Raw Packet Capture in the Cloud: PF_RING and Network Namespaces

Alfredo Cardigliano
cardigliano@ntop.org
@acardigliano
About ntop

- ntop develops high-performance network traffic monitoring technologies and applications, mostly open source, including:
  - Traffic monitoring (ntopng, nProbe)
  - High-speed packet capture (PF_RING)
  - Deep-packet inspection (nDPI)
  - Traffic recording (n2disk/disk2n)
  - DDoS mitigation (nScrub)
  - IDS/IPS acceleration (Suricata, Bro, Snort)
Packet Capture

• Network Monitoring tools need raw, high-speed, promiscuous packet capture

• Commodity network adapters and device drivers are designed for providing host connectivity
PF_RING

- Open source packet processing framework for Linux

- Originally (2003) designed to accelerate packet capture on commodity hardware, using a mmap approach, patched kernel drivers and in-kernel filtering

- Today it includes kernel-bypass zero-copy drivers (PF_RING ZC) for all Intel adapters and supports almost all FPGAs capture adapters on the market

- All ntop’s applications leverage on PF_RING for packet capture acceleration, up to 100 Gbit/s
PF_RING Architecture

- PF_RING consists of:
  - Kernel module (pf_ring.ko)
  - Userspace library (libpfring)
  - Userspace modules implementing multi-vendor support
  - Patched libpcap for legacy applications
Network Drivers

- Application
- Network Stack
- Driver
- Buffer
- RX
- Registers
- DMA copy
- NIC
- Hardware
- Kernel
- Userspace
PF_RING - Standard Drivers

Userspace

Application

PF_RING.ko

Driver

Kernel

NIC

Hardware

Packet copy

mmap()

Ring buffer

Buffer

Buffer

Buffer

RX

DMA copy

Registers
PF_RING - ZC Drivers

- Application
- Buffer
- Buffer
- Buffer
- zero-copy
- PF_RING ZC
- Driver
- RX
- Registers
- DMA copy
- Intel NIC

Userspace
Kernel
Hardware
PF_RING - FPGA Adapters

- PF_RING natively supports the following vendors (1/10/40/100 Gbit)

  ![Accolade Technology](image1)
  ![Endace](image2)
  ![Exablate](image3)
  ![FiberBlaze](image4)
  ![Mellanox](image5)
  ![Myricom](image6)
  ![Napatech](image7)
  ![Netcope Technologies](image8)

- PF_RING or (our) libpcap based applications transparently select the module by means of the interface name:

  • `tcpdump -i eth1` [Linux drivers]
  • `tcpdump -i zc:eth1` [ZC drivers]
  • `tcpdump -i anic:1` [Accolade]
  • `tcpdump -i nt:1` [Napatech]
Performance

Packet acquisition performance @10Gbit - single thread - Intel Xeon E3 - Intel 82599 single-queue

- **PCAP**: 2,976 Mpps
- **AF_PACKET**: 5,952 Mpps
- **PF_RING**: 8,928 Mpps
- **PF_RING (Quick)**: 11,904 Mpps
- **PF_RING ZC**: 14,880 Mpps
- **FPGAs**: 1,8 Mpps
Many Other Features..

- Load balancing with Clustering or RSS
- PF_RING ZC API for Zero-Copy traffic distribution
- nBPF filtering engine supporting hw filtering
- Wireshark local and remote capture modules
- Acceleration modules for Bro, Snort, Suricata
- Stack Injection support
- KVM support (Host-to-VM datapath)
- Containers support..
The Big Picture

- **PF_RING Applications**
  - Libpfring
    - pf_ring
    - Stack Injection
    - ZC Drivers
  - pf_ring.ko
  - Standard Drivers
  - Standard NIC
  - NAPI
  - Linux Stack
  - Intel NIC
  - FGPA Adapters

- **Legacy Applications**
  - Accolade
  - Fiberblaze
  - Endace
  - Invatech
  - Mellanox
  - Myricom
  - Napatech
  - Netcope

- **PF_RING API**

- **Libpcap API**

- **Kernel**

- **Accolade**
- **Fiberblaze**
- **Endace**
- **Invatech**
- **Mellanox**
- **Myricom**
- **Napatech**
- **Netcope**
Containers, why?

- Docker, LXC, are “Virtual Environments”, with much less overhead compared to Virtual Machines as there is no Guest OS.

- Containers applications in Network monitoring:
  - Easy deployment of traffic monitoring applications (e.g. ntopng-docker)
  - NFV (Network Functions Virtualization, e.g. Firewalls, IDSs)
  - Network traffic monitoring as a service
Cgroups and Namespaces

- Containers are built on the following components:
  - cgroups (Control Groups), limit and account resource usage of a collection of processes including CPU/cores, memory, block I/O, network (tc, iptables).
  - Namespaces, isolate and virtualize system resources of a collection of processes, including PIDs, hostnames, user IDs, network, filesystems.
Network Namespaces

- Network namespaces virtualize the network stack: a network namespace is (logically) another copy of the network stack with its own network interfaces, iptables rules, routing tables, sockets.

- On creation a network namespace only contains the loopback device, then you can create virtual interfaces or move physical interfaces to the namespace.

- A network interface belongs to exactly one network namespace.

- Containers usually use virtual interface pairs (veth driver), eth0 in the container namespace is paired (logically cross-connected) with vethXXX in the host namespace.
Under The Hood

Code is worth a thousand words..
A network namespace is represented by a \textit{struct net} in the Linux kernel.

The default initial network namespace is \textit{init\_net} (instance of \textit{struct net}), it includes all the host interfaces.

For any namespace, \textit{struct net} includes a loopback device, SNMP stats, network tables (routing, neighboring), sockets, procfs/sysfs.
pf_ring.ko

• The PF_RING kernel module is namespaces-aware

• It cannot be otherwise:

  • Even when you are not running containers, you are in a namespace (*init_net*)

  • Most of the methods in the stack have *struct net* as a parameter
In the init function of the pf_ring kernel module, we register with register_pernet_subsys to be notified when network namespaces are created or deleted:

```
static struct pernet_operations ring_net_ops = {
    .init = ring_net_init,
    .exit = ring_net_exit,
};

static int __init ring_init(void)
{
    ...
    register_pernet_subsys(&ring_net_ops);
}
```

When a new network namespace is created, the `ring_net_init` callback is called, and we keep track of all active network namespaces:

```
static int __net_init ring_net_init(struct net *net)
{
    printk("[PF_RING] New network namespace\n");
    netns_add(net);
```
• Opening a new socket the `struct net_proto_family.create` callback is called, this is where we determine if the current task has the capability for opening a raw socket:

```c
static int ring_create(struct net *net, 
                        struct socket *sock, int protocol, int kern) 
{
    if(!ns_capable(net->user_ns, CAP_NET_RAW))
        return -EPERM;
```
In the init function of the pf_ring kernel module, we also set the callback to receive all raw packets from the kernel.

```c
static struct packet_type prot_hook;

static int __init ring_init(void)
{
    ...
    prot_hook.func = packet_rcv;
    prot_hook.type = htons(ETH_P_ALL);
    dev_add_pack(&prot_hook);
}
```
For every packet received from / transmitted to the network interfaces, `packet_rcv` is called, that's where we match the device namespace vs the socket namespace:

```c
static int packet_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev){
...
  while(sk != NULL){ /* foreach pf_ring socket */
    if (net_eq(dev_net(skb->dev), sock_net(sk)))
      /* deliver the packet to the application*/
    ...
  }
...}
```
pf_ring.ko - /proc

- PF_RING exports sockets informations under /proc/net/pf_ring/
- There is a /proc/net/pf_ring/ view for each namespace
- /proc/net/pf_ring/ is created when a new network namespace is registered in ring_net_init(struct net *net)
- proc_mkdir() has struct net.proc_net as parameter, which is the proc root for the actual network namespace

```c
static pf_ring_net *netns_add(struct net *net) {
    ...
    netns->proc_dir = proc_mkdir("pf_ring", net->proc_net);
```
Examples

- `ip netns` can be used to play with network namespaces

- Create a network namespace `ns0`:

  ```
  ip netns add ns0
  ```

  This creates an object at `/var/run/netns/ns0`

- Delete a network namespace `ns1`:

  ```
  ip netns del ns1
  ```

- List all network namespaces:

  ```
  ip netns list
  ```
  `ns0`
Examples

- List all interfaces registered with pf_ring in the host:

  ```
  ls /proc/net/pf_ring/dev/
  eth0  eth1  lo
  ```

- List all interfaces registered with pf_ring in the namespace `ns0`:

  ```
  ip netns exec ns0 bash
  ls /proc/net/pf_ring/dev/
  lo
  ```

- Move a network interface to the network namespace `ns0`:

  ```
  ip link set eth1 netns ns0
  ```

- List all interfaces in `ns0` again:

  ```
  ip netns exec ns0 bash
  ls /proc/net/pf_ring/dev/
  eth1  lo
  ```
Running Docker with PF_RING

• Build the Docker image:

docker build -t ubuntu16 -f Dockerfile .

• Dockerfile

FROM ubuntu:16.04
MAINTAINER ntop.org

RUN apt-get update && \
    apt-get -y -q install wget lsb-release && \
    wget -q http://apt.ntop.org/16.04/all/apt-ntop.deb && \
    dpkg -i apt-ntop.deb

RUN apt-get update && \
    apt-get -y install pfring

ENTRYPOINT ["/bin/bash", "-c"]
Running Docker with PF_RING

• Run docker passing the command to execute (in this case *pfcount*, it prints traffic statistics using pf_ring):

  ```
docker run ubuntu16 pfcount -i eth0
  ```

• Or run docker with an interactive bash prompt:

  ```
sudo docker run -i -t --entrypoint /bin/bash ubuntu16
root@60d98f7e92ba:/#
  ```
Docker - Standard Conf

- Run the container in a new namespace, this will capture traffic received by the container:
  
  `docker run ubuntu16 pfcount -i eth0`

- If you are in the host and want to monitor traffic generated by the container, you can sniff from `vethXXX`

- Find the right `vethXXX`:
  
  ```
  root@964868fc9380:/# cat /sys/class/net/eth0/iflink
  99
  root@host:/# grep -l 99 /sys/class/net/veth*/ifindex
  /sys/class/net/vetha7819c4/ifindex
  ```
Docker - Host Network

- Run the container in the host network namespace with `--network=host`, this will capture traffic received by the host:

  docker run --network=host ubuntu16 pfcount -i eth0
Docker - “Passthrough”

- Run the container in a new namespace, and assign a host interface to the container.

- By default docker does not add network namespaces to /var/run/netns as `ip netns` does, we need to create a symlink and then use `ip netns`:

  ```
  docker ps
  98d96967bc1f ubuntu16
  docker inspect -f '{{ .State.Pid }}' 98d96967bc1f
  9318
  ln -s /proc/9318/ns/net /var/run/netns/ubuntu16
  ip link set eth1 netns ubuntu16
  ip netns exec ubuntu16 ifconfig eth1 up
  ```

- Now it’s possible to run `pfcount -i eth1` inside the container.
Example - ntopng

- Dockerfile.ntopng

FROM ubuntu:16.04
MAINTAINER ntop.org

RUN apt-get update && apt-get -y -q install net-tools wget lsb-release &&
    wget -q http://apt.ntop.org/16.04/all/apt-ntop.deb && \
    dpkg -i apt-ntop.deb
RUN apt-get update && apt-get -y install pfring ntopng

- docker-run-ntopng.sh

#!/bin/bash

IMAGE=$1
IFNAME=$2

ID=$(docker run -d -i -t --entrypoint /bin/bash $IMAGE)
mkdir -p /var/run/netns
PID=$(docker inspect -f '{{ .State.Pid }}' $ID)
ln -sf /proc/$PID/ns/net /var/run/netns/$ID
ip link set $IFNAME netns $ID
ip netns exec $ID ifconfig $IFNAME up

docker exec -it $ID /etc/init.d/redis-server start
docker exec -it $ID ntopng -i $IFNAME
Get Started

Source Code (GitHub)

- git clone https://github.com/ntop/PF_RING.git

Packages

- http://packages.ntop.org
Thank you