NS3 Project Report

Qiaoqiao Li
4508300
ET4394 Wireless Networking
Embedded Systems
TU Delft

April 29, 2016

1 Introduction

802.11 and 802.11x refers to a family of specifications developed by the IEEE for wireless LAN (WLAN) technology. 802.11 specifies an over-the-air interface between a wireless client and a base station or between two wireless clients. The IEEE accepted the specification in 1997. In particular, 802.11b (also referred to as 802.11 High Rate or WiFi) is an extension to 802.11 that applies to wireless LANS and provides 11 Mbps transmission (with a fall-back to 5.5, 2 and 1Mbps) in the 2.4 GHz band. 802.11b uses only DSSS (Direct Sequence Spread Spectrum) modulation technique. 802.11b was a 1999 ratification to the original 802.11 standard, allowing wireless functionality comparable to Ethernet.

This project uses NS3 to test IEEE 802.11b throughput versus the number of WiFi stations. NS3 is an open source network simulation tool based on C++ and phyton. In this project, a model with 50 WiFi nodes is setup in an infrastructure mode (1 Access Point). Five senarios are generated with the number of WiFi stations increasing from 1 to 50, namely throughput versus application data rate, throughput versus payload size, throughput versus variant of TCP, and throughput versus physical layer transmission rate, as well as throughput versus with/without RTS/CTS. RTS/CTS (Request to Send / Clear to Send) is the optional mechanism used by the 802.11 wireless networking protocol to reduce frame collisions introduced by the hidden node problem [1]. Besides, users can also choose the simulation time and whether to enable pcap tracing through command line.

2 Simulation Setup

This model uses TCP packets instead of UDP. TCP uses 3-way handshake, congestion control, flow control and other mechanisms to make sure the reliable transmission. UDP is faster than TCP because there is no form of flow control or error correction. TCP is mostly used in cases where the packet loss is more serious than packet delay.

First of all, fragmentation optimization [2] is not examined in this model, so the fragmentation threshold is set to large enough as a value of 999999 bytes to turn off fragmentation.

```
Config::SetDefault ("ns3::WifiRemoteStationManager
::FragmentationThreshold", StringValue ("999999"));
```

Next, the effects of enabling and disabling RTS/CTS on throughput can be seen by uncommenting the first and the second option, respectively. The RTS/CTS threshold is set to 1000 bytes because we will experiment later on with payload size from 1024 bytes up to 10024 bytes with a step of 500 bytes.

```
Config::SetDefault ("ns3::WifiRemoteStationManager
::RtsCtsThreshold", StringValue ("999999"));

//Config::SetDefault ("ns3::WifiRemoteStationManager
::RtsCtsThreshold", StringValue ("1000"));
```

Subsequently, **YansWifiChannelHelper** is used to set up the wifi channel, with **ConstantSpeedPropagationDelayModel** and **FriisPropagationLossModel**. Physical Layer is set up using **YansWifiPhyHelper** with transmission and detection power specified and error rate model added using **YansErrorRateModel**.

Once n Wifi stations and one access point are created and configured. We are able to set up the mobility model using **MobilityHelper**. We first arrange both our stations and access point on an initial grid by setting some attributes controlling the position allocator functionality. We then choose **RandomDirection2dMobilityModel** for stations which makes them move in a random direction at a constant speed around inside a bounding box. Whenever a station hits the boundaries, it pauses with a constant delay and move again with a new direction. In addition, we want the access point remain in a fixed position during the simulation. We accomplish this by setting the mobility model for this node to be the **ConstantPositionMobilityModel**.

```
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::
   RandomDirection2dMobilityModel",
"Bounds", RectangleValue(Rectangle(-500, 500, -500, 500)),
"Speed", StringValue
("ns3::ConstantRandomVariable[Constant=2]"),
"Pause", StringValue
("ns3::ConstantRandomVariable[Constant=0.2]"));

mobility.Install (wifiStaNodes);

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

The next usual step is to install protocol stacks using **InternetStackHelper**, assign IP addresses to our device interfaces using **Ipv4AddressHelper** and populate routing table using **Ipv4GlobalRoutingHelper**.

Now, we start to install TCP receiver on the access point using **PacketSinkHelper**:

```
uint16_t port = 50000;
Address apLocalAddress (InetSocketAddress
(Ipv4Address::GetAny (), port));
PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
apLocalAddress);

ApplicationContainer sinkApp = packetSinkHelper.Install
(wifiApNode.Get (0));
sinkApp.Start (Seconds (0.0));
sinkApp.Stop (Seconds (simulationTime + 1));
```

On the transmitter side, we use an **OnOffHelper** to generate the TCP traffic with transport layer payload size and application layer data rate configurable.

```
OnOffHelper onoff ("ns3::TcpSocketFactory",
Ipv4Address::GetAny ());
onoff.SetAttribute ("OnTime",  StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
onoff.SetAttribute ("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
onoff.SetAttribute
("PacketSize", UintegerValue (payloadSize));
```

```
onoff.SetAttribute
("DataRate", DataRateValue (DataRate (dataRate)));

AddressValue remoteAddress (InetSocketAddress
(ApInterface.GetAddress (0), port));
onoff.SetAttribute ("Remote", remoteAddress);

ApplicationContainer apps;
apps.Add (onoff.Install (wifiStaNodes));
apps.Start (Seconds (1.0));
apps.Stop (Seconds (simulationTime + 1))
```

We now have n WiFi stations sending packets to the access point. The last thing is to calculate the throughput on the access point, which is the sum of data received by the sink divided by simulation time:

```
totalPacketsThrough = DynamicCast<PacketSink>
(sinkApp.Get (0))-\textgreater GetTotalRx ();
throughput = totalPacketsThrough * 8
/ (simulationTime * 1000000.0);
```

3 Results and Analysis

Users can take advantage of 4 separte codes to generate output directly for these 5 senarios, namely, dataRate.cc, payloadSize.cc, phyRate.cc and tcpVariant.cc. Correspondingly, outputs are stored in four *.dat files. Tableau Public was used to plot and analyze these outputs.

3.1 Data rate

We experiment with 8 different data rates from 100Mbps to 800Mbps with a step of 100Mbps. Figure 1 shows the line graph of thoughput versus the number of WiFi stations for each data rate. Darker blue stands for higher data rates. It is obvious that throughputs versus varying number of stations under different data rates share the same pattern. That is the throughput first increases from about 4.1 Mbps to its maximum value above 4.5Mbps at around 5-10 stations and gradually decreases to about 3.9Mbps at more than 22 stations. In other words, changing data rate from 100Mbps to 800Mbps plays an insignificant role in throughput versus number of stations. In figure 2, each column has a boxplot under a certain data rate. 50 blue circles in each column correspond to 50 throughput values with the number of stations increasing from 1 to 50. We can observe that these boxplots have similar median and standard deviation of throughput under all data rates. In other

Figure 1: Throughput vs Data Rate with WiFi stations varying from 1 to 50

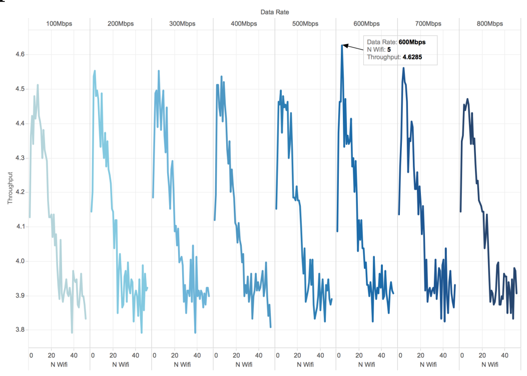
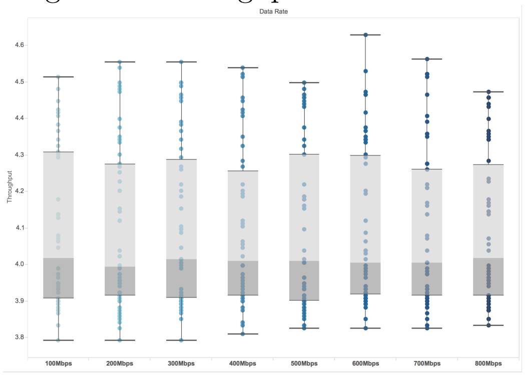


Figure 2: Throughput vs Data Rate



words, the pattern of thoughput versus numeber of stations is not sensitive to the change of data rates between 100Mbps and 800Mbps. Nevertheless, the largest throughput (4.6285Mbps) was found with a data rate of 600Mbps and 5 WiFi stations.

3.2 Payload Size and RTS/CTS

Both figure 3 and figure 4 plot the experimental results for throughput versus payload size increasing between 1024 bytes to 10240 bytes with a step of 500 bytes. In total, there are 50 lines in each figure. Darker black line represents throughput versus payload size with more WiFi stations. There are some similar features in these two figures. First, throughput is smaller with more stations since darker black lines locate in lower positions. Second, throughput increases with very significant fluctuation at the beginning and saturates at the end when payload size increases. Lastly, throughput is more sensitive to the number of stations when payload size are larger since these lines are

Figure 3: Throughput vs payload size without RTS/CTS

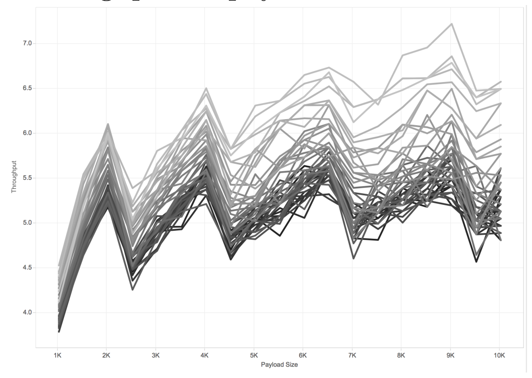
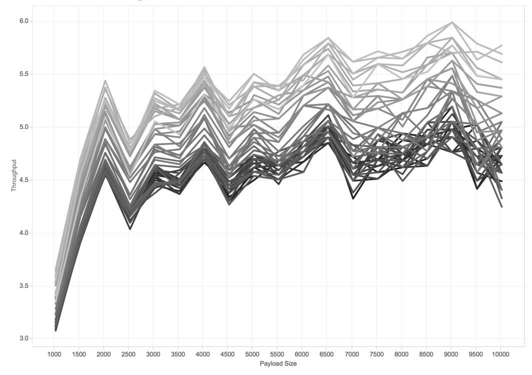


Figure 4: Throughput vs Data Rate with RTS/CTS



more dispersed at the end. On the other hand, there are also differences between these two figures. An obvious finding is that the average througput without RTS/CTS is larger than that with RTS/CTS. This is because when the RTS/CTS handshake is enabled, additional time is needed for ackowlegements. Otherwise the data frame gets sent immediately. Another finding is that throughput saturates earlier in figure 4 with RTS/CTS enabled.

3.3 Physical Layer Transimission Rate

There are only four possible options of transmission rate in physical layer of 802.11b. They are 11Mbps, 5.5Mbps, 2Mbps and 1Mbps. The 802.11b standard has a maximum raw transmission data rate of 11 Mbps. Slower data rates are adopted in order to improve resiliency by improving error correction [3]. In figure 5, darker red represents higher transmission rates. It is obvious that higher transmision rates result in significantly higher throughputs. And in general, throughput decreases with slight fluctuations when more WiFi

Figure 5: Throughput vs physical rate with WiFi stations varying from 1 to 50

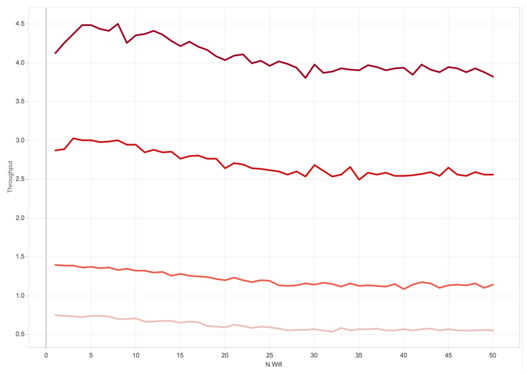
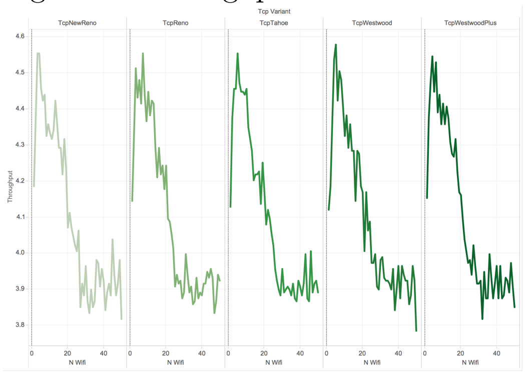


Figure 6: Throughput vs TCP variant

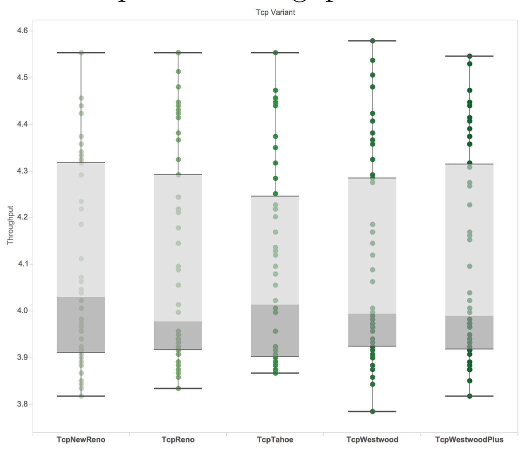


stations share the same channel.

3.4 TCP Variant

The influence on throughput of TCP variants, i.e. different congestion control algorithm to use, is shown in figure 6. Similar to application data rates, using all these algorithms, througput first increases sharply to its maximum value and then decreases gradually and finally fluctuates slightly around a relatively low value with large number of stations. The differences across these algorithms are small. From boxplot in figure 7, we can see that the maximum median value of throughput is found by using TcpNewReno, closely followed by TcpTahoe, TcpWestwood and TcpWestwoodPlus. TcpReno has the smallest median value of throughput. What's more, TcpTahoe has the smallest deviation, while TcpNewReno has the largest standard deviation.

Figure 7: Boxplot of throughput vs TCP variant



4 Conclusion

Here are some conclusions from the experimental results of our simulation:

1. The obtained throughout on the access point with varying number of WiFi stations first increases to a maximum value at the beginning and then decreases gradually and at the end fluctuates around a relatively low value with large number of stations. This is at the beginning, the shared medium channel is not full and is capable of transmitting more packets. When the number of staions become too large, congestion occurs and many packets are lost, thus reducing the throughput significantly.

2. Different application data rates between 100Mbps and 800Mbps have minor differences on the pattern of throughput versus varying number of WiFi stations.

3. The general trend of throughput versus increasing payload size is upward. However, there are sharp increases and decreases in throughput during payload size rises. When payload size gets larger, variance in throughput is larger across varying number of stations.

4. RTS/CTS reduces throughput obtained by the access point because of additional handshake process.

5. Larger physical layer transmission rate results in larger throughput.

6. Using different congestion control algorithms has little effect on improving the pattern of throughput versus varying number of stations.

While the largest median is found using TcpNewReno, the smallest deviation is found using TcpTahoe. TcpWestwood geenrates both the largest and smallest throughputs.

References

[1] https://en.wikipedia.org/wiki/IEEE_802.11_RTS/CTS

[2] Michael F. *A Performance Analysis Of The Ieee 802.11B Local Area Network In The Presence Of Bluetooth Personal Area Network.* June, 2001.

[3] http://www.radio‘electronics.com/info/wireless/wi‘fi/ieee‘802‘11b.php

A Appendix

```
/* -*-  Mode: C++; c-file-style: "gnu"; indent-tabs-mode:
   nil; -*- */
/*
 * Author: Qiaoqiao Li <joyinbritish@me.com>
 *
 * This is a simple example to test TCP over 802.11b.
 * Use the following command to run the default
    configuration:
 * ./waf --run tcp-80211b.cc
 *
 * In this example, n wifi stations send TCP packets to the
     access point.
 * We report the total throughput received by the access
    point during simulation time.
 *
 * The user can change the following parameters through
    command line:
 * 1. the number of STA nodes (Example: ./waf --run "tcp
    -80211b --nWifi=50"),
 * 2. the payload size (Example: ./waf --run "tcp-80211b --
    payloadSize=2000"),
 * 3. application data rate (Example: ./waf --run "tcp
    -80211b --dataRate="1Mbps""),
 * 4. variant of TCP i.e. congestion control algorithm to
    use
 *     (Example: ./waf --run "tcp-80211b --tcpVariant="
    TcpTahoe""),
 * 5. physical layer transmission rate, i.e. four different
     data rates of 1, 2, 5.5 or 11 Mbps
 *     (Example: ./waf --run "tcp-80211b --"DsssRate5_5Mbps
    ""),
 * 6. simulation time (Example: ./waf --run "tcp-80211b --
    simulationTime=10"),
 * 7. enable/disable pcap tracing (Example: ./waf --run "
    tcp-80211b --pcapTracing=true").
 *
 * Network topology:
 *
 *    STA          AP
 *    *            *
 *    |            |
 *    nWifi        nWifi+1
 *
 */

#include "ns3/core-module.h"
#include "ns3/network-module.h"
```

```
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/internet-module.h"
#include <vector>

NS_LOG_COMPONENT_DEFINE ("tcp-80211b");

using namespace ns3;

int
main(int argc, char *argv[])
{
  double throughput = 0;
  uint32_t totalPacketsThrough = 0;

  uint32_t nWifi = 50;
  uint32_t payloadSize = 1024;                       /*
     Transport layer payload size in bytes. */
  std::string dataRate = "100Mbps";                  /*
     Application layer datarate. */
  std::string tcpVariant = "ns3::TcpNewReno";        /* TCP
      variant type. */
  std::string phyRate = "DsssRate11Mbps";            /*
     Physical layer bitrate. */
  double simulationTime = 1;                         /*
     Simulation time in seconds. */
  bool pcapTracing = false;                          /*
     PCAP Tracing is enabled or not. */

  /* Command line argument parser setup. */
  CommandLine cmd;
  cmd.AddValue ("nWifi", "Number of STA nodes", nWifi);
  cmd.AddValue ("payloadSize", "Payload size in bytes",
     payloadSize);
  cmd.AddValue ("dataRate", "Application data rate",
     dataRate);
  cmd.AddValue ("tcpVariant", "Transport protocol to use:
     TcpTahoe, TcpReno, TcpNewReno, TcpWestwood,
     TcpWestwoodPlus ", tcpVariant);
  cmd.AddValue ("phyRate", "Physical layer bitrate:
     DsssRate11Mbps, DsssRate5_5Mbps, DsssRate2Mbps,
     DsssRate1Mbps ", phyRate);
  cmd.AddValue ("simulationTime", "Simulation time in
     seconds", simulationTime);
  cmd.AddValue ("pcap", "Enable/disable PCAP Tracing",
     pcapTracing);
  cmd.Parse (argc, argv);
```

```
/* No fragmentation */
Config::SetDefault ("ns3::WifiRemoteStationManager::
    FragmentationThreshold", StringValue ("999999"));

/* Disable RTS/CTS */
Config::SetDefault ("ns3::WifiRemoteStationManager::
    RtsCtsThreshold", StringValue ("999999"));
/* Enable RTS/CTS for frames larger than 1000 */
//Config::SetDefault ("ns3::WifiRemoteStationManager::
    RtsCtsThreshold", StringValue ("1000"));


/* Configure TCP Options */
Config::SetDefault ("ns3::TcpSocket::SegmentSize",
    UintegerValue (payloadSize));

WifiMacHelper wifiMac;
WifiHelper wifiHelper;
wifiHelper.SetStandard (WIFI_PHY_STANDARD_80211b);

/* Set up Legacy Channel */
YansWifiChannelHelper wifiChannel ;
wifiChannel.SetPropagationDelay ("ns3::
    ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::
    FriisPropagationLossModel", "Frequency", DoubleValue
    (5e9));

/* Setup Physical Layer */
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ()
    ;
wifiPhy.SetChannel (wifiChannel.Create ());
wifiPhy.Set ("TxPowerStart", DoubleValue (10.0));
wifiPhy.Set ("TxPowerEnd", DoubleValue (10.0));
wifiPhy.Set ("TxPowerLevels", UintegerValue (1));
wifiPhy.Set ("TxGain", DoubleValue (0));
wifiPhy.Set ("RxGain", DoubleValue (0));
wifiPhy.Set ("RxNoiseFigure", DoubleValue (10));
wifiPhy.Set ("CcaMode1Threshold", DoubleValue (-79));
wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (-79
     + 3));
wifiPhy.SetErrorRateModel ("ns3::YansErrorRateModel");
wifiHelper.SetRemoteStationManager ("ns3::
    ConstantRateWifiManager",
                                    "DataMode",
                                        StringValue (
                                        phyRate),
                                    "ControlMode",
                                        StringValue (
                                        phyRate));

  NodeContainer wifiStaNodes;
  wifiStaNodes.Create (nWifi);
  NodeContainer wifiApNode;
  wifiApNode.Create (1);

  /* Configure AP */
  Ssid ssid = Ssid ("network");
  wifiMac.SetType ("ns3::ApWifiMac",
                   "Ssid", SsidValue (ssid));

  NetDeviceContainer apDevice;
  apDevice = wifiHelper.Install (wifiPhy, wifiMac,
     wifiApNode);

  /* Configure STA */
  wifiMac.SetType ("ns3::StaWifiMac",
                   "Ssid", SsidValue (ssid),
                   "ActiveProbing", BooleanValue (false));

  NetDeviceContainer staDevices;
  staDevices = wifiHelper.Install (wifiPhy, wifiMac,
     wifiStaNodes);

  /* Mobility model */
  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::
     RandomDirection2dMobilityModel",
                              "Bounds", RectangleValue(
                                 Rectangle(-500, 500,
                                 -500, 500)),
                              "Speed", StringValue ("ns3::
                                 ConstantRandomVariable[
                                 Constant=2]"),
                              "Pause", StringValue ("ns3::
                                 ConstantRandomVariable[
                                 Constant=0.2]"));

  mobility.Install (wifiStaNodes);

  mobility.SetMobilityModel ("ns3::
     ConstantPositionMobilityModel");
  mobility.Install (wifiApNode);


  /* Internet stack */
  InternetStackHelper stack;
  stack.Install (wifiApNode);
  stack.Install (wifiStaNodes);

  Ipv4AddressHelper address;

  address.SetBase ("192.168.1.0", "255.255.255.0");
  Ipv4InterfaceContainer StaInterface;
  StaInterface = address.Assign (staDevices);
  Ipv4InterfaceContainer ApInterface;
  ApInterface = address.Assign (apDevice);

  /* Populate routing table */
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  /* Install TCP Receiver on the access point */
  uint16_t port = 50000;
  Address apLocalAddress (InetSocketAddress (Ipv4Address::
     GetAny (), port));
  PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory
     ", apLocalAddress);

  ApplicationContainer sinkApp = packetSinkHelper.Install (
     wifiApNode.Get (0));
  sinkApp.Start (Seconds (0.0));
  sinkApp.Stop (Seconds (simulationTime + 1));

  /* Install TCP Transmitter on the stations */
  OnOffHelper onoff ("ns3::TcpSocketFactory",Ipv4Address::
     GetAny ());
  onoff.SetAttribute ("OnTime",  StringValue ("ns3::
     ConstantRandomVariable[Constant=1]"));
  onoff.SetAttribute ("OffTime", StringValue ("ns3::
     ConstantRandomVariable[Constant=0]"));
  onoff.SetAttribute ("PacketSize", UintegerValue (
     payloadSize));
```

```
  onoff.SetAttribute ("DataRate", DataRateValue (DataRate (
     dataRate)));

  AddressValue remoteAddress (InetSocketAddress (
     ApInterface.GetAddress (0), port));
  onoff.SetAttribute ("Remote", remoteAddress);

  ApplicationContainer apps;
  apps.Add (onoff.Install (wifiStaNodes));
  apps.Start (Seconds (1.0));
  apps.Stop (Seconds (simulationTime + 1));

  /* Enable Traces */
  if (pcapTracing)
    {
      wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::
         DLT_IEEE802_11_RADIO);
      wifiPhy.EnablePcap ("AccessPoint", apDevice);
      wifiPhy.EnablePcap ("Station", staDevices);
    }

  /* Start Simulation */
  Simulator::Stop (Seconds (simulationTime + 1));
  Simulator::Run ();
  Simulator::Destroy ();

  totalPacketsThrough = DynamicCast<PacketSink> (sinkApp.
     Get (0))->GetTotalRx ();
  throughput = totalPacketsThrough * 8 / (simulationTime *
     1000000.0);

  std::cout << "\nThroughtput: " << throughput << " Mbit/s"
      << std::endl;
  return 0;
}
```