Marc Greis' Tutorial for the UCB/LBNL/VINT Network Simulator

Table of Contents

II. Finding Documentation

[Previous section] [Next section] [Back to the index]

In this section I am going to list some sources for documentation for ns and related packages. If you know any other sources, or if any of the links have expired, please send email to ns-users mailing list.

II.1. Documentation for ns&nam

The following documentation for ns and nam is available from the main ns web page at UCB.

- "ns Notes and Documentation (now renamed the ns Manual)" could be called the "main manual" for ns and is available in Postscript format.
- An HTML version (currently without diagrams) of nsN&D (now renamed ns Manual) is available here.
- A manual page for ns is included in the distribution in the ns directory. There is a HTML-ized version here, but it might be out-dated.
- There is a ps version of the nam user-manual which is available from the nam page. You can also get an ASCII version from here.
- You can also get slides from the second ns workshop from this page. They don't really contain more information than the "ns Notes and Documentation" (now renamed ns Manual) document, though it might be a bit easier to understand and use.
- If you can't get ns to compile, if it crashes, or if you have any other similar problems, take a look at the ns-problems page before you ask on the mailing list.
- If you have any general questions about ns or nam, you can send them to the ns-users mailing list. If you're not sure if your question has been asked before, check the Archive for the mailing list.

II.2. Documentation for Tcl

Tcl is fairly simple and if you already have some programming skills, you should be able to learn most of what you need for simple scenarios as you go along. However, I will try to provide some interesting links for the more ambitious users who are not willing (or able) to buy a Tcl book.

- The Tcl8.0/Tk8.0 Manual is basically a collection of hypertext manual pages.
- A draft for a Tcl/Tk book is available in postscript format for personal use. Only the first 94 pages are relevant for Tcl, the rest of the book is about Tk and more complicated aspects of Tcl.
- I also found a short OTcl Tutorial.
- Another good starting point for looking for Tcl documentation is the Yahoo Tcl/Tk category.

II.3. Documentation for C++

I'd like to note that you don't need **any** C++ programming knowledge unless you want to add new functionality to ns.

- Perhaps the best source for learning C++ is the book "The C++ Programming Language" by Bjarne Stroustrup.
- If you don't want to buy a book, you can take a look at this C++ tutorial.
- You can find an ANSI C++ standard draft on this page
- And again, if you want to look for information on the web yourself, check out Yahoo's C/C++ category.

[Previous section] [Next section] [Back to the index]

ns-users

ns-users@isi.edu

III. The Basics

III.1. Downloading/Installing ns&nam

You can build ns either from the the various packages (Tcl/Tk, otcl, etc.), or you can download an 'all-in-one' package. I would recommend that you start with the all-in-one package, especially if you're not entirely sure which packages are installed on your system, and where exactly they are installed. The disadvantage of the all-in-one distribution is the size, since it contains some components that you don't need anymore after you compiled ns and nam. It's still good for first tests, and you can always switch to the single-package distribution later.

Note: The all-in-one package only works on Unix systems.

You can download the package from the ns download page at UCB. If you have any problems with your installation, take a look at the installation problems page on their server. If that also doesn't solve your problem, you might want to ask the ns-users mailing list.

After the installation is complete, you should make sure that your path points to the 'ns-allinone/bin' directory (if you installed the ns-allinone package) where links to the ns and nam executables in the 'ns-2' and 'nam-1' directories can be found or (if you built ns and nam from the pieces) let your path point directly to the directories with the ns and nam executables.

On some systems you will also have to make sure that ns can find the library 'libotcl.so'. If you installed the ns-allinone package, it should be in 'ns-allinone/otcl/'. On Solaris systems you would have to add this path to the 'LD_LIBRARY_PATH' environment variable. For help with other systems, consult the installation problem page, the ns-users mailing list or your local Unix gurus.

A note concerning the ns-allinone version 2.1b3: There is a bug in it which causes some problems on Solaris systems when nam trace generation is turned on. You can either download ns-allinone version 2.1b2 instead or go to the ns web page to download a current snapshot of ns. If you do that, you have to unzip and untar the file in your allinone directory. Then you change into the new directory and run './configure', then 'make'.

III.2. Starting ns

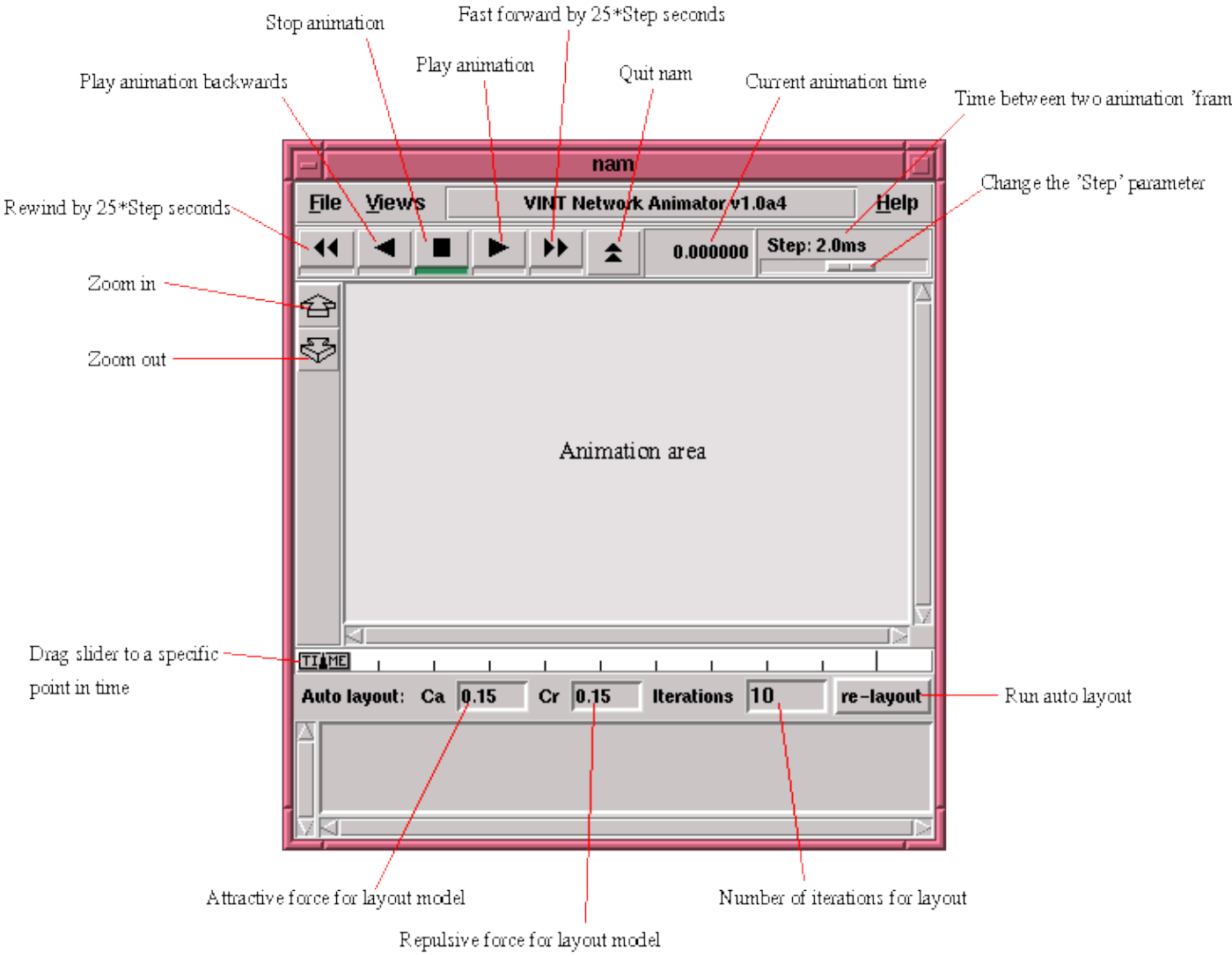
You start ns with the command 'ns <tclscript>' (assuming that you are in the directory with the ns executable, or that your path points to that directory), where '<tclscript>' is the name of a Tcl script file which defines the simulation scenario (i.e. the topology and the events). You could also just start ns without any arguments and enter the Tcl commands in the Tcl shell, but that is definitely less comfortable. For information on how to write your own Tcl scripts for ns, see section IV.

Everything else depends on the Tcl script. The script might create some output on stdout, it might write a trace file or it might start nam to visualize the simulation. Or all of the above. These possibilities will all be discussed in later sections.

III.3. Starting nam

You can either start nam with the command 'nam <nam-file>' where '<nam-file>' is the name of a nam trace

file that was generated by ns, or you can execute it directly out of the Tcl simulation script for the simulation which you want to visualize. The latter possibility will be described in Section IV. For additional parameters to nam, see the nam manual page. Below you can see a screenshot of a nam window where the most important functions are being explained.



[Previous section] [Next section] [Back to the index]

ns-users
ns-users@isi.edu

IV. The first Tcl script

[Previous section] [Next section] [Back to the index]

In this section, you are going to develop a Tcl script for ns which simulates a simple topology. You are going to learn how to set up nodes and links, how to send data from one node to another, how to monitor a queue and how to start nam from your simulation script to visualize your simulation.

IV.1. How to start

Now we are going to write a 'template' that you can use for all of the first Tcl scripts. You can write your Tcl scripts in any text editor like joe or emacs. I suggest that you call this first example 'example1.tcl'.

First of all, you need to create a simulator object. This is done with the command

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

Now we open a file for writing that is going to be used for the nam trace data.

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

The first line opens the file 'out.nam' for writing and gives it the file handle 'nf'. In the second line we tell the simulator object that we created above to write all simulation data that is going to be relevant for nam into this file.

The next step is to add a 'finish' procedure that closes the trace file and starts nam.

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

You don't really have to understand all of the above code yet. It will get clearer to you once you see what the code does.

The next line tells the simulator object to execute the 'finish' procedure after 5.0 seconds of simulation time.

```
$ns at 5.0 "finish"
```

You probably understand what this line does just by looking at it. ns provides you with a very simple way to schedule events with the 'at' command.

The last line finally starts the simulation.

```
$ns run
```

You can actually save the file now and try to run it with 'ns example1.tcl'. You are going to get an error message like 'nam: empty trace file out.nam' though, because until now we haven't defined any objects (nodes, links, etc.) or events. We are going to define the objects in section 2 and the events in section 3.

You will have to use the code from this section as starting point in the other sections. You can download it here.

IV.2. Two nodes, one link

In this section we are going to define a very simple topology with two nodes that are connected by a link. The following two lines define the two nodes. (Note: You have to insert the code in this section **before** the line '$ns run', or even better, before the line '$ns at 5.0 "finish"').

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

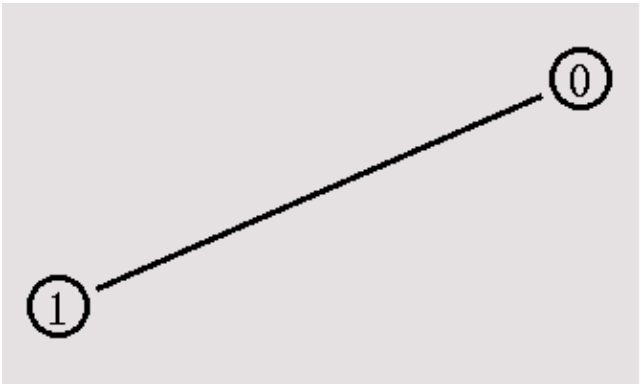
A new node object is created with the command '$ns node'. The above code creates two nodes and assigns them to the handles 'n0' and 'n1'.

The next line connects the two nodes.

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

This line tells the simulator object to connect the nodes n0 and n1 with a duplex link with the bandwidth 1Megabit, a delay of 10ms and a DropTail queue.

Now you can save your file and start the script with 'ns example1.tcl'. nam will be started automatically and you should see an output that resembles the picture below.



You can download the complete example here if it doesn't work for you and you think you might have made a mistake.

IV.3 Sending data

Of course, this example isn't very satisfying yet, since you can only look at the topology, but nothing actually happens, so the next step is to send some data from node n0 to node n1. In ns, data is always being sent from one 'agent' to another. So the next step is to create an agent object that sends data from node n0, and another agent object that receives the data on node n1.

```
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

These lines create a UDP agent and attach it to the node n0, then attach a CBR traffic generatot to the UDP agent. CBR stands for 'constant bit rate'. Line 7 and 8 should be self-explaining. The packetSize is being set to 500 bytes and a packet will be sent every 0.005 seconds (i.e. 200 packets per second). You can find the relevant parameters for each agent type in the ns manual page

The next lines create a Null agent which acts as traffic sink and attach it to node n1.

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

Now the two agents have to be connected with each other.

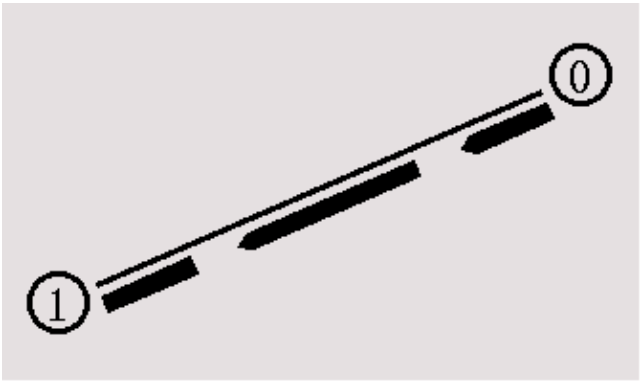
```
$ns connect $udp0 $null0
```

And now we have to tell the CBR agent when to send data and when to stop sending. Note: It's probably best to put the following lines just before the line '$ns at 5.0 "finish"'.

```
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

This code should be self-explaining again.

Now you can save the file and start the simulation again. When you click on the 'play' button in the nam window, you will see that after 0.5 simulation seconds, node 0 starts sending data packets to node 1. You might want to slow nam down then with the 'Step' slider.



I suggest that now you start some experiments with nam and the Tcl script. You can click on any packet in the nam window to monitor it, and you can also click directly on the link to get some graphs with statistics. I also suggest that you try to change the 'packetsize_' and 'interval_' parameters in the Tcl script to see what happens. You can download the full example here.

Most of the information that I needed to be able to write this Tcl script was taken directly from the example files in the 'tcl/ex/' directory, while I learned which CBR agent arguments (packetSize_, interval_) I had to set from the ns manual page.

[Previous section] [Next section] [Back to the index]

ns-users
ns-users@isi.edu

V. Making it more interesting

[Previous section] [Next section] [Back to the index]

In this section we are going to define a topology with four nodes in which one node acts as router that forwards the data that two nodes are sending to the fourth node. I will explain find a way to distinguish the data flows from the two nodes from each other, and I will show how a queue can be monitored to see how full it is, and how many packets are being discarded.

V.1. The topology

As always, the first step is to define the topology. You should create a file 'example2.tcl', using the code from section IV.1 as a template. As I said before, this code will always be similar. You will always have to create a simulator object, you will always have to start the simulation with the same command, and if you want to run nam automatically, you will always have to open a trace file, initialize it, and define a procedure which closes it and starts nam.

Now insert the following lines into the code to create four nodes.

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```

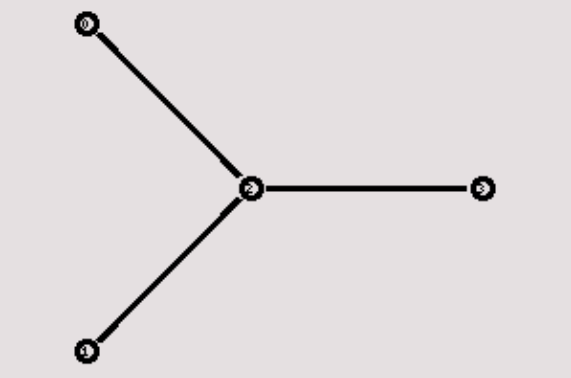
The following piece of Tcl code creates three duplex links between the nodes.

```
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n3 $n2 1Mb 10ms DropTail
```

You can save and start the script now. You might notice that the topology looks a bit awkward in nam. You can hit the 're-layout' button to make it look better, but it would be nice to have some more control over the layout. Add the next three lines to your Tcl script and start it again.

```
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right
```

You will probably understand what this code does when you look at the topology in the nam window now. It should look like the picture below.



Note that the autolayout related parts of nam are gone, since now you have taken the layout into your own hands. The options for the orientation of a link are right, left, up, down and combinations of these orientations. You can experiment with these settings later, but for now please leave the topology the way it is.

V.2. The events

Now we create two UDP agents with CBR traffic sources and attach them to the nodes n0 and n1. Then we create a Null agent and attach it to node n3.

```
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

#Create a UDP agent and attach it to node n1
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1

# Create a CBR traffic source and attach it to udp1
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 attach-agent $udp1

set null0 [new Agent/Null]
$ns attach-agent $n3 $null0
```

The two CBR agents have to be connected to the Null agent.

```
$ns connect $udp0 $null0
$ns connect $udp1 $null0
```

We want the first CBR agent to start sending at 0.5 seconds and to stop at 4.5 seconds while the second CBR agent starts at 1.0 seconds and stops at 4.0 seconds.

```
$ns at 0.5 "$cbr0 start"
```

```
$ns at 1.0 "$cbr1 start"
$ns at 4.0 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"
```

When you start the script now with 'ns example2.tcl', you will notice that there is more traffic on the links from n0 to n2 and n1 to n2 than the link from n2 to n3 can carry. A simple calculation confirms this: We are sending 200 packets per second on each of the first two links and the packet size is 500 bytes. This results in a bandwidth of 0.8 megabits per second for the links from n0 to n2 and from n1 to n2. That's a total bandwidth of 1.6Mb/s, but the link between n2 and n3 only has a capacity of 1Mb/s, so obviously some packets are being discarded. But which ones? Both flows are black, so the only way to find out what is happening to the packets is to monitor them in nam by clicking on them. In the next two sections I'm going to show you how to distinguish between different flows and how to see what is actually going on in the queue at the link from n2 to n3.

V.3. Marking flows

Add the following two lines to your CBR agent definitions.

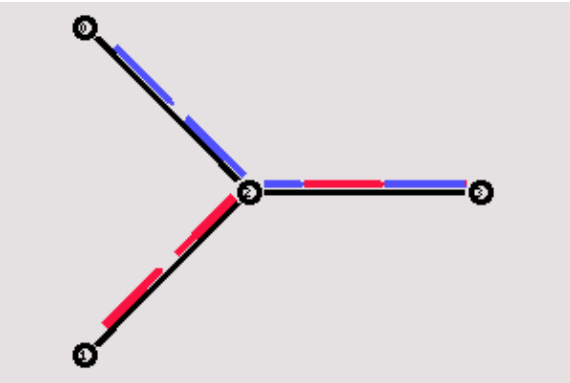
```
$udp0 set class_ 1
$udp1 set class_ 2
```

The parameter 'fid_' stands for 'flow id'.

Now add the following piece of code to your Tcl script, preferably at the beginning after the simulator object has been created, since this is a part of the simulator setup.

```
$ns color 1 Blue
$ns color 2 Red
```

This code allows you to set different colors for each flow id.



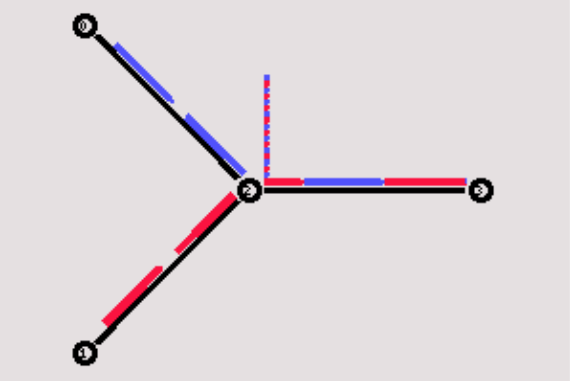
Now you can start the script again and one flow should be blue, while the other one is red. Watch the link from node n2 to n3 for a while, and you will notice that after some time the distribution between blue and red packets isn't too fair anymore (at least that's the way it is on my system). In the next section I'll show you how you can look inside this link's queue to find out what is going on there.

V.4. Monitoring a queue

You only have to add the following line to your code to monitor the queue for the link from n2 to n3.

```
$ns duplex-link-op $n2 $n3 queuePos 0.5
```

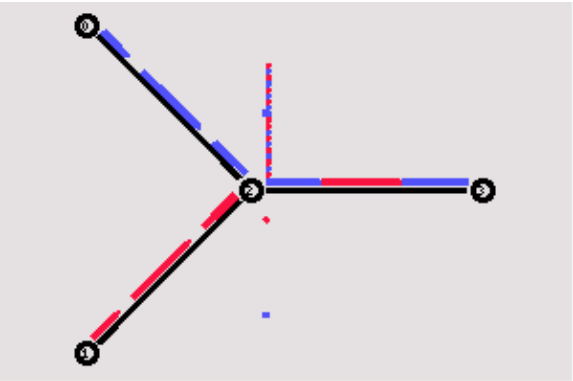
Start ns again and you will see a picture similar to the one below after a few moments.



You can see the packets in the queue now, and after a while you can even see how the packets are being dropped, though (at least on my system, I guess it might be different in later or earlier releases) only blue packets are being dropped. But you can't really expect too much 'fairness' from a simple DropTail queue. So let's try to improve the queueing by using a SFQ (stochastic fair queueing) queue for the link from n2 to n3. Change the link definition for the link between n2 and n3 to the following line.

```
$ns duplex-link $n3 $n2 1Mb 10ms SFQ
```

The queueing should be 'fair' now. The same amount of blue and red packets should be dropped.



You can download the full example here.

[Previous section] [Next section] [Back to the index]

ns-users
ns-users@isi.edu

VI. Network dynamics

[Previous section] [Next section] [Back to the index]

In this section I am going to show you an example for a dynamic network where the routing adjusts to a link failure. On the way there I'll show you how you can keep a larger number of nodes in a Tcl array instead of giving each node its own name.

VI.1. Creating a larger topology

I suggest you call the Tcl script for this example 'example3.tcl'. You can already insert the template from section IV.1 into the file.

As always, the topology has to be created first, though this time we take a different approach which you will find more comfortable when you want to create larger topologies. 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]
}
```

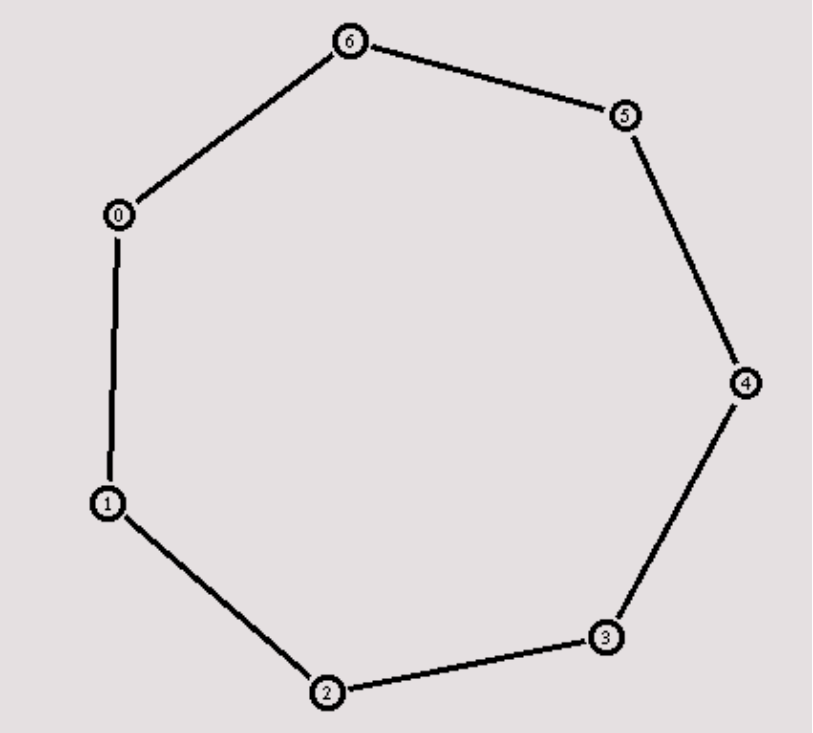
You have certainly seen 'for' loops in other programming languages before, and I am sure you understand the structure at once. Note that arrays, just like other variables in Tcl, don't have to be declared first.

Now we're going to connect the nodes to create a circular topology. The following piece of code might look a bit more complicated at first.

```
for {set i 0} {$i <7} {incr i} {
  $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}
```

This 'for' loop connects all nodes with the next node in the array with the exception of the last node, which is being connected with the first node. To accomplish that, I used the '%' (modulo) operator.

When you run the script now, the topology might look a bit strange in nam at first, but after you hit the 're-layout' button it should look like the picture below.



VI.2. Link failure

The next step is to send some data from node n(0) to node n(3).

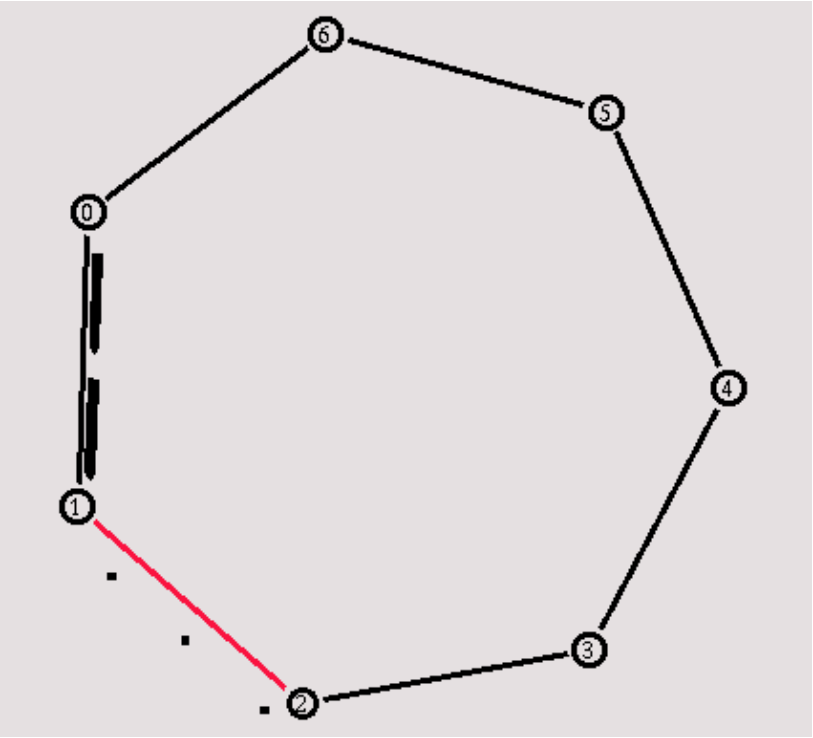
```
#Create a UDP agent and attach it to node n(0)
set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0

# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

set null0 [new Agent/Null]
$ns attach-agent $n(3) $null0

$ns connect $udp0 $null0

$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

The code above should look familiar to you by now. The only difference to the last sections is that now we have to use the node array elements.

If you start the script, you will see that the traffic takes the shortest path from node 0 to node 3 through nodes 1 and 2, as could be expected. Now we add another interesting feature. We let the link between node 1 and 2 (which is being used by the traffic) go down for a second.

```
$ns rtmodel-at 1.0 down $n(1) $n(2)
$ns rtmodel-at 2.0 up $n(1) $n(2)
```

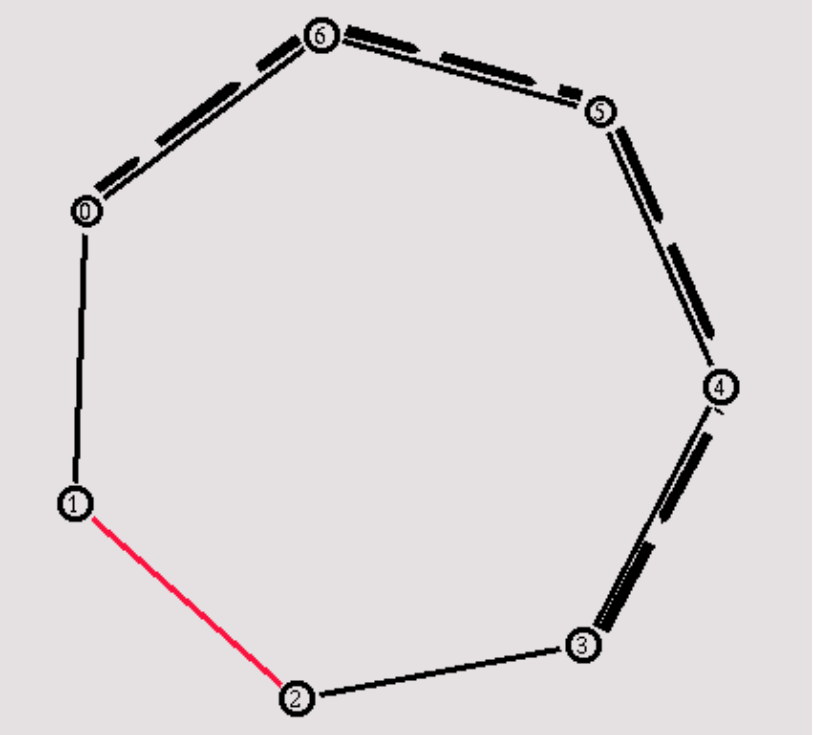
It is probably not too hard to understand these two lines. Now you can start the script again and you will see that between the seconds 1.0 and 2.0 the link will be down, and all data that is sent from node 0 is lost.



Now I will show you how to use dynamic routing to solve that 'problem'. Add the following line at the beginning of your Tcl script, after the simulator object has been created.

```
$ns rtproto DV
```

Start the simulation again, and you will see how at first a lot of small packets run through the network. If you slow nam down enough to click on one of them, you will see that they are 'rtProtoDV' packets which are being used to exchange routing information between the nodes. When the link goes down again at 1.0 seconds, the routing will be updated and the traffic will be re-routed through the nodes 6, 5 and 4.



You can download the full example here.

[Previous section] [Next section] [Back to the index]

ns-users
ns-users@isi.edu

VII. A new protocol for ns

[Previous section] [Back to the index] [Next section]

In this section I will give you an example for a new protocol that could be implemented in ns. You should probably become fairly familiar with ns before you try this yourself, and some C++ knowledge is definitely necessary. You should also read at least the chapters 3.1-3.3 from "ns Notes and Documentation" (now renamed ns Manual) to understand the interaction between Tcl and C++.

The code in this section implements some sort of simple 'ping' protocol (inspired by the 'ping requestor' in chapter 9.6 of the "ns Notes and Documentation" (now renamed ns Manual), but fairly different). One node will be able to send a packet to another node which will return it immediately, so that the round-trip-time can be calculated.

I understand that the code presented here might not be the best possible implementation, and I am sure it can be improved, though I hope it is easy to understand, which is the main priority here. However, suggestions can be sent here.

VII.1. The header file

In the new header file 'ping.h' we first have to declare the data structure for the new Ping packet header which is going to carry the relevant data.

```
struct hdr_ping {
  char ret;
  double send_time;
};
```

The char 'ret' is going to be set to '0' if the packet is on its way from the sender to the node which is being pinged, while it is going to be set to '1' on its way back. The double 'send_time' is a time stamp that is set on the packet when it is sent, and which is later used to calculate the round-trip-time.

The following piece of code declares the class 'PingAgent' as a subclass of the class 'Agent'.

```
class PingAgent : public Agent {
 public:
  PingAgent();
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);
 protected:
  int off_ping_;
};
```

In the following section, I am going to present the C++ code which defines the constructor 'PingAgent()' and the functions 'command()' and 'recv()' which were redefined in this declaration. The int 'off_ping_' will be used to access a packet's ping header. Note that for variables with a local object scope usually a trailing '_' is used.

You can download the full header file here (I suggest you do that and take a quick look at it, since the code that was presented here isn't totally complete).

VII.2. The C++ code

First the linkage between the C++ code and Tcl code has to be defined. It is not necessary that you fully understand this code, but it would help you to read the chapters 3.1-3.3 in the "ns Manual" if you haven't done that yet to understand it.

```
static class PingHeaderClass : public PacketHeaderClass {
public:
  PingHeaderClass() : PacketHeaderClass("PacketHeader/Ping",
                                        sizeof(hdr_ping)) {}
} class_pinghdr;


static class PingClass : public TclClass {
public:
  PingClass() : TclClass("Agent/Ping") {}
  TclObject* create(int, const char*const*) {
    return (new PingAgent());
  }
} class_ping;
```

The next piece of code is the constructor for the class 'PingAgent'. It binds the variables which have to be accessed both in Tcl and C++.

```
PingAgent::PingAgent() : Agent(PT_PING)
{
  bind("packetSize_", &size_);
  bind("off_ping_", &off_ping_);
}
```

The function 'command()' is called when a Tcl command for the class 'PingAgent' is executed. In our case that would be '$pa send' (assuming 'pa' is an instance of the Agent/Ping class), because we want to send ping packets from the Agent to another ping agent. You basically have to parse the command in the 'command()' function, and if no match is found, you have to pass the command with its arguments to the 'command()' function of the base class (in this case 'Agent::command()'). The code might look very long because it's commented heavily.

```
int PingAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {
    if (strcmp(argv[1], "send") == 0) {
      // Create a new packet
      Packet* pkt = allocpkt();
      // Access the Ping header for the new packet:
      hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);
      // Set the 'ret' field to 0, so the receiving node knows
      // that it has to generate an echo packet
      hdr->ret = 0;
      // Store the current time in the 'send_time' field
      hdr->send_time = Scheduler::instance().clock();
      // Send the packet
      send(pkt, 0);
      // return TCL_OK, so the calling function knows that the
      // command has been processed
      return (TCL_OK);
    }
  }
```

```
  // If the command hasn't been processed by PingAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}
```

The function 'recv()' defines the actions to be taken when a packet is received. If the 'ret' field is 0, a packet with the same value for the 'send_time' field, but with the 'ret' field set to 1 has to be returned. If 'ret' is 1, a Tcl function (which has to be defined by the user in Tcl) is called and processed the event (Important note to users of the ns version 2.1b2: 'Address::instance().NodeShift_[1]' has to be replaced with 'NODESHIFT' to get the example to work under ns 2.1b2).

```
void PingAgent::recv(Packet* pkt, Handler*)
{
  // Access the IP header for the received packet:
  hdr_ip* hdrip = (hdr_ip*)pkt->access(off_ip_);
  // Access the Ping header for the received packet:
  hdr_ping* hdr = (hdr_ping*)pkt->access(off_ping_);
  // Is the 'ret' field = 0 (i.e. the receiving node is being pinged)?
  if (hdr->ret == 0) {
    // Send an 'echo'. First save the old packet's send_time
    double stime = hdr->send_time;
    // Discard the packet
    Packet::free(pkt);
    // Create a new packet
    Packet* pktret = allocpkt();
    // Access the Ping header for the new packet:
    hdr_ping* hdrret = (hdr_ping*)pktret->access(off_ping_);
    // Set the 'ret' field to 1, so the receiver won't send another echo
    hdrret->ret = 1;
    // Set the send_time field to the correct value
    hdrret->send_time = stime;
    // Send the packet
    send(pktret, 0);
  } else {
    // A packet was received. Use tcl.eval to call the Tcl
    // interpreter with the ping results.
    // Note: In the Tcl code, a procedure 'Agent/Ping recv {from rtt}'
    // has to be defined which allows the user to react to the ping
    // result.
    char out[100];
    // Prepare the output to the Tcl interpreter. Calculate the round
    // trip time
    sprintf(out, "%s recv %d %3.1f", name(),
            hdrip->src_.addr_ >> Address::instance().NodeShift_[1],
            (Scheduler::instance().clock()-hdr->send_time) * 1000);
    Tcl& tcl = Tcl::instance();
    tcl.eval(out);
    // Discard the packet
    Packet::free(pkt);
  }
}
```

You can download the full file here. The most interesting part should be the 'tcl.eval()' function where a Tcl function 'recv' is called, with the id of the pinged node and the round-trip-time (in miliseconds) as parameters. It will be shown in Section VII.4 how the code for this function has to be written. But first of all, some other files have to be edited before ns can be recompiled.

VII.3. Necessary changes

You will have to change some things in some of the ns source files if you want to add a new agent, especially if it uses a new packet format. I suggest you always mark your changes with comments, use #ifdef, etc., so you can easily remove your changes or port them to new ns releases.

We're going to need a new packet type for the ping agent, so the first step is to edit the file 'packet.h'. There you can find the definitions for the packet protocol IDs (i.e. PT_TCP, PT_TELNET, etc.). Add a new definition for PT_PING there. In my edited version of packet.h, the last few lines of enum packet_t {} looks like the following code (it might look a bit different in earlier/later releases).

```
enum packet_t {
        PT_TCP,
        PT_UDP,
        ......
        // insert new packet types here
        PT_TFRC,
        PT_TFRC_ACK,
        PT_PING,    //  packet protocol ID for our ping-agent
        PT_NTYPE // This MUST be the LAST one
};
```

You also have to edit the p_info() in the same file to include "Ping".

```
class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
                name_[PT_UDP]= "udp";
                ...........
                name_[PT_TFRC]= "tcpFriend";
                name_[PT_TFRC_ACK]= "tcpFriendCtl";

                name_[PT_PING]="Ping";

                name_[PT_NTYPE]= "undefined";
        }
        .....
 }
```

Remember that you have to do a 'make depend' before you do the 'make', otherwise these two files might not be recompiled.

The file 'tcl/lib/ns-default.tcl' has to be edited too. This is the file where all default values for the Tcl objects are defined. Insert the following line to set the default packet size for Agent/Ping.

```
Agent/Ping set packetSize_ 64
```

You also have to add an entry for the new ping packets in the file 'tcl/lib/ns-packet.tcl' in the list at the beginning of the file. It would look like the following piece of code.

```
        { SRMEXT off_srm_ext_}
        { Ping off_ping_ }} {
```

```
set cl PacketHeader/[lindex $pair 0]
```

The last change is a change that has to be applied to the 'Makefile'. You have to add the file 'ping.o' to the list of object files for ns. In my version the last lines of the edited list look like this:

```
sessionhelper.o delaymodel.o srm-ssm.o \
srm-topo.o \
ping.o \
$(LIB_DIR)int.Vec.o $(LIB_DIR)int.RVec.o \
$(LIB_DIR)dmalloc_support.o \
```

You should be able to recompile ns now simply by typing 'make' in the ns directory. If you are having any problems, please email ns-users.

VII.4. The Tcl code

I'm not going to present the full code for a Tcl example for the Ping agent now. You can download a full example here. But I will show you how to write the 'recv' procedure that is called from the 'recv()' function in the C++ code when a ping 'echo' packet is received.

```
Agent/Ping instproc recv {from rtt} {
        $self instvar node_
        puts "node [$node_ id] received ping answer from \
              $from with round-trip-time $rtt ms."
}
```

This code should be fairly easy to understand. The only new thing is that it accesses the member variable 'node_' of the base class 'Agent' to get the node id for the node the agent is attached to.

Now you can try some experiments of your own. A very simple experiment would be to **not** set the 'ret' field in the packets to 1. You can probably guess what is going to happen. You can also try to add some code that allows the user to send ping packets with '$pa send $node' (where 'pa' is a ping agent and 'node' a node) without having to connect the agent 'pa' with the ping agent on 'node' first, though that might be a little bit more complicated than it sounds at first. You can also read the chapter 9.6 from the "ns Manual" to learn more about creating your own agents. **Good luck**.

[Previous section] [Back to the index] [Next section]

Marc Greisgreis@cs.uni-bonn.de

VIII. Creating Output Files for Xgraph

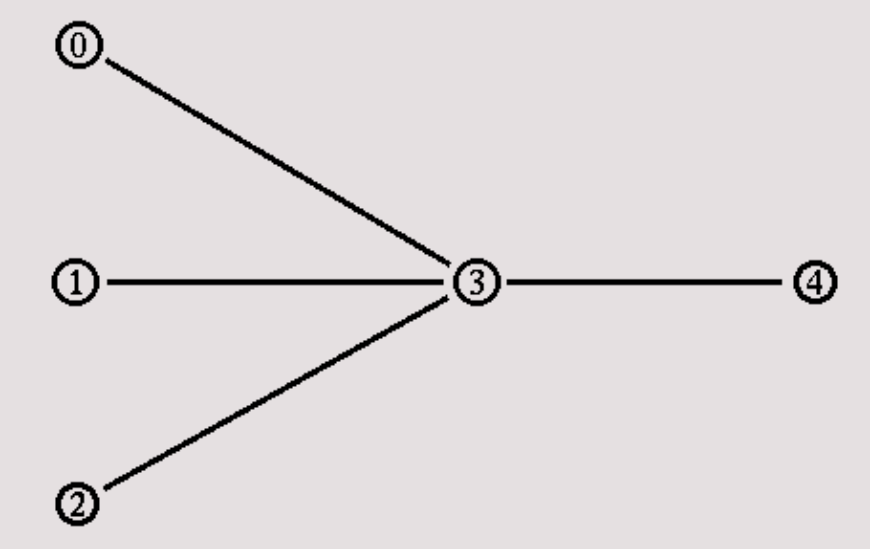
[Previous section] [Next Section] [Back to the index]

One part of the ns-allinone package is 'xgraph', a plotting program which can be used to create graphic representations of simulation results. In this section, I will show you a simple way how you can create output files in your Tcl scripts which can be used as data sets for xgraph. On the way there, I will also show you how to use traffic generators.

A note: The technique I present here is one of many possible ways to create output files suitable for xgraph. If you think there is a technique which is superior in terms of understandablity (which is what I aim for in this tutorial), please let me know.

VIII.1. Topology and Traffic Sources

First of all, we create the following topology:



The following piece of code should look familiar to you by now if you read the first sections of this tutorial.

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]

$ns duplex-link $n0 $n3 1Mb 100ms DropTail
$ns duplex-link $n1 $n3 1Mb 100ms DropTail
$ns duplex-link $n2 $n3 1Mb 100ms DropTail
$ns duplex-link $n3 $n4 1Mb 100ms DropTail
```

We are going to attach traffic sources to the nodes n0, n1 and n2, but first we write a procedure that will make it easier for us to add the traffic sources and generators to the nodes:

```
proc attach-expoo-traffic { node sink size burst idle rate } {
        #Get an instance of the simulator
        set ns [Simulator instance]
```

```
        #Create a UDP agent and attach it to the node
        set source [new Agent/UDP]
        $ns attach-agent $node $source

        #Create an Expoo traffic agent and set its configuration parameters
        set traffic [new Application/Traffic/Exponential]
        $traffic set packetSize_ $size
        $traffic set burst_time_ $burst
        $traffic set idle_time_ $idle
        $traffic set rate_ $rate

        # Attach traffic source to the traffic generator
        $traffic attach-agent $source
        #Connect the source and the sink
        $ns connect $source $sink
        return $traffic
}
```

This procedure looks more complicated than it really is. It takes six arguments: A node, a previously created traffic sink, the packet size for the traffic source, the burst and idle times (for the exponential distribution) and the peak rate. For details about the Expoo traffic sources, please refer to the documentation for ns.

First, the procedure creates a traffic source and attaches it to the node, then it creates a Traffic/Expoo object, sets its configuration parameters and attaches it to the traffic source, before eventually the source and the sink are connected. Finally, the procedure returns a handle for the traffic source. This procedure is a good example how reoccuring tasks like attaching a traffic source to several nodes can be handled. Now we use the procedure to attach traffic sources with different peak rates to n0, n1 and n2 and to connect them to three traffic sinks on n4 which have to be created first:

```
set sink0 [new Agent/LossMonitor]
set sink1 [new Agent/LossMonitor]
set sink2 [new Agent/LossMonitor]
$ns attach-agent $n4 $sink0
$ns attach-agent $n4 $sink1
$ns attach-agent $n4 $sink2

set source0 [attach-expoo-traffic $n0 $sink0 200 2s 1s 100k]
set source1 [attach-expoo-traffic $n1 $sink1 200 2s 1s 200k]
set source2 [attach-expoo-traffic $n2 $sink2 200 2s 1s 300k]
```

In this example we use Agent/LossMonitor objects as traffic sinks, since they store the amount of bytes received, which can be used to calculate the bandwidth.

VIII.2. Recording Data in Output Files

Now we have to open three output files. The following lines have to appear 'early' in the Tcl script.

```
set f0 [open out0.tr w]
set f1 [open out1.tr w]
set f2 [open out2.tr w]
```

These files have to be closed at some point. We use a modified 'finish' procedure to do that.

```
proc finish {} {
        global f0 f1 f2
        #Close the output files
        close $f0
        close $f1
```

```
        close $f2
        #Call xgraph to display the results
        exec xgraph out0.tr out1.tr out2.tr -geometry 800x400 &
        exit 0
}
```

It not only closes the output files, but also calls xgraph to display the results. You may want to adapt the window size (800x400) to your screen size.

Now we can write the procedure which actually writes the data to the output files.

```
proc record {} {
        global sink0 sink1 sink2 f0 f1 f2
        #Get an instance of the simulator
        set ns [Simulator instance]
        #Set the time after which the procedure should be called again
        set time 0.5
        #How many bytes have been received by the traffic sinks?
        set bw0 [$sink0 set bytes_]
        set bw1 [$sink1 set bytes_]
        set bw2 [$sink2 set bytes_]
        #Get the current time
        set now [$ns now]
        #Calculate the bandwidth (in MBit/s) and write it to the files
        puts $f0 "$now [expr $bw0/$time*8/1000000]"
        puts $f1 "$now [expr $bw1/$time*8/1000000]"
        puts $f2 "$now [expr $bw2/$time*8/1000000]"
        #Reset the bytes_ values on the traffic sinks
        $sink0 set bytes_ 0
        $sink1 set bytes_ 0
        $sink2 set bytes_ 0
        #Re-schedule the procedure
        $ns at [expr $now+$time] "record"
}
```

This procedure reads the number of bytes received from the three traffic sinks. Then it calculates the bandwidth (in MBit/s) and writes it to the three output files together with the current time before it resets the bytes_ values on the traffic sinks. Then it re-schedules itself.

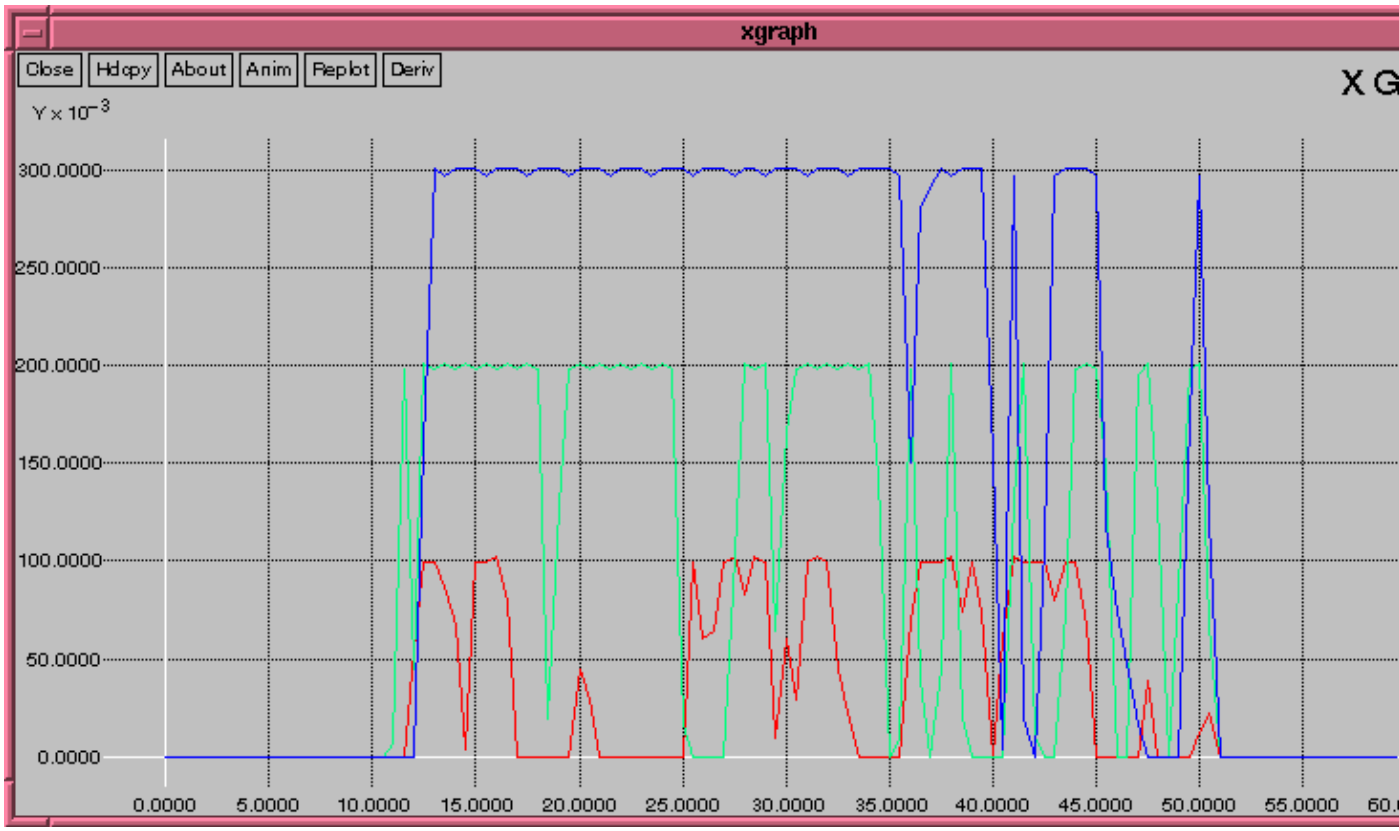
VIII.3. Running the Simulation

We can now schedule the following events:

```
$ns at 0.0 "record"
$ns at 10.0 "$source0 start"
$ns at 10.0 "$source1 start"
$ns at 10.0 "$source2 start"
$ns at 50.0 "$source0 stop"
$ns at 50.0 "$source1 stop"
$ns at 50.0 "$source2 stop"
$ns at 60.0 "finish"

$ns run
```

First, the 'record' procedure is called, and afterwards it will re-schedule itself periodically every 0.5 seconds. Then the three traffic sources are started at 10 seconds and stopped at 50 seconds. At 60 seconds, the 'finish' procedure is called. You can find the full example script here.

When you run the simulation, an xgraph window should open after some time which should look similar to this one:



As you can see, the bursts of the first flow peak at 0.1Mbit/s, the second at 0.2Mbit/s and the third at 0.3Mbit/s. Now you can try to modify the 'time' value in the 'record' procedure. Set it to '0.1' and see what happens, and then try '1.0'. It is very important to find a good 'time' value for each simulation scenario.

Note that the output files created by the 'record' procedure can also be used with gnuplot.

[Previous section] [Next section] [Back to the index]

ns-users
ns-users@isi.edu