PF_RING User Guide

Linux High Speed Packet Capture

Version 6.1.1
July 2015

1.Table of Contents

2. Introduction

PF_RING is a high speed packet capture library that turns a commodity PC into an efficient and cheap network measurement box suitable for both packet and active traffic analysis and manipulation. Moreover, PF_RING opens totally new markets as it enables the creation of efficient application such as traffic balancers or packet filters in a matter of lines of codes.

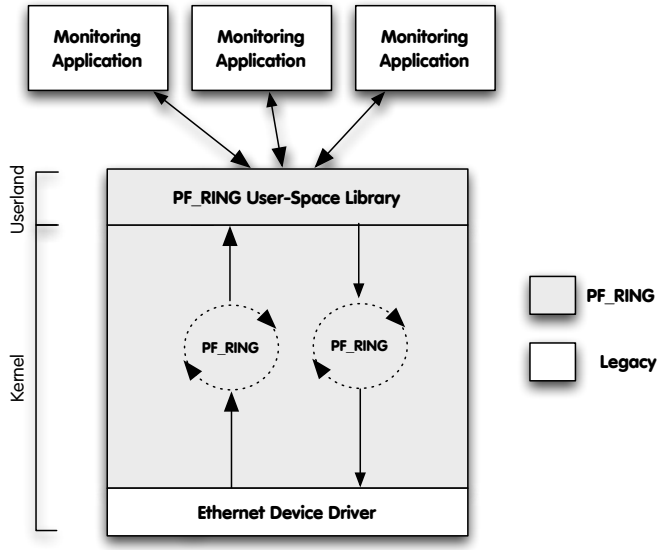
This manual is divided in two parts:

- PF_RING installation and configuration.
- PF_RING SDK.

2.1. What's New with PF_RING User's Guide?

- Release 6.0.0 (Apr 2014)
  - New PF_RING ZC API
- Release 5.4.0 (May 2012)
  - New libzero for zero-copy flexible packet processing on top of DNA.
- Release 5.2.1 (January 2012)
  - New API functions for managing hardware clocks and timestamps.
  - New kernel plugin callbacks.
- Release 4.7.1 (July 2011)
  - Described PF_RING modular library and some modules (DAG, DNA)
- Release 1.1 (January 2008)
  - Described PF_RING plugins architecture.
- Release 1.0 (January 2008)
  - Initial PF_RING users guide.

3.Welcome to PF_RING



PF_RING's architecture is depicted in the figure below.

The main building blocks are:

- The accelerated kernel module that provides low-level packet copying into the PF_RING rings.
- The user-space PF_RING SDK that provides transparent PF_RING-support to user-space applications.
- Specialised PF_RING-aware drivers (optional) that allow to further enhance packet capture by efficiently copying packets from the driver to PF_RING without passing through the kernel. Please note that PF_RING can operate with any NIC driver, but for maximum performance it is necessary to use these specialised drivers that can be found into the drivers/ directory part of the PF_RING distribution.

PF_RING implements a new socket type (named PF_RING) on which user-space applications can speak with the PF_RING kernel module. Applications can obtain a PF_RING handle, and issue API calls that are described later in this manual. A handle can be bound to a:

- Physical network interface.
- A RX queue, only on multi-queue network adapters.
- To the 'any' virtual interface that means packets received/sent on all system interfaces are accepted.

As specified above, packets are read from a memory ring allocated at creation time. Incoming packets are copied by the kernel module to the ring, and read by the user-space applications. No per-packet memory allocation/deallocation is performed. Once a packet has been read from the ring, the space used in the ring for storing the packet just read will be used for accommodating future packets. This means that applications willing to keep a packet archive, must store themselves the packets just read as the PF_RING will not preserve them.

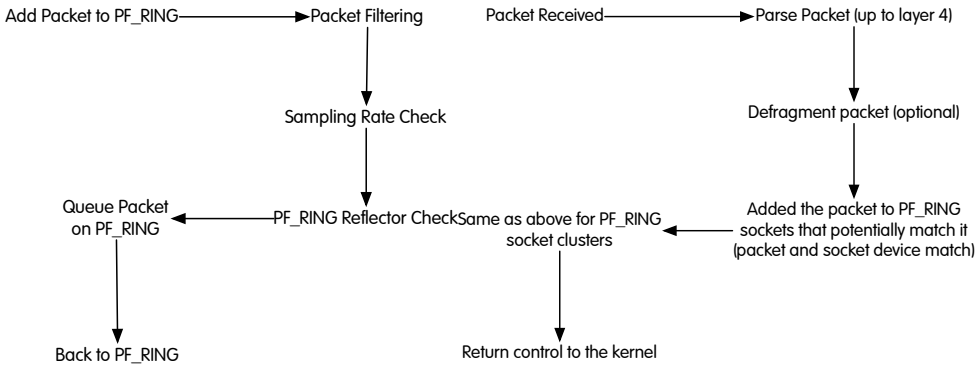
3.1.Packet Filtering

PF_RING supports both legacy BPF filters (i.e. those supported by pcap-based applications such as tcpdump), and also two additional types of filters (named wildcard and precise filters, depending on the fact that some or all filter elements are specified) that provide developers a wide choice of options. Filters are evaluated inside the PF_RING module thus in kernel. Some modern adapters such as Intel 82599-based or Silicom Redirector NICs, support hardware-based filters that are also supported by PF_RING via specified API calls (e.g. pfring_add_hw_rule). PF_RING filters (except hw filters) can have an action

specified, for telling to the PF_RING kernel module what action needs to be performed when a given packet matches the filter. Actions include pass/don't pass the filter to the user space application, stop evaluating the filter chain, or reflect packet. In PF_RING, packet reflection is the ability to transmit (unmodified) the packet matching the filter onto a network interface (this except the interface on which the packet has been received). The whole reflection functionality is implemented inside the PF_RING kernel module, and the only activity requested to the user-space application is the filter specification without any further packet processing.

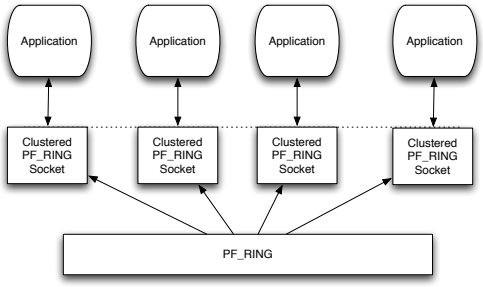
3.2.Packet Journey

The packet journey in PF_RING is quite long before being queued into a PF_RING ring.



3.3.Packet Clustering

PF_RING can also increase the performance of packet capture applications by implementing two mechanisms named balancing and clustering. These mechanisms allow applications, willing to partition the set of packets to handle, to handle a portion of the whole packet stream while sending all the remaining packets to the other members of the cluster. This means that different applications opening PF_RING sockets can bind them to a specific cluster Id (via pfring_set_cluster) for joining the forces and each analyze a portion of the packets.



The way packets are partitioned across cluster sockets is specified in the cluster policy that can be either per-flow (i.e. all the packets belonging to the same tuple <proto, ip src/dst, port src/dst>) that is the default or round-robin. This means that if you select per-flow balancing, all the packets belonging to the same flow (i.e. the 5-tuple specified above) will go to the same application, whereas with round-robin all the apps will receive the same amount of packets but there is no guarantee that packets belonging to the same queue will be received by a single application. So in one hand per-flow balancing allows you to preserve the application logic as in this case the application will receive a subset of all packets but this traffic will be consistent. On the other hand if you have a specific flow that takes most of the traffic, then

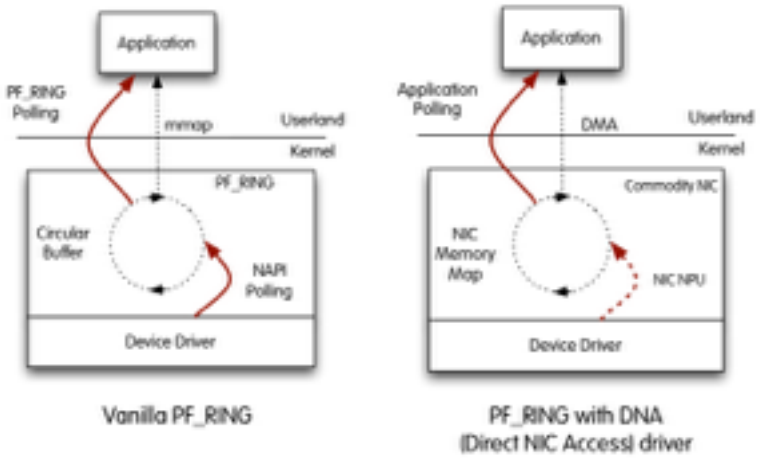
the application that will handle such flow will be over-flooded by packets and thus the traffic will not be heavily balanced.

4.PF_RING Driver Families

As previously stated, PF_RING can work both on top of standard NIC drivers, or on top of specialised drivers. The PF_RING kernel module is the same, but based on the drivers being used some functionality and performance are different.

4.1.DNA (EOL)

For those users that who need maximum packet capture speed with 0% CPU utilization for copying packets to the host (i.e. the NAPI polling mechanism is not used), it is also possible to use DNA (Direct NIC Access) drivers, that allows packets to be read directly from the network interface by simultaneously bypassing both the Linux kernel and the PF_RING module in a zero-copy fashion.



In DNA both RX and TX operations are supported. As the kernel is bypassed, some PF_RING functionality are missing, and they include:

- In kernel packet filtering (BPF and PF_RING filters)
- PF_RING kernel plugins have no effect.

4.2.PF_RING-aware (ZC support)

These drivers (available in PF_RING/driver/PF_RING-aware) are standard drivers with support for the new PF_RING ZC library. These drivers can be used as standard kernel drivers or in zero-copy kernel-bypass mode (using the PF_RING ZC library) adding the prefix "zc:" to the interface name.

Once installed, the drivers operate as standard Linux drivers where you can do normal networking (e.g. ping or SSH). If you open a device in zero copy (e.g. pfcount -i zc:eth1) the device becomes unavailable to standard networking as it is accessed in zero-copy through kernel bypass, as happened with the predecessor DNA. Once the application accessing the device is closed, standard networking activities can take place again. An interface in ZC mode provides the same performance as DNA.

5.PF_RING ZC

PF_RING ZC (Zero Copy) is a flexible packet processing framework that allows you to achieve 1/10 Gbit line-rate packet processing (both RX and TX) at any packet size. It implements zero-copy operations including patterns for inter-process and inter-VM (KVM) communications. It can be considered as the successor of DNA/LibZero that offers a single and consistent API implementing simple building blocks (queue, worker and pool) that can be used from threads, applications and virtual machines.

The following example shows how to create an aggregator+balancer application in 6 lines of code.

```
1 zc = pfring_zc_create_cluster(ID, MTU, MAX_BUFFERS, NULL);
2 for (i = 0; i < num_devices; i++)
3  inzq[i] = pfring_zc_open_device(zc, devices[i], rx_only);
4 for (i = 0; i < num_slaves; i++)
5   outzq[i] = pfring_zc_create_queue(zc, QUEUE_LEN);
6 zw = pfring_zc_run_balancer(inzq, outzq, num_devices,
  num_slaves, NULL, NULL, !wait_for_packet, core_id);
```

PF_RING ZC comes with a new generation of PF_RING aware drivers that can be used both in kernel or bypass mode. Once installed, the drivers operate as standard Linux drivers where you can do normal networking (e.g. ping or SSH). When used from PF_RING they are quicker than vanilla drivers, as they interact directly with it. If you open a device using a PF_RING-aware driver in zero copy (e.g. pfcount -i zc:eth1) the device becomes unavailable to standard networking as it is accessed in zero-copy through kernel bypass, as happened with the predecessor DNA. Once the application accessing the device is closed, standard networking activities can take place again.

PF_RING ZC allows you to forward (both RX and TX) packets in zero-copy for a KVM Virtual Machine without using techniques such as PCIe passthrough. Thanks to the dynamic creation of ZC devices on VMs, you can capture/send traffic in zero-copy from your VM without having to patch the KVM code, or start KVM after your ZC devices have been created. In essence now you can do 10 Gbit line rate to your KVM using the same command you would use on a physical host, without changing a single line of code.

In PF_RING ZC you can use the zero-copy framework even with non-PF_RING aware drivers. This means that you can dispatch, process, originate, and inject packets into the zero-copy framework even though they have not been originated from ZC devices. Once the packet has been copied (one-copy) to the ZC world, from then onwards the packet will always be processed in zero-copy during all his lifetime. For instance the zbalance_ipc demo application can read packet in 1-copy mode from a non-PF_RING aware device (e.g. a WiFI-device or a Broadcom NIC) and send them inside ZC for performing zero-copy operations with them.

6. PF_RING Installation

When you download PF_RING you fetch the following components:

- The PF_RING user-space SDK.
- An enhanced version of the libpcap library that transparently takes advantage of PF_RING if installed, or fallback to the standard behavior if not installed.
- The PF_RING kernel module.
- PF_RING aware drivers for different chips of various vendors.

PF_RING is downloaded by means of GIT or Ubuntu/CentOS repositories as explained in http://www.ntop.org/get-started/.

The PF_RING source code layout is the following:

- doc/
- drivers/
- kernel/
- Makefile
- README
- README.DNA
- README.FIRST
- userland/

You can compile the entire tree typing make (as normal, non-root, user) from the main directory.

6.1. Linux Kernel Module Installation

Please note that for some Linux distributions an installation package is provided (http://packages.ntop.org/). If you choose to install from package please read the section "Configuring a PF_RING Deb/RPM package" below.

In order to compile the PF_RING kernel module you need to have the linux kernel headers (or kernel source) installed.

```
$ cd <PF_RING PATH>/kernel
$ make
$ make install
```

Note that:

- the kernel module installation (via make install) requires root capabilities.
- As of PF_RING 4.x you NO LONGER NEED to patch the linux kernel as in previous PF_RING versions.

7. Running PF_RING

Before using any PF_RING application the pf_ring kernel module should be loaded (as superuser):

```
$ insmod <PF_RING PATH>/kernel/pf_ring.ko [min_num_slots=x]
[enable_tx_capture=1|0] [ enable_ip_defrag=1|0] [quick_mode=1|0]
```

Where:

- min_num_slots
  Min number of ring slots (default — 4096).
- enable_tx_capture
  Set to 1 to capture outgoing packets, set to 0 to disable capture outgoing packets (default — RX +TX).
- enable_ip_defrag
  Set to 1 to enable IP defragmentation, only rx traffic is defragmented.
- quick_mode
  Set to 1 to run at full speed but with up to one socket per interface.

Example:

```
$ cd <PF_RING PATH>/kernel
$ insmod pf_ring.ko min_num_slot=8192 enable_tx_capture=0 quick_mode=1
```

7.1. DNA Drivers

If you want to achieve line-rate packet capture even at 10 Gigabit, you should use these drivers. DNA drivers are part of the PF_RING distribution and can be found in "<PF_RING PATH>/drivers/DNA/".
Currently available DNA drivers are:

- e1000e
- igb
- ixgbe

Please note that:

- the PF_RING kernel module must be loaded before the DNA driver
- in order to correctly configure the device, it is highly recommended to use the load_driver.sh script provided with the drivers (take a look at the script to fine-tune the configuration)

Example loading PF_RING and the ixgbe-DNA driver:

```
$ cd <PF_RING PATH>/kernel
$ insmod pf_ring.ko
$ cd PF_RING/drivers/DNA/ixgbe-X.X.X-DNA/src
$ make
$ ./load_driver.sh
```

7.2. ZC Drivers

ZC is the new generation of DNA drivers, it is highly recommended to use these drivers for line-rate packet capture. ZC drivers are also part of the PF_RING distribution and can be found in "<PF_RING PATH>/drivers/ZC/".

Currently available ZC drivers are:
- e1000e
- igb
- ixgbe

Please note that:
- the PF_RING kernel module must be loaded before the ZC driver
- in order to correctly configure the device, it is highly recommended to use the load_driver.sh script provided with the drivers (take a look at the script to fine-tune the configuration)
- ZC drivers need hugepages, the load_driver.sh script takes care of hugepages configuration

Example loading PF_RING and the ixgbe-ZC driver:

```
$ cd <PF_RING PATH>/kernel
$ insmod pf_ring.ko
$ cd PF_RING/drivers/PF_RING_aware/intel/ixgbe/ixgbe-X.X.X-zc/src
$ make
$ ./load_driver.sh
```

7.3. Configuring a PF_RING Deb/RPM package

In addition to source code it is possible to install PF_RING using the installation packages provided at http://packages.ntop.org/.
Once the "pfring" package, and optionally the ZC drivers, is installed following the procedure on the web page, it is possible to use the init script under /etc/init.d/pf_ring to automate the kernel module and drivers loading. The init script acts as follows:

1. loads the pf_ring.ko kernel module.

2. scans the folders /etc/pf_ring/{zc,dna}/{e1000e,igb,ixgbe}/ searching files:

- {e1000e,igb,ixgbe}.conf containinig the driver parameters
- {e1000e,igb,ixgbe}.start that should be just an empty file

3. loads the drivers whose corresponding {e1000e,igb,ixgbe}.start file is present, unloading the vanilla driver.

4. configures hugepages if a ZC driver has been loaded, reading the configuration from /etc/pf_ring/hugepages. Each line (one per CPU) of the configuration file should contain:

```
node=<NUMA node id> hugepagenumber=<number of pages>
```

Example of a minimal configuration for a dual-port ixgbe card on a uniprocessor:

```
$ mkdir -p /etc/pf_ring/zc/ixgbe
$ echo "RSS=1,1" > /etc/pf_ring/zc/ixgbe/ixgbe.conf
$ touch /etc/pf_ring/zc/ixgbe/ixgbe.start
$ echo "node=0 hugepagenumber=1024" > /etc/pf_ring/hugepages
```

In order to run the init script, after all the files have been configured:

```
$ /etc/init.d/pf_ring start
```

7.4. Checking PF_RING Device Configuration

When PF_RING is activated, a new entry /proc/net/pf_ring is created.

```
# ls /proc/net/pf_ring/
dev  info  plugins_info

# cat /proc/net/pf_ring/info
PF_RING Version          : 6.0.3
Total rings              : 0

Standard (non DNA) Options
Ring slots               : 4096
Slot version             : 16
Capture TX               : Yes [RX+TX]
IP Defragment            : No
Socket Mode              : Standard
Total plugins            : 0
Cluster Fragment Queue   : 0
Cluster Fragment Discard : 0
```

7.5. Libpfring and Libpcap Installation

Both libpfring (userspace PF_RING library) and libpcap are distributed in source format. They can be compiled as follows:

```
$ cd <PF_RING PATH>/userland/lib
$ ./configure
$ make
$ sudo make install
$ cd ../libpcap
$ ./configure
$ make
```

Note that the lib is reentrant hence it's necessary to link your PF_RING-enabled applications also against the -lpthread library.

IMPORTANT
Legacy statically-linked pcap-based applications need to be recompiled against the new PF_RING-enabled libpcap.a in order to take advantage of PF_RING. Do not expect to use PF_RING without recompiling your existing application.

7.6. Application Examples

If you are new to PF_RING, you can start with some examples. The userland/examples folder is rich of ready-to-use PF_RING applications:

```
$ cd <PF_RING PATH>/userland/examples
$ ls *.c
alldevs.c               pfcount.c               pfdump.c
pfutils.c               dummy_plugin_pfcount.c  pfcount_82599.c
pffilter_test.c         pfwrite.c               pcap2nspcap.c
pfcount_aggregator.c    pflatency.c             preflect.c
pcount.c                pfcount_bundle.c        pfmap.c
pfbounce.c              pfcount_dummy_plugin.c  pfsend.c
pfbridge.c              pfcount_multichannel.c  pfsystest.c
$ make
```

For instance, pfcount allows you to receive packets printing some statistics:

```
# ./pfcount -i zc:eth1
Using PF_RING v.6.0.3
...
=========================
Absolute Stats: [64415543 pkts rcvd][0 pkts dropped]
Total Pkts=64415543/Dropped=0.0 %
64'415'543 pkts - 5'410'905'612 bytes [4'293'748.94 pkt/sec - 2'885.39
Mbit/sec]
=========================
Actual Stats: 14214472 pkts [1'000.03 ms][14'214'017.15 pps/9.55 Gbps]
=========================
```

Another example is pfsend, which allows you to send packets (synthetic packets, or optionally a .pcap file can be used) at a specific rate:

```
# ./pfsend -f 64byte_packets.pcap -n 0 -i zc:eth1 -r 5
...
TX rate: [current 7'508'239.00 pps/5.05 Gbps][average 7'508'239.00 pps/
5.05 Gbps][total 7'508'239.00 pkts]
```

7.7. PF_RING Additional Modules

As of version 4.7, the PF_RING library has a new modular architecture, making it possible to use additional components other than the standard PF_RING kernel module. These components are compiled inside the library according to the supports detected by the configure script.

Currently, the set of additional modules includes:

- DAG module.
  This module adds native support for Endace DAG cards in PF_RING. In order to use this module it's necessary to have the dag library (4.x or later) installed and to link your PF_RING-enabled application using the -ldag flag.

- DNA module.

This module can be used to open a device in DNA mode, if you own a supported card and a DNA driver. Please note that the PF_RING kernel module must be loaded before the the DNA driver. With DNA you can dramatically increase the packet capture and transmission speed as the kernel layer is bypassed and applications can communicate directly with the card.
Currently these DNA drivers are available:

- e1000e
- igb
- ixgbe

The drivers are part of the PF_RING distribution and can be found in drivers/DNA/.
With all the drivers you can achieve wire rate at any packet size, both for RX and TX. You can test RX using the pfcount application, and TX using the pfsend application.

Note that in case of TX, the transmission speed is limited by the RX performance. This is because when the receiver cannot keep-up with the capture speed, the ethernet NIC sends ethernet PAUSE frames back to the sender to slow it down. If you want to ignore these frames and thus send at full speed, you need to disable autonegotiation and ignore them (ethtool -A dnaX autoneg off rx off tx off).

- ZC module.
  This module can be used to open a device in ZC mode, if you own a supported card and a PF_RING-aware driver with ZC support. As with DNA, ZC dramatically increases the packet capture and transmission speed as the kernel layer is bypassed and applications can communicate directly with the card.
  Currently these ZC drivers are available:
  - e1000e
  - igb
  - ixgbe

  The drivers are part of the PF_RING distribution and can be found in drivers/PF_RING_aware/ identified by the suffix '-zc'. With all the drivers you can achieve wire rate at any packet size, both for RX and TX. In order to open a device in ZC mode you should use the "zc:" prefix: "zc:ethX".

- Link Aggregation ("multi") module.
  This module can be used to aggregate multiple interfaces in order to capture packets from all of them opening a single PF_RING socket. For example it is possible to open a ring with device name "multi:ethX;ethY;ethZ".

- Libzero consumer ("dnacluster") module.
  This module can be used to attach to a DNA Cluster allowing the application to send and receive packets leveraging on the standard PF_RING API. The sending application has to open a ring by using as device name "dnacluster:X@Y" where X is the cluster identifier and Y is the consumer identifier, or "dnacluster:X" for auto-assigning the consumer identifier.

- Linux TCP/IP Stack injection ("stack") module.
  This module can be used to inject/capture packets to/from the Linux TCP/IP Stack, simulating the arrival/sending of those packets on an interface. The application has to open a ring by using as device name "stack:dnaX" where dnaX is the interface bound to the packets injected into the stack.
  In order to inject a packet to the stack pfring_send() has to be used, in order to capture outgoing packets pfring_recv() has to be used.

8. PF_RING for Application Developers

Conceptually PF_RING is a simple yet powerful technology that enables developers to create high-speed traffic monitor and manipulation applications in a small amount of time. This is because PF_RING shields the developer from inner kernel details that are handled by a library and kernel driver. This way developers can dramatically save development time focusing on the application they are developing without paying attention to the way packets are sent and received.

This chapter covers:

- The PF_RING API overview.
- Extensions to the libpcap library for supporting legacy applications.

Please refer to the doxygen documentation (pfring.h header file) for functions descriptions.

8.1. The PF_RING API

The PF_RING internal data structures should be hidden to the user who can manipulate packets and devices only by means of the available API defined in the include file pfring.h that comes with PF_RING.

8.2. Return Codes

By convention, the library returns negative values for errors and exceptions. Non-negative codes indicate success. In case return code have another meaning, then they are described inside the corresponding function.

8.3. PF_RING Device Name Convention

In PF_RING device names are the same as libpcap and ifconfig. So eth0 and eth5 are valid names you can use in PF_RING. You can specify also a virtual device named 'any' that instructs PF_RING to capture packets from all available network devices.

As previously explained, with PF_RING you can use both the drivers that come with your Linux distribution (thus that are not PF_RING-specific), or some PF_RING-aware drivers (you can find them into the drivers/ directory of PF_RING) that push PF_RING packets much more efficiently than vanilla drivers. If you own a modern multi-queue NIC (e.g. an Intel 10 Gbit adapter), PF_RING allows you to capture packet from a specific RX queue (e.g. ethX@Y). Supposing to have an adapter with Z queues, the queue Id Y, must be in range 0..Z-1. In case you specify a queue that does not exist, no packets will be captured.

As stated in the previous chapter, PF_RING since version 4.7 has a modular architecture. In order to indicate to the library which module we are willing to use, it is possible to prepend the module name to the device name, separated by a colon (e.g. zc:ethX@Y for the ZC module, dna:dnaX@Y for the dna module, dag:dagX:Y for the dag module, "multi:ethA@X;ethB@Y;ethC@Z" for the Link Aggregation module, "dnacluster:A@X" for the Cluster consumer module).

8.4. PF_RING Packet Reflection

Packet reflection is the ability to bridge packets in kernel without sending them to userspace and back. You can specify packet reflection inside the filtering rules.

```
typedef struct {
  ...
  char reflector_device_name[REFLECTOR_NAME_LEN];
  ...
} filtering_rule;
```

In the reflector_device_name you need to specify a device name (e.g. eth0) on which packets matching the filter will be reflected. Make sure NOT to specify as reflection device the same device name on which you capture packets, as otherwise you will create a packet loop.

8.5. PF_RING Packet Filtering

PF_RING allows filtering packets in two ways: precise (a.k.a. hash filtering) or wildcard filtering. Precise filtering is used when it is necessary to track a precise 6-tuple connection <vlan Id, protocol, source IP, source port, destination IP, destination port>. Wildcard filtering is used instead whenever a filter can have wildcards on some of its fields (e.g. match all UDP packets regardless of their destination). If some field is set to zero it will not participate in filter calculation.

8.6. PF_RING In-NIC Packet Filtering

Some multi-queue modern network adapters feature "packet steering" capabilities. Using them it is possible to instruct the hardware NIC to assign selected packets to a specific RX queue. If the specified queue has an Id that exceeds the maximum queueId, such packet is discarded thus acting as a hardware firewall filter.

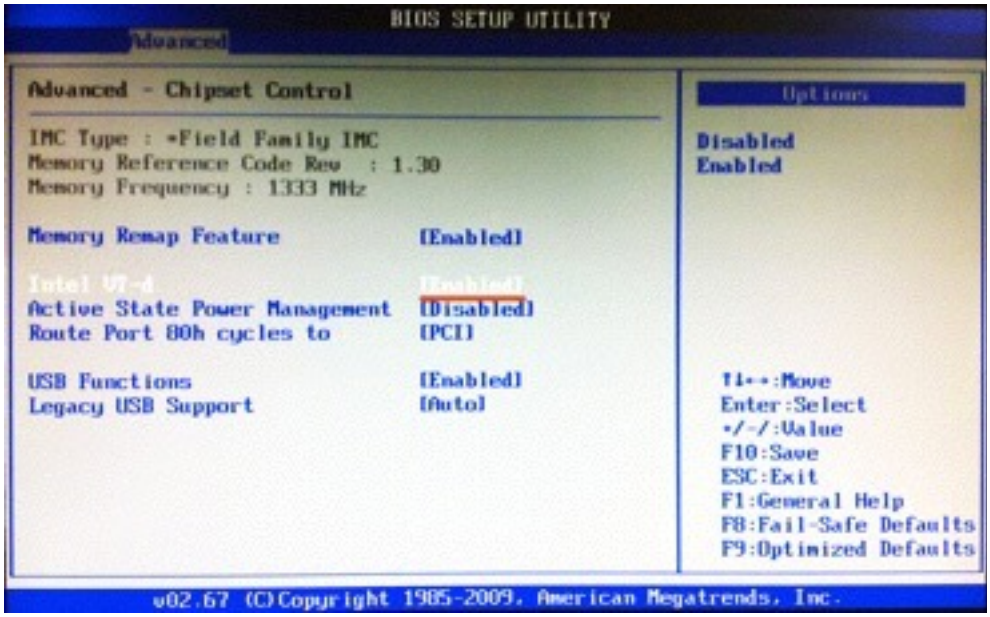
NOTE: Kernel packet filtering is not supported by ZC/DNA.

9. PF_RING ZC/DNA Device Drivers On Virtual Machines

Section 4 contains a brief introduction to the PF_RING ZC/DNA drivers, which allows you to manipulate packets at 10 Gbit wire speed for any packet size. Thanks to Virtualisation Technologies based on IOMMUs (Intel VT-d or AMD IOMMU), it is possible to assign a device to a given guest operating system, benefiting from the PF_RING ZC/DNA drivers acceleration within a VM (Virtual Machine). The following sections show how to configure VMware and KVM (the Linux-native virtualisation system). XEN users can use similar system configurations.

9.1. BIOS Configuration

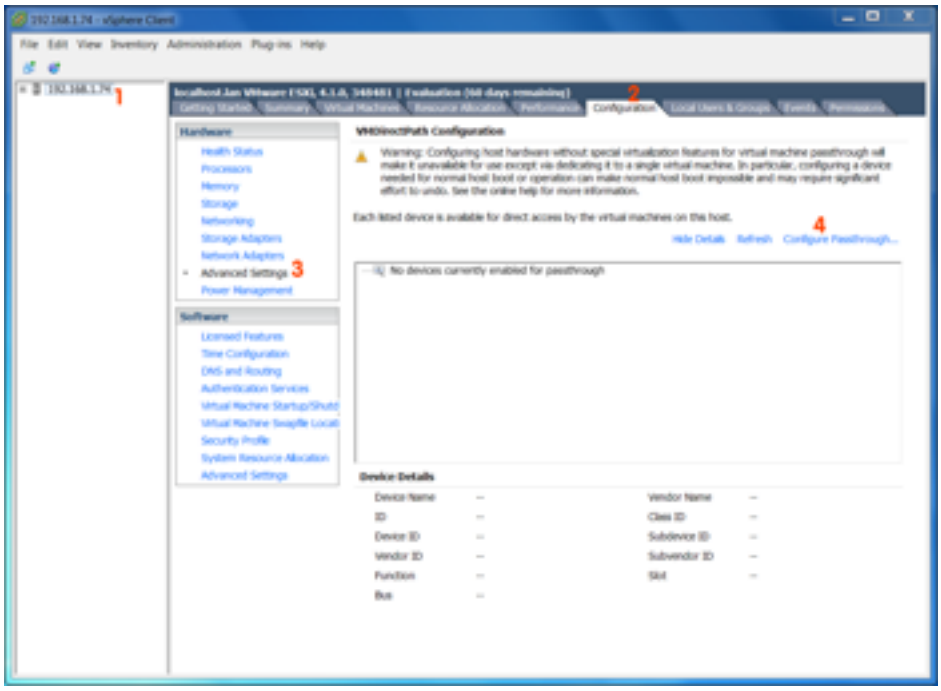
First of all, make sure that your motherboard supports the PCI passthrough and check that it is enabled in your BIOS.



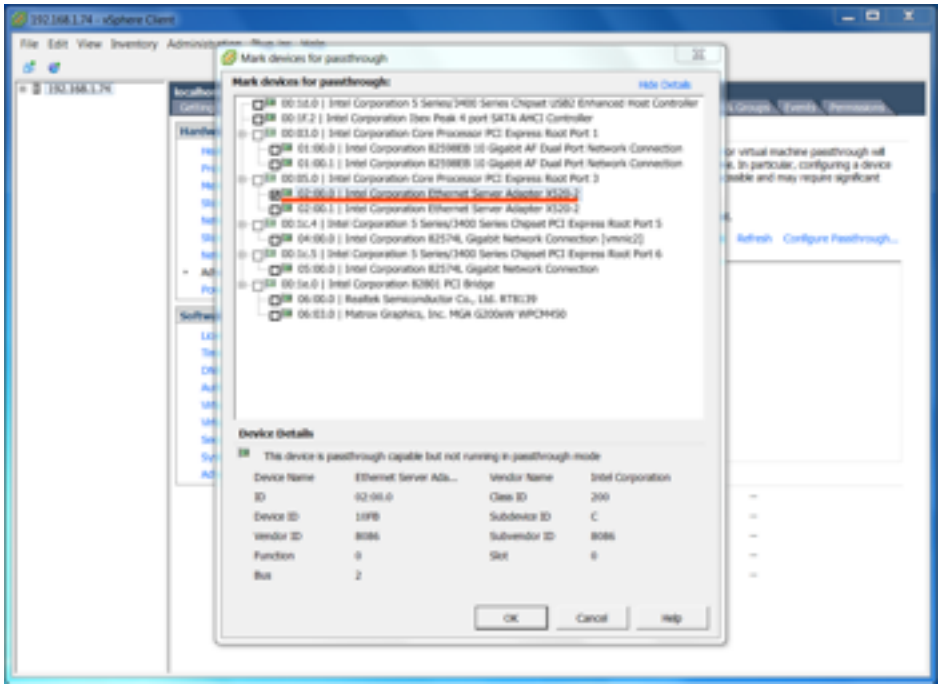
9.2.VMware ESX Configuration

In order to configure the PCI passthrough in VMware, open the vSphere Client and connect to the server.

Select the server, go to "Configuration", "Advanced Settings", "Configure Passthrough".

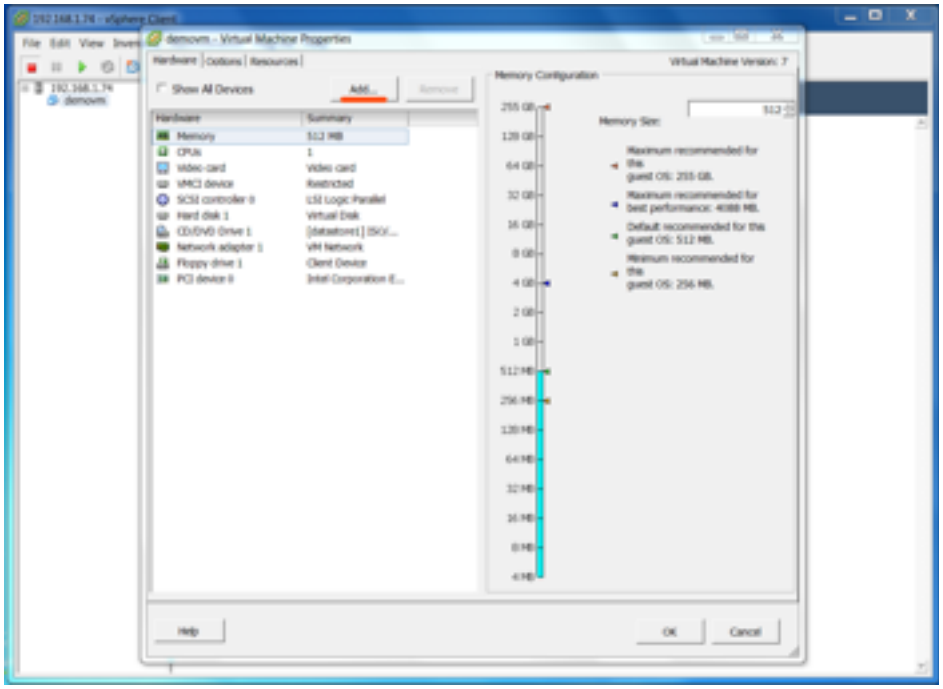


Select the devices you want to assign to the VMs.

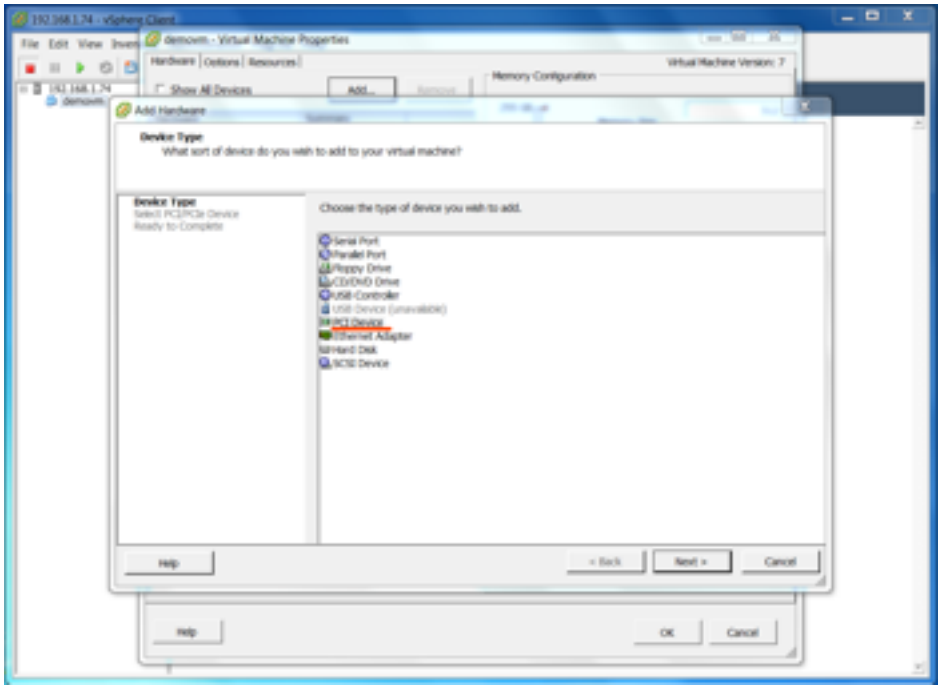


Reboot the server.

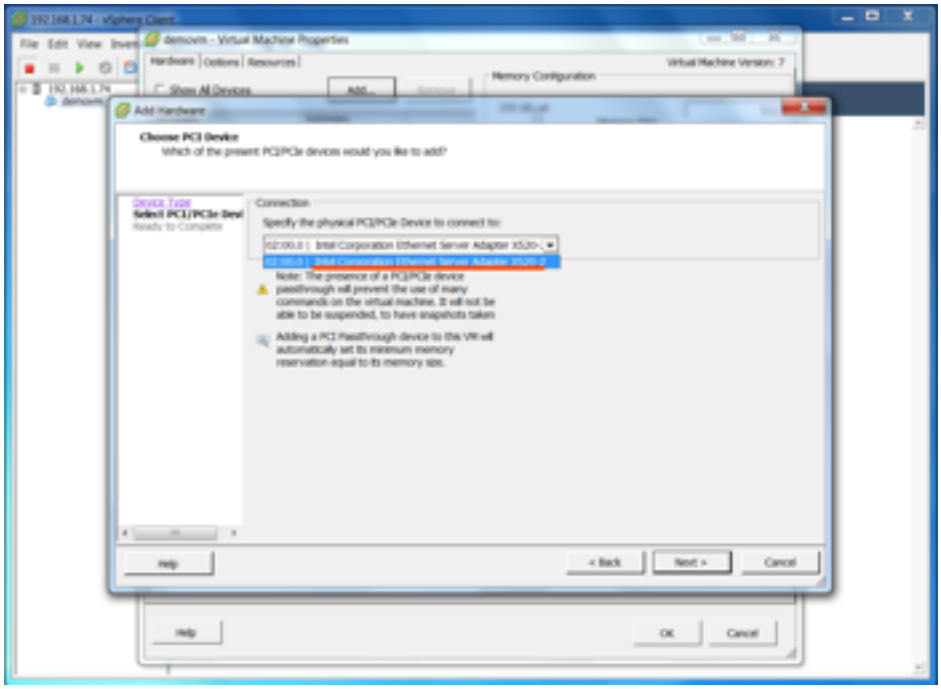
After the reboot, make sure that the VMs where the PCI device will be assigned is in the off state. Open the VM settings, and click on "Add..." in the "Hardware" tab.



Select "PCI Device".



Select the device to assign to the VM.



Boot the VM and install PF_RING with the DNA driver as in the native case.

9.3. KVM Configuration

In order to configure the PCI passthrough with KVM, make sure you have enabled these options in your kernel:

Bus options (PCI etc.)

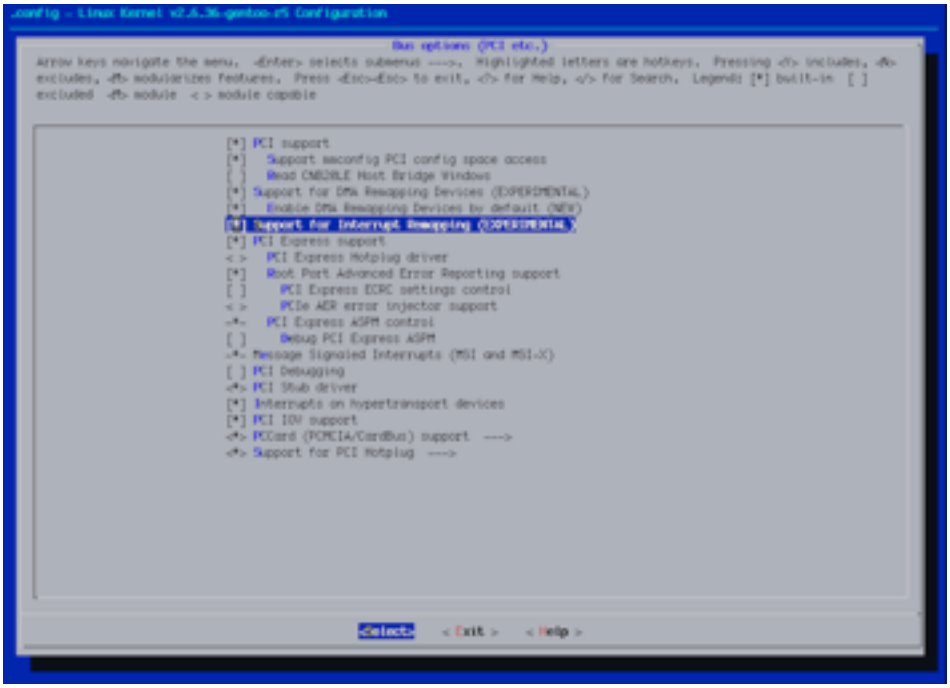
[*] Support for DMA Remapping Devices

[*] Enable DMA Remapping Devices

[*] Support for Interrupt Remapping

<*> PCI Stub driver

```
$ cd /usr/src/linux
$ make menuconfig
```




```
$ make
$ make modules_install
$ make install
```

(or use your distribution-specific way)

Pass "intel_iommu=on" as kernel parameter. For instance, if you are using grub, edit your /boot/grub/menu.lst this way:

```
title Linux 2.6.36
root (hd0,0)
kernel /boot/kernel-2.6.36 root=/dev/sda3 intel_iommu=on
```

Unbind the device you want to assign to the VM from the host kernel driver.

```
$ lspci -n
..
02:00.0 0200: 8086:10fb (rev 01)
..
$ echo "8086 10fb"  > /sys/bus/pci/drivers/pci-stub/new_id
$ echo 0000:02:00.0 > /sys/bus/pci/devices/0000:02:00.0/driver/unbind
$ echo 0000:02:00.0 > /sys/bus/pci/drivers/pci-stub/bind
```

Load KVM and start the VM.

```
$ modprobe kvm
$ modprobe kvm-intel
$ /usr/local/kvm/bin/qemu-system-x86_64 -m 512 -boot c \
      -drive file=virtual_machine.img,if=virtio,boot=on \
      -device pci-assign,host=02:00.0
```

Install and run PF_RING with the ZC/DNA driver as in the native case.