Carnegie Mellon
Computer Science Department.
15-744 Spring 2007 Problem Set 2

This problem set has 4 questions (each with several parts). Answer them as clearly and concisely as possible. You may discuss ideas with others in the class, but your solutions and writeup must be your own. If you do discuss at length with others, please mention in your solution for the problem who you collaborated with. Do not look at anyone else's solutions or copy them from anywhere.

This assignment is due by **5:00pm, Monday, March 5th** either in class or to the course secretary in Wean Hall 8018.

In the first two parts, you will use ns. In the third part, you will examine some BGP tables. Although the first question is long, it is not difficult; it is designed to help you learn the basics of ns.

A NS Tutorial

1. ns is a network simulator that will be useful in homeworks, the class project and in networks research in general.

   The purpose of this question is to give you a gentle introdution to using ns and to make you familiar with the ns manual (http://www.isi.edu/nsnam/ns/ns-documentation.html).

A.1 Setting Up

Here's what you need to do before you start this question:

1. Download and Install ns
   Download ns-allinone from the ns-2 homepage. This version of ns contains all the components you need, including the Tcl interpreter. We recommend using this "build-it-all-at-once" approach not only because it is less hassle-free than to build ns from its constituent pieces; but also because you might get different results or problems if you use a different version of ns or the Tcl interpreter.

   Visit http://www.isi.edu/nsnam/ns/ns-build.html#allinone for instructions to install ns. Make sure you have enough disk space on your machine: to build ns you will need around 250 Mb.

   A version of the ns-all-in-one package has been posted to the course webpage, with install scripts modified, that should easily compile on any facilitized cs machine running Linux. If you don't have access to such a machine, please email Jeff (jeffpang+744@cs.cmu.edu) and he will get you an account on one.
2. Starting ns
   After installation, you can start ns with the command 'ns <tclscript>' (assuming that you are in the directory with the ns executable, or that your path points to that directory), where '<tclscript>' is the name of a Tcl script file which defines the simulation scenario (i.e. the topology and the events). This will be more clear as you go along. You could also just start ns without any arguments (ns will return a % prompt on most systems) and enter the Tcl commands in the Tcl shell but that is definitely less comfortable.
3. Understanding Otcl
   This problem will be easier if you have a basic understanding of Otcl.
   http://www.isi.edu/nsnam/otcl/doc/tutorial.html gives a short introduction to Otcl for C++ programmers, which you might want to read if you are not familiar with Otcl.
4. Hand in answers to the questions in boxes. Hand in printouts of the tcl files you write for questions that request it.

A.2 A brief overview of ns

ns is an event-driven object-oriented simulator, written in C++, with an Otcl interpreter as a front-end. This means that most of your simulation scripts will be written in Tcl. If you want to develop new components for ns you might have to use both Tcl and C++.

ns uses two languages because any network simulator, in general, has two different kinds of things it needs to do. On the one hand, detailed simulations of protocols requires a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important. On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important.

ns meets both of these needs with two languages, C++ and Otcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. Otcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. ns (via tclcl) provides glue to make objects and variables **appear on both languages**.

In the first part of this assignment, you will learn how to write simulation scripts for ns. In the second part of the assignment, you will write new components for ns to extend the functionality of ns.

A.3 A simple example

Figure 1 shows a simple example of an ns script. Even without knowing ns you might be able to guess what the script is doing: it creates two nodes and connects them with a link. Then, 1 second after simulation begins it starts a constant-bit-rate source on one of the nodes. The source stops after 4.5 seconds and after 5 seconds the entire simulation is stopped.

In the next couple of sections we will examine the individual components of a typical ns script like the one in Figure 1.

A.4 How to start: A template for ns scripts

In this section you are going to write a template that you can use for all your simulation scripts.

A simulation in ns is described by a Tcl `class Simulator`. A simulation script, therefore, generally begins by creating an instance of this class. This is done with the command

```
set ns [new Simulator]
```

After creating the simulator object a simulation script usually calls various methods to create nodes and topologies. We will see in Section A.5 how exactly this is done. Next we have to define what exactly we want to happen in the simulation. Since the ns simulator is an event-driven simulator this is done by *scheduling events*: when creating a simulator object a scheduler is created which runs by selecting the next earliest event, executing it to completion, and returning to execute the next event. An event can for example be the sending of data from one node to another node or the failure of a link. The command to schedule an event is

```
$ns at <time in sec> <event>
```

After scheduling all the events being scheduled we are ready to start the simulator:

```
set ns [new Simulator]            # instantiate simulator obj

set nf [open out.nam w]           # file to write sim trace to
$ns namtrace-all $nf

proc finish {} {                  # called at end of sim
   global ns nf
   $ns flush-trace
   close $nf
   exec nam out.nam &                # visualize the sim trace file
   exit 0
}

set n0 [$ns node]                 # create two nodes and a link
set n1 [$ns node]
$ns duplex-link $n0 $n1 1Mb 10ms DropTail


set udp0 [new Agent/UDP]          # attach agents to nodes
$ns attach-agent $n0 $udp0
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0          # attach applications to agents

$ns connect $udp0 $null0          # connect the agents


$ns at 1.0 "$cbr0 start"          # scheduling and control
$ns at 4.5 "$cbr0 stop"

$ns at 5.0 "finish"               # call finish proc (above)

$ns run
```

Figure 1: A simple sample script

```
$ns run
```

Although we have seen the basics of running a simulation we don't have a way yet to get any output from the simulation. Fortunately, ns provides a number of ways of collecting output or trace data on a simulation as we will see in Section A.7.

The following code for example first opens the file 'out.nam' for writing and gives it the file handle 'nf'. It then tells the simulator to trace each packet on every link in the topology and write the data into the file.

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

The trace data is collected in a format that can later be used by the *nam* program to visualize the simulation in an animation.

The only remaining step is to make sure that all the trace data is flushed into the file at the end of the simulation and that the file is closed. Furthermore, we have to run the nam tool on the created trace file to produce the animation. We do this by writing a 'finish' procedure that we call at the end of the simulation and that performs all these tasks.

```
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec nam out.nam &
        exit 0
}
```

The following line calls this procedure at the end of the simulation:

```
$ns at 5.0 ‘‘finish’’
```

A.5 Creating a network topology

Next let's look at how to create topologies in ns.

A.5.1 Nodes and Links

To create a node we can simply use the simulator method `node`. The following two lines create two nodes and assign them to the handles 'n0' and 'n1'.

```
set n0 [$ns node]
set n1 [$ns node]
```

We can then either use the simulator method `simplex-link` or the method `duplex-link` to connect the nodes with a link:

```
$ns simplex-link $n0 $n1 1Mb 10ms DropTail
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
```

The first line creates a unidirectional link between n0 and n1 with bandwidth 1Mbps, a propagation delay of 10ms and a DropTail queue. The second line creates a bidirectional link with the same parameters.

In addition to the DropTail queue, ns also supports many other queueing policies like for example FairQueueing(FQ), Random Early Detection (RED), and Class-based queueing (CBQ).

(a) What is the default maximum buffer size of the queue in a link? Which command can you use to limit the maximum buffer size of the queue in the link between nodes n1 and n2 to 5 packets? (Hint: the variable to look for is called limit_)

A.5.2 Creating larger topologies

When creating larger topologies that have a regular structure it is often more comfortable to use a Tcl loop instead of creating the nodes one by one. The following code creates seven nodes and stores them in the array *n()*.

```
for {set i 0} {$i < 7} {incr i} {
    set n($i) [$ns node]
}
```

Note that arrays like other Tcl variables don't have to be declared first.

The following questions asks you to write a procedure for creating dumb-bell topologies like in Figure 2. The dumb-bell topology is used to study the effect of a bottle-neck link shared by many sources. You will need it later in this homework and also in the next assignment with ns.

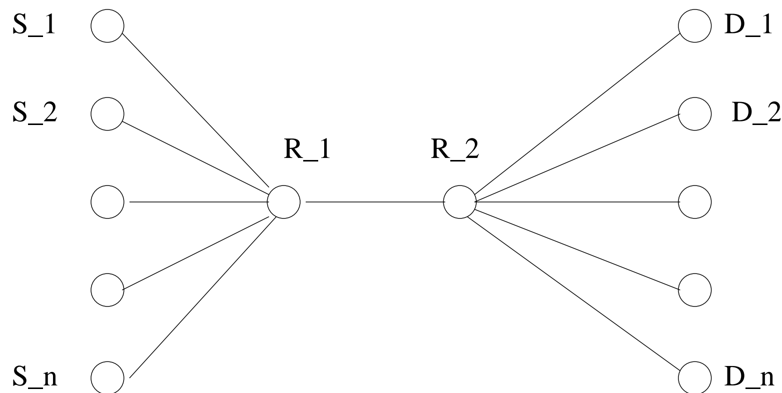

Figure 2: *The dumb-bell topology*

> **(b)** Write a Tcl procedure that takes as an argument an integer n and creates a dumb-bell topology with n nodes on either side. Make the link between R1 and R2 a 5Mbps/10ms duplex links with a DropTail queue. Make all the other links 100Mbps/10ms duplex links with DropTail queues.

Instead of creating your own topologies by hand you can also use one of the *topology generators* that *ns-allinone* comes with. One example is *itm*, which is based on Georgia-Tech's Internetwork Topology Models (GT-ITM).

A.6 Creating network traffic

By now you have some scripts that create topologies but there is not yet traffic. Traffic generation in ns is based on objects of two classes, the `Agent` and the `Application`.

Agents represent endpoints where network-layer packets are constructed or consumed. Every node in the network that needs to send or receive traffic must have an agent attached to it. These agents can be thought of as the implementation of the transport protocol.

On top of an agent runs an application. The application determines the kind of traffic source that is simulated (e.g. ftp or telnet). Applications represent the application layer in an ns-simulation.

A.6.1 Creating Agents

Corresponding to the two most popular transport protocols used in the Internet there are also two types of agents in ns: UDP agents and TCP agents.

The following code shows an example of attaching a UDP agent to nodes n0 and n1:

```
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

ns connect udp0 null0
```

This code first creates a UDP agent and attaches it to n0 using the `attach-agent` procedure. It then creates a Null agent which will act as a traffic sink and attaches it to n1. Finally, the two agents are connected using the simulator method `connect`. In the next section the UDP agent will be used by an application to send data.

(c) Read the ns manual to find out how to write similar code to set up a *TCP* connection. Then use your procedure from Question 3 to create a dumb-bell topology with 5 S_i-D_i pairs. Set up a TCP connection for each of the S_i-D_i pairs. For all connections use an object of type `Agent/TCP`.

(d) The code you wrote in the previous question creates a TCP agent that implements TCP Tahoe. List the other flavors of TCP that are supported in ns.

A.6.2 Creating Applications

In the previous section we have set up the agents implementing the transport layer. We will now create applications that we attach to the transport agents and that will actually generate traffic. In ns there are two basic types of applications: *simulated applications* and *traffic generators*.

Traffic generators generate On/Off traffic: during On-periods, packets are generated at a constant burst rate and during Off-periods no packets are generated. ns provides three different classes of traffic generators which differ in how the lengths of the On and Off-periods are modeled:

1. A traffic generator of the type `Application/Traffic/Exponential` takes the length of the On and Off periods from an Exponential distribution.
2. A `Application/Traffic/Pareto` source generates the lengths of these periods from a Pareto distribution.
3. Finally, the class `Application/Traffic/CBR` has no off periods and generates packets at a constant bit rate.

The following code generates one traffic generator of each class.

```
set exp [new Application/Traffic/Exponential]
set par [new Application/Traffic/Pareto]
```

```
set cbr0 [new Application/Traffic/CBR]
```

See the ns-manual for how to configure these traffic generators.

All traffic generators run on top of a UDP agent. Therefore, we have to attach a traffic generator to a UDP agent before we can use it to send data. The following example illustrates the use of the CBR traffic generator that we created above.

```
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

$ns at 1.0 ''$cbr start''
```

The *simulated applications* currently implemented in ns are `Application/FTP` and `Application/Telnet`. These try to simulate the corresponding applications in the real world: FTP and Telnet. Like the real applications the ns applications can run only on TCP. They therefore have to be attached to a TCP agent.

(e) Create and attach an FTP application to each of the S_i nodes from Question 3 and connect each to the corresponding D_i nodes. Start the FTP sources at 4 sec and stop them at 100 sec. Now change the code so that instead of explicitly stopping the FTP sources after 100 sec they stop after sending 10 packets. Save the resulting scripts in files called ftp_before.tcl and ftp_after.tcl, respectively.

A.7 Tracing and Monitoring

So far, when we wanted to see what is happening in a simulation we have used nam to visualize the events. However, often we want more detailed *trace data* on a simulation which then allows us to compute numbers like packet loss rates or per-connection throughput.

The easiest way to do this is to use the simulator method `trace-all` which traces all the links in the topology:

```
set f[open out.tr w]
$ns trace-all $f
```

When you use `trace-all` you also have to remember to change the `finish` procedure to flush the trace and close the file (as we did previously for the nam tracing).

Sometimes we are only interested in the events at one particular queue and want to avoid the overhead that comes with tracing every single link in the topology. For this case ns provides us with the `trace-queue` method. In addition, Section 22 of the ns-manual describes many other ways of collecting trace data.

(f) Create a trace file for the simulation in the previous question (ftp_before.tcl). Start the tracing 20 sec after simulation begins and report the loss rate on each of the flows, and the number of packets sent, received and dropped on each flow.

Instead of tracing *events* it is often useful to trace C++ or Tcl *instance variables*. Below is an example of setting up variable tracing in ns.

```
# $tcp tracing its own variable cwnd_
$tcp trace cwnd_

# the variable ssthresh_ of $tcp is traced by a generic $tracer
set tracer [new Trace/Var]
$tcp trace ssthresh_ $tracer
```

B NS TCP Simulation

The goal of the problems in this section is to use ns to examine the interaction between different versions of TCP and lossy links.

2. Begin by examining the dynamics of a simple TCP connection between two endpoints on a wired network. First, download this simple ns script and examine it:

   `http://www.cs.cmu.edu/~dga/15-744/S07/ps2/ns-tcp-wired.tcl`

   (a) Draw a very simple diagram of the network that the script simulates, including the one-way latency and bandwidth of the links.

   (b) What kind of queueing discipline does the simulation's router use?

   (c) Run the script and plot the evolution of the sender's TCP window over time. Identify where TCP slow-start ends and where congestion avoidance begins.

   (d) What is the average throughput of the transfer?

3. Next, grab the fake-wireless ns script. It implements the same topology, but the link to the receiver now goes over a link that experiences random losses (it's not really wireless; the wireless extensions to ns are a bit hairy, so we're faking it):

   `http://www.cs.cmu.edu/~dga/15-744/S07/ps2/ns-tcp-fake-wireless.tcl`

   (a) Plot, as you did above, the TCP window evolution.

   (b) What is the throughput of this connection?

   (c) This script originally used TCP/Reno. Newer variants of TCP handle random losses slightly better. Modify the script to use TCP/NewReno and repeat the plot and throughput estimation.

   (d) Briefly explain why NewReno did better.

C BGP Tables

4. Route servers (e.g. those available at `http://www.traceroute.org/#Route%20Servers`) are BGP speaking routers with a publicly accessible interface. In other words, you can telnet to these routers and access their full BGP tables.

   `route-views.oregon-ix.net` is one such route server hosted at the University of Oregon. One use of this route server is that you can potentially get the route(s) from any AS X to any AS Y at an AS path level. You can also get the routes that *almost* any AS X would take to reach a given address prefix P.

   For this exercise, download the following routing table entries from the RouteViews server at:

   `http://www.cs.cmu.edu/~dga/15-744/S07/ps2/oix-full-snapshot-2007-02-14-1800.dat.bz2`

   Warning: this is 13 MB! You can use the `bzcat` command-line program to read it without decompressing it all. Although you only need a small part of this table to answer the following questions, you should to learn how to navigate large datasets like this efficiently. (Hint: write some code)

RouteViews has archived their BGP tables since 1997. You can examine more of them at:

`http://archive.routeviews.org`

(a) CMU owns the address block `128.2.0.0/16`. Using this information, can you figure out the ISP CMU uses (the AS number of the ISP)? Using the `whois` service at `http://www.arin.net/whois/` or the `whois` command-line program, determine who this AS number actually corresponds to (the name of the ISP). Note: some address blocks allocated pre-CIDR appear without the netmask in the table; i.e., CMU's address block appears as `128.2.0.0`. Three, two, and one trailing 0 octets imply a class A (/8), class B (/16), or class C (/24) network, respectively.

(b) Print the best AS route from the route server to CMU.

(c) What is the AS number of MIT? List all providers of `mit.edu` that you can infer from the table. (Hint: MIT is one of the few class A networks. You can use `nslookup` to get the IP address for a host at MIT)

(d) Some of the routes to MIT repeat its AS number multiple times. Why would they do that? What does this tell you about the upstream provider in those paths?

(e) Find the first "Class C" CIDR address in the table (address prefix $\geq$ 192.0.0.0). How many class C networks does this address correspond to? What is the maximum number of routing table entries that this single CIDR address saves? Why is it that we can only infer the maximum, and not the actual, number of addresses that this CIDR address saves?

You can get more information if you log into this route server by executing:

`telnet route-views.oregon-ix.net`

Run `sh ip bgp` at the prompt and you get the entire BGP table, shown one screen after another (much like when you execute `more`). In general you can type `sh ip BGP ?` for help on the possible extensions to the `sh ip bgp` command. For example, you can use the help to figure out that `sh ip bgp 12.0.0.0` will give you all the routes from `oregon-ix` to `12.0.0.0/8`.