Northeastern University

Department of Electrical and Computer Engineering

# EECE 4638: **Wireless Design and Simulation**

Lecturer: ***Rahman Doost-Mohammady***

TAs:

**Subhramoy Mohanti**

Lab # 5 : Wifi Hidden Terminal, RTS/CTS

Group:

*Taylor Skilling (001171439)*

*Chris Kenyon (001112428)*

Semester: Summer II 2015

Date: August 17th, 2015

Lab Session: *Monday-Thursday, 3:20 PM*

##### Lab Location: 429 Dana Research Lab, Northeastern University, Boston, MA 02115

##### Content Page

##### Introduction / Objective 3

##### Design Approach 4

##### Results and Analysis 5

##### Appendix A 8

1. **Introduction / Objectives**

The intent of this experiment is to observe what happens when there simultaneous transmissions from different nodes with and without Ready to Send/Clear to Send (RTS/CTS). The simplest form of WiFi, the Ad-hoc model, was used to support nodes without any backed infrastructure support. The “Hidden Terminal Problem,” which occurs when two nodes are trying to communicate with a single access point, but are unaware of each other. This lack of communication means the two nodes cannot detect collisions when transmitting, To combat this, IEEE 802.11 RTS/CTS was developed to help reduce the number of frame collision by sending a smaller RTS/CTS packet which determines whether or not the access point is available to communicate with the client node.

1. **Design Approach**

We will design this simulation using NS3 along with the help of some provided libraries. For setting up the WiFi simulation, the wifiChannel object from *ns3/wifi-module.h*, *ns3/core-module.h, ns3/network-module.h, ns3/internet-module.h, ns3/applications-module.h, ns3/propagation-module.h,* and *ns3/mobility-module.h*.

To set up the simulation, we first determined the threshold value to active or deactive RTS/CTS. Nodes were created and placed in the wireless simulation. A matrix propagation loss model was used to determine if the nodes and access points could communicate, or were hidden or not, based on the specific losses between them. A wifi channel was created and the devices were installed as ad-hoc operating at a constant rate. The Internet and applications were created before UDP clients were created using the UdpEchoClientHelper class. A FlowMonitor was then installed to monitor the throughput. A flag was created to determine whether RTS or CTS was activated.

The flow stats allowed us to look at solely the CBR streams, which lasted for approximately 9 seconds. The flow between seperates revealed the number of transmitted and received packets, bytes and the overall throughput. See Lab\_5.cc in the Appendix for the code with more details.

Running waf to build the program will simulate the network and produce text files with the statistics. Excel was used to parse the data outputs and plot the relevant results.

1. **Results and Analysis**

The effects of RTS/CTS were examined when two transmitters were both hidden and not hidden to each other. When RTS/CTS was disabled and the two nodes were hidden from each other, the throughput was significantly less, as can been seen in Figure 1 below.

Figure 1 – The effects of RTS/CTS in the Hidden Terminal Problem

Without RTS/CTS activated, interference and collisions occur between the two client nodes. This results in the reduced data rate when compared to the other trials. When RTS/CTS is enabled in the hidden terminal problem, the throughput increases compared to the topology without RTS/CTS. However, both trials in which the nodes could see each other resulted in more throughputs. With RTS/CTS disabled without the hidden terminal problem, the throughput was the highest. This is due to the overhead introduced by using RTS/CTS. This protocol is still faster than both transmissions where the hidden terminal problem was present.

In the second measurement, the hidden terminal problem was introduced again. One client application was fixed at 50 dB loss, while the loss of the other client was incremented in 10 dB intervals. The effects of RTS/CTS on the second application can be seen in Figure 2 below.

The results are as expected based on the reasoning in the previous measurements. The transmissions between the access point (2) and the node that is increasing in distance (3) both have far less throughput than the channel between stationary node (1) and the access point. When RTS/CTS is disabled, the throughput between nodes two and three is significantly less than the throughput through the same nodes when RTS/CTS is enabled initially. However, as the loss between nodes two and three increase, RTS/CTS loses its effectiveness as both transmission channels see around 200 Kbps until no data is received around the 120 dB loss point.

As the loss between nodes two and three increased, the hidden terminal problem became less and less prevalent for nodes one and two. That is why we see an initially low throughput between nodes one and two. With RTS/CTS enabled though, the throughput is higher than if RTS/CTS was disabled. As the loss between nodes two and three increases, the throughput between nodes one and two remains relatively constant until there is no data between nodes two and three, in which case the throughput between nodes one and two increases, as there is no longer a hidden terminal problem at all.

Overall, we see that the inclusion of RTS/CTS mitigates the hidden terminal problem and while it introduces overhead, is generally a good practice in reducing the collision of packets in a simple WiFi network.

1. **Appendix A**

**Lab\_5.cc**

/\* -\*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -\*- \*/

/\*

\* Copyright (c) 2010 IITP RAS

\*

\* 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

\*

\* Authors: Pavel Boyko <boyko@iitp.ru>

\*/

/\*

\* Classical hidden terminal problem and its RTS/CTS solution.

\*

\* Topology: [node 0] <-- -50 dB --> [node 1] <-- -50 dB --> [node 2]

\*

\* This example illustrates the use of

\* - Wifi in ad-hoc mode

\* - Matrix propagation loss model

\* - Use of OnOffApplication to generate CBR stream

\* - IP flow monitor

\*/

#include "ns3/core-module.h"

#include "ns3/propagation-module.h"

#include "ns3/network-module.h"

#include "ns3/applications-module.h"

#include "ns3/mobility-module.h"

#include "ns3/internet-module.h"

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

#include "ns3/wifi-module.h"

using namespace ns3;

/// Run single 10 seconds experiment with enabled or disabled RTS/CTS mechanism

void experiment (bool enableCtsRts)

{

// 0. Enable or disable CTS/RTS

UintegerValue ctsThr = (enableCtsRts ? UintegerValue (100) : UintegerValue (2200));

Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", ctsThr);

// 1. Create 3 nodes

NodeContainer nodes;

nodes.Create (3);

// 2. Place nodes somehow, this is required by every wireless simulation

for (size\_t i = 0; i < 3; ++i)

{

nodes.Get (i)->AggregateObject (CreateObject<ConstantPositionMobilityModel> ());

}

// 3. Create propagation loss matrix

Ptr<MatrixPropagationLossModel> lossModel = CreateObject<MatrixPropagationLossModel> ();

lossModel->SetDefaultLoss (200); // set default loss to 200 dB (no link)

lossModel->SetLoss (nodes.Get (0)->GetObject<MobilityModel>(), nodes.Get (1)->GetObject<MobilityModel>(), 50); // set symmetric loss 0 <-> 1 to 50 dB

lossModel->SetLoss (nodes.Get (2)->GetObject<MobilityModel>(), nodes.Get (1)->GetObject<MobilityModel>(), 130); // set symmetric loss 2 <-> 1 to 50 dB

// 4. Create & setup wifi channel

Ptr<YansWifiChannel> wifiChannel = CreateObject <YansWifiChannel> ();

wifiChannel->SetPropagationLossModel (lossModel);

wifiChannel->SetPropagationDelayModel (CreateObject <ConstantSpeedPropagationDelayModel> ());

// 5. Install wireless devices

WifiHelper wifi;

wifi.SetStandard (WIFI\_PHY\_STANDARD\_80211b);

wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",

"DataMode",StringValue ("DsssRate2Mbps"),

"ControlMode",StringValue ("DsssRate2Mbps"));

YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();

wifiPhy.SetChannel (wifiChannel);

NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();

wifiMac.SetType ("ns3::AdhocWifiMac"); // use ad-hoc MAC

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

// uncomment the following to have athstats output

// AthstatsHelper athstats;

// athstats.EnableAthstats(enableCtsRts ? "rtscts-athstats-node" : "basic-athstats-node" , nodes);

// uncomment the following to have pcap output

// wifiPhy.EnablePcap (enableCtsRts ? "rtscts-pcap-node" : "basic-pcap-node" , nodes);

// 6. Install TCP/IP stack & assign IP addresses

InternetStackHelper internet;

internet.Install (nodes);

Ipv4AddressHelper ipv4;

ipv4.SetBase ("10.0.0.0", "255.0.0.0");

ipv4.Assign (devices);

// 7. Install applications: two CBR streams each saturating the channel

ApplicationContainer cbrApps;

uint16\_t cbrPort = 12345;

OnOffHelper onOffHelper ("ns3::UdpSocketFactory", InetSocketAddress (Ipv4Address ("10.0.0.2"), cbrPort));

onOffHelper.SetAttribute ("PacketSize", UintegerValue (1400));

onOffHelper.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1]"));

onOffHelper.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0]"));

// flow 1: node 0 -> node 1

onOffHelper.SetAttribute ("DataRate", StringValue ("3000000bps"));

onOffHelper.SetAttribute ("StartTime", TimeValue (Seconds (1.000000)));

cbrApps.Add (onOffHelper.Install (nodes.Get (0)));

// flow 2: node 2 -> node 1

/\*\* \internal

\* The slightly different start times and data rates are a workaround

\* for \bugid{388} and \bugid{912}

\*/

onOffHelper.SetAttribute ("DataRate", StringValue ("3001100bps"));

onOffHelper.SetAttribute ("StartTime", TimeValue (Seconds (1.001)));

cbrApps.Add (onOffHelper.Install (nodes.Get (2)));

/\*\* \internal

\* We also use separate UDP applications that will send a single

\* packet before the CBR flows start.

\* This is a workaround for the lack of perfect ARP, see \bugid{187}

\*/

uint16\_t echoPort = 9;

UdpEchoClientHelper echoClientHelper (Ipv4Address ("10.0.0.2"), echoPort);

echoClientHelper.SetAttribute ("MaxPackets", UintegerValue (1));

echoClientHelper.SetAttribute ("Interval", TimeValue (Seconds (0.1)));

echoClientHelper.SetAttribute ("PacketSize", UintegerValue (10));

ApplicationContainer pingApps;

// again using different start times to workaround Bug 388 and Bug 912

echoClientHelper.SetAttribute ("StartTime", TimeValue (Seconds (0.001)));

pingApps.Add (echoClientHelper.Install (nodes.Get (0)));

echoClientHelper.SetAttribute ("StartTime", TimeValue (Seconds (0.006)));

pingApps.Add (echoClientHelper.Install (nodes.Get (2)));

// 8. Install FlowMonitor on all nodes

FlowMonitorHelper flowmon;

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

// 9. Run simulation for 10 seconds

Simulator::Stop (Seconds (10));

Simulator::Run ();

// 10. Print per flow statistics

monitor->CheckForLostPackets ();

Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());

FlowMonitor::FlowStatsContainer stats = monitor->GetFlowStats ();

for (std::map<FlowId, FlowMonitor::FlowStats>::const\_iterator i = stats.begin (); i != stats.end (); ++i)

{

// first 2 FlowIds are for ECHO apps, we don't want to display them

//

// Duration for throughput measurement is 9.0 seconds, since

// StartTime of the OnOffApplication is at about "second 1"

// and

// Simulator::Stops at "second 10".

if (i->first > 2)

{

Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);

std::cout << "Flow " << i->first - 2 << " (" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";

std::cout << " Tx Packets: " << i->second.txPackets << "\n";

std::cout << " Tx Bytes: " << i->second.txBytes << "\n";

std::cout << " TxOffered: " << i->second.txBytes \* 8.0 / 9.0 / 1000 / 1000 << " Mbps\n";

std::cout << " Rx Packets: " << i->second.rxPackets << "\n";

std::cout << " Rx Bytes: " << i->second.rxBytes << "\n";

std::cout << " Throughput: " << i->second.rxBytes \* 8.0 / 9.0 / 1000 / 1000 << " Mbps\n";

}

}

// 11. Cleanup

Simulator::Destroy ();

}

int main (int argc, char \*\*argv)

{

std::cout << "Hidden station experiment with RTS/CTS disabled:\n" << std::flush;

experiment (false);

std::cout << "------------------------------------------------\n";

std::cout << "Hidden station experiment with RTS/CTS enabled:\n";

experiment (true);

return 0;

}