Manual
tc Packet Filtering and netem

Ariane Keller
ETH Zurich

July 20, 2006

Contents

Chapter 1

Brief Introduction

This manual describes the usage of netem. Netem is a network emulator in the linux kernel 2.6.7 and higher that reproduces network dynamics by delaying, dropping, duplicating or corrupting packets. Netem is an extension of tc, the linux traffic control tool in the iproute2 package.

To understand this manual, which describes the configuration of the network emulator, we assume that you have some basic knowledge in IP networking and more specifically in IP packet handling in the Linux kernel. We recall that any linux machine running netem must be configured as a router (see section 4.1 for more details). The simplest network topology to use netem on a router is depicted in figure 1.1.

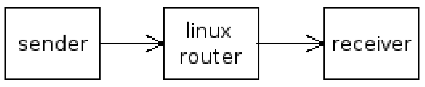

Figure 1.1: Network topology

Inside the router, IP packet handling is performed as follows. Packets enter a network interface card (NIC) and are classified and enqueued before entering linux internal packet handling. After packet handling, packets are classified and enqueued for transmission on the egress NIC.

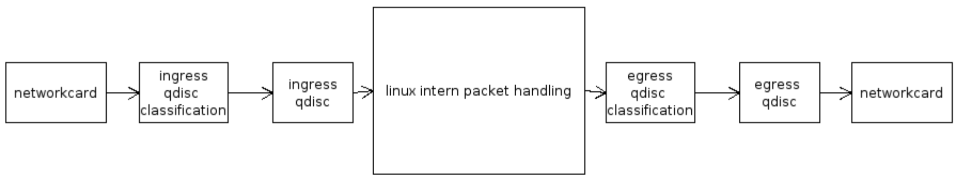
networkcard
ingress qdisc classification
ingress qdisc
linux intern packet handling
egress qdisc classification
egress qdisc
networkcard

Figure 1.2: IP packet handling in the Linux kernel

Packet classification can be performed by analyzing packet header fields (source and destination IP addresses, port numbers, etc.) and payload. The classification can be configured with the traffic control tool tc. Based on this classification, packets are enqueued in one of the ingress/egress queues. In the standard configuration there is solely one queue per interface and the packets are processed in a FIFO manner. The combination from queue and algorithm that decides when to send which packet is called qdisc (short for queueing discipline).

This manual starts with describing how one or multiple qdiscs can be configured on an egress interface and how to configure packet classification to identify specific flows before describing the configuration of the network emulation tool netem. This description includes the configuration of our enhancements for trace controled network emulation TCN. Moreover, we give information how to generate packet action traces that specify the amount of delay for each packet in the emulation as well as, which packets are to be dropped, duplicated, or corrupted.

Chapter 2

tc: Linux Advanced Routing and Traffic Control

Traffic control (tc) is part of the linux iproute2 package which allows the user to access networking features. The package itself has three main features: monitoring the system, traffic classification, and traffic manipulation. The tc part in the package can be used

- to configure qdiscs as well as
- to configure packet classification into qdiscs.

A general description of the iproute2 package can be found in an online manual at http://lartc.org/howto. To save time, this section repeats all relevant parts of creating qdiscs that can be found in the online manual before giving more details on packet classification. The first part of the section describes how different queueing disciplines (qdiscs) can be attached to one outgoing network interface. The second part explains how packets can be classified into the schedulers based on packet properties such as the source or destination ip address header field.

2.1 tc qdiscs and classes

2.1.1 Terminology

Queueing Discipline (qdisc)	packet queue with an algorithm
	that decides when to send which packet
Classless qdisc	qdisc with no configurable internal subdivision
Classful qdisc	qdisc that may contain classes
	classful qdiscs allow packet classification
Root qdisc	a root qdisc is attached to each network interface
	either classful or classless
egress qdisc	works on outgoing traffic
	only egress qdiscs are considered in this manual
ingress qdisc	works on incoming traffic
	see the lartc manual for more detail
Class	classes either contain other classes, or a qdisc is attached
Filter	classification can be performed using filters

2.1.2 General Commands

Generate a root qdisc:

```
tc qdisc add dev DEV handle 1:  root QDISC [PARAMETER]
```

Generate a non root qdisc:

```
tc qdisc add dev DEV parent PARENTID handle HANDLEID QDISC [PARAMETER]
```

Generate a class:

```
tc class add dev DEV parent PARENTID classid CLASSID QDISC [PARAMETER]
```

DEV:	interface at which packets leave, e.g. eth1
PARENTID:	id of the class to which the qdisc is attached e.g. X:Y
HANDLEID:	unique id, by which this qdisc is identified e.g. X:
CLASSID:	unique id, by which this class can be identified e.g. X:Y see section 2.1.3
QDISC:	type of the qdisc attached, see table 2.1
PARAMETER:	parameter specific to the qdisc attached

qdisc	description	type
pfifo_fast:	simple first in first out qdisc	classless
TBF:	Token Bucket Filter, limits the packet rate	classless
SFQ:	Stochastic Fairness Queueing, devides traffic into queues and sends packets in a round robin fashion	classless
PRIO:	allows packet prioritisation	classful
CBQ:	allows traffic shaping, very complex	classful
HTB:	derived from CBQ, but much easier to use	classful

Table 2.1: queueing disciplines (more details are found in the lartc manual)

2.1.3 Building a qdisc Tree

By default each interface has one egress (outgoing) FIFO qdisc (queueing discipline). To be able to treat some packets different than others, a hierarchy of qdiscs can be constructed. Furthermore different kinds of qdiscs exist, each with different properties and parameters that can be tuned. To build a tree, a classful root qdisc has to be chosen. In this example HTB (Hierarchical Token Bucket) is used, since the other qdiscs are either classless or prioritize some traffic (PRIO) or are to complicated (CBQ). For information about HTB see: http://luxik.cdi.cz/~devik/qos/htb/. At the leaves a classless qdisc can be attached. In this example netem is used, the network emulation qdisc, which is explained in detail in chapter 3.

A tree as shown in figure 2.1 with three leaf qdiscs and one root qdisc can be created as follows:

First the default root qdisc is replaced:

```
tc qdisc add dev eth1 handle 1:  root htb
```

then one root class and three children classes are created:

```
tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:11 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:12 htb rate 100Mbps
tc class add dev eth1 parent 1:1 classid 1:13 htb rate 100Mbps
```

The parentid is equal to the classid of the respective parent. The children's class ids have to have the same major number (number before the colon) as

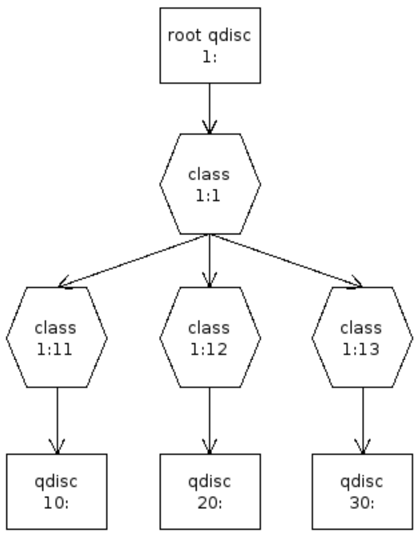

Figure 2.1: qdisc hierarchy.

their parent and a unique minor number (number after the colon). The qdisc is HTB with a maximal rate of 100 Megabits per second.
In the next step a qdisc is added to each class.

```
tc qdisc add dev eth1 parent 1:11 handle 10:  netem delay 100ms
tc qdisc add dev eth1 parent 1:12 handle 20:  netem
tc qdisc add dev eth1 parent 1:13 handle 30:  netem
```

The parent id is the id of the class to which the qdisc is attached. The handle is a unique identifier. Netem is chosen as a qdisc.
Unique numbers must just be unique within an interface.

2.1.4 Changing and Deleting qdiscs

The commands for changing and deleting qdiscs have the same structure as the add command. Qdisc parameters can be adapted using the change command. To change the 100ms delay from the qdisc with handle 10: (from the previous example) to 200ms the following command is used:

```
tc qdisc change dev eth1 parent 1:11 handle 10:  netem delay 200ms
```

To delete a complete qdisc tree only the root needs to be deleted:

```
tc qdisc del dev eth1 root
```

It is also possible to delete only a particular qdisc:

```
tc qdisc del dev eth1 parent 1:11 handle 10:
```

2.2 tc Filter Options

The ability of tc to filter packets is huge. Not only different filter types exist, but also the mechanism of referencing one particular filter is quite complex. The filter commands can be divided into two groups: simple filters and complex filters. Simple filters are restricted in the way that they are referenced. Complex filters have identifier assigned and they have some more knowledge about the packets processed.

2.2.1 "Simple" Filter Commands

Simple filters allow the creation of filters that evaluate fields at a specified constant location. This implies that the IP header is assumed to be of constant size (20 bytes) and therefore must not include any options. Filter deletion can only be done for a complete priority band.

2.2.1.1 Command Structure

Add a filter:

```
tc filter add dev DEV protocol PROTO parent ID prio PRIO FILTER match
SELECTOR [FIELD] PATTERN MASK [at OFFSET] flowid FLOWID
```

Delete filter:

```
tc filter del DEV protocol PROTO parent ID prio PRIO
```

Show filter:

```
tc filter show dev DEV [protocol PROTO [parent ID [prio PRIO]]]
```

DEV:	interface at which packets leave, e.g. eth1
PROTO:	protocol on which the filter must operate, e.g. ip, ipv6, arp, 802_3 see table 2.4 for all supported protocols
ID:	id of the class at which filtering should begin e.g. 1: to start at the root
PRIO:	determines the order in which filters are checked higher numbers → lower priority important: different protocols cannot have the same priority
FILTER:	specifies which filter is used, see section 2.2.1.2
SELECTOR:	depends on the filter, e.g. for u32: u32, u16, u8, ip, ip6
FIELD:	name of the field to be compared only for ip and ip6 selector
PATTERN:	value of the specified field (decimal or hexadecimal)
MASK:	indicates which bits are compared
OFFSET:	start to compare at the location specified by PROTO + OFFSET bytes, only for uX selectors
FLOWID:	references the class to which this filter is attached

2.2.1.2 Filter Overview

tc knows different filters for classifying packets, see table 2.2. The most interesting is the u32 filter which allows classification according to every value in a packet. This filter is discussed in more detail in the following sections.

filter	description
route:	bases the decision on which route the packet will be routed by
fw:	bases the decision on how the firewall has marked the packet
rsvp:	routes packets based on RSVP (ReSerVation Protocol, a reservation based extension to best effort service, not supported in the internet)
tcindex:	used in DSMARK qdisc (DSMARK is used for differentiated services, a priority based extension to the best effort service)
u32:	bases the decision on fields within the packet

Table 2.2: tc filter overview

2.2.1.3 u32 Filter

To simplify the filtering the u32 filter has different selectors:

u32	filters according to arbitrary 32-bit fields in the packets the starting position is indicated with OFFSET (in bytes) OFFSET must be a multiple of 4. A mask of the same length is used to indicate the bits that should match. E.g. 0xFFFFFFFF: all 32 bits have to match
u16	filters according to arbitrary 16-bit fields in the packets OFFSET has to be a multiple of 2. E.g. 0xFFFF: all 16 bits have to match, 0x0FF0: only bits 4 to 11 have to match
u8	filters according to arbitrary 8-bit fields in the packets. E.g. 0xFF all 8 bits have to match, 0x0F only bits 4 to 8 have to match.
ip	bases decision on fields in the ipv4 and upper layer headers (for PROTO=ip) (see ipv4 traffic)
ip6	bases decision on fields in the ipv6 and upper layer headers (for PROTO=ipv6) (see ipv6 traffic).

The desired values for OFFSET can be found by inspecting figure 2.2 and 2.3 for ipv4 and ipv6 packets respectively.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.2: ipv4 header format

```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Traffic Class |           Flow Label                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Payload Length        |  Next Header  |   Hop Limit   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                         Source Address                        +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                      Destination Address                      +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.3: ipv6 header format

ipv4 Traffic

Simple filters assume the ipv4 header length to be constant (without options). Therefore incorrect results may be found if these filters are used for fields belonging to layer 4 (e.g. udp, tcp etc). The complex filters solve this problem by inspecting the ihl (internet header length) field (see section 2.2.3. For ipv4 traffic (PROTO=ip, SELECTOR=ip) the following fields are predefined:

field name	tc short	sample value	mask
IP source	src	10.0.0.2/24	na
IP destination	dst	10.0.1.2/24	na
IP header length	ihl	20	0xF
type of service	tos	10	0xFF
transport protocol (see table 2.3)	protocol	6	0xFF
packet is not fragmented	nofrag	na	na
packet is first (or only) fragment	firstfrag	na	na
don't fragment flag set	df	na	na
more fragments flag set	mf	na	na
udp/tcp source port	sport	5000	0xFFFF
udp/tcp dst port	dport	80	0xFFFF
icmp type	icmp_type	1	0xFF
icmp code	icmp_code	1	0xFF

Example for destination port filtering:

```
tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip dport
```

protocol number	protocol
1	ICMP
2	IGMP
4	IP (encapsulation)
6	TCP
17	UDP
41	IPv6
58	IPv6-ICMP

Table 2.3: some protocol numbers as used in the ipv4 header. For a complete list see: www.iana.org/assignments/protocol-numbers

```
5000 0xffff flowid 1:11
```

Example for ip source address filtering:

```
tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
10.0.0.2 flowid 1:11
```

ipv6 Traffic

The lartc manual says ipv6 filtering doesn't work, but I found some examples from people using it. I did not do any tests on my own.
For ipv6 traffic (PROTO=ipv6, SELECTOR=ip6) the following fields are predefined:

field name	tc short	possible values	mask
IP source	src	any ipv6 address	na
IP destination	dst	any ipv6 address	na
traffic class	priority	1	0xFF
next header	protocol	6	0xFF
flowlabel	flowlabel	1234	0xFFFFF
udp/tcp source port	sport	5000	0xFFFF
udp/tcp dst port	dport	80	0xFFFF
icmp type	icmp_type	1	0xFF
icmp code	icmp_code	1	0xFF

Example for destination port filtering:

```
tc filter add dev eth1 parent 1:  protocol ipv6 prio 2 u32 match ip6
dport 5000 0xffff flowid 1:11
```

Example for ip source address filtering:

```
tc filter add dev eth1 parent 1:  protocol ipv6 prio 2 u32 match ip6
src 2001:6a8:802:7::2 flowid 1:11
```

Other Traffic

There is a large list of accepted protocols (see table 2.4 for details). However, for all protocols other than ip and ipv6, no predefined fields are available. With a u32 selector arbitrary parts of a packet can be filtered. However, one has to know the structure of such a packet to place the "compare-pointer" to the correct location.

An Ethernet packet has the destination address in the first 6 bytes, followed by 6 bytes of source address and a 2 byte type field. Note that the source MAC address is already that of the router. To filter according to the MAC destination address one has to concatenate two u32 filters, since 6 bytes = 48 bits > 32 bits.

loop	pup	ip	irda	control
x25	arp	bpq	mobitex	tr_802_2
dec	dna_dl	dna_rc	ppptalk	localtalk
dna_rt	lat	cust	wan_ppp	ddcm
sca	rarp	atalk	snap	802_2
aarp	ipx	ipv6	all	ax25
802_3				

Table 2.4: Supported protocols for filtering

To match the Ethernet destination address 12:34:56:78:9A:BC two filters must be concatenated in the following manner:

```
tc filter add dev eth1 parent 1:  protocol 802_3 prio 3 u32 match u32
0x12345678 0xffffffff at 0 match u32 0x9abc0000 0xffff0000 at 4 flowid
1:11
```

2.2.2 Filtering Based on Multiple Criterions

Two possibilities exist to combine different filter rules. The logical AND restricts the packets on more than one field, the logical OR allows packets to have different values in one field.

Logical AND
Filtercriteria can be concatenated to allow a more specific filtering. To restrict packets on more than one field a filter with multiple match clauses can be used. To filter a packet according to its source ip address and its destination udp port the following statement can be used:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip protocol 17
0xff match ip dport 5000 0xffff match ip src 10.0.0.2 flowid 1:11
```

Logical OR
A class is allowed to have different filters. All packets that match one of these filters will be processed by the respective class. The filters are traversed according to their priorities and in the order they where created. All packets can be filtered based on destination ip address 10.0.0.2 and 10.0.1.2 to class 1:11 the following way:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.0.2
flowid 1:11
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.1.2
flowid 1:11
```

Remember: different PRIO values must be used for different protocols!
To filter for an address range the common notation can be used:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dst 10.0.1.0/24
```

which matches every packet going to an address in the range 10.0.1.0 to 10.0.1.255.
Filtering for a destination port range is done by adjusting the mask:

```
tc filter add dev eth1 protocol ip prio 1 u32 match ip dport 50000
0xff00
```

This matches every port between 49920 (0xC300) and 50175 (0xC3FF).

2.2.3 Complex Filter Commands

Complex filters make use of user defined handles and hash tables. A hashtable contains slots and the slots contain filter rules. This allows us to specify exactly one filter rule. The hash table memorises which protocol is used (e.g. ip). This allows the evaluation of the ihl (internet header length) field to get the correct start location of the upper layer protocol. In this section only ip traffic and u32 filtering is considered. Before adding a filter rule some prearrangements have to be taken.

2.2.3.1 Command Structure

First a classifier has to be created at the root. Each time a new PRIO value is introduced a new classifier id (CLASSIFIERID) is created. It starts with the value 800 and is incremented for each new PRIO.

```
tc filter add dev eth1 parent 1:0 prio PRIO protocol ip u32
```

A hash table has to be created, this allows the usage of the nexthdr+OFFSET option.

```
tc filter add dev eth1 parent 1:0 prio PRIO handle ID: u32 divisor
1
```

The hash table must be linked to the correct priority classifier:

```
tc filter add dev eth1 parent 1:  protocol ip prio PRIO u32 ht CLASSIFIERID::
match u8 0 0 offset at 0 mask 0x0f00 shift 6 link ID:
```

The actual filter is attached:

```
tc filter add dev eth1 protocol ip parent 1:0 prio PRIO handle 0xHANDLE
u32 ht ID: match SELECTOR [FIELD] PATTERN MASK [at OFFSET | nexthdr+OFFSET]
flowid FLOWID
```

Delete a specific filter:

```
tc filter del dev eth1 protocol ip parent 1:0 prio PRIO handle ID::HANDLE
u32
```

with:

PRIO:	Priority, determines the order in which filters are checked
ID:	Identifier of the hash table
HANDLE:	handle of a specific rule. If no handle is specified
	tc assigns one starting at 800
SELECTOR:	specifies the filter
	see section ipv4 and table table 2.5
	note: udp and tcp are the same filter -> check the protocol separately
FIELD:	name of the header field to be compared
PATTERN:	value of the specified field
MASK:	indicates which bits are compared
OFFSET:	starts to compare at the ip header + OFFSET bytes
nexthdr+OFFSET:	starts to compare at the upper layer header + OFFSET bytes
FLOWID:	references the class to which this filter is attached

It is also possible to divide a hashtable into slots. This helps to find the correct filter rule fast. Some examples are found in the lartc manual. The syntax is as follows:

Create hash table:

```
tc filter add dev eth1 parent 1:0 prio PRIO protocol ip handle ID:
u32 divisor SLOTS
```

selector	field	description	filter
udp	src	udp source port	0xffff
	dst	udp destination port	
tcp	src	tcp source port	0xffff
	dst	tcp destination port	
icmp	type	icmp type field	0xff
	code	icmp code field	

Table 2.5: Filter selectors for complex filters only

Add a filter rule:

```
tc filter add dev eth1 protocol ip parent 1:0 prio PRIO handle 0xHANDLE
u32 ht ID:SLOTNR: match SELECTOR PATTERN MASK [at OFFSET | nexthdr+OFFSET]
flowid FLOWID
```

Delete a filter rule:

```
tc filter del dev eth1 protocol ip parent 1:0 prio PRIO handle ID:SLOTNR:HANDLE
u32
```

with:

SLOTNR:	slot to which the rule is attached
SLOTS:	number of slots in one hash table

I was not able to combine multiple slots with a working nexthdr function.

2.2.3.2 Examples

For all examples a tree as in figure 2.1 is assumed.
Setup the hash table for eth1, priority 1:

```
tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32
tc filter add dev eth1 parent 1:0 prio 1 handle 1:  u32 divisor 1
tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 ht 800::  match
u8 0 0 offset at 0 mask 0x0f00 shift 6 link 1:
```

Filter all traffic that leaves eth1 and has its udp source port equal to 50000 to class 1:11. NOTE: the udp src and tcp src filter are EXACTLY the same. To filter only UDP packets the transport protocol field in the ip header must be examined.

```
tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match udp src 50000
0xffff match ip protocol 17 0xff flowid 1:11
```

Select tcp/telnet traffic to 193.233.7.75 and direct it to class 1:11 (telnet uses port 23 = 0x17). The handle 123 is assigned.

```
tc filter add dev eth1 parent 1:0 prio 1 handle 0x123 u32 ht 1: match
ip dst 193.233.7.75  match tcp dst 0x17 0xffff  flowid 1:11
```

Delete the rule above:

```
tc filter del dev eth1 parent 1:0 prio 1 handle 1::123 u32
```

Chapter 3

netem

This chapter describes the usage of netem, the linux network emulator module. netem is part of each standard linux kernel 2.6.7 and later. However some features are only available in kernel version 2.6.16 and later. The standard part of netem allows packet handling according to statistical proberties. In addition a trace mode has been written though it is not part of the standard kernel. This trace mode allows the specification of an independent value for each packet to be processed. The first part of this chapter describes the general configuration of netem whereas the second part discusses the generation of tracefiles.

3.1 Basic Operation of netem

netem provides functionality for testing protocols by emulating network properties. netem can be configured to process all packets leaving a certain network interface.

Four basic operations are available:

delay	delays each packet
loss	drops some packets
duplication	duplicates some packets
corruption	introduces a single bit error at a random offset in a packet

3.1.1 Original netem

In standard mode the packet delay can be specified by a constant, a variation a correlation and a distribution table. Packet loss, duplication and corruption can be modelled using a percentage and a correlation. An online manual is available at http://linux-net.osdl.org/index.php/Netem.

3.1.2 Trace Control for Netem TCN

For each packet the operation can be specified completely independent from the other packets. By monitoring some internet traffic a tracefile can be produced. This tracefile contains the real characteristics of the internet traffic at the time of monitoring. The tracefile can be used as the source to modify the behavior (delay, duplication, loss, corruption) of a packet. The generation of tracefiles is discussed in section 3.4.

3.2 Prerequisite

As a first step the source code of the netem kernel module and the source code of tc has to be patched.
Switch to the source code directory and type:
`cat /path/to/patch/trace.patch | patch -p1`
This has to be done once for the kernel module and once for tc.

In a next step the linux kernel must be configured and built.
The linux kernel has an internal timer. The frequency of this timer defines the precision with which netem sends packets. Since linux kernel 2.6.13 the frequency of the timer can be set at compile time. The maximum value is 1000, this leads to a timerinterrupt every 1ms. The default value is 250 for kernel versions 2.6.13 and later. For the use of netem it is important to set the timer frequency to 1000 since otherwise the resolution of netem will be 4ms instead of the possible 1ms! The timer frequency can be set after executing make menuconfig .

```
Processor type and features --->
   Timer frequency (250HZ) --->
      ( ) 100 HZ
      ( ) 250 HZ
      (X) 1000 HZ
```

3.3 Using netem

Since netem is a qdisc as described in chapter2.1 it has to be configured with tc, the traffic control tool of linux. The simplest netem command adds a constant delay to every packet going out through a specific interface:
`tc qdisc add dev eth0 root netem delay 100ms`
The command to add a trace file involves a few parameters:
`tc qdisc add dev eth0 root netem trace FILE LOOP [DEFAULTACTION]`
with:

FILE:	the tracefile to be attached
LOOP:	how many times the trace file is traversed 0 means forever
DEFAULTACTION:	if no delay can be read from the tracefile the default value is taken 0: no delay, 1: drop the packet, default is 0

The following example adds the tracefile ”testpattern.bin” to the root qdisc of eth1. The tracefile is repeated 100 times and then all packets are dropped.
`tc qdisc add dev eth1 root netem trace testpattern.bin 100 1`

3.4 Generation of Tracefiles

The trace file contains the packet actions to be performed. Some example tracefiles are available from http://tcn.hypert.net. There are .txt files that contain the original delay values as measured with network probing and .bin files that contain the netem compatible delay values.

```
To create your own trace file some tools are provided:
headgen
generates one packet action value for the trace file.
It takes the type and the delay as an argument.
e.g. headgen <head> <delay>
with <head> = 0 -> delay only
              1 -> drop packet
              2 -> duplicate packet
              3 -> corrupt packet
     <delay> = delay value in microseconds

txt2bin
converts the output form headgen to a netem readable form
e.g txt2bin <inputfile.txt> <outputfile.bin>
    with <inputfile.txt> = file with values as obtained by headgen, one per line
         <outputfile.bin> = file that must be given as argument to netem trace

bin2txt
takes a netem compatible file and converts it to the txt format.
The output is printed to the shell.
If you want to save the output in a file use a pipe.
e.g. bin2txt <inputfile.bin> [u]
     with <inputfile.bin> = Netem compatible file
          u = optional, output in understandable format e.g head and delay are
              reported separately
usage with a pipe:
     bin2txt <inputfile.bin> [u] | <outputfile.txt>

Example
A. The following values have been measured:
1. 1ms delay
2. 2ms delay
3. packet loss
4. 1ms delay
5. 1ms delay and duplication
6. 2ms delay
7. 4ms delay and corruption
8. 3ms delay

B. Obtain the corresponding values for the .txt file and write them in a file
   e. g. myvalues.txt
```

```
# headgen 0 1000   -> 1000
# headgen 0 2000   -> 2000
# headgen 1 0      -> 536870912
# headgen 0 1000   -> 1000
# headgen 2 1000   -> 1073742824
# headgen 0 2000   -> 2000
# headgen 3 4000   -> 1610616736
# headgen 0 2000   -> 2000

myvalues.txt:
1000
2000
536870912
1000
1073742824
2000
1610616736
2000

C. Generate the netem compatible file:
# txt2bin myvalues.txt myvalues.bin
The file myvalues.bin can be used as the trace argument for tc.

D. If you want to see what was originally in your file use bin2txt:
# bin2txt myvalues.bin
1000
2000
536870912
1000
1073742824
2000
1610616736
2000

# bin2txt myvalues.bin u
00000000000000000000001111101000   head:  0  delay:     1000 value:       1000
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
00100000000000000000000000000000   head:  1  delay:        0 value:  536870912
00000000000000000000001111101000   head:  0  delay:     1000 value:       1000
01000000000000000000001111101000   head:  2  delay:     1000 value: 1073742824
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
01100000000000000000111110100000   head:  3  delay:     4000 value: 1610616736
00000000000000000000011111010000   head:  0  delay:     2000 value:       2000
```

3.5 Statistics

netem trace mode collects statistics about different events. To dump this data use:

```
$ cat /proc/netem/stats
```

Note that only stats for flows with packet counter greater than zero are shown. The dump only works with a loaded netem kernel module and only gives an output if at least one flow has a packet counter greater than zero.
Example:

```
# cat /proc/netem/stats
 Statistics for Flow 0
 ----------------------
 Packet count:              158299
 Packets ok:                158299
 Packets with normal Delay: 158299
 Duplicated Packets:        0
 Drops on Request:          0
 Corrupted Packets:         0
 No valid data available:   0
 Uninitialized access:      0
 Bufferunderruns:           0
 Use of empty Buffer:       0
 No empty Buffer:           0
 Read behind Buffer:        0
 Buffer1 reloads:           8
 Buffer2 reloads:           8
 Switches to Buffer1:       8
 Switches to Buffer2:       8
 Switches to empty Buffer1: 0
 Switches to empty Buffer2: 0
```

The statistic for one flow is reset every time a tracefile is changed for that flow. The complete statistic can be reset manually by typing:

```
$ echo > /proc/netem/stats
```

Use

```
$ watch -n1 cat /proc/netem/stats
```

for continuous statistics. End with CTRL-C.

Chapter 4

Examples

The first part of this chapter describes, how computers must be configured to allow a network emulation. The second part gives some ready to use examples, that show all relevant features of packet filtering.

4.1 Hardware Configuration

To be able to test two network devices a PC with netem and with at least two network cards is required (see figure 4.1).

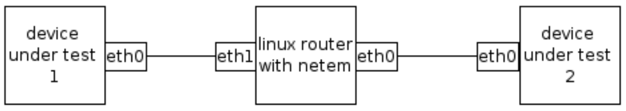

Figure 4.1: Network configuration

Example IP configuration:

-	linux router	
	IP address eth1:	10.0.1.1
	IP address eth2:	10.0.0.1
	netmask:	255.255.255.0 (for both devices)
-	device under test 1	
	IP address eth0:	10.0.1.2
	netmask:	255.255.255.0
	gateway:	10.0.1.1
-	device under test 2	
	IP address eth0:	10.0.0.2
	netmask:	255.255.255.0
	gateway:	10.0.0.1

Allow routing:
```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

make it permanent by adding the line net/ipv4/ip_forward=1 to the file /etc/sysctl.conf
Set IP addresses:

```
ifconfig eth0 10.0.0.2
```

Set netmask:

```
ifconfig eth0 netmask 255.255.255.0
```

Set default gateway:

```
route add default gw 10.0.0.1
```

Set MAC address (normally not needed)

```
ifconfig eth0 hw ether 00:01:6C:E9:38:59
```

These changes can be saved in a shell script and executed after each reboot. In Debian distributions the configuration can also be written to the file /etc/network/interfaces. Suggested contents:

```
auto eth0
iface eth0 inet static
address 10.0.0.2
netmask 255.255.255.0
gateway 10.0.0.1
hwaddress ether 00:01:6C:E9:38:59
```

This script is automatically carried out upon reboot. The correct execution can be checked with `ifconfig eth0`. If the result isn't the expected one, one can type in the shell: `ifup eth0`.

4.2 Qdiscs, filter and netem

Example 1: Outgoing Interface
All packets leaving eth1 will be processed by netem. The tracefile "testpattern1.bin" is read once and afterwards all packets are dropped.

```
tc qdisc add dev eth1 root netem trace testpattern1.bin 1 1
```

Example 2: Tree
In this example all packets coming from the 10.0.1.0/24 network will be processed with the file "testpattern1.bin" and all packets coming form the 10.0.2.0/24 network will be processed with the file "testpattern2.bin". The configuration is shown in figure 4.2. The htb parameter r2q is set to 1700 to suppress some warnings. Ceil = 100Mbps assures that all classes get the bandwith available.

```
tc qdisc add dev eth1 handle 1:  root htb r2q 1700

tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps ceil
100Mbps

tc class add dev eth1 parent 1:1 classid 1:11 htb rate 100Mbps

tc class add dev eth1 parent 1:1 classid 1:12 htb rate 100Mbps

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
10.0.1.0/24 flowid 1:11

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 match ip src
```

```
10.0.2.0/24 flowid 1:12

tc qdisc add dev eth1 parent 1:11 handle 10:  netem trace testpattern1.bin
0 1

tc qdisc add dev eth1 parent 1:12 handle 20:  netem trace testpattern2.bin
100 0
```

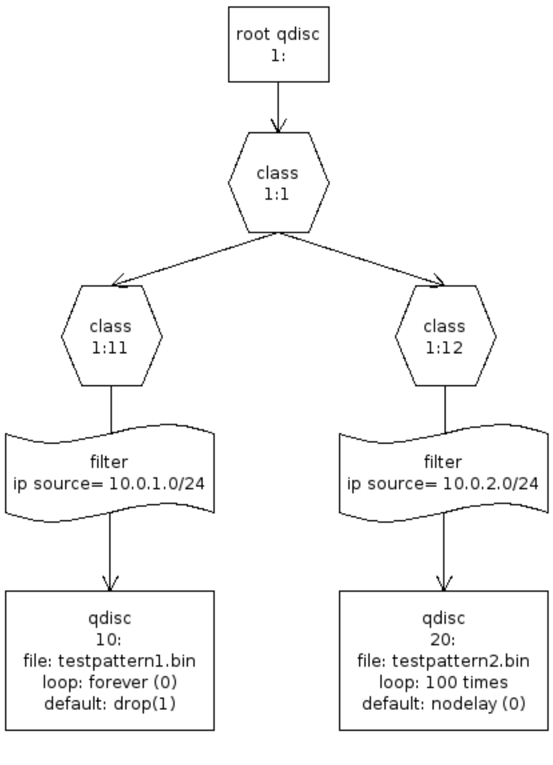

Figure 4.2: Example setup

Example 3: Complex Filter
Filter on eth1 all tcp traffic to 10.0.2.172 and port 80 or 20/21 (http and ftp traffic) that comes from subnet 10.0.1.0/24 to class 1:20. Treat packets as described in the file test.bin. This file is repeated for ever and the default action is dropping packets.
Creating qdiscs and classes:

```
tc qdisc add dev eth1 handle 1:  root htb r2q 1700

tc class add dev eth1 parent 1:  classid 1:1 htb rate 100Mbps ceil
100Mbps
```

```
tc class add dev eth1 parent 1:1 classid 1:20 htb rate 100Mbps

tc qdisc add dev eth1 parent 1:20 handle 12:  netem trace test.bin
0 1
```

Create filter:

```
tc filter add dev eth1 parent 1:0 prio 1 protocol ip u32

tc filter add dev eth1 parent 1:0 prio 1 handle 1:  u32 divisor 1

tc filter add dev eth1 parent 1:  protocol ip prio 1 u32 ht 800::  match
u8 0 0 offset at 0 mask 0x0f00 shift 6 link 1:

tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match tcp dst 80
0xffff match ip protocol 6 0xff match ip src 10.0.1.0/24 match ip dst
10.0.2.172 flowid 1:20

tc filter add dev eth1 parent 1:0 prio 1 u32 ht 1:  match tcp dst 20
0xfffe match ip protocol 6 0xff match ip src 10.0.1.0/24 match ip dst
10.0.2.172 flowid 1:20
```

Chapter 5

Useful Links

- Linux advanced routing and traffic control howto: http://lartc.org/howto
- HTB manual: http://luxik.cdi.cz/ devik/qos/htb
- Netem: http://linux-net.osdl.org/index.php/Netem
- Trace control for netem: http://tcn.hypert.net