High-Speed Passive Packet Capture and Filtering

Luca Deri <deri@ntop.org>

Tutorial Overview

1. Accelerating packet capture and analysis: PF_RING.
2. Layer 7 kernel packet filtering and processing.
3. Even further acceleration: nCap, multithreaded NIC Drivers.
4. Towards 10 Gbit packet capture.

Accelerating Packet Capture and Analysis: PF_RING

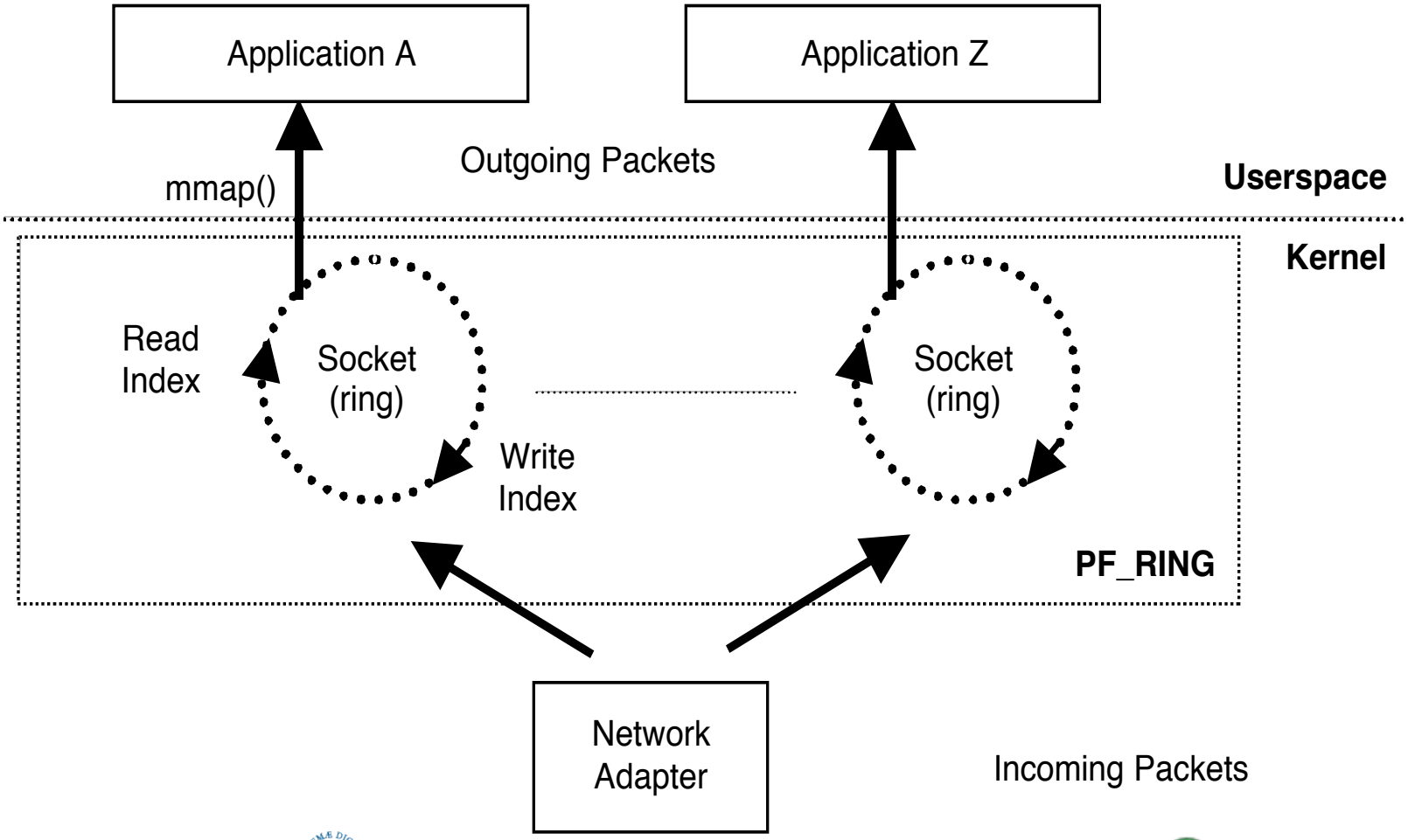
Packet Capture: Open Issues

- Monitoring low speed (100 Mbit) networks is already possible using commodity hardware and tools based on libpcap.
- Sometimes even at 100 Mbit there is some (severe) packet loss: we have to shift from thinking in terms of speed to number of packets/second that can be captured analyzed.
- Problem statement: monitor high speed (1 Gbit and above) networks with common PCs (64 bit/66 Mhz PCI/X/Express bus) without the need to purchase custom capture cards or measurement boxes.
- Challenge: how to improve packet capture performance without having to buy dedicated/costly network cards?

Packet Capture Goals

- Use commodity hardware for capturing packets at wire speed with no loss under any traffic condition.
- Be able to have spare CPU cycles for analyzing packets for various purposes (e.g. traffic monitoring and security).
- Enable the creation of software probes that sport the same performance of hardware probes at a fraction of cost.

Socket Packet Ring (PF_RING)



PF_RING: Benefits

- It creates a straight path for incoming packets in order to make them first-class citizens.
- No need to use custom network cards: any card is supported.
- Transparent to applications: legacy applications need to be recompiled in order to use it.
- No kernel or low-level programming is required.
- Developers familiar with network applications can immediately take advantage of it without having to learn new APIs.

PF_RING: Performance Evaluation

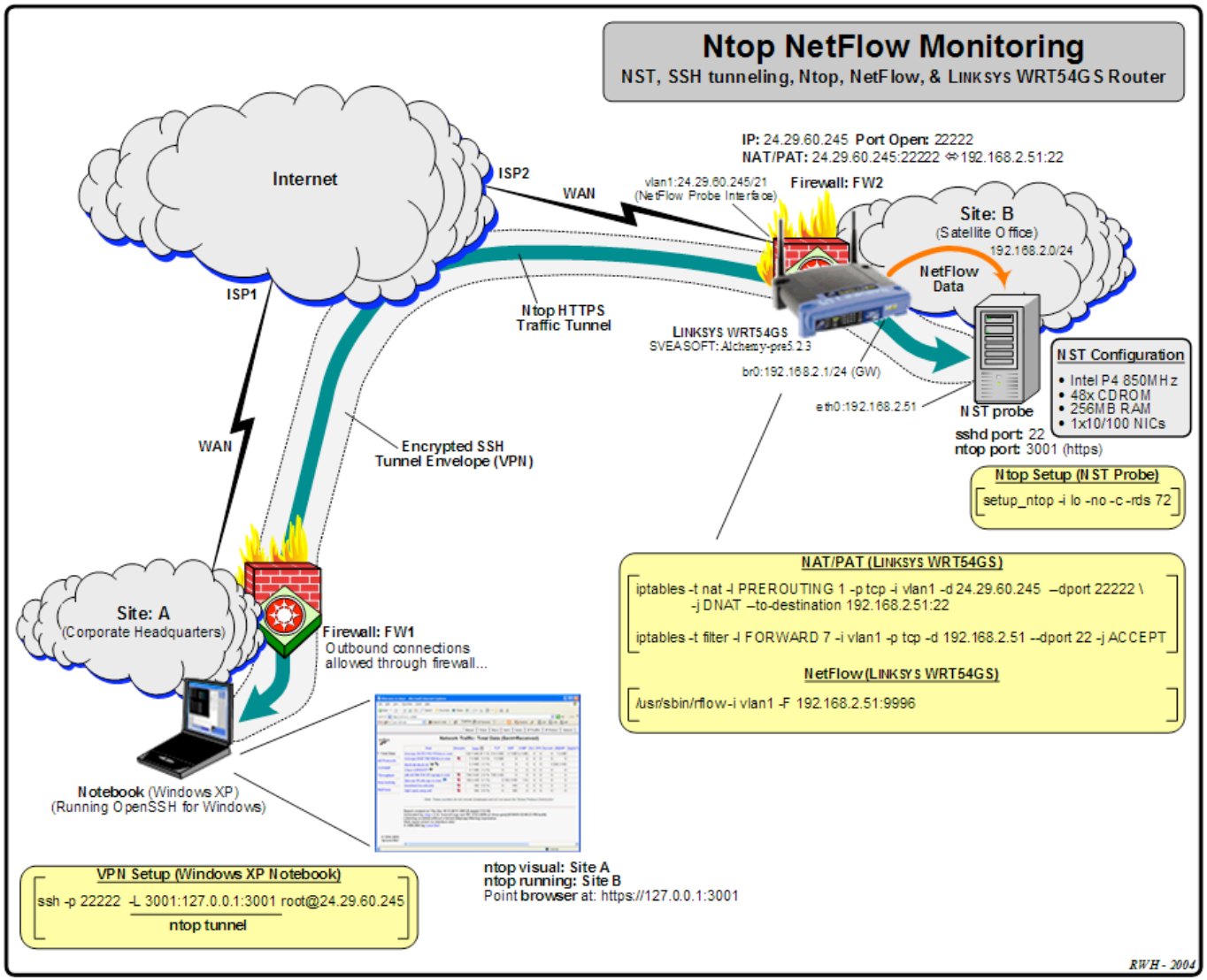
Pkt Size	Kpps	Mbps	% CPU Idle	Wire-Speed
250	259.23	518	> 90%	Yes
250	462.9	925.9	88%	Yes
128	355.1	363.6	86%	Yes
128	844.6	864.8	82%	Yes

Test setup: pcount, full packet size, 3.2 GHz Celeron (single-core) - IXIA 400 Traffic Generator

Socket Packet Ring: Packet Capture Evaluation

- Ability to capture over 1.1 Mpps on commodity hardware with minimal packet sizes (64 bytes).
- Available for Linux 2.4 and 2.6 kernel series.
- Hardware independent: runs on i386, 64bit, MIPS.
- Available for PCs and embedded devices (e.g. OpenWrt, MikroTik routers)

PF_RING on Embedded Devices



http://nst.sourceforge.net/nst/docs/user/ch09s02.html

PF_RING: Packet Filtering [1/2]

- PF_RING has addressed the problem of accelerating packet capture.
- Packet filtering instead is still based on the "ancient" BPF (Berkeley Packet Filter) code used by apps such as tcpdump.
- This means that:
  - Each socket can have up to one filter defined.
  - The packet needs to be parsed in order to match the filter, but the parsing information is not passed to user-space.
  - The BPF filter length can change significantly even if the filter is slightly changed.

PF_RING: Packet Filtering [2/2]

```
# tcpdump -d "udp"
(000) ldh      [12]
(001) jeq      #0x800  jt 2 jf 5
(002) ldb      [23]
(003) jeq      #0x11   jt 4 jf 5
(004) ret      #96
(005) ret      #0
```

```
# tcpdump -d "udp and port 53"
(000) ldh      [12]
(001) jeq      #0x800           jt 2     jf 12
(002) ldb      [23]
(003) jeq      #0x11            jt 4     jf 12
(004) ldh      [20]
(005) jset     #0x1fff          jt 12    jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 14]
(008) jeq      #0x35            jt 11    jf 9
(009) ldh      [x + 16]
(010) jeq      #0x35            jt 11    jf 12
(011) ret      #96
(012) ret      #0
```

Beyond BPF Filtering [1/2]

- VoIP and Lawful Interception traffic is usually very little compared to the rest of traffic.
- Capture starts from filtering signaling protocols and then intercepting voice payload.
- BPF-like filtering is not effective (one filter only).
- It is necessary to add/remove filters on the fly with hundreds active filters.

Beyond BPF Filtering [1/2]

Solution

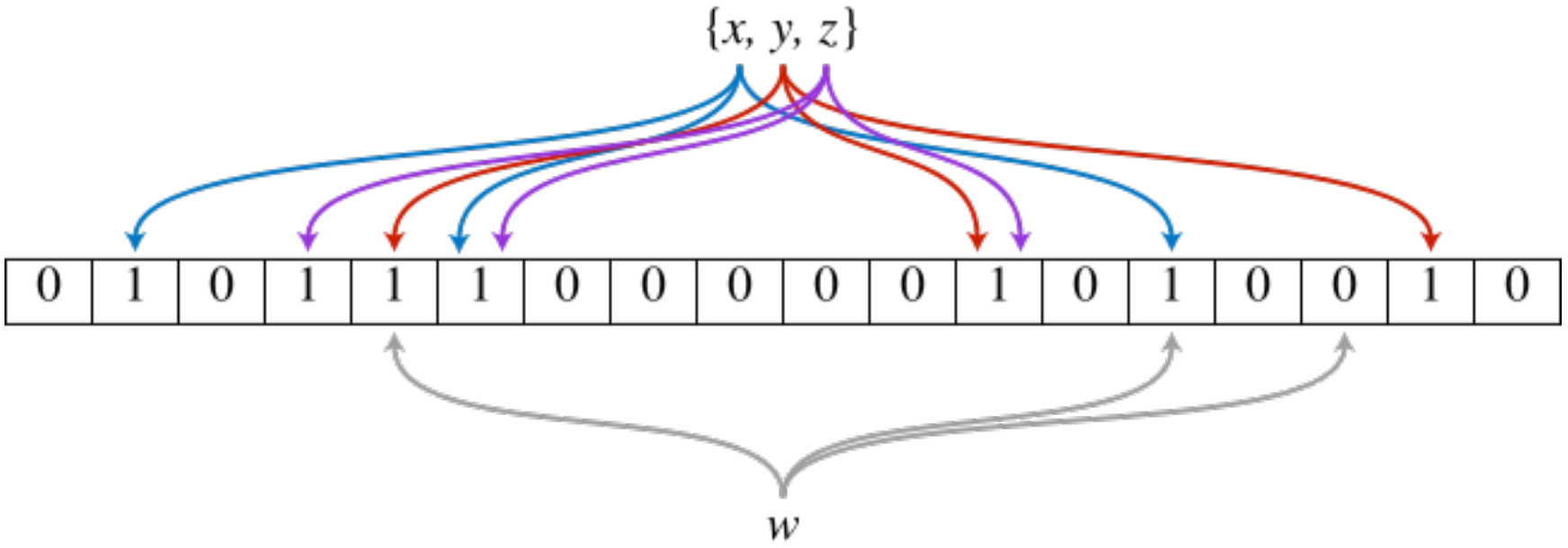
- Filter packets directory on the device driver (not into the kernel layer).
- Implement hash/bloom based filtering (limited false positives).
- Memory effective (doesn't grow as filters are added).
- Implemented on Linux on Intel GE cards. Great performance (virtually no packet loss at 1 Gbit).

Dynamic Bloom Filtering [1/3]

- An empty bloom is a bit array of m bits all set to zero.
- k hash different functions are used to map a key to an array position (0...m-1 hash function range).
- n is the number of elements insert into the dictionary.
- How to add an element: for each k hash function set to 1 the array bit that corresponds to the hash value.
- How to test if an element belongs to the set: for each hash function calculate the hash element value. The element belongs to the set if and only if all the k bits of the hash values are set to 1.
- How to remove an element: fully rebuild the dictionary or use counting blooms.
- False positive rate: $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$
- Optimal number of hash functions: k = (m/n) log(2)

Dynamic Bloom Filtering [2/3]

Insert: hash_1(X), hash_2(X)....hash_n(X)

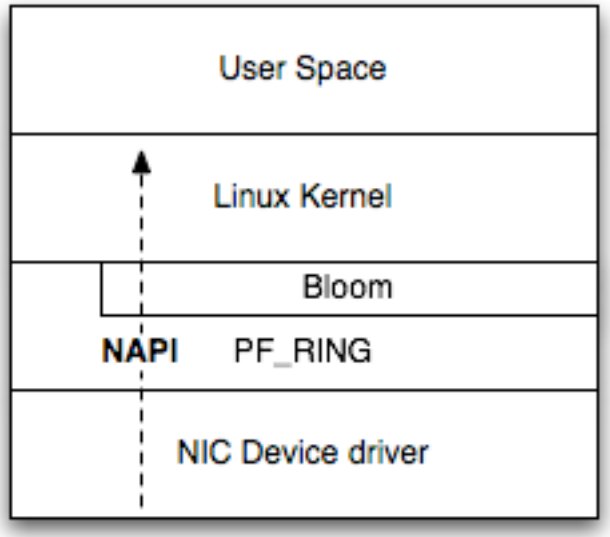


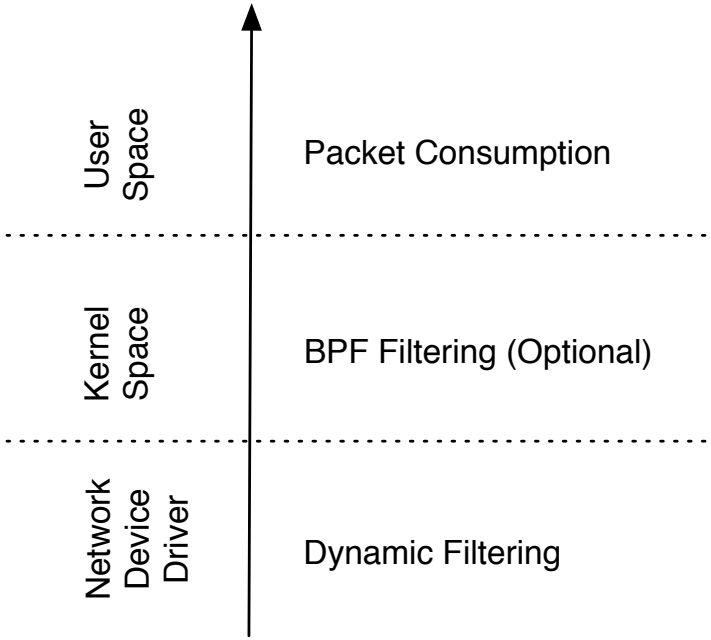
Check for inclusion

Dynamic Bloom Filtering [3/3]

- Ability to specify thousands of different IP packet filters.
- Ability to dynamically add/remove filters without having to interrupt existing applications.
- Only "precise" filters (e.g. host X and port Y) are supported as a new dictionary is required for each filtering criteria (e.g <IP,port,proto> and <IP,port> need two different dictionaries).
- The filter processing speed and memory being used is proportional to the number of filters but independent from their number and complexity.

Dynamic Bloom Filtering





- Available into PF_RING (up to 3.7.x).
- Ability to set per-socket bloom filters

PF_RING: Bloom Evaluation

- Tests performed using a dual Xeon 3.2 GHz CPU injecting traffic with an IXIA 400 traffic generator with 1:256 match rate.
- Packet loss only above 1.8 Mpps (2 x 1 Gbit NICs).
- Ability to specify thousand of filters with no performance degradation with respect to a single filter (only false positive rate increases).

PF_RING Content Inspection

- Ability to (in kernel) search multiple string patterns into packet payload.
- Algorithm based on Aho-Corasick work.
- Ideal for fields like lawful interception and security (including IDSs).
- Major performance improvement with respect to conventional pcap-based applications.

Beyond Bloom Filters

- Bloom filtering has shown to be a very interesting technology for "precise" packet filtering.
- Unfortunately many applications require some features that cannot be easily supported by blooms:
  - port ranges
  - negative expressions (not <expression>)
  - IP address/mask (where mask != /32)
  - in case of match, know what rule(s) matched the filter

Extended PF_RING Filters [1/2]

I have made a survey of network applications and created a list of desirable features, that have then been implemented into PF_RING:

- "Wildcard-ed" filters (e.g. TCP and port 80). Each rule has a rule-id and rules are evaluated according to it.
- Precise 5-tuple filters (VLAN, protocol, IP src/dst, port src/dst) and eventually extend them with other fields (e.g. MPLS label).
- Precise filters (e.g. best match) have priority over (e.g. generic) wilcard-ed filters.

Extended PF_RING Filters [2/3]

- Support of filter ranges (IP and port ranges) for reducing the number of filters.
- Support of mono or bi-directional filters, for reducing number of filters.
- Ability to filter both on standard 5-tuple fields and on L7 fields (e.g. HTTP method=GET).
- Parsing information (including L7 information) needs to be returned to user-space (i.e. do not parse the packet twice).

Extended PF_RING Filters [3/3]

- Per-filter policy in case of match:
  - Stop filtering rule evaluation and drop/forward packet to user-space.
  - Update filtering rule status (e.g. statistics) and stop/ continue rule evaluation without forwarding packet to user-space.
- Filtering rules can pass to user-space both captured packets or statistics/packet digests (this for those apps who need pre-computed values and not just raw packets).

Using PF_RING Filters: HTTP Monitoring [1/5]

- Goal
  - Passively produce HTTP traffic logs similar to those produced by Apache/Squid/W3C.
- Solution
  - Implement plugin that filters HTTP traffic.
  - Forward to userspace only those packets containing HTTP requests for all known methods (e.g. GET, POST, HEAD) and responses (e.g. HTTP 200 OK).
  - All other HTTP packets beside those listed above are filtered and not returned to userspace.
  - HTTP response length is computed based on the "Content-Length" HTTP response header attribute.

Using PF_RING Filters: HTTP Monitoring [2/5]

Plugin Registration (kernel)

```
static int __init http_plugin_init(void)
{
  int rc;

  printk("Welcome to HTTP plugin for PF_RING\n");

  reg.plugin_id                = HTTP_PLUGIN_ID;
  reg.pfring_plugin_filter_skb = http_plugin_plugin_filter_skb;
  reg.pfring_plugin_handle_skb = NULL;
  reg.pfring_plugin_get_stats  = NULL;

  snprintf(reg.name, sizeof(reg.name)-1, "http");
  snprintf(reg.description, sizeof(reg.description)-1, "HTTP protocol analyzer");

  rc = do_register_pfring_plugin(&reg);

  printk("HTTP plugin registered [id=%d][rc=%d]\n", reg.plugin_id, rc);

  return(0);
}
```

Using PF_RING Filters: HTTP Monitoring [3/5]

Plugin Packet Filtering (kernel)

```
static int http_plugin_plugin_filter_skb(filtering_rule_element *rule,
                                         struct pfring_pkthdr *hdr, struct sk_buff *skb,
                                         struct parse_buffer **parse_memory)
{
  struct http_filter *rule_filter = (struct http_filter*)rule->rule.extended_fields.filter_plugin_data;
  struct http_parse *packet_parsed_filter;

  if((*parse_memory) == NULL) {
    /* Allocate (contiguous) parsing information memory */
    (*parse_memory) = kmalloc(sizeof(struct parse_buffer*), GFP_KERNEL);
    if(*parse_memory) {
      (*parse_memory)->mem_len = sizeof(struct http_parse);
      (*parse_memory)->mem = kcalloc(1, (*parse_memory)->mem_len, GFP_KERNEL);
      if((*parse_memory)->mem == NULL) return(0); /* no match */
    }

    packet_parsed_filter = (struct http_parse*)((*parse_memory)->mem);
    parse_http_packet(packet_parsed_filter, hdr, skb);
  } else {
    /* Packet already parsed: multiple HTTP rules, parse packet once */
    packet_parsed_filter = (struct http_parse*)((*parse_memory)->mem);
  }

  return((rule_filter->the_method & packet_parsed_filter->the_method) ? 1 /* match */ : 0);
}
```

Using PF_RING Filters: HTTP Monitoring [4/5]

Plugin Packet Parsing (kernel)

```
static void parse_http_packet(struct http_parse *packet_parsed,
                              struct pfring_pkthdr *hdr,
                              struct sk_buff *skb) {
  u_int offset = hdr->parsed_pkt.pkt_detail.offset.payload_offset; /* Use PF_RING Parsing */
  char *payload = &skb->data[offset];

  /* Fill PF_RING parsing information datastructure just allocated */
  if((hdr->caplen > offset) && !memcmp(payload, "OPTIONS", 7))      packet_parsed->the_method = method_options;
  else if((hdr->caplen > offset) && !memcmp(payload, "GET", 3))      packet_parsed->the_method = method_get;
  else if((hdr->caplen > offset) && !memcmp(payload, "HEAD", 4))    packet_parsed->the_method = method_head;
  else if((hdr->caplen > offset) && !memcmp(payload, "POST", 4))    packet_parsed->the_method = method_post;
  else if((hdr->caplen > offset) && !memcmp(payload, "PUT", 3))     packet_parsed->the_method = method_put;
  else if((hdr->caplen > offset) && !memcmp(payload, "DELETE", 6))  packet_parsed->the_method = method_delete;
  else if((hdr->caplen > offset) && !memcmp(payload, "TRACE", 5))   packet_parsed->the_method = method_trace;
  else if((hdr->caplen > offset) && !memcmp(payload, "CONNECT", 7)) packet_parsed->the_method = method_connect;
  else if((hdr->caplen > offset) && !memcmp(payload, "HTTP ", 4))   packet_parsed->the_method = method_http_status_code;
  else packet_parsed->the_method = method_other;
}
```

Using PF_RING Filters: HTTP Monitoring [5/5]

Userland application

```
if((pd = pfring_open(device, promisc, 0)) == NULL) { printf("pfring_open error\n"); return(-1); }

pfring_toggle_filtering_policy(pd, 0); /* Default to drop */

memset(&rule, 0, sizeof(rule));
rule.rule_id = 5, rule.rule_action = forward_packet_and_stop_rule_evaluation;
rule.core_fields.proto =  6 /* tcp */;
rule.core_fields.port_low = 80, rule.core_fields.port_high = 80;
rule.plugin_action.plugin_id = HTTP_PLUGIN_ID; /* HTTP plugin */
rule.extended_fields.filter_plugin_id = HTTP_PLUGIN_ID; /* Enable packet parsing/filtering */
filter = (struct http_filter*)rule.extended_fields.filter_plugin_data;
filter->the_method = method_get | method_http_status_code;

 if(pfring_add_filtering_rule(pd, &rule) < 0) { printf("pfring_add_filtering_rule() failed\n"); return(-1); }

 while(1) {
  u_char buffer[2048];
  struct pfring_pkthdr hdr;

  if(pfring_recv(pd, (char*)buffer, sizeof(buffer), &hdr, 1) > 0)
    dummyProcesssPacket(&hdr, buffer);
}

pfring_close(pd);
```

Advanced PF_RING Filtering: NetFlow [1/5]

- Goal
  - Using PF_RING for packet capture and processing in user space, the target performance (just packet capture, not flow generation) is:
    - Standard Intel driver: 550 Mbps
    - Enhanced Intel driver (see later in this presentation): 950 Mbps
  - Ability to compute NetFlow/IPFIX flows at wire speed at 1 Gbit regardless of the CPU being used and packet size.
- Solution
  - Use PF_RING plugin to "pack" packets belonging to the same flow. This acts as level-1 NetFlow cache.
  - Periodically (e.g. once every 1-5 sec) flush cache flows by forwarding packet digest to userspace via PF_RING.
  - Forwarded packets contains a header, used for computing flows, but not the packet as this is unnecessary. Each PF_RING slot can host several packets/ flows if needed.

Advanced PF_RING Filtering: NetFlow [2/5]

- Each PF_RING cache entry contains exactly the same information necessary to generate flows.

```
struct pkt_aggregation_info {
  u_int32_t num_pkts, num_bytes;
  struct timeval first_seen, last_seen;
};

struct netflow_l1_pf_ring_packet_cache {
  /* Standard PF_RING fields */
  u_int16_t eth_type;   /* Ethernet type */
  u_int16_t vlan_id;    /* VLAN Id or NO_VLAN */
  u_int8_t  l3_proto, ipv4_tos;   /* Layer 3 protocol/TOS */
  u_int32_t ipv4_src, ipv4_dst;   /* IPv4 src/dst IP addresses */
  u_int16_t l4_src_port, l4_dst_port; /* Layer 4 src/dst ports */
  u_int8_t tcp_flags;   /* TCP flags (0 if not available) */

    struct pkt_aggregation_info aggregation; /* NetFlow */
};
```

- NetFlow cache is walked (for searching expired flows) by user-space application through a dummy call to getsockopt() that allows to keep kernel code simple as there's no need to spawn a kernel thread for walking the cache.

Advanced PF_RING Filtering: NetFlow [3/5]

- The PF_RING cache has (by default) 4096 entries and it is implemented as an array (circular buffer) in order to keep the code simple.
- User-space application can modify cache policy/size when PF_RING is instrumented.
- The plugin is activated with a wildcard-ed rule of 'any' so that any IP packet matching the filter can be computed.
- Modus Operandi
  - When an incoming packet is received, PF_RING parses it, and then it is passed to the plugin.
  - Using parsing information the packet is searched in the cache
    - If found the cache entry is updated
    - if not found the packet is added to the cache. In case the cache slot where the packet is supposed to be stored is already occupied, the slot is flushed (i.e. the entry is forwarded to the userland) and the packet is accommodated.

Advanced PF_RING Filtering: NetFlow [4/5]

- Using the kernel cache, packets are "merged" in kernel without any userland application intervention.
- In-kernel packet merging does not require any memory/packet copy and it's very fast as the packet is already in the CPU cache (thanks to Intel RSS/ DCA, see later in this presentation).
- The "merging rate" increases (in efficiency) with flows speed. In other terms the cache is more efficient as flows are faster. Example:
  - 1 Gbit (1.48 Mpps) flow with minimal packets.
  - Kernel cache duration of 3 sec (i.e. flows older than this duration are exported)
  - "Vanilla" PF_RING: in 3 sec the application receives 4.44 Million packets (3 x 1.48 Mpps).
  - In-kernel cache generates 1 flow for the same amount of traffic.

Advanced PF_RING Filtering: NetFlow [5/5]

- Performance Evaluation
  - Testbed: 1.86 GHz Intel CoreDuo (cost < 100 Euro), IXIA 400 Traffic generator, minimal packet size (64 bytes), Intel e1000 driver, Full 1 Gbit stream, with packet rotation, nProbe (home grown NetFlow probe) used as probe.
- Vanilla PF_RING + nProbe: 100% CPU, ~600 Kpps processed with no loss.
- Kernel NetFlow PF_RING plugin + nProbe: ~60-70% CPU used, wire-rate, 64-128 bytes packet, with no packet-loss.
- Comparison:
  - spare CPU cycles compared to vanilla PF_RING.
  - wire-speed with minimal packet size.
  - not suitable (yet) for generating flows with packet payload information (e.g. HTTP URL).

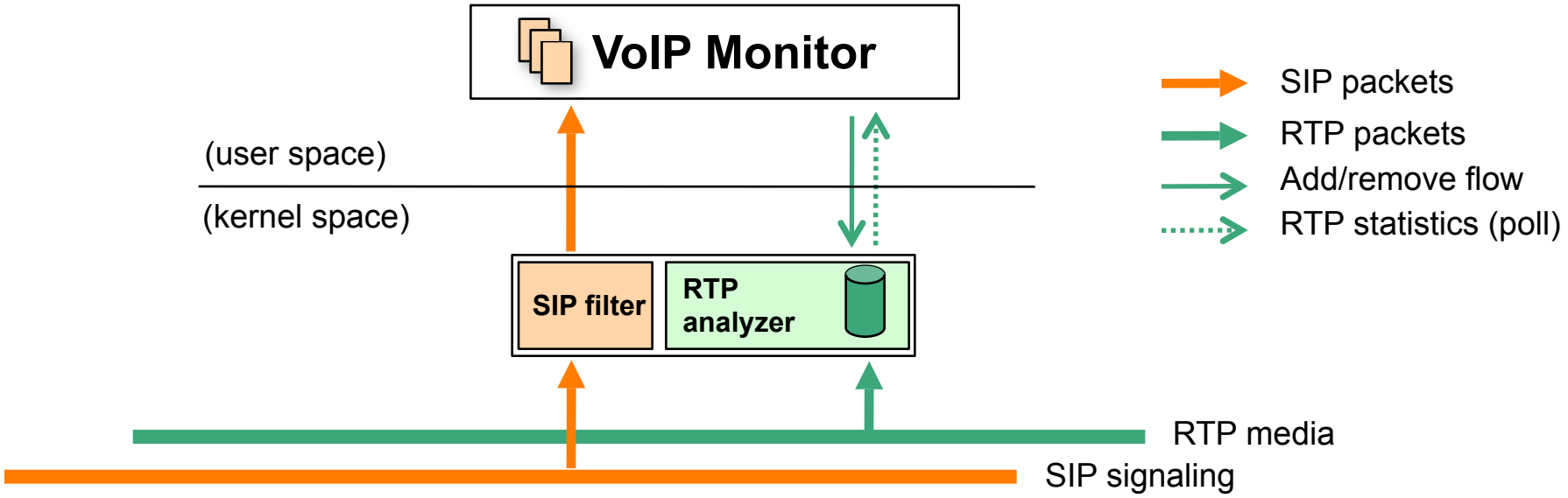
Dynamic PF_RING Filtering: VoIP [1/6]

- Goal
  - Track VoIP (SIP+RTP) calls at any rate on a Gbit link using commodity hardware.
  - Track RTP streams and calculate call quality information such as jitter, packet loss,without having to handle packets in userland.

- Solution
  - Code a PF_RING plugin for tracking SIP methods and filter-out:
    - Uninteresting (e.g. SIP Options) SIP methods
    - Not well-formed SIP packets
    - Dummy/self calls (i.e. calls used to keep the line open but that do not result in a real call).
  - Code a RTP plugin for computing in-kernel call statistics (no pkt forwarding).
  - The SIP plugin adds/removes a precise RTP PF_RING filtering rule whenever a call starts/ends.

Dynamic PF_RING Filtering: VoIP [2/6]

- Before removing the RTP rule though PF_RING library calls, call information is read and then the rule is deleted.
- Keeping the call state in userland and do not forward RTP packets, allows the code of VoIP monitoring applications to be greatly simplified.
- Furthermore as SIP packets are very few compared to RTP packets, the outcome is that most packets are not forwarded to userland contributing to reduce the overall system load.



Dynamic PF_RING Filtering: VoIP [3/6]

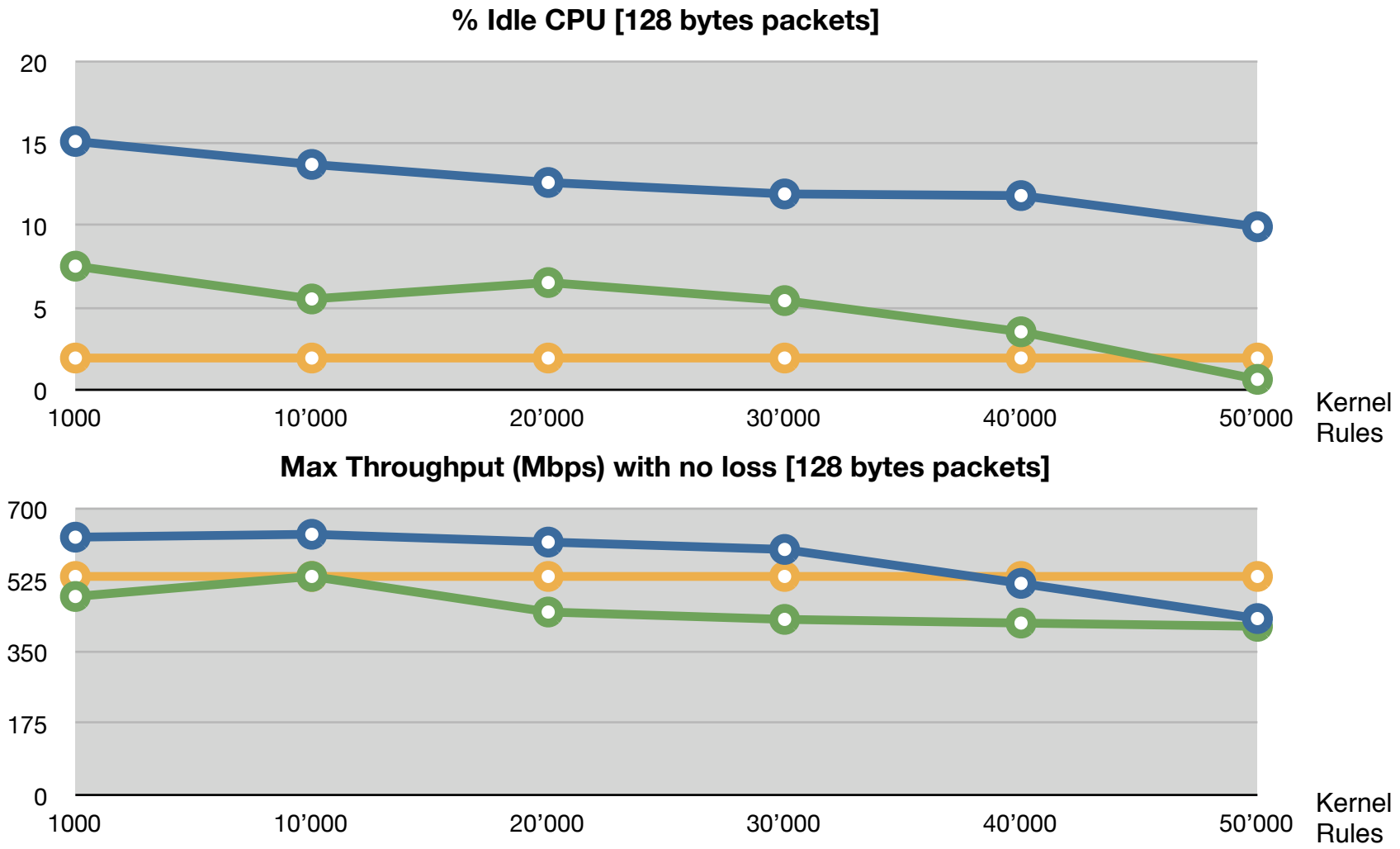
- SIP Plugin
  - It allows filters to be set on SIP fields (e.g. From, To, Via, CallID)
  - Some fields are not parsed but the plugin returns an offset inside the SIP packet (e.g. SDP offset, used to find out the IP:port that will be used for carrying the RTP/RTCP streams).
  - Forwarded packets contain parsing information in addition to SIP payload.

- RTP Plugin
  - It tracks RTP (mono/bi-directional) flows.
  - The following, per-flow, statistics are computed: jitter, packet loss, malformed packets, out of order, transit time, max packet delta.
  - Developers can decide not to forward packets (this is the default behavior) or to forward them (usually not needed unless activities like lawful interception need to be carried on).

Dynamic PF_RING Filtering: VoIP [4/6]

- Validation
  - A SIP test tool and traffic generator (sipp) is used to create synthetic SIP/RTP traffic.
  - A test application has been developed: it receives SIP packets (signaling) and based on them it computes RTP stats.
  - A traffic generator (IXIA 400) is used to generate noise in the line and fill it up. As RTP packets are 100 bytes in average, all tests are run with 128 bytes packets.
  - The test code runs on a cheap single-core Celeron 3.2 GHz (cost < 40 Euro).
  - In order to evaluate the speed gain due to PF_RING kernel modules, the same test application code is tested:
    - Forwarding SIP/RTP packets to userland without exploiting kernel modules (i.e. the code uses the standard PF_RING).
    - RTP packets are not forwarded, SIP packets are parsed/filtered in kernel.

Dynamic PF_RING Filtering: VoIP [5/6]



Dynamic PF_RING Filtering: VoIP [6/6]

- Validation Evaluation
  - In-kernel acceleration has demonstrated that up until 40K rules, kernel plugins are faster than a dummy application that simply captures packets without any processing.
  - On a Gbit link it is possible to have up to ~10K concurrent calls with G.711 (872 Mbit) or ~30K calls with G.729 (936 Mbit). This means that with the current setup and a slow processor, it is basically possible to monitor a medium/ large ISP.

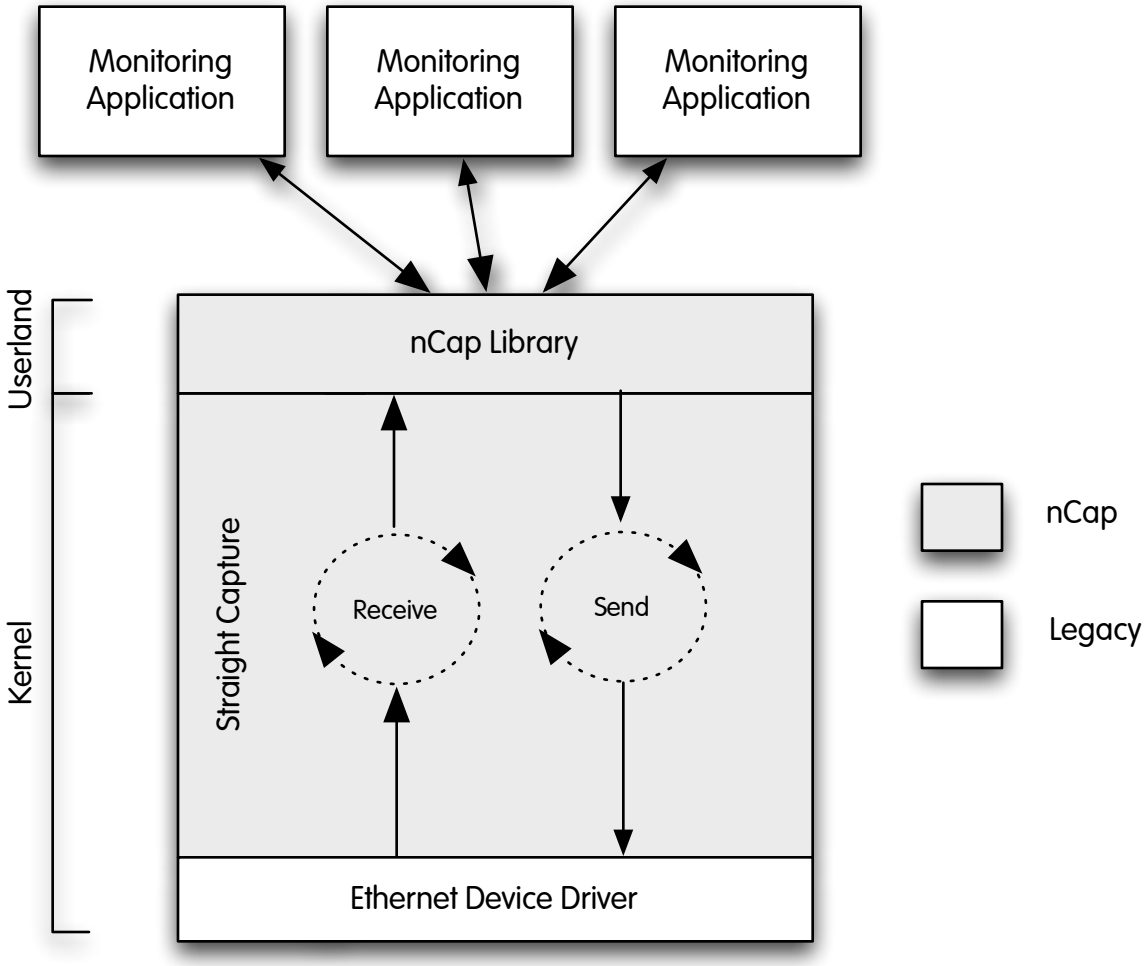
- Future Work Items
  - The plugins are currently used as building blocks glued together by means of the user-space applications.
  - The SIP plugin can dynamically add/remove RTP rules, so that it is possible to avoid (even for SIP) packet forwarding and send to userland just VoIP statistics for even better performance figures.

Even further acceleration: nCap Multithreaded NIC Drivers.

Beyond PF_RING

- PF_RING has shown to be an excellent packet capture acceleration technology compared to vanilla Linux.
- It has reduced the cost of packet capture and forward to userland.
- However it has some design limitations as it requires two actors for capturing packets that result in sub-optimal performance:
  - kernel: copy packet from NIC to ring.
  - userland: read packet from ring and process it.
- PF_RING kernel modules demonstrated that limiting packet processing in user-space by moving it to kernel results in major performance improvements.
- A possible solution is to map a NIC to user-space and prevent the kernel from using it.

nCap [1/2]


Monitoring Application
Monitoring Application
Monitoring Application
Userland
Kernel
nCap Library
Straight Capture
Receive
Send
Ethernet Device Driver
nCap
Legacy

nCap [2/2]

- Evaluation
  - Technology developed by the author in 2004.
  - High-speed packet capture: 1 Gbit wire-rate (1.48 Mpps) using a 3 GHz P4 with HyperThreading.
  - High-speed packet generation: as fast as a a hardware traffic generator at a portion of the price.
  - Solution similar to http://sourceforge.net/projects/channel-sock/.

- Drawbacks
  - Only one application at time can use the NIC (as most accelerator cards including Endace and Napatech).
  - Driver-dependent (it supported only Intel 1 Gbit cards).
  - nCap on 2.4 Linux kernel series is much faster than nCap on 2.6 due to scheduler changes: realtime patches required.

Enhanced NIC Drivers [1/5]

- The current trend in computer architecture is towards multi-core systems.
- Currently 4-core CPUs are relatively cheap, some manufacturers announced a 64-core x86 CPU by the end of 2008.
- Exploiting multi-core in userland applications is relatively simple by using threads.
- Exploiting multi-core in kernel networking code is much more complex.
- Linux kernel networking drivers are single-threaded and the model is still the same since many years.
- It's not possible to achieve good networking performance unless NIC drivers are also accelerated and exploit multi-core.

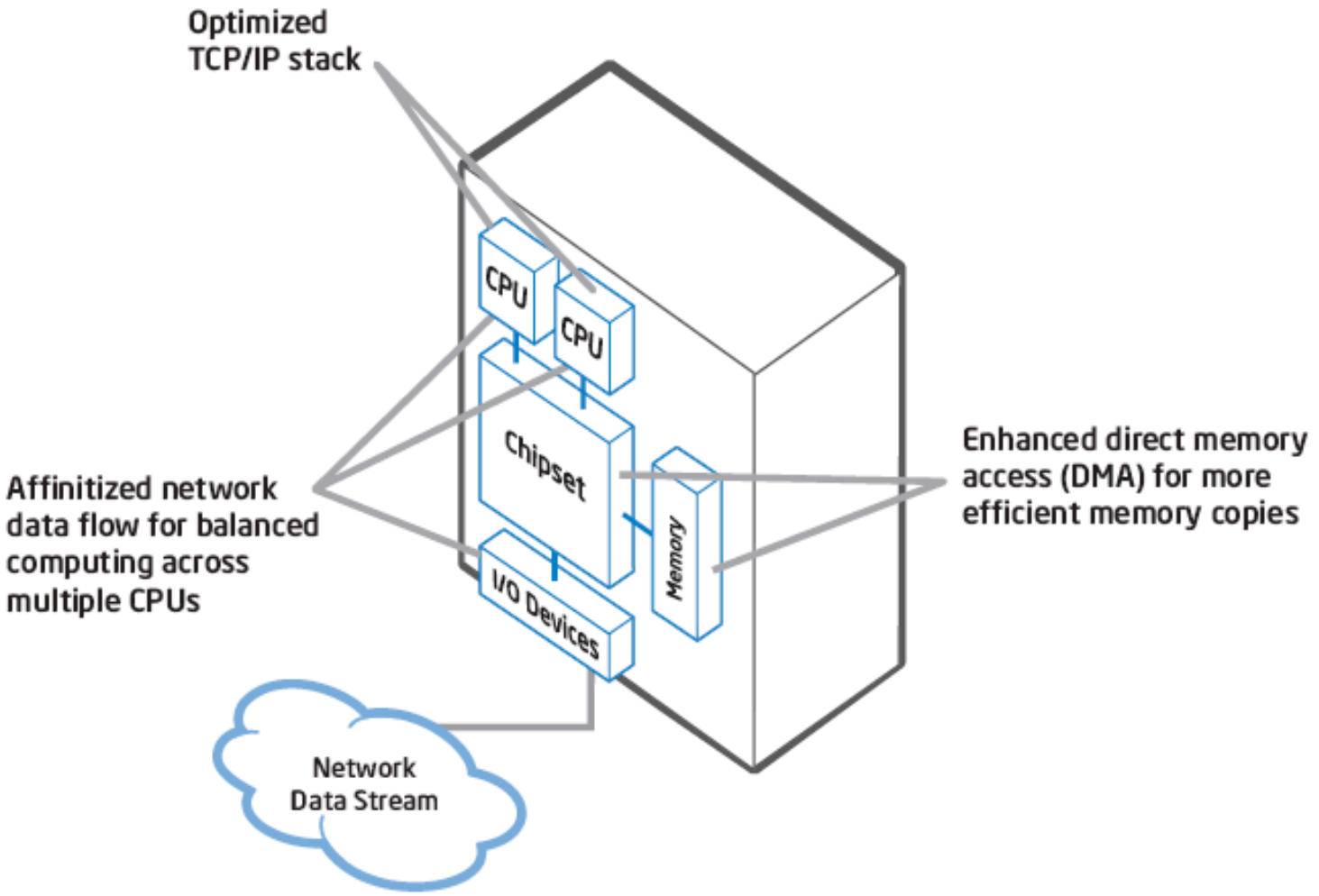
Enhanced NIC Drivers [2/5]

Intel has recently introduced a few innovations in the Xeon 5000 chipset series that have been designed to accelerate networking applications:

- I/O Acceleration Technology (I/OAT)
  - QuickData Technology
  - Direct Cache Access (DCA) move packets directly on CPU's cache
  - Multiple TX/RX queues (one per core) that improve system throughput and utilization
  - MSI-X, low latency interrupts and load balancing across multiple RX queues.
  - RSS (Receive-Side Scaling) balances packets across RX queue/cores
  - Low-latency with adaptive and flexible interrupt moderation

In a nutshell: increase performance by distributing workloads across available CPU cores.

Enhanced NIC Drivers [3/5]



Enhanced NIC Drivers [4/5]

- In order to enhance and accelerate packet capture under Linux, the author has implemented a new Linux driver for Intel 1 and 10 Gbit cards that features:
  - Multithreaded capture (one thread per RX queue, per NIC adapter). The number of rings is the number of cores (i.e. a 4 core system has 4 RX rings). Caveat: interrupts can be disabled per-ring but are enabled per card.
  - RX packet balancing across cores based on RSS: one core, one RX ring.
  - Driver-based packet filtering (PF_RING filters port into the driver) for stopping unwanted packets at the source.
  - Development drivers for Intel 82598 (10G) and 82575 (1G) ethernet controllers.

Enhanced NIC Drivers [5/5]

- Preliminary performance results:
  - 10 Gbit
    - The testbed is a 4 x 1 G ports IXIA 400 traffic generator that are mixed into a 10G stream using a HP ProCurve 3400cl-24 switch.
    - A dual 4-core 3 GHz Xeon has been used for testing.
    - Using the accelerated driver it is possible to driver-filter 512 bytes packets at 7 Gbps with a 1:256 packet forward rate to user-space with no loss.
  - 1 Gbit
    - The same testbed for 10G has been used.
    - The same packet filtering policy applied to 2 x 1 Gbit ports works with no loss and with minimal (~10%) CPU load.
    - The performance improvement also affects packet capture. For instance with a Core 2 Duo 1.86 GHz, packet capture improved from 580 Kpps to over 900 Kpps.

References

- libpcap: http://www.tcpdump.org
- PF_RING: http://www.ntop.org/PF_RING.html
- BPF: http://www.tcpdump.org/papers/bpf-usenix93.pdf
- Bloom Filters: http://en.wikipedia.org/wiki/Bloom_filter
- Intel I/OAT: http://www.intel.com/technology/ioacceleration/

- ntop Web Site: http://www.ntop.org/
- Author Papers: http://luca.ntop.org

All work is open-source and released under GPL.