TU Delft Electrical Engineering, Mathematics and Computer Science
ET4394 - Wireless Network

Throughput Fairness in Wifi

Hedi Krishna

April 2015

1 Introduction

Wifi user, as any other wireless communication system, shares limited wireless spectrum with other users. Theoretically, the more users are located in the same area, the more inferences; thus, the lower user's throughput. In IEEE 802.11b and g, this problem becomes more evident because its physical layer uses Direct Sequence Spread Spectrum (DSSS). In DSSS, one frequency channel is occupied by many users. Power and interference becomes important commodity that directly affects throughput. This become unfair for a user located far from Access Point (AP), as they will experience lower signal strength and get lower throughput, while users closer to the access point get higher throughput. On the other hand, 802.11 standards employs a sophisticated scheduling and the nature of TCP traffic itself allows some level of traffic control.

In this experiment, Wifi network with single Access Point and varying number of nodes will be simulated with NS3. The effect of unfairness based on client node's location is investigated. Further, the effect of DCF in mitigating Wifi throughput unfairness will be explored.

2 Background

In a Wifi network, user's node stations are distributed in an area with different distance to Access Point. Naturally, due to radio propagation loss, users located closer to AP will get higher RSSI (Received Signal Strength Index). Other propagation phenomena such as reflection, scattering and diffraction add more variability in users RSSI.

Throughput unfairness happens when some users get higher throughput than other users. One of the main reasons of unfairness in Wifi is 'capture effect'. When a collision happens, strong signal from nearby node will always 'win' and overwhelm weaker signal from a remote transmitter. Capture effect can be minimized using transmit power control. However, because it is a complicated technique, in 802.11 family standard only 802.11a and 802.11n apply power control.

Kemerlis et al. [1], found that there is unfairness in throughput among Wifi hosts with different signal strength when TCP is used. Their experiment performed in live testbed, using one AP and two hosts with different RSSI. They observed that throughput obtained by client with lower signal strength is lower than client with higher signal quality. They also found that this problem can be alleviated by enabling RTS/CTS.

RTS/CTS (Request to Send / Clear to Send) is a method in wireless networking MAC layer to reduce collision by performing extra handshake before sending actual data. RTS/CTS is employed in IEEE 802.11 Distributed Coordination Function (DCF) to solve hidden and exposed node problem which commonly occur in wireless network with multiple user.

3 Simulation Setup

In this experiment, a simulation with NS3 is performed. NS3 is an open source network simulation tool based on C++ and phyton. The model uses Wifi 802.11b model with DSSS modulation and 11 Mbps transfer rate.

```
std::string phyMode ("DsssRate11Mbps");
Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
    StringValue (phyMode));
```

To simulate propagation loss, log distance propagation loss model is used. Log distance model calculates path loss using following formula:

$$L = L_0 + 10 \times n \times log_{10} \left(\frac{d}{d_0} \right) \tag{3.1}$$

Where L_0, n, d and d_0 are path loss reference distance, path loss distance exponent, distance and reference distance respectively. In the simulation, default value of L_0, n and d_0 are used, and same for all nodes. Hence, the only deciding factor for loss is distance between AP and node.

The Access Point placed in the corner of 100x100 square or cartesian coordinate (0,0). While nodes are randomly placed with uniform distribution from coordinate (10,10) to (60,60). With this setting, hidden nodes problem does not exist because the AP are located in the same relative position from all nodes. This is done purposely to eliminate unnecessary element in the experiment. All nodes are assumed to be in static position during the simulation.

```
ObjectFactory pos;
pos.SetTypeId ("ns3::RandomRectanglePositionAllocator");
pos.Set ("X", StringValue ("ns3::UniformRandomVariable[Min=10.0|Max=60.0]"));
pos.Set ("Y", StringValue ("ns3::UniformRandomVariable[Min=10.0|Max=60.0]"));
Ptr<PositionAllocator> positionAlloc = pos.Create
    ()->GetObject<PositionAllocator> ();
```

The AP acts as a transmission sink. All hosts try to send the data as much as possible to the sink during 10 seconds simulation time. AP communicates with each nodes with different port.

Traffics are generated with OnOff application which generates packet with On/Off pattern. 'On' and 'Off' states change alternately as stated in the attribute. Here, `OnTime` is equal to simulation time, while `OffTime` is 0. This means that the traffics are generated in each of the nodes constantly. Each packet is 1024 bytes long, generated with 5Mbps datarate.

```
OnOffHelper onoff ("ns3::TcpSocketFactory",Ipv4Address::GetAny ());
onoff.SetAttribute ("OnTime", StringValue
    ("ns3::ConstantRandomVariable[Constant=10]"));
onoff.SetAttribute ("OffTime", StringValue
    ("ns3::ConstantRandomVariable[Constant=0]"));
onoff.SetAttribute ("PacketSize", UintegerValue (payloadSize));
onoff.SetAttribute ("DataRate", StringValue ("5Mbps"));
```

TCP packets are used in this simulation, instead of UDP. In UDP, for a system with large number of hosts, packet losses is very high. It happens because UDP is a best effort protocol. Therefore, throughput calculation is biased. In TCP, three-way handshake is performed before actual data sent; thus reducing packet collision.

To simulate transmission with or without RTS/CTS, `RtsCtsThreshold` parameter is used. To simulate transmission with RTS/CTS, `RtsCtsThreshold` is set to 150. By doing so, every frame larger than 150 bytes is preceded with RTS/CTS. For simulation of system without RTS/CTS, we set the threshold to 1500. Since the maximum payload is 1024 bytes, no packet will use RTS/CTS.

```
std::string rtsCts ("150");
```

```
  cmd.AddValue ("rtsCts", "RTS/CTS threshold", rtsCts);
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
      StringValue (rtsCts));
```

To measure throughput, FlowMonitor is used. FlowMonitor is a built-in class in ns3 used to monitor and report packet flows during simulation. Here, throughput is defined as the sum of data received by the sink divided by transmission time. Transmission time is the difference between timestamp of last packet and timestamp of first packet.

```
  for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
      stats.begin (); i != stats.end (); ++i)
  {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
    throughput[i->first] = i->second.rxBytes * 8.0 /
        (i->second.timeLastRxPacket.GetSeconds() -
        i->second.timeFirstTxPacket.GetSeconds())/1024/1024;
  }
```

Seeding is used in the script to simulate uncertainty in the system. Time is used as seed, so it will be different each time simulation is run.

```
time(time_t);
RngSeedManager::SetSeed(timev);
RngSeedManager::SetRun (7);
```

Each simulation instance generates nodes' throughput, received bytes and distances from AP. The simulation is run multiple times with different number of nodes, from 1 to 50. For each number of nodes, the simulation is repeated 10 times. Since nodes position is randomly generated, multiple iterations are conducted and averaged to reduce variance in the result. This process is done automatically by using shell script.

```
tempfile=../log/looptcp3.$(date +%m%d%Y_%H%M%S)
for rts in 150 1500
do
 for node in ‘seq 1 50‘;
 do
  for iteration in ‘seq 1 10‘;
  do
   ../waf --run "wifitcpm3 --n=$node --rtsCts=$rts" >> $tempfile
  done
 done
done
```

This shell script produces text file containing the simulation result and puts it in log directory. This log file is then exported and analysed using Microsoft Excel.

4 Result

The simulation uses NS3 version 3.21 run on Ubuntu Linux 14.01 running on Intel Core i3 with 4 GB of RAM. In total, there are 500 instance of simulations, with 25500 nodes. This simulation produces 1.15 MB log file (attached with this report).

Figure 1 shows average throughput for system different number of nodes. As expected, average throughput per node is smaller for higher number nodes. Total throughput increases slightly when more nodes are added, but reach saturation for number of nodes larger than 20.

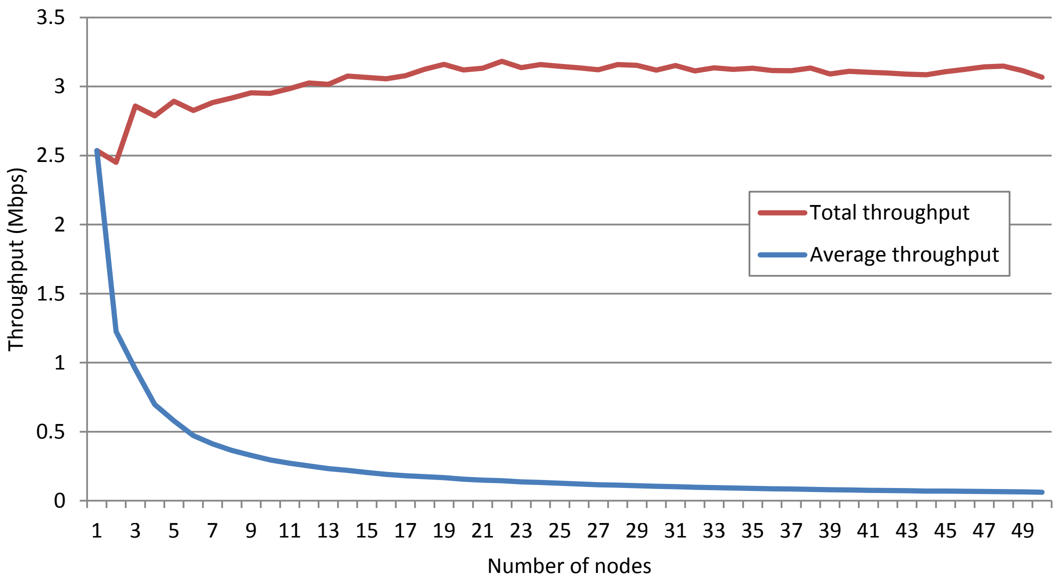

Figure 1: *Throughput for various number of nodes*

In the simulation, nodes distance to AP is obtained. Using equation 3.1, AP's received signal power is calculated. Figure 2 shows graph of throughput versus received signal power for 30, 40 and 50 nodes.

Figure 3 shows the details in scattered plot. Here we can see that there is a sudden decrease in throughput for Rx power around -78 dBm. However, for Rx power larger than -78 dBm, the throughput difference is trivial. Nodes with lower Rx power can get throughput as much as one with higher Rx power. Also, throughput drop at -78 dBm happens for all number of nodes. It possibly happens because receiver sensitivity is around -80 dBm. Thus, it is no longer able to demodulate signal below -78 dBm.

Figure 4 shows the standard deviation of throughput per nodes for system with and

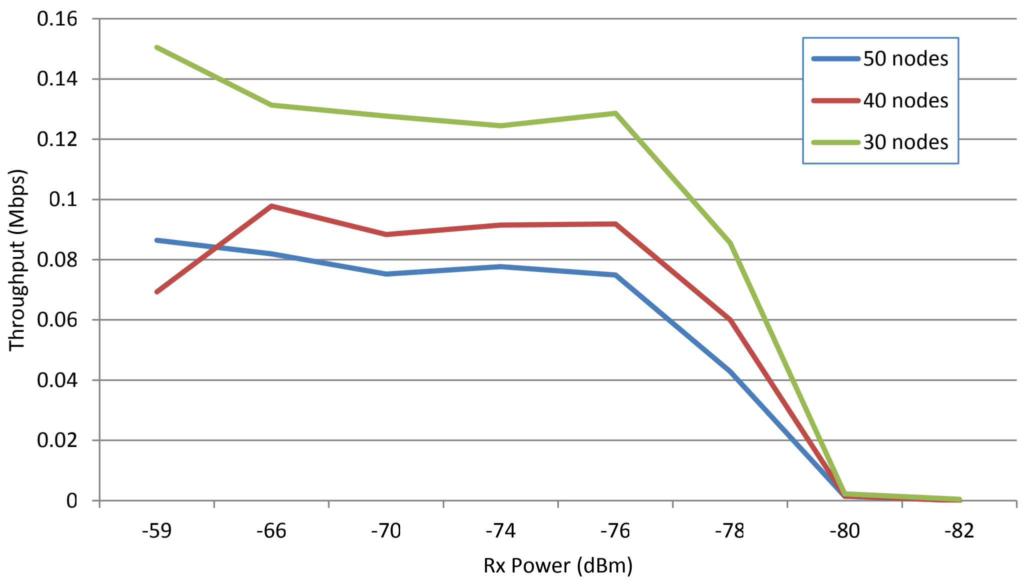

Figure 2: *Rx Power vs Average Throughput*

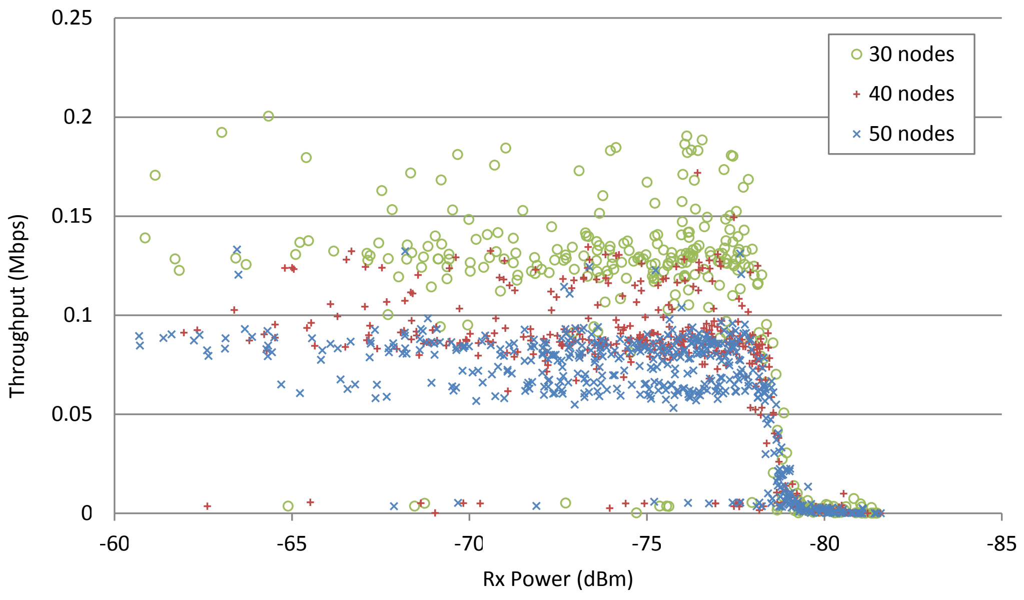

Figure 3: *Scatter plot Rx Power vs Throughput*

without RTS/CTS. System with RTS/CTS has slightly lower standard deviation. However, the difference is trivial and most likely happens because system with RTS/CTS in general has lower throughput.

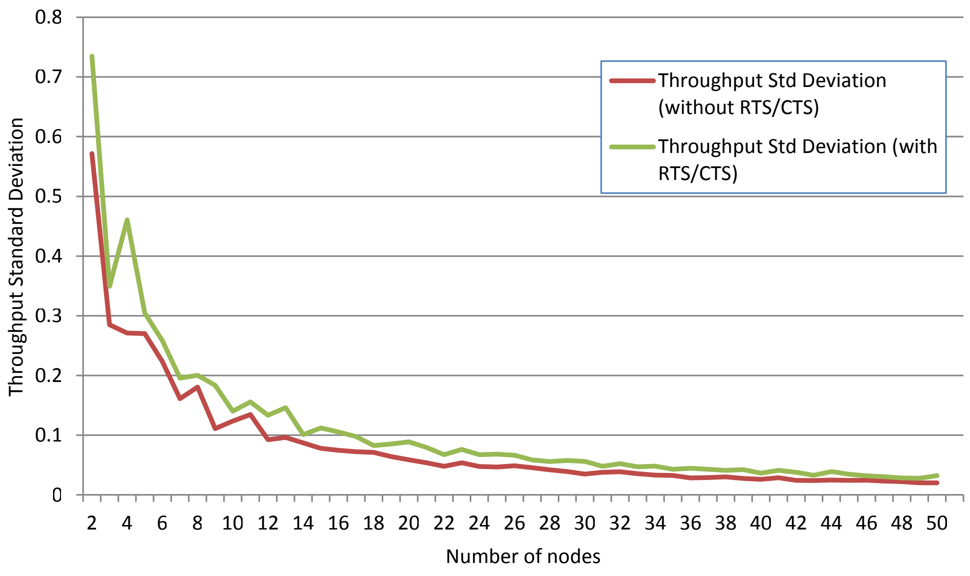

Figure 4: *RTS/CTS effect to throughput standard deviation*

Figure 5 shows how RTS/CTS are lowering the throughput. This happens because RTS/CTS adds more overhead to the system; thus, lowering throughput.

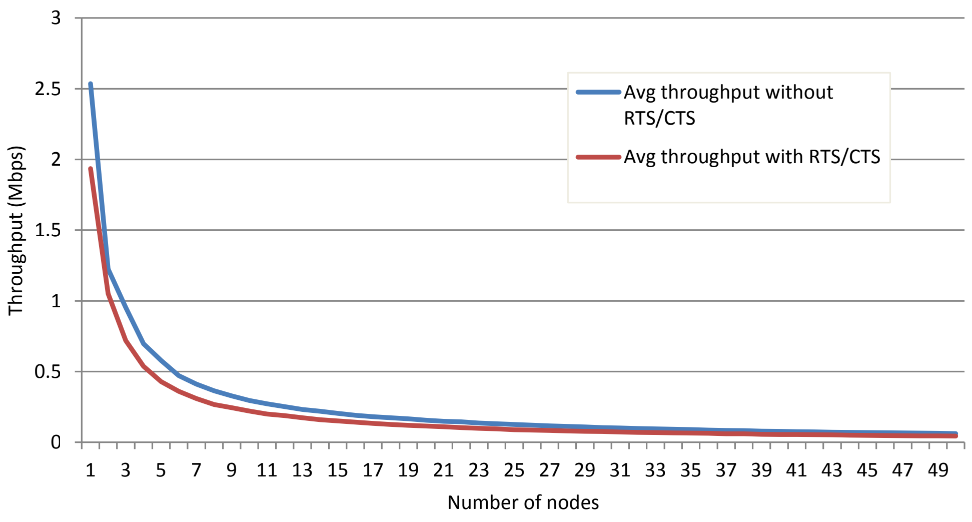

Figure 5: *RTS/CTS effect to average throughput*

Figure 6 compares system with and without RTS/CTS, both with 30 nodes. Here we can see that the critical Rx power, at which throughput begins to drop to zero, is same for both system. It means that RTS/CTS does not increase fairness in Wifi, and it doesn't alleviate capture effect problem.

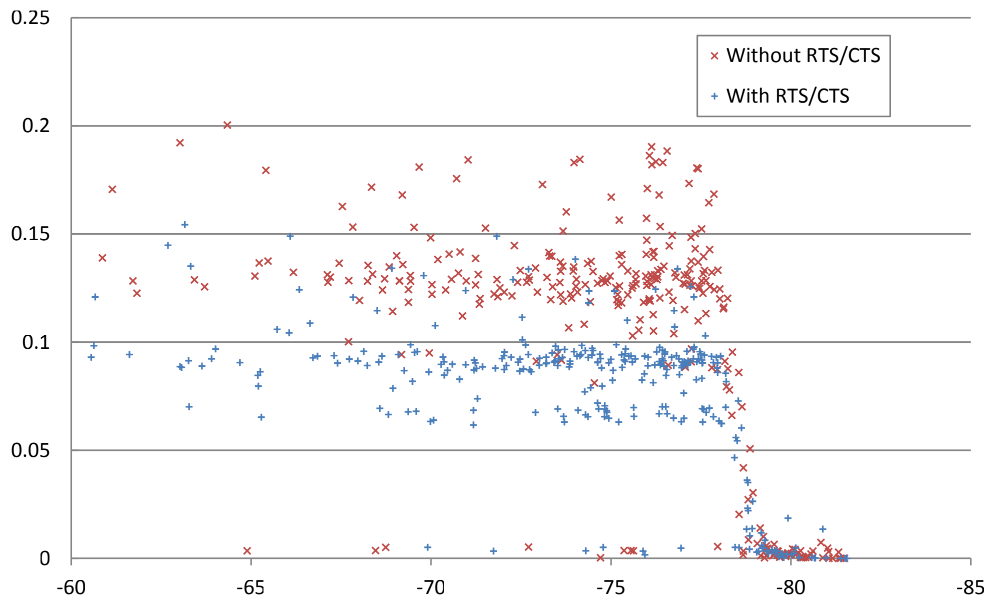

Figure 6: *Power vs Throughput, With and Without RTS/CTS (nodes=30)*

One might expect that RCTS/CTS will increases throughput for higher number of nodes, like the result obtained by Bianchi [2]. However, this is not the case in this simulation, for two reasons.

1. TCP itself has three way handshake mechanisms, it has already reduced collision. Therefore, RTS/CTS does not increase average throughput for system with large number of nodes.

2. In this experiment sink AP located at the corner of a room (coordinate 0,0), while the nodes are spread uniformly between coordinate (0,0) and (100,100). With this setting, hidden nodes problem does not exist because the AP are located in the same relative position from all nodes. Exposed node problem also does not exist because there is only one sink. Thus, RTS/CTS lost its advantage in increasing total throughput.

There is an interesting finding related to total throughput. This simulation use 11 Mbps transfer mode. Interestingly, maximum total throughput (from nodes to AP) achieved by the system is around 3.2 Mbps. This might happens because Wifi is half duplex; thus, uplink throughput is half than the transfer mode. Also, there are overhead bits, like RTS/CTS and TCP handshakes, which are using usable channel, but are not calculated as data bits by FlowMonitor. Same result observed by Kamerlis [1].

5 Conclusion

In this experiment, throughput fairness in Wifi is simulated. The simulation is run with different number of nodes. Throughput difference between nodes is analysed. There are several conclusion for this experiment.

1. In normal usage of Wifi 802.11b, throughput unfairness does not exist. It only happens when difference in distance (RSSI) is very high. Nodes with Rx power higher than -78 dBm, or located within 60 meter from AP (using log distance loss model), get comparable throughput.

2. Because the nature of shared medium, the more the user, the smaller average throughput.

3. IEEE DCF is intended to solve hidden and exposed node. In a system where hidden and exposed node problem is nonexistent, instead of giving any benefit, it is lowering the throughput.

Regarding uses of RTS/CTS to combat unfairness in Wifi, there is a difference between our result and Kamerlis'. However, it is important to note that Kamerlis model only use two nodes with extreme difference in Rx power (-2 dBm and -96 dBm). While our model is more realistic, using multiple nodes with smaller difference in Rx power (between -60 dBm and -82 dBm).

References

[1] Kemerlis, Vasileios P., et al. "Throughput unfairness in TCP over WiFi." WONS 2006: Third Annual Conference on Wireless On-demand Network Systems and Services. 2006.

[2] Bianchi, Giuseppe. "Performance analysis of the IEEE 802.11 distributed coordination function." Selected Areas in Communications, IEEE Journal on 18.3 (2000): 535-547.

[3] https://www.nsnam.org/

[4] http://www.rcmodelreviews.com/fhss_vs_dsss.shtml

Appendices

NS3 Code

```
/* -*- Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 The Boeing Company
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 */

 /*
  Hedi Krishna, EEMCS TU Delft, April 2015
  https://github.com/hedi02/WirelessNetworking
  This script runs Wifi transmission simulation for n number of nodes
  Throughput for each node is then calculated
  Based on wifi-simple-infra.cc
  */

#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/flow-monitor-module.h"
#include "ns3/applications-module.h"

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

using namespace ns3;
```

```
NS_LOG_COMPONENT_DEFINE ("WifiSimpleInfra");

int main (int argc, char *argv[])
{
  double simulationTime = 10; //seconds
  //===========================
  time_t timev;
  time(&timev);
  RngSeedManager::SetSeed(timev);
  RngSeedManager::SetRun (7);
  //===========================

  std::string phyMode ("DsssRate11Mbps");
  std::string rtsCts ("150");

  bool verbose = false;
  //double rss = -80; // -dBm
  uint32_t n=25;
  uint32_t payloadSize = 1024;
  CommandLine cmd;

  cmd.AddValue ("n", "number of nodes", n);
  cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components",
      verbose);
  cmd.AddValue ("rtsCts", "RTS/CTS threshold", rtsCts);

  cmd.Parse (argc, argv);

  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
      StringValue ("2200"));
  // turn off RTS/CTS for frames below 'rtsCts' bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
      StringValue (rtsCts));
  // Fix non-unicast data rate to be the same as that of unicast
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
                      StringValue (phyMode));

  NodeContainer apContainer;
  NodeContainer staContainer;
  apContainer.Create (1);
  staContainer.Create (n);

  // The below set of helpers will help us to put together the wifi NICs we
      want
  WifiHelper wifi;
  if (verbose)
    {
      wifi.EnableLogComponents (); // Turn on all Wifi logging
    }
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

  YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
  // This is one parameter that matters when using FixedRssLossModel
  // set it to zero; otherwise, gain will be added
  wifiPhy.Set ("RxGain", DoubleValue (0) );
  // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
  wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  // The below FixedRssLossModel will cause the rss to be fixed regardless
  // of the distance between the two stations, and the transmit power
  wifiChannel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel");
  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 sta.
  wifiMac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "ActiveProbing", BooleanValue (false));
  NetDeviceContainer staDevice = wifi.Install (wifiPhy, wifiMac, staContainer);
  //NetDeviceContainer devices = staDevice;

  // setup ap.
  wifiMac.SetType ("ns3::ApWifiMac",
                   "Ssid", SsidValue (ssid));
  NetDeviceContainer apDevice = wifi.Install (wifiPhy, wifiMac,
      apContainer.Get (0));
  //devices.Add (apDevice);


  // Note that with FixedRssLossModel, the positions below are not
  // used for received signal strength.
  MobilityHelper mobility, mobilityAp;

  ObjectFactory pos;
  pos.SetTypeId ("ns3::RandomRectanglePositionAllocator");
  pos.Set ("X", StringValue ("ns3::UniformRandomVariable[Min=10.0|Max=60.0]"));
  pos.Set ("Y", StringValue ("ns3::UniformRandomVariable[Min=10.0|Max=60.0]"));
  Ptr<PositionAllocator> positionAlloc = pos.Create
      ()->GetObject<PositionAllocator> ();

  Ptr<ListPositionAllocator> positionAllocAp =
      CreateObject<ListPositionAllocator> ();
  positionAllocAp->Add (Vector (0.0, 0.0, 0.0));
  mobilityAp.SetPositionAllocator (positionAllocAp);
  mobilityAp.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobilityAp.Install (apContainer);

  mobility.SetPositionAllocator (positionAlloc);
  mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  mobility.Install (staContainer);

  InternetStackHelper internet;
  internet.Install (apContainer);
  internet.Install (staContainer);

  Ipv4AddressHelper ipv4;
  NS_LOG_INFO ("Assign IP Addresses.");
  ipv4.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer iap = ipv4.Assign (apDevice);
  Ipv4InterfaceContainer i = ipv4.Assign (staDevice);

  //port number, given in array
  uint16_t port[n];

  ApplicationContainer apps[n], sinkApp[n];

   for( uint16_t a = 0; a < n; a = a + 1 )
      {
     port[a]=8000+a;
         Address apLocalAddress (InetSocketAddress (Ipv4Address::GetAny (),
             port[a]));
         PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
             apLocalAddress);
         sinkApp[a] = packetSinkHelper.Install (apContainer.Get (0));

         sinkApp[a].Start (Seconds (0.0));
         sinkApp[a].Stop (Seconds (simulationTime+1));

         OnOffHelper onoff ("ns3::TcpSocketFactory",Ipv4Address::GetAny ());
         onoff.SetAttribute ("OnTime", StringValue
             ("ns3::ConstantRandomVariable[Constant=10]"));
         onoff.SetAttribute ("OffTime", StringValue
             ("ns3::ConstantRandomVariable[Constant=0]"));
         onoff.SetAttribute ("PacketSize", UintegerValue (payloadSize));
         onoff.SetAttribute ("DataRate", StringValue ("5Mbps")); //bit/s
```

```
        AddressValue remoteAddress (InetSocketAddress (iap.GetAddress (0),
            port[a]));
        onoff.SetAttribute ("Remote", remoteAddress);

        apps[a].Add (onoff.Install (staContainer.Get (a)));
        apps[a].Start (Seconds (1.0));
        apps[a].Stop (Seconds (simulationTime+1));
 }


FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon.InstallAll();

wifiPhy.EnablePcap ("wifitcp", apDevice);
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

Simulator::Stop (Seconds (simulationTime+1));
Simulator::Run ();

monitor->CheckForLostPackets ();
double tot=0;
double totsq=0;
double variance=0;
std::cout << std::fixed;

double throughput[2*n]; //for every node, there are 2 flows in TCP
int psent=0;
int preceived=0;

Vector pos2;

Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>
    (flowmon.GetClassifier ());
std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();
for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i =
    stats.begin (); i != stats.end (); ++i)
  {
   Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
        //std::cout << "Flow " << i->first << " (" << t.sourceAddress << " ->
            " << t.destinationAddress << ")\t";
        //std::cout << " Tx Bytes: " << i->second.txBytes << "\t";
        //std::cout << " Rx Bytes: " << i->second.rxBytes << "\t";
       throughput[i->first] = i->second.rxBytes * 8.0 /
           (i->second.timeLastRxPacket.GetSeconds() -
           i->second.timeFirstTxPacket.GetSeconds())/1024/1024;
         //std::cout << " Throughput: " << throughput[i->first] << " Mbps\n";

    if (t.destinationAddress=="10.1.1.1")
```

```
       {
        int ncount=(i->first)-1;
        Ptr<MobilityModel> mob =
            staContainer.Get(ncount)->GetObject<MobilityModel>();
        pos2 = mob->GetPosition ();

        std::cout << t.sourceAddress <<"\t";
        std::cout << n << "\t" << rtsCts <<"\t";
        std::cout << i->second.txBytes << "\t";
        std::cout << throughput[i->first] << "\t";
        std::cout << pow(pow(pos2.x,2)+pow(pos2.y,2),0.5)<< "\t\n"; //calculate
            node distance to AP

        tot=tot+throughput[i->first];
        totsq=totsq+pow(throughput[i->first],2);
        psent=psent+i->second.txBytes;
        preceived=preceived+i->second.rxBytes;
       }
     }

  //std::cout << tot <<"\t";
  //std::cout << tot/n <<"\t";
  //std::cout << psent <<"\t";
  //std::cout << preceived <<"\t";
  //std::cout << preceived/psent << "\n";

  //std::cout << n << "\t" << throughput/n << "\t" << rtsCts <<"\n";
  //std::cout << "Total throughput: " << tot <<"\n";
  //std::cout << n << "\t" << rtsCts <<"\t";
  std::cout << "Avg: " << tot/n <<"\t";
  std::cout << "Tot: " << tot <<"\t";
  variance=totsq/n-pow(tot/n,2);
  //std::cout << "Variance: " << variance <<"\t";
  std::cout << "Var: " << pow(variance,0.5)/tot/n <<"\t";
  std::cout << "loss: " << psent-preceived <<"\t";
  std::cout << "sent: " << psent <<"\n";
  //std::cout << "Packet received: " << preceived <<"\n";

  Simulator::Destroy ();
  return 0;
}
```