HOWTO: WIRELESS NETWORK EMULATION USING NS2 AND USER-MODE-LINUX (UML).

VERSION 1.1

4TH FEBRUARY 2006

Daniel Mahrenholz, Svilen Ivanov

University of Magdeburg, Germany

Contents

1 About This Document

This document describes how to run a set of User-Mode-Linux virtual machines [2], and use the network simulator ns-2 [1] to transparently connect them in a virtual simulated network.

This document is based on a scientific work from Daniel Mahrenholz and Svilen Ivanov [6].

1.1 Copyright Information

1.2 Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, and although this is highly unlikely, the author does not take any responsibility for that.

1.3 Credits

This work was sponsored by the German Research Foundation (DFG), grand no. NE 837/3-1 and the University of Magdeburg, Germany.

1.4 Feedback

Feedback is most certainly welcome for this document. Without your submissions and input, this document wouldn't exist. Please send your additions, comments and criticisms to the following email address:

<svilen (at) ivs.cs.uni-magdeburg.de>.

1.5 Support

If you encounter problems during the following of this document, you can post a request on our "nse – NS-2 Emulation Extension" mailing list. Information about this mailing list can be found at `http://mail-ivs.cs.uni-magdeburg.de/mailman/listinfo/nse`.

The homepage of this project is: `http://ivs.cs.uni-magdeburg.de/EuK/forschung/projekte/nse/index.shtml`.

2 Quick Walk-through

This section gives the step-by-step procedure to run ns-2 emulation in a virtual network consisting of User-Mode-Linux virtual machines. If you encounter problems here, please read sections 3 and 4, which explain the steps in detail.

2.1 Create user and group

In this section you create a Linux group and an user who will use network emulation in ns-2. You have to execute the following commands as *root*:

1. Create a user that will run the virtual machines and make it member of a special group:

   ```
   [root@host]# groupadd tun
   [root@host]# useradd umluser -G tun
   ```

2. If the device `/dev/net/tun` (in some systems `/dev/tun`) does not exist on your system, you have to create it. You can use the following commands:

   ```
   [root@host]# mkdir /dev/net
   [root@host]# mknod /dev/net/tun c 10 200
   ```

3. Provide the group `tun` read/write access the device `/dev/net/tun` (in some systems `/dev/tun`):

   ```
   [root@host]# chown :tun /dev/net/tun
   [root@host]# chmod g+rw /dev/net/tun
   ```

2.2 Download

1. Download and extract our ns-2 patch and sample scripts from the web page of the project http://www-ivs.cs.uni-magdeburg.de/eukneu/forschung/projekte/nse/ . They are contained in the file `ns2emulation.tgz`.

   ```
   [umluser@host]$ mkdir ~/ns2uml
   [umluser@host]$ cd ~/ns2uml
   [umluser@host]$ wget http://www-ivs.cs.uni-magdeburg.de/eukneu/\
                    forschung/projekte/nse/ns2emulation.tgz
   [umluser@host]$ tar xzf ns2emulation.tgz
   ```

2. Download the User-Mode-Linux kernel and file system from their web page: `http://user-mode-linux.sf.net/`. Point your browser to these links and select your mirror to download the kernel and the file system:

   - `http://prdownloads.sf.net/user-mode-linux/linux-2.4.19-5.bz2`
   - `http://prdownloads.sf.net/user-mode-linux/root_fs_slack8.1.bz2`

3. Extract the kernel and the root file system:

```
[umluser@host]$ bunzip2 -k linux-2.4.19-5.bz2
[umluser@host]$ chmod +x linux-2.4.19-5
[umluser@host]$ ln -s linux-2.4.19-5 linux
[umluser@host]$ bunzip2 -k root_fs_slack8.1.bz2
[umluser@host]$ chmod 444 root_fs_slack8.1
```

2.3 Virtual Network

The following commands initialise two Ethernet TAP devices, start and setup two virtual machines. If you encounter any problems in this step, please refer to section 3 of the document:

1. Setup the virtual network (*root* privileges required):

   ```
   [umluser@host]$ su
   [root@host]# ./ns2emulation/virtnet init 2
   [root@host]# exit
   ```

   `virtnet` is a bash script and you might need to edit it and specify paths to several commands.

2. Start two virtual machines in *two different terminals* and login as *root* (no password required):

   Terminal 1:

   ```
   [umluser@host]$ ./linux ubd0=cow_uml0,root_fs_slack8.1 eth0=tuntap,tap0
   root@darkstar:~# hostname uml0
   ```

   Terminal 2:

   ```
   [umluser@host]$ ./linux ubd0=cow_uml1,root_fs_slack8.1 eth0=tuntap,tap1
   root@darkstar:~# hostname uml1
   ```

   If the `linux` binary does not run on your system, you will have to compile it from source or install an rpm package. Please consider the User-Mode-Linux web page for details `http://user-mode-linux.sf.net/`.

3. Now assign IP addresses inside the virtual machines (*root* privileges inside the virtual machines):

   Terminal 1:

   ```
   root@uml0:~# ifconfig eth0 10.0.0.2 up
   ```

   Terminal 2:

   ```
   root@uml1:~# ifconfig eth0 10.0.0.3 up
   ```

4. (Optional) Test the communication between the virtual machines.

   If you configure a Linux Ethernet bridge to connect the two TAP devices, the User-mode Linux machines should be able to communicate directly through the bridge. See Appendix C for details. In this case do not forget to remove the bridge before running ns-2.

2.4 NS-2 Emulation

The following commands introduce the network simulator ns-2 in the network, configured at the previous step. There are two options to run ns-2 with our emulation extensoins: build from source code, or use a Debian package. In the former case you will need a copy of the source code of ns-2, which you can download from the link `http://sourceforge.net/projects/nsnam`.

If you encounter any problems here, please refer to section 4.

Switch to the host machine and execute the following commands:

1. Obtain ns-2 by one of the two possible ways:

   (a) Build ns-2 from the source code:
   Apply our patch to the ns-2 source code and rebuild it: For a successful compilation you need the `zlib` library and header files [7]. If they are not present in your system, you can download them from the zlib home page `http://www.gzip.org/zlib/`.

   ```
   [umluser@host]$ cd ns-2_directory
   [umluser@host]$ patch -p0 < ~/ns2uml/ns2emulation/ns2emulation.diff
   [umluser@host]$ ./configure
   [umluser@host]$ make clean
   [umluser@host]$ make
   [umluser@host]$ su
   [root@host:~]# ln -s ‘pwd‘/shmlog /usr/bin/shmlog
   ```

   (b) Install a Debian package with ns-2 and the emulation extensions.
   Add the following lines in your `/etc/apt/sources.list` file:

   ```
   deb http://bode.cs.uni-magdeburg.de/~aherms/debian sid ns2
   deb-src http://bode.cs.uni-magdeburg.de/~aherms/debian sid ns2
   ```

   Install the ns-2 by running:

   ```
   [root@host:~]# apt-get update
   [root@host:~]# apt-get install nsemulation
   ```

   Optionally you can get the sources by running:

   ```
   [root@host:~]# apt-get source nsemulation
   ```

2. Start the simulator with the sample setup script that we provide (*root* privileges required):

   ```
   [root@host]# cd ~umluser/ns2uml/ns2emulation
   [root@host]# /path_to_ns-2_directory/nse ns2emulation.tcl
   ```

3. Now switch again to one of the virtual machines and test the connection between them. Now all the traffic should pass through the simulator:

   Terminal 2:

   ```
   root@uml1:~# ping 10.0.0.2
   PING 10.0.0.2 (10.0.0.2): 56 octets data
   64 octets from 10.0.0.2: icmp_seq=0 ttl=64 time=10.9 ms
   64 octets from 10.0.0.2: icmp_seq=1 ttl=64 time=4.8 ms
   64 octets from 10.0.0.2: icmp_seq=2 ttl=64 time=4.9 ms
   64 octets from 10.0.0.2: icmp_seq=3 ttl=64 time=4.4 ms
   --- 10.0.0.2 ping statistics ---
   4 packets transmitted, 4 packets received, 0% packet loss
   round-trip min/avg/max = 4.4/6.2/10.9 ms
   ```

Congratulations! You have successfully configured User-mode Linux virtual machines and used ns-2 to emulate a wireless network among the User-Mode-Linux virtual machines. If you have encountered problems through this procedure, please read sections 3 and 4, which describe the above procedure in detail. Another source of help is the mailing list at http://mail-ivs.cs.uni-magdeburg.de/mailman/listinfo/nse . If you have a question, or a remark, please do not hesitate to contact us.

3 Virtual Network

This section explains how to set up a number of User-Mode-Linux virtual machines and prepare them to be connected by a simulated network through ns-2. An example script, that automates these steps can be found in appendix A.

3.1 Virtual Hosts (UML)

User-Mode-Linux (UML) is an extended Linux kernel, that can run as a virtual machine on a Linux host. It can be assigned a root file system and other physical resources different from the host machine.

A User-Mode-Linux requires at least a kernel and a root file system in order to run. The root file system of an UML is stored in a file on the host system. The User-Mode-Linux kernel uses an `user block device (ubd)`, mapped to this file, to access its root file system. The UML can use the `copy-on-write` feature of the `ubd` driver to share a single file system among a number of virtual machines. In this case the root file system file is treated as read-only and all the writes are performed to a separate `copy-on-write` file for each virtual machine. This is convenient if you want to run several virtual machines with similar configurations.

You can use one of the precompiled kernel images, provided on the UML home page, or compile it by yourself. In the second case you have to use a standard Linux kernel with an UML patch. File system images are also available on the UML website, but you again have the option to create your own one. See the UML web page for more details: `http://user-mode-linux.sf.net`.

There is no need to run the virtual machines with root privileges. However, the user that runs them, should be provided with read/write access to the files, containing the file systems and to the network device files (see below).

3.2 Virtual Ethernet

One of the possible transport mechanisms for User mode Linux is the TAP virtual Ethernet device [4]. Assign a TAP device to each UML and use it to connect the virtual machine to a network. The network in this scenario is emulated by ns-2.

3.2.1 Host Requirements

In order to use the TAP devices for the User mode Linux machines, the TUN/TAP (option `CONFIG_TUN`) driver should be included in the kernel of the host system. In 2.4.26 kernel you can find this option under *"Network device support -> Universal TUN/TAP device driver support"*. If your kernel does not provide you this option you can download a patch from the TUN/TAP home page `http://vtun.sourceforge.net/tun/`. The driver supports kernel series 2.2.x, 2.4.x and 2.6.x.

You need also the `tunctl` tool, which is used to configure TUN/TAP devices. The `uml-utilities` package in the Debian system contains this tool. If your Linux system does not provide you this binary, you can download it from the following link: http://www.user-mode-linux.org/cvs/tools/tunctl/ .

3.2.2 Virtual Interfaces Setup

This section describes how to set up TAP devices for the User-mode Linux machines, and prepare them for network emulation. The commands here should be executed as a privileged user (*root*).

1. For *each* User-Mode-Linux virtual machine do the following:

   (a) Create a TAP device and configure it, so that the user, running the virtual machine, has rights to use it:

   ```
   [root@host:~]# ifname = tunctl -u umluser -b
   ```

   (b) Store the interface name into a file (for later use):

   ```
   [root@host:~]# echo $ifname >> /tmp/active_tap_interfaces
   ```

2. Provide the user that runs the virtual machines read/write access to the device file `/dev/net/tun` (See section 2.1 for details).

3.2.3 Starting the Virtual Machines

Run a virtual machine with the following command:

```
[umluser@host:~]$ linux ubd0=[<cow file>,]<root file system> \
                           eth0=tuntap,<tap device>
```

Here, `linux` is the UML kernel binary, `<root file system>` is the file system image, `<tap device>` is a name of a pre-configured TAP virtual Ethernet device, (e.g. tap0). If you specify a `<cow file>` then the UML uses the file system image as read-only and performs disk writes to this file.

3.2.4 Destroying the Virtual Interfaces

The following commands destroy the virtual network devices, created in section 3.2.2. These commands should be executed after the User-mode Linux machines are stopped. They should also be executed as a privileged user (*root*). The following loop reads the active TAP interface names from the temporary file, brings each interface down and removes it from the kernel:

```
[root@host:~]# for ifname in ‘cat /tmp/active_tap_interfaces‘; do
                    ifconfig $ifname down
                    tunctl -d $ifname
                done
[root@host:~]# rm /tmp/active_tap_interfaces
```

We assume that up to this step you have a fully functional set of virtual machines, assigned to virtual TAP interfaces on the host system. In section 4 we discuss the steps to introduce the network simulator ns-2 into this network, using its emulation facility and our extensions.

At this point you might want to test your configuration, by bridging the virtual interfaces in a Linux bridge[3]. The UMLs should be able to communicate through a bridge on the host system. Guidelines for creating the bridge can be found in Appendix C. If you do this test, do not forget to stop the bridge before running the ns-2. Otherwise you might have duplicated packets.

3.3 Large packets

The IEEE 802.11b standard defines the maximum data size in the data frames as 2312 bytes. The Ethernet on the oder side supports data size only up to 1500 bytes. Since we are using Ethernet devices to transport packets from the applications to the simulator, these packets might be unnecessarily fragmented. Consider for example that the IP layer has to send a 2000 byte packet. If the underlying network is a wireless IEEE 802.11b the packet will be sent without fragmentation. But in our emulation setup it will be fragmented on the Ethernet level into two packets. This will result in two packets, sent through the emulated wireless network.

To avoid this redundant fragmentation we increase the MTU (Maximum Transmission Unit) of the TAP Ethernet interfaces. A reasonable value is 2312 bytes, which is a default setting of the wireless IEEE 802.11 network cards.

Since the TAP is an Ethernet device, the linux kernel does not allow MTUs larger than 1500 bytes. To circumvent this behaviour of the kernel we developed a small patch for it. It allows MTUs up to 3000 bytes for the TAP Ethernet interfaces. The patch is available on the homepage of the project: `http://www-ivs.cs.uni-magdeburg.de/EuK/forschung/projekte/nse/index.shtml`.

If you want to use large packets (larger than Ethernet can handle without fragmentation), you have to do the following:

1. Apply the kernel patch for large MTUs of TAP Ethernet interfaces. This patch should be applied both to the host kernel and to the UML kernel. The patch is tested with linux kernels 2.4.21 and 2.6.8.

```
patch -p1 < tunmtu.diff
make
```

2. Reboot with the new kernel, or load the newly compiled module `tun.o (tun.ko).`
3. Increase the MTU of the TAP interfaces - in the host:

   ```
   ifconfig tap0 mtu 2312
   ```

4. Increase the MTU of the Ethernet interfaces - in the UMLs:

   ```
   ifconfig eth0 mtu 2312
   ```

   It seems that the User mode linux kernel allows large MTUs for the Ethernet interfaces.
5. Increase the maximum packet of the ns-2 emulation agents (it is 1600 bytes by default). In the tcl script add the line:

   ```
   Agent/Tap set maxpkt_ 3100
   ```

4 NS-2 Emulation

The ns-2 emulation feature is used to introduce the simulator into a live network. It can grab packets from a real network, pass them through a simulated network, and then inject them back into the real one. This section shows how to introduce ns-2 in the network, described in the previous section. In this example ns-2 is running on the host system. It acts like a bridge between the virtual Ethernet interfaces (TAP). It reads packets, coming from a source TAP device, passes them through a simulated wireless network and writes them back to the destination TAP device. Each ns-2 node in the simulated network represents an UML (TAP device) virtual machine in the real network.

4.1 Emulation Extensions

The ns-2 emulation facility mainly consists of *network objects* and *tap agents*. The network objects are used to send and receive packets to and from a network. The tap agents are connected to network objects and to ns-2 nodes. They obtain network packets from the network objects and send them through the simulated network. Each tap agent can be connected to at most one network object.

4.1.1 Network Object

The ns-2 network object at the link layer uses the `pcap` library to read packets from a network device. Since at the moment of development (June 2004) the `pcap` library did not support writing packets to a network device, ns-2 provides this facility. However, the implementation is platform dependent and not intended for Linux. So, we implemented a new network object to read and write packets to a network device at the link layer. We use the Linux packet sockets for this purpose.

The network object for link layer access is called `Network/Raw`. It supports the commands `open` and `close` with the following syntax:

```
<Network Object> open <Interface Name> [readwrite | readonly | writeonly]
```

```
<Network Object> close
```

You can use it for example in the following way:

```
     set raw0 [new Network/Raw]
     $raw0 open tap0 readwrite
     [...]
     $raw0 close
```

4.1.2 Tap Agent

Every ns-2 node in our simulation represents a virtual machine (User-mode Linux). So, we need address mapping between virtual machines and ns-2 nodes.

The new tap agent is called `Agent/Tap/Raw` and it is a descendant of `Agent/Tap`. It implements additionally address mapping between MAC addresses of the virtual machines and IP addresses of ns-2 nodes. Each `Agent/Tap/Raw` is assigned to one virtual network interface via a network object Network/Raw.

The only difference in using `Agent/Tap/Raw` is that at the creation you have to specify the MAC address of the machine that the agent corresponds to. For example:

```
set a0 [new Agent/Tap/Raw "FE:FD:0A:00:00:02"]
```

In this case the tap agent registers in a common mapping table that it is responsible for receiving frames with the specified MAC address. The tap agents then use this mapping table to determine the destination network address within the simulation.

The rest of the commands are the same as the ones for `Agent/Tap`. For example:

```
$a0 network $raw0
$ns attach-agent $node_(1) $a0
```

These two commands attach the network object `$raw0` to the tap agent `$a0`, and attach the agent `$a0` to the ns-2 node `$node_(1)`.

4.2 Real-Time Scheduler

The emulation facility of ns-2 should be used together with the real-time scheduler. This scheduler synchronises the simulation virtual clock with the system time and tries to dispatch events at actual moments in time. In this way it ensures that the packets, passing the simulated network, are delayed a proper amount of time.

The real-time scheduler however introduces a problem of *delayed* execution of events. The delays may accumulate during simulation and lead to strange behaviour of the network protocols inside ns-2. One effect is for example the change of chronological order of ns-2 events, which leads to false performance of the IEEE 802.11 protocol.

We developed several techniques to increase the real-time performance of ns-2 and they are described in the paper [6]. These are mainly changes to the real-time scheduler - in synchronisation with the system time and in execution of events. The real-time scheduler in ns-2 is used with the commands:

```
set ns [new Simulator]
$ns use-scheduler RealTime
```

4.3 Trace Extensions

The trace subsystem in ns-2 is used to store event information in log files. The current implementation buffers the trace data in main memory and then performs disk writes at certain periods. These writes block the simulator and may cause serious delays in execution of events in real-time mode.

To improve the real-time performance of ns-2 we developed a new trace subsystem. Here the simulator process only stores trace data in main memory and another low priority process empties the buffer when the simulator has nothing to do. To use the main memory efficiently and reduce disk writes in the system during simulation we perform in-memory compression of the trace file with the `zlib` library [7]. In this case the output of the simulator is a file in `gzip` format.

In order to use this trace system, first you have to specify a standard trace file with the `trace-all` command. Then create an instance of the `BaseTrace/ShmGZ` object to redirect the output from the standard file to the new trace system. The constructor of the `BaseTrace/ShmGZ` object has the following syntax:

```
new BaseTrace/ShmGZ <File Name> <Shm Size> <Max Line> <Compressed Buffer>
```

- `<File Name>` is the output file name
- `<Shm Size>` determines the number of trace lines which can be stored in the buffer between the simulator and the logging process. This number is computed as $2^{<ShmSize>}$. For example <Shm Size> of 10 determines a buffer which can store up to 1024 trace lines.
- `<Max Line>` is the expected maximum length of a line in the ns-2 trace file.
- `<Compressed Buffer>` is the size of memory in bytes, used to buffer the compressed output file. If you use for example 104857600 (100MB), you could easily fit up to 1GB text information there, because the trace files contain many similar patterns and the `zlib` library compresses them significantly.

For example you can use this trace system by these commands:

```
set tracefd [open dummy.tr w]
$ns trace-all $tracefd
set tr0 [new BaseTrace/ShmGZ output.tr.gz 16 400 104857600]
```

Here we first create a dummy trace file and point all the trace data to it. After that we create a `BaseTrace/ShmGZ` object, which redirects the output to the compressed file `output.tr.gz`. The trace object here can hold up to 65536 lines of maximum 400 length in the buffer between the simulator and the low priority log process. The buffer for the in-memory compressed file is 100MB.

Note: The trace system uses IPC shared memory for communication between the simulator and the logger. The requested shared memory segment in the above example is 25MB and most systems will require special privileges(root) for this allocation.

4.4 Sample NS-2 Emulation Script

Appendix B contains an example ns-2 emulation script. It creates a wireless ad-hoc network with four nodes inside. It uses the ns-2 extensions, described in this section. It uses a network object to access the bridge device `br0` and a tap agent for each node in ns-2 to represent a virtual machine.

References

[1] The Network Simulator - ns-2 `http://sourceforge.net/projects/nsnam`

[2] The User-mode Linux Kernel Home Page `http://user-mode-linux.sf.net/`

[3] Linux Ethernet bridging `http://bridge.sf.net/`

[4] Universal TUN/TAP driver `http://vtun.sf.net/tun/`

[5] ebtables home page `http://ebtables.sf.net/`

[6] Daniel Mahrenholz, Svilen Ivanov. Real-time Network Emulation with ns-2. In *8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. IEEE Computer Society, 2004.

[7] Zlib home site: `http://www.gzip.org/zlib/`

A Virtual Network Setup Script

This script can be found as a text file at: `http://ivs.cs.uni-magdeburg.de/EuK/forschung/projekte/nse/ns2emulation.tgz`.

```
#!/bin/bash

# Initialization script for a set of TAP devices
# MUST be executed by a super user (root)

#Paths to different commands
IFCONFIG=/sbin/ifconfig #the standard ifconfig tool
TUNCTL=/usr/sbin/tunctl #configure TUN/TAP devices

#Script variables

# Name of user allowed to use the created TAP interfaces (Read / Write)
# -> It is the user that runs the User-mode Linux virtual machines
UNAME=svilen

#Log file for the created tap interfaces
TAPS=/tmp/taps.log

# Creates the virtual TAP interfaces
init_system () {
    echo "Number of interfaces: $1"
    rm -f $TAPS
    for i in ‘seq 1 $1‘; do
        ifname=‘$TUNCTL -u $UNAME -b‘
        echo $ifname >> $TAPS
        $IFCONFIG $ifname up
    done
}

#Destroys the virtual TAP interfaces
reset_system () {
    for i in ‘cat $TAPS 2> /dev/null‘; do
        $IFCONFIG $i down
        $TUNCTL -d $i
    done
}

case $1 in
    init)
        echo "Initializing TAP interfaces"
        init_system $2
    ;;
    reset)
        echo "Reseting TAP interfaces"
        reset_system
    ;;
    *)
        echo "$0 <init <# of interfaces> | reset>"
esac
```

B NS-2 Emulation Script

This script can be found as a text file at: `http://ivs.cs.uni-magdeburg.de/EuK/forschung/projekte/nse/ns2emulation.tgz`.

```
#! /bin/nse
# An example script for the usage of ns-2 in emulation mode:
# Uses:
# 1. Network/Raw agents to access to a network device at level II
# 2. Tap/Raw agents to map between real MAC addresses
# and NS-2 IP addresses
# 3. In-memory compression of the trace file to reduce
# disk write operations (currently done via gzip)

# Common variables
#
set scriptname          routingdemo
set val(chan)           Channel/WirelessChannel    ;# Channel Type
set val(prop)           Propagation/TwoRayGround   ;# Propagation
set val(netif)          Phy/WirelessPhy            ;# Interface
set val(mac)            Mac/802_11                 ;# MAC type
set val(ifq)            Queue/DropTail/PriQueue    ;# Queue type
set val(ll)             LL                         ;# link layer
set val(ant)            Antenna/OmniAntenna        ;# Antenna
set val(ifqlen)         50                         ;# IFQ size
set val(x)              800                        ;# x range [m]
set val(y)              600                        ;# y range [m]
#set val(rp)            AODV                       ;# Routing
set val(rp)             DumbAgent                  ;# Routing
set val(nn)             4                          ;# number nodes
#set val(stime)         60.0                       ;# Sim. time
set val(stime)          360000.0                   ;# Sim. time

set ns          [new Simulator]
$ns use-scheduler RealTime

set tracefd  [open "|gzip > $scriptname.tr.gz" w]
$ns trace-all $tracefd
#set tr0 [new BaseTrace/ShmGZ test.tr.gz 16 400 104857600]

set namtrace [open "|gzip > $scriptname.nam.gz" w]
#set namtrace [open "|nam -r 0.1 -" w]
$ns namtrace-all-wireless $namtrace $val(x) $val(y)

#Procedure needed when running nam in real-time
proc NamTime {} {
        #Send time to nam periodically
        global ns namtrace
        set now [$ns now]
        set next [expr $now + 0.05]
        puts $namtrace "T -t $now"
        flush $namtrace
        $ns at $next "NamTime"
}

#$ns at 1.0 "NamTime"

proc UniformErr {} {
    set err [new ErrorModel]
    $err unit packet
    $err set rate_ 0.01
    $err ranvar [new RandomVariable/Uniform]
    $err drop-target [new Agent/Null]
    return $err
}

set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)

# Create GOD
create-god $val(nn)
# Create channel
set chan_1_ [new $val(chan)]

# Configure node parameters
$ns node-config -adhocRouting $val(rp) \
        -llType $val(ll) \
        -macType $val(mac) \
        -ifqType $val(ifq) \
        -ifqLen $val(ifqlen) \
        -antType $val(ant) \
        -propType $val(prop) \
        -phyType $val(netif) \
        -topoInstance $topo \
        -agentTrace OFF \
        -routerTrace OFF \
        -macTrace ON \
        -movementTrace OFF \
        -channel $chan_1_ \
        -IncomingErrProc UniformErr

#Procedure to configure an ns-2 node initially
proc setup_node {id x y z color} {
        global ns node_
        set node_($id) [$ns node]
        $node_($id) set X_ $x
        $node_($id) set Y_ $y
        $node_($id) set Z_ $z
        $node_($id) color $color
        $ns at 0 "$node_($id) setdest $x $y 0"
        $ns at 0 "$node_($id) color $color"
        $node_($id) random-motion 0
}

setup_node 1 100 300 0 "black"
setup_node 2 300 300 0 "green"
setup_node 3 500 300 0 "blue"
setup_node 4 700 300 0 "cyan"

for {set i 1} {$i <= $val(nn)} {incr i} {
        $ns at 0 "$node_($i) start";
        $ns at $val(stime) "$node_($i) reset";
}

#Network objects to access the TAP devices at the link layer
set raw1 [new Network/Raw]
set raw2 [new Network/Raw]
set raw3 [new Network/Raw]
set raw4 [new Network/Raw]

$raw1 open tap0 readwrite
$raw2 open tap1 readwrite
$raw3 open tap2 readwrite
$raw4 open tap3 readwrite

#Tap Agent for each node
Agent/Tap set maxpkt_ 3100
set a1 [new Agent/Tap/Raw "FE:FD:C0:A8:2F:01"]
set a2 [new Agent/Tap/Raw "FE:FD:C0:A8:2F:02"]
set a3 [new Agent/Tap/Raw "FE:FD:C0:A8:2F:03"]
set a4 [new Agent/Tap/Raw "FE:FD:C0:A8:2F:04"]

puts "install nets into taps..."

#Assign network objects to TAP agents
$a1 network $raw1
$a2 network $raw2
$a3 network $raw3
$a4 network $raw4
#Assign TAP agents to ns-2 nodes
```

```
$ns attach-agent $node_(1) $a1
$ns attach-agent $node_(2) $a2
$ns attach-agent $node_(3) $a3
$ns attach-agent $node_(4) $a4

$ns at $val(stime) "stop"
$ns at $val(stime) "puts \"NS EXITING ...\" ; $ns halt"

proc stop {} {
        global ns tracefd raw1 raw2 raw3 raw4
        $ns flush-trace
        close $tracefd
        $raw1 close
        $raw2 close
        $raw3 close
        $raw4 close
}

puts "okey"

$ns run
```

C Virtual Ethernet setup

This section describes how to set up a Linux Ethernet bridge and use it to connect the virtual machines. The commands here should be executed as a privileged user (*root*).

1. Create an Ethernet bridge (in this example STP is not needed, so turn it off):

   ```
   [root@host:~]# brctl addbr br0
   [root@host:~]# brctl stp br0 off
   ```

2. For *each* User-Mode-Linux virtual machine do the following (you have to know the name of the TAP device, which is associated to the UML):

   Add the TAP device of the UML to the bridge:

   ```
   [root@host:~]# brctl addif br0 <TAP interface>
   ```

3. Provide the user that runs the virtual machines read/write access to the device file `/dev/net/tun` (See section 2.1 for details).

4. Bring up the bridge interface:

   ```
   [root@host:~]# ifconfig br0 up
   ```

5. (Optional) If you want to access the virtual machines from the host through the bridge, assign an IP address to the bridge interface

```
[root@host:~]# ifconfig br0 10.0.0.1 netmask 255.0.0.0 \
                 broadcast 10.255.255.255
```

In this case the IP addresses that you use for the virtual machines should be from the network `10.0.0.0/8`.

6. In order to stop the bridge, do the following:

```
[root@host:~]# ifconfig br0 down
[root@host:~]# brctl delbr br0
```