EE 595 Experiment 1

Phy, Mac, and Queue Network Simulations

I. Background

This experiment builds on Experiment 0 from the first week in class. Students may want to review the writeup from that Experiment and solution as background.

In this experiment, students will use ns-3 to work with additional concepts covered in the course lectures:

- Monte Carlo simulations
- A two-state path loss model
- A basic queuing model
- A contention-based MAC protocol

Important note: These simulations require Git changeset c250abc4 or later of ns-3-ee-595, and a clean version of `link-performance.cc`. On your own repository, you can take the following steps to ensure that you are on the right version. Suppose that you previously checked out a version of the `course` branch, but you made changes to the program to complete the first assignment. You may not want to throw away those changes but save them for future reference. You can take the following steps:

1) commit your past changes: `git commit -a -m"Experiment 0 changes"`
2) fetch the update from the course repository: `git fetch origin`
3) checkout a new branch, based on course:
   ```
   git checkout -b experiment1 origin/course
   ```
4) confirm that you see changeset 0xc250abc4 (Friday Jan 25) in your history: `git log`

Your local changes to `course` are now stored in your local branch copy named `course`, and you can check out this branch at a future date with the command `git checkout -b course`. You can also observe the diff between your new `experiment1` branch and course by typing `git diff course`.

Rather than the above, you could also just start over in a fresh directory with a fresh clone of the repository, although it takes more disk space to completely clone and build again.

II. PHY Link Models (continued)

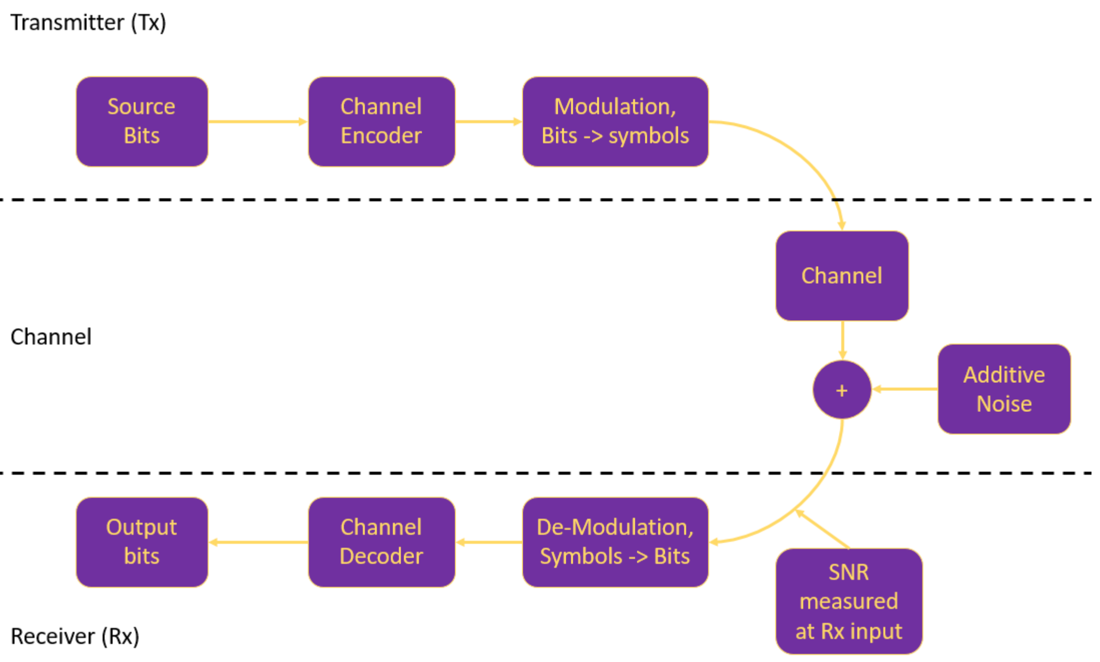

Fig. 1: Flowchart of Phy Link Model

A. Introduction

Recall that in Experiment 0, we walked you through a process flow to develop a packet level abstraction for use in ns-3, based on one of the most simple modulation types (BPSK), without error correcting coding, and one of the most simple path loss models (Friis). With these simple models, one does not even need to resort to simulation to obtain predictions of performance, because the math can easily be solved in closed form or numerically. However, in ns-3, we can plug in more sophisticated models to simulate less-idealized scenarios.

In ns-3, there are models to represent the modulation and coding used in 4G LTE and contemporary Wi-Fi, but the underlying principles that are illustrated in the SimpleWireless models can be used to later understand the elements that have evolved to today's technology.

Recall that we converted received power to received signal to noise ratio (SNR), and then used an analytical formula to convert our SNR estimate to a bit error ratio (BER). From there, we used the assumption that each bit is errored independently to derive an estimate of the PER.

In the next section, we will replace a few of the previous components. We will use a `LogDistance` propagation model and a new `TwoState` propagation loss model, and work with the notion of a link budget. We will do so while exploring the concept of Monte Carlo simulations in the following exercise.

B. Principles of Monte Carlo Simulation

In simulations, we run models that are fed with input data containing some elements of randomness, and draw conclusions about the output, but a crucial question to be able to answer is whether one can make any statistical claims about the accuracy and precision of the simulation results. How do people pick the number of simulation runs in practice? Usually these numbers are set to an arbitrarily large number (such as, e.g. "We ran 10,000 simulations of the upcoming Super Bowl and found that the Rams won 55.2% of the time."). In practice, barring other guidance, you should look at the sample mean and variance of the estimators you are monitoring, and continue to run until the variance is low, and report also some measure of the variance with your averages. It turns out, however, that if our trials are independent, identically distributed Bernoulli trials, we can rest on some formal results to allow us to select the number of runs necessary to reach a given level of precision with our estimates. The colloquial term for this is 'Monte Carlo' simulations.

Running Monte Carlo (statistical) simulations that provide performance bounds is based on the Weak Law of Large Numbers (WLLN), which can be understood in our context as follows. If a total of N packets is sent in a simulation run of which N_e are dropped (deemed incorrect due to channel errors), then the empirical estimate of the PER is given by $P\hat{E}R = \frac{N_e}{N}$ (the 'hat' over a quantity is a standard notation in mathematical statistics for an estimate of the quantity and is thus a random variable). By WLLN, $P\hat{E}R \xrightarrow{\mathcal{P}} PER$ as $N \to \infty$. What this says is that for sufficiently large N, the estimate $P\hat{E}R$ is arbitrarily close to the true PER in a probabilistic sense (i.e. the convergence statement is to be interpreted as follows: the probability that the deviation of the estimate $P\hat{E}R$ from the true value can be made arbitrarily small).

This leads to the obvious question: how large does N have to be such that the estimate $P\hat{E}R$ is arbitrarily close to the true value? For example, if the target probability of error is 10^{-3}, then if only a hundred packets went sent through the system, it is quite likely to observe a $P\hat{E}R = 0$, on average, we expect to send 1000 packets to observe one error event (but again, with a non-zero probability, we will have runs with 0 errors).

A principle of good statistical simulation is define a *confidence interval* around the estimated $P\hat{E}R$, that provides guidelines for how large N must be such that the estimate $P\hat{E}R$ is within a certain distance from the true PER for a specified level of confidence (typically, 95%). Let $X_1, X_2..., X_N$ denote a sequence of binary $0-1$ independent, identically distributed (i.i.d) random variables that represent the output of i-th packet transmission, i.e. $X_i = 1$ implies it is received in error (with prob. PER), and 0 otherwise (correctly received with prob. $1 - PER$). Hence $P\hat{E}R = \frac{1}{N}\sum_{i=1}^{N} X_i$. Then by well-known properties of the Bernoulli distribution $E[X_i] = E[X] = PER$ and thus $Var[X_i] = Var.[X] = PER(1 - PER)$. This yields $E[P\hat{E}R] = PER$, $Var.[P\hat{E}R] = \frac{PER(1-PER)}{N} = \frac{Var.[X]}{N}$. By the Central Limit Theorem, the distribution of $\frac{\sqrt{N}[P\hat{E}R-PER]}{\sqrt{Var.[P\hat{E}R]}}$ converges (in limit $N \to \infty$) to the standard $N(0,1)$ p.d.f (probability density function), i.e.

$$\lim_{N\to\infty} P\left(\frac{\sqrt{N}[P\hat{E}R - PER]}{\sqrt{Var.[X]}} < y\right) = \phi(y) \tag{1}$$

where $\phi(y) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{y} e^{-\frac{u^2}{2}}\,du$ is the cumulative distribution function (c.d.f) of the standard Normal random variable. A 95% confidence interval for the absolute deviation $|P\hat{E}R - PER|$ corresponds to $y = 1.96$ (from standard Normal c.d.f tables), and hence in order to obtain a precision in the estimate such that the error is $x\,PER$ (where $x = 10^{-a}$ represents a places of relative decimal accuracy desired), implies that we require $x\,PER = \frac{Var.[X]y}{\sqrt{N}}$. Hence solving, the required $N = \left(\frac{1.96}{x}\right)^2 \frac{1-PER}{PER}$. The Table I gives some guidance for how many packets are needed to satisfy this condition, using the above result.

PER	N	%prob confidence int.	X (error bar fraction)
0.5	384	95	0.1
0.1	3457	95	0.1
0.05	7299	95	0.1
0.01	38032	95	0.1

TABLE I: Number of packets N needed for different PERs.

Since experiment 0, we have extended the `link-performance.cc` program to provide an option for configuring two other propagation loss models. The first that we will use is the `LogDistancePropagationLossModel`. You can find online documentation at https://www.nsnam.org/docs/release/3.29/models/html/propagation.html and at https://www.nsnam.org/docs/release/3.29/doxygen/classns3_1_1_log_distance_propagation_loss_model.html#details.

In the documentation, one can observe that there are three Attributes that control the behavior: `Exponent`, `ReferenceDistance`, and `ReferenceLoss`. We covered these concepts in class. Notice that the default value of Exponent is 3, the default reference distance is 1m, and the default reference loss is 46.6777 dB. Notice also that there is no frequency parameter; the frequency enters into the model indirectly via the other parameters. If you create such an object and insert it into the simulation, you will get the above default values.

We have added an option to the `link-performance.cc` but would like to explain some aspects of configuration that may be helpful to you later, and to change these defaults in this week's assignment.

In ns-3, there are several ways to change underlying configuration values from their default values. The first is to create a command-line argument and then write some supporting code to convert that argument into underlying configuration values. A second way is to use a built-in facility of the underlying ns-3 attribute system to use the command line processing capability without adding any code to your program. We will show you an example, and then you can use this technique yourself.

The program has a new option `--lossModelType`:

```
$ ./waf --run 'link-performance --PrintHelp'
Program Options:
    --distance:       the distance between the two nodes [25]
    --maxPackets:     the number of packets to send [1000]
    --packetSize:     packet size in bytes [1024]
    --transmitPower:  transmit power in dBm [16]
    --noisePower:     noise power in dBm [-100]
    --frequency:      frequency in Hz [5e+09]
    --lossModelType:  loss model (Friis or LogDistance) [Friis]
    --metadata:       metadata about experiment run []
```

This string controls the selection of the loss model (below, we extend it further to add yet another loss model type).

Let's look at how this is implemented in the program; the below excerpts the relevant code from `contrib/simple-wireless/examples/link-performance.cc`:

```
  std::string lossModelType = "Friis";
  ...
  CmdLine cmd;
  ...
  cmd.AddValue("lossModelType","loss model (Friis or LogDistance)",lossModelType);
  ...
    Ptr<SimpleWirelessChannel> channel = CreateObject<SimpleWirelessChannel> ();
  if (lossModelType == "Friis")
    {
      Ptr<FriisPropagationLossModel> lossModel = CreateObject<FriisPropagationLossModel> ();
      lossModel->SetFrequency (frequency);
      channel->AddPropagationLossModel (lossModel);
    }
  else if (lossModelType == "LogDistance")
    {
      Ptr<LogDistancePropagationLossModel> lossModel = CreateObject<LogDistancePropagationLossModel> ();
      channel->AddPropagationLossModel (lossModel);
    }
  else
    {
      std::cerr << "Error, loss model " << lossModelType << " is unknown " << std::endl;
    }
```

Let's suppose that you need to change the exponent. There are a few techniques to do it.

1) modify the attribute directly on the object. Notice in the public API of the class, there are two methods:

```
SetPathLossExponent (double n)
SetReference (double referenceDistance, double referenceLoss)
```

You can modify and recompile link-performance.cc after inserting statements such as these (e.g. to set to an exponent of 2.75, a reference distance of 10 m, and a reference loss of 45 dB):

```
    ...
    Ptr<LogDistancePropagationLossModel> lossModel = CreateObject<LogDistancePropagationLossModel> ();
    lossModel->SetPathLossExponent (2.75);
    lossModel->SetReference (10, 45);
    channel->AddPropagationLossModel (lossModel);
    ...
```

2) change the default for all such objects in the first part of the program. This uses a `Config::SetDefault` statement, whose syntax looks like this:

```
  ...
  g_maxPackets = 1000;
  Config::SetDefault ("ns3::LogDistancePropagationLossModel::Exponent", DoubleValue (2.75));
  CommandLine cmd;
  ...
```

Again, you need to recompile the program after the above changes.

3) The command line argument facility allows one to change these defaults on attributes without rebuilding the program. The syntax is:

```
$ ./waf --run 'link-performance --ns3::LogDistancePropagationLossModel::Exponent=2.75'
```

C. Question

1) You must design an uncoded BPSK-based communications link to provide coverage at a distance of 100m. The noise power is -100 dBm, packet size is 1024 bytes, frequency is 5 GHz. Someone has previously characterized the propagation environment and determined that it follows a log-distance model, where the parameters are an exponent of 2.5, a reference distance of 1m, and a reference path loss at that reference distance of 50 dB. The target PER is 10^{-3}, and you are asked to provide for a link margin of 5 dB to protect against possible fades or interference. Use the `link-performance.cc` program to find the transmit power by experimentally determining the power required to provide the target PER with a 95% confidence interval that

the PER is within 10% of the true value (i.e., corresponding to an X of 0.1 from Table I), and then account for the requested link margin in your suggested transmit power.

III. Two state propagation loss model

Many real-world channel conditions are modelled as having a 'good' state and a 'bad' state (in reality, the actual system does not reduce to two discrete states but there may be two dominant modes in the distribution of data, such as 'clear line-of-sight' or 'blocked').

The two state models channel conditions over time using a two state Markov chain as shown in figure 2. The two state model serves as a simple way to describe time varying channels such as Rayleigh fading channels.

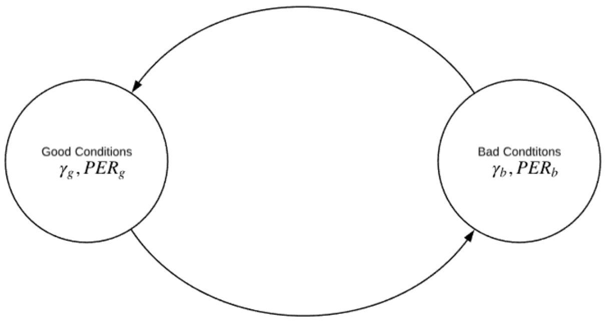

Fig. 2: example of 2 state Markov model

The channel remains in each state for an exponentially distributed amount of time with mean Gamma γ_g or γ_b depending on if the channel is in the "good" state or the "bad". In the "good" state the channel has a PER of PER_g, while in the "bad" state the channel has a PER of PER_b. PER_g is usually some number close to 0 and PER_b is some number closer to 1.

We can find the time average throughput of the two state model by considering the average throughput of each state separately then summing them proportionally to the time spent in each state. So the throughput of the good state is given by PER_g and the throughput for the bad state is PER_b. this means that the overall throughput is:

$$throughput = PER_g\Big(\frac{\gamma_g}{\gamma_g + \gamma_b}\Big) + PER_b\Big(\frac{\gamma_b}{\gamma_g + \gamma_b}\Big) \tag{2}$$

In our model, we can define the throughput as the amount of data received per unit time. Since errors lead to a loss of the packet and the packet is not retransmitted, when the PER is higher, the throughput will be lower.

We have added a new two-state propagation loss model into the simple wireless module on the course branch. The files are at
`contrib/simple-wireless/model/two-state-propagation-loss-model.*`.

The model can be used in the 'link-performance.cc' program by specifying a lossModelType of `"TwoState"`.

Again, there are four attributes of interest in this model:

- `PerG`: PER in good state (default 0.001)
- `PerB`: PER in bad state (default 0.01)
- `GammaG`: Mean time in good state (default 10 seconds)
- `GammaB`: Mean time in bad state (default 10 seconds)

In the below question, you can use one of the previously discussed approaches (for LogDistance loss model) to alter these values. As a hint, the following command will allow you to run with a gammaG of 1 second, gammaB of 10 seconds, PerG of 0.1, and PerB of 0.9:

```
$ ./waf --run 'link-performance --lossModelType=TwoState \
--ns3::TwoStatePropagationLossModel::PerG=0.1 \
--ns3::TwoStatePropagationLossModel::PerB=0.9 \
--ns3::TwoStatePropagationLossModel::GammaG=1s \
--ns3::TwoStatePropagationLossModel::GammaB=10s'
sent 1000 rcv 156 drop 844 per 0.844 error 0.02249
```

This is a lot of typing, so you may want to create a script (or modify and reuse the existing run-link-performance.sh program to generate the plot). Note also that you can still change the `MaxPackets` argument to have the program send more packets and reduce the error bars as needed.

If you want to observe the internal operation of the path loss model, you can enable the ns-3 log component "TwoStatePropagationLossModel".

A. *Question*

2) Simulate a two state channel with $\gamma_g = 1, 2, 5, 10, 20, 50, 100$ and $\gamma_b = 10$ with $PER_g = .1$ and $PER_b = .9$. Plot the long term average throughput as a function of the ratio γ_g/γ_b

IV. Simple Queuing: M/M/1/C Queue

When multiple packets arrive at a server, packets wait in queue according to some discipline - typically first-come, first served (FCFS) - to process the arrivals. Queue operations are denoted by Kendall's notation that captures three factors: a) the input or arrival process, b) the server or departure process and c) the # of servers. The most canonical queue studied is the M/M/1/C queue, where: ”M” indicates a Markovian arrival process, i.e. a Poisson process whereby packet inter-arrival times are exponentially distributed with mean $\frac{1}{\lambda}$ (or equiv. the arrival *rate* in packets/sec is λ). The second ”M” indicates packet service time (the duration from beginning to end of service), that is also distributed exponentially with mean service time μ; the service time is indep. of the inter-arrival process. The ”1” indicates that there is only one server and ”C” denotes the buffer capacity (i.e. a maximum of C packets may wait in queue and any further arrivals are denied entry/lost). An M/M/1/C queue abstraction is shown in Figure 3.

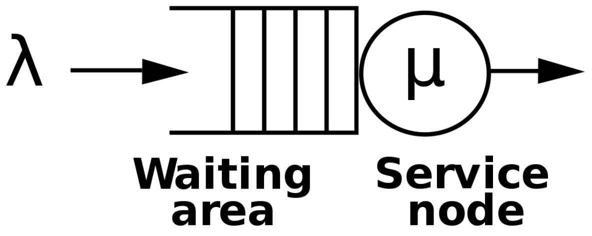

Fig. 3: Example of M/M/1 queue

Clearly, the queue size (# customers waiting for service) varies over time, as a function of the arrival and service process. Time intervals for which the queue is empty implies that the server is (temporarily) idle, whereas the periods when the queue is full (i.e. # in queue equals the capacity C) lead to packet drops for any subsequent arrivals. Both of these can be undesirable in a communications system (one would like to minimize the probability of both occurrences) and are thus the object of queue management policies. As we will see, attempting to minimize server idle fraction negatively impacts the other - thereby setting off a trade-off. The steady-state queue distribution for M/M/1/C queue is obtained by consideration the analysis of M/M/1/∞ (i.e. infinite size queue, and hence nothing is dropped) first and then simply re-normalizing the results for finite (C) size queue. From the queue distribution, all statistics of interest - such as average packet latency, # of packets in queue, blocking probability as function of finite (C) queue size etc. can be obtained.

M/M/1/∞ Queue

Denote by $\rho = \lambda\mu$, the ratio of the (packet) arrival to server rate. It can be intuitively asserted that such a queue is stable only if $\rho < 1$, i.e. if the arrival rate is less than the service rate, as will be backed up by results. Let $\pi_k = P\{X = n\}$ denote the p.m.f for the steady-state queue size distribution X (i.e. if you send a large # of packets through the system, gather data on resulting queue size in steady-state and compute histogram), then the following well-known results can be found in any basic Queueing theory text:

i. $P\{X = n\} = \rho^n\,(1-\rho),\ n = 0, 1, 2.....$ Hence the fraction-of-time that the server is idle $\pi_0 = 1 - \rho$ and corr. busy equals ρ (hence the notion of 'utilization'). The resulting expected value of queue-size $E[X] = \frac{\rho}{1-\rho}$.

ii. The average latency (W) in the system experienced by any packet (this is the sum of the wait-time plus the service time) is given by $E[W] = \frac{1}{\mu-\lambda} =$. Clearly as $\lambda \to \mu$, $E[W] \to \infty$ implying the potential for unbounded queue sizes (and hence instability).

The above result can be deduced from a much broader law that is applicable to a very wide variety of queues (beyond just M/M/1) known as Little's Theorem, that relates to averages for the queuing systems:

Little's Theorem

Let N denote the *average* number of customers (packets) in the system (in our notation, $N = E[X] + 1$, i.e. the X packets in queue plus the one being served), and let T be the *average* time-in-system per packet ('time-in-system' equals the sum of wait time (W) and the service tiem), then by Little's Theorem $N = \lambda T$. Equivalently, if we just focus on the number of queued packets, then the foll. also holds: $E[X] = E[W]$. This has a very intuitive interpretation that can be verified by simulation: the (long-term) average queue size equals the product of the arrival rate λ and the mean wait time ($E[W]$) for a packet in queue.

A. ns-3 implementation

An ns-3 program implementing the M/M/1 queue as a stand-alone queue process (i.e. not integrated with a network device, just existing in isolation with a packet arrival and departure process) can be found in `contrib/simple-wireless/examples/mm1-queue.cc`.

```
$ ./waf --run 'mm1-queue --PrintHelp'
Program Options:
    --lambda:          arrival rate (packets/sec) [1]
    --mu:              departure rate (packets/sec) [2]
    --initialPackets:  initial packets in the queue [0]
    --numPackets:      number of packets to enqueue [0]
    --queueLimit:      size of queue (number of packets) [100]
    --quiet:           whether to suppress all output [false]
```

Note that lambda and mu can be set by command-line argument, and the number of packets (arrivals) to send through the system must be set to a non-zero value for the program to do anything useful.

Let's run it once by sending 10 packets:

```
$ ./waf --run 'mm1-queue --numPackets=10'
Program Options:
    --lambda:          arrival rate (packets/sec) [1]
    --mu:              departure rate (packets/sec) [2]
    --initialPackets:  initial packets in the queue [0]
    --numPackets:      number of packets to enqueue [0]
    --queueLimit:      size of queue (number of packets) [100]
    --quiet:           whether to suppress all output [false]
```

This produces a data file `mm1queue.dat`. This file has the following content:

```
0.352958 + 1
0.712375 + 2
```

```
0.722229 - 1
1.18865 + 2
1.29859 + 3
1.34304 - 2
1.72518 - 1
1.97162 + 2
2.64096 - 1
3.06024 - 0
3.85077 + 1
3.92556 - 0
5.84833 + 1
5.85479 - 0
7.42087 + 1
7.42253 - 0
9.34972 + 1
9.37861 - 0
11.3829 + 1
11.6735 - 0
```

Every '+' operation is an enqueue operation. Every '-' operation is a dequeue operation. The first column is the timestamp in seconds. The last column is the queue length at the end of the operation. The simulation ends when the last packet to send has been enqueued and dequeued (or dropped).

The time that the queue is busy can be deduced from this file: the total time of the simulation (11.6735 seconds) minus the sum of any time intervals for which a dequeue leads to a zero-length queue (e.g. from time 7.42253 to time 9.34972). Time zero until the first enqueue must also be considered. To answer the below questions, you will want to write some kind of script to parse this file, or else modify the C++ program to calculate idle times and delays.

Try rerunning the program with a different random number seed; do the results change much?

```
$ ./waf --run 'mm1-queue --numPackets=10 --RngRun=2'
```

Drops are traced as well. For instance, let's set the queue length to an (unreasonably) low value of 1 packet, and repeat.

```
$ ./waf --run 'mm1-queue --numPackets=10 --queueLimit=1'
```

```
0.352958 + 1
0.712375 d 1
0.722229 - 0
1.18865 + 1
```

```
1.29859 d 1
1.34304 - 0
1.97162 + 1
2.64096 - 0
3.85077 + 1
3.92556 - 0
5.84833 + 1
5.85479 - 0
7.42087 + 1
7.42253 - 0
9.34972 + 1
9.37861 - 0
11.3829 + 1
11.6735 - 0
```

B. Questions

3) For an M/M/1 queuing system, simulate with $\mu = 10$ and $\lambda =$ 1,3,5,7, and 9, and plot the queue idle proportion, mean queue length, and average packet delay, all as a function of the ratio λ/μ. For each trial, send 100,000 packets through the system. Set the queue limit to a large value to avoid drops.

4) Experiment with the buffer overflow outcome of a 100 packet M/M/1 queue by setting $\mu = 10$ and $\lambda =$ 9.5, 9.7, and 9.9. Starting from an empty queue, run a number of trials to measure how long it takes for the queue to overflow. Provide your estimate of the average time until overflow, sample standard deviation, and number of trials conducted for each $lambda$.

Two ways to accomplish the above are 1) to write a script to repeatedly run with the desired configuration but with different RngRun number values, and for each run, find the time of the first drop, or 2) run the program for a very long number of packets and use the entry into an idle queue as the start time to measure until the next drop.

V. Contention Based Protocols

In class, the concept of ALOHA (or Aloha) protocols were introduced. Aloha is a completely distributed protocol in which nodes send packets upon arrival to the system, and if two or more packets collide, the senders are able to detect them and reschedule them (after some amount of backoff time) for some time in the future. Aloha systems have interesting mathematical properties concerning stability of the system and backoff strategies.

In class, we saw that transmissions are vulnerable (or exposed) to possible collisions for a duration twice of their actual transmission time. If the nodes in the system are able to synchronize and transmit on slot boundaries, then the vulnerable period can be cut in half. This type of system is called slotted aloha and will be explored next.

A. *Slotted Aloha Basics*

Slotted ALOHA is a Random Access MAC protocol where multiple users all use a shared channel or set of channels to communicate with some kind of a base station. The channel is broken up into a set of frames in time. During an individual frame the n users sharing the channel select a time slot, or subframe, from among the available subframes to transmit on if they have data to transmit. This process is shown in figure 4.

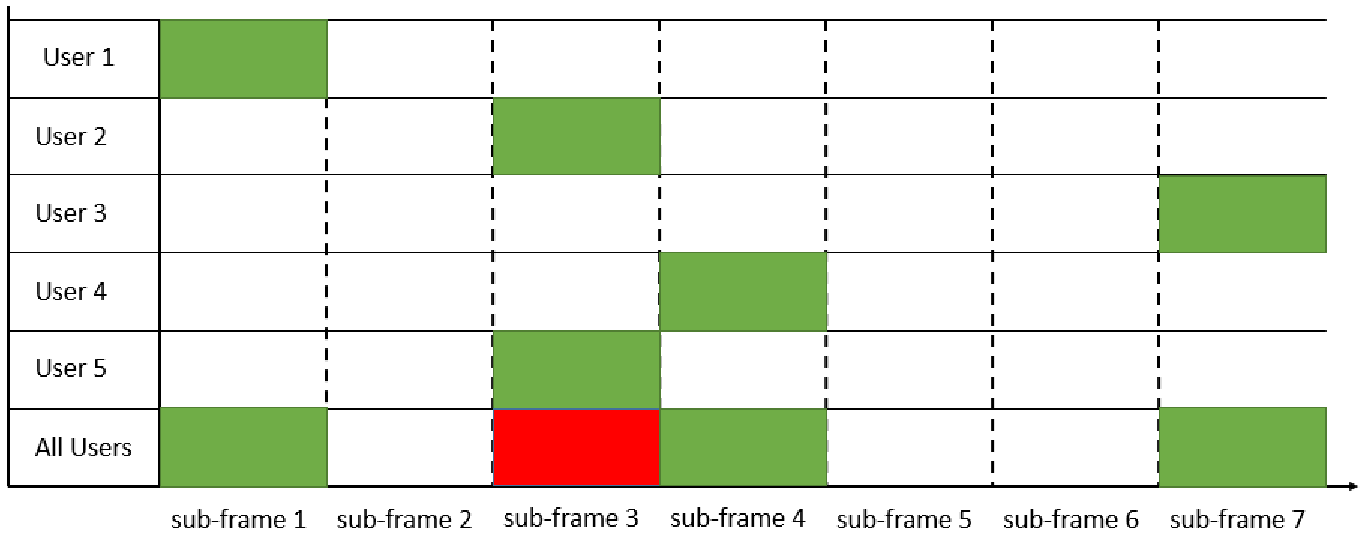

Fig. 4: Example of slotted ALOHA

In figure 4 user 1 chooses sub-frame 1, user 2 and 5 choose sub-frame 3, user 3 chooses sub-frame 7, and user 4 chooses sub-frame 4. The choice of sub-frame 3 by both user 2 and 5 will cause issues, and potentially result in a collision. We will begin by looking at MAC collisions. If we assume MAC collisions then two users choosing the same sub-frame will always result in a collision. In the case of MAC collisions we can analyze the throughput of the system. The lecture slides from January 24 contain some mathematical details.

1) MAC vs PHY Collisions: In the prior section we discussed MAC collisions as occurring when two or more packets choose the same subframe to transmit on, however consider the scenario shown in figure 5

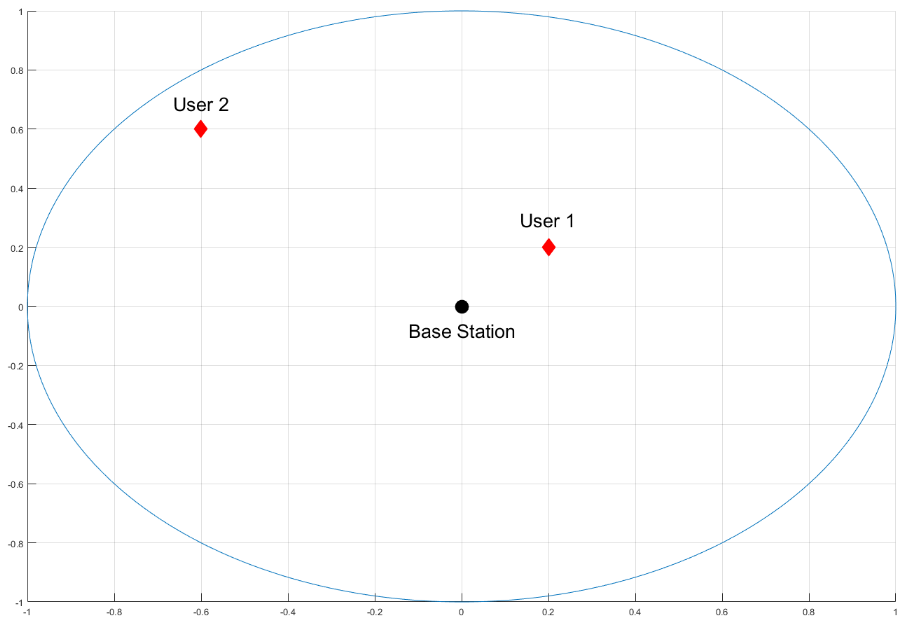

Fig. 5: Example Physical Scenario

It should be plainly seen that if both user 1 and user 2 select the same subframe then user 1's signal would overpower user 2's at the receiver. This leads to a new definition of collisions in wireless networks, PHY collisions. There are many ways to define PHY collisions but for this lab we will consider a PHY collision to be the event that no users have SIR greater than a threshold γ_0 where SIR is defined as:

$$SIR_i = \frac{P_i}{\sum_{j \neq i} P_j} \tag{3}$$

That is, the SIR for user i is its power at the receiver divided by the sum of the powers at the receiver of all other users who've chosen the same subframe. In the case of PHY collisions the throughput is not only a function of the number of users n and the number of subframes but also their distribution in space and definition of receive power. for these reasons a simple throughput calculation can not be performed in the case of PHY collisions, but with some intuition we can see that the PHY collision case is similar to the MAC collision case, except in some instances where in the PHY case receptions packets would be received where in the MAC case they would not, Meaning the the MAC collision throughput serves as a lower bound for the PHY collision throughput.

B. ns-3 model

A program located at `contrib/simple-wireless/examples/slotted-aloha.cc` uses the SimpleWireless NetDevice type to support a simple model of slotted aloha. The program defines a slot time of 100 microseconds, allowing the transmission of at most one packet per slot.

Packets arrive to the system according to a Poisson process with arrival rate lambda, in units of slots, defaulting to 0.1 (i.e. 0.1 packets arrive per slot). There are `numSenders` (default 100) in the network, and the simulation will continue until `maxPackets` are successfully received. This simulation is not concerned about physical layer effects but instead on the MAC that coordinates transmissions and retransmissions.

The program maintains a process that checks, every slot time, how many packet arrivals have been generated by the Poisson process, and sends (on the slot boundary) the number that have accumulated, by iterating on the available sending nodes. For some slots, there are no transmissions; for some, there is one transmission, which will be successfully received; and for some, there will be two or more transmissions (from different nodes). In the last case, all of such transmissions will collide and will need to be retransmitted. The sending process will detect this case and will schedule retransmissions according to a random backoff. The backoff process is simple; if there are numSenders in the system, the retransmission slot will be picked uniformly between 0 and numSenders slots in the future. Each retransmission will arrive to the system like a new transmission, and since the main arrival process continues unabated, the retransmissions have an effect of adding to the intensity of packet arrivals.

Note that the model does not concern itself with the fact that the retransmission will usually be sent from another node; for the throughput analysis, this detail does not matter (and it was simpler to just treat it similar to a new arrival). This is an example of not making the models more complicated than they need to be for the analysis at hand.

A sample program run with defaults is shown below:

```
$ ./waf --run 'slotted-aloha --PrintHelp'
Program Options:
    --lambda:      Arrival lambda (i.e. rate, or 1/mean); units of slots [0.1]
    --maxPackets:  the number of packets to send [1000]
    --numSenders:  number of sendingNodes [100]
$ ./waf --run 'slotted-aloha'
New packet arrival rate (packets/slot): 0.1
Simulation number of slot times: 10053
Total number of packets sent (inc. retransmissions): 1120
Total number of packets received: 1000
Throughput (number received/duration): 0.0994728
```

This simulation shows that when the packet arrival process generates 0.1 packets per slot, the simulation takes 10053 slot times to safely send the 1000 packets. 1120 packets were actually sent, which means that there were over 10% retransmissions due to collisions. The normalized throughput is approximately 0.1 (i.e. only about 10% of slots are used to successfully deliver a packet). We ought to be able to do better, but by how much?

Note: keep in mind that there are debugging log statements in the various .cc files. To look at the debug log for the program itself, use this type of command to indirect the log output into a file named 'log.out':

```
$ NS_LOG="SlottedAlohaExample" ./waf --run 'slotted-aloha' > log.out 2>&1
```

C. Question

5) Use the ns-3 slotted aloha program to explore the throughput performance as a function of lambda. For what value of lambda (to two decimal places) does the throughput maximize? What happens if you push past that value?