NS3 802.11b Throughput Performance

Freek van Tienen
4094123

28-04-2016

Abstract

With the help of NS3 simulations can be performed for several networking protocols. One of these protocols is the 802.11b protocol. This protocol allows us to communicate wireless between a STA(Station) and AP(Access Point). It is being used widely, and the most commonly used frequency range of 2.4GHz is becoming very crowded. This extensive use of the same frequencies using DSSS(Direct Sequence Spread Spectrum), makes it difficult to ensure a consistent performance in the 802.11b protocol. The amount of STA's using a single AP greatly affects the throughput of the STA's and therefor the performance. In this document we will analyse the performance impact of the amount of STA's on a single AP using NS3.

1 Problem description

The 802.11b protocol is a commonly used protocol for wireless communications between AP en STA's. When dealing with a lot of STA's on a single AP the overall performance and performance per STA should be stable enough to ensure fair access for every STA. To analyse these situations where a lot of STA's are using a single AP with a lot of data transmitted from the STA's to the AP several techniques can be used. One of the most commonly used techniques is the use of a simulator like NS3.

With the use of NS3 a full simulation of the 802.11b protocol can be done on both the physical and the software level. To simplify our problem we focus mainly on the software level and thus try to avoid all problems like Signal loss and hidden/exposed nodes. Thus for our problem we use a fixed signal strength (and thus quality) which is equal between all STA's. We also assume that no hidden and exposed nodes exists during communication. Next to that we assume that no one is interfering on our frequency and there is only one AP.

With all the above mentioned assumptions we want to analyse the throughput performance in the network under several conditions. We want to analyse what the amount of STA's in a network would mean in comparison to the data rates these STA's communicate to the AP. Also we want to take a look at what the influence of a change in packet size is for the total and average throughput in the network. In the next chapter we describe how we implement such simulation in NS3 and automate these result to improve replicability.

2 Design and implementation

The basic 802.11b simulator consists of a single *c++* NS3 program, which constructs all simulation data. It has several options which change variables of the experiment, these include the amount of STA's, simulation time, data rate of the STA's, packet size, etc. During each run these variables are changed and eventually this data is then analysed.

To make sure that each of these runs generate use different randomisation a *runNumber* is also set as variable during each run. Next to the seed of the random number generator which is set to the time. But because I wanted it to be designed to run in parallel, to make the simulation time less, this wasn't enough hence the *runNumber*. The implementation can be seen in Figure 1.

```
/* Make sure the simulation is random */
time_t timev;
time(&timev);
RngSeedManager::SetSeed (timev);
RngSeedManager::SetRun (runNumber);
```

Figure 1: Randomness between simulations.

For the simulation of the 802.11b a default *YansWifi* physical and channel is used. The *RxGain* of the physical layer is chosen to be 0, to make sure the no difference exists between the different STA's. For the channels a *ConstantSpeedPropagationDelayModel* is chosen with a fixed loss of $-80dB$. This results in the same receiver characteristics for each STA, to obtain a fair result. How this is setup using NS3 can be seen in Figure 2.

```
/* Create WiFi and Internet stack */
NS_LOG_INFO ("Installing WiFi and Internet stack.");
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b); // Set to 802.11b

YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.Set ("RxGain", DoubleValue (0));
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default ();
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::FixedRssLossModel", "Rss", DoubleValue
    (-80));
wifiPhy.SetChannel (wifiChannel.Create ());
```

Figure 2: Setting up Physical and Channels.

Then to enable the mobility between STA's and obtain fair results without any hidden or exposed, it is chosen to set both the AP and the STA's at a fixed position. The position of the AP is placed 3 meters above the STA's and also separated 3 meters in x and y direction. All the STA's are fixed at the same position, and because the constant speed propagation delay model all the STA's should have the same receiving/transmitting characteristics. This allocation of the positions in the mobility model can be seen in Figure 3.

```
/* Setup mobility */
NS_LOG_INFO ("Defining mobility.");
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<
    ListPositionAllocator> ();

positionAlloc->Add (Vector (0.0, 0.0, 3.0));
for(uint16_t i = 0; i < staCount; i++)
 positionAlloc->Add (Vector (3.0, 3.0, 0.0));

mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (apNode);
mobility.Install (staNodes);
```

Figure 3: Setting up the mobility model.

To generate data in the network to measure the throughput a *OnOff* application is used. This application enables us to constantly transmit data for a certain amount of *on* time versus the amount of *off* time. To simulate a high amount of throughput through the network we set the *on* and *off* time such that it continuously transmits data without any *off* time. For each STA in the network such an application will be generated and will transmit data to the AP. Both the *DataRate* and *PacketSize* are variables for our experiment and are changed with the arguments of our simulation. This application is very easy to implement and will show us the performance of throughput vey easily, because constant data is transmitted between the STA's and the AP. To make sure that no packets are lost this application is run on the TCP stack instead of an UDP stack. The implementation of this application can be seen in Figure 4.

```
// Create the STA application to send packets
OnOffHelper staOnOff ("ns3::TcpSocketFactory", staAddress);
staOnOff.SetAttribute ("OnTime",  StringValue ("ns3::ConstantRandomVariable[
    Constant=1]"));
staOnOff.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[
    Constant=0]"));
staOnOff.SetAttribute ("PacketSize", UintegerValue (packetSize));
staOnOff.SetAttribute ("DataRate", StringValue (dataRate));
staOnOff.SetAttribute ("Remote", AddressValue (apAddress));
```

Figure 4: The STA application.

Since we now have all the elements for a simulation we need to generate some statistics about the throughput(and thus the performance) of our network under various conditions. To do so a *csv* file is opened during the start of the experiment and with the help of a *flowMonitor* some statistics are generated. For each network flow the statistics are generated using the method in Figure 5. These statistics are then combined which will give us the total throughput, average throughput, variance in throughput between STA's and standard deviation of throughput between STA's This calculation can be seen in 6. These total statistics are then saved to the *csv* file.

```
float throughput = i->second.rxBytes * 8.0 / simulationTime / 1024 / 1024;
throughputSum += throughput;
throughputSumSq += powf(throughput, 2);
```

Figure 5: Per STA statistics.

```
double throughputMean = throughputSum / staCount;
double throughputVariance = (throughputSumSq - (powf(throughputSum, 2) /
    staCount)) / staCount;
double troughputStd = sqrtf(throughputVariance);
```

Figure 6: Total statistics.

To generate all these statistics for different data rates, packet sizes and STA's a simple python script is made. This script runs multiple simulations in parallel to speed up the process. The generation of these different runs is done trough simple for loops which can be seen in Figure **??**. It will do 5 different runs for each set of variables, to remove outliers. It will test on various data rates and 2 packet size(512 bytes and 1024 bytes) and this will show us what the difference in packet size means in comparison with throughput. It will start with one single STA and will go up to 39 STA's and analyse the total throughput and average throughput for each of the runs and will save these in a new *csv* file.

```
data_rates = ['5Mbps', '2Mbps', '1Mbps', '500Kbps', '100Kbps']  # Different
    data rates for simulation
packet_sizes = [512, 1024] # Different packet sizes for simulation

# Now lets build our test set
for packet_size in packet_sizes:
  for data_rate in data_rates:
    for stas in range(1, 40):
      for run in range(0, 5):
        addRun(run, stas, data_rate, packet_size)
```

Figure 7: Total statistics.

Then as last a new python script will be run to generate the graphs to visualise the in this paper. For both the total throughput and the average throughput also the variance and standard deviation are calculated between all the 5 different runs. But after analysing these result for the average throughput it seemed that the variances was almost 0 and thus the conclusion was made to leave out these results since they didn't add any value.

The source code for all these scripts can be found on GitHub(`https://github.com/fvantienen/wireless-practical`), including the results.

3 Results

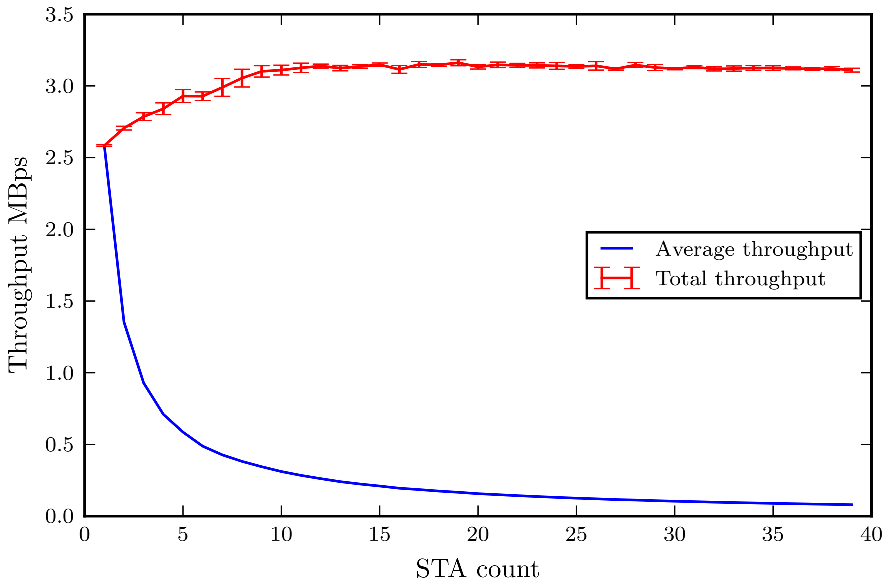

Figure 8: 5Mbps troughput curve with packet size of 1024 bytes.

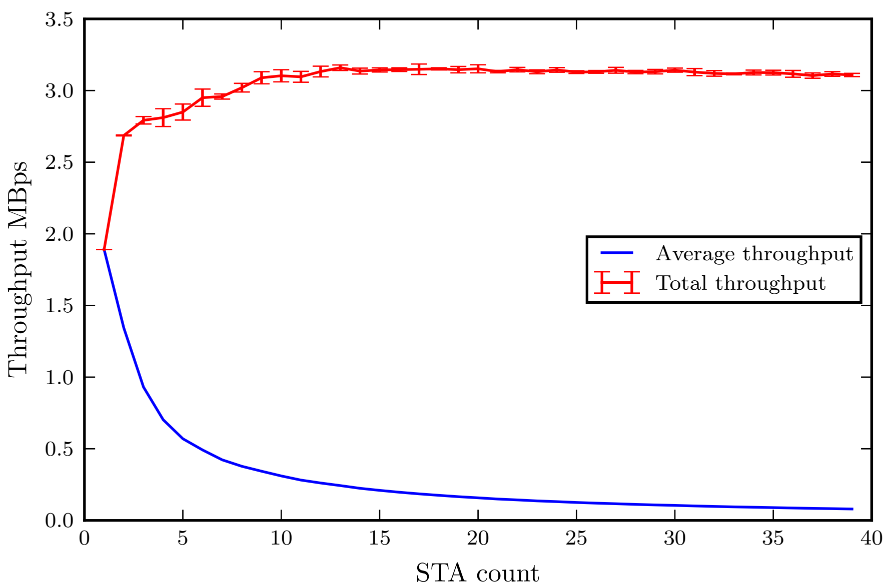

Figure 9: 2Mbps troughput curve with packet size of 1024 bytes.

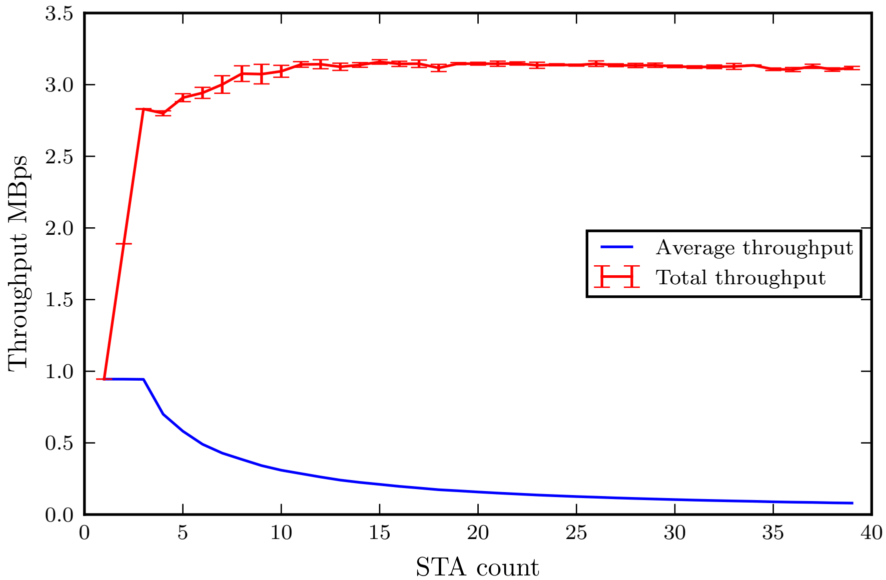

Figure 10: 1Mbps troughput curve with packet size of 1024 bytes.

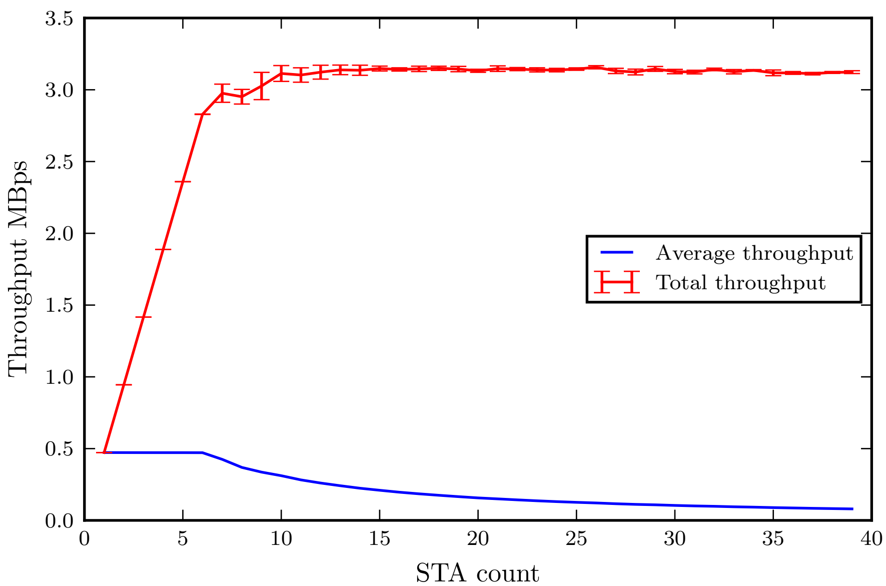

Figure 11: 500Kbps troughput curve with packet size of 1024 bytes.

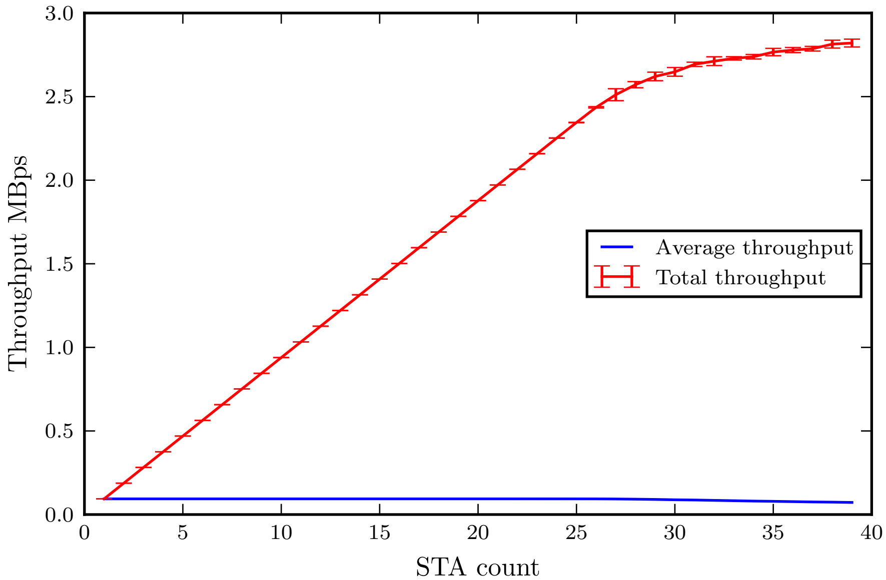

Figure 12: 100Kbps troughput curve with packet size of 1024 bytes.

4 Conclusion