PROJECT REPORT

Report of Lab NS-3, course ET4394, Wireless Networking

Yujing Gong, 4471458
yjgong929@gmail.com

Contents

1. Introduction

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. 802.11b products appeared on the market in early 2000. The dramatic increase in throughput of 802.11b along with simultaneous substantial price reductions led to the rapid acceptance of 802.11b as the definitive wireless LAN technology.

Devices using 802.11b experience interference from other products operating in the 2.4 GHz band. Devices operating in the 2.4 GHz range include microwave ovens, Bluetooth devices, baby monitors, cordless telephones, and some amateur radio equipment.

In this simulation, 3 network scenarios are simulated and analyzed with various numbers of stations. Average throughput and average packet loss ratio per node are used as indicators of the network performance. NS-3 is used as the tool for the simulation. NS-3 is a free open source network simulation software that allows user to easily set different parameters of network for simulation.

2. Project description and hypothesis

The goal of this project is to generate different network scenarios using NS-3 WifiHelper and measure the performance of IEEE 802.11b network with various number of stations. In this simulation, number of nodes, data rate, rate control algorithm and distance between stations and AP are changed to observe different performances of the network in terms of average throughput and average packet loss ratio.

The number of nodes will always be a changing parameter in the following 3 scenarios. It affects average throughput and average packet loss ratio. The network should get a higher average throughput and lower average packet loss ratio as the number of nodes increases.

2.1. Performance vs. Data rate

According to IEEE 802.11b standard, there are 4 possible data rate values for IEEE 802.11b: 11Mbps, 5.5Mbps, 2Mbps and 1Mbps. Each and every of these data rate values will be set with various node numbers, i.e., from 1 node to 15 nodes, to get the average throughput and average packet loss ratio.

It is assumed that the network gets a better performance with a higher data rate.

2.2. Performance vs. Control Algorithm

Four control algorithms are simulated and compared. They are Ideal Rate control algorithm, AARF Rate control algorithm, AARF-CD Rate control algorithm and AMRR Rate control algorithm. Besides, they are compared with the Constant Rate control algorithm with a data rate of 11Mbps.

Ideal Rate control algorithm is assumed to give the best performance not just because it is an “ideal” rate control algorithm according to its name, but also because of the essence of its algorithm. It is similar to RBAR in spirit. RBAR (Receiver Based Auto Rate) [2] is a rate adaptation algorithm whose goal is to optimize the application throughput. This algorithm requires incompatible changes to the IEEE 802.11 standard. It is an algorithm of little practical interest but can be used as a performance reference [1]. As an algorithm with a similar spirit to RBAR, Ideal Rate control algorithm used in this simulation is also impractical.

The AARF (Adaptive Auto Rate Fallback) Rate control algorithm was initially described in [1]. It is an improvement of ARF (Auto Rate Fallback). Using this algorithm, the period between successive failed attempts to use a higher rate will increase. Thus, there will be fewer failed transmissions and retransmissions, which result in a better performance.

The AARF-CD algorithm was first described in [3]. It is a modification of the AARF scheme. However, in this simulation UDP is used instead of RTS/CTS. Thus, this control algorithm may not outperform the AARF Rate control algorithm.

AMRR Rate control algorithm was initially described in [1]. AARF and AMRR respectively designed for low latency and high latency systems.

2.3. Performance vs. Distance

Distance is surely a factor of the network performance. The network performance in relationship to the distance is simulated and analyzed. The distance is set within the range of 0 to 1000.

It is assumed that the wireless network should get a better performance with a smaller distance while get a worse performance with a larger distance.

3. Implementation

The basic infrastructure of IEEE 802.11b network is established using the modules in NS-3. Set-up of the Wi-Fi simulation contains the following parts:

- Nodes
- Wi-Fi
- Mobility
- Internet
- Application

To build the complete IEEE 802.11b network, AP and stations are firstly created using the NodeContainer class. In the simulation, one AP and N (N = 1, 2, …, 15) nodes are created. IEEE 802.11b standard is set using WifiHelper.

Next, YansWifiPhyHelper and YansWifiChannelHelper are used to create and set the channel. They are the default and most widely used model for this part. YansWifiPhyHelper is set to the default working state and the propagation delay and propagation loss of the channel between stations and AP are set using YansWifiChannelHelper.

After the Phy and channel parameters are set, NqosWifiMacHelper is used to set the MAC parameters. The MAC parameters are set to default values; SSID and NetDeviceContainer are configured for both stations and AP. WifiHelper is used to set the rate control algorithm. Here the default control algorithm is Constant Rate control algorithm. In this simulation, control algorithm is changed to Constant Rate control algorithm, AARF Rate control algorithm, AARF-CD Rate control algorithm and AMRR Rate control algorithm to measure different network performances.

Then, mobility of stations and AP will be configured. The Constant position mobility model is chosen in this simulation. Change of distance between stations and AP is made using this model to analyze the relationship between network performance and node-to-AP distance.

Followed, the Internet stack can now be installed for both stations and AP using InternetStackHelper. Then, IP address is allocated to each stations and AP using Ipv4AddressHelper. In this simulation, the addresses of N stations are 10.1.1.1 to 10.1.1.N and the address of AP is 10.1.1.0.

Lastly, network traffic is generated using UdpServerHelper and UdpClientHelper. In this simulation UDP traffic is used as network traffic between stations and AP. The maximum packet size for client is set to 10000 and the payload/packet size is set to 1472, which is the Maximum Transmission Unit (MTU) for UDP. Simulation time is set to 10s.

So far configuration of the wireless network to be observed is done. Now simulation can start and FlowMonitorHelper is used to get the two performance parameters, average throughput and average packet loss ratio. FlowMonitorHelper can measure the number

of received bytes, the time the first packet is transmitted, and the time the last packet is received, to calculate the average throughput. Also, it can measure the total number of transmitted packet and received packet, respectively, to calculate the average packet loss. The following formulas are used for these calculations.

$$Average\ throughput = \frac{rxBytes \times 8.0}{(timeLastRxPacket - timeFirstTxPacket) \times nNodes \times 1024 \times 1024}\ (Mbps)$$

$$Average\ packet\ loss\ ratio = \frac{txPackets - rxPackets}{txPackets \times nNodes}$$

To observer the network performance of the 3 chosen scenarios, the following changes are done for the simulation. Note that the number of nodes is always a changing parameter in the 3 scenarios.

Firstly, data rates are modified to simulate the network. In this scenario, the control algorithm is set to Constant Rate control algorithm and the position of each station is set to be a random position in a 60x60 rectangle and the position of AP is set to be at a vertices of the rectangle, as shown in Figure 1. Data rate is set to 11Mbps, 5.5Mbps, 2Mbps and 1Mbps to measure the network performance.

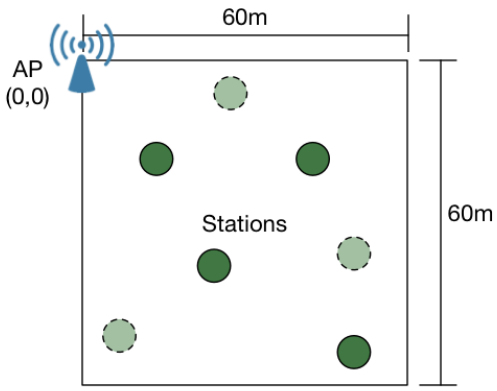

Figure 1. Positions of AP and stations

Secondly, rate control algorithm is modified to simulate the network. In this scenario, it is unnecessary to set the data rate. The positions of stations and AP are set as shown in Figure 1, too.

Thirdly, distance between stations and AP is modified to simulate the network. The rate control algorithm used in this scenario is AARF Rate control algorithm. The positions of stations are also configured using RandomRectanglePositionAllocator, which allocates positions to stations within a determined rectangle following the uniform random distribution. Figure X shows how the distance between stations and AP is changed. The area of the rectangle is remained constant of 100 by 100 and the D (Distance) parameter as shown in Figure 2 is changed to different values in the range of 50 to 1000. In this

scenario the number of nodes is set to 1, 5, and 10 instead of from 1 to 15 for convenience.

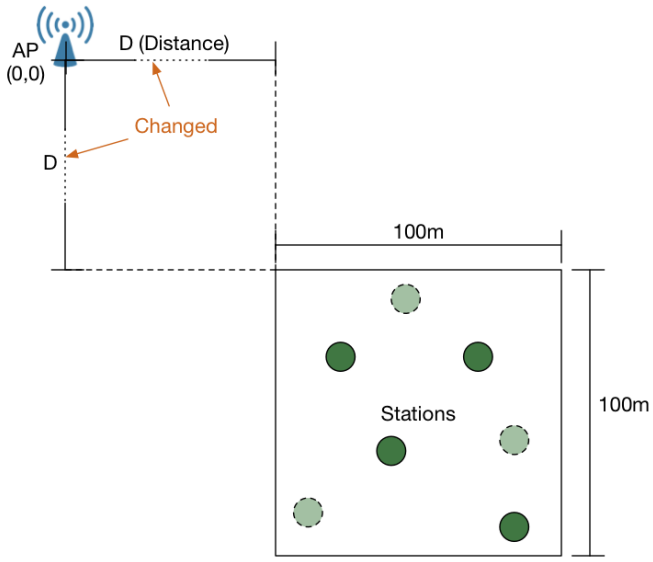

Figure 2. Positions of AP and stations

4. Results & Analysis

Examples of result run using Waf is shown in Figure 3 and 4.

```
Default Run (true:1/false:0)? 1
-------------------------
Number of nodes: 10
Total throughput: 4.50973Mbps   Average throughput: 0.450973
Tx Packets: 5000
Rx Packets: 4137
Lost Packets(flowmonitor): 21
Lost Packets(subtraction): 862
Packet loss ratio: 0.172533
```

Figure 3. Example result of running the default scenario

```
Default Run (true:1/false:0)? 0
Number of stations: 5
Number of simulation times: 3
DataRade (enter 0 for 1Mbps, enter 1 for 2Mbps, enter 2 for 5.5Mbps, enter 3 for 11Mbps):3
Control Algorithm (enter 0 for ConstantRateWifiManager, enter 1 for IdealWifiManager, ente
r 2 for AarfWifiManager, enter 3 for AarfcdWifiManager, enter 4 for AmrrWifiManager:0
Distance: 100
-------------------------
Number of nodes: 5
Total throughput: 4.47711Mbps   Average throughput: 0.895423
Tx Packets: 5000
Rx Packets: 4106
Lost Packets(flowmonitor): 20
Lost Packets(subtraction): 893
Packet loss ratio: 0.178667
```

Figure 4. Example result of running a specified scenario

4.1. Performance vs. Data rate

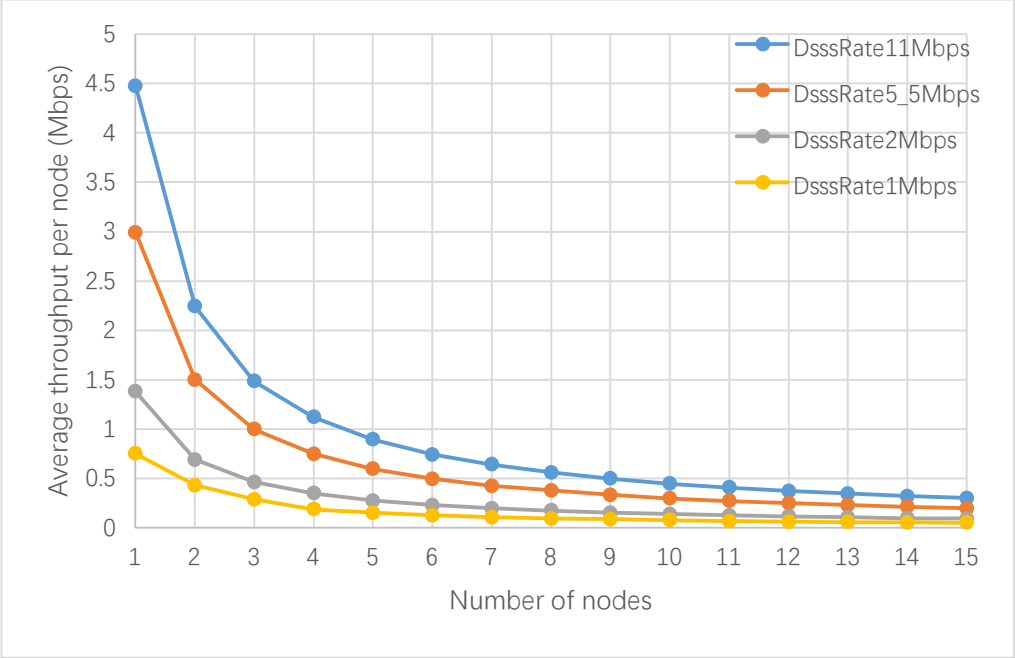

Figure 5. Average throughput vs. Number of nodes plot

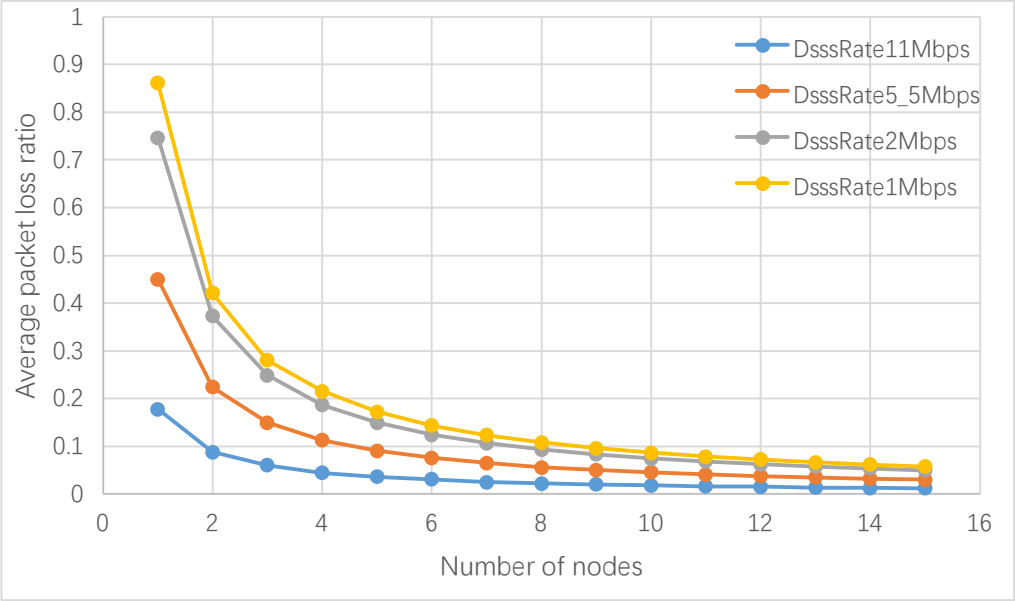

Figure 6. Average packet loss ratio vs. Number of nodes plot

It can be observed from the simulation result that the higher data rate is, the higher throughput and lower packet loss ratio we can get. For each data rate, the average throughput and packet loss ratio are related to the number of nodes. A larger number of nodes gives a relatively lower average throughput and packet loss ratio.

When data rate is set to 11Mbps, constant data rate control algorithm has a similar performance to Ideal Rate control algorithm. As shown before in section 4.1.

4.2. Performance vs. Control Algorithm

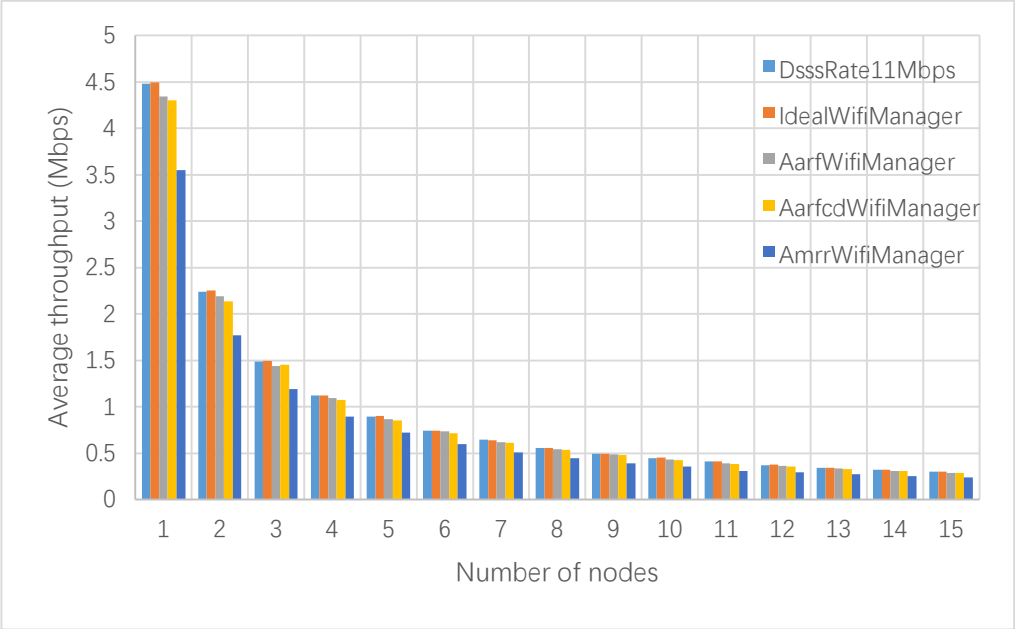

Figure 7. Average throughput vs. Number of nodes plot

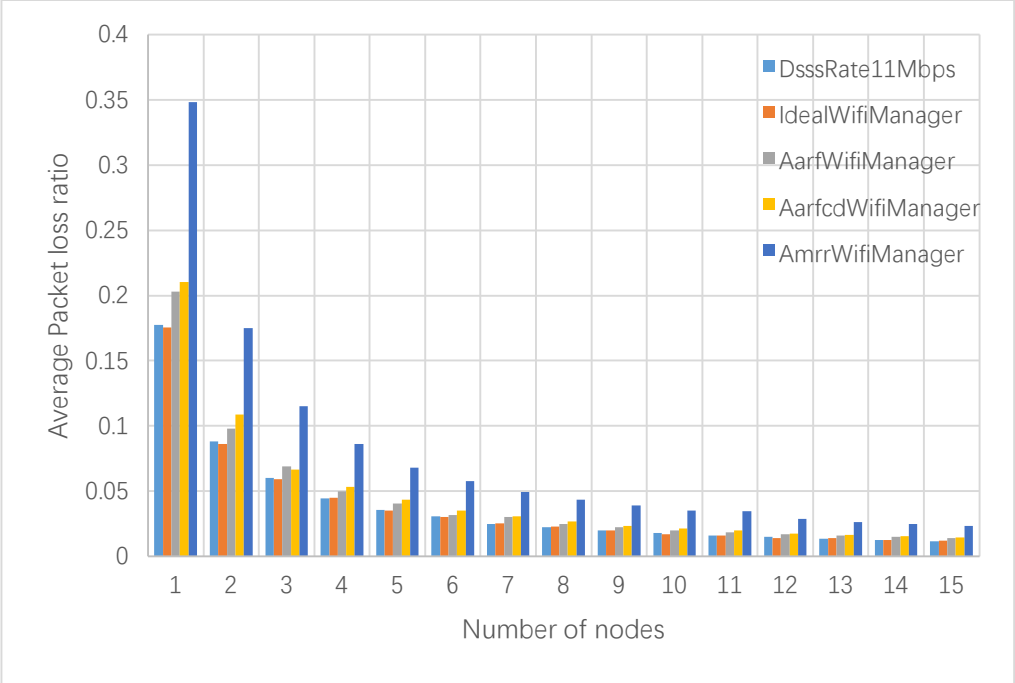

Figure 8. Average packet loss ratio vs. Number of nodes plot

It can be observed from the simulation result that IdealWifiManager gives the best network performance, AarfWifiManager and AarfcdWifiManager give a less good network performance, and AmrrWifiManger gives the worst. When using Constant Rate control algorithm and data rate is set to 11Mbps, the network performance is very close to the ideal one.

4.3. Performance vs. Distance

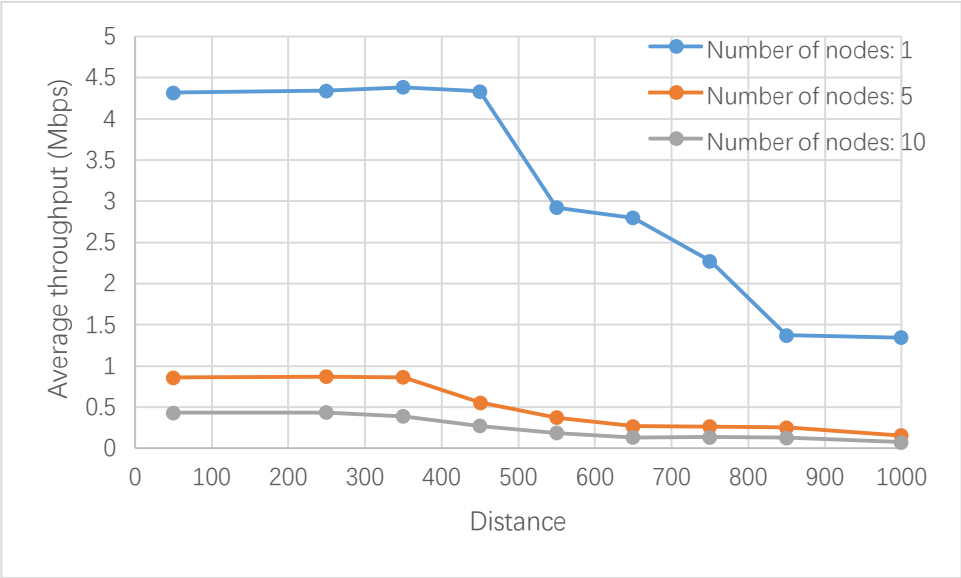

Figure 9. Average throughput vs. Number of nodes plot

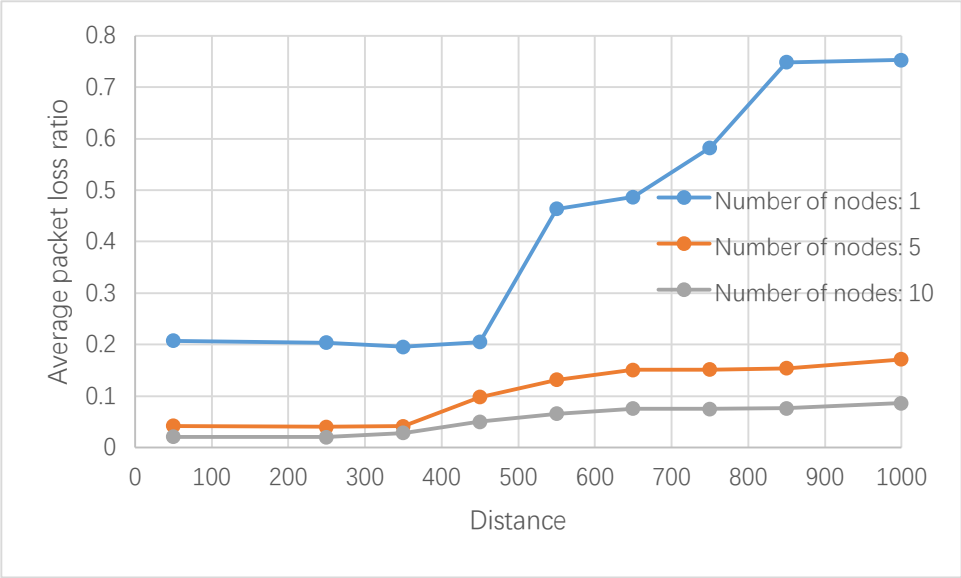

Figure 10. Average packet loss ratio vs. Number of nodes plot

In this scenario, ConstantPositionMobilityModel is used to allocate the position of the station when the number of node is set to 1 and RandomRectanglePositionAllocator is used to allocate the position of then stations when the number of nodes is more than one.

It can be observed from the simulation result that, for the scenario with only one station, the network performance has little change when the distance is smaller than 450. When the distance is set to 450, the network performance degrades substantially. When the distance is set to 850 or larger, the network has a very litter throughput and large packet loss ratio. A similar phenomenon can be observed for the scenario with more than one

stations. When the distance is set to within 350, the network performance changes little. When the distance is larger than 350, the performance starts decreasing greatly.

5. Conclusion

Base on the result and analysis of the three chosen scenarios, conclusions can be made as follows,

The network performance with indicators average throughput and average packet loss ratio is related to the number of nodes. A larger number of stations gives less average throughput and average packet loss ratio

The network performance is close related to the data rate. The highest data rate - 11Mbps gives the best performance.

Rate control algorithms have great influence on network performance. Among the four chosen control algorithms, Ideal rate control algorithm is used as the reference to observe the performance of the others. The Constant rate control algorithm gives the best performance while Aarm Rate control algorithm gives the worst. Aarf and Aarf-CD Rate control algorithms give the medium performances.

The distance between AP and stations have influence on network performance. Within a certain distance rage, the network performance is not affected by the distance. When the distance becomes larger, the network performance degrades greatly and will never become better.

Reference

[1] Lacage M, Manshaei M H, Turletti T. IEEE 802.11 rate adaptation: a practical approach[C]//Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems. ACM, 2004: 126-134.

[2] Holland G, Vaidya N, Bahl P. A rate-adaptive MAC protocol for multi-hop wireless networks[C]//Proceedings of the 7th annual international conference on Mobile computing and networking. ACM, 2001: 236-251.

[3] F. Maguolo, M. Lacage and T. Turletti, "Efficient collision detection for auto rate fallback algorithm," Computers and Communications, 2008. ISCC 2008. IEEE Symposium on, Marrakech, 2008, pp. 25-30.

Appendix

Source code of the simulation.
The source code can also be found on GitHub (https://github.com/yjgong/WirelessNetworking).

```
#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[])
{
    //int count=1;
    //int mob=0;
    int run=1;
    int nNodes=10;
    int simTimes=3;
    int datarate=3;
    int Distance=100;
    int contr=0;

    std::cout << "Default Run (true:1/false:0)? ";
    std::cin >> run;
    if (!run)
    {
        std::cout << "Number of stations: ";
        std::cin >> nNodes;
        std::cout << "Number of simulation times: ";
        std::cin >> simTimes;
```

```
      std::cout << "DataRade (enter 0 for 1Mbps, enter 1 for 2Mbps,
enter 2 for 5.5Mbps, enter 3 for 11Mbps):";
      std::cin >> datarate;
      std::cout << "Control Algorithm (enter 0 for
ConstantRateWifiManager, enter 1 for IdealWifiManager, enter 2 for
AarfWifiManager, enter 3 for AarfcdWifiManager, enter 4 for
AmrrWifiManager:";
      std::cin >> contr;
      std::cout << "Distance: ";
      std::cin >> Distance;
   }

   // Read arguments from command line
   /*CommandLine cmd;
    cmd.AddValue ("n", "Number of stations", n);
    cmd.AddValue ("d", "Distance", r);
    */

   //int nNodes = 1;
   //while(nNodes <= stations)
   //{
   std::cout << "---------------------------" << "\n";
   std::cout << "Number of nodes: " << nNodes << "\n";

   double total = 0;
   float ratio = 0;
   int pkloss = 0;
   int pksent = 0;
   int pkreceived = 0;
   Time delay = Seconds(0);

   int count = 0;
   while(count<simTimes)   //simulate simtTimes times to get the average
values
   {
      double simulationTime = 10.0;
      uint32_t packetSize = 1472;     //max:
      uint32_t nPacket = 10000;
      bool verbose = false;

      StringValue phyMode;
      if (datarate==0) phyMode = StringValue("DsssRate1Mbps");
```

```
        if (datarate==1) phyMode = StringValue("DsssRate2Mbps");
        if (datarate==2) phyMode = StringValue("DsssRate5_5Mbps");
        if (datarate==3) phyMode = StringValue("DsssRate11Mbps");

        // Create randomness based on time
        time_t timex;
        time(&timex);
        RngSeedManager::SetSeed(timex);
        RngSeedManager::SetRun(10);

        CommandLine cmd;
        cmd.Parse (argc,argv);

        //----------------------Create AP & Nodes---------------
        NodeContainer ApNode;
        ApNode.Create (1);

        NodeContainer StaNodes;
        StaNodes.Create (nNodes);


        //-------------------------WifiHelper----------------------
        WifiHelper wifi = WifiHelper::Default ();
        if (verbose)
        {
           wifi.EnableLogComponents (); // Turn on all Wifi logging
        }
        wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

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

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

```
      //--------------------NqosWifiMacHelper (Control algorithm)------
----------------
      NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();     // Set
to a non-QoS upper mac
      if (contr==0)wifi.SetRemoteStationManager
("ns3::ConstantRateWifiManager","DataMode", phyMode, "ControlMode",
phyMode);
      if (contr==1)wifi.SetRemoteStationManager
("ns3::IdealWifiManager");
      if (contr==2)wifi.SetRemoteStationManager
("ns3::AarfWifiManager");
      if (contr==3)wifi.SetRemoteStationManager
("ns3::AarfcdWifiManager");
      if (contr==4)wifi.SetRemoteStationManager
("ns3::AmrrWifiManager");



      //------------------------------Device----------------------
      Ssid ssid = Ssid ("myWifi");
      mac.SetType ("ns3::ApWifiMac", "Ssid", SsidValue (ssid));
      NetDeviceContainer apDevice = wifi.Install (phy, mac, ApNode);
//apcontainer get(0)
      mac.SetType ("ns3::StaWifiMac","Ssid", SsidValue (ssid),
"ActiveProbing", BooleanValue (false));
      NetDeviceContainer staDevices = wifi.Install (phy, mac, StaNodes);

      //------------------------------Mobility---------------------
      MobilityHelper mobilityAp, mobility;

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

      if (nNodes==2)
      {
         Ptr<ListPositionAllocator> positionAlloc =
CreateObject<ListPositionAllocator> ();
         positionAlloc-> Add (Vector (Distance, 0.0, 0.0));
```

```
            mobility.SetPositionAllocator (positionAlloc);
            mobility.SetMobilityModel
("ns3::ConstantPositionMobilityModel");
            mobility.Install (StaNodes);
        }
        else
        {
            //int mob = 0;
            //------------------------------Mobility (Constant)---------
            //if (mob==1)
            //{
            //std::cout << "ConstantPositionMobilityModel" << "\n";
            ObjectFactory pos;
            pos.SetTypeId ("ns3::RandomRectanglePositionAllocator");
            if(Distance>0 && Distance<=100)
            {
                pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=0|Max=100]"));
                pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=0|Max=100]"));
            }
            else if(Distance>100 && Distance<=200)
            {
                pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=100|Max=200]"));
                pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=100|Max=200]"));
            }
            else if(Distance>200 && Distance<=300)
            {
                pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=200|Max=300]"));
                pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=200|Max=300]"));
            }
            else if(Distance>300 && Distance<=400)
            {
                pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=300|Max=400]"));
                pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=300|Max=400]"));
            }
```

```
          else if(Distance>400 && Distance<=500)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=400|Max=500]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=400|Max=500]"));
          }
          else if(Distance>500 && Distance<=600)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=500|Max=600]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=500|Max=600]"));
          }
          else if(Distance>600 && Distance<=700)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=600|Max=700]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=600|Max=700]"));
          }
          else if(Distance>700 && Distance<=800)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=700|Max=800]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=700|Max=800]"));
          }
          else if(Distance>800 && Distance<=900)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=800|Max=900]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=800|Max=900]"));
          }
          else if(Distance>900 && Distance<=1000)
          {
             pos.Set ("X", StringValue
("ns3::UniformRandomVariable[Min=900|Max=1000]"));
             pos.Set ("Y", StringValue
("ns3::UniformRandomVariable[Min=900|Max=1000]"));
          }
```

```
          Ptr<PositionAllocator> positionAlloc = pos.Create
          ()->GetObject<PositionAllocator> ();

          mobility.SetPositionAllocator (positionAlloc);
          mobility.SetMobilityModel
("ns3::ConstantPositionMobilityModel");
          mobility.Install (StaNodes);
          //}
       }

       //-------------------------Internet stack-----------------
       InternetStackHelper internet;
       internet.Install (ApNode);
       internet.Install (StaNodes);

       //---------------------Ipv4AddressHelper----------------------
       Ipv4AddressHelper address;
       NS_LOG_INFO ("Assign IP Addresses.");
       Ipv4Address addr;
       address.SetBase ("10.1.1.0", "255.255.255.0");
       Ipv4InterfaceContainer apNodeInterface = address.Assign
(apDevice);
       Ipv4InterfaceContainer staNodesInterface = address.Assign
(staDevices);

       addr = apNodeInterface.GetAddress(0);

       //---------------------Traffic----------------------
       UdpServerHelper myServer (9);

       ApplicationContainer serverApp;
       serverApp = myServer.Install (StaNodes.Get (0));
       serverApp.Start (Seconds(0.0));
       serverApp.Stop (Seconds(simulationTime));

       UdpClientHelper myClient (addr, 9);
       myClient.SetAttribute ("MaxPackets", UintegerValue (nPacket));
       myClient.SetAttribute ("Interval", TimeValue (Time ("0.002")));
//packets/s
       myClient.SetAttribute ("PacketSize", UintegerValue (packetSize));
```

```
        ApplicationContainer clientApp = myClient.Install (StaNodes.Get
(0));
        clientApp.Start (Seconds(0.0));
        clientApp.Stop (Seconds(simulationTime));

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

        NS_LOG_INFO ("Run Simulation");
        Simulator::Stop (Seconds(simulationTime+2));
        Simulator::Run ();

        monitor->CheckForLostPackets ();
        Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());
        std::map<FlowId, FlowMonitor::FlowStats> stats =
monitor->GetFlowStats ();

        double temp_total = 0;
        float temp_ratio = 0;
        int temp_ploss = 0;
        int temp_psent = 0;
        int temp_preceived = 0;
        Time temp_delay = Seconds(0);

        for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator iter
= stats.begin(); iter != stats.end(); ++iter)
        {
            //Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow
(iter->first);
            temp_total = iter->second.rxBytes * 8.0 /
(iter->second.timeLastRxPacket.GetSeconds() -
iter->second.timeFirstTxPacket.GetSeconds())/1024/1024;
            //std::cout << "Flow " << iter->first << " (" <<
t.sourceAddress << " -> " << t.destinationAddress << ")\n";
            temp_psent = iter->second.txPackets;
            temp_preceived = iter->second.rxPackets;
            temp_ploss = iter->second.lostPackets;
            temp_ratio = (double)(temp_psent-
temp_preceived)/(double)temp_psent;
```

```
          //temp_delay = iter->second.delaySum.GetSeconds() /
temp_pkreceived;
          //temp_delay = iter->second.delaySum / iter->second.rxPackets;
          //std::cout << "Delay: " << iter->second.delaySum /
iter->second.rxPackets << "\n";

          /*if (t.destinationAddress=="10.1.1.1")
           {
           std::cout << "TEST: " << iter->second.txPackets << "\t";
           std::cout << "TEST: " << iter->second.rxPackets << "\n";
           } */
       }
       total = total + temp_total;
       pksent = pksent + temp_pksent;
       pkreceived = pkreceived + temp_pkreceived;
       pkloss = pkloss + temp_pkloss;
       ratio = ratio + temp_ratio;
       //delay = delay + temp_delay;
       Simulator::Destroy ();

       count++;
   }

   std::cout << "Total throughput: " << total/count << "Mbps" <<"\t";
       std::cout << "Average throughput: " << total/nNodes/count <<"\n";
//<< "Mbps" <<"\n"
       std::cout << "Tx Packets: " << pksent/count << "\n";
       std::cout << "Rx Packets: " << pkreceived/count << "\n";
   std::cout << "Lost Packets(flowmonitor): " << pkloss/count << "\n";
   std::cout << "Lost Packets(subtraction): " << (pksent-
pkreceived)/count << "\n";
       std::cout << "Packet loss ratio: " << ratio/count <<"\n";
       //std::cout << "Delay: " << delay/count << "\n";

   //nNodes = nNodes + 1;
   //}
   return 0;
}
```