

Emulab Tutorial

Contents

- [Getting Started](#)
 - [Logging into the Web Interface](#)
 - [Designing a Network Topology](#)
 - [Beginning the Experiment](#)
 - [Using your Nodes](#)
 - [I need **root** access!](#)
 - [My node is wedged!](#)
 - [I've scrogged my disk!](#)
 - [I've finished my experiment](#)
 - [Getting Help!](#)
 - [Advanced Topics](#)
 - [A more advanced example](#)
 - [Installing RPMS automatically](#)
 - [Starting your application automatically](#)
 - [How do I know when all my nodes are ready?](#)
 - [Customizing an OS \(How to create a *delta*\)](#)
 - [Setting up IP routing between nodes](#)
 - [Batch Mode Experiments](#)
 - [Creating your own disk image](#)
-

Getting Started

This section of the tutorial describes how to run your first Testbed experiment. We cover basic NS syntax and various operational issues that you will need to know in order conduct experiments to completion. Later sections of the tutorial will cover more advanced topics such as loading your own RPMs automatically, running programs automatically, running batch jobs, creating your own disk images and loading those images on your nodes.

- [Logging into the Web Interface](#)
- [Designing a Network Topology](#)
- [Beginning the Experiment](#)
- [Using your Nodes](#)
- [I need **root** access!](#)
- [My node is wedged!](#)
- [I've scrogged my disk!](#)
- [I've finished my experiment](#)
- [Getting Help!](#)

• Logging Into the Web Interface

If you already have an account on the Testbed, all you need to do is go to [Emulab Home Page](#), enter your login name and your password, and then click on the "Login" button. If you don't have an account, click on the "Join Project" or "Start Project" links. For an overview of how you go

about getting an Emulab account, go to the ["How To Get Started"](#) page.

• Designing a Network Topology

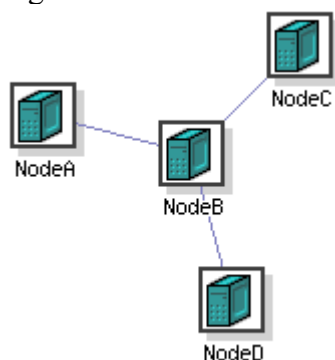
Part of the Testbed's power lies in its ability to assume many different topologies; the description of a such a topology is a necessary part of an experiment.

Emulab uses the "NS" ("Network Simulator") format to describe network topologies. This is substantially the same [Tcl](#)-based format used by [ns-2](#). Since the Testbed offers emulation, rather than simulation, these files are interpreted in a somewhat different manner than ns-2. Therefore, some ns-2 functionality may work differently than you expect, or may not be implemented at all. Please look for warnings of the form:

```
*** WARNING: Unsupported NS Statement!
Link type BAZ, using DropTail!
```

If you feel there is useful functionality missing, please let us know. Also, some [testbed-specific syntax](#) has been added, which with the inclusion of compatibility module ([tb_compat.tcl](#)), will be ignored by the NS simulator. This allows the same NS file to work on both the Testbed and ns-2, most of the time.

For those unfamiliar with the NS format, here is a small example (*We urge all new Emulab users to begin with a small 3-5 node experiment such as this, so that you will become familiar with NS syntax and the practical aspects of Emulab operation*). Let's say we are trying to create a test network which looks like the following:



(A is connected to B, B to C, and B to D.)

An NS file which would describe such a topology is as follows. First off, all NS files start with a simple prolog, declaring a simulator and including a file that allow you to use the special `tb-` commands:

```
# This is a simple ns script. Comments start with #.
set ns [new Simulator]
source tb_compat.tcl
```

Then define the 4 nodes in the topology.

```
set NodeA [$ns node]
set NodeB [$ns node]
```

```
set NodeC [$ns node]
set NodeD [$ns node]
```

Next define the 3 links between the nodes. NS syntax permits you to specify the bandwidth, latency, and queue type. For our example, we will define full speed links between B and C,D and a delayed link from node A to B.

```
$ns duplex-link $NodeA $NodeB 100Mb 50ms DropTail
$ns duplex-link $NodeB $NodeC 100Mb 0ms DropTail
$ns duplex-link $NodeB $NodeD 100Mb 0ms DropTail
```

In addition to the standard NS syntax above, a number of [extensions](#) have been added that allow you to better control your experiment. For example, you may specify what Operating System is booted on your nodes. We currently support FreeBSD 4.3 and Linux RedHat 7.1, as well as [OSKit](#) kernels on the testbed PCs. By default, Linux RedHat 7.1 is selected.

```
tb-set-node-os $NodeA FBSD-STD
tb-set-node-os $NodeC RHL-STD
```

You may also control what IP addresses are assigned to the experimental interfaces on your nodes. The experiment configuration software will select IP addresses for you, but if your experiment depends on particular IP addresses, you may specify them at each link. The following example sets the IP address of node B on the port going to node C:

```
tb-set-ip-interface $NodeB $NodeC 192.168.42.42
```

Lastly, all NS files end with an epilogue that instructs the simulator to start.

```
# Go!
$ns run
```

If you would like to try the above example, the completed [NS file](#) can be run as an experiment in your project. Another example ns script that shows off using the power of Tcl to generate topologies is [here](#).

• Beginning the Experiment

After logging on to the Testbed Web Interface, choose the "Begin Experiment" option from the menu. First select which project you want the experiment to be configured in. Most people will be a member of just one project, and will not have a choice. If you are a member of multiple projects, be sure to select the correct project from the menu.

Next fill in the 'Name' and 'Description' fields. The Name should be a single word (no spaces) identifier, while the Description is a multi word description of your experiment. In the "Your NS file" field, place the *local path* of a NS file which you have created to describe your network topology. This file will be uploaded through your browser when you choose "Submit."

After submission, the Testbed interface will begin processing your request. This will likely take several minutes, depending on how large your topology is, and what other features (such as delay nodes and bandwidth limits) you are using. Assuming all goes well, you will receive an email message indicating success or failure, and if successful, a listing of the nodes and IP address that

were allocated to your experiment.

For the NS file described above, you would receive a listing that looks similar to this:

```

Node Info:
ID              Type      OSID
-----
NodeA           pc        FBSD-STD
NodeB           pc
NodeC           pc        RHL-STD
NodeD           pc

Node Mapping:
Virtual         Physical      Qualified Name
-----
NodeA           pc18         NodeA.myexp.myproj.emulab.net
NodeB           pc29         NodeB.myexp.myproj.emulab.net
NodeC           pc28         NodeC.myexp.myproj.emulab.net
NodeD           pc35         NodeD.myexp.myproj.emulab.net
tbsdelay0       pc15         tbsdelay0.myexp.myproj.emulab.net

Lan/Link Info:
ID              Member      IP              Delay      BW (Kbs)
-----
16              NodeB:eth0   192.168.42.42   0.00       100000
15              NodeB:eth2   192.168.1.3     25.00       100000
15              NodeA:eth0   192.168.1.2     25.00       100000
16              NodeC:eth0   192.168.42.2    0.00       100000
17              NodeB:eth1   192.168.2.2     0.00       100000
17              NodeD:eth0   192.168.2.3     0.00       100000

Delay Node Info:
LinkID          Virtual      Physical      Pipe Numbers
-----
15              tbsdelay0   pc15         100,110

```

A few points should be noted:

- A single delay node was allocated and inserted into the link between NodeA and NodeB. This link is invisible from your perspective, except for the fact that it adds latency, error, or reduced bandwidth. However, the information for the delay links are included so that you can [modify the delay parameters](#) after the experiment has been created.
- Delays of less than 2ms (per trip) are too small to be accurately modeled at this time, and will be silently ignored. A delay of 0ms can be used to indicate that you do not want added delay; the two interfaces will be "directly" connected to each other. Also, please see the [Link Loss Commands](#) section in the [Extensions](#) reference.
- The names in the "Qualified Name" column refer to the control network interfaces for each of your allocated nodes. These names are added to the Emulab nameserver map on the fly, and are immediately available for you to use so that you do not have to worry about the actual physical node names that were chosen. In the names listed above, 'myproj' is the name of the project that you chose to work in, and 'myexp' is the name of the experiment that you provided in the "Begin Experiment" page.
- Since the IP address for the link from NodeB to NodeC was set in the NS file, the system

selected an appropriate IP address for the other end of the link. All of the other links were configured by the system to use 192.168.XXX.XXX subnets.

• Using your Nodes

By the time you receive the email message listing your nodes, the Testbed configuration system will have ensured that your nodes are fully configured and ready to use. If you have selected one of the Testbed supported operating system images (FreeBSD, Linux, NetBSD), this configuration process includes:

- loading fresh disk images so that each node is in a known clean state;
- rebooting each node so that it is running the OS specified in the NS script;
- configuring each of the network interfaces so that each one is "up" and talking to its virtual LAN (VLAN);
- creating user accounts for each of the project members;
- mounting the projects NFS directory in /proj so that project files are easily shared amongst all the nodes in the experiment;
- creating a /etc/hosts file on each node so that you may refer to the experimental interfaces of other nodes by name instead of IP number;
- configuring all of the delay parameters;
- configuring the tip lines so that project members may access the console ports from users.emulab.net.

As this point you may log into any of the nodes in your experiment. You will need to use Secure Shell (ssh), and you should use the 'qualified name' from the nodes mapping table so that you do not form dependencies on any particular physical node. Your login name and password will be the same as your Web Interface login and password.

The /etc/hosts file on each node will provide a local name mapping for the other nodes in your experiments. You should take care to use these names (or IP numbers) and **not** the .emulab.net names listed in the node mapping, since the emulab names refer to the control network LAN that is shared amongst all nodes in all experiments. It is only the experimental interfaces that are entirely private to your experiment.

NOTE: The configuration process just described occurs only on Emulab constructed operating system images. If you are using an OSKit kernel, or your own disk image with your own operating system, you will be responsible for all of the configuration. At some point we hope to provide tools to assist in the configuration, but for now you are on your own.

• I need root access!

If you need to customize the configuration, or perhaps reboot nodes, you can use the "sudo" command, located in /usr/local/bin on FreeBSD and Linux, and /usr/pkg/bin on NetBSD. Our policy is very liberal; you can customize the configuration in any way you like, provided it does not violate Emulab's [administrative policies](#). As an example, to reboot a node that is running FreeBSD:

```
/usr/local/bin/sudo reboot
```

• My node is wedged!

This is bound to happen when running experimental software and/or experimental operating systems. Fortunately we have an easy way for you to power cycle nodes without requiring Testbed Operations to get involved. If you must power cycle a node, log on to users.emulab.net and use the "node_reboot" command:

```
node_reboot <node> [node ... ]
```

where `node' is the physical name, as listed in the node mapping table. You may provide more than one node on the command line. Be aware that you may power cycle only nodes in projects that you are member of. Also, `node_reboot` does its very best to perform a clean reboot before resorting to cycling the power to the node. This is to prevent the damage that can occur from constant power cycling over a long period of time. For this reason, `node_reboot` may delay a minute or two if it detects that the machine is still responsive to network transmission. In any event, please try to reboot your nodes first (see above).

You may also reboot all the nodes in an experiment by using the `-e` option to specify the project and experiment names. For example:

```
node_reboot -e testbed,multicast
```

will reboot all of the nodes reserved in the "multicast" experiment in the "testbed" project. This option is provided as a shorthand method for rebooting large groups of nodes.

• I've scrogged my disk!

Scrogging your disk is certainly not as common, but it does happen. You can either terminate your experiment, and recreate it (which will allocate another group of nodes), or if you prefer you can reload the disk image yourself. You will of course lose anything you have stored on that disk; it is a good idea to store only data that can be easily recreated, or else store it in your project directory in `/proj`. Reloading your disk with a fresh copy of the default image is easy, and requires no intervention by Emulab staff:

```
os_load <node> [node ... ]
```

`os_load` will wait (not exit) until the nodes have been reloaded, so that you do not need to check the console lines of each node to determine when the load is done.

• I've finished my experiment

When your experiment is completed, and you no longer need the resources that have been allocated to it, you will need to terminate the experiment via the Emulab Web Interface. Click on the "End An Experiment" link. You will be presented with a list of all of the experiments in all of the projects for which you have the authorization to terminate experiments. Select the experiment you want to terminate by clicking on the button in the "Terminate" column on the right hand side. You will be asked to **confirm** your choice. The Testbed configuration system will then tear down your experiment, and send you an email message when the process is complete. At this point you are allowed to reuse the experiment name (say, if you wanted to create a similar experiment with different parameters).

• Getting Help!

If you have any questions or problems, or just want to comment on Emulab's operation (maybe you want to suggest an improvement to one of the Web pages), feel free to contact us by sending email to [Testbed Operations](#). Also note that much of the software is in development, and occasionally things might break or not work as you expect. Again, please feel free to contact us.

Advanced Topics

- [A more advanced example](#)
- [Installing RPMS automatically](#)
- [Starting your application automatically](#)
- [How do I know when all my nodes are ready?](#)
- [Customizing an OS \(How to create a *delta*\)](#)
- [Setting up IP routing between nodes](#)

- **A more advanced example**

We have a more [advanced example](#) demonstrating the use of RED queues, traffic generators, and the event system.

- **Installing RPMS automatically**

The Testbed NS extension `tb-set-node-rpms` allows you to specify a (space separated) list of RPMs to install on each of your nodes when it boots:

```
tb-set-node-rpms $nodeA /proj/pid/rpms/silly-freebsd.rpm
tb-set-node-rpms $nodeB /proj/pid/rpms/silly-linux.rpm
```

The above NS code says to install the `silly-freebsd.rpm` file on `nodeA`, and the `silly-linux.rpm` on `nodeB`. RPMs are installed as root when the node first boots, and must reside on the node's local filesystem, or in a directory that can be reached via NFS. This is either the project's `/proj` directory, or a project member's home directory in `/users`.

- **Starting your application automatically**

You can start your application automatically when your nodes boot by using the `tb-set-node-startup` NS extension. The argument is the pathname of a script or program that is run as the `UID` of the experiment creator, after the node has reached multiuser mode. You can specify the same program for each node, or a different program. For example:

```
tb-set-node-startup $nodeA /proj/pid/runme.nodeA
tb-set-node-startup $nodeB /proj/pid/runme.nodeB
```

will run `/proj/pid/runme.nodeA` on `nodeA` and `/proj/pid/runme.nodeB` on `nodeB`. The programs must reside on the node's local filesystem, or in a directory that can be reached via NFS. This is either the project's `/proj` directory, or a project member's home directory in `/users`.

The exit value of the startup command is reported back to the Web Interface, and is made available to you via the "Experiment Information" link. There is a listing for all of the nodes in the

experiment, and the exit value is recorded in this listing. The special symbol `none` indicates that the node is still running the startup command. A log file containing the output of the startup command is created in the project's `logs` directory (`/proj/pid/logs`).

The startup command is especially useful when combined with *batch mode* experiments.

• How do I know when all my nodes are ready?

It is often necessary for your startup program to determine when all of the other nodes in the experiment have started, and are ready to proceed. Sometimes called a *barrier*, this allows programs to wait at a specific point, and then all proceed at once. Emulab provides a primitive form of this mechanism using experiment *ready bits*, which are set and read using the [TMCD/TMCC](#). When an experiment is first configured, the ready bit for each node is cleared. As each node starts its application and reaches the point where it must be sure that all other nodes have started up, it issues a `TMCC ready` command:

```
tmcc ready
```

which tells Emulab's configuration system that the node is ready to proceed. The node can then poll for the *ready count* to determine how many nodes are ready (have issued a `tmcc ready` command):

```
tmcc readycount
```

which will return the ready count as a string:

```
READY=N TOTAL=M
```

where `N` is the number of nodes that are ready, and `M` is the total number of nodes in the experiment. An application can poll the ready count with a simple script, or it can encode the ready bits check directly into its program. For example, here is a simple Perl fragment that issues the ready command, and then polls for the ready count, being sure to delay a small amount between each poll.

```
system("tmcc ready");
while (1) {
    my $bits = `tmcc readycount`;
    if ($bits =~ /READY=(\d*) TOTAL=(\d*)/) {
        if ($1 == $2) {
            last;
        }
    }
    #
    # Please sleep to avoid swamping the TMCD!
    #
    sleep(5);
}
```

Note that the ready count is essentially a use-once feature; The ready count cannot be reinitialized to zero since there is no actual synchronization happening. If in the future it appears that a generalized barrier synchronization would be more useful, we will investigate the implementation of such a feature.

- **Customizing an OS (How to create a *delta*)**

If your set of operating system customizations cannot be easily contained within an RPM (or multiple RPMs), or if you are just not familiar with the RPM mechanism, then you can create your own operating system *delta*. A delta is like an RPM or Tar file in that it contains a bunch of files to be unpacked onto the node. The difference is that with a delta you do not have to figure what files you changed, and how to automate the installation process. Instead, you just allocate a node, change it anyway you like, and then issue the `create-delta` command. The resulting delta file can then be specified in your NS file using the Testbed NS extension `tb-set-node-deltas`. You can create one delta to install on all of your nodes, or several different deltas for various nodes in your experiment. When the nodes in your new experiment boot for the first time, the delta will be installed (all of the files unpacked) very early in the boot phase, and the node rebooted again (in case you have installed daemons that need to be started during initialization). Your experiment can then proceed.

The key point is that the Testbed configuration software deals with figuring out what files you changed, installing the delta on your nodes, rebooting the nodes that have new software installed, and ensuring that any particular delta is installed only once on each node.

Lets step through an example. The first thing you need to do is create an experiment with a single node in it. The following NS file can be submitted to the "Begin Experiment" page.

```
set ns [new Simulator]
source tb_compat.tcl
set nodeA [$ns node]
tb-set-node-os $nodeA FBSD-STD
$ns run
```

When you have received email notification that the experiment has configured, log into the node with `ssh`. Install whatever software you like, making sure to update the necessary files if you have installed daemons that need to be started automatically at boot time. After all of your software is installed, create the delta file with:

```
sudo /usr/local/bin/create-delta /proj/testbed/foo.delta
```

The argument to the `create-delta` command is a complete pathname, which **must** reside someplace in your `/proj` directory (a subdirectory is fine). You cannot write the delta file to any other filesystem. This restriction is enforced so that disk space (and resources in general) can be accounted for on a per-project basis. It should be noted that a delta created on one OS cannot be installed on another. In other words, a delta created on a FreeBSD machine can only be installed on a FreeBSD machine. If you need the same software installed on a Linux machine as well, you will need to repeat this process with a node running Linux. See the section on [tb-set-node-os](#) in the [Extensions](#) reference.

After you have created your delta, you can then use it in subsequent experiments by using the Testbed NS extension `tb-set-node-deltas`. For example, here is an NS file that creates a two node experiment, installs a different delta on each node, and then runs a program automatically on each node. Presumably, the startup program is installed by the delta, and encapsulates the experiment being performed.

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

```

source tb_compat.tcl
set nodeA [$ns node]
set nodeB [$ns node]
tb-set-node-os $nodeA FBSD-STD
tb-set-node-os $nodeB RHL-STD
tb-set-node-deltas $nodeA /proj/testbed/deltas/silly-freebsd.delta
tb-set-node-deltas $nodeB /proj/testbed/deltas/silly-linux.delta
tb-set-node-startup $nodeA /usr/site/bin/run-my-experiment
tb-set-node-startup $nodeB /usr/site/bin/run-my-experiment
$ns run

```

Implementation Notes:

- o Deltas are created and installed with the unix filesystem backup utilities `dump` and `restore`.
- o Beware of changing too many critical systems and/or too many changes to the `/etc/rc` scripts.
- o If you find that your customizations are too much for the Delta mechanism, feel free to contact us so that we can arrange to create a complete snapshot of your system.

• **Setting up IP routing between nodes**

As Emulab strives to make all aspects of the network controllable by the user, we do not attempt to impose any IP routing architecture or protocol by default. However, many users are more interested in end-to-end aspects and don't want to be bothered with setting up routes. For those users we provide an option to automatically set up routes on nodes which run one of our provided FreeBSD or Linux disk images.

You can use the NS `rtproto` syntax in your NS file to enable routing:

```
$ns rtproto protocol
```

where the *protocol* option is limited to one of `Session`, `Static`, or `Manual`. `Session` routing provides fully automated routing support, and is implemented by enabling `gated` running the OSPF protocol on all nodes in the experiment. `Static` routing also provides automatic routing support, but rather than computing the routes dynamically, the routes are precomputed when the experiment is created, and then loaded on each node when it boots.

`Manual` routing allows you to explicitly specify per-node routing information in the NS file. To do this, use the `Manual` routing option to `rtproto`, followed by a list of routes using the `add-route` command:

```
$node add-route $dst $nexthop
```

where the `dst` can be either a node, a link, or a LAN. For example:

```

$client add-route $server $router
$client add-route [$ns link $server $router] $router
$client add-route $serverlan $router

```

Note that you would need a separate `add-route` command to establish a route for the reverse direction; thus allowing you to specify differing forward and reverse routes if so desired. These statements are converted into appropriate `route(8)` commands on your experimental nodes when they boot.

In the above examples, the first form says to set up a manual route between

\$client and \$server, using \$router as the nexthop; \$client and \$router should be directly connected, and the interface on \$server should be unambiguous; either directly connected to the router, or an edge node that has just a single interface.

If the destination has multiple interfaces configured, and it is not connected directly to the nexthop, the interface that you are intending to route to is ambiguous. In the topology shown to the right, \$nodeD has two interfaces configured. If you attempted to set up a route like this:

```
$nodeA add-route $nodeD $nodeB
```

you would receive an error since it cannot be determined (easily, with little programmer effort, by Emulab staff!) which of the two links on \$nodeD you are referring to. Fortunately, there is an easy solution, courtesy of an Emulab extension. Instead of a node, specify the link directly:

```
$nodeA add-route [$ns link $nodeD $nodeC] $nodeB
```

This tells us exactly which link you mean, enabling us to convert that information into a proper route command on \$nodeA.

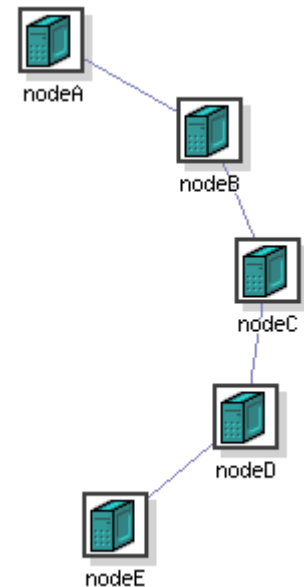
The last form of add-route command is used when adding a route to an entire LAN. It would be tedious and error prone to specify a route to each node in a LAN by hand. Instead, just route to the entire network:

```
set clientlan [$ns make-lan "$nodeE $nodeF $nodeG" 100Mb 0ms]
$nodeA add-route $clientlan $nodeB
```

While all this manual routing infrastructure sounds really nifty, its probably a good idea to use Session routing for all but small, simple topologies. Explicitly setting up all the routes in even a moderately-sized experiment is extremely error prone. Consider this: a recently created experiment with 17 nodes and 10 subnets required 140 hand-created routes in the NS file. Yow!

Two final, cautionary notes on routing:

- o You might be tempted to set the default route on your nodes to reduce the number of explicit routes used. **Don't do it.** That would prevent nodes from contacting the outside world, i.e., you. The default route *must* be set to use the control network interface.
- o If you use your own routing daemon, you must avoid using the control network interface in the configuration. Since every node in the testbed is directly connected to the control network LAN, a naive routing daemon configuration will discover that any node is just one hop away, via the control network, from any other node and *all* inter-node traffic will be routed via that interface.



Batch Mode

- [Batch Mode Introduction](#)
- [A Batch Mode Example](#)

• Batch Mode Introduction

Batch Mode experiments can be created on the Testbed via the "Create an Experiment" link in the operations menu to your left. There is a checkbox near the bottom of the form that indicates you want to use the batch system. There are several important differences between a regular experiment and a batch mode experiment:

- o The experiment is run when enough resources (ie: nodes) are available. This might be immediately, or it might be sometime in the future.
- o Once your NS file is handed off to the system, the batch system is responsible for setting up the experiment and tearing it down once the experiment has completed. You will receive email notifying you when the experiment has been scheduled and when it has been terminated.
- o Your NS file must define a *startup* command to run on each node using the [tb-set-node-startup](#) NS extension. It is the exit value(s) of the startup command(s) that indicates that the experiment is completed; when all of the nodes have run their respective startup commands and exited, the batch system will then tear down the experiment. The output of the startup command is stored in a file in your home directory so you can follow what has happened.

• A Batch Mode Example

Consider example NS file [batch.ns](#). First off, we have to arrange for the experimental software to be automatically installed when the nodes boot. This is done with the [tb-set-node-rpms](#) NS extension:

```
tb-set-node-rpms $nodeA /proj/testbed/rpms/silly-1.0-1.i386-freebsd.rpm
tb-set-node-rpms $nodeB /proj/testbed/rpms/silly-1.0-1.i386-freebsd.rpm
```

The next two lines of the NS file specify what program should be run on each of the nodes. Using the [tb-set-node-startup](#) NS extension, we say that the program run-silly (installed by the silly-1.0 RPM) is to be run on both nodes:

```
tb-set-node-startup $nodeA /usr/site/bin/run-silly
tb-set-node-startup $nodeB /usr/site/bin/run-silly
```

After you have been notified via email that the batch experiment is running, you can track the progress of your experiment by looking in the "Experiment Information" page. As each node completes the startup command, the listing for that node will be updated to reflect the exit status of the command (you may need to hit the Reload button to see the changes). Once all of the nodes have reported in an exit status, the batch system will tear down the experiment and send you email. If your experiment is such that one node is the controller, and runs commands on all the other nodes, then simply run a dummy startup command on the other nodes so that the batch system will receive an exit value for that node. Since the batch is not terminated until *all* nodes have reported in, be sure that the controlling node does not exit from its startup command until all of the nodes have finished. A dummy startup command can be setup

like this:

```
tb-set-node-startup $nodeC /bin/echo
```

The status of your batch experiment can be viewed via the "Experiment Information" link in the Web Interface Options menu. You may also cancel a batch after you have submitted it using the "Terminate" option in the information display. As noted in the section on the [Startupcmd](#), the output of the startup command on each node is written to separate files in your project log directory. You can use these log files to debug your batch experiment.

The batch system is still under development. It appears to be functional, but there are bound to be kinks in the system. Please help us debug and improve it by letting us know what you think and if you have problems with it. Currently, the batch system tries every 10 minutes to run your batch. It will send you email every 5 or so attempts to let you know that it is trying, but that resources are not available. It is a good idea to glance at the message to make sure that the problem is lack of resources and not an error in your NS file.

Custom OS Images

Sorry, this section of the documentation is still under construction. In the meantime, please check the documentation for creating your own custom disk images at https://www.emulab.net/newimageid_explain.php3.

Emulab Tutorial - A More Advanced Example

Here is a slightly more complex example demonstrating the use of RED queues, traffic generation, and the event system. Where possible, we adhere to the syntax and operational model of [ns-2](#), as described in the [NS manual](#).

- **RED/GRED Queues:** In addition to normal DropTail links, Emulab supports the specification of the RED and GRED (Gentle RED) links in your NS file. RED/GRED queuing is handled via the insertion of a traffic shaping delay node, in much the same way that bandwidth, delay, and packet loss is handled. For a better understanding of how we support traffic shaping, see the `ipfw` and `dummysnet` man pages on `users.emulab.net`. It is important to note that Emulab supports a smaller set of tunable parameters than NS does; please read the aforementioned manual pages!
- **Traffic Generation:** Emulab supports Constant Bit Rate (CBR) traffic generation, in conjunction with either Agent/UDP or Agent/TCP agents. We currently use the [TG Tool Set](#) to generate traffic (usermode programs running on FreeBSD 4.3 endpoints).
- **Traffic Generation using NS Emulation (NSE):** Emulab supports TCP traffic generation using NS's Agent/TCP/FullTcp which is a BSD Reno derivative and its subclasses namely Newreno, Tahoe and Sack. Currently two application classes are supported: Application/FTP and Application/Telnet. The former drives the FullTcp agent to send bulk-data according to connection dynamics. The latter uses the NS's [tcplib](#) telnet distribution for telnet-like data. For configuration parameters and commands allowed on the objects, refer to NS documentation [here](#).
- **Event System:** Emulab supports limited use of the NS *at* syntax, allowing you to define a static set of events in your NS file, to be delivered to agents running on your nodes. There is also "dynamic events" support that can be used to inject events into the system on the fly, say from a script running on `users.emulab.net`.
- **Program Objects:** Emulab has added extensions that allow you to run arbitrary programs on your nodes, starting and stopping them at any point during your experiment run.

What follows is a [sample NS file](#) that demonstrates the above features, with annotations where appropriate. First we define the 2 nodes in the topology:

```
set nodeA [$ns node]
set nodeB [$ns node]
```

Next define a duplex link between nodes nodeA and nodeB. Instead of a standard DropTail link, it is declared to be a Random Early Detection (RED) link. While this is obviously contrived, it allows us to ignore [routing](#) issues within this example.

```
set link0 [$ns duplex-link $nodeA $nodeB 100Mb 0ms RED]
```

Each link is has an NS "Queue" object associated with it, which you can modify to suit your needs. (*currently, there is a single queue object per duplex link; you can cannot set the parameters asymmetrically*). The following parameters can be changed, and are defined in the NS manual (see Section 7.3):

```
set queue0 [[ $ns link $nodeA $nodeB ] queue]
$queue0 set gentle_ 0
$queue0 set queue-in-bytes_ 0
$queue0 set limit_ 75
$queue0 set maxthresh_ 20
$queue0 set thresh_ 7
```

```
$queue0 set linterm_ 11
$queue0 set q_weight_ 0.004
```

A UDP agent is created and attached to nodeA, then a CBR traffic generator application is created, and attached to the UDP agent:

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

set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

A TCP agent is created and also attached to nodeA, then a second CBR traffic generator application is created, and attached to the TCP agent:

```
set tcp0 [new Agent/TCP]
$ns attach-agent $nodeA $tcp0

set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 attach-agent $tcp0
```

You must define traffic sinks for each of the traffic generators created above. The sinks are attached to nodeB:

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

set null1 [new Agent/TCPsink]
$ns attach-agent $nodeB $null1
```

Then you must connect the traffic generators on nodeA to the traffic sinks on nodeB:

```
$ns connect $udp0 $null0
$ns connect $tcp0 $null1
```

Here is a good example for NSE FullTcp traffic generation. The following code snippet attaches an FTP agent that drives a Reno FullTcp on NodeA:

```
set tcpfull0 [new Agent/TCP/FullTcp]
$ns attach-agent $nodeA $tcpfull0

set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcpfull0
```

You must then define the sink FullTcp endpoint and call the method "listen" making this agent wait for an incoming connection:

```
set tcpfull1 [new Agent/TCP/FullTcp/Sack]
$tcpfull1 listen
$ns attach-agent $nodeB $tcpfull1
```

Like all other source-sink traffic generators, you need to connect them:

```
$ns connect $tcpfull0 $tcpfull1
```

Lastly, a set of events to control your applications and link characteristics:

```
$ns at 60.0 "$cbr0 start"
$ns at 70.0 "$link0 bandwidth 10Mb duplex"
$ns at 80.0 "$link0 delay 10ms"
$ns at 90.0 "$link0 plr 0.05"
$ns at 100.0 "$link0 down"
$ns at 110.0 "$link0 up"
$ns at 115.0 "$cbr0 stop"

$ns at 120.0 "$ftp0 start"
$ns at 140.0 "$tcpfull0 set segsize_ 256; $tcpfull0 set segsperack_ 2"
$ns at 145.0 "$tcpfull1 set nodelay_ true"
$ns at 150.0 "$ftp0 stop"

$ns at 120.0 "$cbr1 start"
$ns at 130.0 "$cbr1 set packetSize_ 512"
$ns at 130.0 "$cbr1 set interval_ 0.01"
$ns at 140.0 "$link0 down"
$ns at 150.0 "$cbr1 stop"
```

When you receive email containing the experiment setup information (as described in [Beginning an Experiment](#)), you will notice an additional section that lists all of the events that will be delivered during your experiment:

Event List:					
Time	Node	Agent	Type	Event	Arguments
60.000	nodeA	cbr0	TRAFGEN	START	PACKETSIZE=500 RATE=100000 INTERVAL=0.005
70.000	tbsdelay0	link0	LINK	MODIFY	BANDWIDTH=10000
80.000	tbsdelay0	link0	LINK	MODIFY	DELAY=10ms
90.000	tbsdelay0	link0	LINK	MODIFY	PLR=0.05
100.000	tbsdelay0	link0	LINK	DOWN	
110.000	tbsdelay0	link0	LINK	UP	
115.000	nodeA	cbr0	TRAFGEN	STOP	
120.000	nodeA	cbr1	TRAFGEN	START	PACKETSIZE=500 RATE=100000 INTERVAL=0.005
120.000	nodeA	ftp0	TRAFGEN	MODIFY	\$ftp0 start
130.000	nodeA	cbr1	TRAFGEN	MODIFY	PACKETSIZE=512
130.000	nodeA	cbr1	TRAFGEN	MODIFY	INTERVAL=0.01
140.000	tbsdelay0	link0	LINK	DOWN	
140.000	nodeA	tcpfull0	TRAFGEN	MODIFY	\$tcpfull0 set segsize_
140.000	nodeA	tcpfull0	TRAFGEN	MODIFY	\$tcpfull0 set segspera
145.000	nodeB	tcpfull1	TRAFGEN	MODIFY	\$tcpfull1 set nodelay_
150.000	tbsdelay0	link0	LINK	UP	
150.000	nodeA	ftp0	TRAFGEN	MODIFY	\$ftp0 stop
160.000	nodeA	cbr1	TRAFGEN	STOP	

The above list represents the set of events for your experiments, and are stored in the Emulab Database.

When your experiment is swapped in, an *event scheduler* is started that will process the list, and send them at the time offset specified. In order to make sure that all of the nodes are actually rebooted and ready, time does not start ticking until all of the nodes have reported to the event system that they are ready. At present, events are restricted to system level agents (Emulab traffic generators and delay nodes), but in the future we expect to provide an API that will allow experimentors to write their own event agents.

Dynamic Scheduling of Events

NS scripts give you the ability to schedule events dynamically; an NS script is just a TCL program and the argument to the "at" command is any valid TCL expression. This gives you great flexibility in a simulated world, but alas, this cannot be supported in a practical manner in the real world. Instead, we provide a way for you to inject events into the system dynamically, but leave it up to you to script those events in whatever manner you are most comfortable with, be it a PERL script, or a shell script, or even another TCL script! Dynamic event injection is accomplished via the *Testbed Event Client* (tevc), which is installed on your experimental nodes and on `users.emulab.net`. The command line syntax for `tevc` is:

```
tevc -e pid/eid time objname event [args ...]
```

where the `time` parameter is one of:

- now
- +seconds (floating point or integer)
- [[[[yy]mm]dd]HH]MMss

For example, you could issue this sequence of events.

```
tevc -e testbed/myexp now cbr0 set interval_=0.2
tevc -e testbed/myexp +10 cbr0 start
tevc -e testbed/myexp +15 link0 down
tevc -e testbed/myexp +17 link0 up
tevc -e testbed/myexp +20 cbr0 stop
```

Some points worth mentioning:

- There is no "global" clock; Emulab nodes are kept in sync with NTP, which does a very good job of keeping all of the clocks within 1ms of each other.
- The times "now" and "+seconds" are relative to the time at which each event is submitted, not to each other or the start of the experiment.
- The set of events you can send is currently limited to control of traffic generators and delay nodes. We expect to add more agents in the future.
- Sending dynamic events that intermix with statically scheduled events can result in unpredictable behavior if you are not careful.
- Currently, the event list is replayed each time the experiment is swapped in. This is almost certainly not the behaviour people expect; we plan to change that very soon.
- `tevc` does not provide any feedback; if you specify an object (say, `cbr78` or `link45`) that is not a valid object in your experiment, the event is silently thrown away. Further, if you specify an operation or parameter that is not appropriate (say, "link0 start" instead of "link0 up"), the event is silently dropped. We expect to add error feedback in the future.

Supported Events

This is a (mostly) comprehensive list of events that you can specify, either in your NS file or as a dynamic event on the command line. In the listings below, the use of "link0", "cbr0", etc. are included to clarify the syntax; the actual object names will depend on your NS file. Also note that when sending events from the command line with `tevc`, you should not include the dollar (\$) sign. For example:

```
NS File: $ns at 3.0 "$link0 down"
tevc:    tevc -e pid/eid +3.0 link0 down
```

- **Links:**

```
In "ns" script:
$link0 bandwidth 10Mb duplex
$link0 delay 10ms
$link0 plr 0.05
```

```
With "tevc":
tevc ... link0 modify bandwidth=20000      # In kbits/second; 20000 = 2
tevc ... link0 modify delay=10ms          # In msec (the "ms" is ignc
tevc ... link0 modify plr=0.1
```

```
Both:
$link0 up
$link0 down
```

- **Queues:** Queues are special. In your NS file you modify the actual queue, while on the command line you use the link to which the queue belongs.

```
$queue0 set queue-in-bytes_ 0
$queue0 set limit_ 75
$queue0 set maxthresh_ 20
$queue0 set thresh_ 7
$queue0 set linterm_ 11
$queue0 set q_weight_ 0.004
```

- **CBR**

```
$cbr0 start
$cbr0 set packetSize_ 512
$cbr0 set interval_ 0.01
$cbr0 set rate_ 10Mbps
$cbr0 stop
```

- **FullTcp, FTP and Telnet:** Refer to the NS documentation [here](#).

Program Objects

We have added some extensions that allow you to use NS's `at` syntax to invoke arbitrary commands on your experimental nodes. Once you define a program object and initialize its command line and the node

on which the command should be run, you can schedule the command to be started and stopped with NS at statements. To define a program object:

```
set prog0 [new Program $ns]
$prog0 set node $nodeA
$prog0 set command "/bin/ls -lt >& /users/joe/logs/prog0"

set prog1 [new Program $ns]
$prog1 set node $nodeB
$prog1 set command "/bin/sleep 60 >& /tmp/sleep.debug"
```

Then in your NS file a set of static events to run these commands:

```
$ns at 10 "$prog0 start"
$ns at 20 "$prog1 start"
$ns at 30 "$prog1 stop"
```

If you want to schedule starts and stops using dynamic events:

```
tevc -e testbed/myexp now prog0 start "command=/bin/ls /tmp"
tevc -e testbed/myexp now prog1 start "command=sleep 60"
tevc -e testbed/myexp +20 prog1 stop
```

Some points worth mentioning:

- A program must be "stopped" before it is started; if the program is currently running on the node, the start event will be silently ignored.
- The command line is passed to /bin/csh; any valid csh expression is allowed, although no syntax checking is done prior to invoking it. If the syntax is bad, the command will fail. It is a good idea to redirect output to a log file so you can track failures.
- The "stop" command is implemented by sending a SIGTERM to the process group leader (the csh process). If the SIGTERM fails, a SIGKILL is sent.
- When issuing dynamic events using `tevc`, you must specify the command line to invoke. This is intended to make program objects more flexible.

Frequently Asked Questions

Contents

- [Getting Started](#)
 - [Who is Eligible to use Emulab.Net?](#)
 - [How do I start a project?](#)
 - [How do I join a project?](#)
 - [I have an Emulab account. Now what?](#)
 - [Do I need to change my PATH variable?](#)
 - [Can I be in more than one project?](#)
 - [Can I change my Emulab password?](#)
 - [I'm a project leader. Can I designate TAs?](#)
 - [Where do I get help?](#)
- [Using the Testbed](#)
 - [Is there a tutorial?](#)
 - [Do you have a GUI to help me create experiments?](#)
 - [How many nodes can I ask for?](#)
 - [How long can I keep using my nodes?](#)
 - [What if I need more nodes than are free?](#)
 - [Do I get root access on my nodes?](#)
 - [Do my nodes have consoles I can look at?](#)
 - [How do I connect directly to node consoles, without going through users?](#)
 - [Can I reboot \(power cycle\) my nodes?](#)
 - [I've scrogged my disk! Now what?](#)
 - [Where do I store files needed by my experiment?](#)
 - [Are my files on **users.emulab.net** backed up \(filesaved\)?](#)
 - [Are the nodes in my experiment backed up \(filesaved\)?](#)
 - [What is Swapping?](#)
 - [How can I get switch statistics \(such as packet counts\) for my experiment?](#)
- [Hardware setup](#)
 - [How many nodes are there?](#)
 - [How many nodes are currently available?](#)
 - [How many ethernet cards are on each node?](#)
 - [How many nodes are currently available \(free\)?](#)
 - [Can I do traffic shaping on my links?](#)
 - [Can I modify the traffic shaping parameters on my links?](#)
 - [Are there other traffic shaping parameters besides latency, bandwidth, and PLR?](#)
 - [I asked for traffic shaping, but everything seems to be going at full LAN speeds. What's wrong?](#)
- [Software setup](#)
 - [What OS do the nodes run?](#)
 - [How do I select which OS to run on each node?](#)
 - [Can I load my own software \(RPMs\) on my nodes?](#)
 - [Can I schedule programs to run automatically when a node boots?](#)
 - [How can I turn on routing or set up routes automatically in my nodes?](#)
 - [How does my software determine when other nodes in my experiment are ready?](#)
 - [Can I run my own Operating System?](#)
 - [What if I need more disk space on my nodes?](#)
 - [Are there testbed-specific daemons that could interfere with my experiment?](#)

- [Security Issues](#)
 - [Is Emulab Firewalled?](#)
 - [Troubleshooting](#)
 - [My experiment is setup, but I cannot send packets between some of the nodes, why?](#)
-

Getting Started

- **Who is Eligible to use Emulab.Net?**

Emulab.Net is an NSF/DARPA sponsored project, with additional support from these [sponsors](#). As such, eligibility to use Emulab is primarily granted to other NSF/DARPA sponsored projects, as well as current university research projects. There are exceptions of course. If you are unsure about your eligibility to use Emulab, please feel free to send us an email inquiry.

- **How do I start a project?**

If you are new to the Testbed, simply click on the "Start Project" link on the Emulab [Home Page](#). You will need to fill in the forms with your personal information and information about the project. Then click on the "Submit" button. Within a few days you will be contacted via email with an approval message. More information about starting projects can be found in [Authorization Page](#).

If you already have an Emulab account, and wish to start a second project, first log into the Web Interface. Then select the "Start Project" link; all of the personal information will already be filled in. You will need to complete just the project information section.

- **How do I join a project?**

If you are new to the Testbed, simply click on the "Join Project" link on the Emulab [Home Page](#). You will need to fill in the form with your personal information, and provide the name of the project you are trying to join (typically, the *Project Leader* will have told you the name of the project). Then click on the "Submit" button, and wait for the project leader to approve you. When approved you will receive an email message saying so, and you can then log into the Testbed.

- **I have an Emulab account. Now what?**

Once you have been approved to start (or join) your first project, you will be able to log into Emulab's user machine, **users.emulab.net**. We require that all Emulab users use ssh. For example, if your Emulab account name is "joe", then you would do:

```
ssh users.emulab.net -l joe
```

Your password starts out the same as the password you initially supplied to the Start (or Join) web page.

- **Do I need any special directories in my PATH variable?**

There are several useful (although not required) programs installed on `users.emulab.net` in `/usr/testbed/bin`. You should edit your dot files to include that directory in your search path.

- **Can I be in more than one project?**

Yes. You may join (and/or start) as many projects as you like, subject to Emulab [administrative policies](#).

- **Can I change my Emulab password?**

Yes. You can change your Emulab Web password and your Emulab login password (the password you use to log into **users.emulab.net**, as well as nodes in your experiments). To change your Web password, simply click on the "Update User Information" in the menu to your left, and then enter your new password in the location provided. To change your login password, use the unix `passwd` utility when logged into **users.emulab.net**.

- **I'm a project leader. Can I designate TAs?**

Yes. To designate a TA, you must first create a project *group*. A project group is a lot like a unix group, and in fact unix groups is the mechanism used to protect members of one group from members of another group. When you create a group, you designate a *group leader* who is responsible for approving users who apply to join the group. Group leaders may also terminate experiments that have been created by members of the group. As Project Leader, you may also shift members of your project in and out of your project's groups as you like, and you are automatically a member of all groups within your project. As a convenience, all new projects are created with one new group, termed the *default group*. As its name implies, whenever the group is left unspecified in a form, it defaults to the project group (this allows you to create a project without any sub groups at all; new members join the default group, new experiments are created in the default group, etc.).

Project groups are created via the Project Information link at your left. Simply go to the project page in which you want to create a group, and look for the "Create New Group" link. More information on project groups is available via the [Emulab Documentation](#) page in the [Groups Tutorial](#).

- **Where do I get help?**

If you cannot find an answer to your question in the [Emulab Documentation](#), then you can send us an [email message](#). We will try to answer your question as quickly as we can.

Using the Testbed

- **Is there a tutorial?**

Yes, we have an extensive [tutorial](#) on using the Testbed.

- **Do you have a GUI to help me create experiments?**

Yes, we provide a GUI that gives you an easy to use drawing palette on which you can place nodes, lans, and links. Testbed specific attributes such as operating system, hardware type, and link/lan characteristics, may be attached to each object. With a single click, you can instantiate your new topology on the Testbed as an experiment in one of your projects. Alternatively, you can save the auto-generated NS file on your machine, edit as required, and then submit it later when creating an experiment.

To access the GUI, please log in and go to the Begin Experiment page. *Note: you need a Java compliant browser.*

- **How many nodes can I ask for?**

You can ask for as many nodes as are currently available! You can click on the "Node Reservation Status" link at your left to see how many nodes are currently free. If you ask for more than are currently available, your experiment will be rejected (you will receive email notification shortly after you submit your NS file to the web interface).

We urge all new Emulab users to begin with a small 3-4 node experiment so that you will become familiar with NS syntax and the practical aspects of Emulab operation.

- **How long can I keep using my nodes?**

You can keep them as long as you need them, subject to our [Node Usage Policies](#). In general, you should do your work, and then terminate your experiment as soon as you're done with it. If you're not done with it, but are through for a while, you should probably "swap out" your experiment (See the question [What is Swapping](#) in this FAQ). It is especially important to swap out your experiment if you're through with it for the weekend. Emulab usually gets heavy use on the weekends by users who need to make very large experiments, so it is important to leave as many nodes available as possible.

- **What if I need more nodes than are free?**

For example, say you need 50 nodes but there are only 40 free. In general, getting this many nodes is going to require intervention from Testbed Operations, if only so we can ask other experimenters to free up nodes, if possible. Please send us email if you are not able to get the number of nodes you need for your experiment.

Another alternative is to use the [Batch System](#). If your experiment is amenable to being batched (does not require human intervention to start and stop), then you can submit a batch request, which will be serviced when enough nodes become available. Typically, you would start out with a few nodes, getting used to the batch system and creating whatever scripts are needed to make the experiment batchable. Then scale up to larger numbers of nodes. That's the easiest way of getting a lot of nodes!

- **Do I get root access on my nodes?**

Yes. Project leaders get root access to all of the nodes in all of the experiments that are running in their project. Project members get root if their project leader grants them root access, when the leader approves the group [membership request](#). Root privileges are granted via the `sudo`

command. The [tutorial](#) describes this in more detail.

- **Do my nodes have consoles I can look at?**

Yes. Each of the PCs has its own serial console line that you can interact with using the unix `tip` utility. To "tip" to "pc1" in your experiment, ssh into **users.emulab.net**, and then type `tip pc1` at the unix prompt. You may then interact with the serial console. The console output is also saved for each node so that you may look at it later. For each node, the console log is stored as `/var/log/tiplogs/pcXXX.run`. This *run* file is created when nodes are first allocated to an experiment, and the unix permissions of the run files permit just members of the project to view them. When the nodes are deallocated, the run files are cleared, so if you want to save them, you must do so before terminating the experiment.

The Sharks also have serial console lines, but because of the limited number of serial ports available on **users.emulab.net**, only one Shark, the last or "eighth", on each shelf has a console line attached. To tip to that shark, you would type `tip shXX` at the unix prompt, where "XX" is the shark shelf number. The shark shelf number is the first digit in the name. Using shark sh16-8 as an example, the shelf number is sixteen, and the number of the node on the shelf is eight.

- **How do I connect directly to node consoles, without going through users?**

Clicking "Connect to Serial Line" in the Node Options page will send your browser a "text/x-testbed-acl" file. If you have downloaded `tiptunnel` and set it as the handler for that MIME type, `tiptunnel` will launch a new telnet running in a new xterm (this may take a few seconds.) That telnet will be connected to a local port, which is tunneled through SSL to your node's console. Closing the xterm, exiting telnet, or killing `tiptunnel` itself will end the connection.

You can download the `tiptunnel` statically-linked x86 [binary for FreeBSD here](#). You can download the `tiptunnel` statically-linked x86 [binary for Linux here](#). Use `gunzip`, then `tar xvf` on the downloaded file. Move the resulting `tiptunnel` binary into `/usr/local/bin` or another directory of your choice.

Then, when you click on the "Connect to Serial Line" link, tell your browser to always use that binary to open files of type "text/x-testbed-acl".

The source tarball, as well as a binary for Windows, will be available soon.

- **Can I reboot (power cycle) my nodes?**

Yes. Each of the PCs is independently power controlled. If your node becomes wedged, or otherwise unresponsive, you can use the `node_reboot` command, as discussed in the [Emulab Tutorial](#).

The Sharks are also power controlled, but because of the limited number of power ports available, the entire shelf of 8 sharks is on a single controller. The `node_reboot` does its best to cleanly reboot individual sharks, but if a single shark is unresponsive, the entire shelf will be power cycled.

- **I've scrogged my disk! Now what?**

If you manage to corrupt a disk (or slice), no worries. You can easily repair the damage yourself by reloading a fresh copy of the default disk image. You will of course lose anything you have stored on that disk; it is a good idea to store only data that can be easily recreated, or else store it in your project directory in `/proj`. Disk reloading is covered in more detail in the [Emulab Tutorial](#).

- **Where do I store files needed by my experiment?**

Each project has its own directory, rooted at `/proj`, which is available via NFS to all of the nodes in experiments running in that project. For example, when the "RON" project was created, a directory called `/proj/RON` was also created. This directory is owned by the project creator, and is in the unix group "RON." Its permission (mode) is 770; read/write/execute permitted by the project creator and by all of the members of the project RON, but protected against all access by people outside the RON project.

Project members are encouraged to store any files needed by their experiments in the corresponding `/proj` project directory.

- **Are my files on users.emulab.net backed up (filesaved)?**

Yes. All of the files in your home directory on `/users`, and all of the files in your project directory in `/proj` are filesaved. While we can restore lost files in an emergency, we encourage you to back up critical data on your own to avoid (possibly long) delays in conducting your experiments.

- **Are the nodes in my experiment backed up (filesaved)?**

No! The nodes in your experiment are not filesaved. Any changes you make to the local filesystems will be lost if the event of a disk failure. We plan to provide a mechanism for experimenters to create snapshots of their node state, but that is not done yet. In the meantime, any files that must not be lost should be stored in the project directory (`/proj/`), which is available via NFS to all of the nodes in your experiment. You may also store files in your home directory (`/users/`), also available via NFS to all of your nodes, but that is not the preferred location since quotas on `/users` are relatively small.

- **What is Swapping?**

Swapping is when you (or we) temporarily swap your experiment out, releasing all of the nodes in the experiment. Your experiment is still resident in the Emulab database, and you can see its status in the web interface, but no nodes are allocated. Once an experiment is swapped out, you can swap it back in via the web interface by going to the Experiment Information page for your experiment, and clicking on the swapin option.

The swappable checkbox in the Begin Experiment web page is used to determine what experiments can be *automatically* swapped by the testbed scheduling system. Note that all experiments are capable of being swapped; even if you do not check the swappable box, you are free to swap your own experiments as you like. The only difference is that the testbed scheduling system will not consider your experiment when looking for experiments to swap out. You will sometimes notice that the Experiment Information page does not contain the swap link. That is because experiments cannot be swapped when they are in transition. For example, when the

experiment is being swapped in (say, after first being created) the link will disappear until the experiment is fully swapped in, and it is capable of being swapped out. You will need to occasionally reload the page so that the updated state is recognized and the swap link appears.

Be aware that we do not currently save any files that you may have placed on your nodes. When your experiment is swapped back in, you will likely get different nodes, and with fresh copies of the disk images. For that reason, you should not swap your experiment out unless you make arrangements to save and restore any state you need.

Make sure to take a look at our [Node Usage Policies](#) as well.

- **How can I get switch statistics (such as packet counts) for my experiment?**

We have a command called `portstats` that allows you access to some of the port counters on our switches. To use it, you'll need to ssh to **users.emulab.net**. `'portstats <pid> <eid>'` will get you stats for all experimental interfaces in your experiment. Run `'portstats -h'` to get a list of other options, such as different sets of stats.

Note that the numbers returned by `portstats` do not get reset between experiments.

Hardware Setup

- **What kind of computers are used for my nodes?**
- **How many nodes are there?**
- **How many ethernet cards are on each node?**

Please see the [Hardware Overview](#) page for a description and count of the computers that comprise the Testbed.

- **How many nodes are currently available (free)?**

If you click on the "Node Reservation Status" link in the menu to your left, you will see a summary of the number of nodes (by type) that are currently available, followed by a listing of the reservation status of each individual node.

- **Can I do traffic shaping on my links?**

Yes! You can specify the delay, bandwidth, and packet loss rate between any two nodes in your topology. Bandwidth and delay are specified in the NS `duplex-link` statement, while packet loss rate is specified with the Emulab `tb-set-link-loss` extension to NS. You may also specify delay, bandwidth, and packet loss rate between nodes in a regular LAN.

Please see the [Extensions](#) page for a summary of all Emulab NS extensions, and the [Emulab Tutorial](#) for an example.

- **Can I modify the traffic shapping parameters on my links?**

Yes! If your NS file specified traffic shaping on a link, then you can subsequently modify those parameters after the experiment has been swapped in. Note that you cannot *convert* a non shaped link into a shaped link; you can only modify the traffic shaping parameters of a link that is already being shaped. To modify the parameters, log into **users.emulab.net** and use the `delay_config` program. This program requires that you know the symbolic names of the individual links. This information is available via the web interface on the Experiment Information page. The command line syntax for `delay_config` will be displayed when the `-h` option is given.

- **Are there other traffic shapping parameters besides latency, bandwidth, and packet loss rate?**

Yes! However, access to those other parameters is slightly more difficult since you cannot specify them in your NS file. First off, you should log into `users.emulab.net` and read the man page for [ipfw](#). Refer to the section on `dummynet`; `ipfw` is the user interface for the `Dummynet` traffic shaper. As noted in previous section above, you can alter the traffic shapping parameters of any delayed link (one in which you have specified a bandwidth, delay, or PLR that causes a delay node to be inserted). However, you will need to log into the delay node for the link you wish to modify and interact with `ipfw` directly. The easiest approach would be to make a copy of `/etc/testbed/rc.delay` and edit the pipe commands as desired (or replace the pipe commands with "queue" commands). The pipe commands are indexed by number; the mapping from pipe number to virtual link is available via the web interface on the Experiment Information page for your experiment. Be sure to leave the rest of the contents of the file as is. Once you have your changes made, simply execute the file using the [sudo](#) command.

- **I asked for traffic shaping, but everything seems to be going at full LAN speeds. What's wrong?**

The most likely problem is that it is using the unshaped control network for the traffic you're looking at. This occurs when it tries to contact a node using a "pcXXX" address, like `pc76` or `pc76.emulab.net`, or when it tries to ping a fully-qualified name, like `NodeA.myexpt.myproj.emulab.net`, which also resolves to a control network address. On one of your nodes, take a look at the file `/etc/hosts`. It shows the IP addresses and aliases that refer to the different experimental interfaces. These are the names/IPs you can use to see the delays.

For instance, let's say I have an experiment that had NodeA and NodeB connected with a shaped link. The file `/etc/hosts` on NodeA would have a line for NodeB, with an address like `192.168.1.3`, and on NodeB there would be an entry for NodeA with the address `192.168.1.2`. These addresses correspond to the delayed link between them. Any address outside the `192.168.*.*` range that you didn't configure manually corresponds to an unshaped link.

Software Setup

- **What OS do the nodes run?**

Please see the [Software Overview](#) page for a description of the Operating Systems that can be run on each of the Testbed nodes.

- **How do I select which OS to run on each node?**

When a choice of OS is available, you may specify which one you prefer for each node in the NS file using the Emulab `tb-set-node-os` extension to NS. When your experiment is configured, the appropriate disk image will be loaded on your nodes, and the selected operating system will boot up on each.

Please see the [Extensions](#) page for a summary of all Emulab NS extensions, and the [Emulab Tutorial](#) for an example.

- **Can I load my own software (RPMs) on my nodes?**

Yes! If have an RPM (or more than one) that is appropriate for loading on the OS you have selected, you can arrange to have them loaded automatically when your experiment is configured. The Emulab NS extension `tb-set-node-rpms` is used in the NS file to specify a list of RPMS to install. You may specify a different list for each node in the experiment. When the node first boots after the experiment is configured, each of the RPMs will be installed (but only RPMs that have not already been installed).

Please see the [Extensions](#) page for a summary of all Emulab NS extensions, and the [Emulab Tutorial](#) for an example.

- **Can I schedule programs to run automatically when a node boots?**

Yes! You can arrange to run a single program or script when your node boots. The script is run as the UID of the experiment creator, and is run after all other node configuration (including RPM installation) has completed. The exit status of the script (or program) is reported back and is made available for you to view in Experiment Information link in the menu at your left. The Emulab NS extension `tb-set-node-startup` is used in the NS file to specify the path of the script (or program) to run. You may specify a different program for each node in the experiment.

Please see the [Extensions](#) page for a summary of all Emulab NS extensions, and the [Emulab Tutorial](#) for an example.

- **How can I turn on routing or set up routes automatically in my nodes?**

By default, we do not setup any static routes or run any routing daemon on nodes in an experiment. However, we do provide several options for experimenters, which are described in the "[Setting up IP routing between nodes](#)" section of the [Emulab Tutorial](#).

- **How does my software determine when other nodes in my experiment are ready?**

If your application requires synchronization to determine when all of the nodes in your experiment have started up and are ready to proceed, then you can use the Testbed's *ready bits* mechanism. The ready bits are really just a way of determining how many nodes have issued the **ready** command, and is returned to the application as a simple N of M string, where N is the number that have reported in, and M is the total number of nodes in the experiment. Applications can use this as a very simplistic form of barrier synchronization, albeit one that can be used just once and one

that does not actually block!

Use of the ready bits is described in more detail in the [Emulab Tutorial](#) and in the [Testbed Master Control Daemon](#) documentation.

• Can I run my own Operating System?

Yes! You can run your own OS (or a customized version of an Emulab supported OS) on any of the PCs. You can also run [OSKit](#) kernels on the PCs. Each of the PCs is partitioned with two DOS partitions large enough to hold the typical OS installation. The 1st and 2nd partitions are each 3GB. The 3rd partition is 500MB, and is labeled as Linux Swap. The 4th partition is the remainder of the disk, and varies in size depending on the pc type. We recommend that you use the 1st or 2nd partition; using the 4th partition will restrict the number of machines that you can run your OS on since it varies in size. Note that you must leave the MBR (Master Boot Record) in sector 0 alone, and that your custom partition must contain a proper DOS boot record in the first sector.

Please note that while users are free to customize their disks and install their own operating systems, Emulab staff will not be able to offer more than encouragement and advice! We cannot install the OS for you, and we cannot load CDROMS, floppy disks, or tape drives! We *do* provide an easy way for you to boot FreeBSD from a memory based filesystem (MFS) so that you can more easily work with the disk (in case it is not possible to install your OS on a live disk). Beyond that, you are pretty much on your own!

Many users had great success with customizing an Emulab supported OS (FreeBSD or Linux), and then creating a disk image that is autoloaded when the experiment is swapped in. We strongly encourage people to use this approach whenever possible! There is more information available in the [Custom OS](#) section of the [Emulab Tutorial](#).

• What if I need more disk space on my nodes?

Each node has a partition at the end of the disk that you can use if you wish. In Linux, the partition is `/dev/hda4` ; in FreeBSD, it's `/dev/ad0s4` . There is no filesystem on this partition, so you'll need to create it yourself. For example, in Linux:

```
mkfs /dev/hda4;
mount /dev/hda4 /mnt;
```

This partition is only 6 Gigs, the size of the leftover space on our smallest drives. If you need more space than this, it would be possible to enlarge this partition on some machines (for example, our pc850s have 40 GB disks,) but that is outside the scope of this FAQ.

• Are there testbed-specific daemons that could interfere with my experiment?

By default, the testbed startup scripts currently start two daemons in addition to the OS's standard set. Other daemons may be started depending on the network services you ask for in your ns file (see below).

Unconditionally started daemons:

- `healthd` - A low overhead hardware health monitor.

This daemon periodically polls the machine's health monitoring hardware and sends this information back to our `boss` node for analysis. The hardware is polled once per second, and a status datagram is sent out once every five minutes. `Healthd`'s overhead is quite low, but it can be safely killed and disabled from startup if you're worried about possible side effects. It is started by `/etc/testbed/rc.healthd`.

- `slothd` - A low overhead usage analysis tool.

`slothd` is important to efficient testbed utilization and should run on every node whenever possible. Its overhead is almost negligible (essentially less than running `'ls -l /dev'` once per hour), and should not interfere with your work. However if your experiment is exceptionally sensitive, then you may arrange with us to disable `slothd`. Please note that we will restart this daemon if it is not running unless prior arrangements have been made.

Conditionally started daemons:

- `gated` - A network routing daemon.

If you have requested automatic routing on your nodes with the `tb-set-ip-routing` command in your NS file, this will start `gated` on all of your nodes.

We have left all daemons started by the operating systems' default configurations (such as `cron`) enabled, so you should also look at them if you are concerned about running processes affecting your experiment.

Security Issues

• Is Emulab Firewalled?

Yes. Emulab blocks all of the *low numbered* ports (ports below 1024), with the exception of ports 20 and 21 (FTP), 22 (Secure Shell), and 80 (HTTP). This is for the protection of experimenters, as well as to ensure that an errant application cannot become the source of a Denial of Service attack to sites outside of Emulab. If your application requires external access to other low numbered ports, please contact us to make special arrangements.

Troubleshooting

• My experiment is setup, but I cannot send packets between some of the nodes, why?

The most common reason is that your topology includes nodes which are not directly connected, and you have not setup any routing. Refer to "[How can I turn on routing or set up routes automatically in my nodes?](#)" for details. If you cannot send packets between two machines which

are directly connected (via a link or a lan), then there are two possibilities: either the nodes did not properly negotiate their speed and duplex with the Cisco switch, or the physical wire is loose or bad. In these cases, you should contact us for help.

Emulab - Testbed NS Command Extensions

Contents

- [Introduction](#)
 - [TCL, NS, and node names](#)
 - [Ordering issues](#)
 - [Hardware Commands](#)
 - [IP Address Commands](#)
 - [OS Commands](#)
 - [Link Characteristic Commands](#)
 - [Virtual Type Commands](#)
 - [Misc. Commands](#)
-

Introduction

In order to use the testbed specific commands you must include the following line near the top of your NS file (before any testbed commands are used):

```
source tb_compat.tcl
```

If you wish to use your file under NS you can use download this [tb_compat.tcl](#). Place it in the same directory as your NS file. When run in this way under NS the testbed commands will have no effect, but NS will be able to parse your file.

TCL, NS, and node names

In your file you will be creating nodes with something like:

```
set node1 [$ns node]
```

What is really going on is that the simulator, represented by `$ns` is creating a new node, involving a bunch of internal data changes, and returning a reference to it which is stored in the variable `node1`. In almost all cases, when you need to refer to a node you will do it as `$node1`, the `$` indicating that you want the value the variable `node1`, i.e. the reference to the node. Thus you will be issuing commands like:

```
$ns duplex-link $node1 $node2 100Mb 150ms DropTail
tb-set-ip $node1 192.0.0.2
```

(Note the `$`'s)

You will notice that when your experiment is setup the node names and such will be `node1`. This happens because the parser detects what variable you are using to store the node reference and uses that as the node name. In the case that you do something like:


```
set node1 [$ns node2]
set A $node1
```

The node will still be called `node1` as that was the first variable to contain the reference.

If you are dealing with many nodes you may store them in array, perhaps like this:

```
for {set i 0} {$i < 4} {incr i} {
    set nodes($i) [$ns node]
}
```

In this case the names of the node will be `nodes-0`, `nodes-1`, `nodes-2`, `nodes-3`. (In other words, the `(` character is replaced with `-`, and `)` is removed.) This slightly different syntax comes is to avoid any problems that `(` may cause later in the process. For example, the `(` characters cannot appear in DNS entries.

As a final note, everything said above for nodes applies equally to lans. I.e.:

```
set lan0 [$ns make-lan "$node0 $node1" 100Mb 0ms]
tb-set-lan-loss $lan0 .02
```

(Again, note the `$`'s)

Links can also be named just like nodes and lans. The names can then be used to set loss rates or IP addresses. This technique is the only way to set such attributes when there are multiple links between two nodes.

```
set link1 [$ns duplex-link $node0 $node1 100Mb 0ms DropTail]
tb-set-link-loss $link1 0.05
tb-set-ip-link $node0 $link1 192.0.0.128
```

Ordering Issues

`tb-` commands have the same status as all other Tcl and NS commands. Thus the order matters not only relative to each other but also relative to other commands. One common example of this is that IP commands must be issued after the links or lans are created.

Hardware Commands

tb-set-hardware

```
tb-set-hardware node type [args].

tb-set-hardware $node3 pc
tb-set-hardware $node4 shark
```

node - The name of the node.

type - The type of the node.

Notes:

- Currently only `pc`, `pc600`, `pc850`, and `shark` are supported types. `pc` is the default type.
- No current types have any additional arguments.

IP Address Commands

Each node will be assigned an IP address for each interface that is in use. The following commands will allow you to explicitly set those IP addresses. IP addresses will be automatically generated for all nodes that you do not explicitly set IP addresses.

In the common case the IP addresses on either side of a link must be in the same subnet. Likewise, all IP addresses on a LAN should be in the same subnet. Generally the same subnet should not be used for more than one link or LAN in a given experiment, nor should one node have multiple interfaces in the same subnet. Automatically generated IP addresses will conform to these requirements. If part of a link or lan is explicitly specified with the commands below then the remainder will be automatically generated under the same subnet.

IP address assignment is deterministic and tries to fill lower IP's first, starting at 2. Except in the partial specification case (see above), all automatic IP addresses are in the network `192.168`.

tb-set-ip

```
tb-set-ip node ip
tb-set-ip $node1 142.3.4.5
```

node - The node to assign the IP address to.

ip - The IP address.

Notes:

- This command should only be used for nodes that have a single link. For nodes with multiple links the next commands should be used. Mixing `tb-set-ip` and any other IP command on the same node will result in an error.

tb-set-ip-link

```
tb-set-ip-link node link ip
tb-set-ip-link $node0 $link0 142.3.4.6
```

node - The node to set the IP for.

link - The link to set the IP for.

ip - The IP address.

Notes:

- One way to think of the arguments is a link with the node specifying which side of the link to set

the IP for.

- This command can not be mixed with `tb-set-ip` on the same node.

tb-set-ip-lan

```
tb-set-ip-lan node lan ip
```

```
tb-set-ip-lan $node1 $lan0 142.3.4.6
```

node - The node to set the IP for.

lan - The lan the IP is on.

ip - The IP address.

Notes:

- One way to think of the arguments is a node with the LAN specifying which port to set the IP address for.
- This command can not be mixed with `tb-set-ip` on the same node.

tb-set-ip-interface

```
tb-set-ip-interface node dst ip
```

```
tb-set-ip-interface $node2 $node1 142.3.4.6
```

node - The node to set the IP for.

dst - The destination of the link to set the IP for.

IP - The IP address.

Notes:

- This command can not be mixed on the same node with `tb-set-ip`. (See above)
- In the case of multiple links between the same pair of nodes there is no way to distinguish which link to set the IP for. This should be fixed soon.
- This command is converted internally to either `tb-set-ip-link` or `tb-set-ip-lan`. It is possible that error messages will report either of those commands instead of `tb-set-ip-interface`.

OS Commands

tb-set-node-os

```
tb-set-node-os node os
```

```
tb-set-node-os $node1 FBSD-STD
```

```
tb-set-node-os $node1 MY_OS
```

node - The node to set the OS for.

os - The id of the OS for that node.

Notes:

- The OSID can either be one of the standard OS's we provide or a custom OSID, created via the web interface.
- If no OS is specified for a node a default OS is chosen based on the node's type. This is currently RHL-STD for PCs.
- The currently available standard OS types are: FBSD-STD, RHL-STD, NBSD14-STD (should not be used on PC nodes), and NETBOOT-STD (oskit netboot kernel for loading other operating systems over the network).

tb-set-node-rpms

```
tb-set-node-rpms node rpms...

tb-set-node-rpms $node0 rpm1 rpm2 rpm3
```

Notes:

- This command sets which rpms are to be installed on the node.
- This command sets which rpms are to be installed on the node when it first boots after being assigned to an experiment.
- See the [tutorial](#) for more information.

tb-set-node-startup

```
tb-set-node-startup node startupcmd

tb-set-node-startup $node0 {mystart.sh -a}
```

Notes:

- Specify a script or program to be run when the node is booted.
- See the [tutorial](#) for more information.

tb-set-node-cmdline

```
tb-set-node-cmdline node cmdline

tb-set-node-cmdline $node0 {???
```

Notes:

- Set the command line, to be passed to the *kernel* when it is booted.
- Currently, this is supported on OSKit kernels only.

tb-set-node-tarfiles

```
tb-set-node-tarfiles node dir tarfile

tb-set-node-tarfiles $node0 /bin mybinmods.tar /sbin mysbinmods.tar
```

Notes:

- This installs tarfiles in specified directories when the node first boots after being assigned to an experiment.
 - Each tar file is installed just one. Tarfiles that have been loaded, are not reloaded after subsequent reboots.
-

Link Loss Commands

This is the NS syntax for creating a link:

```
$ns duplex-link $node1 $node2 100Mb 150ms DropTail
```

Note that it does not allow for specifying link loss rates. Emulab does, however, support link loss. The following commands can be used to specify link loss rates.

tb-set-link-loss

```
tb-set-link-loss src dst loss
tb-set-link-loss link loss

tb-set-link-loss $node1 $node2 0.05
tb-set-link-loss $link1 0.02
```

src, dst - Two nodes to describe the link.

link - The link to set the rate for.

loss - The loss rate (between 0 and 1).

Notes:

- There are two syntax's available. The first specifies a link by a source/destination pair. The second explicitly specifies the link.
- The source/destination pair is incapable of describing an individual link in the case of multiple links between two nodes. Use the second syntax for this case.

tb-set-lan-loss

```
tb-set-lan-loss lan loss

tb-set-lan-loss $lan1 0.3
```

lan - The lan to set the loss rate for.

loss - The loss rate (between 0 and 1).

Notes:

- This command sets the loss rate for the entire LAN.

tb-set-node-lan-delay

```
tb-set-node-lan-delay node lan delay
```

```
tb-set-node-lan-delay $lan0 $node0 40ms
```

node - The node we are modifying the delay for.

lan - Which LAN the node is in that we are affecting.

delay - The new node to switch delay (see below).

Notes:

- This command changes the delay between the node and the switch of the LAN. This is only half of the trip a packet must take. The packet will also traverse the switch to the destination node, possibly incurring additional latency from any delay parameters there.
- If this command is not used to overwrite the delay, then the delay for a given node to switch link is taken as one half of the delay passed to `make-lan`. Thus in a LAN where no `tb-set-node-delay` calls are made the node to node latency will be the latency passed to `make-lan`.
- The behavior of this command is not defined when used with nodes that are in the same LAN multiple times.
- Delays of less than 2ms (per trip) are too small to be accurately modeled at this time, and will be silently ignored. As a convenience, a delay of 0ms can be used to indicate that you do not want added delay; the two interfaces will be "directly" connected to each other.

tb-set-node-lan-bandwidth

```
tb-set-node-lan-bandwidth node lan bandwidth
```

```
tb-set-node-lan-bandwidth $lan0 $node0 20Mb
```

node - The node we are modifying the bandwidth for.

lan - Which LAN the node is in that we are affecting.

bandwidth - The new node to switch bandwidth (see below).

Notes:

- This command changes the bandwidth between the node and the switch of the LAN. This is only half of the trip a packet must take. The packet will also traverse the switch to the destination node which may have a lower bandwidth.
- If this command is not used to overwrite the bandwidth, then the bandwidth for a given node to switch link is taken directly from the bandwidth passed to `make-lan`.
- The behavior of this command is not defined when used with nodes that are in the same LAN multiple times.

tb-set-node-lan-loss

```
tb-set-node-lan-loss node lan loss
```

```
tb-set-node-lan-loss $lan0 $node0 0.05
```

node - The node we are modifying the loss for.

lan - Which LAN the node is in that we are affecting.

loss - The new node to switch loss (see below).

Notes:

- This command changes the loss probability between the node and the switch of the LAN. This is only half of the trip a packet must take. The packet will also traverse the switch to the destination node which may also have a loss chance. Thus for packet going to switch with loss chance A and then going on the destination with loss chance B the node to node loss chance is $(1 - (1 - A)(1 - B))$.
- If this command is not used to overwrite the loss, then the loss for a given node to switch link is taken from the loss rate passed to the `make-lan` command. If a loss rate of L is passed to `make-lan` then the node to switch loss rate for each node is set to $(1 - \sqrt{1 - L})$. Thus as each packet will have two such chances to be lost the node to loss rate comes out as the desired L .
- The behavior of this command is not defined when used with nodes that are in the same LAN multiple times.

tb-set-node-lan-params

```
tb-set-node-lan-params node lan delay bandwidth loss
```

```
tb-set-node-lan-params $node0 $lan0 40ms 20Mb 0.05
```

node - The node we are modifying the loss for.

lan - Which LAN the node is in that we are affecting.

delay - The new node to switch delay.

bandwidth - The new node to switch bandwidth.

loss - The new node to switch loss.

Notes:

- This command is exactly equivalent to calling each of the above three commands appropriately. See above for more information.

tb-set-link-simplex-params

```
tb-set-link-simplex-params link src delay bw loss
```

```
tb-set-link-simplex-params $link1 $srcnode 100ms 50Mb 0.2
```

link - The link we are modifying.

src - The source, defining which direction we are modifying.

delay - The source to destination delay.

bw - The source to destination bandwidth.

loss - The source to destination loss.

Notes:

- This commands modifies the delay characteristics of a link in a single direction. The other direction is unchanged.
- This command only applies to links. Use `tb-set-lan-simplex-params` below for LANs.

tb-set-lan-simplex-params

```
tb-set-lan-simplex-params lan node todelay tobw toloss fromdelay frombw fromloss
```

```
tb-set-lan-simplex-params $lan1 $node1 100ms 10Mb 0.1 5ms 100Mb 0
```

lan - The lan we are modifying.

node - The member of the lan we are modifying.

to delay - Node to lan delay.

to bw - Node to lan bandwidth.

to loss - Node to lan loss.

from delay - Lan to node delay.

from bw - Lan to node bandwidth.

from loss - Lan to node loss.

Notes:

- This command is exactly like `tb-set-node-lan-params` except that it allows the characteristics in each direction to be chosen separately. See all the notes for `tb-set-node-lan-params`.

Virtual Type Commands

Virtual Types are a method of defining fuzzy types. I.e. types that can be fulfilled by multiple different physical types. The advantage of virtual types (vtypes) is that all nodes of the same vtype will usually be the same physical type of node. In this way, vtypes allows logical grouping of nodes.

As an example, imagine we have network with internal routers connecting leaf nodes. We want the routers to all have the same hardware, and the leaf nodes to all have the same hardware, but the specifics of this hardware do not matter. We have the following fragment in our NS file:

```
...
tb-make-soft-vtype router {pc600 pc850}
tb-make-soft-vtype leaf {pc600 pc850}

tb-set-hardware $router1 router
tb-set-hardware $router2 router
tb-set-hardware $leaf1 leaf
tb-set-hardware $leaf2 leaf
```

Here we have set up two soft (see below) vtypes, router and leaf. Our router nodes are then specified to be of type router, and the leaf nodes of type leaf. When the experiment is swapped in the testbed will attempt to make router1 and router2 be of the same type, and similarly, leaf1 and leaf2 of the same type. However, the routers/leaves may be pc600s or they may be pc850s, whichever is easier to fit in to the available resources.

As a basic use, vtypes can be used to request nodes that are all the same type, but can be of any available type:

```
...
tb-make-soft-vtype N {pc600 pc850}

tb-set-hardware $node1 N
tb-set-hardware $node2 N
```


...

Vtypes come in two varieties, hard and soft. With soft vtypes, the testbed will try to make all nodes of that vtype the same physical type, but may do otherwise if resources are tight. Hard vtypes behave just like soft vtypes except that the testbed will give higher priority to vtype consistency and swapping in will fail if the vtypes can not be satisfied. So, if you use soft vtypes you are more likely to swap in but there is a chance your node of a specific vtype will not all be the same. If you use hard vtypes all nodes of a given vtype will be the same, but swapping in may fail.

Finally, you can have weighted soft vtypes. Here you assign a weight from 0 to 1 exclusive to your vtype. The testbed will give higher priority to consistency in the higher weighted vtypes. The primary use of this is to rank multiple vtypes by importance of consistency. Soft vtypes have a weight of 0.5 by default.

As a final note, when specifying the types of a vtype, use the most specific type possible. For example: `tb-make-soft-vtype router {pc pc600}`, is not very useful, as pc600 is a sub type of pc. You may very well end up with two routers as type pc with different hardware, as pc covers multiple types of hardware.

tb-make-soft-vtype

```
tb-make-soft-vtype vtype {types}
tb-make-hard-vtype vtype {types}
tb-make-weighted-vtype vtype weight {types}

tb-make-soft-vtype router {pc600 pc850}
tb-make-hard-vtype leaf {pc600 pc850}
tb-make-weighted-vtype A 0.1 {pc600 pc850}
```

vtype - The name of the vtype to create.

types - One or more physical types.

weight - The weight of the vtype, $0 < weight < 1$.

Notes:

- These commands create vtypes. See notes above for description of vtypes and the difference between soft and hard.
- `tb-make-soft-vtype` creates vtypes with weight 0.5.
- vtype commands must appear before `tb-set-hardware` commands that use them.
- Do not use `tb-fix-node` with nodes that have a vtype.

Misc. Commands

tb-fix-node

```
tb-fix-node vnode pnode

tb-fix-node $node0 pc42
```

vnode - The node we are fixing.

pnode - The physical node we want used.

Notes:

- This command forces the virtual node to be mapped to the specified physical node. Swap in will fail if this can not be done.
- Do not use this command on nodes that are a virtual type.

tb-set-uselatestwadata

```
tb-set-uselatestwadata 0
tb-set-uselatestwadata 1
```

Notes:

- This command indicates which widearea data to use when mapping widearea nodes to links. The default is 0, which says to use the aged data. Setting it to 1 says to use the most recent data.

tb-set-wasolver-weights

```
tb-set-wasolver-weights delay bw plr
tb-set-wasolver-weights 1 10 500
```

delay - The weight to give delay when solving.

bw - The weight to give bandwidth when solving.

plr - The weight to give lossrate when solving.

Notes:

- This command sets the relative weights to us when assigning widearea nodes to links. Specifying a zero says to ignore that particular metric when doing the assignment. Setting all three to zero results in an essentially random selection.

Introduction to Netbuild

Basic usage:

- To create a Node, click and drag the "new node" icon in the palette (the left panel) to the work area (the middle panel.)
- To link two nodes together, click one node, then hold "ctrl" and click the node you want to link it to.
- To create a LAN, click and drag the "new LAN" icon in the palette to the work area.
- To link a node to a LAN, click the node, then hold "ctrl" and click on the LAN you want to link it to (or vice versa.)
- Note that you may not link a LAN directly to a LAN.
- Nodes and LANs may be moved around the work area by clicking and dragging them.
- Nodes, links, and LANs may be eliminated by dragging them into the trash.
- Little circles between links and nodes represent the network interface on that node, and may not be moved or eliminated (though eliminating the attached link will get rid of them.)
- When your experiment topology has been built, clicking "create experiment" will take you into the testbed experiment creation page; from there, you may view the generated NS file as well as create an actual experiment.

Selection:

- Clicking a single item in the work area will select it.
- Clicking a single item while holding down shift will add it to the pool of selected items (unless it was already selected, then it will become deselected.)
- Clicking an empty part of the canvas will deselect everything.
- Clicking an empty part of the canvas and dragging will create a selection rectangle. When the mouse button is released, all items in the rectangle will be selected.
- When multiple items are selected, clicking one selected item and dragging it will drag all selected items.
- When multiple items are selected, clicking one selected item and dragging it into the trash will eliminate all selected items.

Properties:

- When one item is selected, a properties view will open in the rightmost panel. This is where

properties, including name, may be set.

- Invalid characters typed into a properties text box will be ignored.
- If a property is "<auto>," it will automatically be determined in the experiment creation process. Change this to specify your own value for the property.
- Clicking "default" will reset the property of the adjoining box to its default value (in many cases "<auto>".)
- If multiple items of the same type are selected, some boxes may show up as "<multiple>." This means that the value of this property is different between at least two nodes in the selection. Changing such a property value (or setting it to default) will result in *all* selected nodes assuming the new value for that property. Some properties (such as name and IP address) may not be set simultaneously in this way.
- If multiple items of varying types are selected, the property boxes will still appear on the right, but may be in a collapsed state. Clicking the '+' will expand one of these.
- Clicking the '-' on an expanded box will collapse it again.
- Click "copy selection" to make a duplicate of the current selection.

Project Groups

As an instructional aid, project leaders may designate TAs to lead small groups of project members. This is accomplished by creating a *group* (sometimes referred to as a "subgroup"), and designating the TA as the leader of the group. A project group is a lot like a unix group, and in fact unix groups is the mechanism used to protect members of one group from members of another group on Emulab nodes. For each group created, a new unix group is created, and the members of the group added. When a group member starts an experiment, he/she indicates the group in the Begin Experiment form. All of the nodes in the experiment will have user accounts built for only those members of the group. In this way, multiple subgroups of a project can work independently, and be protected from each other via the generally well understood unix group protection mechanism.

As a convenience, all new projects are created with one new group, termed the *default group*. All project members are in the default group, and as its name implies, whenever the group is left unspecified in a form, it defaults to the project group (this allows you to create a project without any sub groups at all; new members join the default group, new experiments are created in the default group, etc.).

As project leader, you may create and destroy subgroups, and add or remove project members from your groups. You are automatically a member of new groups you create; even though you are not the designated leader of the group, you still retain all of the same permissions that you have as project leader, within the group. This means that you can terminate experiments that have been created within the group, and edit the personal information for group members. To create a group, simply go to the Project Information link at your left, and look for the "Create New Group" link, or go to the [Create New Group](#) page directly. Once you have created a group, you can edit the members of the group by clicking on the "Edit" option in the group information page.

As group leader, you may approve new user applications to join your group. You may also create and destroy experiments created within the group. If you are a TA managing a group, you can have new Emulab users *Join* your group by telling them to go to the [Join Project](#) link at your left, and specifying the name of your group where it asks for a group name. You will receive an email message for each person that applies to join your group. To approve (or deny) membership in your group, use the [New User Approval](#) link. If the user who wishes to join your group is already a member of the project, then the project leader must add them to your group. In other words, there is no mechanism to join multiple groups via a web form; the Project Leader must do it on the Edit Group page.

These are some important security issues to keep in mind:

- Unix groups are used to protect members of one group from members of another group. Users may create shared directories by using the unix `chgrp` command. When accounts are created on the experimental nodes after a new experiment is started, only those members of the group will get accounts on the nodes; other members of the same project, not in the group, will not get accounts.
- Emulab uses NFS mounted filesystems for `/users` and `/proj` on the experimental nodes. Because of the nature of NFS, giving root privileges to a user will allow them to read/write any files on any filesystems that are mounted, since root access allows them to `su` as any other user. Thus, any files in the project directory and in the home directories of other members of the group, can be "compromised" by a group member. Please note that no other directories are NFS mounted; other projects and users on Emulab are safe.
- It is important to remember that granting "root" permissions to a user in the project (or default group) and "user" permissions in a subgroup, is inconsistent and can result in a breach of privacy.

Consider this example; user Joe has "root" permissions in the default group, and "user" permissions in a subgroup. Another user Bob is in the same subgroup as Joe, and since Joe has "user" permissions, it was probably intended that Joe would not be able to read Bob's files. Joe now creates an experiment, specifying the default group in the web form. The nodes in Joe's experiment would get NFS mounts for all of the members in the project (including Bob), and since Joe has "root" permission in the default group, would be granted root access on his nodes. Joe can now access the files of all members of the project, including Bob. The correct approach is to specify "user" permissions in the default group, and either "user" or "root" in the subgroup (depending on whether subgroup members are mutually trust each other and need to create experiments).

- Equally dangerous is specifying different levels of trust for a user that is in multiple subgroups of a project. In this case, any *other* users that are in the same groups (overlapping) are potentially at risk. For example, if Joe has "root" permissions in one subgroup, and "user" permissions in a second subgroup, and there is another user Bob who is in both subgroups, then Joe can access Bob's files when creating an experiment in one of the subgroups, but not the other. If Joe is really not supposed to access Bob's files, then Joe should not have "root" permission in a subgroup that contains Bob.

You have the following choices for Trust:

- User - User may log into machines in your experiments
- Local Root - User may create/destroy experiments in your project and has root privileges on machines in your experiments
- Group Root - In addition to Local Root privileges, user may also approve new group members and modify user info for other users within the group. This level of trust is typically given only to TAs and the like.

Testbed Master Control Daemon/Client Reference

Contents

- [Introduction](#)
 - [TMCC client program](#)
 - [Node Setup Script](#)
 - [Command Reference](#)
 - [reboot](#)
 - [status](#)
 - [ifconfig](#)
 - [accounts](#)
 - [mounts](#)
 - [delay](#)
 - [hostnames](#)
 - [rpms](#)
 - [startupcmd](#)
 - [startstatus](#)
 - [ready](#)
 - [readycount](#)
 - [log](#)
-

• Introduction

The **Testbed Master Control Daemon** (TMCD) is a program that runs on **boss.emulab.net**, and provides configuration information to Testbed nodes when they boot up. The **Testbed Master Control Client** (TMCC), is a small program that is installed on each node, and is used to connect to the TMCD to issue requests and get the response. In addition, Testbed nodes use the TMCC/TMCD to communicate events of interest back to the Emulab Database and to the user via the Web interface. The TMCD interface is text based; clients issue requests in the form of strings consisting of a command and an optional argument. The response is also a string, in a very generic format that can be easily parsed by any C/C++ program or shell interpreter. For example, to determine how to configure the experimental interfaces on each testbed node in your experiment when it boots, you would do the following:

```
tmcc ifconfig
```

The response to this request would be:

```
INTERFACE=1 INET=10.0.0.1 MASK=255.255.255.0  
INTERFACE=2 INET=10.0.1.1 MASK=255.255.255.0
```

which indicates that interfaces eth1 and eth2 (or perhaps fxp1 and fxp2) should be configured to the given IP addresses and netmasks.

• TMCC

The **TMCC** is a simple client program that runs on the testbed nodes and handles the details of

connecting to the **TMCD**, issuing the request, getting the response, and printing it out. It has been compiled on FreeBSD 4.x, Redhat Linux 6.2 and 7.1, and Netbsd 1.4, and should compile on just about any operating system. Alternatively, you can integrate the TMCC into your own programs. Briefly, the TMCC connects to port 7777 (UDP or TCP) on **boss.emulab.net**, writes a single string to the connection, and then waits for an **optional** response, which is a newline separated list of strings. The TMCC exits when the other side of the connection is closed by the TMCD. The source code for the TMCC is available upon request by sending email to [Testbed Operations \(testbed-ops@flux.cs.utah.edu\)](mailto:testbed-ops@flux.cs.utah.edu)

• Node Setup Script

The Emulab versions of FreeBSD 4.3, Redhat Linux 7.1, and Netbsd 1.4 all run a *setup* script at bootup that uses the TMCC client to configure the node. All of the interfaces are configured, user accounts for each of the members of the project are created, NFS mounts are made, etc. These setup scripts are located in /etc/testbed on FreeBSD and Netbsd, and in /etc/rc.d/testbed on Linux. You can use these scripts as a guide when writing setup code to configure your custom operating systems.

• Command Reference

◦ reboot

Report that a node has rebooted to the TMCD. This is an informational message that is used by the TMCD to determine when a node reboots for the first time after its disk has been reloaded. No response is returned.

◦ status

Request status information about the project and experiment that the node is currently part of. Returns the project ID, experiment ID, and the node *name* from the NS file that described the topology. This command is typically used to determine if the setup script needs to do any further configuration; if the node is free, then no other information is going to be provided by the TMCD. The format of the reply is one of:

```
FREE
ALLOCATED=pid/eid NICKNAME=name
```

The first form indicates that the node is not currently allocated to an experiment. The second form says that the node is running as part of the "eid" experiment in the "pid" project, and was named "name" in the NS file that described the topology.

◦ ifconfig

Request the configuration information for each of the network interfaces on the node, as determined by the topology described in the NS file, and the assignment of IP addresses to interfaces that is performed when the experiment is configured. The information that is returned is typically converted into corresponding `ifconfig` commands on the node. However, the information can be used in any manner that is appropriate for the operating system that is running on the node. The reply to this request is one or more lines in the following format (in the unlikely case that the topology describes a node with no network

links, the response to this request will be null):

```
INTERFACE=Z INET=X.X.X.X MASK=Y.Y.Y.Y MAC=AA:BB:CC:DD:EE:FF
```

Which says that the network interface with MAC address "AA:BB:CC:DD:EE:FF" is assigned to IP address "X.X.X.X" with netmask "Y.Y.Y.Y". *The INTERFACE specification is currently invalid, since there no way to achieve a consistent ordering of interfaces between various operating systems.* Rather, the MAC address is used to determine which interface to configure. A utility program called `/etc/testbed/findif` is provided to map the MAC address to an interface name suitable for use with the `ifconfig` program. On Redhat 7.1, the setup script would take this information and issue the following shell commands.

```
iface=`/etc/testbed/findif AA:BB:CC:DD:EE:FF`  
/sbin/ifconfig $iface inet X.X.X.X netmask Y.Y.Y.Y
```

○ accounts

Request group and login account information for each of the project members of the project that the experiment is running. This information can be used to generate login accounts for project members on each of the nodes in an experiment. The Emulab versions of FreeBSD, Linux, and Netbsd all have stub password/group files that do not contain any user accounts or groups. When a node first boots after being allocated to an experiment, this command is used to find out what accounts to build. The reply to this request is one or more lines of group information, followed by one or more lines of login account information:

```
ADDGROUP NAME=pid GID=XXXX  
ADDUSER LOGIN=joe PSWD=ABCD UID=YYYY GID=XXXX ROOT=N NAME="Joe User" \  
HOMEDIR=/users/joe GLIST=ZZZ0,ZZZ1
```

The `ADDGROUP` reply gives the name of the group and the numeric `gid` for that group. The `ADDUSER` reply has the following fields:

LOGIN	The user/account name.
PSWD	The <i>encrypted</i> password string, suitable for direct insertion into the password file.
UID	The numeric uid.
GID	The primary group for the user, as a numeric gid.
ROOT	Indicates whether the user should be granted root access by placing the user into the root group (wheel group on FreeBSD/NetBSD).
NAME	The full name of the user, suitable for insertion into the <i>gecos</i> field of the user's password entry.
HOMEDIR	The absolute path to be used for the home directory.
GLIST	A (possibly null) comma separated list of auxiliary group ids, as numeric gids.

On Linux, this information would be converted into the following commands:

```
groupadd -g XXXX pid
useradd -u YYYY -g XXXX -p ABCD -G root,ZZZ0,ZZZ1 -d /users/joe -c "Joe"
```

○ mounts

Request the list of remote directories that need to be NFS mounted on the node when it boots. The reply to this request is one or more lines in the following format:

```
REMOTE=fs.emulab.net:/users/joe LOCAL=/users/joe
REMOTE=fs.emulab.net:/proj/testbed LOCAL=/proj/myproj
```

On Linux, this information would be converted into the following commands:

```
mkdir /users/joe
/sbin/mount fs.emulab.net:/users/joe /users/joe
mkdir /proj/myproj
/sbin/mount fs.emulab.net:/proj/myproj /proj/myproj
```

○ rpms

Request the list of RPMs that should be installed on the node when it boots, as specified in the NS file on a per-node basis. The reply to this request is null if there are no RPMs to install, or one or more lines in the following format:

```
RPM=/path/to/name.rpm
```

On Linux and FreeBSD, each RPM is installed with the `rpm` command, which will install the RPM only if it has not already been installed:

```
rpm -i /path/to/name.rpm
```

○ startupcmd

Request the name of the startup script (or program) that should be run when the node boots, as specified in the NS file on a per-node basis. The reply to this request is null if a startup script was not specified, or a single line in the following format:

```
CMD=/path/to/runme UID=joe
```

Which says to run `/path/to/runme` as user `joe` when the node boots. The UID is always the experiment creator. On FreeBSD, Linux, and NetBSD, the command is run once the node is running multiuser. If the node reboots before the experiment is terminated, the command will be run again.

○ startstatus

Report the numeric exit value of the `startupcmd` back to the TMCD so that it can be recorded and displayed in the "Experiment Information" Web page. In fact, this does not have to be the result of the `startupcmd`, but can be the result of any application program. The intent is to report back information that can be used by the experimenter to determine when the experiment has finished. Each node reports back status individually. The format of

this **command** is:

```
tmcc startstatus XX
```

which sends the numeric value *XX* back to the TMCD. There is no response from the TMCD to this command.

- **ready**

Report an application level *ready* status back to the TMCD so that it can record it. A count of nodes (in your experiment) reporting ready is maintained by the TMCD, and is made available to nodes via the *readycount* request below. There is no response from the TMCD to this command.

- **readycount**

Request the count of nodes that have reported in *ready* with the *ready* command above. This is an application level count; nodes can use this as a very primitive form of synchronization to determine when all of the nodes in the experiment have started the application (say, via the *startupcmd* above) and have reached a point where it is necessary to wait until all of the nodes have reached the same point. The reply to this request is a single line in the following format:

```
READY=N TOTAL=M
```

which says that *N* nodes have reported in, of a total number of *M* nodes in the experiment. The application can continue to poll until *N==M*, but be sure to add some delay between each poll to avoid livelock at the TMCD. Note that the ready count is essentially a use-once feature; The ready count cannot be reinitialized to zero since there is no actual synchronization happening. If in the future it appears that a generalized barrier synchronization would be more useful, we will investigate the implementation of such a feature.

- **hostnames**

Request information about the IP addresses and node names of all of the nodes in the experiment. The intent is to provide the ability to easily generate a suitable */etc/hosts* file that allows experiments to operate using the symbolic names of the nodes (as defined in the *NS* file), instead of IP addresses, which are generally assigned by the configuration software, not the experimenter. Since nodes can use multiple experimental interfaces, the reply gives the IP address for each interface on each node. An additional alias is returned for nodes that are directly connected to the node making the hostnames request. Secondary interfaces, and interfaces that are not directly connected are named with a *-X* suffix, where *X* is the ordinal number of the interface. The reply to this request is one or more lines in the following format:

```
NAME=nodeA LINK=X IP=X.X.Y.A ALIAS=nodeA
NAME=nodeB LINK=Y IP=X.X.Y.B ALIAS=nodeB
NAME=nodeC LINK=Z IP=X.X.Z.C ALIAS=
```

The `LINK` field is the number of the network interface on the destination node, that this node is connected to. The `/etc/hosts` file that would be created for this response is:

```
X.X.X.A      nodeA-X nodeA
X.X.X.B      nodeB-Y nodeB
X.X.Z.C      nodeC-Z
```

Say that nodeA is making this request. NodeA is obviously connected to itself, so it gets an alias pointing to its own interface. NodeA is directly connected to NodeB on NodeB's `Y` interface, so it to gets an alias so that an application running on nodeA can just use the name NodeB. NodeC is not directly connected to NodeA (perhaps it is connected to NodeB on one of NodeB's other interfaces), so it does not get an alias. Referring to nodeC on nodeA would be confusing and possibly incorrect.

o **log**

The `log` command can be used by an application to write a message to a log file on **users.emulab.net**. This is especially useful on the Sharks, most of which do not have console serial lines attached. The argument to the `log` command is a single string, in double quotes if operating within the shell:

```
tmcc log "This is a log message"
```

The log file is stored in `/proj/pid/logs/eid.log`, where `pid` is the name of the project and `eid` is the name of the experiment. The file is appended to each time; it is the responsibility of the experimenter to zero the log file when done, or if a new experiment with the same name is started.