NS3 Wifi

By
Salah Amean Ahmmed Saeed
outline

• NS3 Wi-Fi implementation
• Model overview
• Example:
  • Code explanation
Wifi NetDevice

• *ns-*3 nodes can contain a collection of NetDevice objects
  • Similar to real computer containing interface card, Ethernet, wifi, etc.

• By adding WifiNetDevice objects to *ns-*3 nodes,
  • one can create models of 802.11-based infrastructure and ad hoc networks.
WifiNetDevice Model overview

• Models a wireless network interface controller based on the IEEE 802.11 standard.
• basic 802.11 DCF with infrastructure and adhoc modes
• 802.11a and 802.11b physical layers
• QoS-based EDCA and queueing extensions of 802.11e
• various propagation loss models including Nakagami, Rayleigh, Friis, LogDistance, FixedRss, Random
• various rate control algorithms including Aarf, Arf, Cara, Onoe, Rraa, ConstantRate, and Minstrel
Modular implementation

- Ns3 implementation is modular and contains:
  - **PHY layer models**: they implement DCF and EDCAF
  - **MAC low models**: they implement the MAC-level beacon generation, probing, and association state machines, and
  - **Rate control algorithms** used by the MAC low models
Mac high models

- Provide 3 models
  - **Access Point (AP) (ns3::ApWifiMac)**:
    - implements an AP that generates periodic beacons, and that accepts every attempt to associate
  - **non-AP Station (STA) (ns3::StaWifiMac)**:
    - implements an active probing and association state machine that handles automatic re-association
  - **Ad hoc network (ns3::AdhocWifiMac)**:
    - implements a Wi-Fi MAC that does not perform any kind of beacon generation, probing, or association

- Above models are children of **ns3:regularWifiMac**
  - Attribute of **QosSupported** is added to allow configuration of 802.11e/WMM-style QoS
  - Attribute of **HtSupported** is added to allow configuration of 802.11n
Mac low layer

• Three components:
  • ns3::MacLow:
    • Takes care of RTS/CTS/DATA/ACK transactions.
  • ns3::DcfManager and ns3::DcfState
    • Implements the DCF and EDCAF functions.
  • ns3::DcaTxop and ns3::EdcaTxopN
    • 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 used by QoS-enabled high MACs and also performs QoS operations like 802.11n-style MSDU aggregation.
The WifiChannel and WifiPhy models

- **ns3::WifiPhy** class implement it,
  - Based on YANS simulator
  - Validation of ns-3 802.11b PHY model

- Yans PHY layer states:
  - **TX**: the physical layer is transmitting on behalf of its associated MAC
  - **SYNC**: PHY is synchronized on a signal and it is waiting till it receives the last bit to forward it to MAC.
  - **CCA_BUSY**: PHY is not in TX or SYNC state but the measured energy higher than the energy detection threshold.
  - **IDLE**: PHY is not in TX or SYNC state

- When in TX or SYNC state,
  - Any new packet arriving will be dropped
Places to modifying Wifi model

• wifi-mac-header:
  • To Create or modify wifi frame/header

• MAC low modification:
  • handling new/modified control frames (think RTS/CTS/ACK/Block ACK),
  • making changes to two-way transaction/four-way transaction

• Mac Low: receiveOk
  • Handling of control frame

• MAC high modification
  • handling new management frames (think beacon/probe), beacon/probe generation
  • By modification of; regular-wifi-mac.*, sta-wifi-mac.*, ap-wifi-mac.*, or adhoc-wifi-mac.*

• Wifi Queue Management:
  • dca-txop.* and edca-txop-n.*
Modifying Wifi model

• Channel access management:
  • modify the files dcf-manager.*, which grant access to DcaTxop and EdcaTxopN.

• Fragmentation and RTS thresholds are handled by Wi-Fi remote station manager
  • Fragmentation is handled by DcaTxop or EdcaTxopN while
  • RTS/CTS transaction is handled 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.
TraceSources in ns::wifiPhy

- **PhyTxBegin**: Trace source indicating a packet has begun transmitting over the channel medium
- **PhyTxEnd**: Trace source indicating a packet has been completely transmitted over the channel. NOTE: the only official *WifiPhy* implementation available to this date (*YansWifiPhy*) never fires this trace source.
- **PhyTxDrop**: Trace source indicating a packet has been dropped by the device during transmission
- **PhyRxBegin**: Trace source indicating a packet has begun being received from the channel medium by the device
- **PhyRxEnd**: Trace source indicating a packet has been completely received from the channel medium by the device
- **PhyRxDrop**: Trace source indicating a packet has been dropped by the device during reception
- **MonitorSnifferRx**: Trace source simulating a wifi device in monitor mode sniffing all received frames
- **MonitorSnifferTx**: Trace source simulating the capability of a wifi device in monitor mode to sniff all frames being transmitted
Upper-Level (MAC) Hooks

• The first trace hook is called "Rx" and is fired using the 
  \texttt{ns3::WifiNetDevice::m\_rxLogger} trace hook. The perspective 
  here is looking down into the WifiNetDevice so a receive 
  indicates a packet being sent up from the channel to be 
  forwarded up to higher layers.

• The second trace hook is called "Tx" and is fired using the 
  \texttt{ns3::WifiNetDevice::m\_txLogger} trace hook. This trace hook 
  indicates a packet has been sent from higher layers down to 
  the net device for transmission onto the network.
Use case

• AddTraceSource ("Tx", "Send payload to the MAC layer.", MakeTraceSourceAccessor (&WifiNetDevice::m_txLogger))

• AddTraceSource ("Tx", "Packet transmission is starting.", MakeTraceSourceAccessor (&WifiPhy::m_txTrace))

• Above could be used to calculate the delay across the MAC layer (how long between enqueue and transmission, due to busy MAC), using the echo application for adhoc wifi nodes
Low-Level (PHY) Hooks

• First, we provide a trace hook to indicate **state changes**. This trace source is called "State" and is fired using the `ns3::WifiPhyStateHelper::m_stateLogger` trace source.

• We also provide a trace hook to indicate the successful reception of a packet from the channel. This trace source is called "RxOk" and is accessed using the `ns3::WifiPhyStateHelper::m_rxOkTrace` trace source.

• There also exists a trace hook to indicate an unsuccessful reception of a packet from the channel. This trace source is called "RxError" and is accessed using the `ns3::WifiPhyStateHelper::m_rxErrorTrace` trace source.

• There is a trace hook to indicate that transmission of a packet is starting onto the channel. This trace source is called "Tx" (don't confuse it with the higher layer "Tx" hook) and is fired using the `ns3::WifiPhyStateHelper::m_txTrace` trace source.
Remote Station Hooks

• We provide access to changes in the per-remote-station RTS counter through the "Ssrc" trace source which is fired using the $\text{ns3::WifiRemoteStation::m_ssrc}$ (station short retry count) trace hook.

• Finally, we provide access to the per-remote-station SLRC counter that indications the number of retransmissions of data. Changes to this counter are traced using the $\text{ns3::WifiRemoteStation::m_slrc}$ source.
Rate control algorithms

- Ns3 implements rate adaptation algorithms like:
  - IdealWifiManager
  - ns3::ArfMacStations
  - ns3::AArfMacStations
  - ns3::IdealMacStations
  - ns3::CrMacStations
  - ns3::OnoeMacStations
  - ns3::AmrrMacStation
DCF manager

• keep track of the state needed for a single DCF function.
• Multiple instances of a DcfState can be registered in a single DcfManager.
• The first DcFstate added gets the higher priority and second gets the second and so on.
ns3::DcfManager Class

Public Member Functions

- void SetupPhyListener (Ptr< WifiPhy > phy)
- void SetupLowListener (Ptr< MacLow > low)
- void SetSlot (Time slotTime)
- void SetSifs (Time sifs)
- void SetEifsNoDifs (Time eifsNoDifs)
- Time GetEifsNoDifs () const
- void Add (DcfState *dcf)
- void RequestAccess (DcfState *state)
- void NotifyRxStartNow (Time duration)
- void NotifyRxEndOkNow (void)
- void NotifyRxEndErrorNow (void)
- void NotifyTxStartNow (Time duration)
- void NotifyMaybeCcaBusyStartNow (Time duration)
- void NotifySwitchingStartNow (Time duration)
- void NotifyNavResetNow (Time duration)
- void NotifyCtsTimeoutStartNow (Time duration)
- void NotifyCtsTimeoutResetNow ()

Diagram:

- m_eventImpl
- m_data
- ns3::EventId
- ns3::Time
- m_accessTimeout
- ns3::DcfManager

Legend:
- m_lastBusyStart
- m_eifsNoDifs
- m_lastTxStart
- m_lastRxStart
- m_lastNavStart
- m_lastBusyDuration
- m_sifs
- m_lastAckTimeoutEnd
- m_lastSwitchingStart
- m_lastNavDuration
...
DcfState

- keep track of the state needed for a single DCF function.
- Multiple instances of a DcfState can be registered in a single DcfManager to implement 802.11e-style relative QoS.
- DcfState::SetAifsn and DcfState::SetCwBounds allow the user to control the relative QoS differentiation.
# ns3::DcfState Class Reference

## Public Member Functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>SetAifsn (uint32_t aifsn)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>SetCwMin (uint32_t minCw)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>SetCwMax (uint32_t maxCw)</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>GetAifsn (void) const</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>GetCwMin (void) const</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>GetCwMax (void) const</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>ResetCw (void)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>UpdateFailedCw (void)</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>StartBackoffNow (uint32_t nSlots)</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>IsAccessRequested (void) const</td>
<td></td>
</tr>
</tbody>
</table>
Dca_txop(1)

- handle packet fragmentation and retransmissions. This class implements the packet fragmentation and retransmission policy.
- It uses the ns3::MacLow and ns3::DcfManager helper classes to respectively send packets and decide when to send them. Packets are stored in a ns3::WifiMacQueue until they can be sent.
Dca_txop(2)

• Fragmentation policy:
  • uses a simple fragmentation threshold: any packet bigger than this threshold is fragmented in fragments whose size is smaller than the threshold.

• Retransmission policy:
  • The retransmission policy is also very simple: every packet is retransmitted until it is either successfully transmitted or it has been retransmitted up until the ssr or slr thresholds.
Simple Wi-Fi Process(1)

• A data packet, from upper layer, should be inserted into the queue (i.e., temporary storage for data packets).

• In ns-3, the IEEE 802.11 MAC protocol is implemented with several C++ classes.

• Five classes are most important to understand the basic procedure of the transmission and reception:
  • DcaTxop,
  • DcfManager,
  • MacLow,
  • YansWifiPhy, and YansWifiChannel
Simple Wi-Fi Process(2)

- **DCA** and **DCF** mean the dynamic channel assignment and the discrete contention function, respectively.
- When a data packet is inserted into the queue, DCA and DCF check whether another packet transmission or reception is processing.
- If there is no other packet, or when the packet transmission is finished, one data packet is extracted from the queue and forwarded to *MacLow* class.
- *MacLow* checks whether this data packet is multicast packet, or requires the ready to send (RTS) or the fragmentation.
- After estimating the duration of the transmission, *MacLow* forwards the data packet to *YansWifiPhy* and *YansWifiPhy* forwards it to *YansWifiChannel*. 
Simple Wi-Fi Process(3)

- The backoff procedure
  - Selecting a random number and waiting for a contention
  - Is required only if channel was busy just before. This means that if there was no transmission or reception (i.e., channel is idle) for a specific time (i.e., during DIFS), MAC protocol sends a data packet immediately upon getting a data packet from an upper layer.

- Here is an example of WiFi infrastructure network.
- One node is transmitting UDP packets to another node. One node is overhearing packets
Basic transmission
Basic transmission

• Upper layer: WifiNetDevice::send(pkt, address, protocol)
• Called from higher layer to send packet into Network Device to the specified destination address

```cpp
namespace ns3::WifiNetDevice
{
  bool Send(Ptr<Packet> packet,
             const Address &dest,
             uint16_t protocolNumber);
}
```

- `packet`: packet sent from above down to Network Device
- `dest`: mac address of the destination (already resolved)
- `protocolNumber`: identifies the type of payload contained in this packet. Used to call the right L3Protocol when the packet is received.
Basic transmission

- adhocWifiMac::enqueue()
Basic transmission

- **ns3::DcaTxop::Queue**:

```cpp
void ns3::DcaTxop::Queue ( Ptr<const Packet> packet,
                          const WifiMacHeader & hdr
                          )
```

**Parameters**
- `packet` packet to send
- `hdr` header of packet to send.

Store the packet in the internal queue until it can be sent safely.
Basic transmission

- DcaTxop::startAccessIfNeeded
  - Request to access dcf manager if needed when there is no pending packet (i.e., if m_accessRequested==true)

```c
void ns3::DcaTxop::StartAccessIfNeeded ( void )
```

Request access from DCF manager if needed.
Basic transmission

- DcfManager::RequestAccess(),
  - Notify the DcfManager that a specific DcfState needs access to the medium.
  - The DcfManager is then responsible for starting an access timer and, invoking DcfState::DoNotifyAccessGranted when the access is granted if it ever gets granted.
- If there is another packet pending, a signal collision through DcTxop::notifyCollision() which select a random back-off number and try to access gain.
Basic transmission

- **DcfManager::DoGrantAccess()**
  - Check the internal collision among four different access among different queues (for EDCA of IEEE802.11e).

- **DcfState::notifyAccessGranted**
  - Set m_accessRequested to false
  - Notify channel access is granted

- **DcaTxop::NotifyAccessGranted()**
  - Extract one packet from the queue.
  - Set new sequence
  - Check pkt (multicast msg, or requiring fragment or RTS)

- **MacLow::startTransmission()**
  - If RTS is required, call SendRtsForPacket().
  - If not, call SendDataPacket().

- **MacLow::sendDataPacket**
  - Estimate the duration of sum of transmission, ACK reception, and/or next transmission (for Block ACK or fragmentation)

- **MacLow::ForwardDown()**
  - Toss the packet to PHY layer
Basic transmission

- `YansWifiPhy::SendPacket()`
  - Change current PHY state to TX (this command will eventually inform DCF that a transmission is starting);
  - Toss the packet to CHANNEL layer.

- `YansWifiChannel::Send()`
  - Estimate receive power;
  - Set the propagation delay to trigger `YansWifiChannel::Receive()` of neighbors after a short period.
Basic reception

- **YansWifiChannel::Receive()**
  - Triggered after the propagation delay (set by sender through YansWifiChannel::Receive()).
  
- **YansWifiPhy::StartReceivePacket()**
  - Calculate the current interference level; Drop the received packet if the state of PHY is not idle or the receive power is below than a threshold;
  - Set the delay to trigger YansWifiPhy::EndSync() after the packet transmission.

- **YansWifiPhy::EndSync()**
  - Estimate the packet error rate (PER) from the signal to noise ratio (SNR) (i.e., signal power / (noise + interference power));
  - If a random value is smaller than the PER value, the packet is dropped.
Basic reception

- MacLow::ReceiveOk()
  - If the packet is the RTS packet, schedule MacLow::SendCtsAfterRts();
  - If it is CTS, schedule MacLow::SendDataAfterCts();
  - If it is DATA, forward it to the upper; If the destination is me, schedule MacLow::SendAckAfterData().

- MacRxMiddle::Receive()
  - Check whether the received packet is a duplicated one with a sequence number.

- AdhocWifiMac::ForwardUp()
  - Just forward the received packet to the upper.

- WifiNetDevice::ForwardUp()
  - To Upper Layer
  - Unless the destination is other node, forward the received packet to the upper layer.
```cpp
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"

using namespace ns3;

int main (int argc, char *argv[])
{
    NodeContainer wifiNodes;
    wifiNodes.Create (3);

    YansWifiPhyHelper wifiPhy =
        YansWifiPhyHelper::Default ();
    YansWifiChannelHelper wifiChannel =
        YansWifiChannelHelper::Default ();

    Lpv4AddressHelper address;
    address.SetBase ("10.1.1.0", "255.255.255.0");
    Lpv4InterfaceContainer wifiInterfaces;
    wifiInterfaces = address.Assign (wifiDevices);

    UdpEchoServerHelper echoServer (9);
    ApplicationContainer serverApps =
        echoServer.Install (wifiNodes.Get (0));
    serverApps.Start (Seconds (1.0));
    serverApps.Stop (Seconds (10.0));

    UdpEchoClientHelper echoClient
        (wifiInterfaces.GetAddress (0), 9);
    echoClient.SetAttribute ("MaxPackets",
                             UintegerValue (2));
    echoClient.SetAttribute ("Interval", TimeValue
                             (Seconds (1.0)));
    echoClient.SetAttribute ("PacketSize",
                             UintegerValue (1024));
    ApplicationContainer clientApps =
        echoClient.Install (std::randomAppContainer (2));
    clientApps.Start (Seconds (0.0));
    clientApps.Stop (Seconds (10.0));
}
```
wifiPhy.SetChannel (wifiChannel.Create ());
WifiHelper wifi = WifiHelper::Default ();
NqosWifiMacHelper wifiMac =
NqosWifiMacHelper::Default ();
wifi.SetRemoteStationManager
("ns3::ConstantRateWifiManager");
wifiMac.SetType("ns3::AdhocWifiMac");
NetDeviceContainer wifiDevices = wifi.Install(
 wifiPhy, wifiMac, wifiNodes );
MobilityHelper mobility;
mobility.SetPositionAllocator
("ns3::GridPositionAllocator",
"MinX", DoubleValue (0.0),
"MinY", DoubleValue (0.0),
"DeltaX", DoubleValue (5.0),
"DeltaY", DoubleValue (10.0),
"GridWidth", UintegerValue (3),
"LayoutType", StringValue ("RowFirst"));
mobility.SetMobilityModel
("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiNodes);

IntegerValue (1024));
ApplicationContainer clientApps =
echoClient.Install ( wifiNodes.Get( 2 ) );
clientApps.Start( Seconds (2.0) );
clientApps.Stop( Seconds (10.0) );
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
Simulator::Stop (Seconds (5.05));
Simulator::Run ();
Simulator::Destroy ();
return 0;
Basic csma/ca
Design Approach

Functionalities
- MSDN functions that are visible to users
- Attribute and configuration

Reusability
- Module can be reused in another simulation

Dependencies
- IPv4/6 dependence
- Remove ip dependence
**ErrorModel** example of model creation

• An error model exists in *ns-2*. It allows packets to be passed to a stateful object that determines, based on a random variable, whether the packet is corrupted

• Then the caller decides what to do with the corrupted packet (drop it)

• main API
  • It is a function (packet is its parameter) that return a Boolean value
  • Packet data buffer may (or not) get corrupted

```cpp
class ErrorModel
{
public:
    /*
     * returns true if the Packet is to be considered as errored/corrupted
     * @param pkt Packet to apply error model to
     * */
    bool IsCorrupt (Ptr<Packet> pkt);
};
```
Scaffolding

• Building project skeleton
  • location of the project e.g., src/network/(error model)
    • Where the packet implemented

• Revising the coding style of ns3

• Placing source code of the new module
  • E.g., error model is placed in network model (src/network) to detect any corrupted packets

• Integrating the new model into waf build system
  • add your files to the wscript file found in each directory

• Then testing the new model using ./
Initial Implementation

• After knowing what are you supposed to achieve with the model, start to define main classes to add functionality to it later
• Creating subclasses
References

- http://www2.engr.arizona.edu/~junseok/ns3_wifi.htm
- https://www.gitorious.org/efikamx/linux-kernel/source/3ebc73888f8ae3d22f64fb8c85f5fff08ceb3aa6:net/mac80211/sta_info.c#L29-47
Visual studio

- An effort to provide a version of ns-3 which can be built using a native Windows compiler
- ns3.18
- Most essential modules
- Works with 2012 only
Prerequisites

• Visual Studio Express 2012 for C++
  • 2012 only
  • http://www.nsnam.org/wiki/Ns-3_on_Visual_Studio_2012

• Mercurial
  • To get the ns3 for windows
Supported modules

- antenna
- aodv
- applications
- bridge (partial)
- buildings
- config-store (partial)
- core
- csma
- csma-layout
- dsdv
- dsr
- energy
- flow-monitor
- internet
- lte
- mesh
- mobility
- mpi (partial)
- netanim
- network
- nix-vector-routing
- olsr
- point-to-point
- point-to-point-layout
- propagation
- spectrum
- stats (partial)
- uan
- virtual-net-device
- **wifi**
- wimax
Unsupported modules

• Pure unix-centric
• Third party libraries
• brite
• fd-net-device
• topology-read
• visualize
output

• C:\ns-3-win2\windows\ns-3-dev\Debug

```
C:\ns-3-win2\windows\ns-3-dev\Debug>main.exe
Testing 1 packets sent with receiver rss -90
Received one packet!

C:\ns-3-win2\windows\ns-3-dev\Debug>main.exe
Testing 1 packets sent with receiver rss -99
C:\ns-3-win2\windows\ns-3-dev\Debug>main.exe
Testing 10 packets sent with receiver rss -99
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
Received one packet!
```

C:\ns-3-win2\windows\ns-3-dev\Debug>
This program configures a grid (default 5x5) of nodes on an 802.11b physical layer, with 802.11b NICs in adhoc mode, and by default, sends one packet of 1000 (application) bytes to node 1.

The default layout is like this, on a 2-D grid:

```
 n20  n21  n22  n23  n24
 n15  n16  n17  n18  n19
 n10  n11  n12  n13  n14
 n5   n6   n7   n8   n9
 n0   n1   n2   n3   n4
```

The layout is affected by the parameters given to GridPositionAllocator; by default, GridWidth is 5 and numNodes is 25.

There are a number of command-line options available to control the default behavior. The list of available command-line options can be listed with the following command:

```
./waf --run "wifi-simple-adhoc-grid --help"
```

Note that all ns-3 attributes (not just the ones exposed in the below script) can be changed at command line; see the ns-3 documentation.

For instance, for this configuration, the physical layer will stop successfully receiving packets when distance increases beyond the default of 500m. To see this effect, try running:
// ./waf --run "wifi-simple-adhoc --distance=500"
// ./waf --run "wifi-simple-adhoc --distance=1000"
// ./waf --run "wifi-simple-adhoc --distance=1500"
//
// The source node and sink node can be changed like this:
// ./waf --run "wifi-simple-adhoc --sourceNode=20 --sinkNode=10"
//
// This script can also be helpful to put the Wifi layer into verbose
// logging mode; this command will turn on all wifi logging:
// ./waf --run "wifi-simple-adhoc-grid --verbose=1"
//
// By default, trace file writing is off-- to enable it, try:
// ./waf --run "wifi-simple-adhoc-grid --tracing=1"
//
// When you are done tracing, you will notice many pcap trace files
// in your directory. If you have tcpdump installed, you can try this:
// tcpdump -r wifi-simple-adhoc-grid-0-0.pcap -nn -tt
//
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/olsr-helper.h"
#include "ns3/ipv4-static-routing-helper.h"
#include "ns3/ipv4-list-routing-helper.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

NS_LOG_COMPONENT_DEFINE ("WifiSimpleAdhocGrid");

using namespace ns3;

void ReceivePacket (Ptr<Socket> socket)
{
    NS_LOG_UNCOND ("Received one packet!");
    std::cout<<"received one packet"<<std::endl;
}

static void GenerateTraffic (Ptr<Socket> socket, uint32_t pktSize,
                             uint32_t pktCount, Time pktInterval )
{
    if (pktCount > 0)
    {
        socket->Send (Create<Packet> (pktSize));
        std::cout<<"socket is sending a Packet with size"<<pktSize<<std::endl;
        std::cout<<"socket is sending a Packet number"<<pktCount<<std::endl;
        std::cout<<"socket is sending a Packet at time"<<pktInterval<<std::endl;
        Simulator::Schedule (pktInterval, &GenerateTraffic,
                             socket, pktSize, pktCount-1, pktInterval);
        std::cout<<"packet is scheduled to be sent in Simulator::schedule"<<std::endl;
    }
    else
    {
        std::cout<<"socket is closed since no packet to send"<<std::endl;
    }
socket->Close();

int main (int argc, char *argv[])
{
    std::string phyMode ("DsssRate1Mbps");
    std::cout<<"phyMode is DsssRate1Mbps"<<phyMode<<std::endl;

    double distance = 20;  // m
    std::cout<<"distance is "<<distance<<std::endl;

    uint32_t packetSize = 1000;  // bytes
    std::cout<<"Packet size"<<packetSize<<std::endl;

    uint32_t numPackets = 10;
    std::cout<<"numPackets "<<numPackets<<std::endl;

    uint32_t numNodes = 25;  // by default, 5x5
    uint32_t sinkNode = 0;
    uint32_t sourceNode = 24;
    double interval = 1.0; // seconds
    bool verbose = false;
    bool tracing = false;

    CommandLine cmd;
    /**
    * \ingroup core
    * \defgroup commandline Command Line Parsing
    * 
    * A uniform way to specify program documentation,
    * allowed command line arguments and help strings,
    * and set any attribute or global value, all from
    * the command line directly.
    *
    * The main entry point is CommandLine
    */
    /**
Instances of this class can be used to parse command-line arguments. Programs can register a general usage message with `CommandLine::Usage`, and arguments with `CommandLine::AddValue`. POD argument variables will be set directly; more general arguments can be processed via a Callback.

`CommandLine` also provides handlers for these standard arguments:

```plaintext
--PrintGlobals:         Print the list of globals.
--PrintGroups:          Print the list of groups.
--PrintGroup=[group]:   Print all TypeIds of group.
--PrintTypeIds:         Print all TypeIds.
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintHelp:            Print this help message.
```

The more common `--help` is a synonym for `--PrintHelp`; an example is given below.

Finally, `CommandLine` processes Attribute and GlobalValue arguments.

In use, arguments are given in the form

```plaintext
--arg=value  --toggle
```

Most arguments expect a value, as in the first form, `--arg=value`.  

Toggles, corresponding to boolean arguments, can be given in any of
the forms
\code
--toggle1 --toggle2=1 --toggle3=t --toggle4=true
\endcode
* The first form changes the state of toggle1 from its default;
* all the rest set the corresponding boolean variable to true.
* \c 0, \c f and \c false are accepted to set the variable to false.
* CommandLine can set the initial value of every attribute in the system
* with the
* \c --TypeIdName::AttributeName=value syntax, for example
* \code
* --Application::StartTime=3s
* \endcode
* CommandLine can also set the value of every GlobalValue in the system with the \c --GlobalValueName=value syntax, for example
* \code
* --SchedulerType=HeapScheduler
* \endcode
* A simple example is in \c src/core/example/command-line-example.cc
* The heart of that example is this code:
* \code
* int         intArg = 1;
* bool        boolArg = false;
* std::string strArg  = "strArg default";
* \endcode
* CommandLine cmd;
* cmd.Usage ("CommandLine example program.\n"
}
This little program demonstrates how to use CommandLine.
"

\endcode

after which it prints the values of each variable. (The \c SetCbArg function
is not shown.)

* Here is the output from a few runs of that program:
* 
* \endcode
* $ ./waf --run="command-line-example"
* intArg:   1
* boolArg:  false
* strArg:   "strArg default"
* cbArg:    "cbArg default"
* 
* $ ./waf --run="command-line-example --intArg=2 --
boolArg=Hello --cbArg=World"
* intArg:   2
* boolArg:  true
* strArg:   "Hello"
* cbArg:    "World"
* 
* $ ./waf --run="command-line-example --help"
* ns3-dev-command-line-example-debug [Program Arguments]

[General Arguments]
* 
* CommandLine example program.
This little program demonstrates how to use CommandLine.

Program Arguments:
- --intArg: an int argument [1]
- --boolArg: a bool argument [false]
- --strArg: a string argument [strArg default]
- --cbArg: a string via callback

General Arguments:
- --PrintGlobals: Print the list of
  globals.
- --PrintGroups: Print the list of
groups.
- --PrintGroup=[group]: Print all TypeIds of
group.
- --PrintTypeIds: Print all TypeIds.
- --PrintAttributes=[typeid]: Print all attributes
  of typeid.
- --PrintHelp: Print this help
  message.
*/

```cpp
    cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
    cmd.AddValue ("distance", "distance (m)", distance);
    cmd.AddValue ("packetSize", "size of application packet
    sent", packetSize);
    cmd.AddValue ("numPackets", "number of packets generated",
    numPackets);
    cmd.AddValue ("interval", "interval (seconds) between
    packets", interval);
    cmd.AddValue ("verbose", "turn on all WifiNetDevice log
    components", verbose);
    cmd.AddValue ("tracing", "turn on ascii and pcap tracing",
    tracing);
    cmd.AddValue ("numNodes", "number of nodes", numNodes);
    cmd.AddValue ("sinkNode", "Receiver node number",
    sinkNode);
```
cmd.AddValue("sourceNode", "Sender node number", sourceNode);

cmd.Parse(argc, argv);
/**
 * Parse the program arguments
 *
 * \param argc the 'argc' variable: number of arguments
 * (including the main program name as first element).
 * \param argv the 'argv' variable: a null-terminated array of strings,
 * each of which identifies a command-line argument.
 *
 * Obviously, this method will parse the input command-line arguments and
 * will attempt to handle them all.
 *
 * As a side effect, this method saves the program basename, which
 * can be retrieved by GetName().
 */

// Convert to time object
Time interPacketInterval = Seconds(interval);
std::cout<<"interpacket interval"<<interPacketInterval<<std::endl;
/**
 * \brief create ns3::Time instances in units of seconds.
 * *
 * \code
 * Time t = Seconds(2.0);
 * Simulator::Schedule(Seconds(5.0), ...);
 * \endcode
 * \param seconds seconds value
 * \relates ns3::Time
 */
// disable fragmentation for frames below 2200 bytes
Config::SetDefault("ns3::WifiRemoteStationManager::FragmentationThreshold",
StringValue("2200"));

/*
Attributes

IsLowLatency: If true, we attempt to modelize a so-called low-latency device: a device where decisions about tx parameters can be made on a per-packet basis and feedback about the transmission of each packet is obtained before sending the next. Otherwise, we modelize a high-latency device, that is a device where we cannot update our decision about tx parameters after every packet transmission.
Set with class: BooleanValue
Underlying type: bool
Flags: read
MaxSsrc: The maximum number of retransmission attempts for an RTS. This value will not have any effect on some rate control algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 7
Flags: construct write read
MaxSlrc: The maximum number of retransmission attempts for a DATA packet.
This value will not have any effect on some rate control algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 7
Flags: construct write read
RtsCtsThreshold: If the size of the data packet + LLC header + MAC header + FCS trailer is bigger than this value, we use an RTS/CTS handshake before sending the data, as per IEEE Std. 802.11-2012, Section 9.3.5.
This value will not have any effect on some rate control algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 2346
Flags: construct write read
FragmentationThreshold: If the size of the data packet + LLC header + MAC header + FCS trailer is bigger than this value, we fragment it such that the size of the fragments are equal or smaller than this value, as per IEEE Std. 802.11-2012, Section 9.5.
This value will not have any effect on some rate control algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 2346
Flags: construct write read
NonUnicastMode: Wifi mode used for non-unicast transmissions.
Set with class: WifiModeValue
Underlying type: WifiMode
Initial value: Invalid-WifiMode
Flags: construct write read
DefaultTxPowerLevel: Default power level to be used for transmissions. This is the power level that is used by all those WifiManagers that do not implement TX power control.
Set with class: ns3::UintegerValue
Underlying type: uint8_t 0:255
Initial value: 0
Flags: construct write read
TraceSources

MacTxRtsFailed: The transmission of a RTS by the MAC layer has failed
MacTxDataFailed: The transmission of a data packet by the MAC layer has failed
MacTxFinalRtsFailed: The transmission of a RTS has exceeded the maximum number of attempts
MacTxFinalDataFailed: The transmission of a data packet has exceeded the maximum number of attempts

*/
/*
WifiRemoteStationManager::SetFragmentationThreshold (uint32_t threshold)
{
    DoSetFragmentationThreshold (threshold);
}
*/
/**
* \param name the full name of the attribute
* \param value the value to set.
*
* This method overrides the initial value of the matching attribute. This method cannot fail: it will crash if the input attribute name or value is invalid.
*/
/*
void SetDefault (std::string name, const AttributeValue &value)
{
    NS_LOG_FUNCTION (name << &value);
    if (!SetDefaultFailSafe(name, value))
    {
        NS_FATAL_ERROR ("Could not set default value for " << name);
    }
}
*/
// turn off RTS/CTS for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"));
// Fix non-unicast data rate to be the same as that of unicast
Config::SetDefault("ns3::WifiRemoteStationManager::NonUnicastMode",
    StringValue(phyMode));

NodeContainer c;
c.Create (numNodes);
/**
* \brief keep track of a set of node pointers.
* 
* Typically ns-3 helpers operate on more than one node at a time. For example
* a device helper may want to install devices on a large number of similar
* nodes. The helper Install methods usually take a
NodeContainer as a
* parameter. NodeContainers hold the multiple Ptr<Node>
which are used
* to refer to the nodes.
*/
/**
* \brief Create n nodes and append pointers to them to the end of this
* NodeContainer.
* 
* Nodes are at the heart of any ns-3 simulation. One of the first tasks that
* any simulation needs to do is to create a number of nodes. This method
* automates that task.
* 
* \param n The number of Nodes to create
*/

// The below set of helpers will help us to put together the wifi NICs we want
WifiHelper wifi;
/*
 * helps to create WifiNetDevice objects:
 * 1-This class can help to create a large set of similar
 *    WifiNetDevice objects and
 * 2-to configure a large set of their attributes during
 *    creation.
 * /
 /**
 * Create a Wifi helper in an empty state: all its
 * parameters
 * must be set before calling ns3::WifiHelper::Install
 * /
 // WifiHelper ();

 /**
 * \returns a new WifiHelper in a default state
 * 
 * The default state is defined as being an Adhoc MAC
 * layer with
 * an ARF rate control algorithm
 * and both objects using their default attribute values.
 * By default, configure MAC and PHY
 * for 802.11a.
 * */

 if (verbose)
 {  
     wifi.EnableLogComponents ();
     // Turn on all Wifi logging
     /*
        LogComponentEnable ("Aarfcd", LOG_LEVEL_ALL);
        LogComponentEnable ("AdhocWifiMac", LOG_LEVEL_ALL);
        LogComponentEnable ("AmrrWifiRemoteStation",
                         LOG_LEVEL_ALL);
        LogComponentEnable ("ApWifiMac", LOG_LEVEL_ALL);
        LogComponentEnable ("ns3::ArfWifiManager", LOG_LEVEL_ALL);
        LogComponentEnable ("Cara", LOG_LEVEL_ALL);
        LogComponentEnable ("DcaTxop", LOG_LEVEL_ALL);
        LogComponentEnable ("DcfManager", LOG_LEVEL_ALL);
    */
}
LogComponentEnable ("DsssErrorRateModel", LOG_LEVEL_ALL);
LogComponentEnable ("EdcaTxopN", LOG_LEVEL_ALL);
LogComponentEnable ("InterferenceHelper", LOG_LEVEL_ALL);
LogComponentEnable ("Jakes", LOG_LEVEL_ALL);
LogComponentEnable ("MacLow", LOG_LEVEL_ALL);
LogComponentEnable ("MacRxMiddle", LOG_LEVEL_ALL);
LogComponentEnable ("MsduAggregator", LOG_LEVEL_ALL);
LogComponentEnable ("MsduStandardAggregator", LOG_LEVEL_ALL);
LogComponentEnable ("NistErrorRateModel", LOG_LEVEL_ALL);
LogComponentEnable ("OnoeWifiRemoteStation", LOG_LEVEL_ALL);
LogComponentEnable ("PropagationLossModel", LOG_LEVEL_ALL);
LogComponentEnable ("RegularWifiMac", LOG_LEVEL_ALL);
LogComponentEnable ("RraaWifiManager", LOG_LEVEL_ALL);
LogComponentEnable ("StaWifiMac", LOG_LEVEL_ALL);
LogComponentEnable ("SupportedRates", LOG_LEVEL_ALL);
LogComponentEnable ("WifiChannel", LOG_LEVEL_ALL);
LogComponentEnable ("WifiPhyStateHelper", LOG_LEVEL_ALL);
LogComponentEnable ("WifiPhy", LOG_LEVEL_ALL);
LogComponentEnable ("WifiRemoteStationManager", LOG_LEVEL_ALL);
LogComponentEnable ("YansErrorRateModel", LOG_LEVEL_ALL);
LogComponentEnable ("YansWifiChannel", LOG_LEVEL_ALL);
LogComponentEnable ("YansWifiPhy", LOG_LEVEL_ALL);

YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();

YansWifiPhyHelper YansWifiPhyHelper::Default (void)
{
  YansWifiPhyHelper helper;
  helper.SetErrorRateModel ("ns3::NistErrorRateModel");
//model for the error rate for different modulations

return helper;
describes NistErrorRateModel
This technical note publishes a new frame error
rate model for OFDM signals for use in the ns-3
discrete event network simulator wireless models.
The new error rate model is more closely aligned
with recently published experimental results from
a physical-layer testbed.
}

* \brief Make it easy to create and manage PHY objects for
the yans model.
* The yans PHY model is described in "Yet Another Network
Simulator",
* http://cutebugs.net/files/wns2-yans.pdf
* The Pcap and ascii traces generated by the EnableAscii
and EnablePcap methods defined
* in this class correspond to PHY-level traces and come to
us via WifiPhyHelper
*/

The PHY layer can be in one of four states:
• TX: the PHY is currently transmitting a signal on behalf
of its associated MAC
• SYNC: 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 the TX or SYNC but
the energy measured on the medium is higher than
Energy Detection Threshold (as defined by the Clear
Channel Assessment function mode 1).
• IDLE: the PHY is not in the TX or SYNC states.
When the first bit of a new packet is received while the
PHY is in the TX or the SYNC states, the packet received is dropped. Otherwise, if the PHY is IDLE or CCA BUSY, we calculate the received energy of the first bit of this new signal and compare it against our EDThreshold. If the energy of the packet $k$ is higher, then the PHY moves to SYNC state and schedules an event when the last bit of the packet is expected to be received. Otherwise, the PHY either switches to CCA BUSY if the total energy received reached EDTreshold or stays in IDLE state. In both cases, the packet is dropped.

```cpp
/*
// set it to zero; otherwise, gain will be added
wifiPhy.Set ("RxGain", DoubleValue (-10));
*/
void YansWifiPhyHelper::Set (std::string name, const AttributeValue &v)
{
    m_phy.Set (name, v);
}
```

// ns-3 supports RadioTap and Prism tracing extensions for 802.11b

```cpp
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
/*
void YansWifiPhyHelper::SetPcapDataLinkType (enum SupportedPcapDataLinkTypes dlt)
{
    switch (dlt)
    {
    case DLT_IEEE802_11:
        m_pcapDlt = PcapHelper::DLT_IEEE802_11;
```
return;
case DLT_PRISM_HEADER:
    m_pcapDlt = PcapHelper::DLT_PRISM_HEADER;
    return;
case DLT_IEEE802_11_RADIO:
    m_pcapDlt = PcapHelper::DLT_IEEE802_11_RADIO;
    return;
default:
    NS_ABORT_MSG ("YansWifiPhyHelper::SetPcapFormat(): Unexpected format");
}

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
//the propagation speed is constant

wifiChannel.AddPropagationLoss("ns3::FriisPropagationLossModel");/*
a Friis propagation loss model

The Friis propagation loss model was first described in
"A Note on a Simple
 Transmission Formula", by "Harald T. Friis".

* Add a propagation loss model to the set of currently-configured loss models.
* This method is additive to allow you to construct complex propagation loss models
  * such as a log distance + jakes model, etc.
* The order in which PropagationLossModels are added may be significant. Some
  * propagation models are dependent of the "txPower" (eg. Nakagami model), and
* are therefore not commutative. The final receive power (excluding receiver gains) are calculated in the order the models are added.

    /*
     * wifiPhy.SetChannel (wifiChannel.Create ());
     /**
     * \returns a new channel
     * Create a channel based on the configuration parameters set previously.
     */

    // Add a non-QoS upper mac, and disable rate control
    NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();

    /**
     * \brief create non QoS-enabled MAC layers for a ns3::WifiNetDevice.
     * This class can create MACs of type ns3::ApWifiMac,
     * ns3::StaWifiMac, and, ns3::AdhocWifiMac, with QosSupported attribute set to False.
     */
    NqosWifiMacHelper NqosWifiMacHelper::Default (void)
    {
        NqosWifiMacHelper helper;
        // We're making non QoS-enabled Wi-Fi MACs here, so we set the necessary attribute. I've carefully positioned this here so that someone who knows what they're doing can override with explicit attributes.
        helper.SetType ("ns3::AdhocWifiMac", "QosSupported", BooleanValue (false));
        return helper;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
/**
* standard the phy standard to configure during installation
* By default, all objects are configured for 802.11a
*/
void WifiHelper::SetStandard (enum WifiPhyStandard standard)
{
    m_standard = standard;
}
/**
** OFDM PHY for the 5 GHz band (Clause 17) */
// WIFI_PHY_STANDARD_80211a,
/** DSSS PHY (Clause 15) and HR/DSSS PHY (Clause 18) */
// WIFI_PHY_STANDARD_80211b,
/** ERP-OFDM PHY (Clause 19, Section 19.5) */
// WIFI_PHY_STANDARD_80211g,
/** OFDM PHY for the 5 GHz band (Clause 17 with 10 MHz channel bandwidth) */
// WIFI_PHY_STANDARD_80211_10MHZ,
/** OFDM PHY for the 5 GHz band (Clause 17 with 5 MHz channel bandwidth) */
// WIFI_PHY_STANDARD_80211_5MHZ,

    wifi.SetRemoteStationManager
("ns3::ConstantRateWifiManager", "DataMode", StringValue
(phyMode),
void WifiHelper::SetRemoteStationManager (std::string type, std::string n0, const AttributeValue &v0, std::string n1, const AttributeValue &v1, std::string n2, const AttributeValue &v2, std::string n3, const AttributeValue &v3, std::string n4, const AttributeValue &v4, std::string n5, const AttributeValue &v5, std::string n6, const AttributeValue &v6, std::string n7, const AttributeValue &v7)
{
    m_stationManager = ObjectFactory ();
    m_stationManager.SetTypeId (type);
    m_stationManager.Set (n0, v0);
    m_stationManager.Set (n1, v1);
    m_stationManager.Set (n2, v2);
    m_stationManager.Set (n3, v3);
    m_stationManager.Set (n4, v4);
    m_stationManager.Set (n5, v5);
    m_stationManager.Set (n6, v6);
    m_stationManager.Set (n7, v7);
}

use constant rates for data and control transmissions

This class uses always the same transmission rate for every packet sent.
Attributes

DataMode: The transmission mode to use for every data packet transmission
Set with class: WifiModeValue
Underlying type: WifiMode
Initial value: OfdmRate6Mbps
Flags: construct write read

ControlMode: The transmission mode to use for every control packet transmission.
Set with class: WifiModeValue
Underlying type: WifiMode
Initial value: OfdmRate6Mbps
Flags: construct write read

Attributes defined in parent class
ns3::WifiRemoteStationManager

IsLowLatency: If true, we attempt to modelize a so-called low-latency device:
a device where decisions about tx parameters can be made on a per-packet basis
and feedback about the transmission of each packet is obtained before sending
the next. Otherwise, we modelize a high-latency device, that is a device
where we cannot update our decision about tx parameters after every packet transmission.
Set with class: BooleanValue
Underlying type: bool
Flags: read
MaxSsrc: The maximum number of retransmission attempts for an RTS.
This value will not have any effect on some rate control algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 7
Flags: construct write read
MaxSlrc: The maximum number of retransmission attempts for a
DATA packet.
This value will not have any effect on some rate control
algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 7
Flags: construct write read
RtsCtsThreshold: If the size of the data packet + LLC header
+ MAC header
+ FCS trailer is bigger than this value, we use an RTS/CTS
handshake before
sending the data, as per IEEE Std. 802.11-2012, Section
9.3.5.
This value will not have any effect on some rate control
algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 2346
Flags: construct write read
FragmentationThreshold: If the size of the data packet +
LLC header + MAC header + FCS trailer is bigger than this
value,
we fragment it such that the size of the fragments are equal
or
smaller than this value, as per IEEE Std. 802.11-2012, Section
9.5.
This value will not have any effect on some rate control
algorithms.
Set with class: ns3::UintegerValue
Underlying type: uint32_t 0:4294967295
Initial value: 2346
Flags: construct write read
NonUnicastMode: Wifi mode used for non-unicast
transmissions.
Set with class: WifiModeValue
Underlying type: WifiMode
Initial value: Invalid-WifiMode
Flags: construct write read
DefaultTxPowerLevel: Default power level to be used for transmissions.
This is the power level that is used by all those WifiManagers that do not implement TX power control.
Set with class: ns3::UintegerValue
Underlying type: uint8_t 0:255
Initial value: 0
Flags: construct write read
No TraceSources are defined for this type.
TraceSources defined in parent class ns3::WifiRemoteStationManager

MacTxRtsFailed: The transmission of a RTS by the MAC layer has failed
MacTxDataFailed: The transmission of a data packet by the MAC layer has failed
MacTxFinalRtsFailed: The transmission of a RTS has exceeded the maximum number of attempts
MacTxFinalDataFailed: The transmission of a data packet has exceeded the maximum number of attempts

TypeId
WifiRemoteStationManager::GetTypeId (void)
{
    static TypeId tid = TypeId
("ns3::WifiRemoteStationManager")
    .SetParent<Object> ()
    .AddAttribute ("IsLowLatency", "If true, we attempt to modelize
a so-called low-latency device: a device" 
    "where decisions about tx parameters can be made on a per-packet basis and
feedback about the"
    "transmission of each packet is obtained before sending the next.
Otherwise, we modelize a "
    "high-latency device, that is a device
where we cannot update
our decision about tx parameters"
" after every packet transmission.",
BooleanValue (true), // this value is
ignored

because there is no setter
MakeBooleanAccessor
(WifiRemoteStationManager::IsLowLatency),
MakeBooleanChecker ()

.AddAttribute ("MaxSsrc", "The maximum number of
retransmission
attempts for an RTS. This value"
" will not have any effect on some rate
control algorithms.",
    UintegerValue (7),
    MakeUintegerAccessor
(WifiRemoteStationManager::m_maxSsrc),
    MakeUintegerChecker<uint32_t> ()

.AddAttribute ("MaxSlrc", "The maximum number of
retransmission attempts for
a DATA packet. This value"
" will not have any effect on some rate
control algorithms.",
    UintegerValue (7),
    MakeUintegerAccessor
(WifiRemoteStationManager::m_maxSlrc),
    MakeUintegerChecker<uint32_t> ()

.AddAttribute ("RtsCtsThreshold", "If the size of the
data packet +
    LLC header + MAC header + FCS trailer
is bigger than "
"this value, we use an RTS/CTS handshake
before sending
the data, as per IEEE Std. 802.11-2007,
Section 9.2.6."
"This value will not have any effect on
some rate control algorithms.",
    UintegerValue (2346),
    MakeUintegerAccessor
(WifiRemoteStationManager::m_rtsCtsThreshold),
MakeUintegerChecker<uint32_t> ()
.AddAttribute ("FragmentationThreshold", "If the size of the data packet + LLC header + MAC header + FCS trailer is bigger than this value, we fragment it such that the size of the fragments are equal or smaller" "than this value, as per IEEE Std. 802.11-2007, Section 9.4." "This value will not have any effect on some rate control algorithms.",
UintegerValue (2346),
MakeUintegerAccessor (&WifiRemoteStationManager::DoSetFragmentationThreshold, &WifiRemoteStationManager::DoGetFragmentationThreshold),
MakeUintegerChecker<uint32_t> ()
.AddAttribute ("NonUnicastMode", "Wifi mode used for non-unicast transmissions.",
WifiModeValue (),
MakeWifiModeAccessor (&WifiRemoteStationManager::m_nonUnicastMode),
MakeWifiModeChecker ()
.AddAttribute ("DefaultTxPowerLevel", "Default power level to be used for transmissions. " "This is the power level that is used by all those WifiManagers that do not" "implement TX power control.",
UintegerValue (0),
MakeUintegerAccessor (&WifiRemoteStationManager::m_defaultTxPowerLevel),
MakeUintegerChecker<uint8_t> ()
.AddTraceSource ("MacTxRtsFailed", "The transmission of a RTS by the MAC layer has failed",
MakeTraceSourceAccessor (&WifiRemoteStationManager::m_macTxRtsFailed))
.AddTraceSource ("MacTxDataFailed",
    "The transmission of a data packet by
the MAC layer has failed",
    MakeTraceSourceAccessor
(&WifiRemoteStationManager::m_macTxDataFailed))
    .AddTraceSource ("MacTxFinalRtsFailed",
    "The transmission of a RTS has exceeded
the maximum number of attempts",
    MakeTraceSourceAccessor
(&WifiRemoteStationManager::m_macTxFinalRtsFailed))
    .AddTraceSource ("MacTxFinalDataFailed",
    "The transmission of a data packet has
exceeded the maximum number of attempts",
    MakeTraceSourceAccessor
(&WifiRemoteStationManager::m_macTxFinalDataFailed))
    ;
    return tid;
}
WifiRemoteStationManager::SetupPhy (Ptr<WifiPhy> phy)
{
    // We need to track our PHY because it is the object that
knows the
    // full set of transmit rates that are supported. We need
    // this in order to find the relevant mandatory rates when
    // choosing a
    // transmit rate for automatic control responses like
    // acknowledgements.
    /*
    // Set it to adhoc mode
    wifiMac.SetType ("ns3::AdhocWifiMac");
    // from regular wifi MAC
    */

base class for all MAC-level wifi objects.

This class encapsulates all the low-level MAC functionality
DCA, EDCA, etc)
and all the high-level MAC functionality (association/disassociation state machines).

Config Paths

ns3::RegularWifiMac is accessible through the following paths with Config::Set and Config::Connect:

/NodeList/[i]/DeviceList/[i]/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac

Attributes

QosSupported: This Boolean attribute is set to enable 802.11e/WMM-style QoS support at this STA
Set with class: BooleanValue
Underlying type: bool
Initial value: false
Flags: construct write read

HtSupported: This Boolean attribute is set to enable 802.11n support at this STA
Set with class: BooleanValue
Underlying type: bool
Initial value: false
Flags: construct write read

CtsToSelfSupported: Use CTS to Self when using a rate that is not in the basic set rate
Set with class: BooleanValue
Underlying type: bool
Initial value: false
Flags: construct write read

DcaTxop: The DcaTxop object
Set with class: ns3::PointerValue
Underlying type: ns3::Ptr< ns3::DcaTxop>
Flags: read

VO_EdcaTxopN: Queue that manages packets belonging to AC_VO access class
Set with class: ns3::PointerValue
Underlying type: ns3::Ptr< ns3::EdcaTxopN>
Flags: read
VI_EdcaTxopN: Queue that manages packets belonging to AC_VI access class
Set with class: ns3::PointerValue
Underlying type: ns3::Ptr< ns3::EdcaTxopN>
Flags: read
BE_EdcaTxopN: Queue that manages packets belonging to AC_BE access class
Set with class: ns3::PointerValue
Underlying type: ns3::Ptr< ns3::EdcaTxopN>
Flags: read
BK_EdcaTxopN: Queue that manages packets belonging to AC_BK access class
Set with class: ns3::PointerValue
Underlying type: ns3::Ptr< ns3::EdcaTxopN>
Flags: read
Attributes defined in parent class ns3::WifiMac

CtsTimeout: When this timeout expires, the RTS/CTS handshake has failed.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +75000.0ns
Flags: construct write read
AckTimeout: When this timeout expires, the DATA/ACK handshake has failed.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +75000.0ns
Flags: construct write read
BasicBlockAckTimeout: When this timeout expires, the BASIC_BLOCK_ACK_REQ/BASIC_BLOCK_ACK handshake has failed.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +281000.0ns
Flags: construct write read
CompressedBlockAckTimeout: When this timeout expires, the COMPRESSED_BLOCK_ACK_REQ/COMPRESSED_BLOCK_ACK handshake has failed.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +107000.0ns
Flags: construct write read
Sifs: The value of the SIFS constant.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +16000.0ns
Flags: construct write read
EifsNoDifs: The value of EIFS-DIFS
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +60000.0ns
Flags: construct write read
Slot: The duration of a Slot.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +9000.0ns
Flags: construct write read
Pifs: The value of the PIFS constant.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +25000.0ns
Flags: construct write read
Rifs: The value of the RIFS constant.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +2000.0ns
Flags: construct write read
MaxPropagationDelay: The maximum propagation delay. Unused for now.
Set with class: ns3::TimeValue
Underlying type: Time – 9223372036854775808.0ns:+9223372036854775807.0ns
Initial value: +3333.0ns
Flags: construct write read
Ssid: The ssid we want to belong to.
Set with class: SsidValue
Underlying type: Ssid
Initial value: default
Flags: construct write read
TraceSources

TxOkHeader: The header of successfully transmitted packet
TxErrHeader: The header of unsuccessfully transmitted packet
TraceSources defined in parent class ns3::WifiMac

MacTx: A packet has been received from higher layers and is being processed in preparation for queueing for transmission.
MacTxDrop: A packet has been dropped in the MAC layer before being queued for transmission.
MacPromiscRx: A packet has been received by this device, has been passed up from the physical layer and is being forwarded up the local protocol stack. This is a promiscuous trace,
MacRx: A packet has been received by this device, has been passed up from the physical layer and is being forwarded up the local protocol stack. This is a non-promiscuous trace,
MacRxDrop: A packet has been dropped in the MAC layer after it has been passed up from the physical layer.

*/

NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, c);

MobilityHelper mobility;
// assigning mobility to nodes
mobility.SetPositionAllocator("ns3::GridPositionAllocator",
"MinX", DoubleValue (0.0),
"MinY", DoubleValue (0.0),
"DeltaX", DoubleValue (distance),
"DeltaY", DoubleValue (distance),
"GridWidth", UintegerValue (5),
"LayoutType", StringValue ("RowFirst"));

/**
 * Set the position allocator which will be used to
 * allocate the initial
 * position of every node initialized during
 * MobilityModel::Install.
 *
 * \param allocator allocate initial node positions
 */
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
/* Calls to MobilityHelper::Install will create an
instance of a matching
 * mobility model for each node.
 */
mobility.Install (c);

/**
 * Layout a collection of nodes according to the
 * current position allocator type.
 *
 * For each node in the provided NodeContainer, this
 * method creates an instance
 * of a ns3::MobilityModel subclass (the type of which was
 * set with
 * MobilityHelper::SetMobilityModel), aggregates it to the
 * node, and sets an
* initial position based on the current position allocator (set through MobilityHelper::SetPositionAllocator).
  *
  * \param container The set of nodes to layout.
  */

Olsrc\* olsr;
// Enable OLSR which is a proactive approach to MANET routing seeks to maintain a constantly updated topology
//This results in a constant overhead of routing traffic, but no initial delay in communication.
/**
 * \brief Helper class that adds OLSR routing to nodes.
 *
 * This class is expected to be used in conjunction with ns3::InternetStackHelper::SetRoutingHelper
 */
Ipv4StaticRoutingHelper staticRouting;

Ipv4ListRoutingHelper list;
list.Add (staticRouting, 0);
list.Add (olsr, 10);

InternetStackHelper internet;
internet.SetRoutingHelper (list); // has effect on the next Install ()
/**
 * \brief aggregate IP/TCP/UDP functionality to existing Nodes.
 *
 * This helper enables pcap and ascii tracing of events in the internet stack
* associated with a node. This is substantially similar to the tracing
* that happens in device helpers, but the important
difference is that, well,
* there is no device. This means that the creation of
output file names will
* change, and also the user-visible methods will not
reference devices and
* therefore the number of trace enable methods is reduced.
*
* Normally we avoid multiple inheritance in ns-3, however,
the classes
* PcapUserHelperForIpv4 and AsciiTraceUserHelperForIpv4 are
* treated as "mixins". A mixin is a self-contained class
that
* encapsulates a general attribute or a set of
functionality that
* may be of interest to many other classes.
*
* This class aggregates instances of these objects, by
default, to each node:
* - ns3::ArpL3Protocol
* - ns3::Ipv4L3Protocol
* - ns3::Icmpv4L4Protocol
* - ns3::UdpL4Protocol
* - a TCP based on the TCP factory provided
* - a PacketSocketFactory
* - Ipv4 routing (a list routing object and a static
routing object)
*/

/**
 * \param routing a new routing helper
 * 
 * Set the routing helper to use during Install. The
 * helper is really an object factory which is used to create
an object of type ns3::Ipv4RoutingProtocol per node. This routing
* object is then associated to a single ns3::Ipv4 object through its
  * ns3::Ipv4::SetRoutingProtocol.
*/
internet.Install (c);
/** *
* For each node in the input container, aggregate implementations of the
* ns3::Ipv4, ns3::Ipv6, ns3::Udp, and, ns3::Tcp classes. The program will assert
  if this method is called on a container with a node that already has
  * an Ipv4 object aggregated to it.
* 
* \param c NodeContainer that holds the set of nodes on
which to install the
  * new stacks.
*/
void
InternetStackHelper::Install (NodeContainer c) const
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
  {
    Install (*i);
  }
}/**
*
This class is a very simple IPv4 address generator. You can think of it
* as a simple local number incremener.
  It has no notion that IP addresses are part of a global address space.

Ipv4AddressHelper ipv4;
If you have a complicated address assignment situation you may want to look at the Ipv4AddressGenerator which does recognize that IP address and network number generation is part of a global problem. Ipv4AddressHelper is a simple class to make simple problems easy to handle.

* We do call into the global address generator to make sure that there are no duplicate addresses generated.

* @see Ipv4AddressGenerator

    NS_LOG_INFO ("Assign IP Addresses.");
    ipv4.SetBase ("10.1.1.0", "255.255.255.0");

/**
 * @brief Set the base network number, network mask and base address.
 * The address helper allocates IP addresses based on a given network number and mask combination along with an initial IP address.
 * For example, if you want to use a /24 prefix with an initial network number of 192.168.1 (corresponding to a mask of 255.255.255.0) and you want to start allocating IP addresses out of that network beginning at 192.168.1.3, you would call
 * SetBase ("192.168.1.0", "255.255.255.0", "0.0.0.3");
 * If you don't care about the initial address it defaults to "0.0.0.1" in which case you can simply use,
* SetBase ("192.168.1.0", "255.255.255.0");
* 
* and the first address generated will be 192.168.1.1.
*
* @param network The Ipv4Address containing the initial network number to use during allocation. The bits outside the network mask are not used.
* @param mask The Ipv4Mask containing one bits in each bit position of the network number.
* @param base An optional Ipv4Address containing the initial address used for IP address allocation. Will be combined (ORed) with the network number to generate the first IP address. Defaults to 0.0.0.1.
* @returns Nothing.
*/
Ipv4InterfaceContainer i = ipv4.Assign (devices);
/** *
* The address helper allocates IP addresses based on a given network number and initial IP address. In order to separate the network number and IP address parts, SetBase was given an initial value and a network mask. The one bits of this mask define the prefix category from which the helper will allocate new network numbers. An initial value for the network numbers was provided in the base parameter of the SetBase method in the bits corresponding to positions in the mask that were 1. An initial value
for the IP address counter was also provided in the base parameter in the
* bits corresponding to positions in the mask that were 0.

* This method gets new addresses for each net device in the container. For
each net device in the container, the helper finds the associated node and
* looks up the Ipv4 interface corresponding to the net device. It then sets
* the Ipv4Address and mask in the interface to the appropriate values. If
* the addresses overflow the number of bits allocated for them by the network
* mask in the SetBase method, the system will NS_ASSERT and halt.
* @param c The NetDeviceContainer holding the collection of net devices we
* are asked to assign Ipv4 addresses to.
* @returns Nothing
* @see SetBase
* @see NewNetwork
*/
TypeId tid = TypeId::LookupByName("ns3::UdpSocketFactory");

/**
 * \param name the name of the requested TypeId
 * \returns the unique id associated with the requested name.
 * This method cannot fail: it will crash if the input name is not a valid TypeId name.
 */
/**
 * \brief a unique identifier for an interface.
 *
This class records a lot of meta-information about a subclass of the Object base class:
* - the base class of the subclass
* - the set of accessible constructors in the subclass
* - the set of 'attributes' accessible in the subclass
*
\internal
* See the discussion in IidManager about hash chaining of TypeId's.
*
*/

Ptr<Socket> recvSink = Socket::CreateSocket (c.Get(sinkNode), tid);
// C IS node container that returns pointer to the node(sinkNode)
/**
The typeid operator allows the type of an object to be determined at run time.

*/

/**
* \brief A low-level Socket API based loosely on the BSD Socket API.
* \ingroup socket
* 
* A few things to keep in mind about this type of socket:
* - it uses ns-3 API constructs such as class ns3::Address instead of
  * C-style structs
* - in contrast to the original BSD socket API, this API is asynchronous:
  * it does not contain blocking calls. Sending and receiving operations
  * must make use of the callbacks provided.
* - It also uses class ns3::Packet as a fancy byte buffer,
data to be passed across the API using an ns-3 Packet
instead of
a raw data pointer.
- Not all of the full POSIX sockets API is supported
*
* Other than that, it tries to stick to the BSD API to make it
easier for those who know the BSD API to use this API.
* More details are provided in the ns-3 tutorial.
*/

//Get
/**
 * \brief Get the Ptr<Node> stored in this container at a given
 * index.
 *
 * Nodes can be retrieved from the container in two ways.
 * First, directly by an index into the container, and second,
 * using an iterator.
 * This method is used in the direct method and is used to retrieve the
 * indexed Ptr<Application>.
 *
 * \code
 * uint32_t nNodes = container.GetN ();
 * for (uint32_t i = 0; i < nNodes; ++i)
 * {
 *   Ptr<Node> p = container.Get (i)
 *     i->method (); // some Node method
 * }
 * \endcode
 *
 * \param i the index of the requested node pointer.
 * \returns the requested node pointer.

Ptr<Socket>
Socket::CreateSocket (Ptr<Node> node, TypeId tid)
{
    NS_LOG_FUNCTION (node << tid);
Ptr<Socket> s;
NS_ASSERT (node != 0);
Ptr<SocketFactory> socketFactory = node->GetObject<SocketFactory> (tid);
NS_ASSERT (socketFactory != 0);
s = socketFactory->CreateSocket ();
NS_ASSERT (s != 0);
return s;
}

*/
InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), 80);
recvSink->Bind (local);
recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));

Ptr<Socket> source = Socket::CreateSocket (c.Get (sourceNode), tid);
InetSocketAddress remote = InetSocketAddress (i.GetAddress (sinkNode, 0), 80);
source->Connect (remote);

if (tracing == true)
{
    AsciiTraceHelper ascii;
    wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("wifi-simple-adhoc-grid.tr");
    wifiPhy.EnablePcap ("wifi-simple-adhoc-grid", devices);
    // Trace routing tables
    Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("wifi-simple-adhoc-grid.routes", std::ios::out);

    /* OutputStreamWrapper class
       * @brief A class encapsulating an STL output stream.
       * 
       * This class wraps a pointer to a C++ std::ostream and provides

    */
reference counting of the object. This class is recommended for users who want to pass output streams in the ns-3 APIs, such as in callbacks or tracing.

This class is motivated by the fact that in C++, copy and assignment of iostreams is forbidden by std::basic_ios<>, because it is not possible to predict the semantics of the stream desired by a user. When writing traced information to a file, the tempting ns-3 idiom is to create a bound callback with an ofstream as the bound object. Unfortunately, this implies a copy construction in order to get the ofstream object into the callback. This operation, as mentioned above, is forbidden by the STL.

Using this class in ns-3 APIs is generally preferable to passing global pointers to ostream objects, or passing a pointer to a stack allocated ostream (which creates object lifetime issues).

One could imagine having this object inherit from stream to get the various overloaded operator<< defined, but we're going to be using a Ptr<OutputStreamWrapper> when passing this object around. In this case, the Ptr<> wouldn't understand the operators and we would have to dereference it to access the underlying object methods. Since we would have to dereference the Ptr<*>, we don't bother and just expect the user to get a saved pointer to an ostream and dereference it him or herself. As in:
```cpp
* 
* \verbatim
* void TraceSink (Ptr<OutputStreamWrapper> streamWrapper,
Ptr<const Packet> packet)
* {
*   std::ostream *stream = streamWrapper->GetStream ();
*   *stream << "got packet" << std::endl;
* }
* \endverbatim
*
* This class uses a basic ns-3 reference counting base
* class but is not
* an ns3::Object with attributes, TypeId, or aggregation.
*/

olsr.PrintRoutingTableAllEvery (Seconds (2), routingStream);

/**
 * \brief prints the routing tables of all nodes at
 * regular intervals
 * \param printInterval the time interval for which the
 * routing table
 * is supposed to be printed.
 * \param stream The output stream object to use
 * 
 * This method calls the PrintRoutingTable() method of the
 * Ipv4RoutingProtocol stored in the Ipv4 object, for all
 * nodes at the
 * specified time interval; the output format is routing
 * protocol-specific.
 */
// To do-- enable an IP-level trace that shows forwarding events only
```
// Give OLSR time to converge-- 30 seconds perhaps
Simulator::Schedule (Seconds (30.0), &GenerateTraffic,
    source, packetSize, numPackets,
    interPacketInterval);

// Output what we are doing
NS_LOG_UNCOND ("Testing from node " << sourceNode
        " to " << sinkNode << " with grid distance " <<
        distance);

Simulator::Stop (Seconds (32.0));
Simulator::Run ();
Simulator::Destroy ();

    return 0;
}