Zartash Uzmi

Programming Assignment 3

Due Saturday, Nov 19, 2016 at 11:59pm

Late penalty: 30% for up to 24hr; no credit afterwards

In this assignment, you will be working with ns2, a popular tool for simulating a variety of networks. The network simulator ns2 is often used to test out new ideas, such as measuring the performance of a new protocol that you may have come up with. In this assignment, you will learn ns2 and test out some of the known things about networks.

About cooperation with other students:

This assignment is meant to be done alone. Absolutely no cooperation is allowed. If there are questions, ask the course staff; they are there to help you. For this assignment, you must do all thinking, research, and coding by yourself. Do not even discuss with anyone how far you are with this assignment

ANY HELP (EXCEPT FROM COURSE STAFF) IS PROHIBITED!!!

Individuals found guilty of violating above policy will be referred to the disciplinary committee.

(Warning: We will use software to measure the code similarity)

More Important Note:

Once again, we are giving you the responsibility to REPORT all incidences of cooperation. You MUST report to the instructor (or the TAs) such incidences. If you do not, we will consider you as guilty as those who you witnessed cooperating. **All coding, debugging, web search for functions, etc. must be done by individual students!**

Preamble:

You will use **Linux programming environment** for this assignment. Use your existing venus account for this purpose. As a reminder, you MUST NOT share your password with anyone including the TAs and the instructor.

You will need to submit all the required files zipped in a zip file whose name is derived from your student ID. See submission instructions (at the end) for more details.

Where to begin?

It would be helpful to go through an ns2 tutorial before doing this assignment. There is a bunch of tutorials out there. We have added a few to the LMS resources section. We are also providing you a few quick examples in this handout to get you started. Go over the examples in this handout as well as those provided in the tutorials on LMS.

Additional tutorials: Tracing and monitoring queues in ns2 is given at:

http://www.mathcs.emory.edu/~cheung/Courses/558-old/Syllabus/90-NS/trace.html

Yet another one focusing on TCP performance is available at:

http://sps.utm.my/wp-content/uploads/2013/12/NS2-Manual9.pdf

Basics:
The simulator we are using (ns2) is a discrete event simulator, which means that events such as packet arrival and departure, failure of a network router, etc. which may happen at various times can all be studied through simulation using ns2. Thus, we do not need to have an actual network to send actual traffic on.

As you might imagine, ns2 is a software program. This has already been installed for you on venus.lums.edu.pk (the Venus server) where you already have an account. Create a directory within your account to complete work related to this assignment.

You can invoke ns2 from venus command line by just entering ns at the command prompt. You will get a new prompt (of the simulator) where you can enter and execute ns2-specific commands. More often, however, you create a TCL script enlisting all the commands you would want ns2 to execute one after the other and then invoke ns2 asking it to run the specific TCL script file. For example, if your script file is named myfirstexample.tcl, then at the venus command prompt you would simply type:

```
Venus prompt >> ns myfirstexample.tcl
```

In this assignment, you will be using ns2-specific commands within the script to create a network, to send traffic (e.g., TCP flows) on that network and to measure the statistics of the traffic and/or the network (flow throughput, queue size within a router, etc.).

It is advised to practice the following examples, however simple they may seem.

Example 1: Hello World

This example will familiarize us with a few ns2 commands and the basic TCL syntax. We will create a TCL script (in a file with .tcl extension) and write our own Hello World program. Our script will contain variables and objects just like any other script you may have seen elsewhere. In order to create a variable we use the 'set' command in the TCL script (also works on the ns2 command prompt). Here is an example:

```
set x 5
set y 10
set z 0.45
set name "Alice"
set rollNumber 20099999
```

We can see that the set command is followed by the name of the variable which is in turn followed by its value. To further use these variables in our script we must put a '$' sign before the variables. This will become clear in a bit.

Let’s perform some basic operations on the variables we just created. For instance we might want to add the variables 'x' and 'y' then multiply the result with 'z'. We can do this by using the 'expr' command within the square brackets. Here is how we may do it:

```
set temp [expr $x + $y]
set result [expr $temp * $z]
```

The first line creates a variable 'temp' and sets its value to the sum of 'x' and 'y' (Notice the use of the dollar sign). The next line creates a variable 'result' and sets its value to the product of 'temp' and 'z'. Be careful to place the dollar sign before the variable name.

Now that we have a bunch of variables, we would like to verify whether our program is running as we expect it to. In other words, we would like to test or debug our code. A simple way to do this is to print the variables and see whether the values are as we expect them to be. To print variables to the terminal, we use the 'puts' command. Suppose the variable 'result' is the age of a girl whose name is Alice. Here is how we will print this information on the terminal:

```
puts "The name of the girl is $name and her age is $result"
```

Now that you are familiar with the basic syntax, it is advisable to write a simple script (in a .tcl file) that prints your name and the result of your roll number divided by your age on the terminal. Your output should look something like this: “My name is Alice and this strange number is 2977777.6296296297”

After you are done writing your script, you can run it by typing “ns myhelloworld.tcl” on the terminal (the Venus command prompt). Just test it; nothing to submit about it.

Example 2: Hello Networks

A network is just an interconnection of nodes via links. We will begin by making a very simple 2-node network, also known as the 'Dumbbell topology'. While such a network might seem trivial, it is an important one for testing out new protocols and designs.

The TCL scripts will contain different objects such as nodes, links, TCP agents etc. The most important of these is the simulator object and we need to create it in the script before we do anything else. Without creating this object, we can only make Hello World type programs. In order to create a simulator object we use the set command again. Here is how to do it:

```
set mySimulator [new Simulator]
```

This line creates a new Simulator object (a built-in object in ns2) and gives it a name mySimulator (the variable). We will use this variable further in our script when we have to create network objects. Now that we have created a Simulator object, we can create nodes and link them up. Here is how nodes can be created:

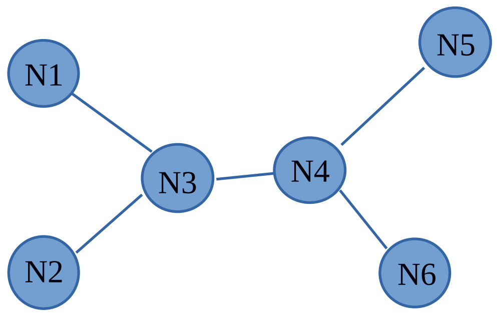
```
set node1 [$mySimulator node]
set node2 [$mySimulator node]
```

We have now created two nodes and given them names 'node1' and 'node2'. Let’s now connect them.

```
$mySimulator duplex-link $node1 $node2 2Mb 10ms DropTail
```

The line above creates a two-way (duplex) link of bandwidth 2Mbps, 10ms propagation delay between our two nodes. DropTail means that the input and output queues of this link will be simple FIFO queues (other queues also exist – RED queues, Priority queues etc, but you do not need to worry about those at this point).

Now that you know how to create a 2-node network, try creating another, slightly larger yet simple, network shown in the figure below. Set all link capacities to 10Gbps with a propagation delay of 1ms, except for link N3→N4. For this link (N3→N4), take the values for link capacity and propagation delay as input from the user (see below).



Make sure your TCL script runs without any errors. Whether the script is doing the right thing or not is a separate thing, and will be verified in subsequent exercises.

How to take the user input?

User input may be provided as additional command line arguments when executing a TCL script with ns. For example, you may execute the following:

Venus prompt >> `ns mysecondexample.tcl 74`

To store user input (74, in this case) in a variable (say, for example, firstInput) we use the following command within the TCL script:

```
set firstInput [lindex $argv 0]
```

If, for example, you enter two values as additional command line arguments and store those values in variables input1 and input 2, then you may create a link as follows:

```
$mySimulator duplex-link $node1 $node2 $input1 $input2 DropTail
```

This will create a link between (previously created nodes) node1 and node2 where link speed and delay are specified by the user at the command line. Try out self-created examples for handling user input (and printing those from within the script).

Example 3: Hello TCP

After completing the above examples, we only have a bunch of nodes connected with each other via links. A real-world network will have a number of nodes connected via links and there will be communication (network traffic) between nodes. In this example, we will establish a TCP connection between two nodes. In ns2, a TCP connection has two components: a TCP agent which is built on the sender node and a TCP sink which is built on the receiver node. The following lines of code show how to create a TCP connection:

```
set myTcp [new Agent/TCP]
$mySimulator attach-agent $nodeS $myTcp
$myTcp set packetSize_ 1460

set myTcpSink [new Agent/TCPSink]
$mySimulator attach-agent $nodeD $myTcpSink

$mySimulator connect $myTcp $myTcpSink
```

The first three lines in the above piece of code create a TCP agent object and attach it to the sending node. Next two lines create a TCP sink object and attach it to the destination node. The last line simply connects the TCP agent (sender) with TCP sink (receiver). Similarly, if we want to create another TCP connection between another pair of nodes, we will simply create new TCP objects and attach them to their respective nodes.

A nice way to think about this is that our nodes are essentially machines physically connected to each other while TCP agent and TCP sink, linked to processes running on these machines, are logically connected to each other.

Two TCP connections:
You are now ready to try out code for creating two TCP connections: one between N1 and N5 and another between N2 and N6 (in the above topology/network).

Note: Since TCP is end-to-end, you might note that we just create a TCP connection between **end nodes** that may or may not be directly connected to each other. The simulator automatically takes care of finding the right path through the network between the source and destination node by running some routing protocol in the background.

There are several parameters that we can change: for example, we have set the packet size for each TCP connection to be 1460 bytes. The default value for the packet size is 1000 bytes. Note that although we have set the packet size, we still have not sent any traffic yet.

Example 4: Hello Simulation

In this example we will learn about a few necessary commands to simulate sending the traffic on our network. Note that we have merely created nodes and established TCP connections between them up till now. We will familiarize ourselves with commands required to schedule traffic start, traffic stop and simulation end. In ns2 whenever we have to simulate an event, we tell our simulator object ($mySimulator) to schedule an event at a given time. Let’s first see how we can schedule the traffic flow.

We can send traffic from one node to another after we have established a TCP connection between them (previous example). In this example, we will attach an application to the existing TCP agent. Specifically, we will use FTP (FTP is file transfer protocol, studied in class) to send its traffic through TCP. Another type of application traffic is constant bit rate (CBR[2]) traffic which is completely characterized by specifying its rate.

Let's now see how we can set up a FTP source and attach it to a TCP connection.

```
set myFTP [new Application/FTP]
$myFTP attach-agent $myTcp
$myFTP set type_ FTP
```

The only thing left now is to schedule the start and stop of this traffic. We can do this statically or in a loop. For the purpose of this example, a static scheduling is sufficient. Here is how to do it:

```
$mySimulator at 0.1 "$myFTP start"
$mySimulator at 0.9 "$myFTP stop"
```

We have simply told our simulator object (mySimulator) that at 0.1 seconds it should start the traffic from myFTP and at 0.9 seconds it should stop the traffic.

Now that we have scheduled the traffic flow, we would also require the simulation to stop at some point. As you may have seen in most tutorials/examples, a procedure (analogue of the c++ function) titled 'finish' is invoked. The finish procedure is nothing but a function that closes all trace[3] files (if any) and closes the simulator gracefully. We need to schedule when the procedure 'finish' should be called. We can do this in a manner similar to the one used for scheduling the traffic flow:

```
$mySimulator at 1.0 “finish”
```

In the reference TCL script given, we have implemented this procedure. Go through it to get a better idea. Note that 'finish' is not a key word. We could have very well named our procedure differently and scheduled it when we wanted the simulation to end.

The final command of the TCL script needs to tell the simulator to run all the events that were scheduled at their respective times. We need to add the following line at the end of our script:

```
$mySimulator run
```

To summarize, in ns2 we must schedule events, such as the start or the stop of a traffic flow or the calling of a procedure and most importantly scheduling a 'finish' procedure. We can schedule all events independently of each other but they must make chronological sense: traffic from an application should not be scheduled to stop before it is scheduled to start.

[2] Commands to set up CBR traffic can be found in the reference code (distributed with this assignment).
[3] A trace file contains simulation data (queue statistics, packet drop information, etc.)

Try building on the previous examples and augment the TCP connections between (i) N1→N5 and (ii) N2→N6 with FTP objects.

Notice again that this sets very nicely with the layering structure that we studied in class. We first created physical connections between nodes, then attached TCP objects on nodes and finally we attached applications to those TCP objects.

Note: Simulation time and actual time is different. 0.1 simulation seconds are not really 0.1 seconds in real time.

The Assignment (Task-1)

In this task, you have to:

1. Create a 3 node topology (Src→Router→Dst)
2. Assign 2Mbps BW and 10ms delay to link Src→Router
3. Assign 1Mbps BW and 10ms delay to link Router→Dst
4. Set queue size (see reference code) to be equal to the last three digits of your roll number mod 9 plus 4. For example if the last three digits are 187 then the queue size will be 11 (187%9 + 4)
5. Create a TCP connection between Src and Dst – packet size: 1000 bytes
6. Setup a TCP trace object to trace cwnd of our tcp sender (see reference code)
7. Setup a queue monitor at the link Router→ Dst (see reference code) to track queue statistics (sample time = 0.04s)
8. We will also reset this queue monitor (procedure given in reference code) every 0.04 sec (Do not worry about why we do this)
9. Create a FTP agent at Src – start time: 0.01 stop time: 10.0 sec
10. Stop simulation at 11.0 sec
11. Ug the trace files (see format in the tutorial) produced, generate a graph (on the software of your choice) of the congestion window versus time of the TCP sender
12. Plot on the same axis the number of packet drops at the queue (see format of queue monitor)

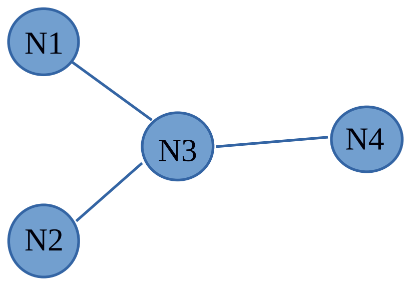
For this task you have to submit:

1. Code (.tcl file)
2. TCP trace file
3. Queue monitor trace file
4. Graph specified above (either .pdf or .png)

<u>Why are we doing this?</u>
In class we studied TCP and claimed that it slows down when a packet drops. We will see in this simulation that the congestion window looks like a saw tooth where the sudden decrease in the congestion window coincides with a packet drop in the queue.

Task-2: TCP fairness

Begin this task by making the following topology:



In this task you have to:

1. Assign 2Mbps BW and 10ms delay to link N1→N3
2. Assign 2Mbps BW and 10ms delay to link N2→N3
3. Assign 1Mbps BW and 10ms delay to link N3→N4
4. Set queue size to be equal to the last three digits of your roll number mod 11 plus 11 (187%11 + 11)
5. Create two TCP connections one between N1→N4 another one between N2→N4
6. Create and attach a FTP source for each of the TCP connections
7. Setup a TCP trace object to trace cwnd of each of the TCP sender
8. Setup a queue monitor for the link N3→N4 (sample rate = 0.04s)
9. Start the FTP at N1 at 0.01 sec
10. Start the FTP at N2 at 0.1 sec
11. Stop both FTP sources at 10.0 sec
12. End simulation at 11.0 sec
13. Generate a graph (on the software of your choice) of the congestion window versus time of each TCP sender (plot on the same axis is required)

For this task you have to submit:

1. Code (.tcl file)
2. Both TCP trace files
3. Queue monitor trace file
4. Graph specified above (either .pdf or .png)

<u>Why are we doing this?</u>
Recall that TCP ensures fairness and the bandwidth is divided equally amongst all TCP sessions. In this example we will observe that the congestion windows of both TCP connections converge, indicating fair sharing.

Task-3: Parallel TCP connections

We will now make a slight modification to our code for task 2. Instead of having only one TCP connection from N1$\rightarrow$N4, we will make two separate TCP connections from N1$\rightarrow$N4.

In addition to your code for task 2, you are required to:

1. Create another TCP connection from N1$\rightarrow$N4
2. Create a TCP trace file for this connection
3. Create a FTP source and attach it to this TCP connection
4. Change the queue size to the last three digits of your roll number mod 10 plus 35 (187%10 + 35)
5. Start all FTP sources at 0.01 sec
6. Stop all FTP sources at 10.0 sec
7. Stop Simulation at 11.0 sec
8. Generate a graph (on the software of your choice) of the congestion window (aggregate) versus time of the two TCP connections from N1$\rightarrow$N4 and of the TCP connection from N2$\rightarrow$N4 (plot on the same axis is required)

For this task you have to submit:

1. Code
2. All three TCP trace files
3. Queue monitor trace file
4. Graph specified above (either .pdf or .png)

<u>Why are we doing this?</u>
By opening parallel TCP connections a user can attain a higher rate. We will observe that the aggregate congestion window of the TCP connection from N1$\rightarrow$N4 will be higher than the congestion window of the TCP connection from N2$\rightarrow$N4. This implies that N1 attains a higher rate than N2, indicating unfairness.

Task-4: UDP versus TCP

Recall that TCP was needed for congestion control by modulating the rate of traffic. In this task we will see that the network will start to misbehave if not all senders are using TCP. So in this task we will modify our code for task 2 and replace one of the TCP connections with a UDP agent.

In this task you are required to modify the code for task 2 in the following way:

1. Set queue size to be equal to the last three digits of your roll number mod 18 plus 10 (187%18 + 10)
2. Replace the TCP connection from N2→N4 with a UDP connection (see reference code)
3. Replace the FTP source at N2 with a CBR traffic source and attach it to the newly created UDP sender (rate = 2Mbps, packet size = 1000 bytes)
4. Since we have removed a TCP connection, we must also remove the trace file associated with it
5. Start the FTP source at 0.01 sec
6. Start the CBR traffic at 5.0 sec and stop it at 15.0 sec
7. Stop the FTP source at 20.0 sec
8. End simulation at 22.0 sec
9. Generate a graph (on the software of your choice) of the congestion window of the TCP sender

For this task you have to submit:

1. Code
2. TCP trace file
3. Queue monitor trace file
4. Graph specified above (either .pdf or .png)

<u>Why are we doing this?</u>
Since UDP does not offer congestion control, it will keep sending traffic at a rate that might be unacceptable for the network causing smart protocols like TCP to back down. We will observe that when traffic from the UDP agent flows, TCP will see this as a sign of extreme congestion and reduce its rate significantly i.e. UDP will choke TCP. Many DDOS attacks are launched by UDP flooding causing links to fill up and queues to saturate so even if a TCP sender sends a single packet it drops, effectively reducing the rate of a TCP sender to zero.

What is the BIGGEST piece of advice?
Start early. This assignment will require you to debug a lot of client/server code and will take considerable amount of time. Keep the entire weekend(s) for the assignment and add another several days here and there! The only other advice is to get help from the course staff and/or the gdb debugger. Learn to debug programs using *cout*/*printf* as well.

What and Where to submit?

1. Very importantly, **strictly** stick to the following submission guidelines.
2. Create a folder for each task. All required files for that task should go in its respective folder (task-1, task-2, etc.).
3. Graph files can only be in PDF format of PNG format.
4. Zip all these files together in one file named <your_student_num>.zip where <your_student_num> is an **8-digit** number. Example, if your roll no is 14100055, your submission file will be 14100055.zip. Be very careful with this, our script **might** throw away zip files that do not follow this convention. No contest will be allowed in that case.
5. **Do NOT email your assignment to us.**
6. Do just one submission of your zip file through LMS system (use the ***Assignment link*** in your portfolio, and **NOT** the Dropbox).