NS3 Project Report

Yuzhu Yan 4468023
yuzhuyan@yahoo.com

April 30, 2016

1 Introduction

NS3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. This simulator is written from scratch, using C++ programming language. And it can generate Python bindings and use of the Waf to build the system. The general process of creating a simulation can be divided into several steps[1]: Topology definition, model development, node and link configuration, execution, performance analysis and graphical visualization.

In this project, performance of IEEE802.11 is investigated by constructing an infrastructure mode which consists of one access port and stations with the help of NS3 simulator. The infrastructure is constructed by C++ programming language. For the sake of the reliability of experiment, we also use a shell script to get massive results. In this simulation, payload, throughput, delay, loss and RTS/CTS technology are what we mainly focus on.

This report consists of seven parts. The first section is introduction and followed by the background knowledge. The third section is hypothesis of performance of IEEE 802.11 and the implementation methods are introduced in the following part. All the statistics are showed in section 5 and conclusions are illustrated in section 6. The last section shows the reference materials.

2 Background

2.1 IEEE 802.11

IEEE 802.11 is a set of media access control (MAC) and physical layer (PHY) specifications for implementing wireless local area network (WLAN) computer communication in the 900MHz and 2.4, 3.6, 5, and 60GHz frequency bands. They are created and maintained by the Institute of Electrical and Electronics Engineers (IEEE) LAN/MAN Standards Committee (IEEE 802). The base version of this standard was released in 1997, and has had subsequent amendments. IEEE 802.11b is one of them which was released at September, 1999. In this standard, the frequency is 2.4HZ, brand width is 22 MHZ, data rate should be 1, 2, 5.5 or 11Mbps. Due to the CSMA/CA protocol overhead, in practice the maximum 802.11b throughput that an application can achieve is about

5.9 Mbit/s using TCP and 7.1Mbits using UDP[2]. And DSSS(direct-sequence spread spectrum) modulation is ruled in this protocol. Our investigation is based on IEEE 802.11b in this project.

2.2 RTS/CTS technology

RTS/CTS mechanism is known as a request-to-send/clear-to-send mechanism, which has been proved as an effective way to increase the system performance by reducing the duration of a collision when long messages are transmitted[2]. In paper [2], 802.11 throughput performance was evaluated, compare to Basic Access Mechanism, RTS/CTS mechanism can improve the throughput efficiently, that is because this four ways hand-shaking technique reduce collision during transmission significantly. In RTS/CTS, there is a special value which is considered as a threshold. RTS/CTS only be activated when the payload over this value. That is why we say this mechanism mainly solving long messages collision problems. Before a terminal sending data to another terminal, it always sends out a RTS request. Then the destination terminal sends a CTS message which contains the duration of its communication to all nodes. The communication with other nodes continue until this communication finishes and an idle is sensed by other nodes. CSMA/CA-Hidden terminal problem can be solved by this mechanism(but not completely solved). Furthermore, in IEEE802.11 protocol, retransmission of collided packets is resolved by binary exponential backoff rules.

3 Hypothesis

As we mentioned above, four parameters, including payload, throughput, delay and loss, are investigated in this project. Based on these four parameters, four conditions are explored by means of NS3 simulator. These four conditions are listed below:

1. Throughput vs data rate

In this condition, we observe throughput with four data rates of IEEE 802.11b: 1, 2, 5.5 and 11 Mbps. We also vary payload to see its effects on throughput as well.

2. Payload size vs throughput

In NS3 script, RTS/CTS threshold is set as 900 bytes, which implies this mechanism can only be turned on when payload size is above 900 bytes. In order to investigate the influence of RTS/CTS mechanism, we determine payload size as an variable. By varying the payload size, we compare throughput under two status: RTS/CTS turned off status(payload size less than 900bytes) and RTS/CTS turned on status(payload size larger than 900 bytes). We also set data rate as 11 Mpbs in this condition, in which the variation of throughput is more explicit(This can be illustrated by Figure 2).

3. Payload size vs Delay

Transmission delay is observed with the increasing of payload size. This observation is also based on two conditions: basic access mechanism and RT-S/CTS mechanism.

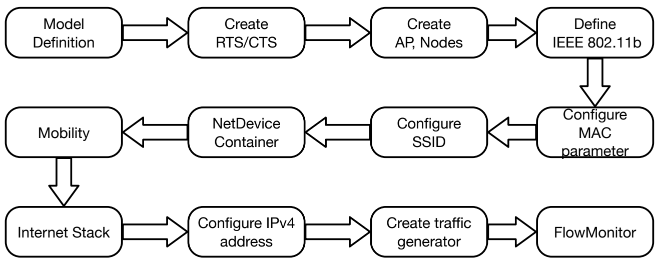
4. Payload size vs Loss

Similarly to aforementioned case, this simulation also bases on two conditions. But packets loss is observed.

4 Implementation

Construction flow for this model by NS 3 is showed in Figure 1.

Figure 1: Infrastructure of Model



In the first step, payload size, data rate and number of nodes are defined. In this project, UPD traffic is used. So the maximum payload size is 1472 bytes. We set payload as 128, 256, 384, 412, 640, 768, 896, 1024,1152,1280 and 1472 bytes respectively.

In above infrastructure, YansWifiPhyHelper will be used to define channel. In this part, ConstantSpeedPropagationDelay Model and FriisPropagationLossModel are used to define propagation delay and propagation loss, respectively. The former model provide a constant propogation speed and the latter is a short distance propagation loss model. And then WifiHelper is used to define IEEE802.11b. RandomDiscPositionAllocator is used to define the mobility of nodes. In this model, nodes are allocated random positions within a disc according to a given distribution for the polar coordinates of each node with respect to the provided center of the disc[5]. The radius can be resized by define parameter Rho. This model aims at minimize unfairness for each node to a very great extent. Furthermore, ConstantPositionMobilityModel is adopted to fix AP at (0,0). In the end, FlowMonitorHelper is used to monitor the performance of this model.

In this experiment, we only take 5 nodes into account which are allocated randomly , this leads to the fluctuation of the result. Thus, we an experiment 30 times in each condition in order to get a more accurate consequence. Then we bases the analysis on mean throughput of these experiments. We use a shell script which can run 30 times automatically and get 30 results for each configured value. Following code fragment shows the details of this shell script, and the 30 results are written in a file named output.txt automatically. In order to make it clear, we use a line of * to split these results. After that, we analyze these data by Microsoft Excel. Hence, 5100 nodes were under tests, and the script(see appendix) run 1020 times. 34 files were created by the shell script. And All relevant files and an example output.txt file can be seen on GitHub.

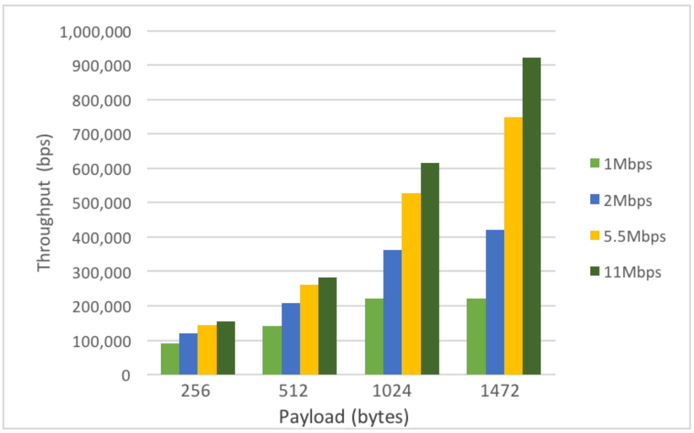
Furthermore, the version of this NS3 software is 3.24.1 and the operating system is OS X Yosemite 10.10.5

```
#!/bin/sh
./waf
for ((i=1;i ¡ 31;i++))
do
now=$(./waf –run scratch/basicwithRTS1)
echo ”$now” ≫ output.txt
echo ”\ n*********************************************”≫ output.txt
done
```

5 Result and Analysis

5.1 Throughput vs Data rate

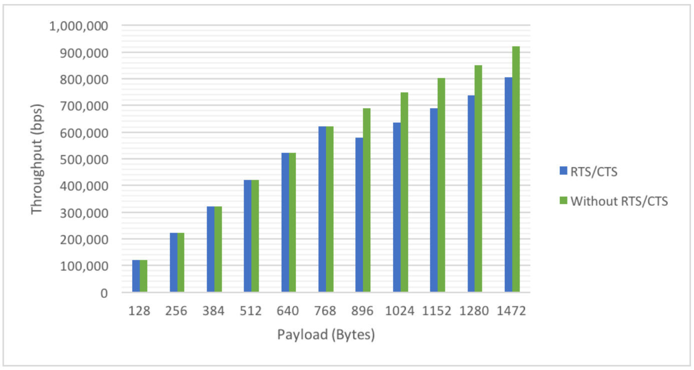
Figure 2



In this condition, we compare throughput of different payload under various data rate, thus, RTS/CTS threshold was set as 2000, so no packet is transmitted under RTS/CTS mechanism. From above figure, it can be seen that throughput increases with the increasing of payload, no matter what the data rate is. Compare to other column, throughput rises more significantly when data rate is 11Mbps. The columns of 1 Mpbs increases slightly, and it reaches to a saturated status when payload arrives at 1024 bytes.

5.2 Payload vs Throughput

Figure 3: Payload vs Throughput



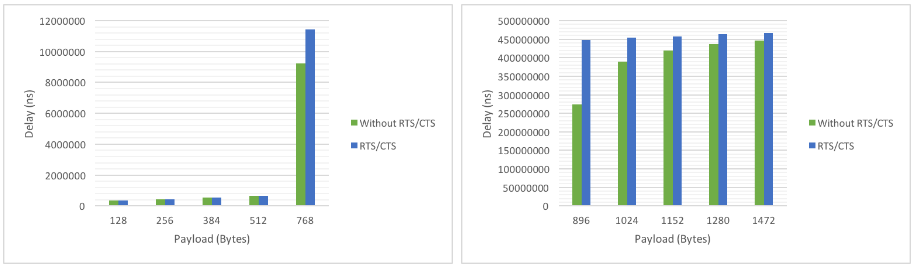
According to paper[1], collision could be avoided by implementing RTS/CTS mechanism. And throughput should be improved in this condition. However, Figure 3 depicts an opposite result. It can be seen that throughput are similar before payload size reaches to 896 bytes. And the throughput goes up with the rising payload. After 896 bytes, throughput under RTS/CTS are much lower than the model without RTS/CTS mechanism. This can be explained as that RTS/CTS mechanism aims at reducing frame collisions introduced by the hidden node problem. However, in this simulation, there is only one access point, so hidden node problem does not happen. Besides, RTS is always sent out after sensing idle for a distributed interframe space (DIFS) which follows the back-off rules, causing long transmission time which lower the throughput. Besides, RTS request itself consumes time which also cause delay. The following equation shows the calculation of delay by throughput.

$$Throughput = \frac{rxBytes * 8.0}{(timeLastRxPacket - timeFirstTxPacket) * 1024 * nNodes}$$

It is also notable that the decrease of RTS/CTS column happens at 896 bytes rather than 1024 bytes, because the threshold is set as 900 bytes. My explanation is payload size is not exactly the size of a packet. For UDP packet, the length of its header is 8 bytes, so when payload size is at 896 bytes, corresponding packet size is already larger than 900 bytes. That explains why RTS/CTS influences start at 896 bytes.

5.3 Payload size vs Delay

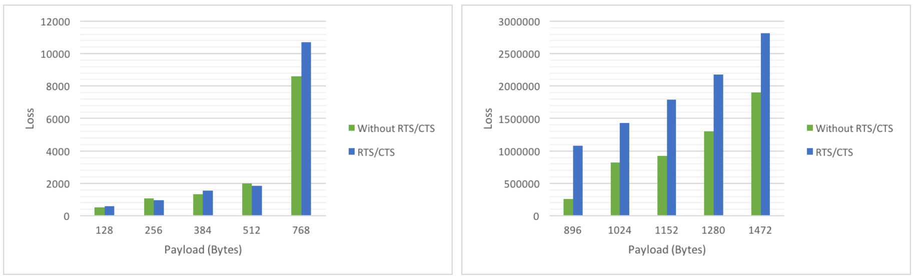
Figure 4: Payload size vs Delay



In 5.2, we assume low throughput caused by longer time delay, in order to verify this, delay of two mechanisms are measured and showed in figure 4. In the first figure, we can see that delay of two mechanisms are very similar, there is slightly different when the payload size is equal to 768 bytes. However, in the second figure where the payload size is in a range from 896 to 1472 bytes, delay of two mechanism become fairly different. But delay of RTS/CTS reaches to a saturated status earlier than basic access mechanism.

5.4 Throughput vs Loss

Figure 5: Payload size vs Loss



Similarly to 5.3, there is also no obvious difference of loss before payload reaches to 896 bytes. And loss of two mechanism both increase dramatically after threshold and reach to the maximum loss when payload is 1472 bytes.

6 Conclusion

Based on above experiments, partial performance of IEEE 802.11b can be concluded as following items:

1. Infrastructure with higher data rate shows higher throughput no matter in basic access mechanism or RTS/CTS mechanism.

2. In the infrastructure with only one access port, RTS/CTS will not show its superiority but result in longer delay due to the hand-shaking process.

3. Under UDP protocol, no matter if the RTS/CTS technology is used, the larger the payload is, the more delay, throughput, loss are.

4. It is tough to define a "Best" packet size. Although the throughput increases significantly with the increasing of packet size, the loss, delay increases as well. Throughput is not the only benchmark of a good performance. So we could not say that larger packet size accompanies with better performance.

In the future investigation, new topology should be constructed containing more than access ports. In this new topology, hidden nodes problem emerges, in which we could verify if hand-shaking technique could show its advantages by comparing throughput with basic access mechanism.

7 Reference

[1] NS3 Wiki pedia. https://en.wikipedia.org/wiki/Ns_(simulator)

[2] Ciuseppe Bianchi, "Performance Analysis of the IEEE 802.11 Distributed Coordination Function," IEEE Journal. Commun, vol. 18, pp. 535–547, March. 2000.

[3] IEEE 802.11b-1999 https://en.wikipedia.org/wiki/IEEE_802.11b-1999

[4] Wi-Fi Module in ns-3 https://www.nsnam.org/tutorials/consortium14/ns-3-training-session-5.pdf

[5] NS3 official tutorial https://www.nsnam.org/doxygen/classns3_1_1_random_disc_position_allocator.html#details

Appendix

```
#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 "ns3/flow-monitor-module.h"
#include "ns3/random-variable-stream.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace ns3;
NS_LOG_COMPONENT_DEFINE ("MyWifi");

int
main (int argc, char *argv[])
{
 double StartTime = 0.0;
 double StopTime = 10.0;
 int nNodes = 5; /*number of node*/
 uint32_t payloadSize = 896 /*payload 1472*/;
 uint32_t maxPacket = 10000 /*10000*/ ;
 StringValue DataRate;
 DataRate = StringValue("DsssRate11Mbps");/*1,2,5_5,11*/
        std::string rtsCts("900");

        // Create randomness based on time
        time_t timex;
        time(&timex);
        RngSeedManager::SetSeed(timex);
        RngSeedManager::SetRun(1);

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

```
    FragmentationThreshold",
         StringValue ("2200"));
Config::SetDefault ("ns3::
    WifiRemoteStationManager::NonUnicastMode",
         StringValue (DataRate));

         // Create access point
NodeContainer wifiApNode;
wifiApNode.Create (1);
std::cout << "Access point created.." << '\n';

         // Create nodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nNodes);
std::cout << "Nodes created.." << '\n';

YansWifiPhyHelper phy = YansWifiPhyHelper::
    Default ();
phy.Set ("RxGain", DoubleValue (0) );

YansWifiChannelHelper channel;
channel.SetPropagationDelay ("ns3::
    ConstantSpeedPropagationDelayModel");
channel.AddPropagationLoss ("ns3::
    FriisPropagationLossModel");
phy.SetChannel (channel.Create ());

WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
std::cout << "Wifi 802.11b Phy & Channel
    configured.." << '\n';

         // configure MAC parameter
wifi.SetRemoteStationManager ("ns3::
    ConstantRateWifiManager","DataMode", DataRate
    , "ControlMode", DataRate);


         // wifi.SetRemoteStationManager ("ns3::
             AarfWifiManager");

NqosWifiMacHelper mac = NqosWifiMacHelper::
    Default ();
std::cout << "Control rate configured.." << '\n';

         // configure SSID
Ssid ssid = Ssid ("myWifi");

mac.SetType ("ns3::StaWifiMac",
         "Ssid", SsidValue (ssid),
        "ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes
    );

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

NetDeviceContainer apDevice;
apDevice = wifi.Install (phy, mac, wifiApNode);
std::cout << "SSID, ApDevice & StaDevice
    configured.." << '\n';
// Configure nodes mobility
MobilityHelper mobility, mobilityAp;
// Random Walk 2D Node Mobility Model
/*mobility.SetMobilityModel ("ns3::
    RandomWalk2dMobilityModel",
        "Bounds", RectangleValue (Rectangle
            (-1000, 1000, -1000, 1000)),
        "Distance", ns3::DoubleValue (300.0));*/
//sta node mobility
        mobility.SetPositionAllocator("ns3::
            RandomDiscPositionAllocator",
                "Rho", StringValue("ns3::
                    UniformRandomVariable[Min
                    =40.0|Max=60.0]")
                );
mobility.Install (wifiStaNodes);
        // Constant Mobility for Access Point,
            fix AP at (0,0)
        // Constant Mobility for Access Point
mobilityAp.SetMobilityModel ("ns3::
    ConstantPositionMobilityModel");
mobilityAp.Install (wifiApNode);
std::cout << "Node mobility configured.." << '\n
    ';

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

// Configure IPv4 address
Ipv4AddressHelper address;
Ipv4Address addr;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer staNodesInterface;
Ipv4InterfaceContainer apNodeInterface;
staNodesInterface = address.Assign (staDevices);
```

```
apNodeInterface = address.Assign (apDevice);

for(int i = 0 ; i < nNodes; i++)
{
        addr = staNodesInterface.GetAddress(i);
        std::cout << " Node " << i+1 << "\t "<< "
           IP Address "<<addr << std::endl;
}
addr = apNodeInterface.GetAddress(0);
std::cout << "Internet Stack & IPv4 address
    configured.." << '\n';

// Create traffic generator (UDP)
ApplicationContainer serverApp;
UdpServerHelper myServer (4001); //port 4001
serverApp = myServer.Install (wifiStaNodes.Get
    (0));
serverApp.Start (Seconds(StartTime));
serverApp.Stop (Seconds(StopTime));
UdpClientHelper myClient (apNodeInterface.
    GetAddress (0), 4001); //port 4001
myClient.SetAttribute ("MaxPackets",
    UintegerValue (maxPacket));
myClient.SetAttribute ("Interval", TimeValue (
    Time ("0.002"))); //packets/s
myClient.SetAttribute ("PacketSize",
    UintegerValue (payloadSize));
ApplicationContainer clientApp = myClient.Install
     (wifiStaNodes.Get (0));
clientApp.Start (Seconds(StartTime));
clientApp.Stop (Seconds(StopTime+5));
std::cout << "UDP traffic generated.." << '\n';

// Calculate Throughput & Delay using Flowmonitor
FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon.InstallAll();

Simulator::Stop (Seconds(StopTime+2));
Simulator::Run ();

monitor->CheckForLostPackets ();
int psent=0;
int preceived=0;
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 << ")\n";
                std::cout << " Tx Bytes: " << i->second.
                    txBytes << "\n";
                std::cout << " Rx Bytes: " << i->second.
                    rxBytes << "\n";
                std::cout << " Average Throughput: " << i
                    ->second.rxBytes * 8.0 / (i->second.
                    timeLastRxPacket.GetSeconds() - i->
                    second.timeFirstTxPacket.GetSeconds()
                    )/1024/nNodes << " kbps\n";
                std::cout << " Delay : " << i->second.
                    delaySum / i->second.rxPackets << "\n
                    ";
                psent=psent+i->second.txBytes;
                preceived=preceived+i->second.rxBytes;
                std::cout <<"loss:"<<psent-preceived<<"\n
                    ";
                std::cout <<"send:"<<psent<<"\n";
                std::cout <<"received"<<preceived<<"\n";

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