pktgen the linux packet generator

Robert Olsson
Uppsala Universitet & SLU

Abstract

pktgen is a high-performance testing tool included in the Linux kernel. Being part of the kernel is currently best way to test the TX process of device driver and NIC. pktgen can also be used to generate ordinary packets to test other network devices. Especially of interest is the use of pktgen to test routers or bridges which use the Linux network stack. Because pktgen is "in-kernel", it can generate very high packet rates and with few systems saturate network devices as routers or bridges.

Introduction

This paper describes a novel, major rework of pktgen intended for Linux 2.7 and is now publicly available for testing. Much of the rework has been focused on multi-threaded, SMP support. The main goal is to have one pktgen thread per CPU which can then drive one or more NICs. An in-kernel pseudo driver offers unique possibilities in performance and capabilities. The trade-off is additional responsibility in terms of robustness and avoiding kernel bloat (vs user mode application). Pktgen is not an all-in-one testing tool. It offers a very efficient direct access to the host system NIC driver/chip TX-process and bypasses most of the Linux networking stack. Because of this, use of pktgen requires root access. The packet stream generated by pktgen can be used as input to other network devices. Pktgen also tests other subsystems as packet memory allocators and I/O buses. The author has done tests sending packets from memory to several GIGE interfaces on different PCI-buses using several CPU's. Rates > 10 GBit/s have been seen.

Other testing tools

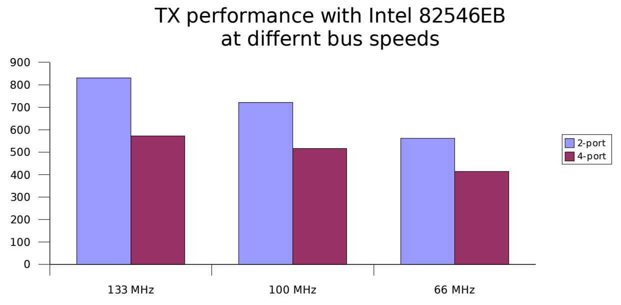
There are lots of good testing tools for network and TCP testing. netperf and ttcp is probably among the most widespread. Pktgen is not a substitute for those tools and complements for some types of tests. The test possibilities is described later in this paper. Worth to note that pktgen cannot do any TCP testing.

Pktgen performance

Performance varies of course with hardware and type of test. Some examples. A single flow of 870 kpps is seen with a PIII 733 MHz over e1000 NIC (64 byte packets) also with bcm5703 in rx2600 (Itanium2, 1Ghz). Aggregated performance of >10 Gbit/s (1500 byte packets) from 12 GIGE NIC's with DUAL XEON 2.66 MHz with hyperthreading (motherboard with 4 independent PCI-X buses) and of 2.4 Mpps with DUAL1.6 GHz Opterons. Tests involving lots of alloc's results in lower sending performance see clone_skb.

Many other things also effects performance: PCI bus speed, PCI vs PCI-X, PCI-PCI Bridge, CPU speed, memory latency, DMA latency, number of MMIO reads/writes etc.

The graph below shows performance on an Dual Opteron 242 with Linux 2.6.7 64 bit with e1000 driver with Intels DUAL NIC (2 x 82546EB) and Intels QUAD NIC (4 x 82546EB and PCI-X bridge towards the NIC uses 120 Mhz). Sending small packets involves many PCI transactions. The graph shows a faster I/O bus gives higher performance as this probably lowers DMA latency. We also see the effects of the PCI-X bridge as the bridge is the difference between the DUAL and QUAD boards,



Getting pktgen to run

Enable CONFIG_NET_PKTGEN in the .config, compile and build pktgen.o either in-kernel or as module, insmod pktgen if needed. Once running, pktgen creates a kernel process on each running CPU. Each process has CPU-affinity. Devices are added to different processes. A device can only belong to one process to give full control of the device to CPU relationship. Modern platforms allow interrupts to be assigned to a CPU (aka IRQ affinity) and this is necessary to minimize cache-line bouncing. Generally, we want the same CPU that generates the packets to also take the interrupts given a symmetrical configuration (several CPUs, several NICs).

On a dual system we see two pktgen processes: [pktgen/0], [pktgen/1]

pktgen is controlled and monitored via the /proc file system. To help document a test configuration and parameters, shell scripts are recommended to setup and start a test. Again referring to our dual system, at start up the files below are created in
 /proc/net/pktgen/ kpktgend_0, kpktgend_1, pgctrl

Assigning devices (e.g. eth1, eth2) to kpktgend_X thread, makes instances of the devices show up in /proc/net/pktgen/ to be further configured at the device level.

A test can be configured to run forever or terminate after a fixed number of packets. Ctrl-C aborts the run. pktgen sends UDP packets to port 9 (discard port) by default. IP, MAC addresses, etc. can be configured. Pktgen packets can hence be identified within the kernel network stack for profiling and testing.

Pktgen versioninfo

The pktgen version is printed in dmesg when pktgen starts. Version info is also in
 /proc/net/pktgen/pgctrl.

Interrupt affinity
When adding a device to a specific pktgen thread, setting /proc/irq/X/smp_affinity binds the associated NIC to the same CPU. This reduces cache line bouncing when freeing skb's. The clone_skb can, to some extent, mitigate the effect of cache line bouncing as skb's is not fully freed. Some experimentation might be required to achieve maximum performance.

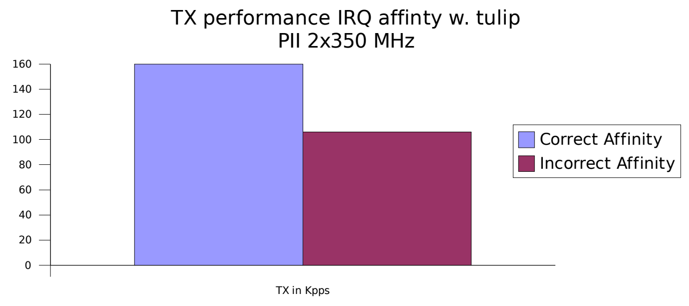
The irq numbers assigned to particular NICs can be seen in /proc/interrupts In the example below, eth0 uses irq 26, eth1 uses irq 27 etc.

```
 26:     933931         0   IO-APIC-level  eth0
 27:     936392         0   IO-APIC-level  eth1
 28:          8    936457   IO-APIC-level  eth2
 29:          8    939310   IO-APIC-level  eth3
```

The example below assigns eth0, eth1 to CPU0, and eth2, eth3 to CPU1

```
echo 1 > /proc/irq/26/smp_affinity
echo 1 > /proc/irq/27/smp_affinity
echo 2 > /proc/irq/28/smp_affinity
echo 2 > /proc/irq/29/smp_affinity
```

The graph below illustrates the performance effects of affinity assignment of PII system.

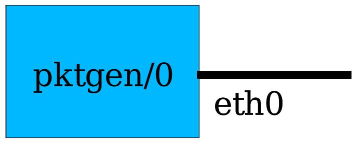


Controlling memory allocation
pktgen uses a trick to increment the skb's refcnt to avoid full path of kfree and alloc when sending identical skb's. This generally gives very high sending rates. For Denial of Service (DoS) and flow tests this technique can not be used as each skb has to be modified.
The parameter clone_skb controls this functionality. Think of clone_skb as the number of packet clones followed by a master packet. Setting clone_skb=0 gives no clones just master packets and clone_skb=1000000 givs 1 master packet followed by one million clones.

Inter-Packet Gap
pktgen can insert an extra artificial delay (ipg) between packets, the unit is nanoseconds. For small delays pktgen busywaits before putting this skb on TX-ring this means traffic is still bursty and somewhat hard to control. Experimentation is probably needed.

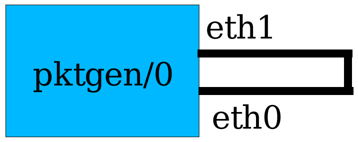
Setup examples

Below a very simple example of pktgen sending on eth0. One only needs to bring up the link.

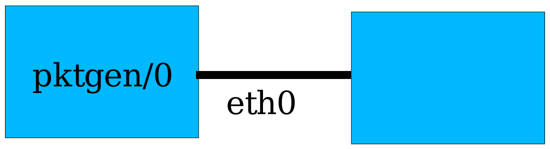


Keeping link up can be done even with the same box using a crossover cable. If generated packets should be seen (ie Received) by the same host just set dstmac to match the NIC on the cross over cable.

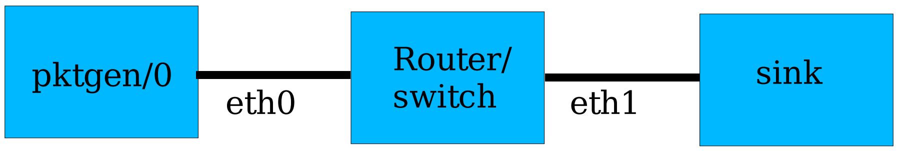
On SMP systems, it's better if the TX flow (pktgen thread) is on a different CPU from the RX flow (set IRQ affinity). One way to test Full Duplex functionality is to connect two hosts and point the TX flows to each other's NIC.



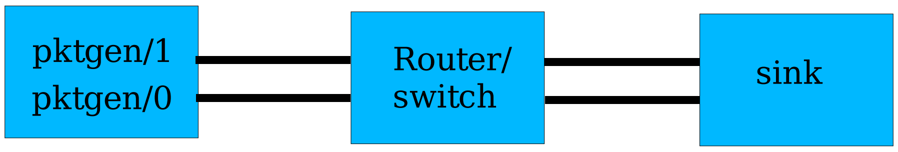
Next, the box with pktgen is used just a packet source to inject packets into a local or remote system. Note you need to configure dstmac of localhost or gateway appropriate.



Below pktgen in a forwarding setup. The sink host receives and discards packets. Of course, forwarding has to be configured on all boxes. It might be possible to use a dummy device instead of sink box.



Forwarding setup using dual devices. Pktgen can use different threads to achieve high load in terms of small packets or concurrent flows.



Viewing pktgen processes

```
/proc/net/pktgen/kpktgend_0

Name: kpktgend_0  max_before_softirq: 10000
Running:
Stopped: eth1
Result: OK: max_before_softirq=10000
```

Viewing pktgen devices

'Parm sections holds configured info. Current holds running stats. Result is printed after run or after interruption for example:

```
/proc/net/pktgen/eth1

Params: count 10000000  min_pkt_size: 60  max_pkt_size: 60
     frags: 0  ipg: 0  clone_skb: 1000000  ifname: eth1
     flows: 0 flowlen: 0
     dst_min: 10.10.11.2  dst_max:
     src_min:   src_max:
     src_mac: 00:00:00:00:00:00  dst_mac: 00:07:E9:13:5C:3E
     udp_src_min: 9  udp_src_max: 9  udp_dst_min: 9  udp_dst_max: 9
     src_mac_count: 0  dst_mac_count: 0
     Flags:
Current:
     pkts-sofar: 10000000  errors: 39192
     started: 1076616572728240us  stopped: 1076616585502839us idle: 1037781us
     seq_num: 11  cur_dst_mac_offset: 0  cur_src_mac_offset: 0
     cur_saddr: 0x10a0a0a  cur_daddr: 0x20b0a0a
     cur_udp_dst: 9  cur_udp_src: 9
     flows: 0
Result: OK: 12774599(c11736818+d1037781) usec, 10000000 (64byte) 782840pps 382Mb/sec
(400814080bps)  errors: 39192
```

10 millon 64 byte packets were sent on eth1 with a rate at 783 kpps

Configuring

Configuring is done via the /proc interface this is easiest done via scripts. Select a suitable script and customize. See the section with example scripts in this paper.

ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/examples/
Other examples has been contributed by Grant Grundler <grundler@parisc-linux.org>
ftp://gsyprf10.external.hp.com/pub/pktgen-testing/

Below is short description for current implemented commands.
Pgcontrol commands

```
start                  Starts sending on all processes
stop
```

Process commands

```
add_device             Add a device to process i.e eth0
rem_device_all         Removes all devices from this process config
max_before_softirq     do_softirq() after sending a number of packets
```

Device commands

```
debug
clone_skb              Number of identical copies of the same packet
                       0 means alloc for each skb. For DoS etc we must
                       alloc new skb's.
clear_counters
```

```
pkt_size              Link packet size minus CRC (4)
min_pkt_size          Range pkt_size setting If < max_pkt_size, then
                      cycle through the port range.
max_pkt_size
frags                 Number of fragments for a packet
count                 Number of packets to send, zero for continious sending
ipg                   Inter-Packet Gap. Artificial gap inserted between packets
                      in nanoseconds
dst                   IP destination address i.e 10.0.0.1
dst_min               Same as dst If < dst_max, then
                      cycle through the port range.
dst_max               Maximum destination IP. i.e 10.0.0..1

src_min               Minimum (or only) source IP. i.e 10.0.0.254 If < src_max, then
                      cycle through the port range.
src_max               Maximum source IP.
dst6                  IPV6 destination address i.e  fec0::1
src6                  IPV6 source address i.e  fec0::2
dstmac                MAC destination adress 00:00:00:00:00:00
srcmac                MAC source adress. If omitted  it's automatically taken
                      from source device
src_mac_count         Number of MACs we'll range through.
                      Minimum' MAC is what you set with srcmac.
dst_mac_count         Number of MACs we'll range through.
                      Minimum' MAC is what you set with dstmac.

flag [name]           Flag to modify behaviour.
                      IPSRC_RND             IP Source is random (between min/max),
                      IPDST_RND             Etc
                      TXSIZE_RND
                      UDPSRC_RND
                      UDPDST_RND
                      MACSRC_RND
                      MACDST_RND

udp_src_min           UDP source port min, If < udp_src_max, then
                      cycle through the port range.
udp_src_max           UDP source port max.
udp_dst_min           UDP destination port min, If < udp_dst_max, then
                      cycle through the port range.
udp_dst_max           UDP destination port max.

stop                  Aborts packet injection. Ctrl-C also aborts generator.
                      Note: It is generally better to use count 0 (forever)
                      and stop the run with Ctrl-C when multiple devices
                      are assigned to one pktgen thread.
                      This avoids some devices finishing before others and
                      skewing the results since we are primarily interested
                      in packets over time, not absolute number of packets.
flows                 Number of concurrent flows
flowlen               Length flows
```

Example scripts

A collection of small tutorial scripts for pktgen are in examples dir.

pktgen.conf-1-1	# 1 CPU 1 dev
pktgen.conf-1-2	# 1 CPU 2 dev
pktgen.conf-2-1	# 2 CPU's 1 dev
pktgen.conf-2-2	# 2 CPU's 2 dev
pktgen.conf-1-1-rdos	# 1 CPU 1 dev w. route DoS
pktgen.conf-1-1-ip6	# 1 CPU 1 dev ipv6
pktgen.conf-1-1-ip6-rdos	# 1 CPU 1 dev ipv6 w. route DoS
pktgen.conf-1-1-flows	# 1 CPU 1 dev multiple flows.

Run in shell: ./pktgen.conf-X-Y It does all the setup including sending. The scripts will need to be adjusted for actually configuration based on which NICs one wishes to test.

The full pktgen.conf-1-1 script

```
#! /bin/sh

#modprobe pktgen


function pgset() {
    local result

    echo $1 > $PGDEV

    result=`cat $PGDEV | fgrep "Result: OK:"`
    if [ "$result" = "" ]; then
         cat $PGDEV | fgrep Result:
    fi
}

function pg() {
    echo inject > $PGDEV
    cat $PGDEV
}

# Config Start Here -----------------------------------------------------------


# thread config
# Each CPU has own thread. Two CPU exammple. We add eth1, eth2 respectivly.

PGDEV=/proc/net/pktgen/kpktgend_0
  echo "Removing all devices"
 pgset "rem_device_all"
  echo "Adding eth1"
 pgset "add_device eth1"
  echo "Setting max_before_softirq 10000"
 pgset "max_before_softirq 10000"


# device config
# ipg is inter packet gap. 0 means maximum speed.

CLONE_SKB="clone_skb 1000000"
# NIC adds 4 bytes CRC
PKT_SIZE="pkt_size 60"

# COUNT 0 means forever
#COUNT="count 0"
COUNT="count 10000000"
IPG="ipg 0"

PGDEV=/proc/net/pktgen/eth1
  echo "Configuring $PGDEV"
 pgset "$COUNT"
 pgset "$CLONE_SKB"
 pgset "$PKT_SIZE"
 pgset "$IPG"
 pgset "dst 10.10.11.2"
 pgset "dst_mac  00:04:23:08:91:dc"


# Time to run
PGDEV=/proc/net/pktgen/pgctrl

 echo "Running... ctrl^C to stop"
 pgset "start"
 echo "Done"

# Result can be vieved in /proc/net/pktgen/eth1
```

Configuration examples

Below is concentrated anatomi of the example scripts. This should be easy to follow.

pktgen.conf-1-2

```
A script fragment assigning eth1, eth2 to CPU on single CPU system.

PGDEV=/proc/net/pktgen/kpktgend_0
 pgset "rem_device_all"
 pgset "add_device eth1"
 pgset "add_device eth2"
```

pktgen.conf-2-2

```
A script fragment assigning eth1 to CPU0 respectivly eth2 to CPU1.

PGDEV=/proc/net/pktgen/kpktgend_0
 pgset "rem_device_all"
 pgset "add_device eth1"

PGDEV=/proc/net/pktgen/kpktgend_1
 pgset "rem_device_all"
 pgset "add_device eth2"
```

pktgen.conf-2-1

```
A script fragment assigning eth1 and eth2 to CPU0 on a dual CPU system.

PGDEV=/proc/net/pktgen/kpktgend_0
 pgset "rem_device_all"
 pgset "add_device eth1"
 pgset "add_device eth2"


PGDEV=/proc/net/pktgen/kpktgend_1
 pgset "rem_device_all"
```

pktgen.conf-1-2

```
A script fragment assigning eth1, eth2 to CPU on single CPU system.

PGDEV=/proc/net/pktgen/kpktgend_0
 pgset "rem_device_all"
 pgset "add_device eth1"
 pgset "add_device eth2"
```

pktgen.conf-1-1-rdos

```
A script fragment for route DoS testing. Note clone_skb 0

PGDEV=/proc/net/pktgen/eth1
 pgset ”clone_skb 0”
 # Random address with in the min-max range
 # pgset "flag IPDST_RND"
 pgset "dst_min 10.0.0.0"
 pgset "dst_max 10.255.255.255"
```

pktgen.conf-1-1-ipv6

Setting device ipv6 addresses.

```
 PGDEV=/proc/net/pktgen/eth1
 pgset "dst6 fec0::1"
 pgset "src6 fec0::2"
```

pktgen.conf-1-1-ipv6-rdos

```
 PGDEV=/proc/net/pktgen/eth1
 pgset ”clone_skb 0”
# pgset "flag IPDST_RND"
 pgset "dst6_min fec0::1"
 pgset "dst6_max fec0::FFFF:FFFF"
```

pktgen.conf-1-1-flows

A script fragment for route flow testing. Note clone_skb 0

```
PGDEV=/proc/net/pktgen/eth1
 pgset ”clone_skb 0”
 # Random address with in the min-max range
 # pgset "flag IPDST_RND"
 pgset "dst_min 10.0.0.0"
 pgset "dst_max 10.255.255.255"
# 8k Concurrent flows at 4 pkts
 pgset "flows 8192"
 pgset "flowlen 4"
```

2x4+2 script

```
# Script contributed by Grant Grundler <grundler@parisc-linux.org>
# Note! 10 devices

PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth3"
pgset "add_device eth5"
pgset "add_device eth7"
pgset "add_device eth9"
pgset "add_device eth11"
pgset "max_before_softirq 10000"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
pgset "add_device eth2"
pgset "add_device eth4"
pgset "add_device eth6"
pgset "add_device eth8"
pgset "add_device eth10"
pgset "max_before_softirq 10000"


# Configure the individual devices

for i in 2 3 4 5 6 7 8 9 10 11
do
        PGDEV=/proc/net/pktgen/eth$i
        echo "Configuring $PGDEV"

        pgset "clone_skb 500000"
        pgset "min_pkt_size 60"
        pgset "max_pkt_size 60"
        pgset "dst 192.168.3.10$i"
        pgset "dst_mac 01:02:03:04:05:0$i"
        pgset "count 0"
done
echo "Running... ctrl^C to stop"
PGDEV=/proc/net/pktgen/pgctrl
pgset "start"

cat /proc/net/pktgen/eth* | fgrep Result
```

Some suggestions for driver/chip testing

When testing a particular driver/chip/platform, start with TX flows using pktgen on the host system to get a sense of which ptkgen parameters are optimal and how well a particular NIC can perform. Try with a range of packet sizes from 64 bytes to 1500 bytes or jumbo frames.

Then start looking at the RX flows on the target platform. Use pktgen to inject packets directed at (or routed through) the target system. Again, vary the packet size as with the TX to get a sense of how many packets a particular NIC (or pair of NICs) can handle a single flow. To isolate driver/chip from other parts of kernel stack pktgen packets can be counted and dropped at various points. See section on detecting pktgen packets.

Then repeat the process with additional flows, one at a time. Multiple flows are trickier since one needs to know I/O bus topology. Typically one tries to balance I/O loads by installing the NICs in the "right" slots or utilizing built-in devices appropriately.

With multiple flows, it is best to use ^C to stop a test run. This prevents any pktgen thread from stopping before others and skewing the test results. Sometimes, one NIC will TX packets faster than another NIC just because of bias in the DMA latency or PCI bus arbiter (to name only two of several possibilities). Using ^C to stop a test run aborts all pktgen threads at once and results in a much better snapshot of how many packets a given configuration could generate. After the ^C is received, pktgen will print the statistics the same as if the test had been stopped by a counter going to zero.

If the tested system has only one interface the dummy interface can be setup as the output device. The advantage with this test is we can test the system at very high load and that results are very reproduceable. Of course other functions as different types of offload and checksumming should be tested as well.

Other testing aspects

Besides knowing the hardware topology, one also needs to know what other workloads are expected to be present on the target system when places in production eg real world use. An FTP server can see quite a different workload than a web server, mail handler, or router etc.

Roughly about 160 kpps seems fill a Gbit link with a FTP server. Of course this can change but may give some idea about packet per second (pps) versus bandwidth for this type of production systems.

For routers the number of routes in the routing table is also an issue as lookup times and other behaviour may be affected. The author has taken snapshots from current Internet routing table IPV4 and IPV6 (BGP) and formed into scripts for this purpose. The routes are added via the ip utility so the tested system does not need any routing connectivity nor routing daemon. Some scripts are available from:

ftp://robur.slu.se/pub/Linux/net-development/inet_routes/

Detecting pktgen packets in kernel

Detecting pktgen packets in kernel. Some times it's very useful to monitor/drop pktgen packets within the driver/network stack either at ingress or egress. The technique is very much the same. The little patchlet below drops at ingress and uses an unused counter.

```
--- linux/net/ipv4/ip_input.c.orig      Mon Feb 10 19:37:57 2003
+++ linux/net/ipv4/ip_input.c   Fri Feb 21 21:42:45 2003
@@ -372,6 +372,23 @@
                IP_INC_STATS_BH(IpInDiscards);
                goto out;
        }

+               {
+                       __u8 *data = (__u8 *) skb->data+20;
+
+                       /* src and dst port 9 --> pktgen */
+
+                       if(data[0] == 0 &&
+                          data[1] == 9 &&
+                          data[2] == 0 &&
+                          data[3] == 9) {
+                              netdev_rx_stat[smp_processor_id()].
fastroute_hit+
+;
+                               goto drop;
+                       }
+               }
+

        if (!pskb_may_pull(skb, sizeof(struct iphdr)))
                goto inhdr_error;
```

Thanks to Grant Grundler, Jamal Hadi Salim Jens Låås, Hans Wassen for comments and useful insights on this paper.

Relevant site ftp://robur.slu.se://pub/Linux/net-development/pktgen-testing/
Good luck with the linux net-development!