



# **NetEmulator**

## **User Manual**

Version 1.1

## Table of Contents

<b>License Agreement</b> .....	<b>3</b>
<b>Overview</b> .....	<b>4</b>
The NetEmulator.....	4
<b>Requirements &amp; Installation</b> .....	<b>6</b>
Requirements .....	6
Installation Instructions .....	6
<b>Modeling a Network</b> .....	<b>8</b>
Example Scenario File.....	10
<b>A Quick Start to Using the NetEmulator Tools</b> ....	<b>12</b>
NetEmulator PC Configuration .....	12
Launching the NetEmulator Core Tools .....	13
Launching the NetEmulator GUI Tools.....	14
<b>Media Models</b> .....	<b>18</b>
Provisioned and Slotted Aloha Access Models .....	18
CSMA Access Model.....	21
<b>Man Pages</b> .....	<b>22</b>
netemu-arbitrator .....	22
netemu-bridge.....	24
netemu-controller.....	26
netemu-hist.....	28
netemu-monitor.....	29
netemu-updater .....	30

# License Agreement

*The NetEmulator software is subject to the terms of the following license agreement.*

---

## NetEmulator License Agreement

Copyright (c) 2004, The MITRE Corporation (<http://www.mitre.org>). All Rights Reserved.

The terms "MITRE" and "The MITRE Corporation" are trademarks of The MITRE Corporation and must not be used to endorse or promote products derived from this software or in redistribution of this software in any form.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

You accept this software on the condition that you indemnify and hold harmless MITRE, its Board of Trustees, officers, agents, and employees, from any and all liability or damages to third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this software irrespective of the cause of said liability.

The export from the United States or the subsequent reexport of this software is subject to compliance with United States export control and munitions control restrictions. You agree that in the event you seek to export this software you assume full responsibility for obtaining all necessary export licenses and approvals and for assuring compliance with applicable reexport restrictions.

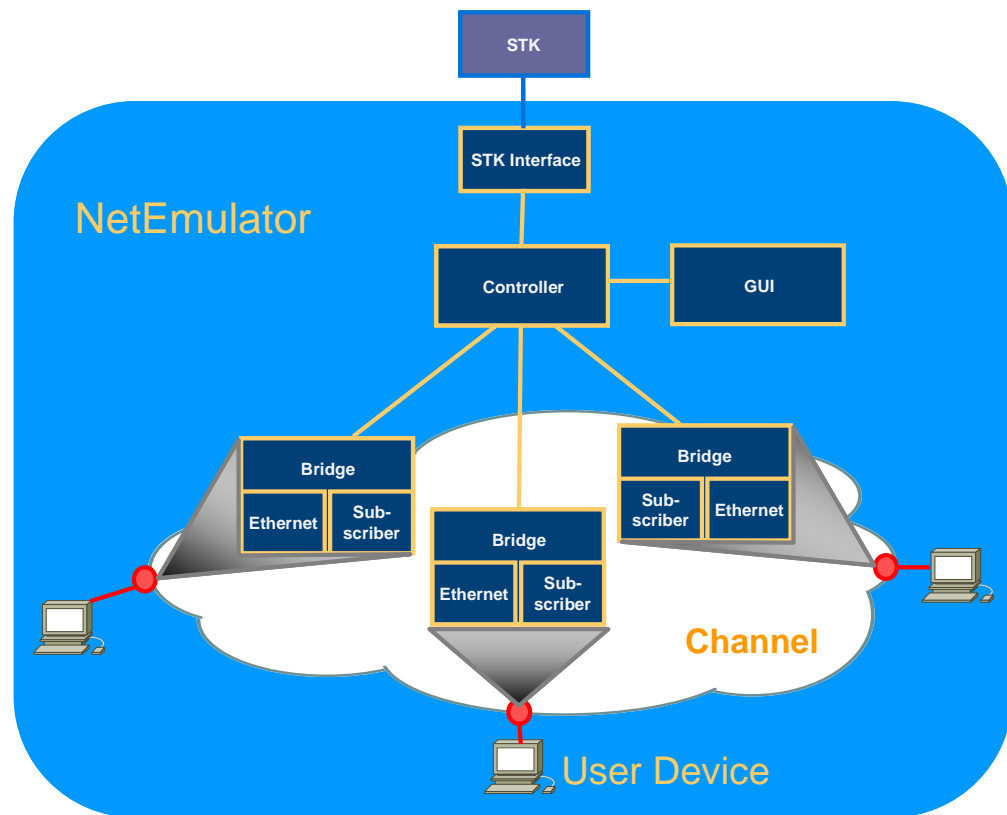
# Overview

*This section provides a brief overview of the NetEmulator and its main components.*

## The NetEmulator

The NetEmulator is a set of Linux based software components that emulate network/link level effects enabling users to test the performance of applications and network protocols and services in a realistic yet synthetic environment. Network effects like packet corruption due to high bit error rates, throughput restrictions due to data rate limitations, and delay due to propagation and queuing are modeled. The NetEmulator is comprised of a few building blocks that together provide the ability to emulate arbitrary network topologies.

The NetEmulator's building blocks are implemented as separate Linux application programs; please refer to Figure 1.



*Figure 1 – The NetEmulator Components*

User devices, like hosts, routers, firewalls, etc., plug-in to the NetEmulator through standard Ethernet interfaces. The NetEmulator is transparent to the user devices and appears like an Ethernet LAN with possibly much worse delay, reliability, and capacity. This approach requires no changes to the user devices/applications plugging into the NetEmulator.

Ethernet frames transmitted between user devices are received and forwarded via “bridge” applications that emulate communication devices such as radios. The “bridges” or more precisely, the subscribers within the “bridge” application, impose network/link level effects, like delay, corruption, and throughput restrictions. The parameters that define these effects are supplied to the “bridge” applications periodically by the “controller”, thus allowing emulated effects to dynamically change with time. The “controller” application provides a single point of control for all the emulated radios (ie. “bridges”) participating in the emulation.

Models for three different types of layer 2/1 media types are provided, and, because it's Open Source, the models can be customized and new models created. The provided models, briefly described below, are generalized abstractions of common classes of wireless media access types:

- The "Provisioned" access model emulates traditional FDMA/TDMA systems in which each node is allocated a fixed amount of bandwidth.
- The "Csma" access model emulates shared contention based channels in which nodes compete against each other for the wireless channel using carrier sensing; this model includes a configurable exponential backoff algorithm and retransmission scheme.
- The "Slotted Aloha" access model emulates shared contention based channels in which nodes probabilistically choose a slot for transmitting data.

To accommodate network scenarios in which network nodes may be mobile, resulting in time varying link qualities, the NetEmulator includes an application (“ifstk”) which interfaces to the Satellite Tool Kit (STK) a third party software tool developed by Analytical Graphics, Inc. (AGI). This allows the NetEmulator to accurately emulate changes in link delay and reliability based upon mobility scenarios defined within STK.

The NetEmulator provides a few different graphical user interface applications for controlling and monitoring emulations. The ‘netemu-updater’ tool enables a user to interactively change delay and bit error rate (BER) on a per link basis, and change the transmit data rate on a per sender basis. The ‘netemu-monitor’ tool allows the current delay, BER, and datarate settings to be viewed. And, the ‘netemu-hist’ tool provides a recording capability that allows the delay, BER, and datarate settings, specified either manually via the ‘netemu-updater’ tool or automatically via an external STK simulation, to be captured to a file and later played back; this enables network emulations to be easily repeated.

At this point, we’ve really just scratched the surface. Each one of the tools mentioned in this section is described more thoroughly later on this manual. Before we get there though, in the next section we discuss how to install the NetEmulator software and system requirements for operating the software.

# Requirements & Installation

*This section describes requirements that users should address prior to installing, followed by directions on how to install the NetEmulator software tools.*

## Requirements

The NetEmulator is comprised of a number of Linux based applications. These applications should run on most Linux systems that have a 2.4 kernel or newer. However, the preferred kernel version for the NetEmulator is 2.6 and higher. The 2.6 Linux kernel has a 1000 Hz system clock, as opposed to the 2.4 kernel which has a 100 Hz system; the result is that with the 2.6 kernel, user processes can be scheduled with a granularity of 1 millisecond rather than 10 milliseconds. For the NetEmulator, most folks will want to have the higher precision provided by the 2.6 kernel.

We also recommend using the Fedora Core 2 Linux distribution; the 2.6 kernel is a standard component of Fedore Core 2. For compiling the NetEmulator, gcc-3.2 or newer is required; this is also a standard component in Fedore Core 2.

The various GUI tools (eg. 'netemu-updater', 'netemu-monitor', 'netemu-hist') are Qt based applications. In order to compile and run them, Qt must also be installed.

The STK Interface tools require a couple files that are only available by contacting Analytical Graphics Inc., the company that markets STK. For details on installing and using the STK Interface, please refer to the "NetEmulator STK Interface Manual".

## Installation Instructions

The NetEmulator applications are grouped into one of three categories: core, qt, and stkif. The "core" category includes those applications which are fundamental to the operation of the NetEmulator and include the 'netemu-controller' and 'netemu-bridge'. The "qt" category includes the GUI applications used to control and monitor emulations, namely 'netemu-updater', 'netemu-monitor', and 'netemu-hist'. The "stkif" category includes the STK Interface tools 'netemu-ifstk' and 'netemu-mkscenario', which communicate with a remote instance of STK.

1. If the NetEmulator source files do not already reside on your machine, obtain the latest NetEmulator source distribution, eg. "netemulator-1.0.tar.gz", and unpack it into a desired directory. For example:

- `cd /usr/src`

- `tar -xzvf netemulator-1.0.tar.gz`

2. If the source files have not been previously built, compile them:

- `cd netemulator-1.0`

- `make depends`

- `make`

3. If you'd like to install the compiled programs into a system directory, edit the file "netemulator-1.0/Makefile" and modify the "bindir" and "mandir" variables so that they point to the desired system directories, eg. `bindir=/usr/bin` and `mandir=/usr/man`. Then:

- `make install`

After successfully compiling the code, and optionally installing it, the NetEmulator is ready to run. In the next couple sections we'll describe how to define a NetEmulator scenario and then we'll walk through an example of launching all the various tools and conducting a simple network emulation.

# Modeling a Network

*This section describes how the NetEmulator models a communication network and how that model is represented using a NetEmulator scenario file.*

There are four basic types of abstractions that the NetEmulator uses to model a communication network. These elements are: channels, entities, radios, and subscribers. We'll talk more about these and then show how they can be represented with a NetEmulator scenario file.

An **entity** is basically a vehicle, facility, or other large object that occupies a location and contains one or more radios. An entity may or may not be mobile. The key idea here is that an entity provides a way of collecting together radios that in the real world would be co-located.

In the NetEmulator, a **radio** exposes a unique Ethernet interface to provide a way for external user devices (eg. hosts and routers) to "plug in" to the distributed emulation. A radio, in general, may communicate with other radios on multiple channels simultaneously and the attributes of the channels may differ. To accommodate this possibility, a radio may have one or more subscribers and each subscriber attaches to exactly one channel.

A **subscriber** emulates channel delay and corruption effects, as well as throughput constraints of communications between radios. There are currently three types of subscribers that model different abstract media types. These types were briefly described in the Introduction section and include "Provisioned", "Csma", and "Slotted Aloha". For more details on these models, see the "Media Models" section later in this document.

A **channel** logically connects two or more subscribers. Subscribers attached to the same channel can communicate; subscribers not connected to the same channel can not communicate.

A radio (also referred to in this document as a bridge) bridges Ethernet frames it receives from an external user device on its Ethernet port to one or more of its subscribers. A subscriber imposes datarate limitations on a frame prior to transmission, then encapsulates the Ethernet frame in a UDP multicast packet addressed to a multicast address and port specific to a channel, and transmits. Any subscribers attached to the same channel (ie. bound to the same UDP multicast address and port) will receive the UDP multicast packet, decapsulate it, impose delay and corruption, and, if not corrupted, the radio will deliver the frame to the external destination user device. Each channel utilizes a unique UDP multicast port, thus segregating traffic that occupy different channels and preventing subscribers on different channels from communicating.

The radios in the distributed emulation are transparent to the external user devices. The user devices perceive that they are connected to the same local area network, albeit one that may have a delay, reliability, and datarate that are much worse than those of a wired network. This transparency allows the performance of host applications and network protocols and services to be evaluated in a realistic wireless environment, without requiring any modifications to the end hosts or routers.

In order to emulate a specific network scenario, the user must first capture the entities, radios, subscribers, and channels that will be emulated in a scenario file.

To define a channel in the scenario file, the following syntax is used:

```
Channel {
    CID=<cid>
    type=<type>
    slots=<aloha slots per second>
}
```

where <cid> is the channel's unique ID and <type> is the channel type. At present, three channel types are currently supported: 'Provisioned', 'Csma' and 'SlottedAloha'. In the future, more channel types may exist. More than one channel may be defined in the scenario file though all channels must have different ID's. ID's are alphanumeric strings (ie. may contain A-Z, a-z, 0-9) and may also include the underscore character: \_. The 'slots' entry is only used for SlottedAloha channels, in which case it specifies how many slots occur each second.

To define an entity (in the example, the entity has a single radio subscribed to a single channel. Additional radios may be contained in the entity definition, and additional subscribers may be contained in the radio definition):

```
Entity {
    EID=<eid>
    x=<xpos>
    y=<ypos>
    z=<zpos>
    Radio {
        RID=<rid>
        Subscriber {
            channelId=<cid>
            power=<power>
            datarate=<datarate>
            kind={TxRx | TxOnly | RxOnly}
        }
        ...Additional Subscriber's may follow
    }
    ...Additional Radio's may follow
}
```

where <eid> is the entity's unique identifier. <xpos>, <ypos>, and <zpos> are the coordinates of the entity in an unspecified coordinate system.

The Entity definition may contain zero or more Radio definitions. For a Radio definition, <rid> is the radio's unique identifier and must be composed of its enclosing entity's <eid> field concatenated with a hyphen, concatenated with an additional alphanumeric string describing the radio. The example scenario file below should help clarify how the <rid> field is formed.

The Radio definition may contain zero or more Subscriber definitions. For a Subscriber definition, the <cid> field must match exactly with the <cid> field in a previously defined Channel. <power> is in unspecified dimensions and can assume any floating point value. <datarate> specifies the subscriber's transmission datarate in bits per second. The kind attribute indicates whether the Subscriber can both transmit and receive (TxRx), or can only transmit (TxOnly), or can only receive (RxOnly).

All identifiers in the scenario file must only contain alphanumeric characters and underscores; hyphens may also be used but only in specific circumstances which are illustrated below in the example scenario file.

Comment lines in the scenario file are permitted and must begin with a '#' character.

## Example Scenario File

A simple example scenario file is shown below:

```
# Simple scenario
# Two entities, each with a single radio on the same channel
#

Channel {
  CID=prov_c0
  type=Provisioned
}

Entity {
  EID=VehicleA
  x=10.0
  y=20.0
  z=30.0
  Radio {
    RID= VehicleA-r0
    Subscriber {
      channelId=prov_c0
      kind=TxRx
      power=10.0
      datarate=500000.0
    }
  }
}

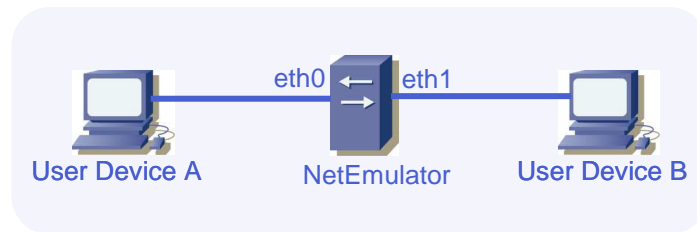
Entity {
  EID=VehicleB
  x=4000.0
  y=5000.0
  z=6000.0
  Radio {
    RID= VehicleB-r0
    Subscriber {
      channelId=prov_c0
      kind=TxRx
      power=10.0
      datarate=500000.0
    }
  }
}
```

In the above example, two entities, namely "VehicleA" and "VehicleB" are defined. Each entity has a single radio and each radio has a single subscriber that is attached to provisioned channel whose channel ID is "prov\_c0". Each subscriber has a transmit data rate of 500kb/s.

# A Quick Start to Using the NetEmulator Tools

*This section provides a quick walkthrough of the steps required to conduct an emulation of a simple network scenario.*

So let's say we want to emulate a simple two node network like the one defined in the example scenario file described in the previous section. In this scenario, there are two entities, each with a single radio, and each radio has a single subscriber that attaches to a provisioned channel. The first step is to make sure we have the physical hardware we need for the emulation. We need a PC with at least two Ethernet NIC's, and we need two user devices; these can be PC's, routers, firewalls, or even a piece of test equipment that can generate Ethernet frames. Given that we have the required hardware, we need to cable it together like in the figure below:



*Figure 2 – A Simple Emulation Setup*

Here we assume that the PC on which we will run the NetEmulator tools has an Ethernet interface 'eth0' that is connected to User Device A, and 'eth1' that is connected to User Device B. Before we launch any NetEmulator tools we need to do a little system administration to configure 'eth0' and 'eth1' and to insure that multicast packets that the NetEmulator tools generate stays local within the NetEmulator PC. On this latter point, we'll add a route to the forwarding table that directs NetEmulator multicast traffic to the system's loopback interface.

## NetEmulator PC Configuration

1. Login to the NetEmulator PC as root
2. Use the "ifconfig" utility to configure "eth0" and "eth1" as simple Ethernet interfaces with no IP addresses
  - `ifconfig eth0 up`
  - `ifconfig eth1 up`
3. Add a multicast route for the multicast address used by the NetEmulator that direct packets to the loopback interface:
  - `route add -net 225.0.0.0 netmask 255.0.0.0 lo`

## Launching the NetEmulator Core Tools

Now that we have taken care of the system administration tasks, we are ready to run the core NetEmulator tools. The first step is to launch the “netemu-controller” application which acts as a controller for the distributed emulation (refer to the Introduction and Figure 1 for the high level architecture of the NetEmulator). The netemu-controller requires three command line (refer to the “Running the Tools” section later in this manual for a detailed description of the required and optional command line arguments). The first required argument is the name of the scenario file; let’s assume the example scenario file is named “sc.ini”. The second argument is the IP address and TCP port number on which the netemu-controller will listen for connection requests from the various emulated radios. Since the emulated radios all reside on this same machine, we only have to specify the port number; let’s assume the port number is 22222. The third argument is the multicast address and port number on which emulated radios will communicate (actually, the port number is a base or starting port number from which each emulated channel will be assigned a unique and increasing port number...more details on that are in the “Running the Tools” section); let’s assume we’ll use 225.0.0.1:33333.

1. Launch the controller using the assumed values specified above:

```
➤ netemu-controller sc.ini :22222 225.0.0.1:33333
```

Now we need to get the emulated radios up and running. We need to launch an instance of “netemu-bridge” for each of the two emulated radios defined in the scenario. This application has three required command line arguments. The first is the radio ID of the emulated radio; this must match with one of the radio ID’s in the scenario file used by the netemu-controller. The second required arguments is the IP address and port number that the netemu-controller is listening on for connections. The third required argument is the system name of the physical Ethernet interface (eg. eth0) on which the external user device is connected. Using the configuration shown in Figure 2, we have:

2. Launch the emulated radio for User Device A:

```
➤ netemu-bridge VehicleA-r0 :22222 eth0
```

3. Launch the emulated radio for User Device B:

```
➤ netemu-bridge VehicleB-r0 :22222 eth1
```

At this point, any frames transmitted by either User Device A should be received by the NetEmulator and delivered to User Device B, and vice versa, using the emulated channel data rate specified in the scenario file. Since we have not yet specified any delay or corruption to impose on the link between the emulated radios, the default NetEmulator behavior is to deliver all user traffic with no delay and no corruption.

## Launching the NetEmulator GUI Tools

To start imposing delay or corruption, we need to use the graphical user interface tools. In particular, the netemu-updater tool enables us to specify delay and BER on a per link basis; it also allows us to specify the transmit data rate for each emulated radio subscriber. The netemu-updater tool has two required command line arguments: the scenario filename used by the netemu-controller and the IP address/port number on which the netemu-controller listens for connections. Using the values from the prior steps:

1. Launch the updater GUI tool:

```
➤ netemu-updater sc.ini :22222
```

The initial window which will be displayed should look like Figure 3. The tabbed dialog window has two panes: “Link Properties” (which is currently displayed) and “Transmitter Properties”.

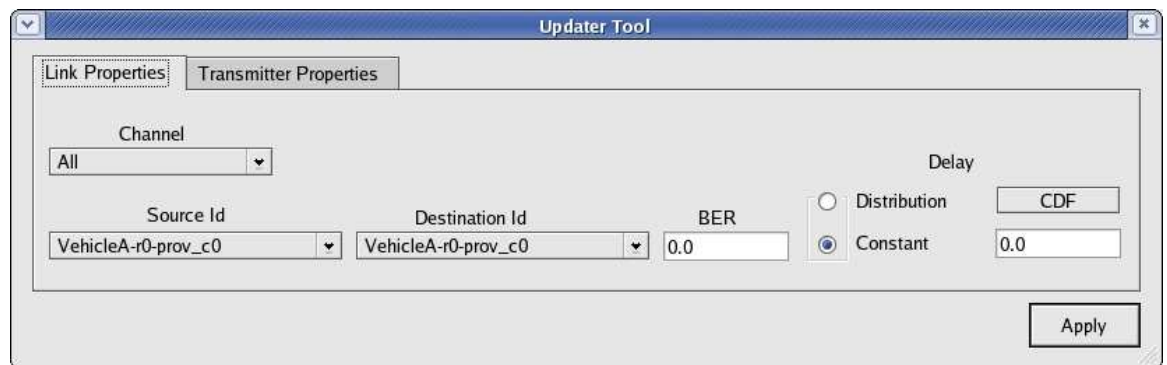


Figure 3 – Initial Updater Window

On the Link Properties panel, you can specify the BER and delay between a source and destination subscriber pair. For example, to specify a delay of 0.5 seconds between the source subscriber VehicleA-r0-prov\_c0 and the destination subscriber VehicleB-r0-prov\_c0, as shown in Figure 4, change the subscriber by clicking on the “Destination Id” widget, then type 0.5 in the Delay text box, then click the “Apply” button. When the Apply button is pushed, a message is sent to the netemu-controller application with the current delay & BER values that are shown.

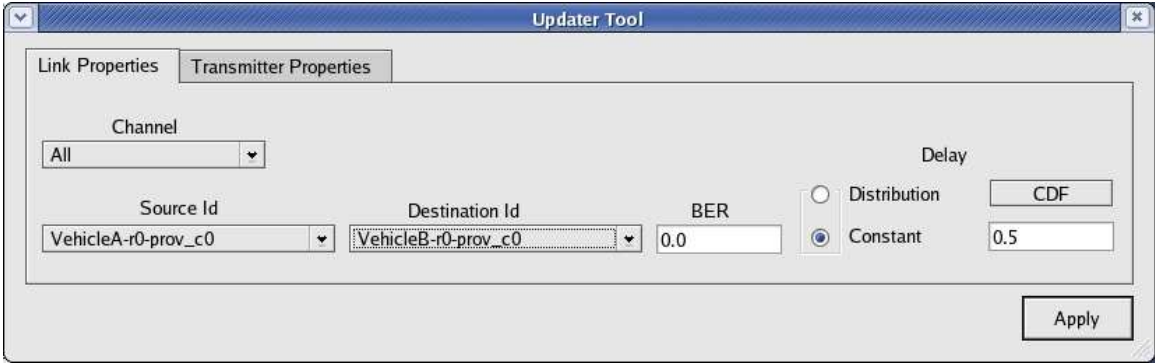


Figure 4 – Specifying Link Delay and BER

To change the transmit data rate of a subscriber, click on the “Transmitter Properties” tab, then select the desired subscriber Id and enter a value in the “Data Rate” entry box. As an example, Figure 5 shows the subscriber VehicleA-r0-prov\_c0 with a transmit rate of 16000 bps. When satisfied with the displayed values, be sure to click the Apply button to generate a message to the netemu-controller to apply the new settings.

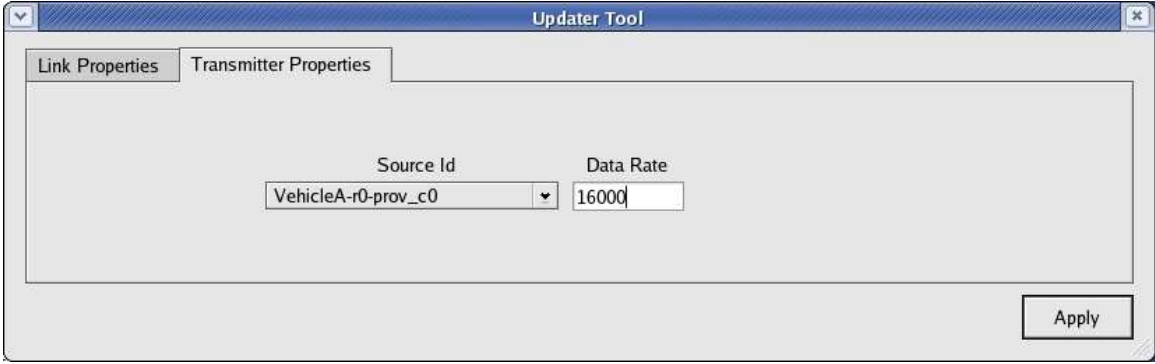


Figure 5 – Specifying Transmit Data Rate

To change the transmit datarate of a subscriber, click on the “Transmitter Properties” tab, then select the desired subscriber Id and enter a value in the “Data Rate” entry

Up to this point, we have been seen how to set the delay and BER on a particular link and how to adjust the transmit data rate of a sender. It's often helpful to also view what the actual emulated settings and to do that we use the netemu-monitor tool. Using the address of the netemu-controller specified previously:

2. Launch the monitor GUI tool:

```
➤ netemu-monitor :22222
```

The Monitor window will have a single menu item, Channels, which will in turn have a menu item for each channel in the emulation. Click on the Channels menu and you should see something similar to Figure 6:

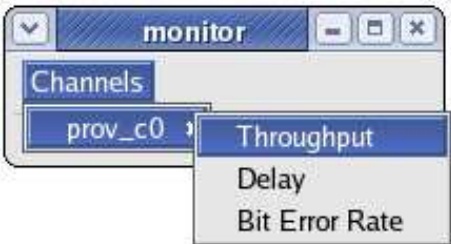
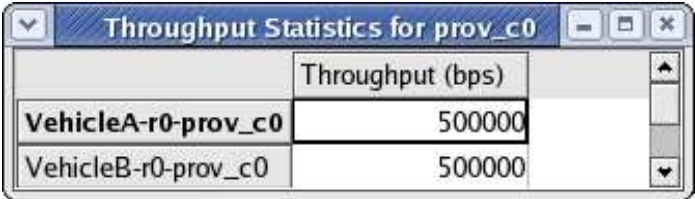


Figure 6 – Monitor Window and Channel Menu

As you can see, you get the option of viewing data rate (ie. Throughput), delay, and BER. For example if you select Throughput, you should see a new window similar to Figure 7 appear:



	Throughput (bps)
VehicleA-r0-prov_c0	500000
VehicleB-r0-prov_c0	500000

Figure 7 – Transmit Data Rates

If instead you select Delay, you should see a new window similar to Figure 8 where the link source is listed down the first column and the link destination is listed along the top row:

	VehicleB-r0-prov_c0	VehicleA-r0-prov_c0
VehicleA-r0-prov_c0	Constant:0.5	
VehicleB-r0-prov_c0		Constant:0.1

Figure 8 – Link Delays

And finally, if you select Bit Error Rate, you should see a new window like Figure 9 where the link source is listed down the first column and the link destination is listed along the top row:

	VehicleB-r0-prov_c0	VehicleA-r0-prov_c0
VehicleA-r0-prov_c0	0.0001	
VehicleB-r0-prov_c0		1e-06

Figure 9 – Link BER's

To recap, we have now completed a quick walk through of using the NetEmulator for a simple emulation. We saw that there some system administration steps to take first, and then the NetEmulator core tools are launched, followed by the NetEmulator GUI tools which provide a way of controlling and monitoring the emulation.

In the next section we'll drill down a little bit into the different media access models that the NetEmulator supports.

# Media Models

*This section describes the three different media models, namely Provisioned, Csma, and SlottedAloha, currently supported by the NetEmulator.*

As we saw in prior sections, each emulated channel has an associated channel access model and there are three different types of models: Provisioned, Csma, and SlottedAloha. These models are generalized abstractions of common classes of wireless media access types:

- The "Provisioned" access model emulates traditional FDMA/TDMA systems in which each node is allocated a fixed amount of bandwidth.
- The "Csma" access model emulates shared contention based channels in which nodes compete against each other for the wireless channel using carrier sensing; this model includes a configurable exponential backoff algorithm and retransmission scheme.
- The "Slotted Aloha" access model emulates shared contention based channels in which nodes probabilistically choose a slot for transmitting data.

## Provisioned and Slotted Aloha Access Models

We'll now take a look at the overall structure used in processing user Ethernet frames that is common across the Provisioned and SlottedAloha channel access models. The processing structure is somewhat modified for the Csma access model; we'll cover that structure later in this section.

Figure 10 below, shows the processing of user Ethernet frames by a sender for the Provisioned and Slotted Aloha models.

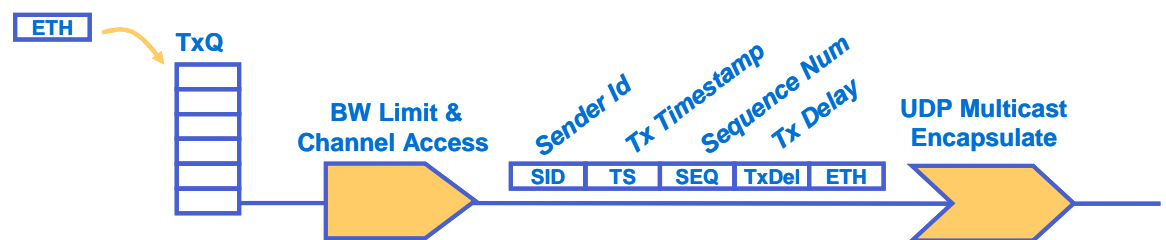


Figure 10 – Sender Processing

Starting from left to right, an Ethernet frame that is received from the user device is placed at the end of a transmit queue where it will await transmission across the emulated channel. The transmit queue is serviced at a rate based upon the channel access model. For the Provisioned access model, where a node has exclusive access to a channel, the service rate of the queue is equal to the sender's emulated data rate. For the Slotted Aloha model, the service rate depends upon not only the sender's emulated data rate but with the additional constraint that frames can only be dequeued at the beginning on an emulated slot. This reduces the effective service rate and the amount of this reduction depends upon the number

of emulated slots per second; the lower the number of emulated slots per second, the lower the service rate.

When the head of the transmit queue is serviced, the dequeued Ethernet frame is prepended with extra information. A sender subscriber Id is added which enables receiving subscribers to apply delay and BER on a per link basis; the sender id also enables a sender to know when it has heard its own transmission so that it can discard it.

An optional timestamp is also prepended to the Ethernet frame. This timestamp is intended to allow emulated radios that are distributed across multiple PC's to subtract the inherent physical LAN delay that exists between senders and receivers. In order for timestamps to be used effectively, all PC's in the distributed emulation must have tightly synchronized system clocks. If timestamps are used, we recommend insuring that clocks are synchronized to within a few microseconds. This likely requires a GPS based hardware solution; the Network Time Protocol (NTP) is probably not accurate enough. Fortunately, we have found that the inherent physical LAN delays between distributed radios are typically small enough that timestamps are not needed. These delays can be further reduced by using Gigabit Ethernet to interconnect PC's.

The sequence number that is prepended to the Ethernet frame enables receiving subscribers to detect inadvertent frame loss. Inadvertent frame loss can occur when a CPU in the distributed emulation is heavily loaded and fails to service NIC hardware before a receive buffer overflows. The NetEmulator, while not able to overcome this problem, is at least able to detect that it has occurred and report it so that the user may investigate the source of loss (eg. trying to emulate too high of a data rate) and make appropriate changes.

The transmit delay prepended to the Ethernet frame indicates how long it takes to modulate the data bits at the emulated data rate. This allows the receiving subscriber to impose accurate delays on received frames.

For SlottedAloha subscribers, the slot number in which the frame is being transmitted is also prepended. This allows receiving subscribers to determine whether a collision occurred by comparing the slot numbers of received frames.

Once the various fields of information have been prepended to the Ethernet frame, the result is then encapsulated in a UDP datagram and sent to the IP multicast address associated with the channel. All subscribers attached to the channel will each receive a copy of the datagram, and this brings us to the next part of our description: receiver processing.

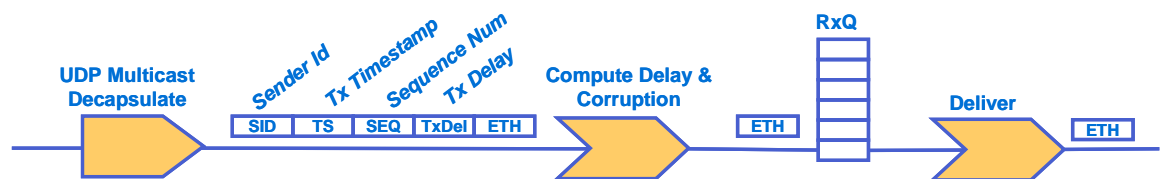


Figure 11 –Receiver Processing

Continuing from left to right, in Figure 11, when a receiver obtains a copy of a UDP datagram, it decapsulates the Ethernet frame along with the information that had been prepended by the sender. Then, using delay and BER tables supplied by the NetEmulator's controller (if you'll recall, in Figures 8 & 9, we showed an example of a delay table and a BER table using the netemu-monitor program), the receiver computes delay and corruption.

To compute delay, the receiver combines a couple different delays to arrive at a total delay. The first delay component is the transmit delay value included in the prepended header. The

second delay component is obtained using the controller provided delay table. Before we continue, we must describe the delay table in more detail.

Each entry in the delay table is a delay distribution. The delay distribution is defined by either a cumulative density function (CDF) for continuously distributed delay values, or a cumulative mass function (CMF) for a discrete distribution. The receiver references the delay distribution in the table associated with the link from the sender to itself, and then obtains from the delay distribution an actual delay value. For many scenarios, it makes sense to have a constant delay distribution for each link; this would satisfactorily model propagation delay. But to accommodate additional sources of delay, such as variable access delay, the NetEmulator generalizes delay as a distribution which the user may define (using the “CDF Editor” feature of the netemu-updater tool).

Once the receiver obtains a delay value from the appropriate delay distribution in the controller provided delay table, it adds to it the transmit delay from the prepended header to arrive at a total delay to impose on the Ethernet frame. The receiver strips off the prepended header and remembers the time at which the Ethernet frame should be delivered to the receiving user device (the SlottedAloha model also remembers the slot number in which the packet was sent).

The receiver then determines if the frame would have been corrupted during reception. Using the controller provided BER table, the receiver looks up the BER value associated with the link from the sender to itself. Then, using this BER value and the length of the received Ethernet frame, the receiver computes the probability that one or more bits would have been received in error. The receiver then flips an appropriately weighted coin to determine whether in fact the frame was corrupted; if so, the frame is discarded. Otherwise, the frame along is placed in a receive queue where it will wait until the computed delivery time. For the SlottedAloha model, an additional check is made to determine if any frames in the receive queue were transmitted in the same slot; if so, a collision is assumed and the frames are removed and discarded.

When the computed delivery time occurs, the frame is dequeued sent out the receiving bridge’s Ethernet port to the external user device.

## CSMA Access Model

The contention based CSMA media access model processes user frames in a somewhat different manner than the Provisioned and Slotted Aloha access models. For a shared access media, it is important to insure that while one node transmits, the other nodes defer from transmitting. The distributed processing used by the Provisioned and Slotted Aloha models makes it difficult to accurately model this kind of contention. For the CSMA model, a centralized "Arbitrator" determines when nodes can access the channel. The "Arbitrator" is yet another software program (namely "netemu-arbitrator") in the NetEmulator tool suite.

When a user frame is received on a bridge's Ethernet port, the sending CSMA subscriber immediately encapsulates it in a UDP datagram and sends it to the Arbitrator. The Arbitrator maintains a queue of packets from each contending subscriber along with a simple state machine for each subscriber. The Arbitrator executes each state machine and determines which node would have transmitted first. It then dequeues the corresponding packet and sends the packet to the multicast address allowing all subscribers on the channel to receive the packet.

In theory, the state machine which models the nodes could use a number of different policies for determining which packet gets access to the channel. For now, we implement a very simple exponential backoff scheme. After every transmission, a node desiring to transmit randomly picks a slot from among a window of slots to begin its transmission. The node must wait until the beginning of its slot before transmitting and may only transmit if no other node has begun transmitting in a prior slot. If another node begins its transmission first, then the node must defer and double its window size, up to a user configurable maximum. After successfully transmitting, a node reduces its window by a half, down to a user configurable minimum. The duration of a slot is user configurable; all slots are the same size. If two nodes choose the same slot to begin their transmission, a collision occurs. In this case, the Arbitrator does not release any packets, it simply waits until the longer of the packets would have completed, increments the retry count for each packet (and discards any which have exceeded the maximum number of retries), then conducts another round to determine who goes first.

An ARQ scheme is also modeled. When a unicast packet is chosen for transmission, the Arbitrator determines if the packet would have been delivered successfully based upon link BER. If not, the packet is not sent and is retried later, up to some maximum number of retries. This means that shared contention subscribers do not have to perform any corruption modeling (on unicast transmissions) since the Arbitrator has already done so.

# Man Pages

For completeness and ease of reference, this section contains the man pages for the GUI and core NetEmulator tools. The STK Interface tools are documented in the separate “STK Interface Manual.”

## netemu-arbitrator

---

### NAME

netemu-arbitrator – NetEmulator arbitrator program for distributed network emulations

### SYNOPSIS

**netemu-arbitrator** [**options**] **<cid>** **<local addr>** **<controller addr>**

supported options, default values are enclosed in []:

--minWindow	<slots>	Minimum possible backoff window	[32]
--maxWindow	<slots>	Maximum possible backoff window	[1024]
--slotDuration	<seconds>	Duration of a backoff slot	[0.0005]
--maxTries	<count>	Max possible transmission attempts for a packet before discarding	[7]
--priority	<level>	Realtime process priority 1-99	[90]
--scheduler	<policy>	Scheduler policy, either SCHED_FIFO or SCHED_RR	[SCHED_RR]
--verbose		Displays copious debug output	

### DESCRIPTION

The **netemu-arbitrator** application is part of the NetEmulator tools package and is used in conjunction with emulated radios using a shared contention based CSMA channel. The **netemu-arbitrator** performs much of the emulated processing normally performed by subscribers for other channel types. A **netemu-arbitrator** provides a central point for resolving contention on a shared (not provisioned) channel. For a provisioned channel, sending subscribers simply encapsulate frames in UDP packets that are sent to a multicast address specifically allocated for the provisioned channel. Other subscribers attached to the channel listen on the multicast address and hear the UDP packets directly. For shared contention channels, however, sending subscribers send their UDP packets to the **netemu-arbitrator** for the channel. The **netemu-arbitrator** resolves contention amongst all sending subscribers. The **netemu-arbitrator** maintains a small queue of packets from each contending subscriber along with a simple state machine for each subscriber. The **netemu-arbitrator** executes each state machine and determines which node would have transmitted first. It then dequeues the corresponding packet and sends the packet to the multicast address allowing all subscribers on the channel to receive the packet.

In theory, the state machines which model the nodes could use a number of different policies for determining which packet gets access to the channel. For now, we implement a very simple exponential backoff scheme. After every transmission, a node desiring to transmit randomly picks a slot from among a window of slots to begin its transmission. The node must wait until the beginning of its slot before transmitting and may only transmit if no other node has begun transmitting in a prior slot. If another node begins its transmission first, then the node must defer and double its window size, up to a user configurable maximum. After successfully transmitting, a node reduces its window by a half, down to a user configurable minimum. The duration of a slot is user configurable; all slots are the same size. If two nodes choose the same slot to begin their transmission, a collision occurs. In this case, the **netemu-arbitrator** does not release any packets, it simply waits until the longer of the packets would have completed then conducts another round to determine who goes first.

An ARQ scheme is also modeled. When a unicast packet is chosen for transmission, the **netemu-arbitrator** determines if the packet would have been delivered successfully based upon link BER. If not, the packet is not sent and is retried later, up to some maximum number of retries. This means that shared contention subscribers do not have to perform any corruption modeling (on unicast transmissions) since the **netemu-arbitrator** has already done so.

## COMMAND LINE ARGUMENTS

<cid> is the channel identifier of the channel for which this arbitrator will operate. Only channels defined in the emulator network scenario file of type "Csma" may be specified.

The <local addr> field indicates the address and port upon which this **netemu-arbitrator** should listen for packets from emulated radios (ie. **netemu-bridge** programs).

The <controller addr> field indicates the hostname/address and port of a currently running **netemu-controller**. The **netemu-controller** must be launched prior to launching any **netemu-arbitrator** applications. The format of the address field allows either a hostname or dotted IP address to be specified, followed by a ":", followed by the port number. If no hostname or IP address is specified, localhost is assumed; a port number, however, must be specified. Some legal examples of addresses are:

## netemu-bridge

---

### NAME

netemu-bridge – a program which emulates a radio within a NetEmulator distributed network emulation

### SYNOPSIS

**netemu-bridge** [**options**] <rid> <controller addr> <ethX>

supported options, default values are enclosed in []:

--priority	<level>	Realtime process priority 1-99	[90]
--scheduler	<policy>	Scheduler policy, either SCHED_FIFO or SCHED_RR	[SCHED_RR]
--verbose		Displays copious debug output	
--inputQueueLen	<frames>	Max number of user frames awaiting transmission, -1 for no limit	[-1]
--flowcontrol		Enable pause frames on ethernet port	
--flowspeed	<speed>	For NIC's which do not support 'ethtool', specify speed of ethernet port (10, 100, 1000)	[100]

### DESCRIPTION

The **netemu-bridge** application is part of the NetEmulator tools package and provides an emulated radio for distributed network emulations. A distributed emulation requires a **netemu-controller** to be present prior to starting any of the emulated radios. The controller reads in a scenario file and parses it to learn about the entities, radios, subscribers, and channels which comprise the emulated network. For more details on scenario files and the operation of the **netemu-controller**, see the **netemu-controller(8)** manpage.

An entity is basically a vehicle, facility, or other object that occupies a location and contains one or more radios.

A radio (or more precisely, a **netemu-bridge**) exposes an Ethernet interface to provide a way for external user devices (eg. hosts and routers) to "plug in" to the distributed emulation. Each bridge requires a separate Ethernet interface. A radio, in general, may communicate with other radios on multiple channels simultaneously and the attributes of the channels may differ. To accommodate this possibility, a radio may have one or more subscribers and each subscriber attaches to exactly one channel.

A subscriber emulates channel delay and corruption effects, as well as throughput constraints of communications between radios.

A channel logically connects two or more subscribers. Subscribers attached to the same channel can communicate; subscribers not connected to the same channel can not communicate.

A radio bridges Ethernet frames received from a user device on its Ethernet port to one or more of its subscribers. A subscriber imposes datarate limitations on a frame prior to transmission, then encapsulates the Ethernet frame in a UDP multicast packet addressed to a multicast address and port specific to a channel, and transmits. Any subscribers attached to the same channel (ie. bound to the same UDP multicast address and port) will receive the UDP multicast packet, deencapsulate it, impose delay and corruption, and, if not corrupted, the radio will deliver the frame to the external destination user device. Each channel utilizes a unique UDP multicast port, thus segregating traffic that occupy different channels and preventing subscribers on different channels from communicating.

The radios in the distributed emulation are transparent to the external user devices. The user devices perceive that they are connected to the same local area network, albeit one that may have a delay, reliability, and datarate that are much worse than those of a wired network. This transparency allows the performance of host applications and network protocols and services to be evaluated in a realistic

wireless environment, without requiring any modifications to the end hosts or routers.

## COMMAND LINE ARGUMENTS

<rid> is the radio identifier for the **netemu-bridge** being launched. The <rid> must match exactly the <rid> field of a radio defined in the scenario file being used by the associated **netemu-controller**.

The <controller addr> field indicates the hostname/address and port of a currently running **netemu-controller**. As mentioned above, the **netemu-controller** must be launched prior to launching any **netemu-bridge** applications. The format of the address field allows either a hostname or dotted IP address to be specified, followed by a ":", followed by the port number. If no hostname or IP address is specified, localhost is assumed; a port number, however, must be specified. Some legal examples of addresses are:

- 192.168.0.1:23456
- :23456
- localhost:23456

<ethX> is the name of the Ethernet device interface on the host machine which will be used by an external user device to "plug in" to this emulated radio.

## FLOW CONTROL

The **netemu-bridge** can optionally employ 802.3 PAUSE frames for flow controlling user devices which send at rates higher than the emulated data rate. The PAUSE frames include a field which tells the user device's NIC how long to pause before resuming sending. This field depends upon the speed of the ethernet link which connects the user device to the netemu-bridge's ethernet port. By default, **netemu-bridge** queries the interface to obtain this value. If the NIC does not support 'ethtool' then this information can not be queried and should be specified using the --flowspeed option, iff the --flowcontrol option is present. The --flowspeed option should specify either 10 (for 10 Mb/s ethernet), 100 (for FastE) or 1000 for (GigE) interfaces.

## THEORY OF OPERATION

When the **netemu-bridge** launches, it contacts the **netemu-controller** to obtain its configuration. Rather than have each **netemu-bridge** have to read the same scenario file, only the **netemu-controller** has knowledge of the details within the scenario file. When a **netemu-bridge** is launched, it is unaware what subscribers it should have; this is provided by contacting the **netemu-controller** which responds with the configurations of each subscriber for the new **netemu-bridge**.

After obtaining its configuration, the **netemu-bridge** is ready to transmit Ethernet frames that it receives from its external user device, and, it is ready to deliver to the external user device any frames it receives from other radios across the emulated channel(s).

## SPECIAL CONSIDERATIONS

The accuracy of delays imposed by the **netemu-bridge** application is highly dependent upon the underlying Linux scheduler and process priority. Using the default Linux scheduler and a regular process priority is not sufficient to insure that frames are delayed with millisecond accuracy. For highest accuracy, the **netemu-bridge** applications should be executed with a scheduling policy of SCHED\_FIFO or SCHED\_RR and a priority higher than most other processes.

## netemu-controller

---

### NAME

netemu-controller – NetEmulator control program for distributed network emulations

### SYNOPSIS

**netemu-controller** <scenarioFile> <listenAddr> <channelBaseAddr>

supported options, default values are enclosed in []:

--updatePeriod <secs>	Period between sending updates to bridges	[5.0]
--priority <level>	Realtime process priority 1-99	[90]
--scheduler <policy>	Scheduler policy, either SCHED_FIFO or SCHED_RR	[SCHED_RR]
--verbose	Displays copious debug output	
--timestamps	Enable timestamps (requires synchronized clocks)	

### DESCRIPTION

The **netemu-controller** application is part of the NetEmulator tools package and provides a single point of control for distributed network emulations. A distributed emulation requires exactly one **controller** to be present prior to starting any of the emulated radios (using the **netemu-bridge** program). The controller reads in the <scenarioFile> provided on the command line and parses it to learn about the entities, radios, subscribers, and channels which comprise the emulated network.

An entity is basically a vehicle, facility, or other large object that occupies a location and contains one or more radios.

Each radio exposes a unique Ethernet interface to provide a way for external user devices (eg. hosts and routers) to "plug in" to the distributed emulation. A radio, in general, may communicate with other radios on multiple channels simultaneously and the attributes of the channels may differ. To accommodate this possibility, a radio may have one or more subscribers and each subscriber attaches to exactly one channel.

A subscriber emulates channel delay and corruption effects, as well as throughput constraints of communications between radios.

A channel logically connects two or more subscribers. Subscribers attached to the same channel can communicate; subscribers not connected to the same channel can not communicate.

A radio (more specifically, a bridge) bridges Ethernet frames from its Ethernet port to one or more of its subscribers. A subscriber imposes datarate limitations on a frame prior to transmission, then encapsulates the Ethernet frame in a UDP multicast packet addressed to a multicast address and port specific to a channel, and transmits. Any subscribers attached to the same channel (ie. bound to the same UDP multicast address and port) will receive the UDP multicast packet, deencapsulate it, impose delay and corruption, and, if not corrupted, the radio will deliver the frame to the external destination user device. Each channel utilizes a unique UDP multicast port, thus segregating traffic that occupy different channels and preventing subscribers on different channels from communicating.

The radios in the distributed emulation are transparent to the external user devices. The user devices perceive that they are connected to the same local area network, albeit one that may have a delay, reliability, and datarate that are much worse than those of a wired network. This transparency allows the performance of host applications and network protocols and services to be evaluated in a realistic wireless environment, without requiring any modifications to the end hosts or routers.

The **netemu-controller** opens a TCP socket bound to the IP address and port number specified by <listenAddr>. The format of all address fields allows either a hostname or dotted IP address to be specified, followed by a ":", followed by the port number. If no hostname or IP address is specified,

localhost is assumed; a port number, however, must be specified. Some legal examples of addresses are:

- 192.168.0.1:23456
- :23456
- localhost:23456

The **netemu-controller** binds a TCP socket to the <listenAddr> and listens for connections from clients. Clients include all the **netemu-bridge** (ie. radio) applications in the emulation as well as the NetEmulator's graphical user interface tools: **updater**, **monitor**, **hist**. The NetEmulator also provides a program **netemu-ifstk** which can extract dynamic link delay and bit error rate information from a popular commercial modeling tool called "Satellite Tool Kit"; the program is also a client which connects to the **netemu-controller**.

The <channelBaseAddr> field specifies the multicast address and starting UDP port number which will be used to assign unique UDP port numbers to each channel. Port numbers are assigned by the **netemu-controller** to the first channel defined in the scenario starting at the port number specified in <channelBaseAddr>, and incrementing by one for each additional channel.

## THEORY OF OPERATION

After starting up, the **netemu-controller** performs a few basic tasks. It acts as a server waiting for **netemu-bridge** clients to connect to obtain their respective configurations. Rather than have each **netemu-bridge** have to read the same scenario file, only the **netemu-controller** has knowledge of the details within the scenario file. When a **netemu-bridge** is launched, it is unaware what subscribers it should have; this is provided by contacting the **netemu-controller** which responds with the configurations of each subscriber for the new **netemu-bridge**.

Besides configuring **netemu-bridge** applications, the **netemu-controller** also performs the task of periodically updating each subscriber in the distributed emulation with a transmit datarate setting along with delay and bit error rate tables. These tables allow the subscriber to impose delay and corruption on all user data frames on a per link basis. These settings can be changed through either the graphical user interface **updater** tool or via dynamic updates from the STK interface **netemu-ifstk** tool.

## netemu-hist

---

### NAME

netemu-hist – graphical qt app to record and playback datarate, delay, and bit error rate settings for the NetEmulator

### SYNOPSIS

**netemu-hist** <controller addr:port> [-v]

### DESCRIPTION

The **netemu-hist** application is part of the NetEmulator tools package. **netemu-hist** provides a history recording and playback capability for the NetEmulator. The user is presented with a dialog window containing three buttons: "Record", "Play", and "Stop" and a text edit box for specifying a file name for recording or playback.

By clicking the "Record" button, any messages sent to the **netemu-controller** by other clients (eg. **netemu-fstk**, **netemu-updater** ) to update datarates, delays, or bit error rates, will also be recorded into a history file, specified by the user. Each recorded message is timestamped with a time relative to when the "Record" button was clicked. When the user clicks "Stop", the recording ceases, and the history file is closed.

By clicking the "Play" button, the history file specified by the user will be opened and stored messages will be played back to the **netemu-controller** , using the stored timestamps.

This tool makes it possible to record the link settings from rather complex scenarios defined in STK. Later, the history file can be used without the need to run STK and **netemu-fstk**.

### USAGE

<controller addr:port> is the hostname or IP address, and port number, of the NetEmulator **netemu-controller** for the distributed emulation.

An optional -v switch may be provided at the end of the command line to cause the **netemu-hist** to generate verbose output.

## netemu-monitor

---

### NAME

netemu-monitor – graphical qt app to view current datarate, delay, and bit error rate settings for the NetEmulator

### SYNOPSIS

**netemu-monitor** <controller addr:port> [-v]

### DESCRIPTION

The **netemu-monitor** application is part of the NetEmulator tools package. **netemu-monitor** displays the current datarate, delay, and bit error rate settings of links within a NetEmulator distributed network emulation. The initial **netemu-monitor** window provides a single menu from which the user first selects the channel in which he's interested. He then selects either datarate, delay, or BER and a new window pops up displaying the selected information. Datarate information is displayed for each sender. Delay and BER information is displayed in a matrix or table format in which senders are listed in the first column and receivers are list across the top in the first row.

### USAGE

<controller addr:port> is the hostname or IP address, and port number, of the NetEmulator **netemu-controller** for the distributed emulation.

An optional -v switch may be provided at the end of the command line to cause the **netemu-monitor** to generate verbose output.

## netemu-updater

---

### NAME

netemu-updater – graphical qt app to set datarate, delay, and bit error rate for the NetEmulator

### SYNOPSIS

**netemu-updater** <scenario file> <controller addr:port> [-v]

### DESCRIPTION

The **netemu-updater** application is part of the NetEmulator tools package. **netemu-updater** provides "knobs" for controlling the datarate, delay, and bit error rate of links within a NetEmulator distributed network emulation. The user is presented with a tabbed dialog window. On the "Link Properties" panel, combo boxes allow the user to select the source and destination subscriber identifiers of a link, and text edit boxes allow the user to enter floating point values for delay and bit error rate. By clicking on the "Apply" button, a message containing the link properties will be sent to the NetEmulator **netemu-ontroller**.

On the "Transmitter Properties" panel, a combo box allows the user to select a transmit subscriber, and a text edit box allows the user to enter a floating point value for datarate, in bits per second, for that subscriber. Clicking on the "Apply" button causes a message containing the new datarate of the subscriber to be sent to the **netemu-ontroller**.

### USAGE

<scenario file> is the name of the NetEmulator scenario file that the **netemu-ontroller** is currently using.

<controller addr:port> is the hostname or IP address, and port number, of the NetEmulator **netemu-ontroller** for the distributed emulation.

An optional -v switch may be provided at the end of the command line to cause the **netemu-updater** to generate verbose output.