ns Tutorial

1. Introduction

Disclaimer: This tutorial was originally developed by Marc Greis. Currently the tutorial is maintained and being expanded by the VINT group.

Please note: if there is any problem with the example scripts provided below, please refer to the test suites (~ns/tcl/test/test-suite-greis.tcl and test-suite-WLtutorial.tcl) for and lastest updates and let us know, thanks.

Since you have found this page, I assume that you already know what ns is and where you can get it from. If not, I suggest you either go to the web page for the VINT project or the web page for ns (version 2). Note: In these pages I describe ns version 2. Version 1 is different, though there is a backwards compatibility library in version 2.

The purpose of these pages is to make it easier for new ns users to use ns and nam, to create their own simulation scenarios for these tools and to eventually add new functionality to ns. I have found the existing documentation to be rather useful for users who already know the basic features of ns, while it can be a bit tedious for new users to extract the necessary information from the manuals and the example scripts. In this tutorial I will lead you through some simple examples, introducing more and more new features as we go along. The ultimate goal is that after a short time you are able to efficiently use ns and to find any further information you might need in the existing documentation. For this purpose I will also try to tell you where I found the information in this tutorial myself, so you not only learn how to use ns, but also how to use its documentation.

The web is probably the best medium for a tutorial like this, because it's not only possible to add pictures (or even animations) for the examples, but you can also directly download the examples if you don't have the time for typing them in yourself (though I would suggest doing that at least for the first few examples).

If you have any suggestions, find any bugs or problems, have any comments and also if you have any new (well-documented) examples that could be added here, please send email to ns-users mailing list.

2. Documentation

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.

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.

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.

3. The Basics

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'.

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.

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.

4. The First Tcl

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.

Nam snap shot

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.

Nam snap shot

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.

5. More Interesting

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.

Nam snap shot

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.

Nam snap shot

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.

Nam snap shot

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.

Nam snap shot

You can download the full example here.

6. Network Dynamics

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.

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.

Nam snap shot

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.

Nam snap shot

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.

Nam snap shot

You can download the full example here.

7. A New Protocol for ns

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 Greis
greis@cs.uni-bonn.de

8. Xgraph

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:

Nam snap shot

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 packet-size $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:

XGraph snapshot

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

9. Wireless Simulations

In this section, you are going to learn to use the mobile wireless simulation model available in ns. The section consists of two parts. In the first subsection, we discuss how to create and run a simple 2-node wireless network simulation. In second subsection, we will extend our example (in subsection 1) to create a relatively more complex wireless scenario.

IMPORTANT: This tutorial chapter uses new node APIs which are available as of ns-2.1b6, released January 18, 2000. If you have an earlier version of ns you must upgrade to use these features.

IX.1. Creating a simple wireless scenario

We are going to simulate a very simple 2-node wireless scenario. The topology consists of two mobilenodes, node_(0) and node_(1). The mobilenodes move about within an area whose boundary is defined in this example as 500mX500m. The nodes start out initially at two opposite ends of the boundary. Then they move towards each other in the first half of the simulation and again move away for the second half. A TCP connection is setup between the two mobilenodes. Packets are exchanged between the nodes as they come within hearing range of one another. As they move away, packets start getting dropped.

Just as with any other ns simulation, we begin by creating a tcl script for the wireless simulation. We will call this file simple-wireless.tcl. If you want to download a copy of simple-wireless.tcl click here.

A mobilenode consists of network components like Link Layer (LL), Interface Queue (IfQ), MAC layer, the wireless channel nodes transmit and receive signals from etc. For details about these network components see section 1 of chapter 15 of ns Notes & Documentation (now renamed as ns Manual). At the beginning of a wireless simulation, we need to define the type for each of these network components. Additionally, we need to define other parameters like the type of antenna, the radio-propagation model, the type of ad-hoc routing protocol used by mobilenodes etc. See comments in the code below for a brief description of each variable defined. The array used to define these variables, val() is not global as it used to be in the earlier wireless scripts. For details and available optional values of these variables, see chapter 15 (mobile networking in ns) of ns documentation. We begin our script simple-wireless.tcl with a list of these different parameters described above, as follows:

```
# ======================================================================
# Define options
# ======================================================================
set val(chan)          Channel/WirelessChannel  ;# channel type
set val(prop)          Propagation/TwoRayGround ;# radio-propagation model
set val(ant)           Antenna/OmniAntenna      ;# Antenna type
set val(ll)            LL                       ;# Link layer type
set val(ifq)           Queue/DropTail/PriQueue  ;# Interface queue type
set val(ifqlen)        50                       ;# max packet in ifq
set val(netif)         Phy/WirelessPhy          ;# network interface type
set val(mac)           Mac/802_11               ;# MAC type
set val(rp)            DSDV                     ;# ad-hoc routing protocol
set val(nn)            2                        ;# number of mobilenodes
```

Next we go to the main part of the program and start by creating an instance of the simulator,

```
set ns_    [new Simulator]
```

Then setup trace support by opening file simple.tr and call the procedure trace-all {} as follows:

```
set tracefd     [open simple.tr w]
$ns_ trace-all $tracefd
```

Next create a topology object that keeps track of movements of mobilenodes within the topological boundary.

```
set topo        [new Topography]
```

We had earlier mentioned that mobilenodes move within a topology of 500mX500m. We provide the topography object with x and y co-ordinates of the boundary, (x=500, y=500) :

```
$topo load_flatgrid 500 500
```

The topography is broken up into grids and the default value of grid resolution is 1. A diferent value can be passed as a third parameter to load_flatgrid {} above.

Next we create the object God, as follows:

```
create-god $val(nn)
```

Quoted from CMU document on god, "God (General Operations Director) is the object that is used to store global information about the state of the environment, network or nodes that an omniscent observer would have, but that should not be made known to any participant in the simulation." Currently, God object stores the total number of mobilenodes and a table of shortest number of hops required to reach from one node to another. The next hop information is normally loaded into god object from movement pattern files, before simulation begins, since calculating this on the fly during simulation runs can be quite time consuming. However, in order to keep this example simple we avoid using movement pattern files and thus do not provide God with next hop information. The usage of movement pattern files and feeding of next hop info to God shall be shown in the example in the next sub-section.

The procedure create-god is defined in ~ns/tcl/mobility/com.tcl, which allows only a single global instance of the God object to be created during a simulation. In addition to the evaluation functionalities, the God object is called internally by MAC objects in mobilenodes. So even though we may not utilise God for evaluation purposes,(as in this example) we still need to create God.

Next, we create mobilenodes. The node creation APIs have been revised and here we shall be using the new APIs to create mobilenodes. IMPORTANT NOTE: The new APIs are not available with ns2.1b5 release. Download the daily snapshot version if the next release (2.1b6 upwards) is not as yet available.

First, we need to configure nodes before we can create them. Node configuration API may consist of defining the type of addressing (flat/hierarchical etc), the type of adhoc routing protocol, Link Layer, MAC layer, IfQ etc. The configuration API can be defined as follows:

```
                                   (parameter examples)
# $ns_ node-config -addressingType flat or hierarchical or expanded
#                  -adhocRouting   DSDV or DSR or TORA
#                  -llType         LL
#                  -macType        Mac/802_11
#                  -propType       "Propagation/TwoRayGround"
#                  -ifqType        "Queue/DropTail/PriQueue"
#                  -ifqLen         50
#                  -phyType        "Phy/WirelessPhy"
#                  -antType        "Antenna/OmniAntenna"
#                  -channelType    "Channel/WirelessChannel"
#                  -topoInstance   $topo
#                  -energyModel    "EnergyModel"
#                  -initialEnergy  (in Joules)
#                  -rxPower        (in W)
#                  -txPower        (in W)
#                  -agentTrace     ON or OFF
#                  -routerTrace    ON or OFF
#                  -macTrace       ON or OFF
#                  -movementTrace  ON or OFF
```

All default values for these options are NULL except:
addressingType: flat

We are going to use the default value of flat addressing; Also lets turn on only AgentTrace and RouterTrace; You can experiment with the traces by turning all of them on. AgentTraces are marked with AGT, RouterTrace with RTR and MacTrace with MAC in their 5th fields. MovementTrace, when turned on, shows the movement of the mobilenodes and the trace is marked with M in their 2nd field.

The configuration API for creating mobilenodes looks as follows:

```
# Configure nodes
        $ns_ node-config -adhocRouting $val(rp) \
                         -llType $val(ll) \
                         -macType $val(mac) \
                         -ifqType $val(ifq) \
                         -ifqLen $val(ifqlen) \
                         -antType $val(ant) \
                         -propType $val(prop) \
                         -phyType $val(netif) \
                         -topoInstance $topo \
                         -channelType $val(chan) \
                         -agentTrace ON \
                         -routerTrace ON \
                         -macTrace OFF \
                         -movementTrace OFF
```

Next we create the 2 mobilenodes as follows:

```
        for {set i 0} {$i < $val(nn) } {incr i} {
                set node_($i) [$ns_ node ]
                $node_($i) random-motion 0       ;# disable random motion
        }
```

The random-motion for nodes is disabled here, as we are going to provide node position and movement(speed & direction) directives next.

Now that we have created mobilenodes, we need to give them a position to start with,

```
#
# Provide initial (X,Y, for now Z=0) co-ordinates for node_(0) and node_(1)
#
$node_(0) set X_ 5.0
$node_(0) set Y_ 2.0
$node_(0) set Z_ 0.0

$node_(1) set X_ 390.0
$node_(1) set Y_ 385.0
$node_(1) set Z_ 0.0
```

Node0 has a starting position of (5,2) while Node1 starts off at location (390,385).

Next produce some node movements,

```
#
# Node_(1) starts to move towards node_(0)
#
$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0"
$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"

# Node_(1) then starts to move away from node_(0)
$ns_ at 100.0 "$node_(1) setdest 490.0 480.0 15.0"
```

$ns_ at 50.0 "$node_(1) setdest 25.0 20.0 15.0" means at time 50.0s, node1 starts to move towards the destination (x=25,y=20) at a speed of 15m/s. This API is used to change direction and speed of movement of the mobilenodes.

Next setup traffic flow between the two nodes as follows:

```
# TCP connections between node_(0) and node_(1)

set tcp [new Agent/TCP]
$tcp set class_ 2
set sink [new Agent/TCPSink]
$ns_ attach-agent $node_(0) $tcp
$ns_ attach-agent $node_(1) $sink
```

```
$ns_ connect $tcp $sink
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns_ at 10.0 "$ftp start"
```

This sets up a TCP connection betwen the two nodes with a TCP source on node0.

Then we need to define stop time when the simulation ends and tell mobilenodes to reset which actually resets thier internal network components,

```
#
# Tell nodes when the simulation ends
#
for {set i 0} {$i < $val(nn) } {incr i} {
    $ns_ at 150.0 "$node_($i) reset";
}
$ns_ at 150.0001 "stop"
$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
proc stop {} {
    global ns_ tracefd
    close $tracefd
}
```

At time 150.0s, the simulation shall stop. The nodes are reset at that time and the "$ns_ halt" is called at 150.0002s, a little later after resetting the nodes. The procedure stop{} is called to flush out traces and close the trace file.

And finally the command to start the simulation,

```
puts "Starting Simulation..."
$ns_ run
```

Save the file simple-wireless.tcl. In order to download a copy of the file click here. Next run the simulation in the usual way (type at prompt: "ns simple-wireless.tcl")

At the end of the simulation run, trace-output file simple.tr is created. As we have turned on the AgentTrace and RouterTrace we see DSDV routing messages and TCP pkts being received and sent by Router and Agent objects in node _0_ and _1_. Note that all wireless traces starts with WL in their first field. See Chapter 15 of ns documentation for details on wireless trace. We see TCP flow starting at 10.0s from node0. Initially both the nodes are far apart and thus TCP pkts are dropped by node0 as it cannot hear from node1. Around 81.0s the routing info begins to be exchanged between both the nodes and around 100.0s we see the first TCP pkt being received by the Agent at node1 which then sends an ACK back to node0 and the TCP connection is setup. However as node1 starts to move away from node0, the connection breaks down again around time 116.0s. Pkts start getting dropped as the nodes move away from one another.

IX.2. Using node-movement/traffic-pattern files and other features in wireless simulations

As an extension to the previous sub-section, we are going to simulate a simple multihop wireless scenario consisting of 3 mobilenodes here. As before, the mobilenodes move within the boundaries of a defined topology. However the node movements for this example shall be read from a node-movement file called scen-3-test. scen-3-test defines random node movements for the 3 mobilenodes within a topology of 670mX670m. This file is available as a part of the ns distribution and can be found, along with other node-movement files, under directory ~ns/tcl/mobility/scene. Random node movement files like scen-3-test can be generated using CMU's node-movement generator "setdest". Details on generation of node movement files are covered in section XI.2 of this tutorial.

In addition to node-movements, traffic flows that are setup between the mobilenodes, are also read from a traffic-pattern file called cbr-3-test. cbr-3-test is also available under ~ns/tcl/mobility/scene. Random CBR and TCP flows are setup between the 3 mobilenodes and data packets are sent, forwarded or received by nodes within hearing range of one another. See cbr-3-test to find out more about the traffic flows that are setup. These traffic-pattern files can also be generated using CMU's TCP/CBR traffic generator script. More about this is discussed in section XI.1 of this tutorial.

We shall make changes to the script, simple-wireless.tcl, we had created in section IX.1. and shall call the resulting file wireless1.tcl. For a copy of wireless1.tcl download from here. In addition to the variables (LL, MAC, antenna etc) that were declared at the beginning of the script, we now define some more parameters like the connection-pattern and node-movement file, x and y values for the topology boundary, a seed value for the random-number generator, time for the simulation to stop, for convinience. They are listed as follows:

```
set val(chan)       Channel/WirelessChannel
set val(prop)       Propagation/TwoRayGround
set val(netif)      Phy/WirelessPhy
set val(mac)        Mac/802_11
set val(ifq)        Queue/DropTail/PriQueue
set val(ll)         LL
set val(ant)        Antenna/OmniAntenna
set val(x)              670   ;# X dimension of the topography
set val(y)              670   ;# Y dimension of the topography
set val(ifqlen)         50            ;# max packet in ifq
set val(seed)           0.0
set val(adhocRouting)   DSR
set val(nn)             3             ;# how many nodes are simulated
set val(cp)             "../mobility/scene/cbr-3-test"
set val(sc)             "../mobility/scene/scen-3-test"
set val(stop)           2000.0           ;# simulation time
```

Number of mobilenodes is changed to 3; Also we use DSR (dynamic source routing) as the adhoc routing protocol inplace of DSDV (Destination sequence distance vector);

After creation of ns_, the simulator instance, open a file (wireless1-out.tr) for wireless traces. Also we are going to set up nam traces.

```
set tracefd  [open wireless1-out.tr w]      ;# for wireless traces
$ns_ trace-all $tracefd

set namtrace [open wireless1-out.nam w]           ;# for nam tracing
$ns_ namtrace-all-wireless $namtrace $val(x) $val(y)
```

Next (after creation of mobilenodes) source node-movement and connection pattern files that were defined earlier as val(sc) and val(cp) respectively.

```
#
# Define node movement model
#
puts "Loading connection pattern..."
source $val(cp)

#
# Define traffic model
#
puts "Loading scenario file..."
source $val(sc)
```

In node-movement file scen-3-test, we see node-movement commands like

```
$ns_ at 50.000000000000 "$node_(2) setdest 369.463244915743 \
170.519203111152 3.371785899154"
```

This, as described in earlier sub-section, means at time 50s, node2 starts to move towards destination (368.4,170.5) at a speed of 3.37m/s. We also see other lines like

```
$god_ set-dist 1 2 2
```

These are command lines used to load the god object with the shortest hop information. It means the shortest path between node 1 and 2 is 2 hops. By providing this information, the calculation of shortest distance between nodes by the god object during simulation runs, which can be quite time-consuming, is prevented.

The setdest program (see section XI.2) generates movement pattern files using the random waypoint algorithm. The node-movement files generated using setdest (like scen-3-test) already include lines like above to load the god object with the appropriate information at the appropriate time.

A program called calcdest (~ns/indep-utilities/cmu-scen-gen/setdest/calcdest) can be used to annotate movement pattern files generated by other means with the lines of god information. calcdest makes several assumptions about the format of the lines in the input movement pattern file which will cause it to fail if the file is not formatted properly. If calcdest rejects a movement pattern file you have created, the easiest way to format it properly is often to load it into ad-hockey and then save it out again. If ad-hockey can read your input correctly, its output will be properly formatted for calcdest.

Both setdest and calcdest calculate the shortest number of hops between nodes based on the nominal radio range, ignoring any effects that might be introduced by the propagation model in an actual simulation. The nominal range is either provided as an argument to the programs, or extracted from the header in node-movement pattern files.

The path length information provided to god was used by CMU's Monarch Project to analyze the path length optimality of ad hoc network routing protocols, and so was printed out as part of the CMUTrace output for each packet.

Other uses that CMU has found for the information are:

- Characterizing the rate of topology change in a movement pattern.
- Identifying the frequency and size of partitions.
- Experimenting with the behavior of the routing protocols if the god information is used to provide them with ``perfect" neighbor information at zero cost.

Next add the following lines for providing initial position of nodes in nam. However note that only node movements can currently be seen in nam . Dumping of traffic data and thus visualization of data pkt movements in nam for wireless scenarios is still not supported (future work).

```
# Define node initial position in nam
for {set i 0} {$i < $val(nn)} {incr i} {

        # 20 defines the node size in nam, must adjust it according to your
        # scenario size.
        # The function must be called after mobility model is defined
        $ns_ initial_node_pos $node_($i) 20
}
```

Next add informative headers for the CMUTrace file, just before the line "ns_ run" :

```
puts $tracefd "M 0.0 nn $val(nn) x $val(x) y $val(y) rp $val(adhocRouting)"
puts $tracefd "M 0.0 sc $val(sc) cp $val(cp) seed $val(seed)"
puts $tracefd "M 0.0 prop $val(prop) ant $val(ant)"
```

The rest of the script remains unchanged.

Save the file wireless1.tcl. Make sure the connection-pattern and node-movement files exist under the directories as declared above.
Run the script by typing at the prompt:

```
ns  wireless1.tcl
```

On completion of the run, CMUTrace output file "wireless1-out.tr" and nam output file "wireless1-out.nam" are created. Running wireless1-out.nam we see the three mobilenodes moving in nam window. However as mentioned earlier no traffic flow can be seen (not supported as yet). For a variety of coarse and fine grained trace outputs turn on/off AgentTrace, RouteTrace, MacTrace and movementTrace as shown earlier in the script. From the CMUTrace output we find nodes 0 and 2 are out of range and so cannot hear one another. Node1 is in range with nodes 0 and 2 and can communicate with both of them. Thus all pkts destined for nodes 0 and 2 are routed through node 1. For details on CMUTraces see chapter 15 of ns documentation.

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

VINT

ns-users@isi.edu

10. Wired-cum-Wireless and MobileIP Simulations

IMPORTANT: This tutorial chapter uses new node APIs which are not available in the ns2.1b5 version. So please download the daily snapshot unless a release is made for version ns2.1b6 or higher. The current snapshot version is updated daily, so please check the validation test results for that day before downloading, as these snapshots can sometimes be unstable due to ongoing changes made by ns-developers.

X.1. Creating a simple wired-cum-wireless scenario

The wireless simulation described in section IX, supports multi-hop ad-hoc networks or wireless LANs. But we may need to simulate a topology of multiple LANs connected through wired nodes, or in other words we need to create a wired-cum-wireless topology.

In this section we are going to extend the simple wireless topology created in section IX to create a mixed scenario consisting of a wireless and a wired domain, where data is exchanged between the mobile and non-mobile nodes. We are going to make modifications to the tcl script called wireless1.tcl created in section IX.2 and name the resulting wired-cum-wireless scenario file wireless2.tcl.

For the mixed scenario, we are going to have 2 wired nodes, W(0) and W(1), connected to our wireless domain consisting of 3 mobilenodes (nodes 0, 1 & 2) via a base-station node, BS. Base station nodes are like gateways between wireless and wired domains and allow packets to be exchanged between the two types of nodes. For details on base-station node please see section 2 (wired-cum-wireless networking) of chapter 15 of ns notes&doc (now renamed as ns Manual). Fig1. shows the topology for this example described above.

Fig.1

Let us begin by checking what changes need to be made to the list of variables defined at the beginning of wireless1.tcl.

The Adhoc routing protocol is changed to DSDV. Also, we define TCP and CBR connections between the wired and wireless nodes in the script itself. So we won't need to use the connection pattern file used in earlier simulation. Also change the simulation stop time. Note here that we use array opt() instead of val() simply to illustrate that this is no longer a global array variable and its scope is defined only in the test script.

```
set opt(adhocRouting)   DSDV
set opt(cp)             ""       ;# cp file not used
set opt(stop)           300      ;# time to stop simulation
```

We define the start times for TCP flows here:

```
set opt(ftp1-start)      160.0
set opt(ftp2-start)      170.0
```

Also add the following line to define number of wired and base-station nodes:

```
set num_wired_nodes      2
set num_bs_nodes         1
```

Now we move to the main part of the program. For mixed simulations we need to use hierarchical routing in order to route packets between wireless and wired domains. As explained in section 15.2.1 of ns Manual, in ns, the routing information for wired nodes are based on connectivity of the topology, i.e how are nodes connected to one another through Links. This connectivity information is used to populate the forwarding tables in each wired node. However wireless nodes have no concept of "links". Packets are routed in a wireless topology using their adhoc routing protocols which build forwarding tables by exchanging routing queries among its neighbours. So inorder to exchange pkts among these wired and wireless nodes, we use base-stations which act as gateways between the two domains. We seggregate wired and wireless nodes by placing them in different domains. Domains and sub-domains (or clusters as they are called here) are defined by means of hierarchical topology structure as shown below. After line "set ns [new Simulator]", add the following lines:

```
$ns_ node-config -addressType hierarchical
AddrParams set domain_num_ 2           ;# number of domains
lappend cluster_num 2 1                ;# number of clusters in each
                                       ;#domain
AddrParams set cluster_num_ $cluster_num
lappend eilastlevel 1 1 4              ;# number of nodes in each cluster
AddrParams set nodes_num_ $eilastlevel ;# for each domain
```

In the above lines we first configure the node object to have addresstype as Hierarchical. Next the topology hierarchy is defined. Number of domains in this topology is 2 (one for the wired nodes and one for the wireless). Number of clusters in each of these domains is defined as "2 1" which indicates the first domain (wired) to have 2 clusters and the second (wireless) to have 1 cluster. The next line defines the number of nodes in each of these clusters which is "1 1 4"; i.e one node in each of the first 2 clusters (in wired domain) and 4 nodes in the cluster in the wireless domain. So the topology is defined into a 3-level hierarchy (see the topology figure above).

Next we setup tracing for the simulation. Note here that for wired-cum-wireless simulation traces may be generated for both wired and wireless domains. Both the traces are written into the same output file defined here as wireless2-out.tr. In order to differentiate wireless traces from wired ones, all wireless traces begin with "WL". We also setup nam traces. As mentioned earlier nam traces for wireless nodes currently show node movements only.

```
set tracefd  [open wireless2-out.tr w]
set namtrace [open wireless2-out.nam w]
$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $opt(x) $opt(y)
```

Next we need to create the wired, wireless and base-station nodes. Note here that for all node creations, you have to pass the hierarchical address of the node. So after line "create-god $opt(nn)", add the following lines for creating wired nodes:

```
# create wired nodes
set temp {0.0.0 0.1.0}           ;# hierarchical addresses to be used
for {set i 0} {$i < $num_wired_nodes} {incr i} {
    set W($i) [$ns_ node [lindex $temp $i]]
}
```

In order to create base-station node, we need to configure the node structure as shown below. This is part of the new node API which consists of first configuring and then creating nodes. Refer to node API in chapterIX for details about the new node API. Since base-station nodes are gateways between wired and wireless domains they need to have wired routing mechanism turned on which is done by setting node-config option -wiredRouting ON. After creating the base-station node we reconfigure for wireless node and so turn wiredRouting OFF. All other node-config options used for base-station remains the same for mobilenode. Also the BS(0) node is assigned as the base-station node for all the mobilenodes in the wireless domain, so that all pkts originating from mobilenodes and destined outside the wireless domain, will be forwarded by mobilenodes towards their assigned base-station.

Note that it is important for the base-station node to be in the same domain as the wireless nodes. This is so that all pkts originating from the wired domain, and destined for a wireless node will reach the base-station which then uses its adhoc routing protocol to route the pkt to its correct destination. Thus in a mixed simulation involving wired and wireless nodes its necessary :
1) to turn on hierarchical routing
2) to create separate domains for wired and wireless nodes. There may be multiple wired and wireless domains to simulate multiple networks.
3) to have one base-station node in every wireless domain, thru which the wireless nodes may communicate with nodes outside their domain.

Let us go step by step for this example to see how the hierarchy is created. Here we have two domains, domain 0 , for wired and domain 1, for wireless. The two wired nodes are placed in 2 separate clusters, 0 and 1; thus their addresses look like 0(domain 0).0(cluster 0).0(only node) and 0 (same domain 0).1(cluster 1).0(again only node).
As for the wireless nodes, they are in domain 1; we have defined one cluster (0), so all nodes are in this cluster. Hence the addresses are:

Base-station: 1(second domain,1).0(cluster 0).0(first node in cluster)
WL node#1 : 1.0.1(second node in cluster)
WL node#2 : 1.0.2(third node)
WL node#3 : 1.0.3(fourth node)
We could have placed the two wired nodes in the same cluster in wired domain 0. Also we could have placed other wireless nodes in different clusters in wireless domain 1. Also depending on our topology we may have got rid of clusters altogether, and simply have had 2 layers of hierarchy, the domains and the nodes.

```
# configure for base-station node
$ns_ node-config -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propType $opt(prop) \
                 -phyType $opt(netif) \
                 -channelType $opt(chan) \
                 -topoInstance $topo \
                 -wiredRouting ON \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace OFF

#create base-station node
set temp {1.0.0 1.0.1 1.0.2 1.0.3}   ;# hier address to be used for
                                     ;# wireless domain
set BS(0) [ $ns_ node [lindex $temp 0]]
$BS(0) random-motion 0               ;# disable random motion

#provide some co-ordinates (fixed) to base station node
$BS(0) set X_ 1.0
$BS(0) set Y_ 2.0
$BS(0) set Z_ 0.0

# create mobilenodes in the same domain as BS(0)
# note the position and movement of mobilenodes is as defined
# in $opt(sc)
# Note there has been a change of the earlier AddrParams
# function 'set-hieraddr' to 'addr2id'.

#configure for mobilenodes
$ns_ node-config -wiredRouting OFF

# now create mobilenodes
for {set j 0} {$j < $opt(nn)} {incr j} {
    set node_($j) [ $ns_ node [lindex $temp \
            [expr $j+1]] ]
    $node_($j) base-station [AddrParams addr2id \
            [$BS(0) node-addr]]   ;# provide each mobilenode with
                                  ;# hier address of its base-station
}
```

Next connect wired nodes and BS and setup TCP traffic between wireless node, node_(0) and wired node W(0), and between W(1) and node_(2), as shown below:

```
#create links between wired and BS nodes
$ns_ duplex-link $W(0) $W(1) 5Mb 2ms DropTail
$ns_ duplex-link $W(1) $BS(0) 5Mb 2ms DropTail

$ns_ duplex-link-op $W(0) $W(1) orient down
$ns_ duplex-link-op $W(1) $BS(0) orient left-down

# setup TCP connections
set tcp1 [new Agent/TCP]
$tcp1 set class_ 2
set sink1 [new Agent/TCPSink]
$ns_ attach-agent $node_(0) $tcp1
$ns_ attach-agent $W(0) $sink1
$ns_ connect $tcp1 $sink1
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ns_ at $opt(ftp1-start) "$ftp1 start"

set tcp2 [new Agent/TCP]
$tcp2 set class_ 2
set sink2 [new Agent/TCPSink]
$ns_ attach-agent $W(1) $tcp2
```

```
$ns_ attach-agent $node_(2) $sink2
$ns_ connect $tcp2 $sink2
set ftp2 [new Application/FTP]
$ftp2 attach-agent $tcp2
$ns_ at $opt(ftp2-start) "$ftp2 start"
```

This would be followed by the remaining lines from wireless1.tcl (sourcing cp and sc files, telling mobilenodes when to stop and finally running ns). It is possible that some lines of code in wireless2.tcl have not been discussed here. For a complete copy of script wireless2.tcl, download from here.

Run the script. The ns and nam trace files are generated at the end of simulation run. Running wireless2-out.nam shows the movement of mobilenodes and traffic in the wired domain. As mentioned earlier, traffic flow for mobilenodes is not as yet supported in nam. In trace file wireless2-out.tr we see traces for both wired domain and wireless domain (preceeding with "WL" for wireless). At 160.0s, a TCP connection is setup between _3_, (which is node_(0)) and 0, (which is W(0)). Note that the node-ids are created internally by the simulator and are assigned in the order of node creation. At 170s, another TCP connection is setup in the opposite direction, from the wired to the wireless domain. For details on CMUTraces see chapter 15 of ns documentation.

X.2. Running MobileIP in a simple wired-cum-wireless topology

So far we have created a wired-cum-wireless topology and have exchanged pkts between a wired and wireless domain via a base-station. But a mobilenode may roam outside the domain of its basestation and should still continue to receive packets destined to it. In other words it would be interesting to extend mobileIP support in the wired-cum-wireless scenario we created in section X.1.
For this example we have the same wired domain consisting of 2 wired nodes, W0 and W1. We have 2 base-station nodes and call them HomeAgent(HA) and ForeignAgent(FA) respectively. The wired node W1 is connected to HA and FA as shown in the figure below. There is a roaming mobilenode called MobileHost(MH) that moves between its home agent and foreign agents. We will set up a TCP flow between W0 and MH. As MH moves out from the domain of its HA, into the domain of FA, we will observe how pkts destined for MH is redirected by its HA to the FA as per mobileIP protocol definitions. See fig2 below for the topology described above.

Fig.2

We shall edit wireless2.tcl created in section X.1 to create the wireless-mip script called wireless3.tcl. It may be possible that the whole of wireless3.tcl is not discussed here. So for convinience, you may download a copy of wireless3.tcl from here.

Change number of mobilenodes and time of simulation,

```
set opt(nn)             1           ;# just one MH
set opt(stop)           250
```

In this example we will set up the TCP connection as well as define movement of the MH in the script itself. Hence we are not going to use the cp and sc files.

```
set opt(cp)                ""
set opt(sc)                ""
```

Define the TCP flow starttime,

```
set opt(ftp1-start)     100.0
```

Change number of wired, base-station and mobile nodes. However note that the variable num_bs_nodes is not really used in this script. The base-station nodes, HA and FA, are individually created and handled.

```
set num_wired_nodes      2
```

After the 2 lines creating ns instance and setting address format to hierarchical, add the following lines to define the topological hierarchy. It is quite similar to that of wireless2.tcl except that now we have a third domain for the FA. Change the cluster and node parameters accordingly.

```
AddrParams set domain_num_ 3           ;# number of domains
lappend cluster_num 2 1 1              ;# number of clusters in each domain
AddrParams set cluster_num_ $cluster_num
lappend eilastlevel 1 1 2 1            ;# number of nodes in each cluster
AddrParams set nodes_num_ $eilastlevel ;# of each domain
```

Next set up ns trace and nam files for wireless-mip,

```
set tracefd  [open wireless3-out.tr w]
set namtrace [open wireless3-out.nam w]
$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $opt(x) $opt(y)
```

So in this topology we have one wired domain (denoted by 0) and 2 wireless domains (denoted by 1 & 2 respectively). Hence as described in section X.1, the wired node addresses remain the same, 0.0.0 and 0.1.0. In the first wireless domain (domain 1) we have base-station, HA and mobilenode, MH, in the same single cluster. Their addresses are 1.0.0 and 1.0.1 respectively. For the second wireless domain (domain 2) we have a base-station, FA with an address of 2.0.0. However in the course of the simulation, the MH will move into the domain of FA and we shall see how pkts originating from a wired domain and destined to MH will reach it as a result of the MobileIP protocol.

Wired nodes will be created as earlier. However in place of a single base-station node, a HA and FA will be created. Note here that to turn the mobileIP flag on we have configure the node structure accordingly using option -mobileIP ON.

```
# Configure for ForeignAgent and HomeAgent nodes
$ns_ node-config -mobileIP ON \
                 -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propType $opt(prop) \
                 -phyType $opt(netif) \
                 -channelType $opt(chan) \
                 -topoInstance $topo \
                 -wiredRouting ON \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace OFF

# Create HA and FA
set HA [$ns_ node 1.0.0]
set FA [$ns_ node 2.0.0]
$HA random-motion 0
$FA random-motion 0

#provide some co-ord (fixed) to these base-station nodes.
$HA set X_ 1.000000000000
$HA set Y_ 2.000000000000
$HA set Z_ 0.000000000000

$FA set X_ 650.000000000000
$FA set Y_ 600.000000000000
$FA set Z_ 0.000000000000
```

Next create the mobilehost as follows. Note as before we have to turn off the option -wiredRouting (used for creation of base-station nodes) before creating mobilenodes. Also the HA is setup as the home-agent for the mobilehost. The MH has an address called the care-of-address (COA). Based on the registration/beacons exchanged between the MH and the base-station node (of the domain the MH is currently in), the base-station's address is assigned as the MH's COA. Thus in this simulation, address of HA is assigned initially as the COA of MH. As MH moves in to the domain of FA, its COA changes to that of the FA. For details on MobileIP implementation in ns, read section 15.2.2 of (in wireless networking chapter) ns documentation. Also see files mip.{cc,h}, mip-reg.{cc,h}, tcl/lib/{ns-mip.tcl, ns-wireless-mip.tcl}.

```
# configure for mobilehost
$ns_ node-config -wiredRouting OFF
```

```
# create mobilehost that would be moving between HA and FA.
# note address of MH indicates its in the same domain as HA.
set MH [$ns_ node 1.0.1]
set node_(0) $MH
set HAaddress [AddrParams set-hieraddr [$HA node-addr]]
[$MH set regagent_] set home_agent_ $HAaddress

# movement of the MH
$MH set Z_ 0.000000000000
$MH set Y_ 2.000000000000
$MH set X_ 2.000000000000

# MH starts to move towards FA (640, 610) at a speed of 20m/s
$ns_ at 100.000000000000 "$MH setdest 640.000000000000 610.000000000000
20.000000000000"

# and goes back to HA (2, 2) at a speed of 20 m/s
$ns_ at 200.000000000000 "$MH setdest 2.000000000000 2.000000000000
20.000000000000"
```

Create links between Wired nodes and HA/FA and setup TCP connection:

```
# create links between wired and BaseStation nodes
$ns_ duplex-link $W(0) $W(1) 5Mb 2ms DropTail
$ns_ duplex-link $W(1) $HA 5Mb 2ms DropTail
$ns_ duplex-link $W(1) $FA 5Mb 2ms DropTail

$ns_ duplex-link-op $W(0) $W(1) orient down
$ns_ duplex-link-op $W(1) $HA orient left-down
$ns_ duplex-link-op $W(1) $FA orient right-down

# setup TCP connections between a wired node and the MobileHost

set tcp1 [new Agent/TCP]
$tcp1 set class_ 2
set sink1 [new Agent/TCPSink]
$ns_ attach-agent $W(0) $tcp1
$ns_ attach-agent $MH $sink1
$ns_ connect $tcp1 $sink1
```

```
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ns_ at $opt(ftp1-start) "$ftp1 start"
```

The rest of the script remains unchanged (i.e tell mobilenodes when the simulation stops). Save and run the script. Click here for a copy of the file wireless3.tcl.

While running the script, you may see warnings like "warning: Route to base_stn not known: dropping pkt". This means that as the MH moves from the domain of one base-station into domain of another there may be interim periods when it is not registered to any base-station and thus doesnot know whom to forward pkts destined outside its domain. On completion of the run, ns and nam trace output files "wireless3-out.tr" and "wireless3-out.nam" are created. The nam output shows the movement of the mobilehost and traffic flow in the wired domain. The ns trace output shows traces for both the wired nodes as well as the wireless domain. We see routine beacon broadcast/solicitations sent out by HA/FA and the MH. Initially the TCP pkts are handed down to MH directly by its HA. As MH moves away from HA domain into the domain of the FA, we find the pkts destined for MH, being encapsulated and forwarded to the FA which then strips off or decapsulates the pkt and hands it over to the MH.

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

VINT

ns-users@isi.edu

11. Generating Traffic-Connection and Movement Files

XI.1. Creating random traffic-pattern for wireless scenarios.

Random traffic connections of TCP and CBR can be setup between mobilenodes using a traffic-scenario generator script. This traffic generator script is available under ~ns/indep-utils/cmu-scen-gen and is called cbrgen.tcl. It can be used to create CBR and TCP traffics connections between wireless mobilenodes. In order to create a traffic-connection file, we need to define the type of traffic connection (CBR or TCP), the number of nodes and maximum number of connections to be setup between them, a random seed and incase of CBR connections, a rate whose inverse value is used to compute the interval time between the CBR pkts. So the command line looks like the following:

```
ns cbrgen.tcl [-type cbr|tcp] [-nn nodes] [-seed seed] [-mc connections]
[-rate rate]
```

The start times for the TCP/CBR connections are randomly generated with a maximum value set at 180.0s. Go through the tcl script cbrgen.tcl for the details of the traffic-generator implementation.

For example, let us try to create a CBR connection file between 10 nodes, having maximum of 8 connections, with a seed value of 1.0 and a rate of 4.0. So at the prompt type:

```
ns cbrgen.tcl -type cbr -nn 10 -seed 1.0 -mc 8 -rate 4.0 > cbr-10-test
```

From cbr-10-test file (into which the output of the generator is redirected) thus created, one of the cbr connections looks like the following:

```
#
# 2 connecting to 3 at time 82.557023746220864
#
set udp_(0) [new Agent/UDP]
$ns_ attach-agent $node_(2) $udp_(0)
set null_(0) [new Agent/Null]
$ns_ attach-agent $node_(3) $null_(0)
set cbr_(0) [new Application/Traffic/CBR]
$cbr_(0) set packetSize_ 512
$cbr_(0) set interval_ 0.25
$cbr_(0) set random_ 1
$cbr_(0) set maxpkts_ 10000
$cbr_(0) attach-agent $udp_(0)
$ns_ connect $udp_(0) $null_(0)
$ns_ at 82.557023746220864 "$cbr_(0) start"
```

Thus a UDP connection is setup between node 2 and 3. Total UDP sources (chosen between nodes 0-10) and total number of connections setup is indicated as 5 and 8 respectively, at the end of the file cbr-10-test.

Similarly TCP connection files can be created using "type" as tcp. An example would be:

```
ns cbrgen.tcl -type tcp -nn 25 -seed 0.0 -mc 8 > tcp-25-test
```

A typical connection from tcp-25-test looks like the following:

```
#
# 5 connecting to 7 at time 163.0399642433226
#
set tcp_(1) [$ns_ create-connection  TCP $node_(5) TCPSink $node_(7) 0]
$tcp_(1) set window_ 32
$tcp_(1) set packetSize_ 512
set ftp_(1) [$tcp_(1) attach-source FTP]
$ns_ at 163.0399642433226 "$ftp_(1) start"
```

XI.2. Creating node-movements for wireless scenarios.

The node-movement generator is available under ~ns/indep-utils/cmu-scen-gen/setdest directory and consists of setdest{.cc,.h} and Makefile. CMU's version of setdest used system dependent /dev/random and made calls to library functions initstate() for generating random numbers. That was replaced with a more portable random number generator (class RNG) available in ns. In order to compile the revised setdest.cc do the following:
1. Go to ns directory and run "configure" (you probably have done that already while building ns). This creates a makefile for setdest.
2.Go to indep-utils/cmu-scen-gen/setdest. Run "make" , which first creates a stand-alone object file for ~ns/rng.cc (the stand-alone version doesnot use Tclcl libs) and then creates the executable setdest.
3. Run setdest with arguments as shown below:

```
./setdest [-n num_of_nodes] [-p pausetime] [-s maxspeed] [-t simtime] \
      [-x maxx] [-y maxy] > [outdir/movement-file]
```

Lets say we want to create a node-movement scenario consisting of 20 nodes moving with maximum speed of 10.0m/s with an average pause between movement being 2s. We want the simulation to stop after 200s and the topology boundary is defined as 500 X 500. So our command line will look like:

```
./setdest -n 20 -p 2.0 -s 10.0 -t 200 -x 500 -y 500 > scen-20-test
```

The output is written to stdout by default. We redirect the output to file scen-20-test. The file begins with the initial position of the nodes and goes on to define the node movements.

```
$ns_ at 2.000000000000 "$node_(0) setdest 90.441179033457 44.896095544010
1.373556960010"
```

This line from scen-20-test defines that node_(0) at time 2.0s starts to move toward destination (90.44, 44.89) at a speed of 1.37m/s. These command lines can be used to change direction and speed of movement of mobilenodes.

Directives for GOD are present as well in node-movement file. The General Operations Director (GOD) object is used to store global information about the state of the environment, network, or nodes that an omniscent observer would have, but that should not be made known to any participant in the simulation.

Currently, the god object is used only to store an array of the shortest number of hops required to reach from one node to an other. The god object does not calculate this on the fly during simulation runs, since it can be quite time consuming. The information is loaded into the god object from the movement pattern file where lines of the form

```
$ns_ at 899.642 "$god_ set-dist 23 46 2"
```

are used to load the god object with the knowledge that the shortest path between node 23 and node 46 changed to 2 hops at time 899.642.

The setdest program generates node-movement files using the random waypoint algorithm. These files already include the lines to load the god object with the appropriate information at the appropriate time.

Another program calcdest (also available in ~ns/indep-utils/cmu-scen-gen/setdest) can be used to annotate movement pattern files generated by other means with the lines of god information. calcdest makes several assumptions about the format of the lines in the input movement pattern file which will cause it to fail if the file is not formatted properly. If calcdest rejects a movement pattern file you have created, the easiest way to format it properly is often to load it into ad-hockey and then save it out again. If ad-hockey can read your input correctly, its output will be properly formatted for calcdest.

Both calcdest and setdest calculate the shortest number of hops between nodes based on the nominal radio range, ignoring any effects that might be introduced by the propagation model in an actual simulation. The nominal range is either provided as an argument to the programs, or extracted from the header on the movement pattern file.

The path length information was used by the Monarch Project to analyze the path length optimality of ad hoc network routing protocols, and so was printed out as part of the CMUTrace output for each packet.

Other uses that CMU found for the information:

- Characterizing the rate of topology change in a movement pattern.
- Identifying the frequency and size of partitions.
- Experimenting with the behavior of the routing protocols if the god information is used to provide them with ``perfect'' neighbor information at zero cost.

Thus at the end of the node-movement file are listed information like number of destination unreachable, total number of route and connectivity changes for mobilenodes and the same info for each mobilenode.

The revised (more portable) version of setdest (files revised are: setdest{.cc,.h}, ~ns/rng{.cc,.h}, ~ns/Makefile.in) should be available from the latest ns distribution. If not, you could download the daily snapshot version of ns from ns-build page.

[Previous section] [Back to the index]

VINT

ns-users@isi.edu
