netem

netem provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering.

If you run a current 2.6 distribution, (Fedora, OpenSuse, Gentoo, Debian, Mandriva, Ubuntu), then netem is already enabled in the kernel and a current version of iproute2 is included. The netem kernel component is enabled under:

```
Networking -->
  Networking Options -->
    QoS and/or fair queuing -->
       Network emulator
```

Netem is controlled by the command line tool 'tc' which is part of the iproute2 package of tools. The tc command uses shared libraries and data files in the /usr/lib/tc directory.

Examples

Emulating wide area network delays

This is the simplest example, it just adds a fixed amount of delay to all packets going out of the local Ethernet.

```
# tc qdisc add dev eth0 root netem delay 100ms
```

Now a simple ping test to host on the local network should show an increase of 100 milliseconds. The delay is limited by the clock resolution of the kernel (HZ). On most 2.4 systems, the system clock runs at 100hz which allows delays in increments of 10ms. On 2.6, the value is a configuration parameter from 1000 to 100 hz.

Later examples just change parameters without reloading the qdisc

Real wide area networks show variability so it is possible to add random variation.

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms
```

This causes the added delay to be 100ms ± 10ms. Network delay variation isn't purely

random, so to emulate that there is a correlation value as well.

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms 25%
```

This causes the added delay to be 100ms ± 10ms with the next random element depending 25% on the last one. This isn't true statistical correlation, but an approximation.

Delay distribution

Typically, the delay in a network is not uniform. It is more common to use a something like a normal distribution to describe the variation in delay. The netem discipline can take a table to specify a non-uniform distribution.

```
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
```

The actual tables (normal, pareto, paretonormal) are generated as part of the iproute2 compilation and placed in /usr/lib/tc; so it is possible with some effort to make your own distribution based on experimental data.

Packet loss

Random packet loss is specified in the 'tc' command in percent. The smallest possible non-zero value is:

232 = 0.0000000232%

```
# tc qdisc change dev eth0 root netem loss 0.1%
```

This causes 1/10th of a percent (i.e 1 out of 1000) packets to be randomly dropped.

An optional correlation may also be added. This causes the random number generator to be less random and can be used to emulate packet burst losses.

```
# tc qdisc change dev eth0 root netem loss 0.3% 25%
```

This will cause 0.3% of packets to be lost, and each successive probability depends by a quarter on the last one.

Probn = .25 * Probn-1 + .75 * Random

Caveats

- When loss is used locally (not on a bridge or router), the loss is reported to the upper level protocols. This may cause TCP to resend and behave as if there was no loss. When testing protocol reponse to loss it is best to use a netem on a bridge or router

Packet duplication

Packet duplication is specified the same way as packet loss.

```
# tc qdisc change dev eth0 root netem duplicate 1%
```

Packet corruption

Random noise can be emulated (in 2.6.16 or later) with the corrupt option. This introduces a single bit error at a random offset in the packet.

```
# tc qdisc change dev eth0 root netem corrupt 0.1%
```

Packet re-ordering

There are two different ways to specify reordering. The first method **gap** uses a fixed sequence and reorders every *Nth* packet. A simple usage of this is:

```
# tc qdisc change dev eth0 root netem gap 5 delay 10ms
```

This causes every 5th (10th, 15th, ...) packet to go to be sent immediately and every other packet to be delayed by 10ms. This is predictable and useful for base protocol testing like reassembly.

The second form **reorder** of re-ordering is more like real life. It causes a certain percentage of the packets to get mis-ordered.

```
# tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
```

In this example, 25% of packets (with a correlation of 50%) will get sent immediately, others will be delayed by 10ms.

Newer versions of netem will also re-order packets if the random delay values are out of order. The following will cause some reordering:

```
# tc qdisc change dev eth0 root netem delay 100ms 75ms
```

If the first packet gets a random delay of 100ms (100ms base - 0ms jitter) and the second packet is sent 1ms later and gets a delay of 50ms (100ms base - 50ms jitter); the second packet will be sent first. This is because the queue discipline *tfifo* inside netem, keeps packets in order by time to send.

Caveats

- Mixing forms of reordering may lead to unexpected results
- Any method of reordering to work, some delay is necessary.
- If the delay is less than the inter-packet arrival time then no reordering will be seen.

Rate control

There is no rate control built-in to the netem discipline, instead use one of the other disciplines that does do rate control. In this example, we use Token Bucket Filter (TBF) to limit output.

```
# tc qdisc add dev eth0 root handle 1:0 netem delay 100ms
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 256kbit buffer 1600 limit 3000
# tc -s qdisc ls dev eth0
qdisc netem 1: limit 1000 delay 100.0ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0 )
qdisc tbf 10: rate 256Kbit burst 1599b lat 26.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0 )
```

Check on the options for buffer and limit as you might find you need bigger defaults than these (they are in bytes)

For more explanation about how to use classful queuing disciplines see: Linux Advanced Routing HOWTO - classes

Non FIFO queuing

Just like the previous example, any of the other queuing disciplines (GRED, CBQ, etc) can be used.

Delaying only some traffic

Here is a simple example that only controls traffic to one IP address.

```
 # tc qdisc add dev eth0 root handle 1: prio
 # tc qdisc add dev eth0 parent 1:3 handle 30: \
tbf rate 20kbit buffer 1600 limit  3000
 # tc qdisc add dev eth0 parent 30:1 handle 31: \
netem  delay 200ms 10ms distribution normal
 # tc filter add dev eth0 protocol ip parent 1:0 prio 3 u32 \
     match ip dst 65.172.181.4/32 flowid 1:3
```

The commands makes a simple priority queueing discipline, then a TBF is added to do rate control, then attaches a basic netem. Finally, a filter classifies all packets going to 65.172.181.4 as being priority 3. For more info on traffic classification see LARTC -- filters

FAQ

How come first ping takes longer?

The first ICMP packet in a ping requires an ARP request/response as well.

How come TCP is so slow over netem?

When you run TCP over large Bandwidth Delay Product links, you need to do some TCP tuning to increase the maximum possible buffer space.

How can I use netem on incoming traffic?

You need to use the Intermediate Functional Block pseudo-device IFB . This network device allows attaching queuing discplines to incoming packets.

```
 # modprobe ifb
 # ip link set dev ifb0 up
 # tc qdisc add dev eth0 ingress
 # tc filter add dev eth0 parent ffff: \
   protocol ip u32 match u32 0 0 flowid 1:1 action mirred egress redirect dev ifb0
 # tc qdisc add dev ifb0 root netem delay 750ms
```

Another way is to use another machine as an Ethernet bridge , and apply netem to both Ethernet devices.

How to reorder packets based on jitter?

Starting with version 1.1 (in 2.6.15), netem will reorder packets if the delay value has lots of jitter.

If you don't want this behaviour then replace the internal queue discipline *tfifo* with a pure packet fifo *pfifo*. The following example has lots of jitter, but the packets will stay in order.

```
# tc qdisc add dev eth0 root handle 1: netem delay 10ms 100ms
# tc qdisc add dev eth0 parent 1:1 pfifo limit 1000
```

How does the value of HZ impact Netem?

In the 2.6 line of kernels, HZ is a configurable parameter that takes values of either 100, 250, or 1000. Because it affects the granularity with which Netem is able to delay packets, it is most beneficial to set HZ to 1000, which will allow for delays in increments of 1ms. See this mailing list post for a more detailed discussion of the impact of HZ.

In kernel versions, 2.6.22 or later, netem will use high resolution timers, if they are enabled. This allows for finer granularity (sub-jiffie) resolution.

Links

- Linux Conf Au presentation and paper.
- dummynet an network emulator in FreeBSD
- NISTnet

Contact Info

Since netem is part of the core Linux subsystem, all bug reports and patches should be sent to Linux Network Developers mailing list.

The netem@osdl.org mailing list is available for user discussions. Mail from non-subscribers is moderated to prevent spam. To subscribe or unsubscribe use the Netem mailing list interface.

Iproute2 is a collection of utilities for controlling TCP / IP networking and traffic control in Linux. It is currently maintained by Stephen Hemminger <shemminger@osdl.org>. The original author, Alexey Kuznetsov, is well known for the QoS implementation in the

Linux kernel.

Most network configuration manuals still refer to ifconfig and route as the primary network configuration tools, but ifconfig is known to behave inadequately in modern network environments. They should be deprecated, but most distros still include them. Most network configuration systems make use of ifconfig and thus provide a limited feature set. The /etc/net project aims to support most modern network technologies, as it doesn't use ifconfig and allows a system administrator to make use of all iproute2 features, including traffic control.

iproute2 is usually shipped in a package called iproute or iproute2 and consists of several tools, of which the most important are *ip* and *tc*. ip controls IPv4 and IPv6 configuration and tc stands for traffic control. Both tools print detailed usage messages and are accompanied by a set of manpages.

Download

The current version is in the **download** directory on kernel.org.

New versions will be announced on the netdev mailing list.

The current iproute2 source is maintained in the GIT repository. To get the current source use:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git
```

You can also browse the source online via gitweb.

Documentation

There are many sources of documentation on the web, and it is mostly up to date.

- LARTC HOWTO
- IPROUTE2 Utility Suite HOWTO
- Iproute2_examples
- Documentation on how to integrate both firewall (IP Tables) and Linux advanced routing.
- Documentation on how to integrate both firewall (IP Tables) and Linux advanced routing - Brazilian Portuguese translation