 802.11ac-style Very High throughput (VHT) support is required.

NqosWifiMacHelper and QosWifiMacHelper

The `ns3::NqosWifiMacHelper` and `ns3::QosWifiMacHelper` configure an object factory to create instances of a `ns3::WifiMac`. They are used to configure MAC parameters like type of MAC.

The former, `ns3::NqosWifiMacHelper`, supports creation of MAC instances that do not have 802.11e/WMM-style QoS nor 802.11n-style High throughput (HT) nor 802.11ac-style Very High throughput (VHT) support enabled.

For example the following user code configures a non-QoS and non-HT MAC that will be a non-AP STA in an infrastructure network where the AP has SSID `ns-3-ssid`:

```
NqosWifiMacHelper wifiMacHelper = NqosWifiMacHelper::Default ();
Ssid ssid = Ssid ("ns-3-ssid");
wifiMacHelper.SetType ("ns3::StaWifiMac",
                       "Ssid", SsidValue (ssid),
                       "ActiveProbing", BooleanValue (false));
```

To create MAC instances with QoS support enabled, `ns3::QosWifiMacHelper` is used in place of `ns3::NqosWifiMacHelper`.

The following code shows an example use of `ns3::QosWifiMacHelper` to create an AP with QoS enabled:

```
QosWifiMacHelper wifiMacHelper = QosWifiMacHelper::Default ();
wifiMacHelper.SetType ("ns3::ApWifiMac",
```

```
                    "Ssid", SsidValue (ssid),
                    "BeaconGeneration", BooleanValue (true),
                    "BeaconInterval", TimeValue (Seconds (2.5)));
```

With QoS-enabled MAC models it is possible to work with traffic belonging to four different Access Categories (ACs): **AC_VO** for voice traffic, **AC_VI** for video traffic, **AC_BE** for best-effort traffic and **AC_BK** for background traffic. In order for the MAC to determine the appropriate AC for an MSDU, packets forwarded down to these MAC layers should be marked using **ns3::QosTag** in order to set a TID (traffic id) for that packet otherwise it will be considered belonging to **AC_BE**.

To create ad-hoc MAC instances, simply use `ns3::AdhocWifiMac` instead of `ns3::StaWifiMac` or `ns3::ApWifiMac`.

HtWifiMacHelper

The `ns3::HtWifiMacHelper` configures an object factory to create instances of a `ns3::WifiMac`. It is used to supports creation of MAC instances that have 802.11n-style High throughput (HT) and QoS support enabled. In particular, this object can be also used to set:

- an MSDU aggregator for a particular Access Category (AC) in order to use 802.11n MSDU aggregation feature;
- block ack parameters like threshold (number of packets for which block ack mechanism should be used) and inactivity timeout.

For example the following user code configures a HT MAC that will be a non-AP STA with QoS enabled, aggregation on AC_VO, and Block Ack on AC_BE, in an infrastructure network where the AP has SSID `ns-3-ssid`:

```
HtWifiMacHelper wifiMacHelper = HtWifiMacHelper::Default ();
Ssid ssid = Ssid ("ns-3-ssid");
wifiMacHelper.SetType ("ns3::StaWifiMac",
                       "Ssid", SsidValue (ssid),
                       "ActiveProbing", BooleanValue (false));

wifiMacHelper.SetMsduAggregatorForAc (AC_VO, "ns3::MsduStandardAggregator",
                                      "MaxAmsduSize", UintegerValue (3839));
wifiMacHelper.SetBlockAckThresholdForAc (AC_BE, 10);
wifiMacHelper.SetBlockAckInactivityTimeoutForAc (AC_BE, 5);
```

This object can be also used to set in the same way as `ns3::QosWifiMacHelper`.

VhtWifiMacHelper

The `ns3::VhtWifiMacHelper` configures an object factory to create instances of a `ns3::WifiMac`. It is used to supports creation of MAC instances that have 802.11ac-style Very High throughput (VHT) and QoS support enabled. This object is similar to `HtWifiMacHelper`.

2.1.4 WifiHelper

We're now ready to create WifiNetDevices. First, let's create a WifiHelper with default settings:

```
WifiHelper wifiHelper = WifiHelper::Default ();
```

What does this do? It sets the default wifi standard to **802.11a** and sets the RemoteStationManager to `ns3::ArfWifiManager`. You can change the RemoteStationManager by calling the `WifiHelper::SetRemoteStationManager` method. To change the wifi standard, call the `WifiHelper::SetStandard` method with the desired standard.

Now, let's use the wifiPhyHelper and wifiMacHelper created above to install WifiNetDevices on a set of nodes in a NodeContainer "c":

```
NetDeviceContainer wifiContainer = WifiHelper::Install (wifiPhyHelper, wifiMacHelper, c);
```

This creates the WifiNetDevice which includes also a WifiRemoteStationManager, a WifiMac, and a WifiPhy (connected to the matching WifiChannel).

The `WifiHelper::SetStandard` method set various default timing parameters as defined in the selected standard version, overwriting values that may exist or have been previously configured. In order to change parameters that are overwritten by `WifiHelper::SetStandard`, this should be done post-install using `Config::Set`:

```
WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211n_2_4GHZ);
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode", StringValue("HtMcs7"), "Co
HtWifiMacHelper mac = HtWifiMacHelper::Default ();

//Install PHY and MAC
Ssid ssid = Ssid ("ns3-wifi");
mac.SetType ("ns3::StaWifiMac",
"Ssid", SsidValue (ssid),
"ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevice;
staDevice = wifi.Install (phy, mac, wifiStaNode);

mac.SetType ("ns3::ApWifiMac",
"Ssid", SsidValue (ssid));

NetDeviceContainer apDevice;
apDevice = wifi.Install (phy, mac, wifiApNode);

//Once install is done, we overwrite the standard timing values
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Slot", TimeValue (MicroSeconds (slot))
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Sifs", TimeValue (MicroSeconds (sifs))
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/AckTimeout", TimeValue (MicroSeconds
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/CtsTimeout", TimeValue (MicroSeconds
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Rifs", TimeValue (MicroSeconds (rifs))
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/BasicBlockAckTimeout", TimeValue (Mic
Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/CompressedBlockAckTimeout", TimeValue
```

There are many *ns-3* attributes that can be set on the above helpers to deviate from the default behavior; the example scripts show how to do some of this reconfiguration.

2.1.5 Mobility configuration

Finally, a mobility model must be configured on each node with Wi-Fi device. Mobility model is used for calculating propagation loss and propagation delay. Two examples are provided in the next section. Users are referred to the chapter on *Mobility* module for detailed information.

2.1.6 Example configuration

We provide two typical examples of how a user might configure a Wi-Fi network – one example with an ad-hoc network and one example with an infrastructure network. The two examples were modified from the two examples in the `examples/wireless` folder (`wifi-simple-adhoc.cc` and `wifi-simple-infra.cc`). Users are encouraged to see examples in the `examples/wireless` folder.

AdHoc WifiNetDevice configuration

In this example, we create two ad-hoc nodes equipped with 802.11a Wi-Fi devices. We use the `ns3::ConstantSpeedPropagationDelayModel` as the propagation delay model and `ns3::LogDistancePropagationLossModel` with the exponent of 3.0 as the propagation loss model. Both devices are configured with `ConstantRateWifiManager` at the fixed rate of 12Mbps. Finally, we manually place them by using the `ns3::ListPositionAllocator`:

```
std::string phyMode ("OfdmRate12Mbps");

NodeContainer c;
c.Create (2);

WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211a);

YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// ns-3 supports RadioTap and Prism tracing extensions for 802.11
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel",
                                "Exponent", DoubleValue (3.0));
wifiPhy.SetChannel (wifiChannel.Create ());

// Add a non-QoS upper mac, and disable rate control (i.e. ConstantRateWifiManager)
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode",StringValue (phyMode),
                              "ControlMode",StringValue (phyMode));
// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, c);

// Configure mobility
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (0.0, 0.0, 0.0));
positionAlloc->Add (Vector (5.0, 0.0, 0.0));
mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (c);

// other set up (e.g. InternetStack, Application)
```

Infrastructure (access point and clients) WifiNetDevice configuration

This is a typical example of how a user might configure an access point and a set of clients. In this example, we create one access point and two clients. Each node is equipped with 802.11b Wi-Fi device:

```
std::string phyMode ("DsssRate1Mbps");

NodeContainer ap;
ap.Create (1);
NodeContainer sta;
sta.Create (2);
```

```
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// ns-3 supports RadioTap and Prism tracing extensions for 802.11
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel;
// reference loss must be changed since 802.11b is operating at 2.4GHz
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel",
                                "Exponent", DoubleValue (3.0),
                                "ReferenceLoss", DoubleValue (40.0459));
wifiPhy.SetChannel (wifiChannel.Create ());

// Add a non-QoS upper mac, and disable rate control
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode",StringValue (phyMode),
                              "ControlMode",StringValue (phyMode));

// Setup the rest of the upper mac
Ssid ssid = Ssid ("wifi-default");
// setup ap.
wifiMac.SetType ("ns3::ApWifiMac",
                 "Ssid", SsidValue (ssid));
NetDeviceContainer apDevice = wifi.Install (wifiPhy, wifiMac, ap);
NetDeviceContainer devices = apDevice;

// setup sta.
wifiMac.SetType ("ns3::StaWifiMac",
                 "Ssid", SsidValue (ssid),
                 "ActiveProbing", BooleanValue (false));
NetDeviceContainer staDevice = wifi.Install (wifiPhy, wifiMac, sta);
devices.Add (staDevice);

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

// other set up (e.g. InternetStack, Application)
```

CHAPTER

THREE

TESTING DOCUMENTATION

At present, most of the available documentation about testing and validation exists in publications, some of which are referenced below.

3.1 Error model

Validation results for the 802.11b error model are available in this technical report

Two clarifications on the results should be noted. First, Figure 1-4 of the above reference corresponds to the *ns-3* NIST BER model. In the program in the Appendix of the paper (80211b.c), there are two constants used to generate the data. The first, packet size, is set to 1024 bytes. The second, "noise", is set to a value of 7 dB; this was empirically picked to align the curves the best with the reported data from the CMU testbed. Although a value of 1.55 dB would correspond to the reported -99 dBm noise floor from the CMU paper, a noise figure of 7 dB results in the best fit with the CMU experimental data. This default of 7 dB is the RxNoiseFigure in the `ns3::YansWifiPhy` model. Other values for noise figure will shift the curves leftward or rightward but not change the slope.

The curves can be reproduced by running the `wifi-clear-channel-cmu.cc` example program in the `examples/wireless` directory, and the figure produced (when GNU Scientific Library (GSL) is enabled) is reproduced below in Figure *Clear channel (AWGN) error model for 802.11b*.

Validation results for the 802.11 OFDM error model are available in this technical report. The curves can be reproduced by running the `ofdm-validation.cc` example program in the `examples/wireless` directory, and the figure is reproduced below in Figure *Frame error rate (NIST model) for OFDM Wi-Fi*.

3.2 MAC validation

Validation of the MAC layer has been performed in [baldo2010].

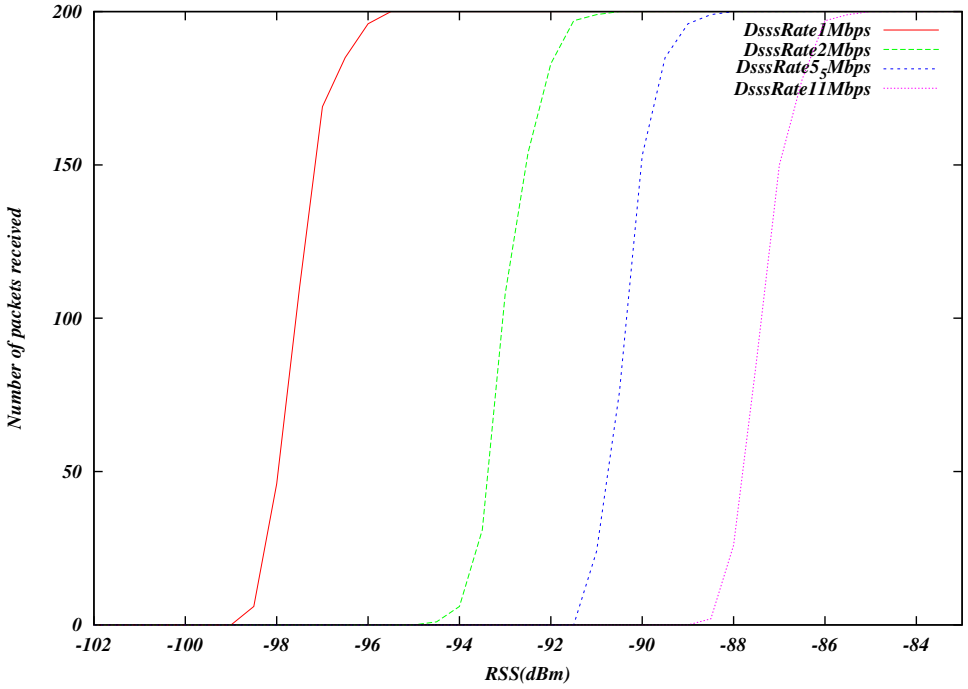

Figure 3.1: Clear channel (AWGN) error model for 802.11b

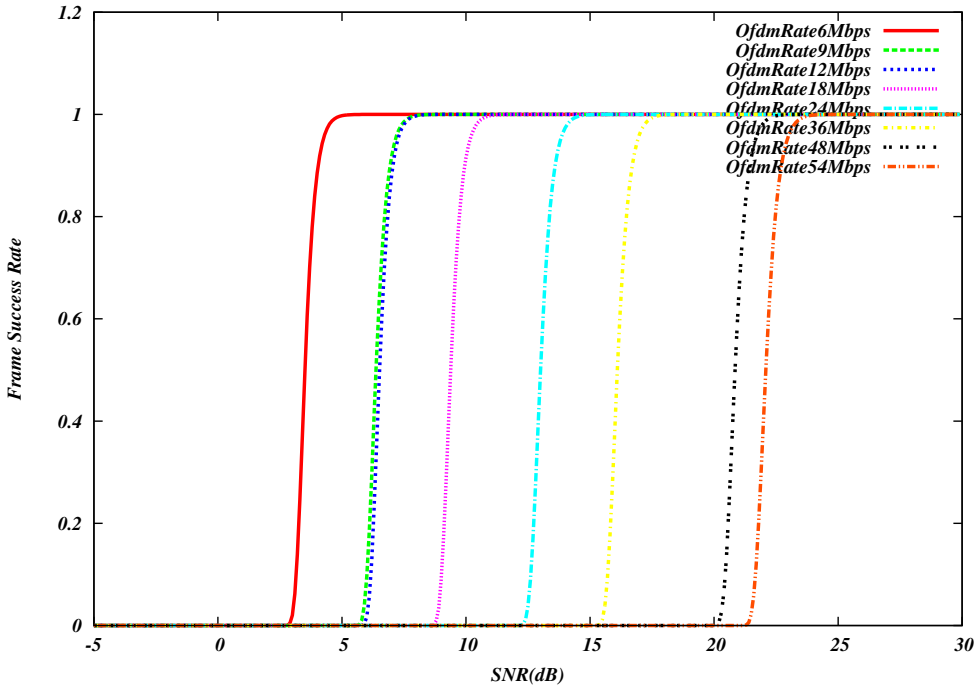

Figure 3.2: Frame error rate (NIST model) for OFDM Wi-Fi

CHAPTER

FOUR

REFERENCES

[ieee80211] IEEE Std 802.11-2012, *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*

[pei80211b] G. Pei and Tom Henderson, Validation of ns-3 802.11b PHY model

[pei80211ofdm] G. Pei and Tom Henderson, Validation of OFDM error rate model in ns-3

[lacage2006yans] M. Lacage and T. Henderson, Yet another Network Simulator

[Haccoun] D. Haccoun and G. Begin, *High-Rate Punctured Convolutional Codes for Viterbi Sequential Decoding*, IEEE Transactions on Communications, Vol. 32, Issue 3, pp.315-319.

[Frenger] Pâl Frenger et al., "Multi-rate Convolutional Codes".

[ji2004sslswn] Z. Ji, J. Zhou, M. Takai and R. Bagrodia, *Scalable simulation of large-scale wireless networks with bounded inaccuracies*, in Proc. of the Seventh ACM Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems, October 2004.

[linuxminstrel] minstrel linux wireless

[lacage2004aarfamrr] M. Lacage, H. Manshaei, and T. Turletti, *IEEE 802.11 rate adaptation: a practical approach*, in Proc. 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems, 2004.

[kim2006cara] J. Kim, S. Kim, S. Choi, and D. Qiao, *CARA: Collision-Aware Rate Adaptation for IEEE 802.11 WLANs*, in Proc. 25th IEEE International Conference on Computer Communications, 2006

[wong2006rraa] S. Wong, H. Yang, S. Lu, and V. Bharghavan, *Robust Rate Adaptation for 802.11 Wireless Networks*, in Proc. 12th Annual International Conference on Mobile Computing and Networking, 2006

[maguolo2008aarfcd] F. Maguolo, M. Lacage, and T. Turletti, *Efficient collision detection for auto rate fallback algorithm*, in IEEE Symposium on Computers and Communications, 2008

[proakis2001] J. Proakis, Digital Communications, Wiley, 2001.

[miller2003] L. E. Miller, "Validation of 802.11a/UWB Coexistence Simulation." Technical Report, October 2003. Available online

[ferrari2004] G. Ferrari and G. Corazza, "Tight bounds and accurate approximations for DQPSK transmission bit error rate", Electronics Letters, 40(20):1284-85, September 2004.

[pursley2009] M. Pursley and T. Royster, "Properties and performance of the IEEE 802.11b complementary code key signal sets," IEEE Transactions on Communications, 57(2);440-449, February 2009.

[akella2007parf] A. Akella, G. Judd, S. Seshan, and P. Steenkiste, 'Self-management in chaotic wireless deployments', in Wireless Networks, Kluwer Academic Publishers, 2007, 13, 737-755. http://www.cs.odu.edu/~nadeem/classes/cs795-WNS-S13/papers/enter-006.pdf

[chevillat2005aparf] Chevillat, P.; Jelitto, J., and Truong, H. L., 'Dynamic data rate and transmit power adjustment in IEEE 802.11 wireless LANs', in International Journal of Wireless Information Networks, Springer, 2005, 12, 123-145. http://www.cs.mun.ca/~yzchen/papers/papers/rate_adaptation/80211_dynamic_rate_power_adjustment_chevillat_j2005.pdf

[hepner2015] C. Hepner, A. Witt, and R. Muenzner, "In depth analysis of the ns-3 physical layer abstraction for WLAN systems and evaluation of its influences on network simulation results", BW-CAR Symposium on Information and Communication Systems (SInCom) 2015. http://sincom.informatik.hs-furtwangen.de/index.php?id=85

[baldo2010] N. Baldo et al., "Validation of the ns-3 IEEE 802.11 model using the EXTREME testbed", Proceedings of SIMUTools Conference, March 2010.