Delft University of Technology

ET4394 Wireless Networking

A Project On

Performance Analysis of 802.11 Rate Adaptation Algorithms

Author:
Varun Nair

Student Number:
4504550

May 2, 2016

Contents

Abstract

Rate adaptation is a critical functionality defined in the IEEE 802.11 standard and implemented at the MAC layer. As a result, several PHY layer data rates have been defined targeting different channel qualities. The role of a rate adaptation algorithm is in determining how and when to switch between a higher and a lower rate, with(Closed Loop) or without(Open Loop) channel feedback. In this project, we study three open-loop algorithms- Auto Rate Fallback (ARF), Adaptive Auto Rate Fallback (AARF) and Adaptive Auto Rate Fallback with Collision Detection (AARF-CD). Their performance is evaluated under three loss scenarios- due to channel errors only, due to collisions only and losses due to both collisions and channel errors. Results show that ARF is not an effective algorithm under slow-changing channels and only performs at the same level as ARF at best. AARF-CD is most effective under first(pure channel losses) and third scenarios(both channel and collision). In the second scenario, if RTS-CTS is enabled, then AARF-CD performs better only under low contention.It is recommended that nodes use a combination of either ARF/AARF or AARF-CD depending on the type of scenario they are in.

1. Introduction

1.1 Need for Rate Adaptation

It is a known and established fact that wireless channels are prone to variations. Variations could refer to changes in signal to noise ratio at the receiver (due to interference or mobility), number of contending stations for channel access, multipath effects etc. Under such circumstances, a constant PHY layer data rate is ineffective as it may be too high or too low for a given channel condition.For instance, if the SNR is very low, then using a high bit rate will lead to increased bit error rate and retransmissions, thereby decreasing the overall throughput. On the other hand, using a low bit rate in a high SNR channel leads to under-utilization of the channel (with respect to its Shannon Capacity value). Hence, there is a need to 'adapt' the data rate as per the prevailing channel conditions so as to maximize throughput or minimise packet losses.

All the existing IEEE 802.11 standards (a/b/g/n) support rate adaptation algorithms i.e a node can transmit in multiple possible rates. For example, 802.11b supports 4 data rates namely 1 Mbps, 2 Mbps, 5.5 Mbps and 11 Mbps.Similarly 802.11g supports twelve data rates between 1-54 Mbps.

1.2 Effectiveness of a Rate Adaptation Algorithm

There are **two major factors** which determine the **effectiveness** of rate adaptation algorithms. **Firstly**, in the context of WiFi, where there is statistical multiplexing, it is important to determine the cause of packet losses before adapting the rate. If the losses are only due to collisions and not due to poor SNR, then adjusting the rate is not effective. **Secondly**, the rate of variation of channel must also be considered. If the channel varies rapidly, then the adaptation must also be fast and keep up with the nature of the channel. If the variation is slow, then the adaptation must be steady.

1.3 Classes of Rate Adaptation Algorithms

There are two broad classes of rate adaptation algorithms: -

1. **Open Loop Algorithms**
   In an Open Loop approach, decision to increase or decrease the rate is taken on the basis of success or failure in frame transmission respectively. There is no channel related feedback taken from the receiver. Hence, the name 'open loop'. Examples are Auto Rate Fallback (ARF) algorithms, Adaptive Auto Rate Fallback (AARF),Adaptive Auto Rate Fallback with Collision Detection (AARF-CD) Adaptive Multi Rate Retry (AMRR) etc. The main drawback of the 'open loop' approach is that it is not able to distinguish collision errors from channel errors. The advantage is that it is easier to implement and no changes are required in the frame format or handshake mechanisms. Hence, such algorithms have been widely adopted.

2. **Closed Loop Algorithms**
   In the closed loop approach, the transmitter changes its rate based on a feedback information from the receiver. It is proposed that channel information such as SNR be conveyed via the CTS message itself, which will require changes to the header format.This has limited the adoption of closed loop algorithms.Examples are Receiver Based Auto Rate(RBAR)and Robust Rate Adaption Algorithm (RRAA).

In this project, three open-loop rate adaptation algorithms (ARF, AARF and AARF-CD) have been compared under different scenarios (explained in Sec 3). The outline of the document is as follows : Section 2 gives a brief overview of the three algorithms, section 3 provides a description of the simulation scenarios, section 4 discusses the results of the simulations and finally Section 4 concludes the report.

2. Overview of the ARF, AARF and AARF-CD Algorithms

2.1 ARF

ARF or Auto Rate Fallback [1] was the first rate adaptation algorithm to be proposed.In ARF, ,after a fixed number of successful transmissions at a given rate, the current rate is increased to the next higher rate.It is decreased to a lower rate when two consecutive transmissions fail. Once this happens,a timer is started. When either the timer expires or the number of successfully received per-packet acknowledgments reaches 10, the transmission rate is increased to a higher data rate and the timer is reset to a default value. The first transmission immediately following a rate increase must be successful, otherwise the rate is lowered.

Two **issues** were identified by [2], in this algorithm: -

1. ARF is unable to keep up if the channel conditions change quickly. In ad-hoc networks, for instance, the channel quality may vary with each packet thus requiring optimal rate to be determined per packet. But, since ARF waits for 10 successful packet transmissions, the response is very slow and it may never be able to synchronize with the channel conditions.
2. If the channel conditions change slowly or not at all, ARF will still try to increase the rate after every 10 successful packets.If channel is not suitable for a higher rate, then the packet transmission fails.Yet, the algorithm does not adapt and retransmissions repeat after every 10 packets (sent at the lower rate). This leads to drop in application throughput.

Apart from the above, another **drawback** is that ARF does not distinguish between collisions and channel errors. This impacts the throughput when RTS-CTS is enabled and the frame losses are due to channel errors. This will investigated and discussed further as part of the simulation experiment in Section 3.

2.2 AARF

Adaptive Auto Rate Fallback (AARF) was proposed by [2] to address primarily the second issue mentioned above, since it commonly affects the infrastructure mode based WiFi networks. The idea is to adapt the threshold for increasing the transmission rate (previously 10) based on the history of unsuccessful packet transmissions. A Binary Exponential Back-off scheme is used to modify the threshold.

When the transmission of the first packet after switching to a higher rate fails, then it is immediately switched back to the previous rate and the threshold number of successful transmissions is increased by a factor of 2. Thus, after the first unsuccessful transmission , the threshold increases to 20 (and so on upto a maximum of 60). Therefore, unlike in ARF, the node will try to increase the rate fewer number of times in a given time interval.With fewer failed retransmission attempts, the application throughput is increased.

The major **drawbacks** for AARF are: -

1. Issue 1, as mentioned in Sec 2.1 (ARF) is still not addressed by AARF since there is no logic to decrease the threshold so as to adapt to fast changing channel conditions. Therefore, the algorithm is not expected to perform well in dense ad-hoc networks.

2. No mechanism is incorporated to distinguish between frame collisions and channel errors. Therefore, it is expected to suffer from the same drawbacks during RTS-CTS mode as in ARF.

2.3 AARF-CD

The main motivation to propose Adaptive Auto Rate Fallback with Collision Detection(AARF-CD)[3] was to improve the performance of earlier ARF and AARF algorithms under RTS-CTS mode. RTS-CTS is not beneficial when losses are due to channel errors only and therefore must be ideally disabled during that time and re-enabled only when collisions occur. AARF-CD attempts to achieve this in the following way:-
Two additional counters have been introduced- an rtsCounter and a nFailed counter. The former is used to check whether to disable or enable RTS-CTS and the latter checks whether the current rate must be decreased or not. The rtsCounter starts from a maximum value rtsWnd and decrements by one with each successful RTS-CTS handshake.RTS-CTS is disabled when the counter reaches 0. If the number of successful data transmissions reach a threshold nSuccess packets, the current rate is increased with RTS-CTS turned on. If the number of unsuccessful transmissions after a rate increase reach a threshold nFailed, then the rate is immediately decreased and the threshold nSuccess is doubled (just like in AARF).If the failed transmissions occur above threshold in normal circumstances when RTS-CTS is used (and not immediately after rate increase), then the rate is decreased but nSuccess is not doubled (it is set to minimum value). This is because there is a possibility that channel conditions may have changed only for a short term. So, doubling the nSucccess threshold may result in prolonged period of unnecessary transmission in a lower rate.
Every time the rate is decreased or when the rtsCounter reaches 0, the RTS-CTS mode is switched off. This helps in avoiding the use of RTS-CTS when losses are due to channel errors.But if a data transmission failure occurs(due to channel error or collision) without using RTS-CTS, then RTS-CTS is immediately enabled although the rate is not decreased.This accounts for the fact that collision may have occurred and there is no need to decrease the rate.

So we see that RTS-CTS is **switched off** every time the **rate is decreased** or rtsCounter becomes 0 and it is **switched on** every time the **rate is increased** or a failure occurs without it.

With the above behaviour AARF-CD is at least expected to perform better than ARF and AARF (where both have RTS-CTS enabled) in a single-user scenario where, losses are only due to channel errors.

3. Simulation Overview

To investigate the performance of the rate control algorithms discussed above, three different scenarios have been simulated in the NS3 Discrete-Event simulator[4] on an IEEE 802.11b network, in infrastructure mode.The description of each of the scenarios is given below.

3.1 Scenario Descriptions

In each of the below scenarios, an IEEE 802.11b network in infrastructure mode has been simulated. To simulate a realistic environment of a public WiFi hotspot, the access point (AP) is kept at a fixed height of 5 m and all the Station Nodes (STAs) are at variable height (between 0.5 -1 m) to depict laptops placed on tables and mobile devices held by people. Depending on the scenario, the users may be static or mobile. In each scenario, a saturated condition is assumed.Each node runs a UDP socket application with a data rate of 11 Mbps and a payload size of 1500 bytes. The node placement is ideal i.e without a hidden node condition. The propagation model used is the Log Shadowing model, unless specified otherwise.
Scenario 1: Single-User
The purpose of this scenario is to analyse the performance when only a single user is present and losses are due to channel errors only. Note that RTS-CTS is disabled here for ARF and AARF (since there is only a single user). Only 1 STA is present in the network and it moves away from the AP at a velocity of 2 m/s. The simulation is run for 80s and throughput of the node is measured in time intervals of 1s.

Scenario 2: Multi-user with Fixed RSS (Static Users)
The purpose of this scenario is to analyse the performance when multiple users are present and losses are due to collisions only. All the STAs are placed randomly in a 5x5x5 room. Here, a fixed RSS propagation model is used with RSS = -40 dBm.Thus each of them experiences a fixed RSS.This is done to observe the performance when losses are only due to collisions. The simulation duration is 10s.

Scenario 3: Mixed users (Static and Mobile)
The purpose of this scenario is to analyse the performance when losses are due to both channel errors and collisions. Multiple users are present, arranged in a circle of radius 20m as in scenario 2. But one of the users moves away from the AP at a fixed velocity of 2 m/s. The simulation duration is 10s.

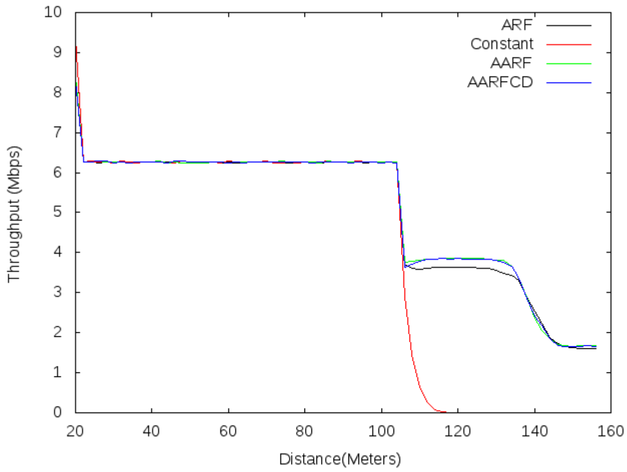


Figure 1: Plot of Throughput vs Distance from AP (Single User Scenario)

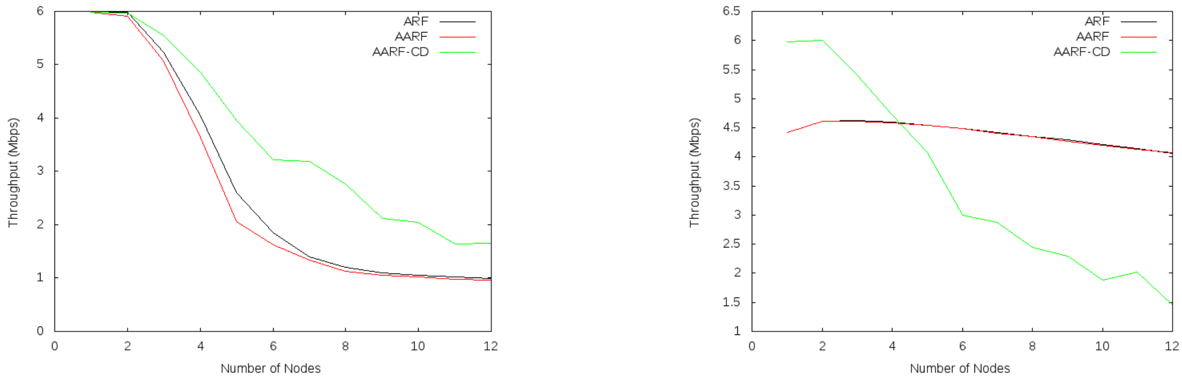
4. Simulation Results and Discussion

The simulation is executed on NS3 (version 3.24), running on Ubuntu 14.04 platform and Intel Core i5 CPU (2.5 GHz). The plot values are an average of 10 simulation runs.

4.1 Scenario 1 (Only Channel Errors)

The real-time throughput of the node is calculated over an interval of 1s based on the number of packets successfully transmitted during that interval. This value is plotted over distance traveled(meters) for the three rate algorithms in Figure 1. To give an impression of the effectiveness of rate adaptation, the results of the simulation using a fixed rate (11 Mbps) is also plotted. We can see that all the rate adaptation algorithms perform better than the constant rate algorithm, when the STA's distance from AP is such that the channel is no longer suitable for transmission at highest rate. The rate control algorithms force the PHY layer to switch to a lower rate (5.5 Mbps) from 110m to 140m. After 140m, the rate is switched to 2 Mbps. During the stable portion of the channel (110m to 140m), AARF and AARF-CD perform better than ARF because of the doubling of threshold for successful transmissions after every failure at the higher rate (see Sec 2.2,2.3). ARF, on the other continues to try a higher rate(11 Mbps) after every 10 successful transmissions at 5.5 Mbps which results in increased failures and lower average throughput.

Between AARF and AARF-CD , there is not much difference in performance during the stable region as their behaviour is the same with regards to doubling of the success threshold and RTS-CTS is disabled in this scenario for AARF(because of only 1 user). Although RTS-CTS is enabled for AARF-CD, as per protocol, it is disabled whenever there is a rate decrease and enabled only when a failure



(a) Basic Access (b) RTS-CTS Enabled

Figure 2: Plot of Throughput vs Number of Nodes for the Multi User Scenario

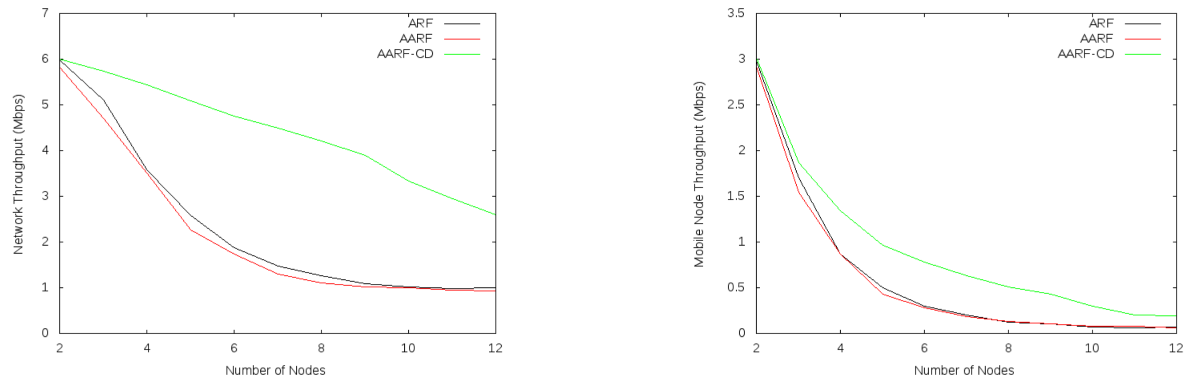
occurs without it. This happens when the STA attempts to send at a higher rate in between the stable region.(after a rate decrease). RTS-CTS is disabled later,as soon as the rtsCounter reaches 0. As per specifications in [3], the counter value rtsCounter in the initial stages is in the order of 2 or 4. Therefore, RTS-CTS handshakes don't occur too frequently to affect the overall throughput for AARF-CD. Hence, no significant performance difference is observed.

4.2 Scenario 2 (Only Collision Losses)

The aggregated throughput(Mbps) of the network (over the entire simulation duration) is calculated. The variation of this throughput is observed for the three algorithms, as the number of nodes are increased. Two cases are considered- Basic Access mode and RTS-CTS mode. Note that for AARF-CD, the transmission switches between the two modes based on the protocol and counter values (described in Sec 2.3) and not on the basis of size of payload (which is the case in ARF and AARF-CD). Figure 2(a) and 2(b) depict the variation of throughput with the number of nodes for Basic Access and RTS-CTS modes respectively. The values are a mean of 6 simulation runs.

For the Basic Access case, AARF-CD clearly has better performance,particularly when the contention increases causing more collision errors. In this scenario, channel errors do not happen as the nodes are static and at a fixed distance from the AP. Better performance of AARF-CD is explained due to the fact that it switches to RTS-CTS mode whenever collision errors occur whereas ARF and AARF operate in Basic Access throughout the simulation. Since RTS-CTS mode improves throughput during high contention, AARF-CD performs better.

For RTS-CTS mode, AARF-CD performs better than ARF and AARF only when the number of nodes are low i.e when collision errors do not occur. This is because AARF-CD disables RTS-CTS during the time when there are no collisions. When there are no collisions, RTS-CTS causes



(a) Average Network Throughput

(b) Average Throughput for Mobile Node

Figure 3: Throughput vs Number of Nodes for Mixed User Scenario(Basic Access)

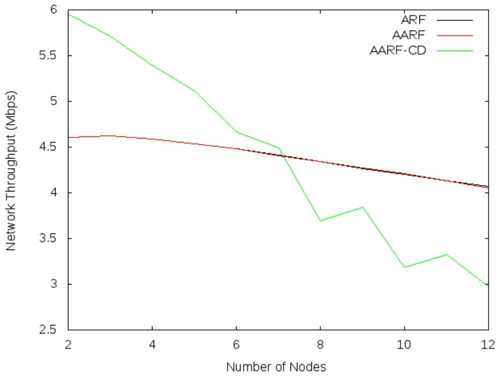
additional unnecessary overhead which leads to decreased throughput. But as soon as the contention increases, the use of RTS-CTS leads to better performance for ARF and AARF. We can observe a certain threshold number of nodes(4) beyond which AARF-CD performs worse than AARF and ARF. AARF-CD's performance drops because of the fact that it disables RTS-CTS occasionally (when rtsCounter reaches 0). This leads to an initial dip in throughput. But, since the maximum value of rtsCounter is doubled with each unsuccessful transmission (without RTS-CTS), the algorithm is expected to stabilize with time. The curve is more stable when the number of nodes reaches 11 and 12 (i.e more frequent collisions).

4.3 Scenario 3 (Both Collision and Channel Losses)

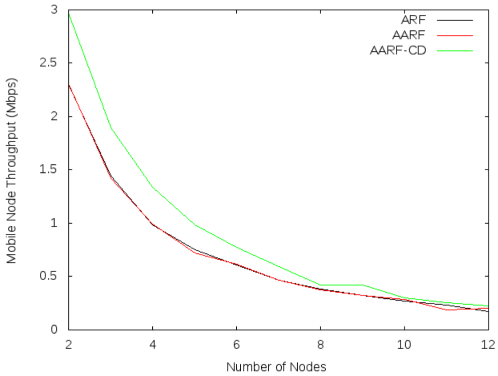
In the mixed-user scenario, two types of nodes are present- static and mobile. The main objective is to analyse which algorithm works best under conditions of both channel and collision errors. This is analysed by observing the average throughput of the mobile node,as the number of nodes(static) are increased in the network. For each of these cases, the performance under Basic access and RTS-CTS mode are also observed. Log distance propagation loss model has been considered.

Figure 3 shows the plot of throughput for the network and the mobile node, for the Basic Access case.The results are similar to the multi-user scenario, wherein AARF-CD is able to perform better than ARF and AARF due to the effective use of RTS-CTS during collision losses.

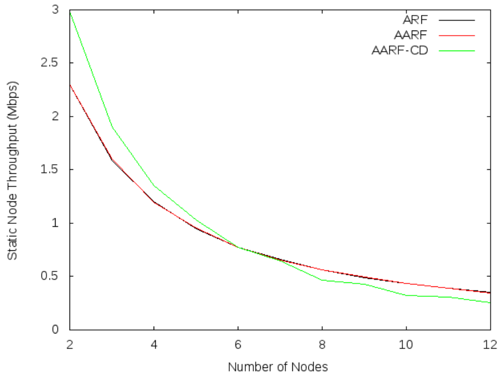
Figure 4 shows the plots when RTS-CTS is enabled for ARF and AARF. The general trend for overall network throughput observed is same as in Scenario 2. AARF-CD performs better upto a certain threshold number of nodes in the network (as seen in Figure 2b). But the threshold is higher(6 nodes). For the mobile node, AARF-CD always performs better even with higher number of nodes. At a point of high contention the throughput for all three algorithms nearly converges.This is an interesting



(a) Network Throughput



(b) Mobile Node Throughput



(c) Static Node Throughput

Figure 4: Average Throughput vs Number of Nodes for Mixed User Scenario(Basic Access)

result as it shows that AARF-CD is the best option under both collision and channel losses. For the static nodes, the average throughput per node shows similar behaviour as the aggregate network throughput but the performance difference between the three algorithms is much narrower. This shows that under pure collision scenarios, the per node average throughput achieved is almost the same for all the three algorithms.

5. Summary and Conclusion

In this project, three major IEEE 802.11 rate adaptation algorithms were studied- ARF, ARF and AARF-CD. The simulation was carried out for three different scenarios: Single-User (Mobile) ,Multi-User (Static) and Mixed-User (Static and Mobile). Based on the results in each of the scenarios, the following conclusions can be made: -

- ARF is not an effective algorithm when used in slow-varying channels. AARF and AARF-CD perform better since they use the history of unsuccessful transmissions to decrease the frequency at which they attempt to transmit at a higher rate.
- AARF-CD is most effective in scenarios where losses are only due to channel errors, because of its adaptive mechanism to enable RTS-CTS only when required (i.e during high contention). AARF and ARF are not adaptive in this sense and are unable to distinguish between collision and channel errors.
- In pure high contention scenarios, AARF-CD is not as effective as ARF and AARF as it disables RTS-CTS in the intial stages of its operation, leading to decreased throughput.
- In a scenario where both collisions and channel losses are present, AARF-CD is more effective than ARF and AARF. But, as the contention increases, their performance converges.
- As a recommendation, we can say that in practical scenarios, given these rate control algorithms, nodes must choose between the three depending on the scenario it is in. If it is static and in a crowded environment, it may choose between either ARF or ARF. If it is moving continuously, then it must switch to AARF-CD.

References

[1] A. Kamerman and L.Monteban *WaveLAN-II: A High-performance wireless LAN for the unlicensed band*, Bell-Labs Technical Journal, 118-133,1997

[2] M. Lacage and M.H.Manshaei and T.Turletti IEEE 802.11 Rate Adaptation: A Practical Approach *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*,2004

[3] Federico Maguolo and Mathieu Lacage and Thierry Turletti, Efficient collision detection for auto rate fallback algorithm, *In MediaWiN 2008*

[4] NSNAM, *The Network Simulator - ns-3*,http://www.nsnam.org

A. Appendix

Listing 1: NS3 Code

```
//Name : Varun Nair
//Student Number : 4504550

// Class NodeStatistics is used from the NS3 example rate-adaptation
//distance.cc By Matias Richart <mrichart@fing.edu.uy>


#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/applications-module.h"
#include "ns3/stats-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/buildings-propagation-loss-model.h"
#include "ns3/internet-module.h"
#include "ns3/enum.h"
#include "ns3/nstime.h"
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <iomanip>
#include <sstream>



using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("wifiRateAdaptation");

class NodeStatistics
{
public:
  NodeStatistics (NetDeviceContainer aps, NetDeviceContainer stas);

  void CheckStatistics (double time);

  void RxCallback (std::string path, Ptr<const Packet> packet, const Address &from);
  void SetPosition (Ptr<Node> node, Vector position);
  void AdvancePosition (Ptr<Node> node, int stepsSize, int stepsTime);
  Vector GetPosition (Ptr<Node> node);

```

```
  Gnuplot2dDataset GetDatafile ();

private:
  uint32_t m_bytesTotal;
  Gnuplot2dDataset m_output;
};

NodeStatistics::NodeStatistics (NetDeviceContainer aps, NetDeviceContainer stas)
{
  m_bytesTotal = 0;
}

void
NodeStatistics::RxCallback (std::string path, Ptr<const Packet> packet, const Address &fro
{
  m_bytesTotal += packet->GetSize ();
}

void
NodeStatistics::CheckStatistics (double time)
{

}

void
NodeStatistics::SetPosition (Ptr<Node> node, Vector position)
{
  Ptr<MobilityModel> mobility = node->GetObject<MobilityModel> ();
  mobility->SetPosition (position);
}

Vector
NodeStatistics::GetPosition (Ptr<Node> node)
{
  Ptr<MobilityModel> mobility = node->GetObject<MobilityModel> ();
  return mobility->GetPosition ();
}

void
NodeStatistics::AdvancePosition (Ptr<Node> node, int stepsSize, int stepsTime)
{
  Vector pos = GetPosition (node);
  double mbs = ((m_bytesTotal * 8.0) / (1000000.0 * stepsTime));
  mbs = ceilf(mbs*1000)/1000.0;
  mbs = mbs + 0.00;
  m_bytesTotal = 0;
  m_output.Add (pos.x, mbs);
  pos.x += stepsSize;
  SetPosition (node, pos);
```

```
  Simulator::Schedule (Seconds (stepsTime), &NodeStatistics::AdvancePosition, this, node,
}

Gnuplot2dDataset
NodeStatistics::GetDatafile ()
{
  return m_output;
}


int
main (int argc, char *argv[])
{
  double simulationTime = 10.0; //Simulation Time seconds
  double StartTime = 0.0;
  double StopTime = 10.0;
  double throughput = 0.0;
  double throughput_n1 = 0.0;
  double throughput_n2 = 0.0;
  double transmitTime = 0.0;
  double Successk = 2;
  std::string outputFileName = "single_user";
  //uint32_t appRxPackets = 0;
  Time delay = Seconds(0.0);
  std::string manager;
  double rss = -40;   //RSS Vaue for Fixed RSS Loss Model
  uint32_t rtsCtsThresh = 2500;  //RTS CTS Threshold
  //int steps = 10;
  int stepsSize = 2;
  int stepsTime = 1;



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


  bool verbose = false;


  //Command Line Arguments
  uint32_t nWifi802_11b = 18;     // Number of nodes on 802.11b
  std::string propModel = "fixed";
  std::string wifiMgr = "constant";
  std::string scenario = "multi";
  std::string rtscts = "n"; //Disabled by default
```

```



  CommandLine cmd;
  cmd.AddValue ("nWifi802_11b", "Total Number of 802.11b STA devices", nWifi802_11b);
  cmd.AddValue ("propModel", "Propagation Loss Model To : 'Random' for Random loss model,
  cmd.AddValue ("wifiMgr","Type of Rate Adaptation to use: 'constant(default)','aarf','arf
  cmd.AddValue ("rtscts","Enter y (Default) to enable RTSCTS else n",rtscts);
  cmd.AddValue ("scenario","Type of Scenario: single, multi or mixed",scenario);
  cmd.AddValue ("verbose", "Enable Wifi logging if true", verbose);

  cmd.Parse (argc,argv);

  NS_LOG_INFO ("Creating Topology");

  //Create STA and AP nodes
  NodeContainer staNodes802_11b;
  staNodes802_11b.Create (nWifi802_11b);
  //std::cout << "Total "<<nWifi802_11b<<" STA Nodes created.." << '\n';


  NodeContainer apNode;
  apNode.Create (1);
  NS_LOG_INFO ("AP Node created..");



 //Configure the PHY layer model (Default YANS)
  YansWifiChannelHelper channel;

 //Configure the YANS Channel parameters (Loss and Delay)
  channel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");

 if (scenario=="single")  //Adjust the simulation time for Single User Scenario
  {
    simulationTime = 70;
    StopTime = 70;
  }

 if (rtscts == "y")    // Enable RTS CTS
  rtsCtsThresh = 200;


 if (propModel == "Log")
 {
  channel.AddPropagationLoss ("ns3::LogDistancePropagationLossModel","ReferenceLoss",Doub
 }

 else
```

```
{
 channel.AddPropagationLoss ("ns3::FixedRssLossModel","Rss",DoubleValue(rss));
}


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

 phy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

 phy.SetErrorRateModel ("ns3::YansErrorRateModel");

 phy.SetChannel (channel.Create ());

 NS_LOG_INFO ("PHY channel created..");


 //Create the net device containers
 NetDeviceContainer staDevices802_11b;


 //Configure the PHY & MAC Layers for 802.11b and Install the Devices
  WifiHelper wifi;

 wifi.SetStandard(WIFI_PHY_STANDARD_80211b );


 //Configure the RATE CONTROL ALGORITHMS
 if (wifiMgr == "aarf")
   {
    manager = "ns3::AarfWifiManager";
    wifi.SetRemoteStationManager ("ns3::AarfWifiManager","SuccessK",DoubleValue(Successk)
   }
 else if (wifiMgr == "arf")
   {
    manager = "ns3::ArfWifiManager";
    wifi.SetRemoteStationManager ("ns3::ArfWifiManager","RtsCtsThreshold",UintegerValue(r
   }
else if (wifiMgr == "aarfcd")
   {
    manager = "ns3::AarfcdWifiManager";
    wifi.SetRemoteStationManager ("ns3::AarfcdWifiManager","SuccessK",DoubleValue(Success
   }

 else
   {
    manager = "ns3::ConstantRateWifiManager";
    std::string phyMode ("DsssRate11Mbps");
    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager","DataMode",StringValue(p
```

```
      }

    if (verbose)                    // If true, enable all logging components of WifiNetDevices
      {
        wifi.EnableLogComponents();
      }



    //Configure a non-QoS upper mac
    NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();
    Ssid ssid = Ssid ("Hybrid-ssid");
    mac.SetType ("ns3::StaWifiMac","Ssid", SsidValue (ssid),"ActiveProbing", BooleanValue (f
    staDevices802_11b = wifi.Install (phy, mac, staNodes802_11b);

    NS_LOG_INFO ("802.11b device configured..");


    //Configure the MAC layer for AP
    NetDeviceContainer apDevices;
    mac.SetType ("ns3::ApWifiMac","Ssid", SsidValue (ssid));
    apDevices = wifi.Install (phy, mac, apNode); //Adding 802.11b Net Device to AP



    //Mobility Configuration

    MobilityHelper mobilityAp,mobilitySta;
    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();

    positionAlloc->Add (Vector (0.0, 0.0, 5));
    mobilityAp.SetPositionAllocator (positionAlloc);
    mobilityAp.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobilityAp.Install (apNode.Get(0));


    //Placing of the Nodes
   if (propModel == "fixed")                    //For Scenario 2
    {
       //Configure the attributes of the RandomBox3dPositionAllocator
       double min = 0.0;
       double max = 5.0;
       double minz = 1;
       double maxz = 2;
       Ptr<UniformRandomVariable> x = CreateObject<UniformRandomVariable> ();
       Ptr<UniformRandomVariable> z = CreateObject<UniformRandomVariable> ();
       x->SetAttribute ("Min", DoubleValue (min));
       x->SetAttribute ("Max", DoubleValue (max));
       z->SetAttribute ("Min", DoubleValue (minz));
```

```
      z->SetAttribute ("Max", DoubleValue (maxz));

      mobilitySta.SetPositionAllocator ("ns3::RandomBoxPositionAllocator",
                                        "X", PointerValue (x),
                                        "Y", PointerValue (x),
                                        "Z", PointerValue (z));

  }

   else                                    // For scenario 1 & 3
  {
    Ptr<ListPositionAllocator> positionAllocSt = CreateObject<ListPositionAllocator> ();
     double xpos = 20;                     // Radius of 20 m
     double ypos;

     //Arrange the nodes in a circle
     for (uint32_t i = 1; i<=nWifi802_11b;i++)
    {
       ypos = sqrt(400 - pow(xpos,2));
       positionAllocSt->Add (Vector (xpos, ypos, 0.5));
       xpos = xpos - 1;
    }

    mobilitySta.SetPositionAllocator (positionAllocSt);
   }


       mobilitySta.SetMobilityModel ("ns3::ConstantPositionMobilityModel");

       mobilitySta.Install (staNodes802_11b);

       if (scenario == "mixed")              // Configure the 1st node to move with Veloci
     {
       Vector3D pos = Vector3D(0.0,0.0,20);
       Vector3D vel = Vector3D(2.0,0.0,0.0);  // 2 m/s in the + x dir
       mobilitySta.SetMobilityModel ("ns3::ConstantVelocityMobilityModel","Position", Vecto
       mobilitySta.Install (staNodes802_11b.Get(0));
     }

       //Statistics counter
       NodeStatistics atpCounter = NodeStatistics (apDevices, staDevices802_11b.Get(0));

       if (scenario == "single")
     {
      //Move the STA by stepsSize meters every stepsTime seconds
       Simulator::Schedule (Seconds (0.5 + stepsTime), &NodeStatistics::AdvancePosition, &a
 stepsTime);

     }
```

```


  //Install the Internet stack in all the Nodes
  InternetStackHelper stack;
  stack.Install (apNode);
  stack.Install (staNodes802_11b);


  //Configure the IPv4 Addresses
  Ipv4AddressHelper address;
  NS_LOG_INFO ("Assign IP Addresses.");

  address.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer apInterfaces, staInterfaces802_11b;
  apInterfaces = address.Assign (apDevices);
  staInterfaces802_11b = address.Assign (staDevices802_11b);

//Configure Routing
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  //Create the OnOff application to send UDP datagrams of size
  // 2000 bytes at a rate of 1.5 Mb/s
  NS_LOG_INFO ("Create Applications.");
  uint16_t port = 9;   // Discard port (RFC 863)

  OnOffHelper onoff ("ns3::UdpSocketFactory",
                     Address (InetSocketAddress (apInterfaces.GetAddress (0), port)));
  onoff.SetConstantRate (DataRate ("11Mb/s"),1500); //11 Mbps transfer mode with 1500 byte
  ApplicationContainer apps = onoff.Install (staNodes802_11b);
  apps.Start (Seconds (StartTime));
  apps.Stop (Seconds (StopTime));


// packet sink to receive these packets
  PacketSinkHelper sink ("ns3::UdpSocketFactory",
                         Address (InetSocketAddress (Ipv4Address::GetAny (), port)));
  apps = sink.Install (apNode);
  apps.Start (Seconds (StartTime));
  apps.Stop (Seconds (StopTime));

//////------------STATS and DATA Collection--------------/////////

  if (scenario == "single")
    {
      //Register packet receptions to calculate throughput
       Config::Connect ("/NodeList/1/ApplicationList/*/$ns3::PacketSink/Rx",
                        MakeCallback (&NodeStatistics::RxCallback, &atpCounter));
    }

```

```


  //Set up FlowMon
  FlowMonitorHelper flowmon;
  Ptr<FlowMonitor> monitor = flowmon.InstallAll();



  //Configure Simulation
  Simulator::Stop (Seconds (simulationTime));
  Simulator::Run ();

 if (scenario == "single")
 {
  std::ofstream outfile (("throughput_" + outputFileName + wifiMgr + ".dat").c_str ());
  Gnuplot gnuplot;
  gnuplot.AddDataset (atpCounter.GetDatafile ());
  gnuplot.GenerateOutput (outfile);
 }

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

  for (std::map<FlowId, FlowMonitor::FlowStats>:: const_iterator i = stats.begin(); i!= st
 {
   Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);

   transmitTime = i->second.timeLastRxPacket.GetSeconds() - i->second.timeFirstTxPacket.G
   if (transmitTime!= 0)
   throughput = throughput + (i->second.rxBytes * 8.0 / (transmitTime)/1024/1024);
   else
   continue;

   if (t.sourceAddress == "10.1.1.2")
      throughput_n1 = (i->second.rxBytes * 8.0 / (transmitTime)/1024/1024);
   else if (t.sourceAddress !="10.1.1.1" && t.sourceAddress !="10.1.1.2")
      throughput_n2 = throughput_n2 + (i->second.rxBytes * 8.0 / (transmitTime)/1024/1024
   else
      throughput = throughput;


   if (i->second.rxPackets != 0)
    delay = delay + (i->second.delaySum / i->second.rxPackets);
   else
    continue;
 }

    std::ofstream avg("avg.dat");  //Output stream for average throughput
```

```
      std::ofstream mob("mob.dat");  // Output stream for mobile node throughput
      std::ofstream sta("sta.dat");  // Output stream for stationary node throughput
      if (scenario =="mixed")
     {
      avg << std::fixed;
      avg << std::setprecision(2);
      mob << std::fixed;
      mob << std::setprecision(2);
      sta << std::fixed;
      sta << std::setprecision(2);
      avg << nWifi802_11b <<" "<<throughput<<'\n';     //Aggregate Network Throughput
      mob << nWifi802_11b <<" "<<throughput_n1<<'\n'; //Average Throughput of Mobile Node
      sta << nWifi802_11b <<" "<<throughput_n2/(nWifi802_11b-1)<<'\n'; //Average Node Throug
     }
      else if (scenario =="multi")
     {std::cout << std::fixed;
      std::cout << std::setprecision(2);
      std::cout << nWifi802_11b <<" "<<throughput<<'\n';
     }

    Simulator::Destroy ();


    return 0;
}
```