High-Speed Traffic Capture and Analysis Using Open-Source Software and Commodity Hardware

Part 1: Packet Capture

Luca Deri <deri@ntop.org>

Overview

- Accelerating packet capture and analysis: PF_RING.
- Layer 7 kernel packet filtering and processing.
- Direct NIC Access: PF_RING DNA.
- Towards 10 Gbit packet capture using commodity hardware.
- Strong Multicore NIC: Tilera Tile64

Accelerating Packet Capture and Analysis: PF_RING

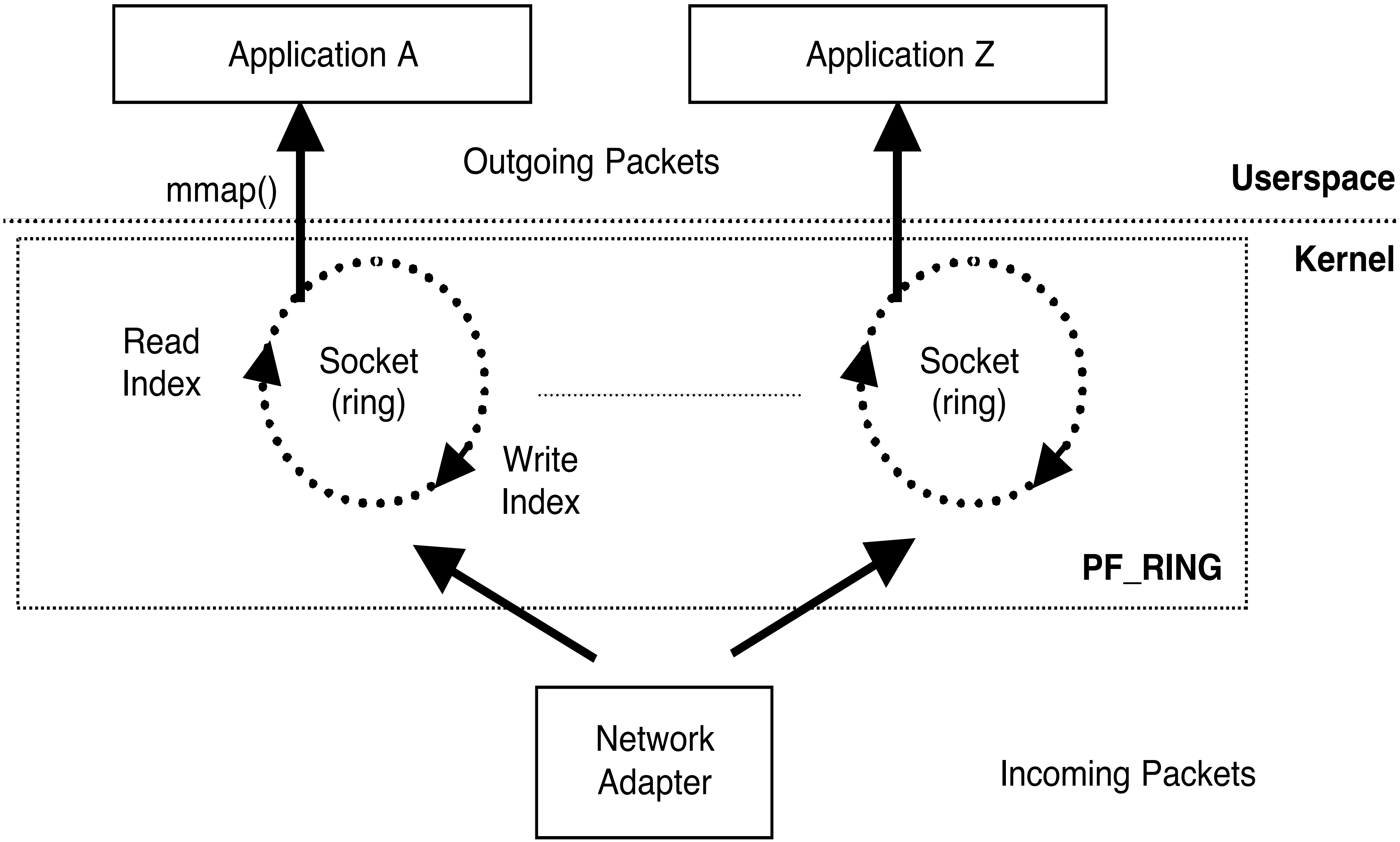
Packet Capture: Open Issues

- 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 term 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?

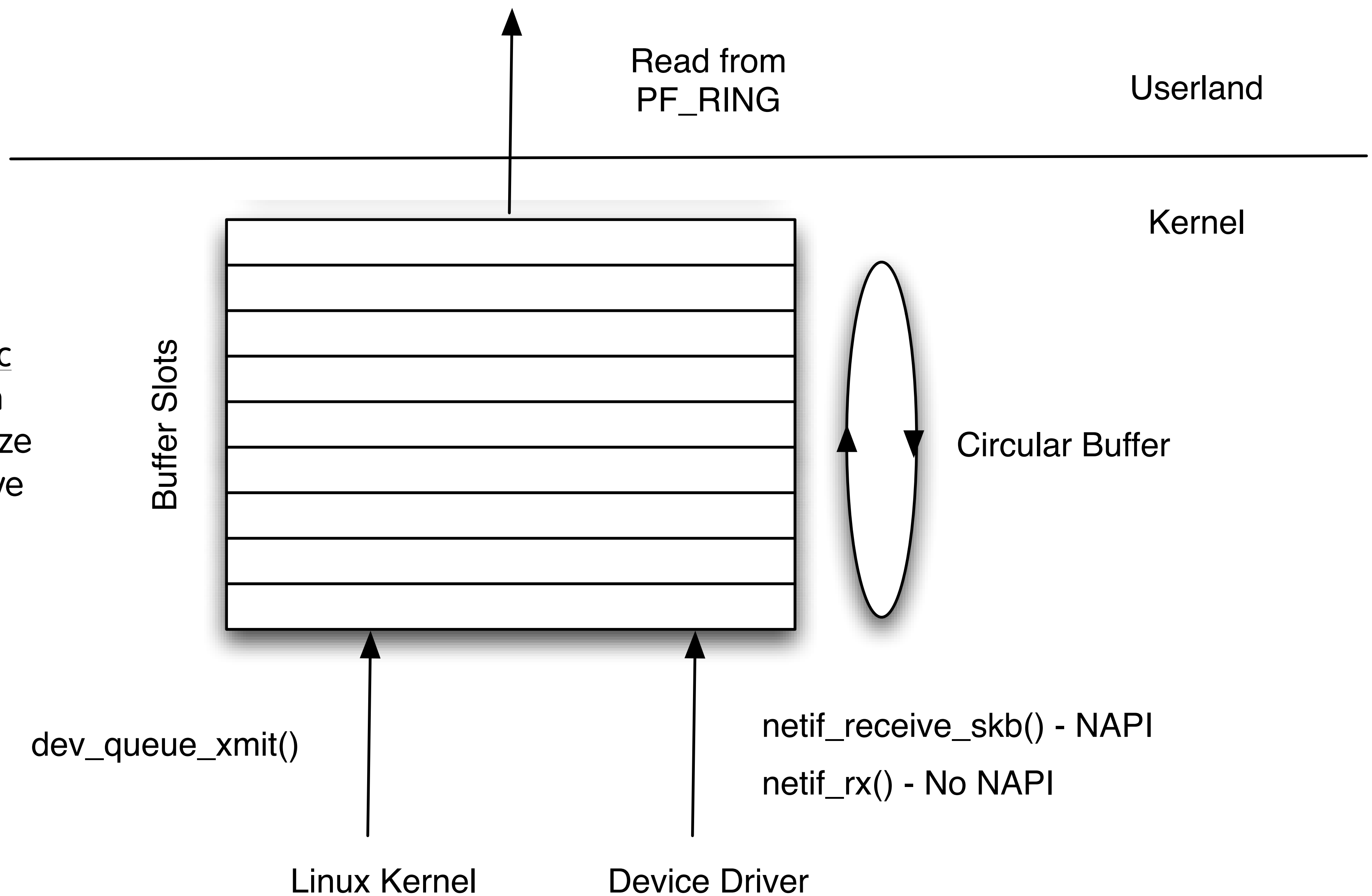
Packet Capture Goals

- 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 Internals



Slot size is dynamic as they are filled in according to the size of packets that have been received

http://en.wikipedia.org/wiki/Circular_buffer

PF_RING Packet Journey [1/2]

Packet Received → Parse Packet (up to layer 4)

↓

Defragment packet (optional)

↓

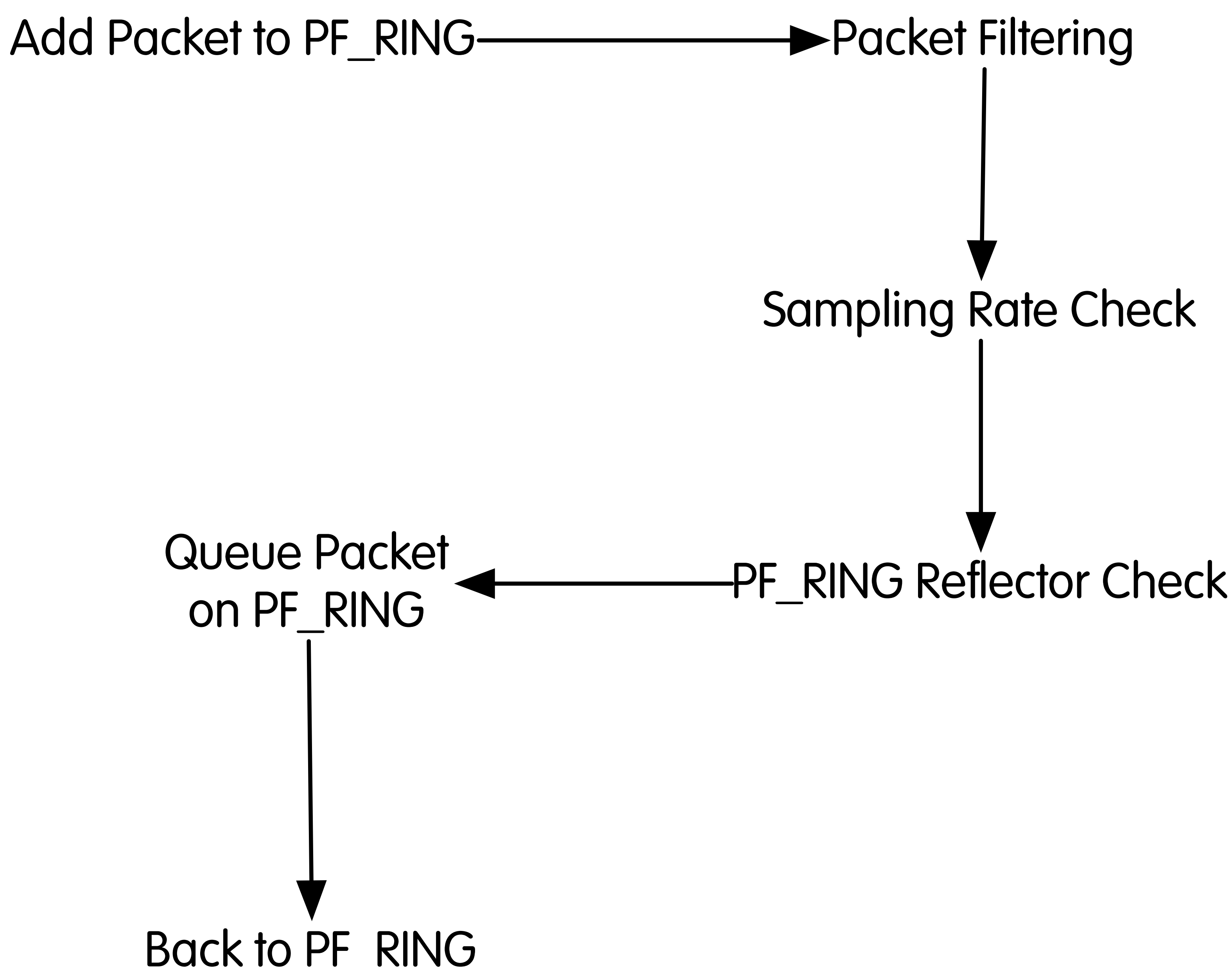
Added the packet to PF_RING sockets that potentially match it (packet and socket device match)

→ Same as above for PF_RING socket clusters

↓

Return control to the kernel

PF_RING Packet Journey [2/2]



PF_RING: Benefits

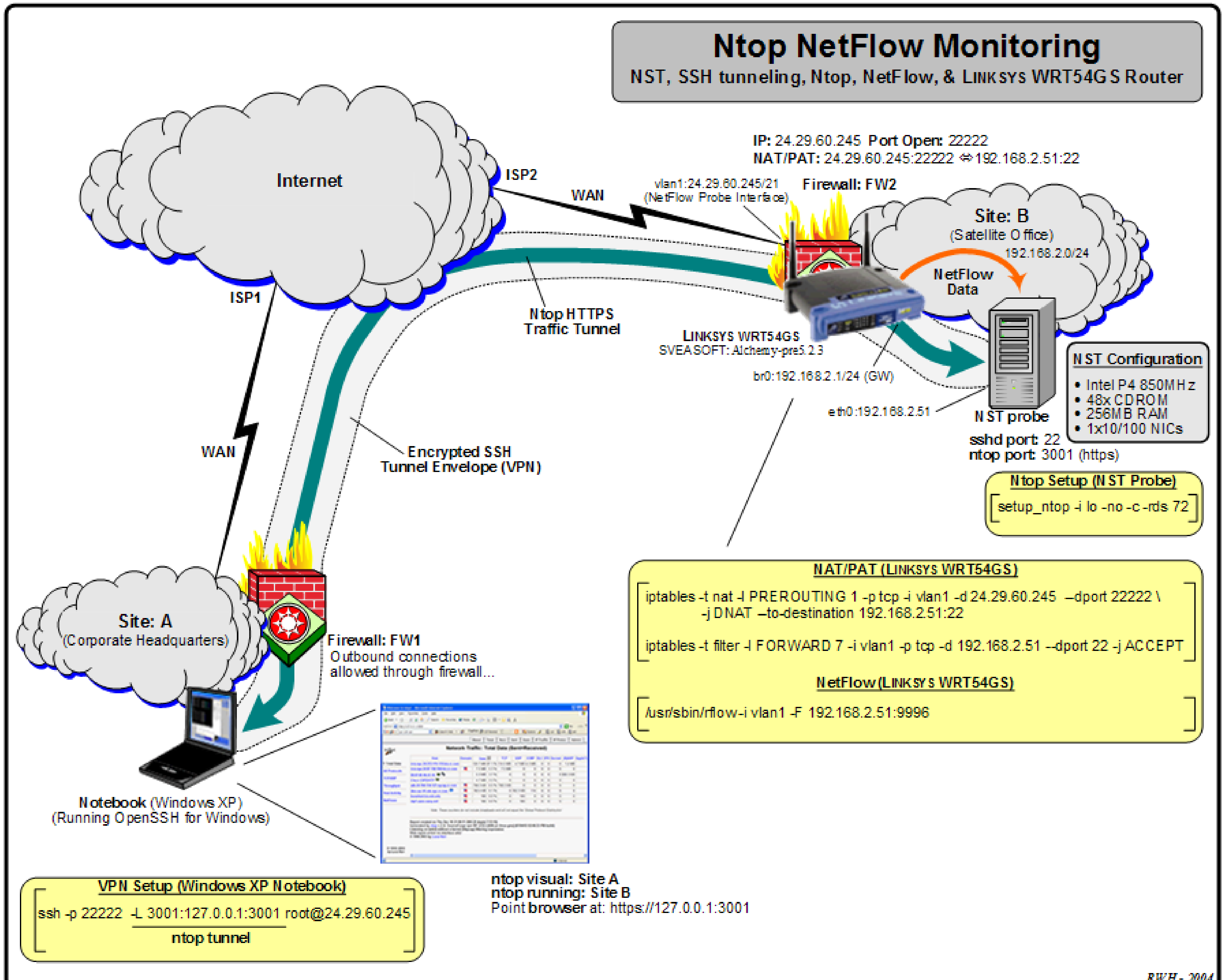
- It creates a straight path for incoming packets in order to make them first-class citizens.
- No need to use custom network cards: any card is supported.
- Transparent to applications: legacy applications need to be recompiled in order to use it.
- No kernel or low-level programming is required.
-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

PF_RING: Performance Evaluation

Pkt Size	Kpps	Mpps	% 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: pfcount, full packet size, 3.2 GHz Celeron (single-core) - IXIA 400 Traffic Generator

PF_RING on Embedded Devices



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

PF_RING Socket Clustering [1/2]

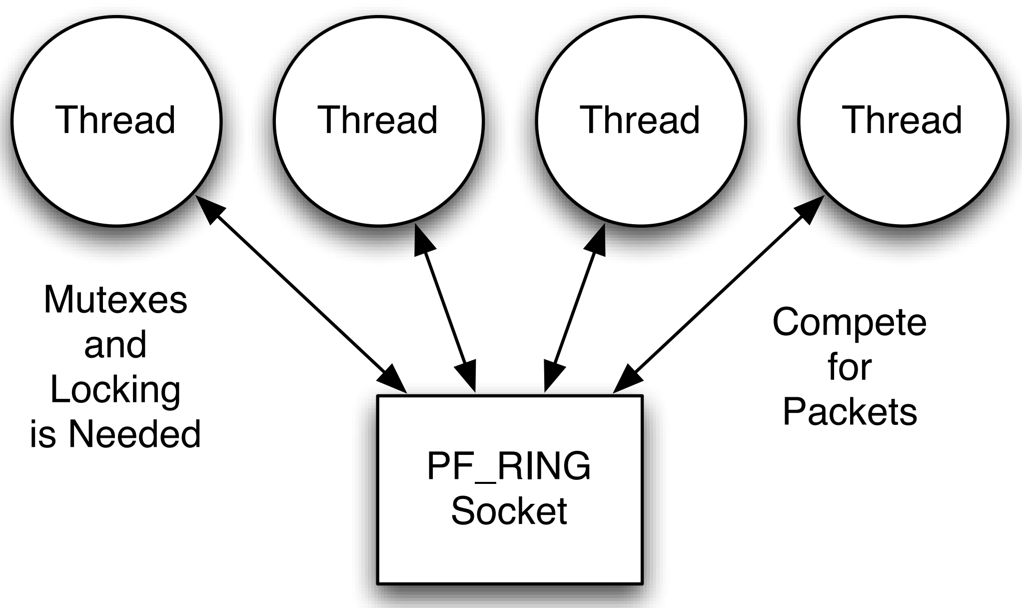
- In order to exploit modern computer architectures either multiprocessing or threading have to be used.
- Often computer programs are monolithic and hard to split into several concurrent and collaborating elements.
- In other cases (proprietary applications) source code is not available hence the application cannot be modified and split.
- There are hardware products (e.g. see cPacket's cTap) that split/ balance network traffic across network hosts.
- What is lacking at the operating system level is the concept of distributing sockets across applications. This is because network sockets are proprietary to an application/address-space.

PF_RING Socket Clustering [2/2]

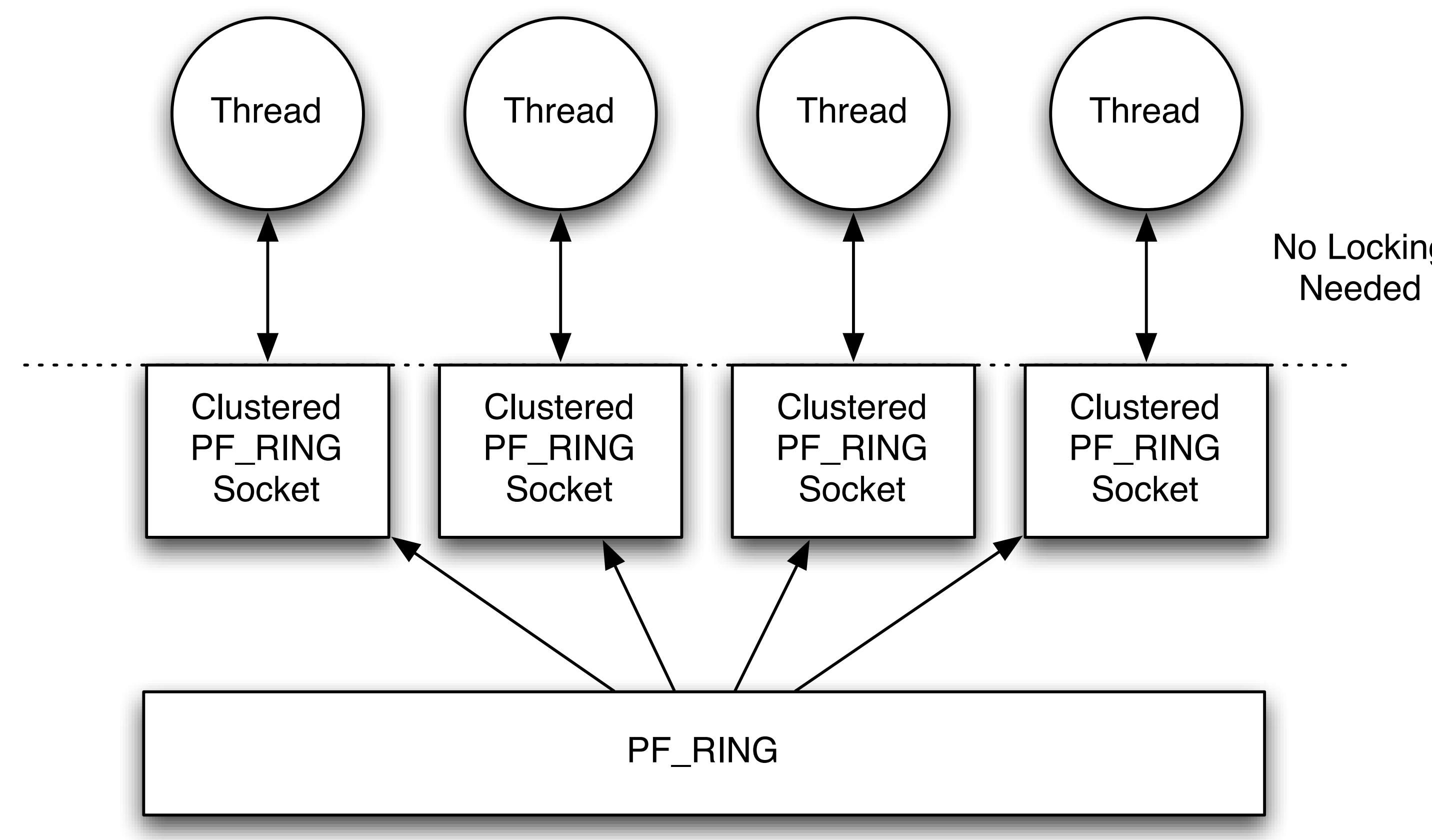
- Socket clustering is the ability to federate PF_RING sockets similar, but opposite, to network interface bonding.

- The idea is simple:
  - Run several monitoring applications, each analyzing a portion of the overall traffic.
    and/or
  - Create multithreaded applications that instead of competing for packets coming from the same socket, have private per-thread sockets.

PF_RING Clustering: Threads

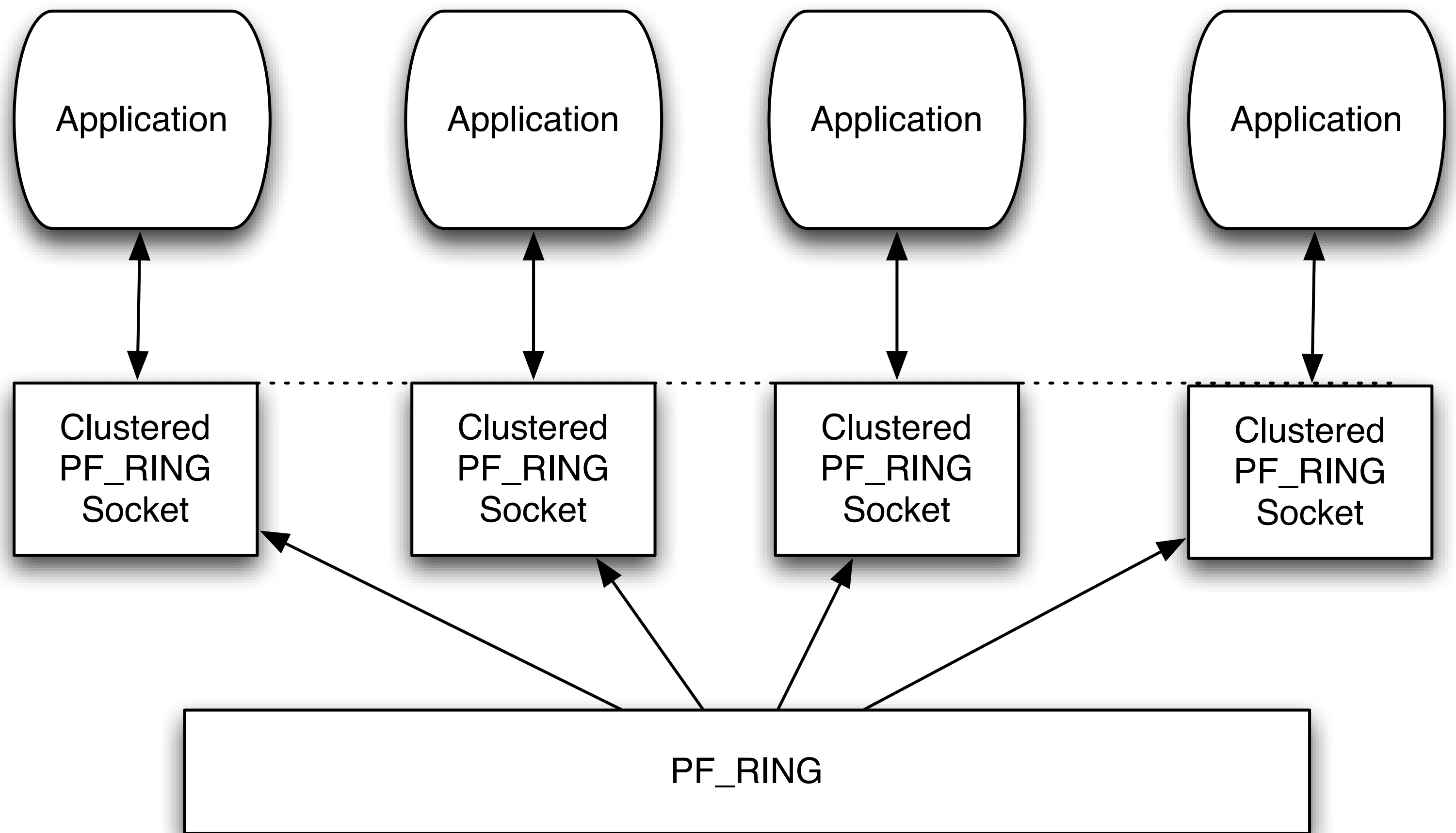


Vanilla PF_RING Application



PF_RING Socket Cluster

PF_RING Clustering: Applications



- Same as clustering with threads, but across address spaces.
- PF_RING allows clustering to be enabled seamlessly both at thread and application level.

PF_RING Clustering: Code Example

```
if((pd = pfring_open(device, promisc, snaplen, 0)) == NULL) {
    printf("pfring_open error\n");
    return(-1);
  } else {
    u_int32_t version;

    pfring_version(pd, &version);
    printf("Using PF_RING v.%d.%d.%d\n",
           (version & 0xFFFF0000) >> 16,  (version & 0x0000FF00) >>
8,
           version & 0x000000FF);
  }

  if(clusterId > 0)  {
      int rc = pfring_set_cluster(pd, clusterId);
      printf("pfring_set_cluster returned %d\n", rc);
  }
```

PF_RING Clustering: Summary

- Network traffic balancing policy across socket clusters
  - Per-flow (default)
  - Round-Robin

- Advantages:
  - No locking required when threads are used
  - Ability to distribute the load across multiple applications
  - Very fast as clustering happens into the kernel.

- Socket clustering has been the first attempt to make PF_RING more multi-processing/core friendly.

PF_RING: Packet Filtering [1/2]

- PF_RING has addressed the problem of accelerating packet capture.
- Packet filtering instead is still based on the "legacy" BPF code.
- 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
(002) ldb      [23]
(003) jeq      #0x11jt 4
(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 (i.e. there is a lot of incoming traffic but very few packets match the filters).
- Capture starts from filtering signaling protocols and then intercepting voice payload.
- BPF-like filtering is not effective (one filter only). When multiple filters need to be enforced, each one has to be executed individually.
- It is necessary to add/remove filters on the fly with hundred active filters.

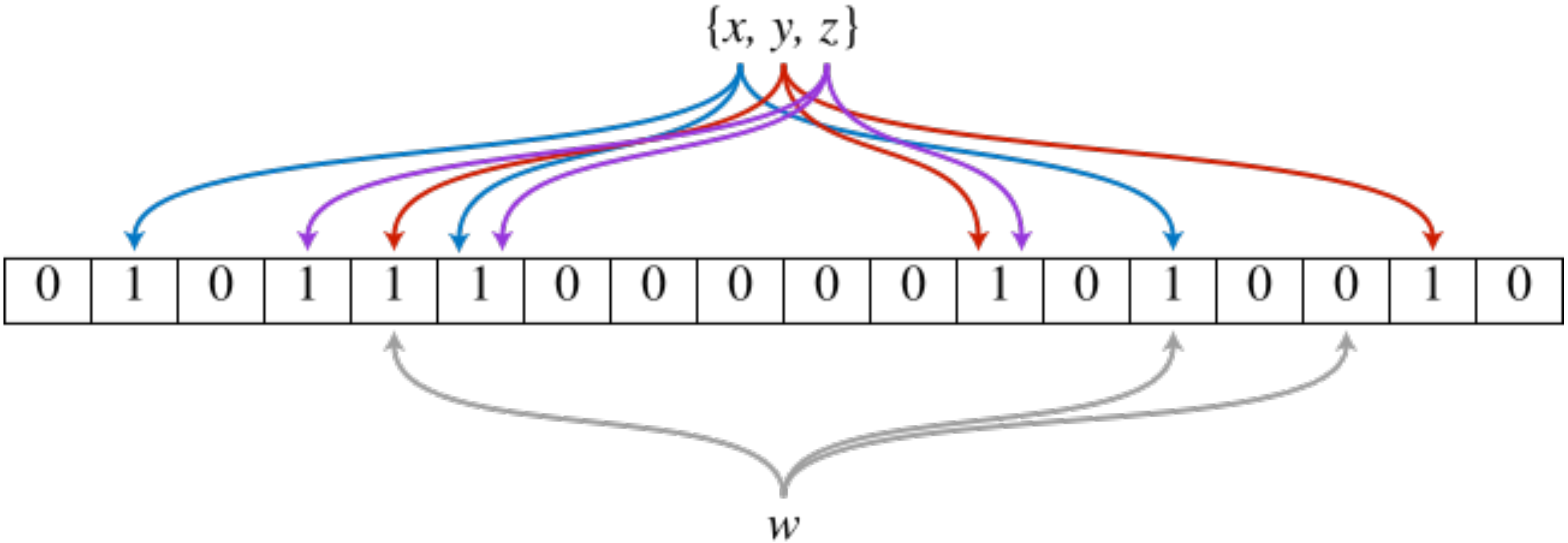
Beyond BPF Filtering [2/2]

Solution

- Filter packets directly on device drivers (initial release) and PF_RING (second release).
- Implement hash/bloom based filtering (limited false positives) but not BPF at all.
- 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).
- No much difference between PF_RING and driver filtering hence the code has been moved to PF_RING.

Dynamic Bloom Filtering [1/2]

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

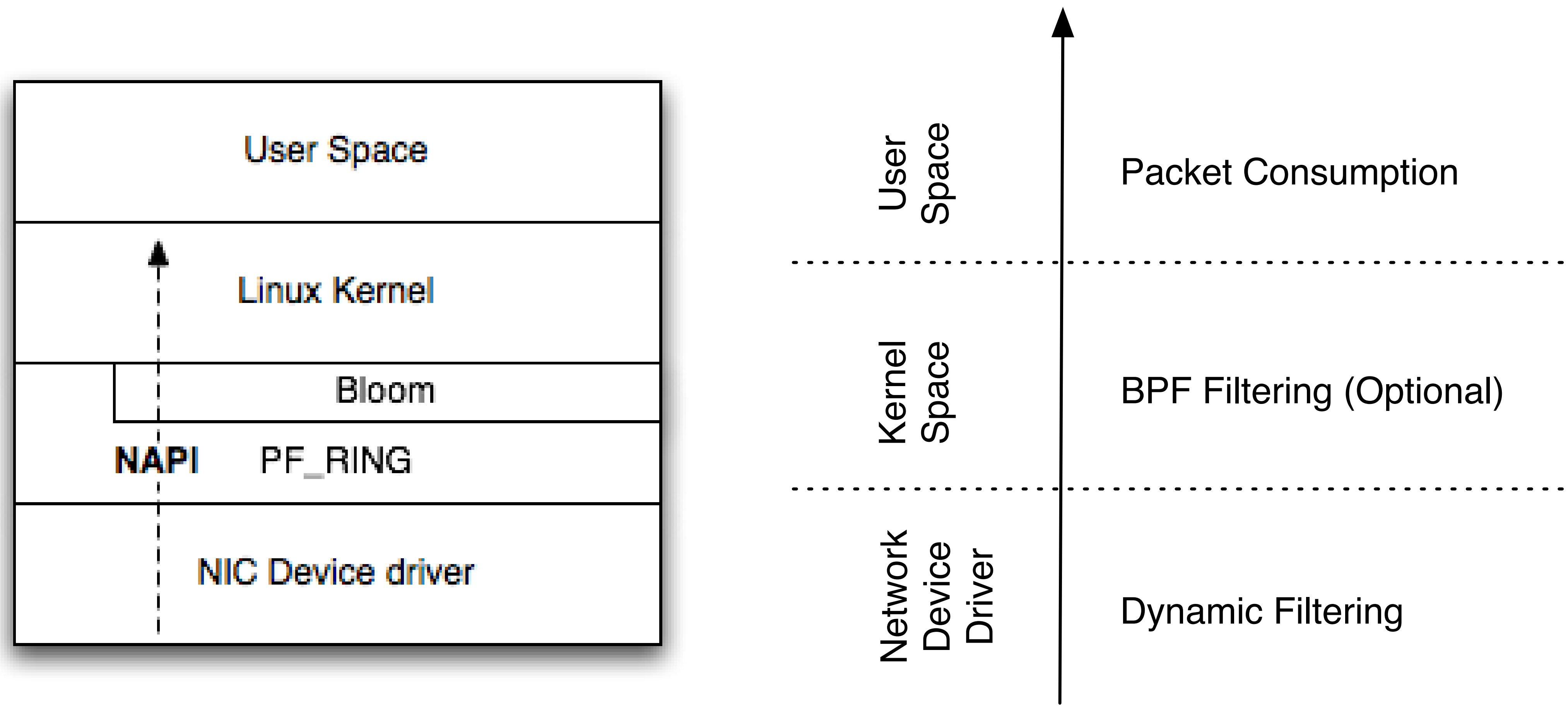


Check for inclusion

Bloom Filters [2/2]

- Ability to specify a thousand 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.
- 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 (in 3.x series 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).

Bloom Filters Limitations [1/2]

- Bloom filtering has shown to be a very interesting technology for "precise" packet filtering.
- Unfortunately many application 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

Bloom Filters Limitations [2/2]

- Possible workarounds
  - Support ranges by calculating the hash on various combinations
    - 5-tuple for perfect matching (proto, ip/port src, ip/port dst)
    - multiple bloom dictionaries for /32, /24, /16, and /8 networks for network match
- Note that as bloom matching is not exact, using a bloom dictionary for storing negative values (i.e. for implementing the not) is not a good idea. This is because not(false positive) means that a packet might be discarded as the filter is not match although this packet passed the filter.
- In a nutshell:
  - Bloom filters are a fantastic technology for exact packet matching
  - PF_RING must also offer support for 'partial' filtering.

Extended PF_RING Filters [1/2]

The author has 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).
- Precise filters (e.g. best match) have priority over (e.g. generic) wilcard-ed filters.
- Support of filter ranges (IP and port ranges) for reducing the number of filters.
- Support of mono or bi-directional filters, yet for reducing number of filters.
- Ability to filter both on standard 5-tuple fields and on L7 fields (e.g. HTTP method=GET).

Extended PF_RING Filters [2/2]

- Parsing information (including L7 information) need to be returned to user-space (i.e. do not parse the packet twice) and to all PF_RING components that for various reasons (e.g. due to socket clustering) need to have accessed to this information.

- 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.
  - Execute action and continue rule evaluation (via PF_RING plugins).

- 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).

PF_RING Packet Parsing [1/4]

- Contrary to BPF that basically does parse packets while filtering them, PF_RING filtering requires packet to be parsed first.
- Parsing information is propagated up to the userland.
- The basic PF_RING engine contains parsing up to TCP/UDP.

PF_RING Packet Parsing [2/4]

```
struct pkt_parsing_info {
  /* Core fields (also used by NetFlow) */
  u_int8_t dmac[ETH_ALEN], smac[ETH_ALEN];  /* MAC src/dst addresses */
  u_int16_t eth_type;   /* Ethernet type */
  u_int16_t vlan_id;    /* VLAN Id or NO_VLAN */
  u_int8_t  ip_version;
  u_int8_t  l3_proto, ip_tos; /* Layer 3 protocol/TOS */
  ip_addr   ip_src, ip_dst;   /* IPv4 src/dst IP addresses */
  u_int16_t l4_src_port, l4_dst_port; /* Layer 4 src/dst ports */
  struct {
    u_int8_t flags;   /* TCP flags (0 if not available) */
    u_int32_t seq_num, ack_num; /* TCP sequence number */
  } tcp;
  u_int16_t last_matched_plugin_id; /* If > 0 identifies a plugin to that matched the packet */
  u_int16_t last_matched_rule_id; /* If > 0 identifies a rule that matched the packet */
  struct pkt_offset offset; /* Offsets of L3/L4/payload elements */

  /* Leave it at the end of the structure */
  packet_user_detail pkt_detail;
};
```

PF_RING Packet Parsing [3/4]

- The decision to always parse the packet is motivated as follows:
  - Packet parsing is very cheap (in terms of computation) and its slow-down is negligible.
  - Beside rare exceptions (e.g. for packet-to-disk applications), user space applications will need to parse packets.
- PF_RING does not natively include layer-7 packet filtering as this is delegated by plugins as shown later in this presentation.

PF_RING Packet Parsing [4/4]

Extended Parsing

```
struct pfring_pkthdr {
  struct timeval ts;     /* time stamp */
  u_int32_t caplen;      /* length of portion present */
  u_int32_t len;         /* length this packet (off wire) */
  struct pkt_parsing_info parsed_pkt; /* packet parsing info */
  u_int16_t parsed_header_len; /* Extra parsing data before packet */
};
```

PF_RING: Exact Filters [1/2]

- Exact filters (called hash filtering rules) are used whenever all the filtering criteria are present in the filter.

Filter Elements

```
typedef struct {
  u_int16_t vlan_id;
  u_int8_t  proto;
  u_int32_t host_peer_a, host_peer_b;
  u_int16_t port_peer_a, port_peer_b;

  [...]
} hash_filtering_rule;
```

- Exact filters are stored in a hash table whose key is calculated on the filter values.
- When a packet is received, the key is calculated and searched into the filter hash.

Parse Filter Expression

↓

Calculate Filter Hash Key

↓

Insert the Key into the Filter Hash

PF_RING: Exact Filters [2/2]

- Filters can have a rule associated to it such as:
  - Pass packet to userland in case of match.
  - Drop packet in case of match.
  - Execute the action associated with the packet.
- Actions are implemented into plugins. Typical action include:
  - Add/delete filtering rule
  - Increment specific traffic counters.
  - Interact with the Linux kernel for performing specific actions.

```
typedef struct {
  [...]

  rule_action_behaviour rule_action; /* What to do in case of match */
  filtering_rule_plugin_action plugin_action;
  unsigned long jiffies_last_match;
} hash_filtering_rule;
```

Filter Actions

PF_RING: Wildcard-ed Filters [1/2]

- This filter family has to be used whenever:
  - Not all filter elements are set to a specific value.
  - The filter contains value ranges.
- Filters are bi-directional (i.e. they are checked on both source and destinations fields.
- Filtering rules have a unique (in the PF_RING socket) numeric identifier that also identifies the rule evaluation order.

```
typedef struct {
  u_int8_t dmac[ETH_ALEN], smac[ETH_ALEN]; /* Use '0' (zero-ed MAC address) for any MAC address.
                                              This is applied to both source and destination. */
  u_int16_t vlan_id;                    /* Use '0' for any vlan */
  u_int8_t  proto;                      /* Use 0 for 'any' protocol */
  ip_addr   host_low, host_high;        /* User '0' for any host. This is applied to both source
                                           and destination. */
  u_int16_t port_low, port_high;        /* All ports between port_low...port_high
                                           0 means 'any' port. This is applied to both source
                                           and destination. This means that
                                           (proto, sip, sport, dip, dport) matches the rule if
                                           one in "sip & sport", "sip & dport" "dip & sport"
                                           match. */
} filtering_rule_core_fields;
```

PF_RING: Wildcard-ed Filters [2/2]

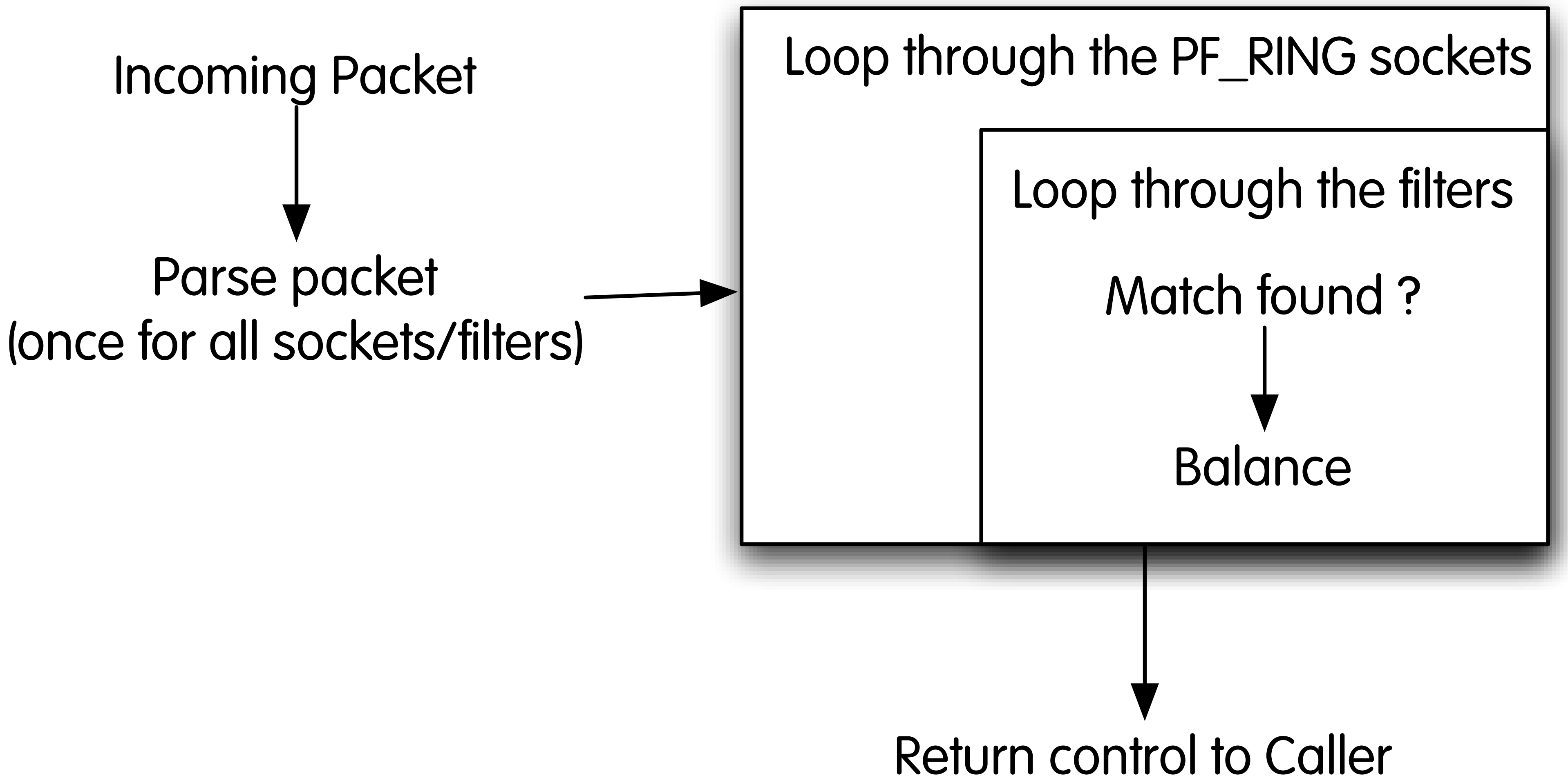
- Filters can optionally contain some extended fields used for:
  - Matching packet payload
  - Implementing more complex packet filtering by means of plugins (see later).
- User-space PF_RING library allows plugins to specify some parameters to be passed to filters (e.g. pass only HTTP packets with method POST).

```
typedef struct {
  char payload_pattern[32];         /* If strlen(payload_pattern) > 0, the packet payload
   must match the specified pattern */
  u_int16_t filter_plugin_id;       /* If > 0 identifies a plugin to which the data structure
   below will be passed for matching */
  char      filter_plugin_data[FILTER_PLUGIN_DATA_LEN];
/* Opaque data structure that is interpreted by the
   specified plugin and that specifies a filtering
   criteria to be checked for match. Usually this data
   is re-casted to a more meaningful data structure
*/
} filtering_rule_extended_fields;
```

Combining Filtering with Balancing [1/4]

- PF_RING clustering allows socket to be grouped so that they be used for effectively sharing load across threads and processes.
- Clustering works at PF_RING socket level and it's basically a mechanism for balancing traffic across packet consumers.
- PF_RING filtering rules combine the best of these technologies by implementing traffic balancing for those packets that match a certain filter.
- The idea is to have the same filter specified for various sockets that are the grouped together. Packets matching the filter are then forwarded only to one of the sockets.

Combining Filtering with Balancing [2/4]



Combining Filtering with Balancing [3/4]

- Filtered packets are balanced across sockets as follows

```
typedef struct {
  [...]
  u_int8_t balance_id, balance_pool; /* If balance_pool > 0, then pass the
    packet to PF_RING caller only if
    (hash(proto, sip, sport, dip, dport) %
    balance_pool) = balance_id */
  [...]
} filtering_rule;
```

Filter match found

Compute balance Value

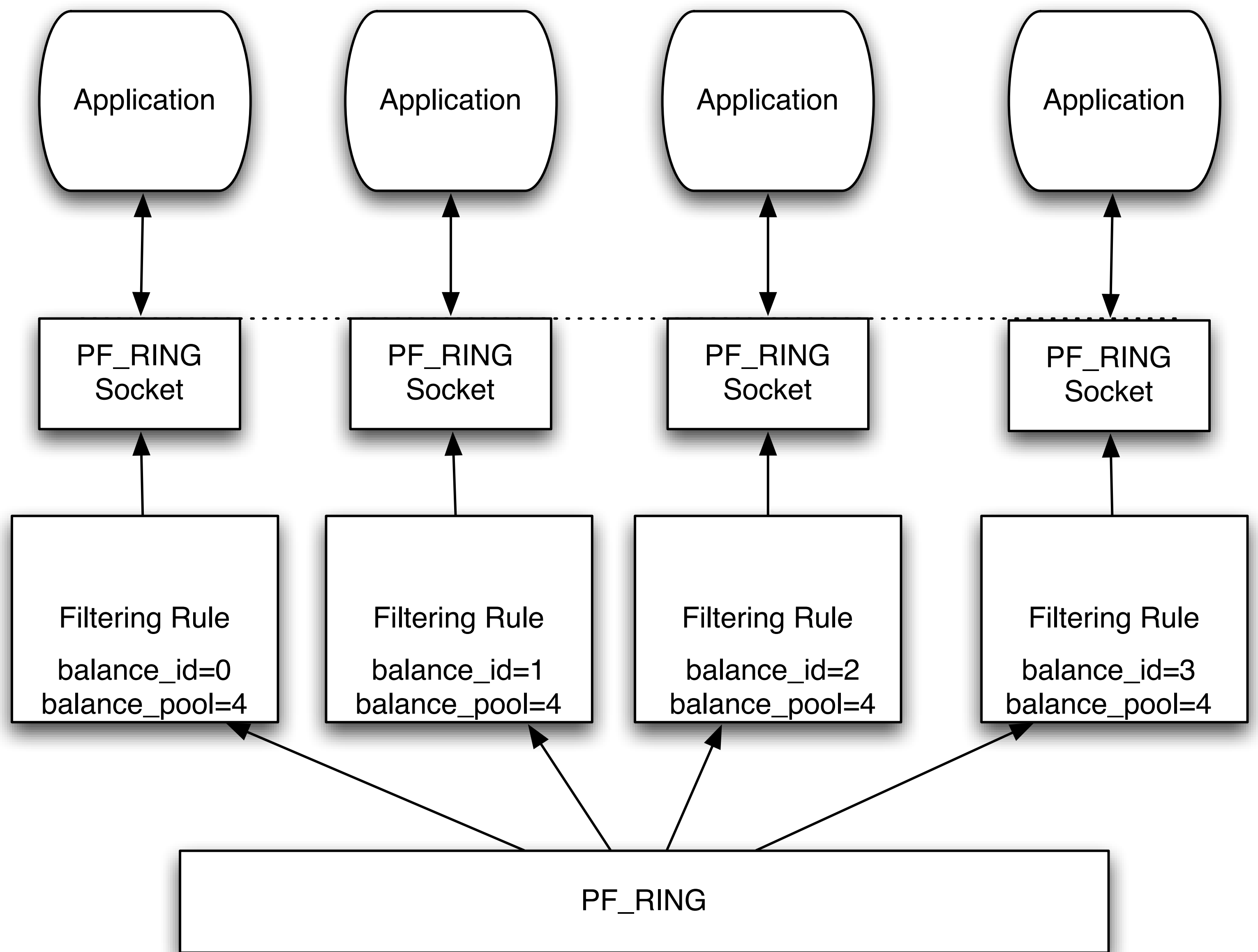
`hash(proto, sip, sport, dip, dport) % balance_pool`

Is balance Value == balance_id ?
(i.e. per-flow balancing)

Pass the Packet

Drop the Packet

Combining Filtering with Balancing [4/4]



- Using balancing for distributing load across applications/ threads is very effective for exploiting multi-processor/core architectures.

PF_RING Packet Reflection [1/3]

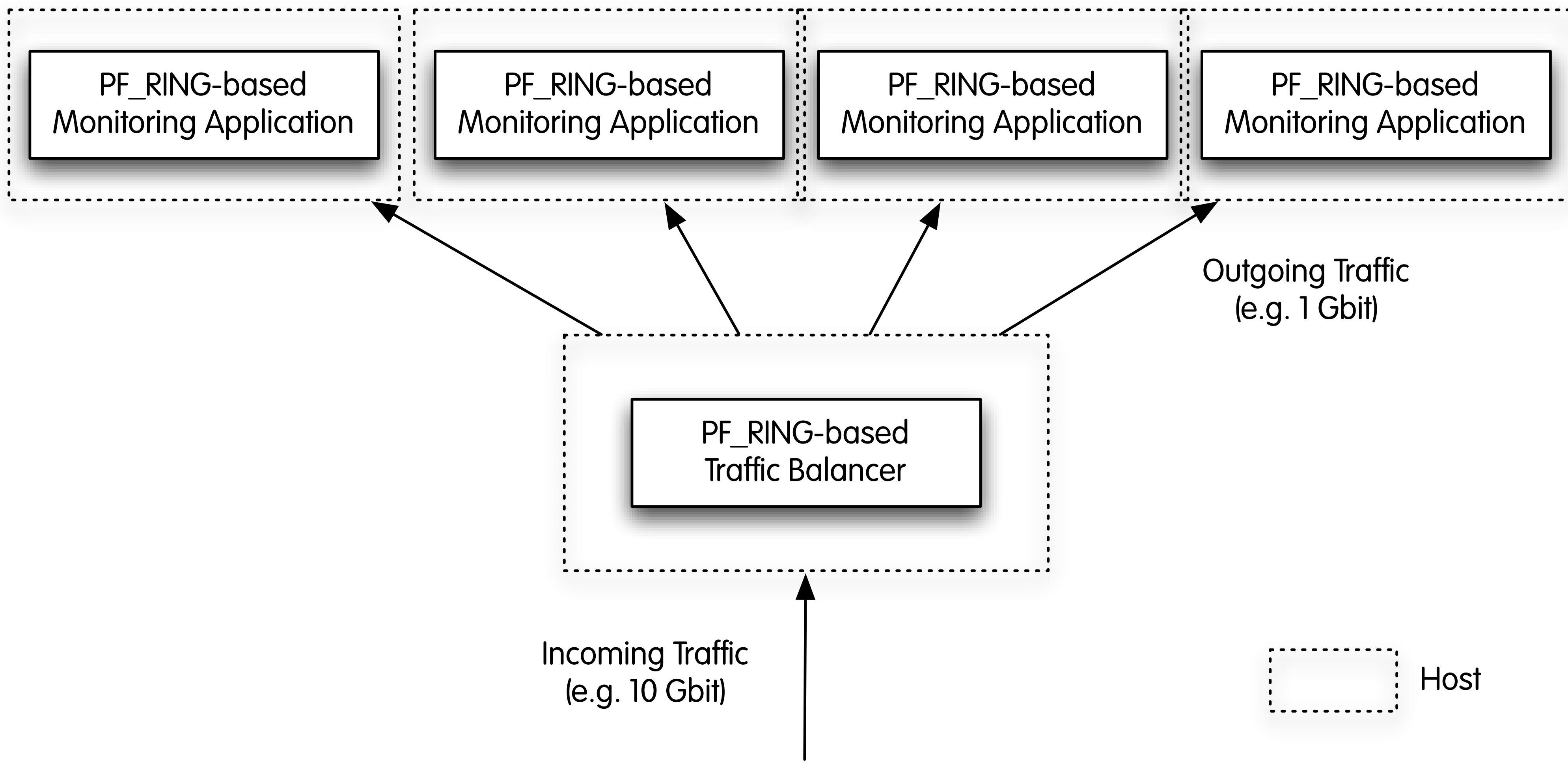
- Often, monitoring applications need to forward filtered packets to remote systems or applications.
- Traffic balancers for instance are basically a "filter & forward" application.
- Moving packets from the kernel to userland and then back to the kernel (for packet forwarding) is not very efficient as:
  - Too many actors are involved.
  - The packet journey is definitively too long.
- PF_RING packet reflection is a way to forward packets that matched a certain filter towards a remote destination on a specific NIC (that can be different from the one on which the packet has been received).
- Packet reflection is configured from userland at startup.
- All forwarding is performed inside the kernel without any application intervention at all.

PF_RING Packet Reflection [2/3]

```
/* open devices */
 if((pd = pfring_open(in_dev, promisc, 1500, 0)) == NULL)
 {
   printf("pfring_open error for %s\n", in_dev);
   return -1;
 }  else
   pfring_set_application_name(pd, "forwarder");

 if ((td = pfring_open(out_dev, promisc, 1500, 0)) == NULL) {
   printf("pfring_open error for %s\n", out_dev);
   return -1;
 } else
   pfring_set_application_name(td, "forwarder");

 /* set reflector */
 if (pfring_set_reflector(pd, out_dev) != 0)
 {
   printf("pfring_set_reflector error for %s\n", out_dev);
   return -1;
 }

 /* Enable rings */
 pfring_enable_ring(pd);
 pfring_enable_ring(td);

 while(1) sleep(60); /* Loop forever */
```

PF_RING Packet Reflection [3/3]

- PF_RING packet reflection allows easily and efficiently to implement:
  - Filtering packet balancers
  - (Filtering) Network bridges
- In a nutshell this technique allows to easily implement the "divide and conquer" principle and to combine it with techniques just presented.



PF_RING Kernel Plugins [1/3]

- Implementing into the kernel is usually more efficient than doing the same in userland because:
  - Packets do not need to travel from kernel to userland.
  - If a packet is supposed to be received by multiple applications it is not duplicated on the various sockets, but processed once into the kernel
- For packet filtering, it is important to filter as low as possible in the networking stack, as this prevents packet not matching the filter to be propagated and the discarded later on.
- PF_RING plugins allow developers to code small software modules that are executed by PF_RING when incoming packets are received.
- Plugins can be loaded and unloaded dynamically via insmod/ rmmod commands.

PF_RING Kernel Plugins [2/3]

- Each plugin need to declare a data structure according to the format below.

```
struct pfring_plugin_registration {
  u_int16_t plugin_id;
  char name[16];          /* Unique plugin name (e.g. sip, udp) */
  char description[64];   /* Short plugin description */

  plugin_filter_skb    pfring_plugin_filter_skb; /* Filter skb: 1=match, 0=no match */
  plugin_handle_skb    pfring_plugin_handle_skb;
  plugin_get_stats     pfring_plugin_get_stats;
  plugin_free_ring_mem pfring_plugin_free_ring_mem;
  plugin_add_rule      pfring_plugin_add_rule;
  plugin_register      pfring_plugin_register;

  kernel_packet_start  pfring_packet_start;
  kernel_packet_reader pfring_packet_reader;
  kernel_packet_term   pfring_packet_term;
};
```

- The various pfring_plugin_* variables are pointers to functions that are called by PF_RING when:
  - A packet has to be filtered.
  - An incoming packet has been received and needs to be processed.
  - A userland application wants to know stats about this plugin.
  - A filtering rule will be removed and the memory allocated by the plugin needs to be released.

PF_RING Kernel Plugins [3/3]

- Plugins are associated with filtering rules and are triggered whenever a packet matches the rule.
- If the plugin has a filter function, the this function is called in order to check whether a packet passing the header filter will also pass other criteria. For instance:
  - 'tcp and port 80' is a rule filter used to select http traffic
  - The HTTP plugin can check the packet payload (via DPI) to verify that the packet is really http and it's not another protocol that hides itself on the http port.
- In order to perform complex checks, rules need to be stateful hence to allocate some memory, private to the plugin, that is used to keep the state.
- PF_RING delegates to the plugin the duty of managing this opaque memory that is released by PF_RING when a rule is deleted, by calling the plugin callback.

Efficient Layer 7 Packet Analysis

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

```
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

```
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

```
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);
```

YouTube Monitoring [1/2]

- YouTube monitoring is an extension of the HTTP plugin.
- HTTP is used by YouTube to transport videos usually encoded in H.264 or Flash Video.
- The HTTP plugin can be used for monitoring, from the network point of view, the YouTube traffic and detecting whether the network quality is adequate or if the user should have experienced unstable playback.
- Video streams are tracked by checking the URL (e.g. GET /get_video?video_id=...) and the server host (www.youtube.com).
- Whenever a YouTube video stream is detected, the HTTP plugin adds an exact matching rule on the hash, used to track the stream, with the YouTube plugin specified as rule action.

YouTube Monitoring [2/2]

- The YouTube plugin is able to measure some stream statistics such as throughput, jitter, bandwidth used.

```
struct youtube_http_stats {
  u_int32_t initialTimestamp, lastTimestamp, lastSample; /* Packet Timestamps [jiffies] */
  struct timeval initial_tv;
  u_int32_t tot_pkts, tot_bytes, cur_bytes;
  u_int32_t num_samples;
  u_int8_t signaling_stream; /* 1=signaling, 2=real video stream */
  char url[URL_LEN];
  char video_id[VIDEO_ID_LEN], video_playback_id[VIDEO_ID_LEN];
  u_int32_t min_thpt, avg_thpt, max_thpt;       /* bps */
  u_int32_t min_jitter, avg_jitter, max_jitter; /* jiffies */
  u_int32_t duration_ms;
  char content_type[CONTENT_TYPE_LEN];
   u_int32_t tot_jitter, num_jitter_samples;
};
```

- When a stream is over, the plugin return to userland a packet with the stream statistics.
- Note that all stream packets are not returned to userland, but just the statistics, that contributes to reduce load on the probe and improve performance.

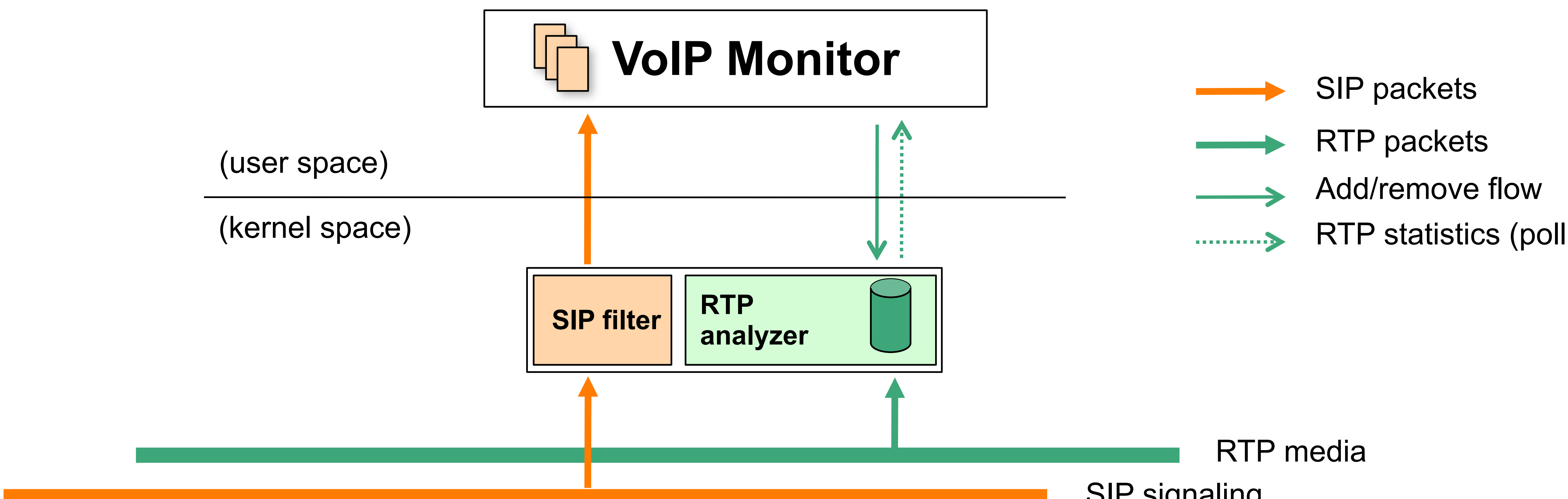
Dynamic PF_RING Filtering: VoIP [1/6]

- 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 forwarding 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 to set filters based 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.

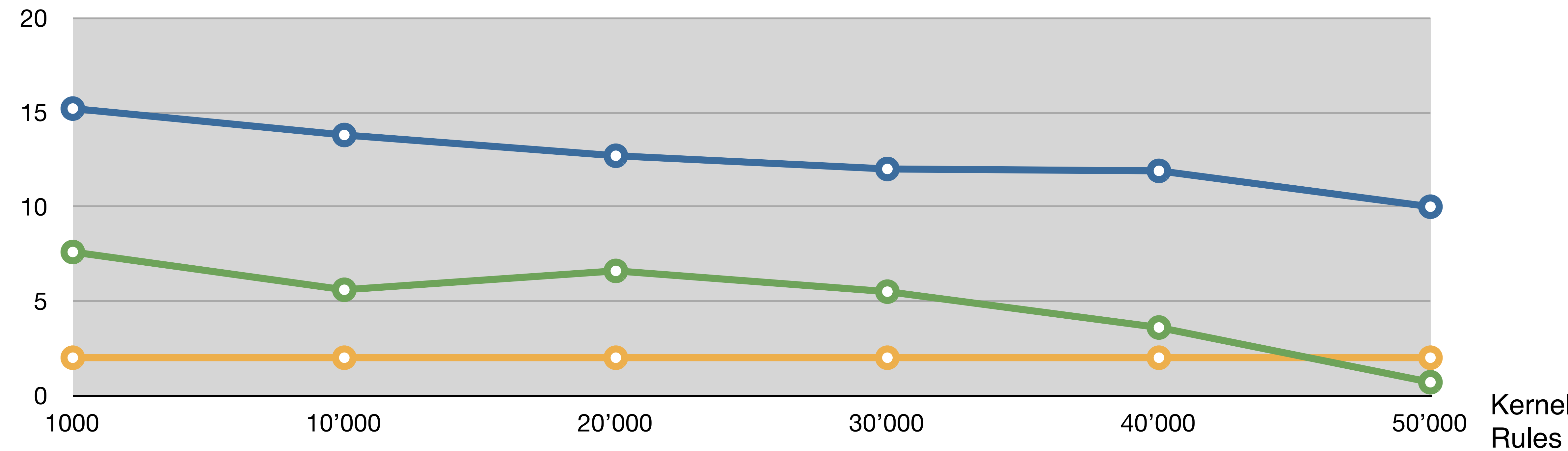
- RTP Plugin
  - It tracks RTP (mono/by-directional) flows.
  - The following, per-flow, statistics are computed: jitter, packet loss, malformed packets, out of order, transit time, max packet delta.
  - Developers can decide not to forward packets (this is the default behavior) or to forward them (usually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

Dynamic PF_RING Filtering: VoIP [4/6]

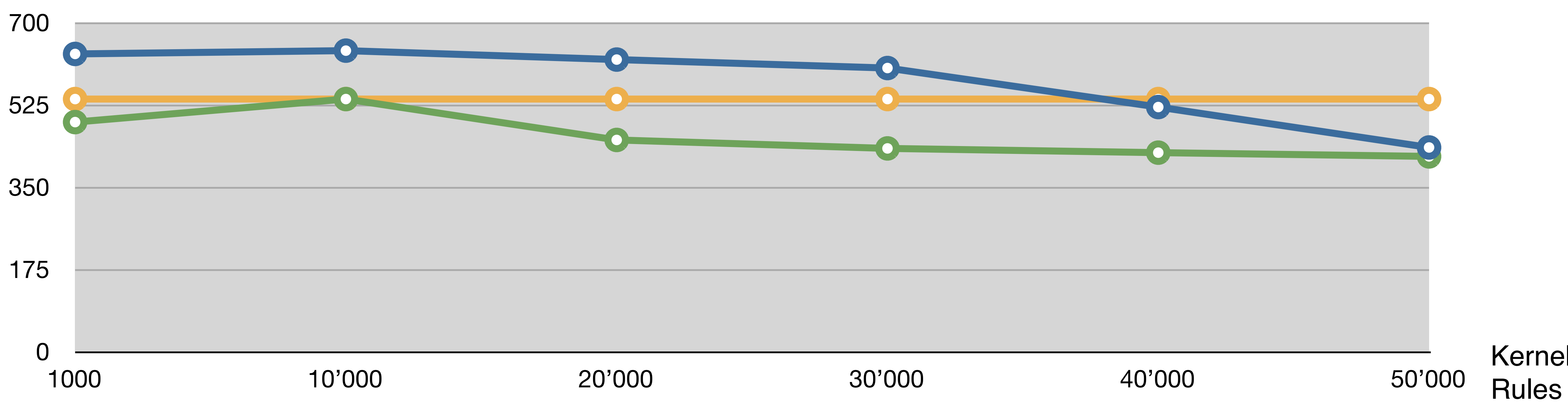
- Validation
  - A SIP test tool and traffic generator (sipp) is used to create synthetic SIP/RTP traffic.
  - A test application has been developed: it receives SIP packets (signaling) and based on them it computes RTP stats.
  - A traffic generator (IXIA 400) is used to generate noise in the line and 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]

% Idle CPU [128 bytes packets]



Max Throughput (Mbps) with no loss [128 bytes packets]



- RTP Plugin
- RTP stats computed in userland
- PF_RING capture only (no RTP analysis)

Dynamic PF_RING Filtering: VoIP [6/6]

- Validation Evaluation
  - In-kernel acceleration has demonstrated that 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.

PF_RING Content Inspection

- PF_RING allows filtering to be combined with packet 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.

L7 Analysis: Summary

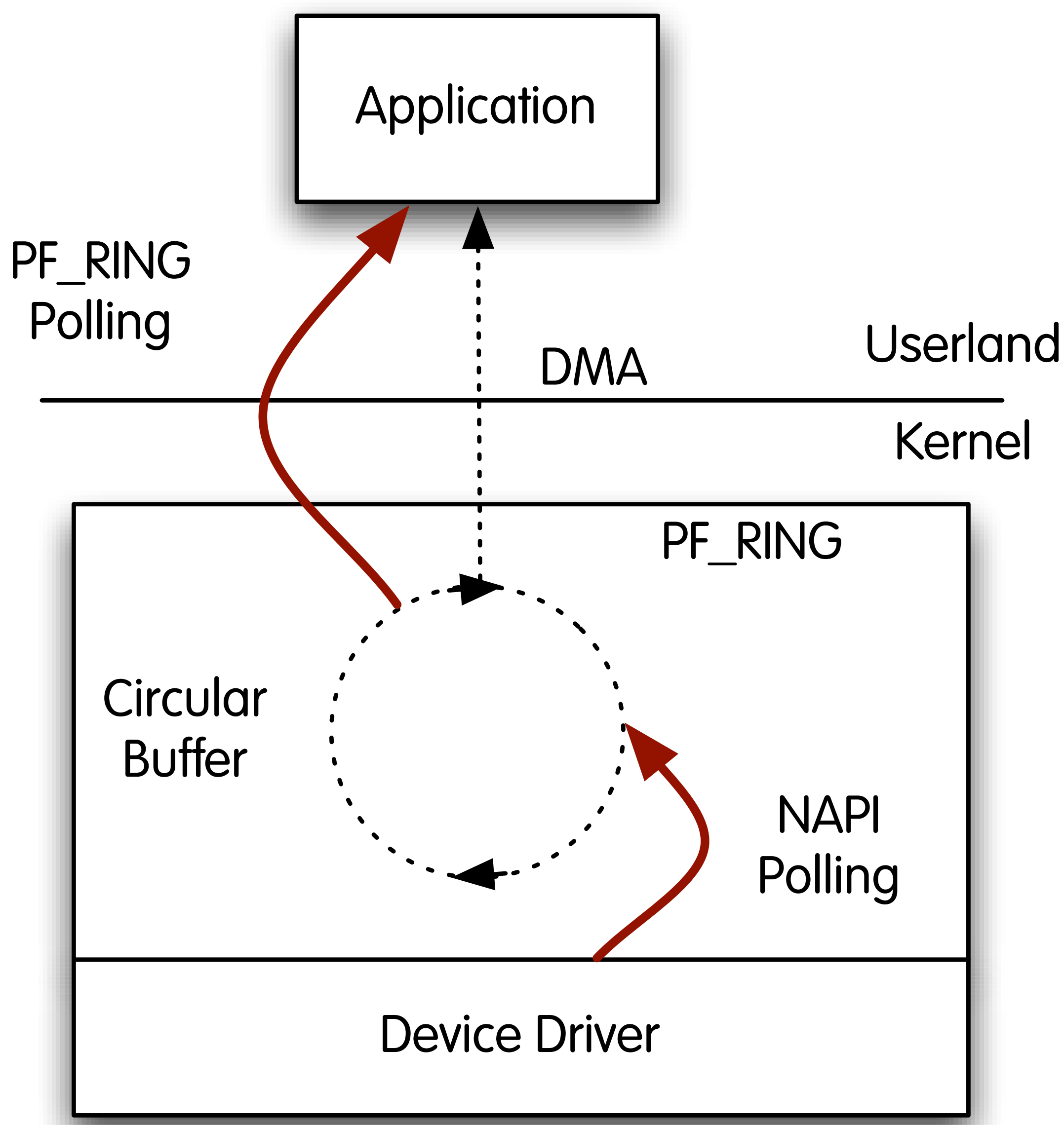
- The use of kernel plugins allows packets to have a short journey towards the application.
- In-kernel processing is very efficient and it avoids the bottleneck of several userland application threads competing for packets.
- As PF_RING requires minimal locking (when the filtering rule is accessed and updated), packets are processed concurrently without any intervention from userland applications.
- As the Linux kernel concurrently fetches packets from adapters, this is a simple way to exploit multi-processing/core without having to code specific (multithreaded) userland applications and serialize packets on (PF_RING) sockets.

Direct Access to NICs

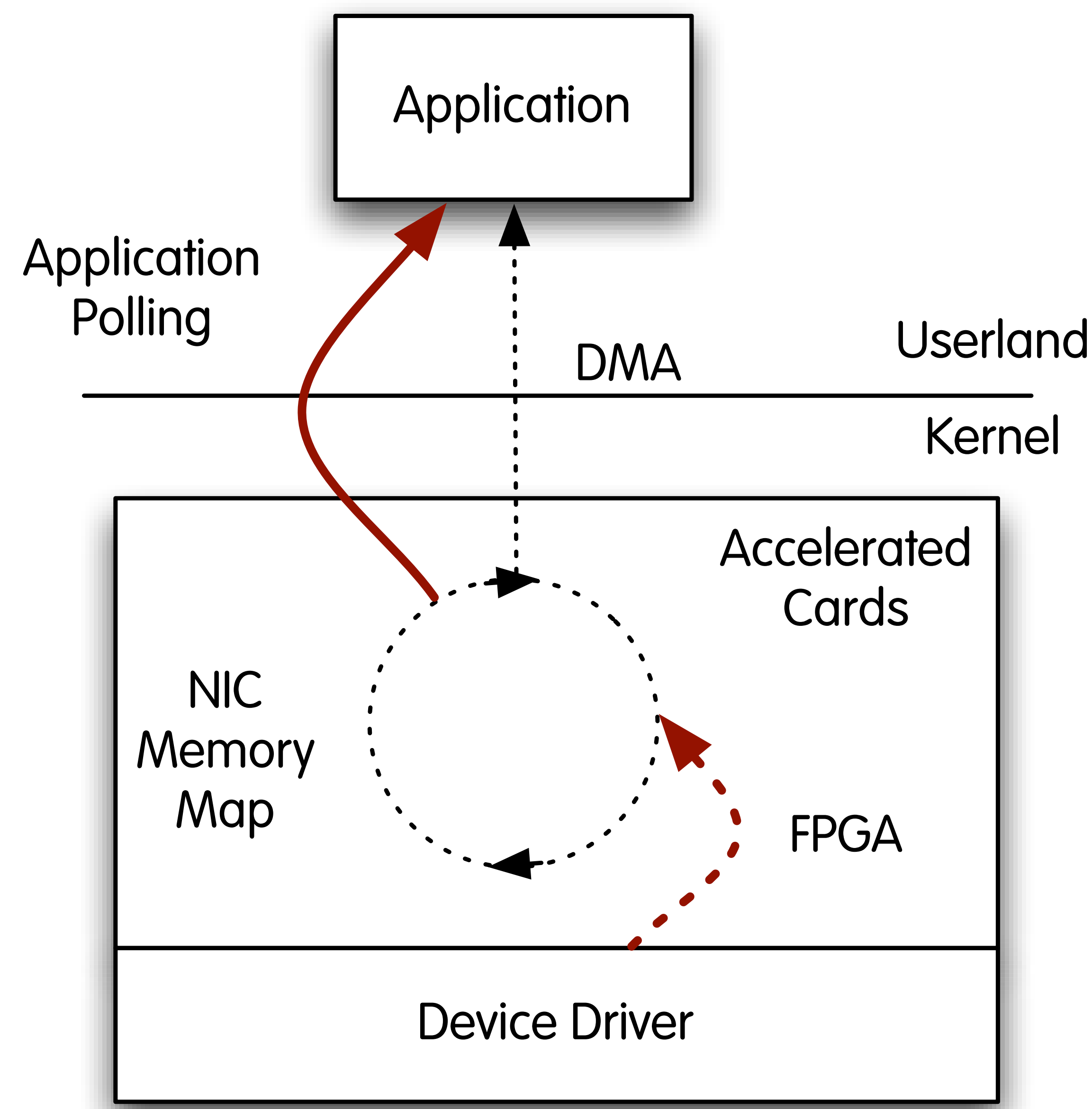
Direct NIC Access: Introduction

- Commercial accelerated NICs are accelerated either using ASIC (rare) or FPGAs (often) chips.
- Accelerators improve common activities such as packet filtering and are also responsible of pushing packets to memory with very limited (< 1%) load on the main CPU.
- Applications access packets directly without any kernel intervention at all.
- A kernel-mapped DMA memory allows the application to manipulate card registers and to read packets from this memory where incoming packets are copied by the hardware accelerators.
- Cards falling in this category include:
  - Endace DAG
  - Napatech
  - NetFPGA

Direct NIC Access: Comparison [1/2]



PF_RING

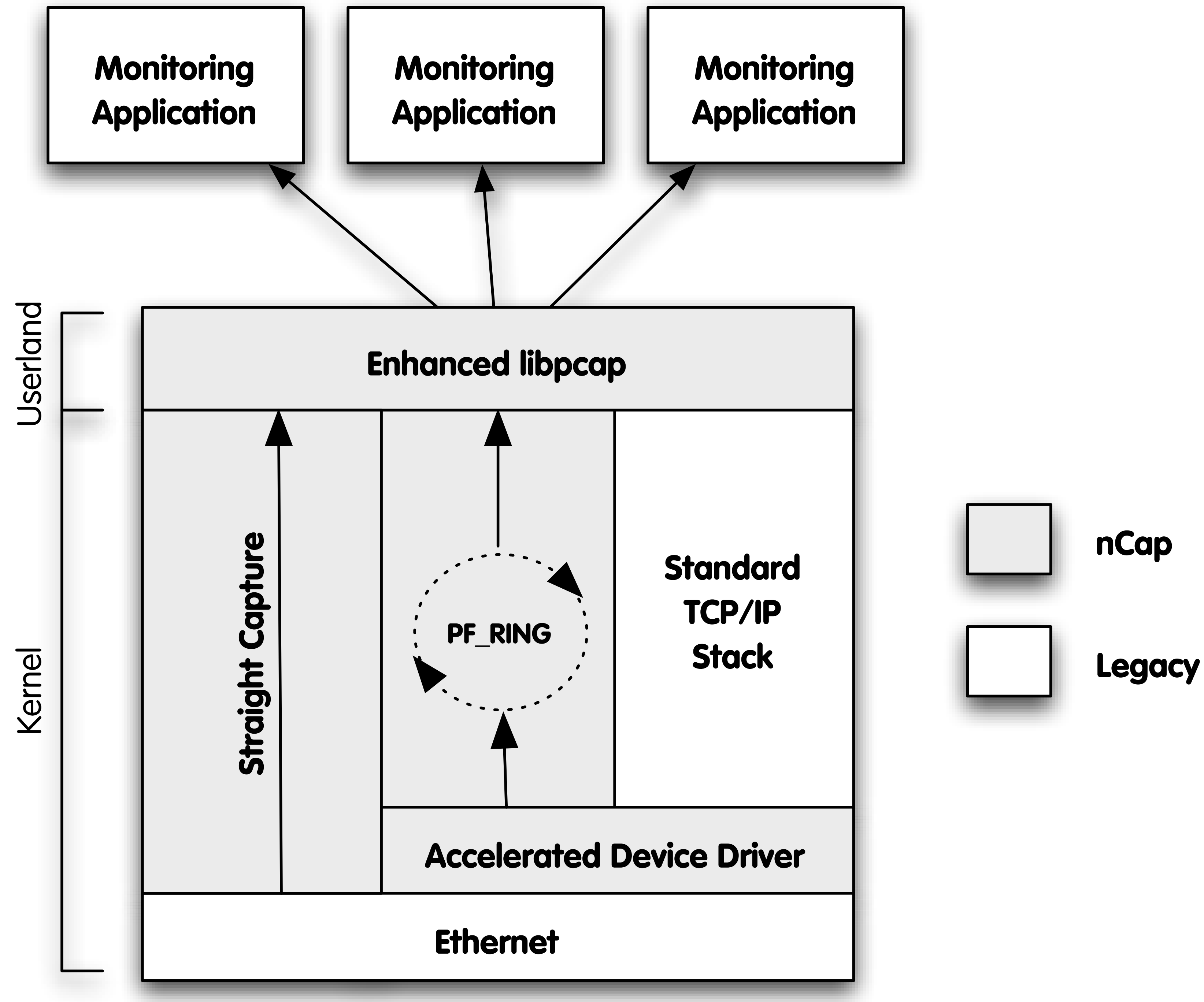


Hardware Acceleration

Direct NIC Access: Comparison [2/2]

- The reason why accelerated cards are so efficient are:
  - The FPGA polls packets as fast as possible without any intervention from the main CPU. In Linux the main CPU has to periodically read packets through NAPI from the NIC.
  - Received packets are copied on a pre-allocated large memory buffer so no per-packet allocation/deallocation is necessary at all, as it happens in vanilla Linux.
  - Similar to PF_RING, packets are read from circular buffer without any kernel interaction (beside packet polling).
- Limitations
  - As applications access packets directly, if they improperly manipulate card's memory the whole system might crash.
  - FPGA filtering is very limited and not as rich as PF_RING.
  - Contrary to PF_RING, only one application at time can read packers from the ring.

Welcome to nCap (Circa 2003)



nCap Features

	Packet Capture Acceleration	Wire Speed Packet Capture	Number of Applications per Adapter
Standard TCP/IP Stack with accelerated driver	Limited	No	Unlimited
PF_RING with accelerated driver	Great	Almost	Unlimited
Straight Capture	Extreme	Yes	One

nCap Internals

- nCap maps at userland the card registers and memory.
- The card is accessed by means of a device /dev/ncap/ethX
- If the device is closed it behaves as a “normal” NIC.
- When the device is open, it is completely controlled by userland the application.
- A packet is sent by copying it to the TX ring.
- A packet is received by reading it from the RX ring.
- Interrupts are disabled unless the userland application wait for packets (poll()).
- On NIC packet filtering (MAC Address/VLAN only).

nCap Comparison (1 Gbit)

	Maximum Packet Loss at Wire Speed	Estimated Card Price	Manufacturer
DAG	0%	> 5-7 K Euro	Endace.com
nCap	0.8%	100 Euro	
Combo 6 (Xilinx)	5%	> 7-10 K Euro	Liberouter.com

Source Cesnet (http://luca.ntop.org/ncap-evaluation.pdf)

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.

PF_RING DNA (Direct NIC Access)

- PF_RING DNA is an extension for PF_RING that allows NICs to be accessed in direct mode fully bypassing Linux NAPI.
- Based on the lessons learnt while developing nCap, DNA is a technology developed in clean-room that has been designed to be NIC-neutral in order to allows various NICs to be supported.
- The NIC mapping is driver dependent hence it requires some driver modifications in order to:
  - Disable NAPI when the NIC is accessed in DNA mode.
  - Contiguously allocate RX card's memory in one shot (and not per packet).
  - Register the NIC with PF_RING so the card is accessed seamlessly from PF_RING applications without the need to know the NIC internals and its memory layout.

PF_RING DNA (De)Registration

```
/* Register with PF_RING */
do_ring_dna_device_handler(add_device_mapping,
   adapter->tnapi.dma_mem.packet_memory,
   adapter->tnapi.dma_mem.packet_num_slots,
   adapter->tnapi.dma_mem.packet_slot_len,
   adapter->tnapi.dma_mem.tot_packet_memory,
   rx_ring->desc,
   rx_ring->count, /* # of items */
   sizeof(struct e1000_rx_desc),
   rx_ring->size, /* tot len (bytes) */
   0, /* Channel Id */
   (void*)netdev->mem_start,
   netdev->mem_end,
   netdev,
   intel_e1000,
   &adapter->tnapi.packet_waitqueue,
   &adapter->tnapi.interrupt_received,
   (void*)adapter,
   wait_packet_function_ptr);
```

NIC Memory Pointers

NIC DMA Ring

NIC Registers Memory

Packet Polling

PF_RING DNA: Current Status

- As of today, DNA is available for Intel-based 1 Gbit (e1000 driver) and 10 Gbit (ixgbe) NICs.
- Any modern dual-core (or better) system can achieve wire rate packet capture at any packet size using DNA.
- A userland library used to manipulate card registers has been integrated into PF_RING.
- Applications do not need to do anything different from standard PF_RING with the exception that the ring memory has to be open using `pfring_open_dna()` instead of the standard `pfring_open()`.
- When an application opens the adapter in DNA mode, other applications using the same adapter in non-DNA mode will stop receiving packets until the application quits.

Towards 10 Gbit Packet Capture Using Commodity Hardware

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 [1/4]

- The current trend in computer architecture is towards multi-core systems.
- Currently 4-core CPUs are relatively cheap and rather common on the market. Intel announced Xeon Nehalem-EX with 16 threads (8 cores) for late 2009. The core rush is not yet over.
- 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.

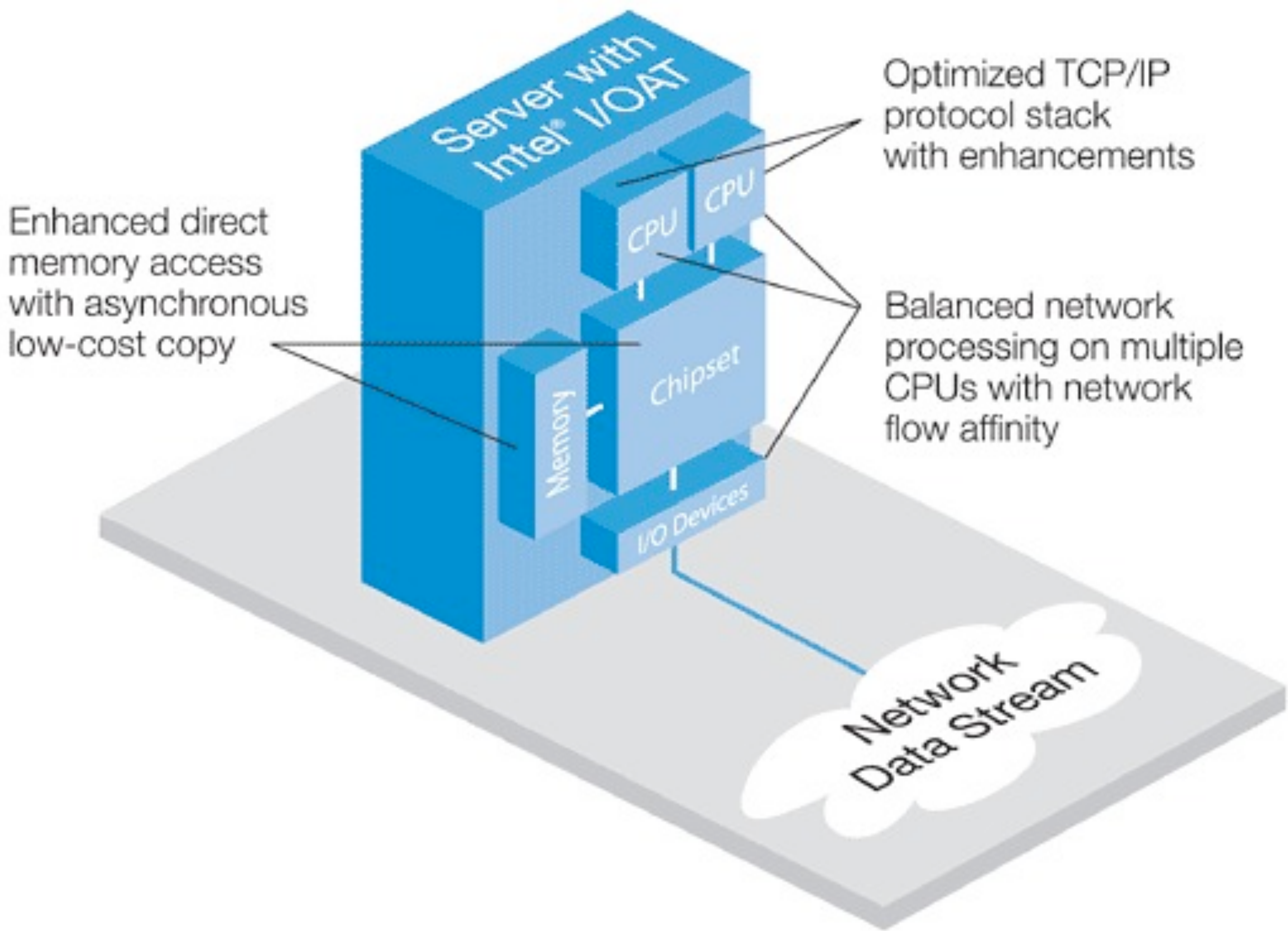
Enhanced NIC Drivers [2/4]

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

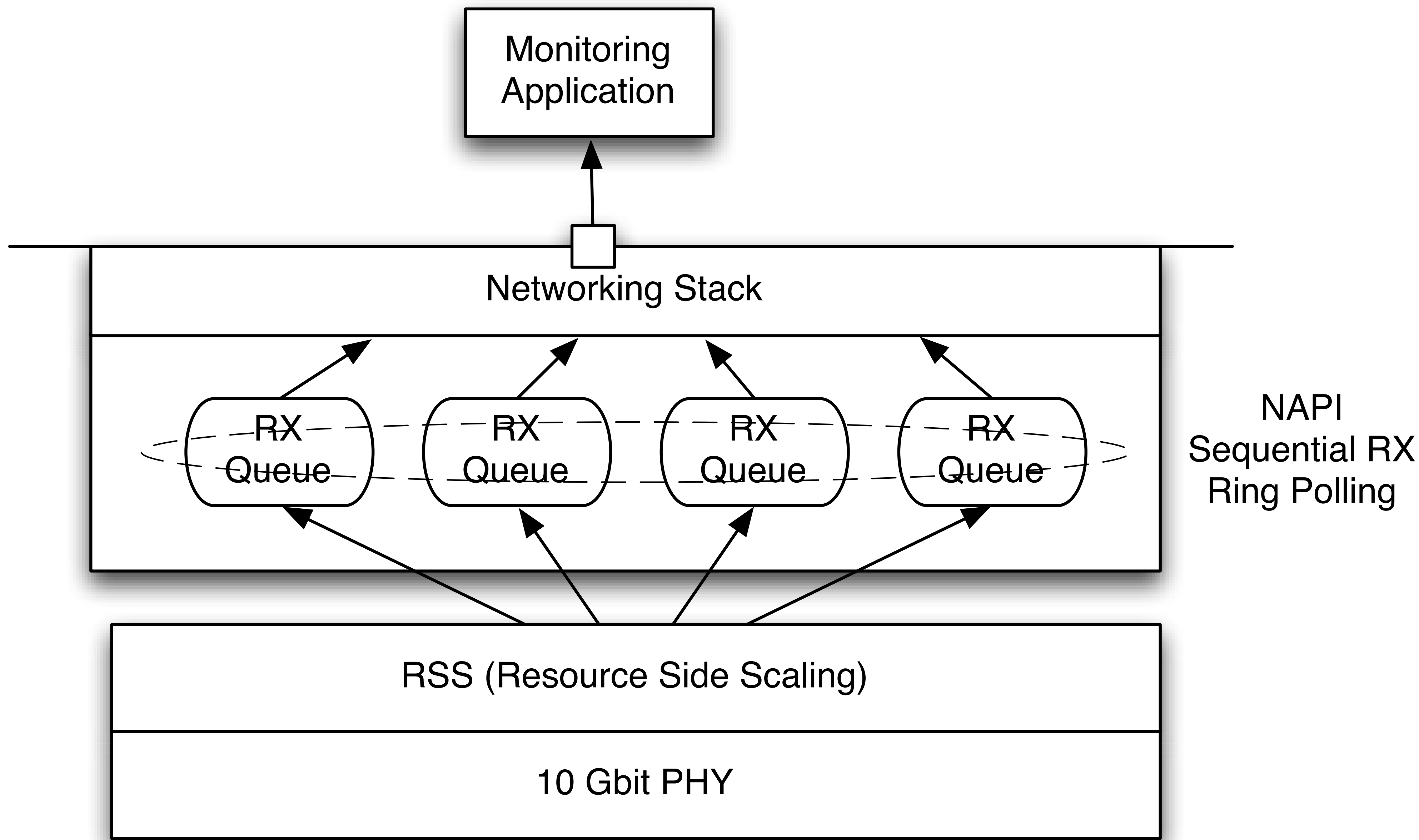
- I/O Acceleration Technology (I/OAT)
  - Direct Cache Access (DCA) asynchronously move packets from NIC directly on CPU's cache in DMA.
  - Multiple TX/RX queues (one per core) that improve system throughput and utilization.
  - MSI-X, low latency interrupts and load balancing across multiple RX queues.
  - RSS (Receive-Side Scaling) balances (network flow affinity) 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.

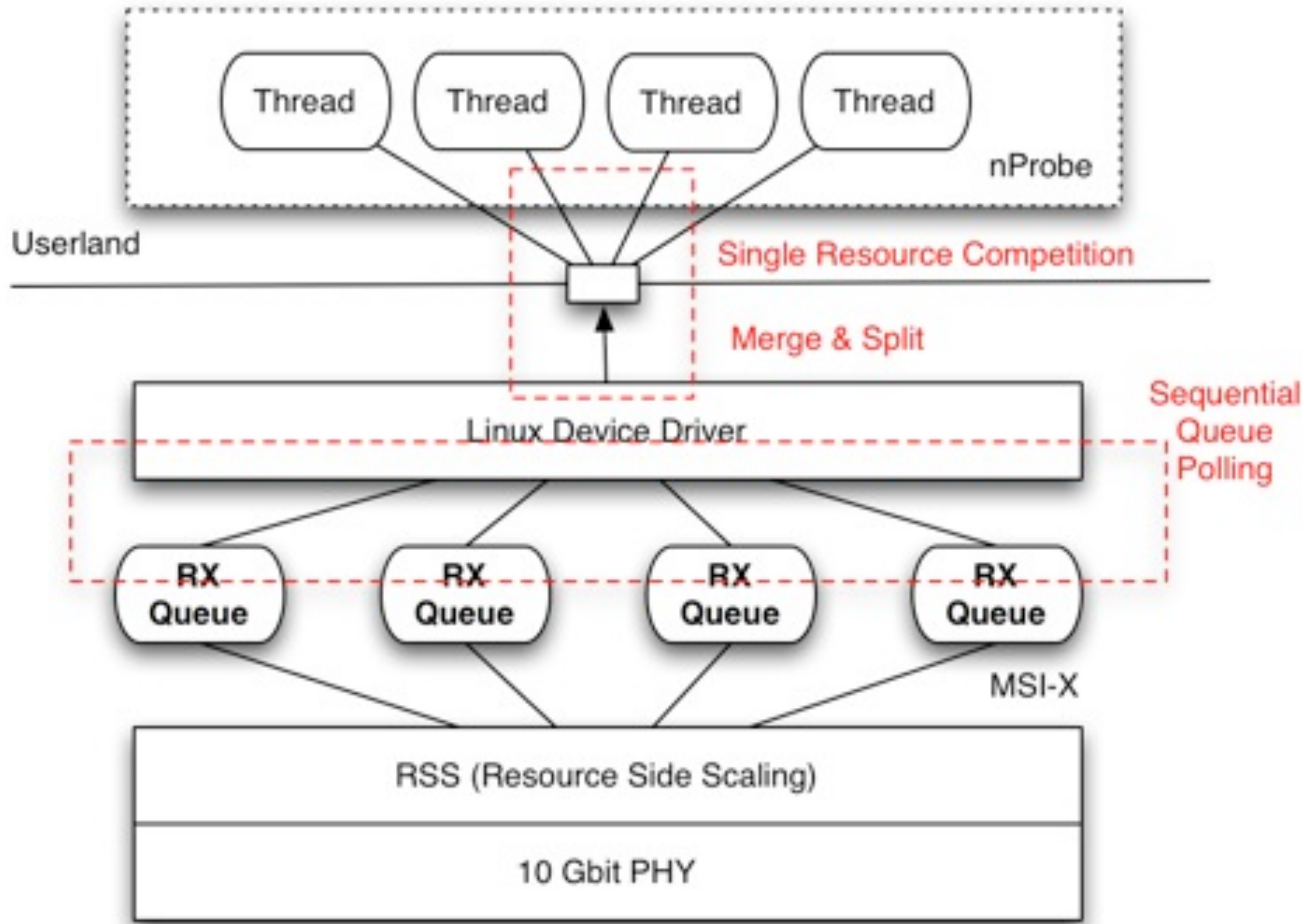
Enhanced NIC Drivers [3/4]



Enhanced NIC Drivers: Linux NAPI [4/4]



Linux NAPI Limitations [1/2]



Linux NAPI Limitations [2/2]

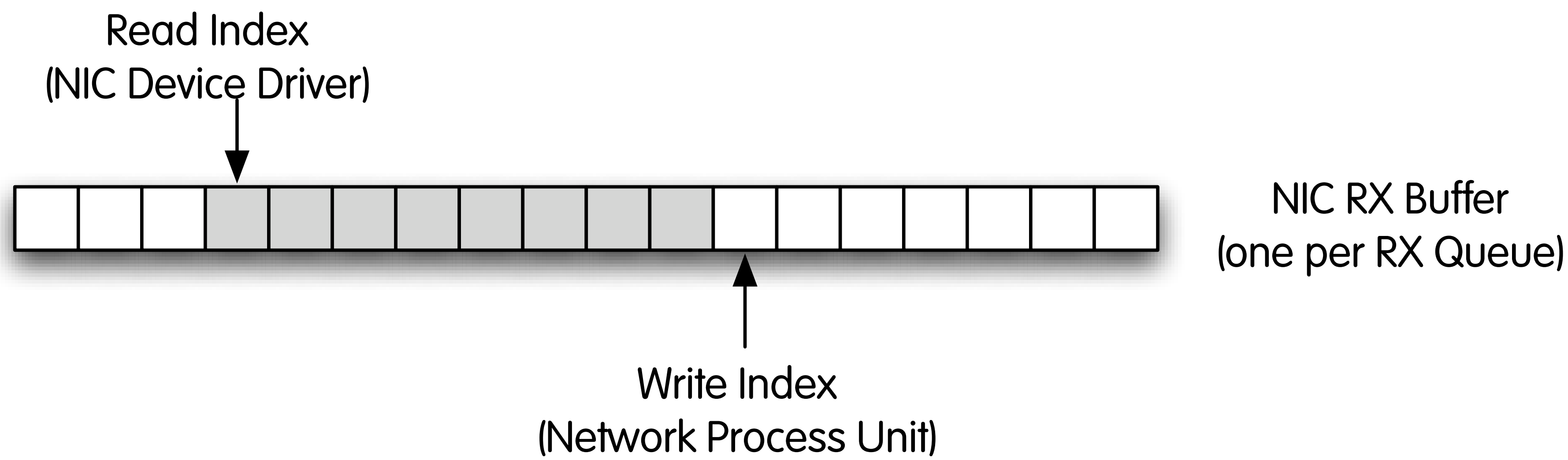
- Multiple-RX queues are not fully exploited by Linux as NAPI polls them in sequence and not concurrently
- Interrupts are enabled/disabled globally (i.e. for all queues at the same time) whereas they should be managed queue-per-queue as not all queues have the same amount of traffic (it depends on how balance-able is the ingress traffic).
- Original queue index (that can be used as flow identifier) is lost when the packet is propagated inside the kernel and then to userland.
- Userland applications see the NIC as a single entity and not as a collection of queues as it should be. This is a problem as the software could take advantage of multiple queues by avoiding threads competing for incoming packets all coming from the same NIC but from different queues.

Example of Multi-Queue NIC Statistics

```
# ethtool -S eth5
NIC statistics:
     rx_packets: 161216
     tx_packets: 0
     rx_bytes: 11606251
     tx_bytes: 0
     lsc_int: 1
     tx_busy: 0
     non_eop_descs: 0
     rx_errors: 0
     tx_errors: 0
     rx_dropped: 0
     tx_dropped: 0
     multicast: 4
     broadcast: 1
     rx_no_buffer_count: 2
     collisions: 0
     rx_over_errors: 0
     rx_crc_errors: 0
     rx_frame_errors: 0
     rx_fifo_errors: 0
     rx_missed_errors: 0
     tx_aborted_errors: 0
     tx_carrier_errors: 0
     tx_fifo_errors: 0
     tx_heartbeat_errors: 0
     tx_timeout_count: 0
     tx_restart_queue: 0
     rx_long_length_errors: 0
     rx_short_length_errors: 0
     tx_tcp4_seg_ctxt: 0
     tx_tcp6_seg_ctxt: 0
     tx_flow_control_xon: 0
     rx_flow_control_xon: 0
     tx_flow_control_xoff: 0
     rx_flow_control_xoff: 0
     rx_csum_offload_good: 153902
     rx_csum_offload_errors: 79
     tx_csum_offload_ctxt: 0
     rx_header_split: 73914
     low_latency_interrupt: 0
     alloc_rx_page_failed: 0
     alloc_rx_buff_failed: 0
     lro_flushed: 0
     lro_coal: 0
     tx_queue_0_packets: 0
     tx_queue_0_bytes: 0
     rx_queue_0_packets: 79589
     rx_queue_0_bytes: 5721731
     rx_queue_1_packets: 81627
     rx_queue_1_bytes: 5884520
```

Memory Allocation Life Cycle [1/5]

- Incoming packets are stored into kernel's memory that has been previously allocated by the driver.



- As soon that a packet is received, the NIC NPU (Network Process Unit) checks if there's an empty slot and if so it copies the packet in the slot.
- The slot is removed from the RX buffer and propagated through the kernel.
- A new bucket is allocated and places on the same position of the old slot.

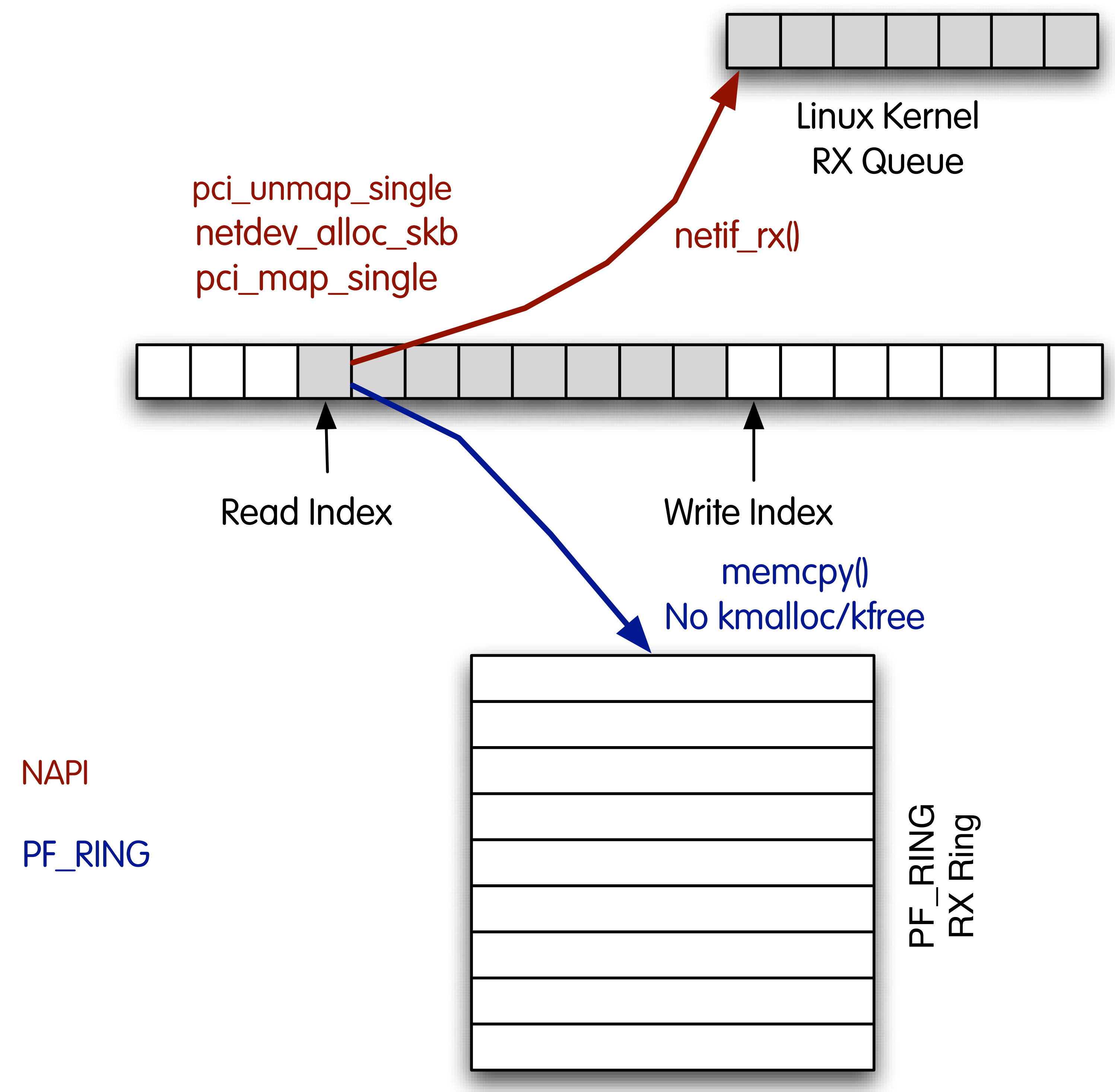
Memory Allocation Life Cycle [2/5]

- The consequence of this allocation policy is that:
  - Every new packet requires one slow allocation (and later-on a free).
  - As the traffic rate increases, increasing allocations/free will happen.
  - In particular at 10 Gbit, if there's a traffic spike or a traffic shot, the system may run out of memory as incoming packets:
    - require memory hence the memory allocator does its best to allocate new slots.
    - are stuck in the network kernel queue because the packet consumers cannot keep-up with the ingress traffic rate.
  - When the system runs in low memory it tries to free cached memory in order to free some space.
  - Unfortunately when the ingress rate is very high, the memory recover process does not have enough time hence the system runs out of memory and the result is that Linux's OOM (Out Of Memory) killer has to kill some processes in order to recover some memory.

Memory Allocation Life Cycle [3/5]

```
  if(rx_desc->status & E1000_RXD_STAT_DD) {
       /* A packet has been received */
#if defined (CONFIG_RING) || defined(CONFIG_RING_MODULE)
          handle_ring_skb ring_handler = get_skb_ring_handler();

          if(ring_handler && adapter->soncap.soncap_enabled) {
            ring_handler(skb, 0, 1, (hash_value % MAX_NUM_CHANNELS));
          } else {
#endif
          [.....]
              if (++i == rx_ring->count) i = 0;
              next_rxd = E1000_RX_DESC(*rx_ring, i);
              prefetch(next_rxd);
              next_buffer = &rx_ring->buffer_info[i];
              cleaned = TRUE;
              cleaned_count++;
              pci_unmap_single(pdev, buffer_info->dma, PAGE_SIZE, PCI_DMA_FROMDEVICE);
              [.....]
              skb = netdev_alloc_skb(netdev, bufsz);
              buffer_info->dma = pci_map_single(pdev,
                                                 skb->data,
                                                 adapter->rx_buffer_len,
                                                 PCI_DMA_FROMDEVICE);
              [.....]
```

Memory Allocation Life Cycle [4/5]



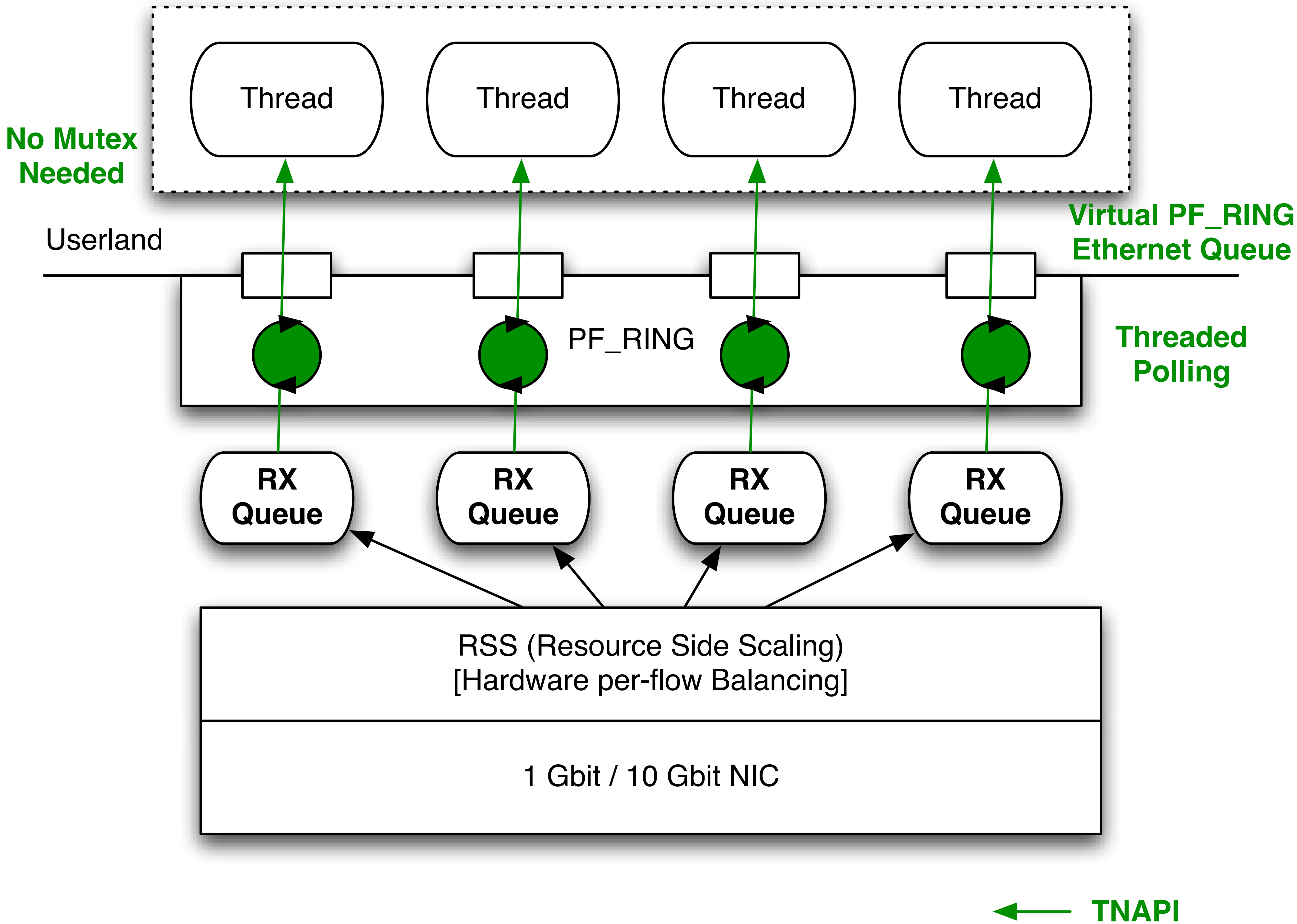
Memory Allocation Life Cycle [5/5]

- Avoiding memory allocation/deallocation has several advantages:
  - No need to allocate/free buffers
  - No need to map memory though the PCI bus
  - In case of too much incoming traffic, as the kernel has more priority than userland applications, there's no risk to run out of memory as it happens with standard NAPI.
- The last advantage of doing a packet copy to the PF_RING buffer is the speed. Depending on the setup, the packet capture performance can be increased of 10-20% with respect to standard NAPI.

Enhanced NIC Drivers: TNAPI [1/8]

- In order to enhance and accelerate packet capture under Linux, a new Linux driver for Intel 1 and 10 Gbit cards has been developed. Main features are:
  - 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)
  - 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 82598/9 (10G) and 82575/6 (1G) ethernet controllers.
- For this reason the driver has been called TNAPI (Threaded NAPI).

Enhanced NIC Drivers: TNAPI [2/8]



Enhanced NIC Drivers: TNAPI [3/8]

- Packet capture has been greatly accelerated thanks to TNAPI as:
  - Each RX queue is finally independent (interrupts are turned on/off per queue and not per card)
  - Each RX queue has a thread associated and mapped on the same CPU core as the one used for RSS (i.e. cache is not invalidated)
  - The kernel thread pushes packets as fast as possible up on the networking stack.
  - Packets are copied from the NIC directly into PF_RING (allocation/deallocation of skbuffers is avoided).
  - Userland applications can capture packets from a virtual ethernet NIC that maps the RX ring directly into userspace via PF_RING.

Enhanced NIC Drivers: TNAPI [4/8]

- TNAPI Issues: CPU Monopolization
  - As the thread pushes packets onto PF_RING, it should be avoided that this thread monopolizes. This is because of the all CPU is used by the kernel for receiving packets, then packet loss won't happen in kernel but in userspace (i.e. the packet loss problem is not solved, but just moved).

```
while(<polling packets from RX queue X>) {
  /* Avoid CPU monopolization */
    if(rx_budget > 0)
      rx_budget--;
    else {
      rx_budget = DEFAULT_RX_BUDGET;
      yield();
    }
}
```

  - Solution: every X polling cycles, the thread has to give away some CPU cycles. This is implemented as follow rx_budget that's consumed whenever a packet is received and sent to PF_RING.

Enhanced NIC Drivers: TNAPI [5/8]

- TNAPI Issues: Interrupts and Cores Allocation
  - RX ring interrupts must be sent to the right core that's manipulating the queue in order to preserve cache coherency.
  - The userland application that's fetching packets from queue X, should also be mapped to core X.
  - As interrupts are now sent per-queue (and not per-nic as it used to be) we must make sure that they are sent to the same core that's fetching packets.

```
# cat /proc/interrupts
         CPU0     CPU1     CPU2     CPU3     CPU4     CPU5     CPU6     CPU7
191:        1        1     2656        1        2        2        1        2   PCI-MSI-edge     eth3
192:        1        4        0        0     2655        3        1        2   PCI-MSI-edge     eth2
193:    78634       14        7       13        9       13       13       18   PCI-MSI-edge     eth1
194:        3    15964        6        3        3        5        3        5   PCI-MSI-edge     eth0
195:        0        0        0        0        0        0        0        0   PCI-MSI-edge     eth7:lsc
196:        1        2        2        0        0     2658        1        0   PCI-MSI-edge     eth7:v8-Tx
197:        1        0        2        0        1        0        1     5309   PCI-MSI-edge     eth7:v7-Rx
198:        1        0        0     5309        1        2        0        1   PCI-MSI-edge     eth7:v6-Rx
199:        0        1        0        1        0        1        2     5309   PCI-MSI-edge     eth7:v5-Rx
200:        0        1        1     5307        2        2        1        0   PCI-MSI-edge     eth7:v4-Rx
201:        1        0        1        2        1     5307        2        0   PCI-MSI-edge     eth7:v3-Rx
202:        2        2        0        1        1        0     5307        1   PCI-MSI-edge     eth7:v2-Rx
203:        0        1     5309        1        1        1        0        1   PCI-MSI-edge     eth7:v1-Rx
204:        2        2        1        0     5307        1        1        0   PCI-MSI-edge     eth7:v0-Rx
```

Enhanced NIC Drivers: TNAPI [6/8]

- Example:
  - RX ring 6 and 4 use the same CPU 3.
  - We want to move RX ring 6 to CPU 1

```
# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
198:          1          0          0       5309          1          2          0          1   PCI-MSI-edge      eth7:v6-Rx
200:          0          1          1       5307          2          2          1          0   PCI-MSI-edge      eth7:v4-Rx

# cat /proc/irq/198/smp_affinity
00000008
# echo 2 > /proc/irq/198/smp_affinity [00000010 where 1 = CPU 1]
# cat /proc/irq/198/smp_affinity
00000002
# cat /proc/interrupts |grep "eth7:v6-Rx"
198:         0         67            0       5309          1          2          0          1   PCI-MSI-edge      eth7:v6-Rx
```

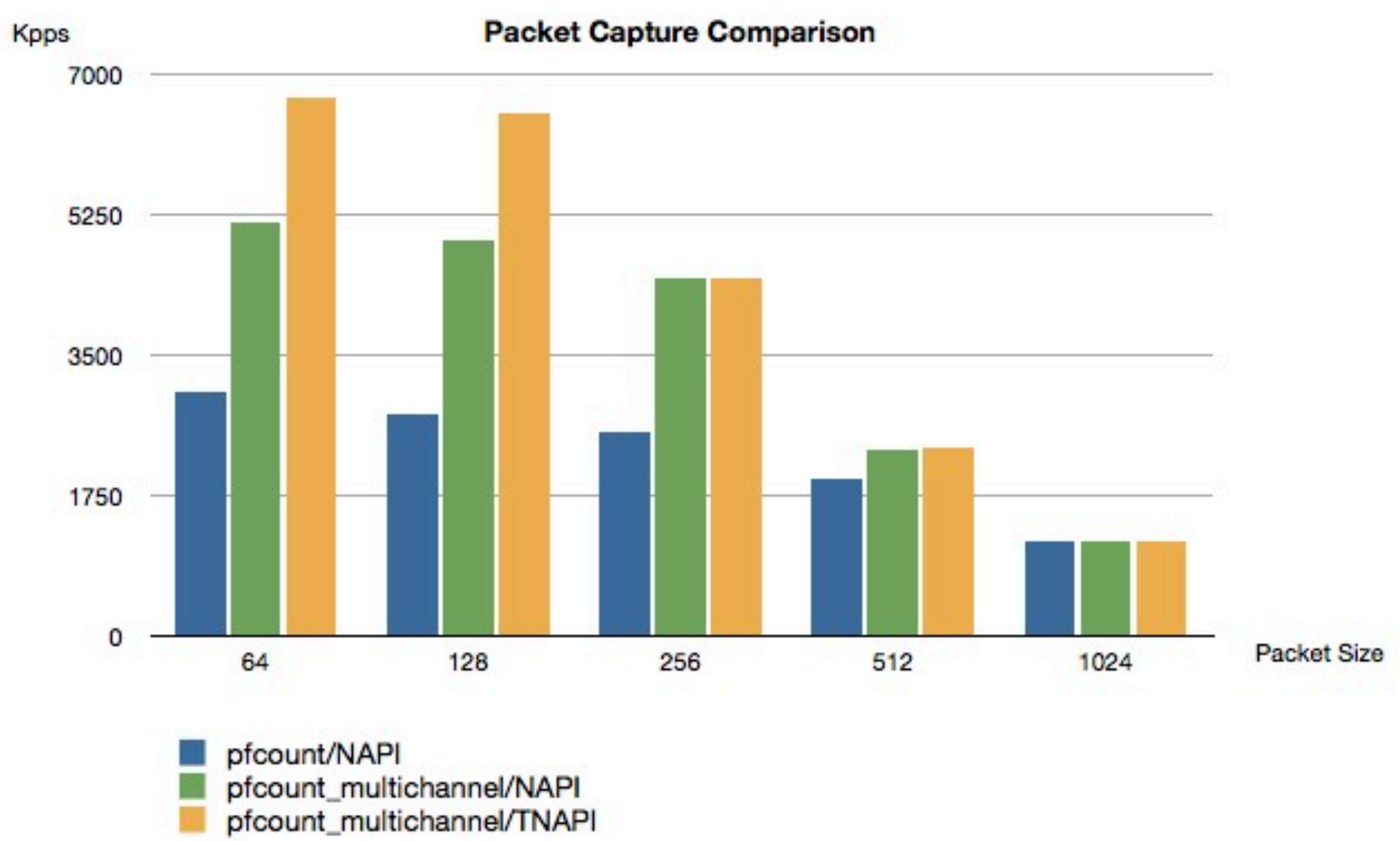
```
unsigned long mask = 7; /* processors 0, 1, and 2 */
unsigned int len = sizeof(mask);
if (sched_setaffinity(0, len, &mask) < 0) {
    perror("sched_setaffinity");
}
```

  - How to map a process to a CPU/core

Enhanced NIC Drivers: TNAPI [7/8]

Test Type	Max Packet Capture Speed	
PF_RING	300K pps	560K pps
PF_RING+TNAPI Mono RX queue	750K pps	920K pps
PF_RING+TNAPI Multi RX queue	860K pps	Wire Rate (1 Gbit) ~ 3 Million pps (10 Gbit) ~ 5 Million pps (10 Gbit - 2 x Xeon)
	Intel Core2Duo 1.86 GHz (Dual Core) No Intel I/OAT	CPU Intel Xeon 2.4 GHz (Quad Core) Intel 5000 chipset (I/OAT support)

Enhanced NIC Drivers: TNAPI [8/8]



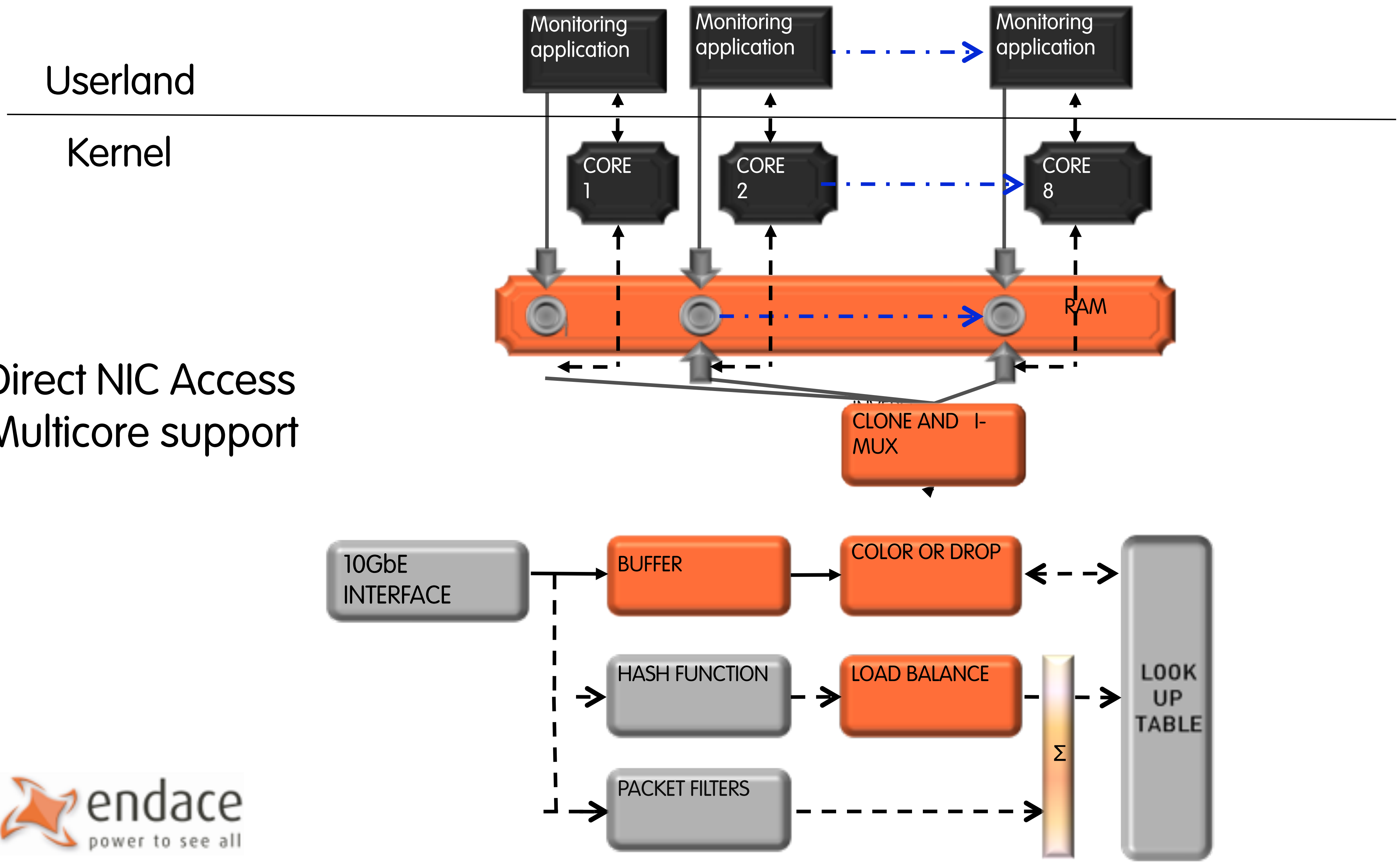
Testbed: Xeon X3450 @ 2.67GHz

RX Multi-Queue and DNA

- As previously explained, DNA is an excellent technology for those application developers who need wire speed packet capture, but that do not need features such as:
  - packet filtering
  - multiple application packet consumers.

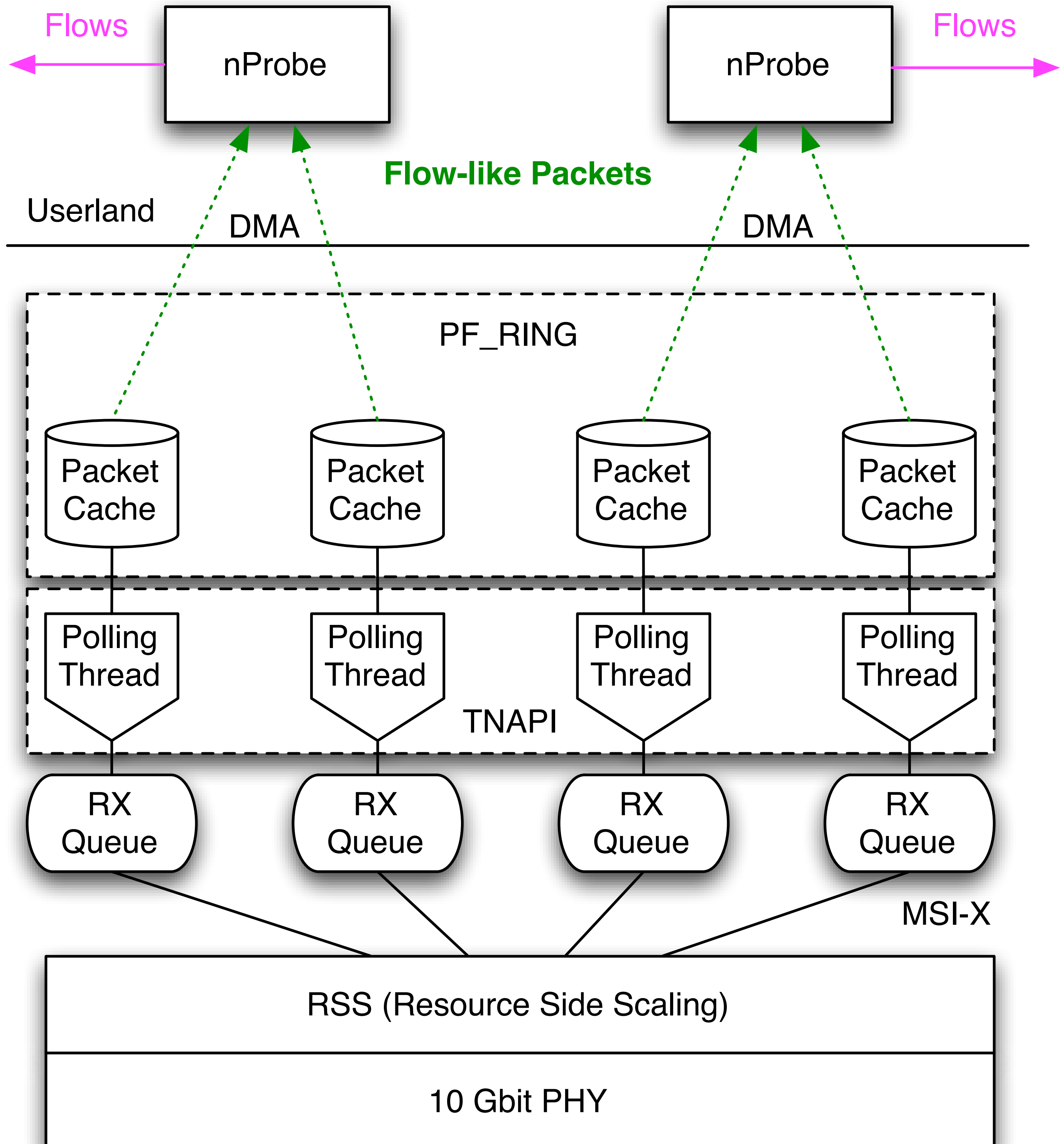
- DNA so far has been ported to the Intel mono-queue 1 Gbit driver (e1000) and multi-queue 10 Gbit driver (ixgbe).
- Currently the port of DNA to 1 Gbit RX multi-queue driver (igb) is in progress and it will be available later this year.
- Combining DNA with multi-queue allows applications to be split into concurrent execution threads that enables multicore architectures to be further exploited.
- Additionally by exploiting hardware traffic balancing, it allows flow-based applications such as netflow probes, to be further accelerated.

Multi-Queue on Accelerated NICs



- Direct NIC Access
- Multicore support

Exploiting PF_RING Multi-Queue: nProbe



- Packet balancing across cores.
- Peak nProbe performance: 1.48 Mpps (packet rate) x 2 Cores.

Strong Multicore NICs: Tilera Tile64

Towards Strong Multicore [1/2]

General Perception is that people usually think that multicore is a good idea, although difficult to implement.

- General PC market
  - Input data is unstructured, sequential
  - Billions of lines of sequential applications
  - Hard to migrate it to parallel code

- Embedded market
  - Data is inherently parallel
  - Engineers have designed parallel applications
  - Their main challenge is complexity of design

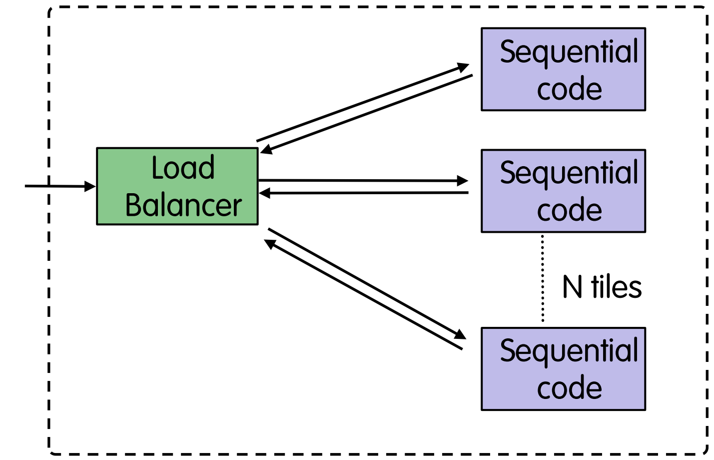
Towards Strong Multicore [2/2]

- Some applications are naturally parallel as in networking where a network pipe is a multiplex of many "flows" or distinct streams.

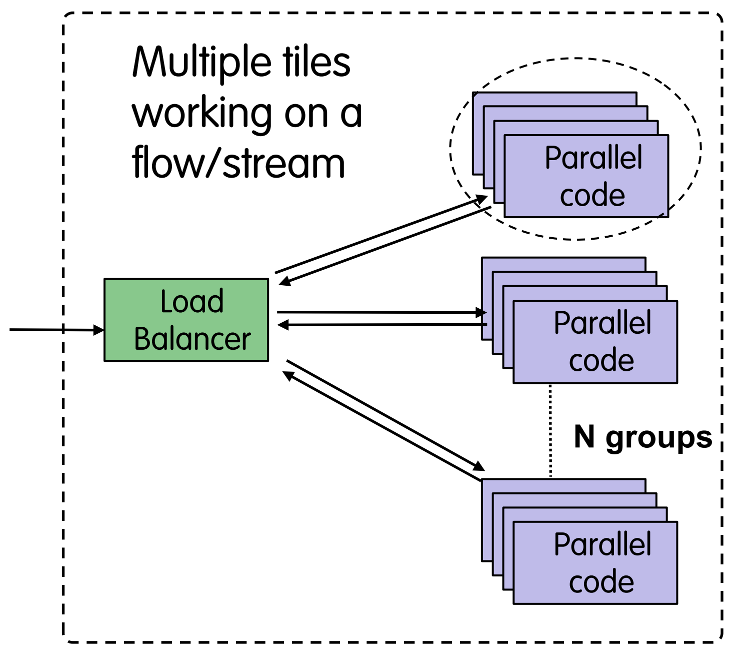
- The only barriers towards adopting strong multicore are:
  - Design the application program so that it can take advantage of multicore without sequentially performing activities that could be carried on in parallel.
  - Entry ticket for learning multicore development tools.
  - Low-level programming required to take advantage of the technology.

Programming Paradigms

- Run to completion model
  - Sequential C/C++ applications
  - Run multiple application instances one/core
  - Use load balancer library for distribution
  - Use tools to tune performance

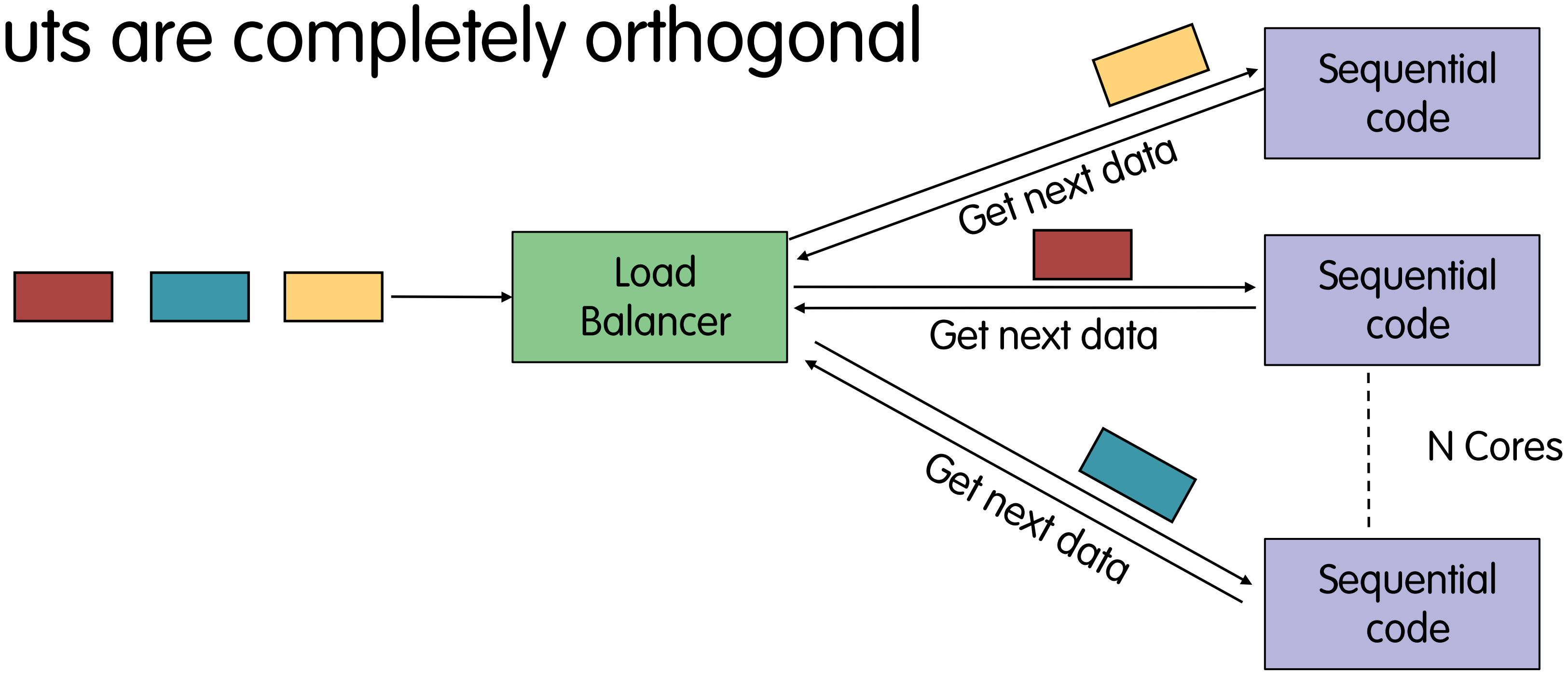


- Parallel programming
  - Parallelize application with pthreads shared memory
  - Run on multiple cores
  - Use communication libraries to optimize
  - Use tools to tune performance



Parallel Processing Without Parallel Programming

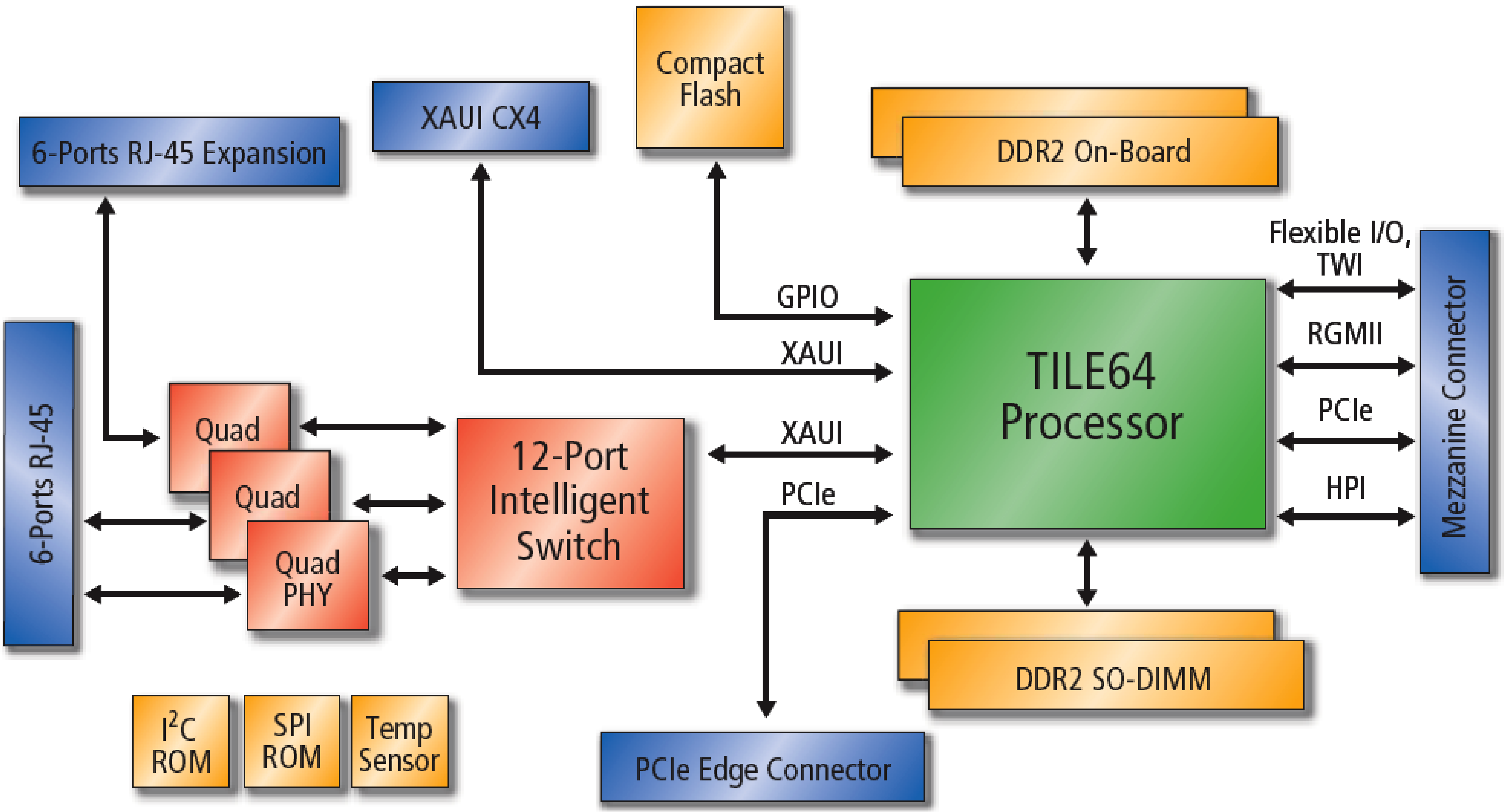
- Standard model in the embedded world
  - Facilitates immediate results using off-the-self code
- Simple architecture
  - Each core runs complete application and handles one or multiple flows or channels
  - I/O management and load distribution
  - Most embedded applications fit this category
  - Large numbers of flows, frames channels, streams, etc…
  - Most inputs are completely orthogonal



Tilera TILExpress64

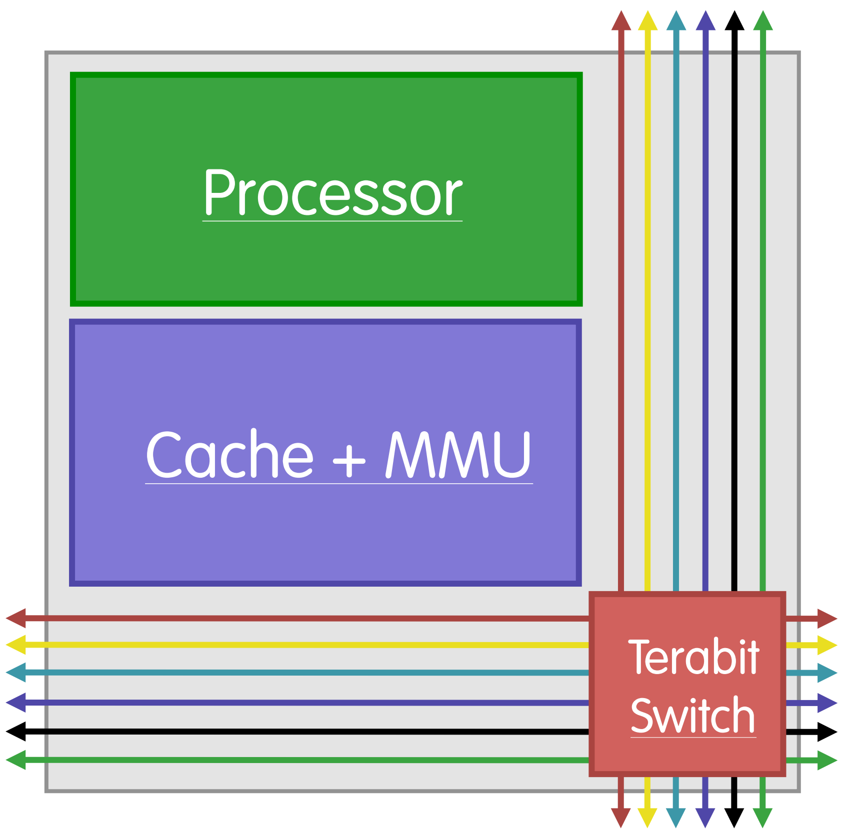
- 64-core CPU.
- Linux-based 2.6 operating system running on board.
- Programmable in C/C++.
- Eclipse Integration for easing software development and debugging.

TILE64 Architecture [1/2]



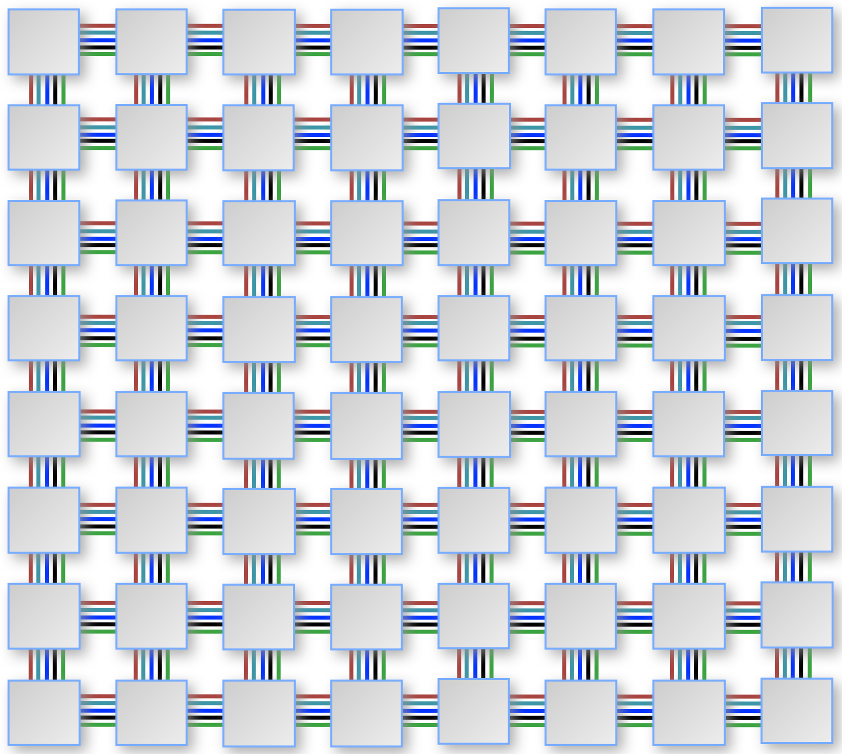
TILE64 Architecture [2/2]

Tile = Processor + Cache + Switch



Each tile is a complete processor

2 Dimensional iMesh connects tiles

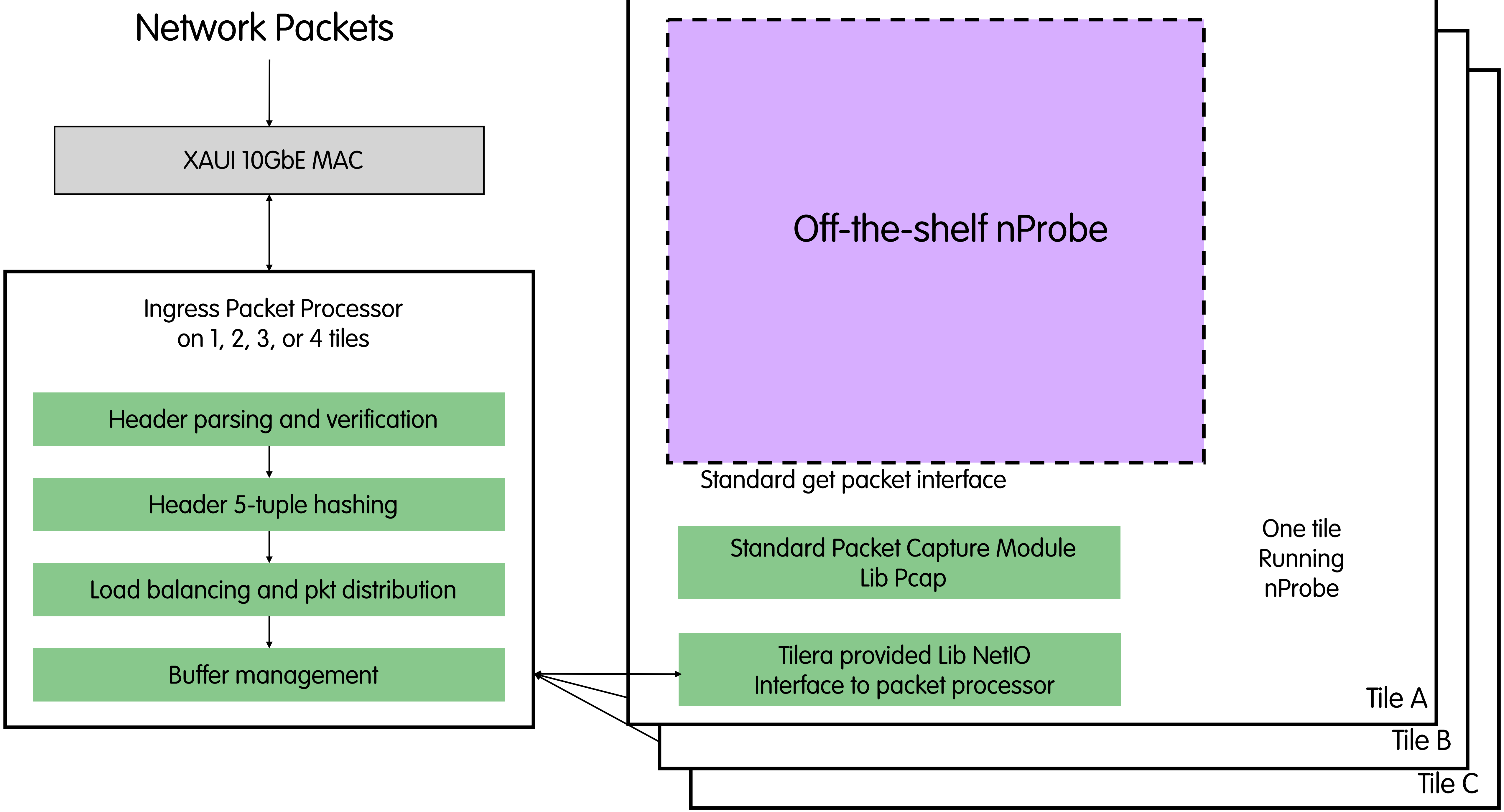


38 terabits of on-chip bandwidth

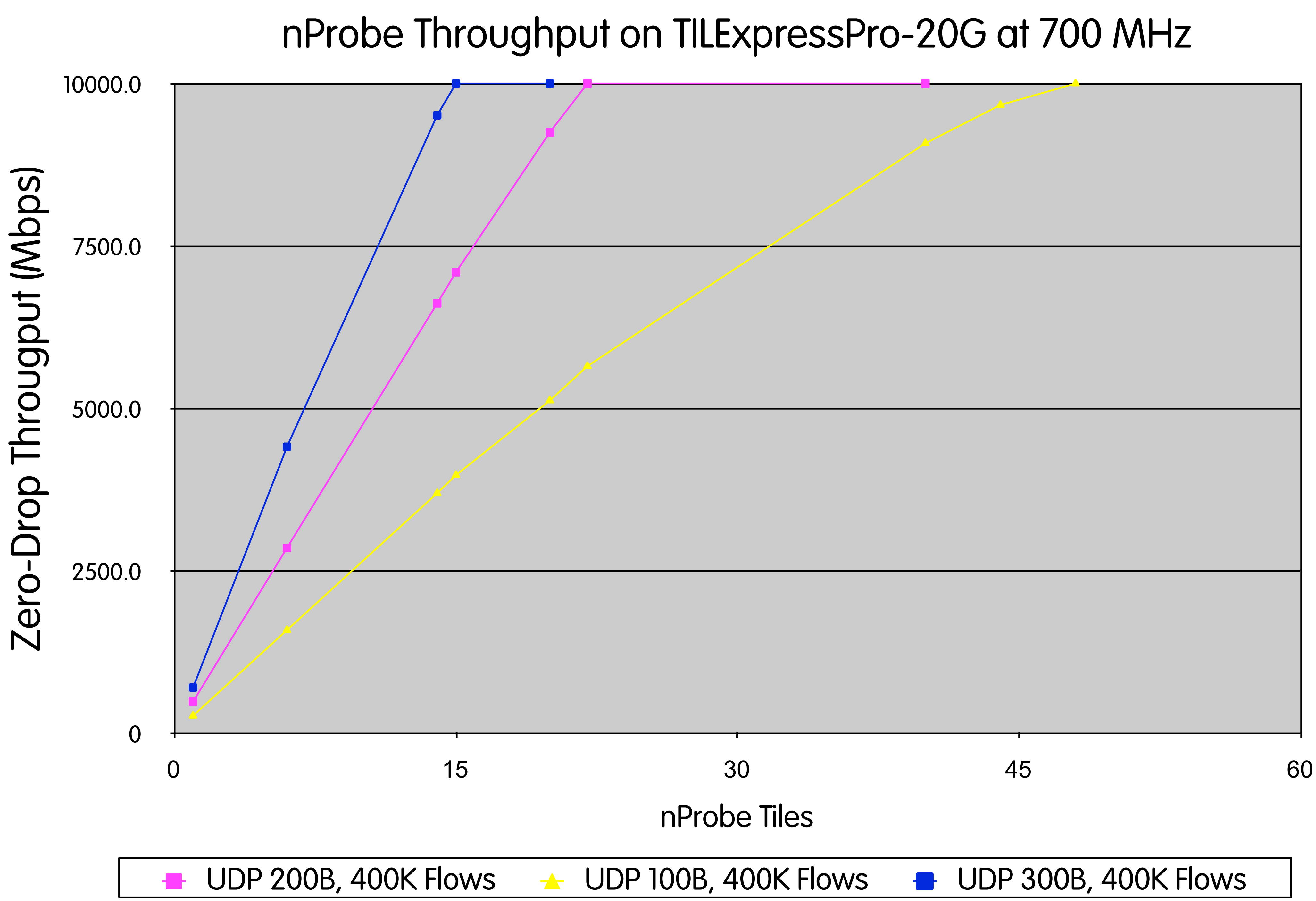
Tilera Advantages

- No need to capture packets as it happens with PCs.
- 12 x 1 Gbit, or 6 x 1Gbit and 1 x 10 Gbit Interfaces (XAUI connector).
- Ability to boot from flash for creating stand-alone products.
- Standard Linux development tools available including libpcap for packet capture.
- Application porting is very quick and simple: less than 100 lines of code changed in nProbe.

Porting Exiting Applications to Tile64: nProbe



nProbe Performance on Tile64



Final Remarks

Programming for Multicore [1/4]

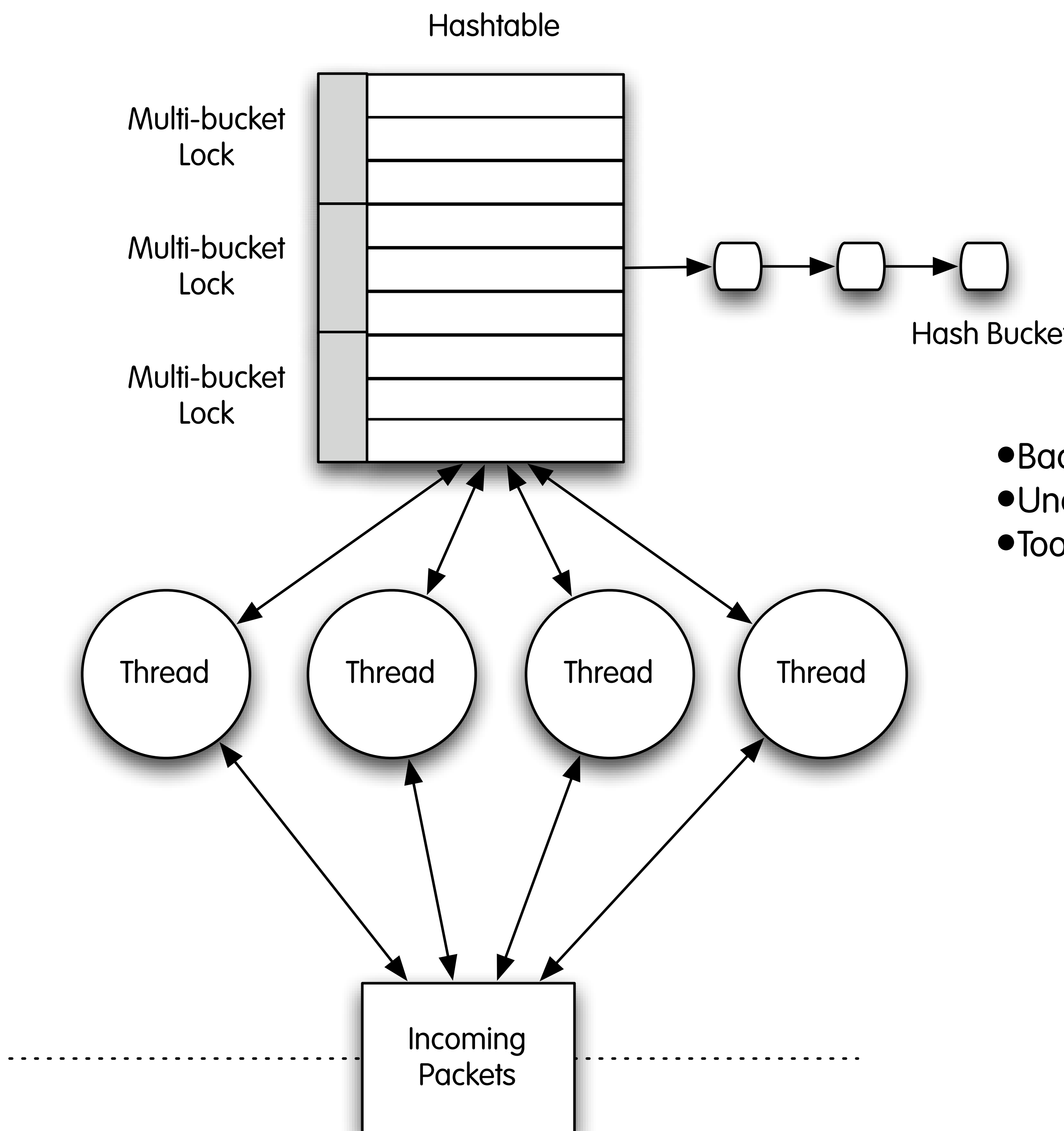
- Multicore is not the solution to all performance and scalability problems.
- Actually it can decrease the performance of poorly designed applications.
- Like it or not, multicore is the future of CPUs, and programmers have to face with it.
- From author's experience before adding threads and semaphores to parallelize an existing program, it's worth to think if instead the basic algorithm used are compatible with multicore.

Programming for Multicore [2/4]

- When multiple cores are used, efficient memory caching is the way to improve application performance.
- Hardware CPU caches are rather sophisticated, however they cannot work optimally without programmer's assistance.
- Cache coherence can be rather costly if programs invalidate it when not necessary.
- False sharing (when a system participant attempts to periodically access data that will never be altered by another party, but that data shares a cache block with data that is altered, the caching protocol may force the first participant to reload the whole unit despite a lack of logical necessity) is just an example of performance degrading due to poor programming.

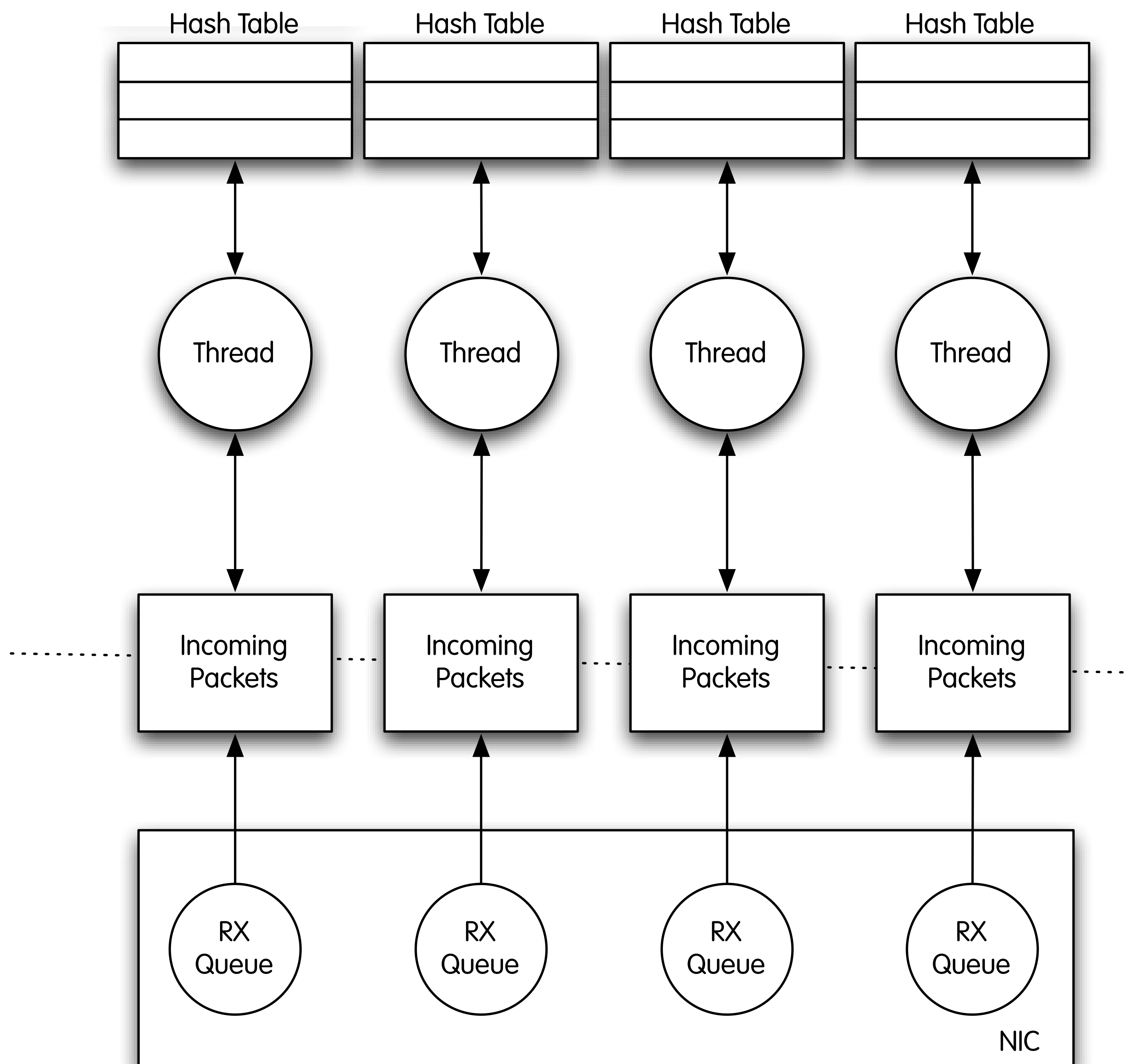
- Reference
  - U. Drepped, What Every Programmer Should Know About Memory, http://people.redhat.com/drepper/cpumemory.pdf, RedHat 2007.

Programming for Multicore [3/4]



- Bad Application Design
- Unable to scale
- Too much locking

Programming for Multicore [4/4]



- Great Application Design
- Exploit Native Multicore
- Fully Lockless Hash

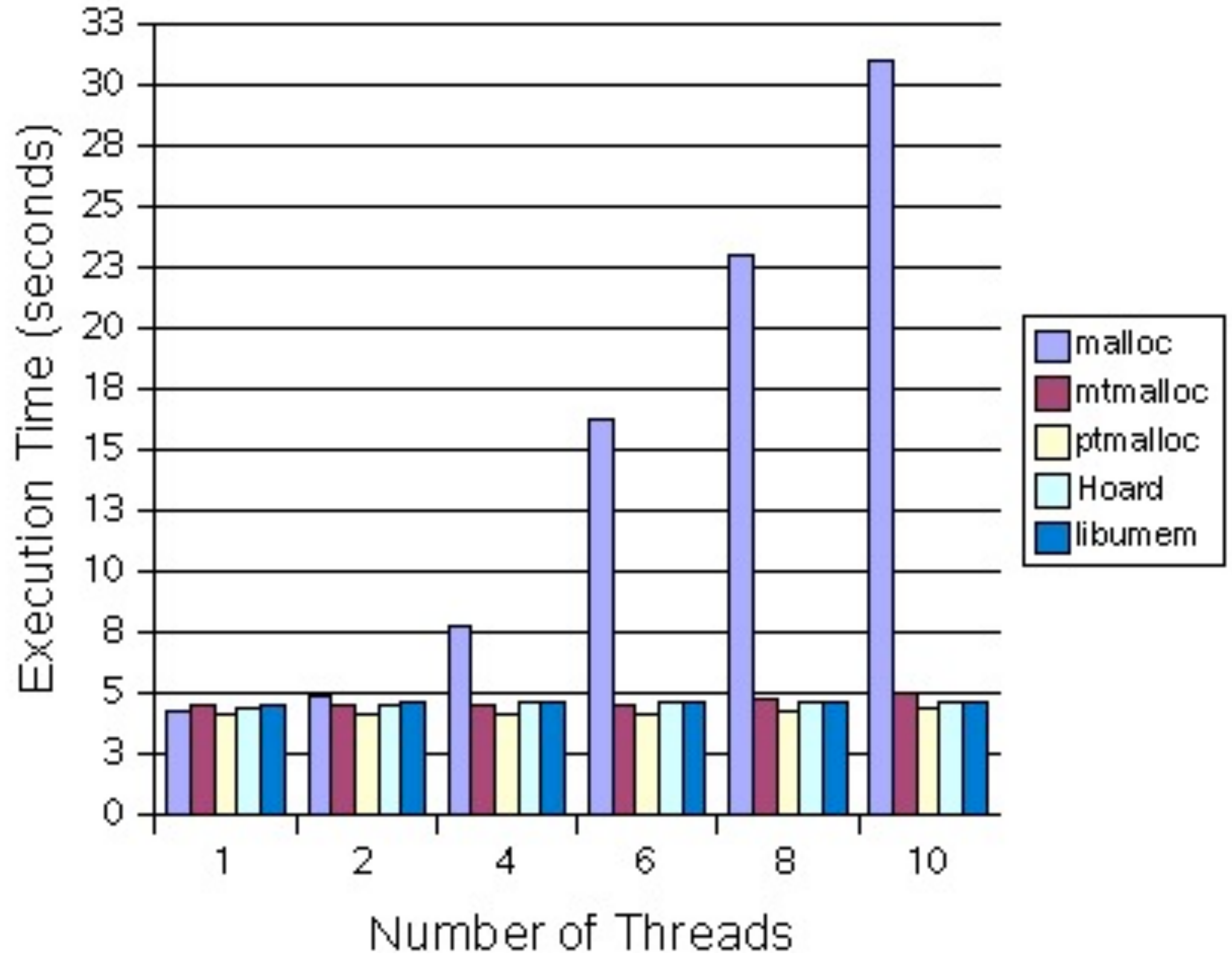
Lockeless hashes:

http://video.google.com/videoplay?docid=2139967204534450862

Memory Allocation [1/2]

Limit Memory Allocation (if not necessary)

- Multithreaded programs often do not scale because the heap is a bottleneck.
- When multiple threads simultaneously allocate or deallocate memory from the allocator, the allocator will serialize them.
- Programs making intensive use of the allocator actually slow down as the number of processors increases.



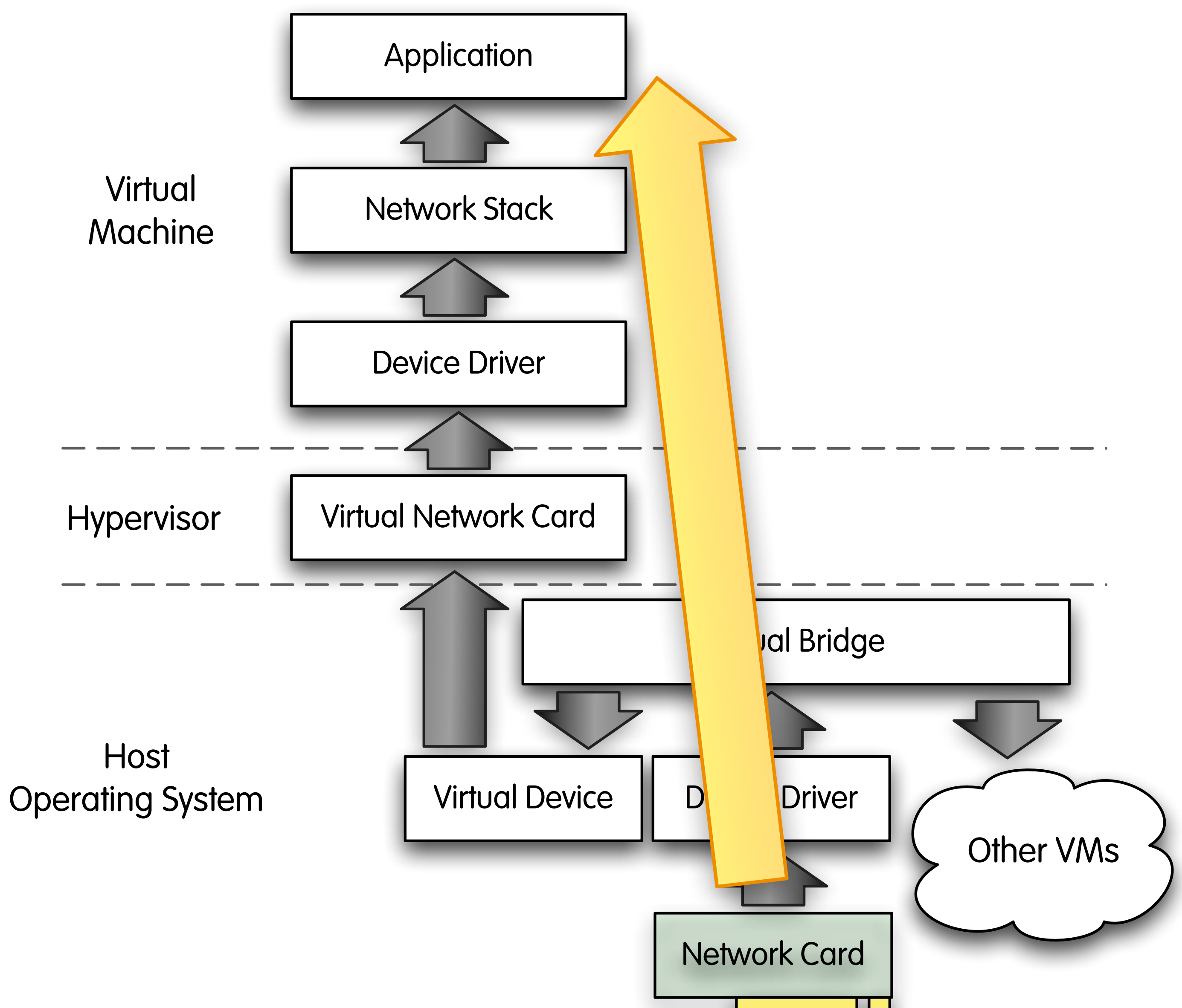
Memory Allocation [2/2]

- Programs should avoid, if possible, allocating/deallocations memory too often and in particular whenever a packet is received.
- In the Linux kernel there are available kernel/driver patches for recycling skbuff (kernel memory used to store incoming/outgoing packets).
- Using PF_RING (into the driver) for copying packets from the NIC to the circular buffer without any memory allocation increases the capture performance (around 10%) and reduces congestion issues.

References:

- A Comparison of Memory Allocators
  http://developers.sun.com/solaris/articles/multiproc/multiproc.html
- The Hoard Memory Allocator
  http://www.hoard.org/

PF_RING on VMs [1/4]



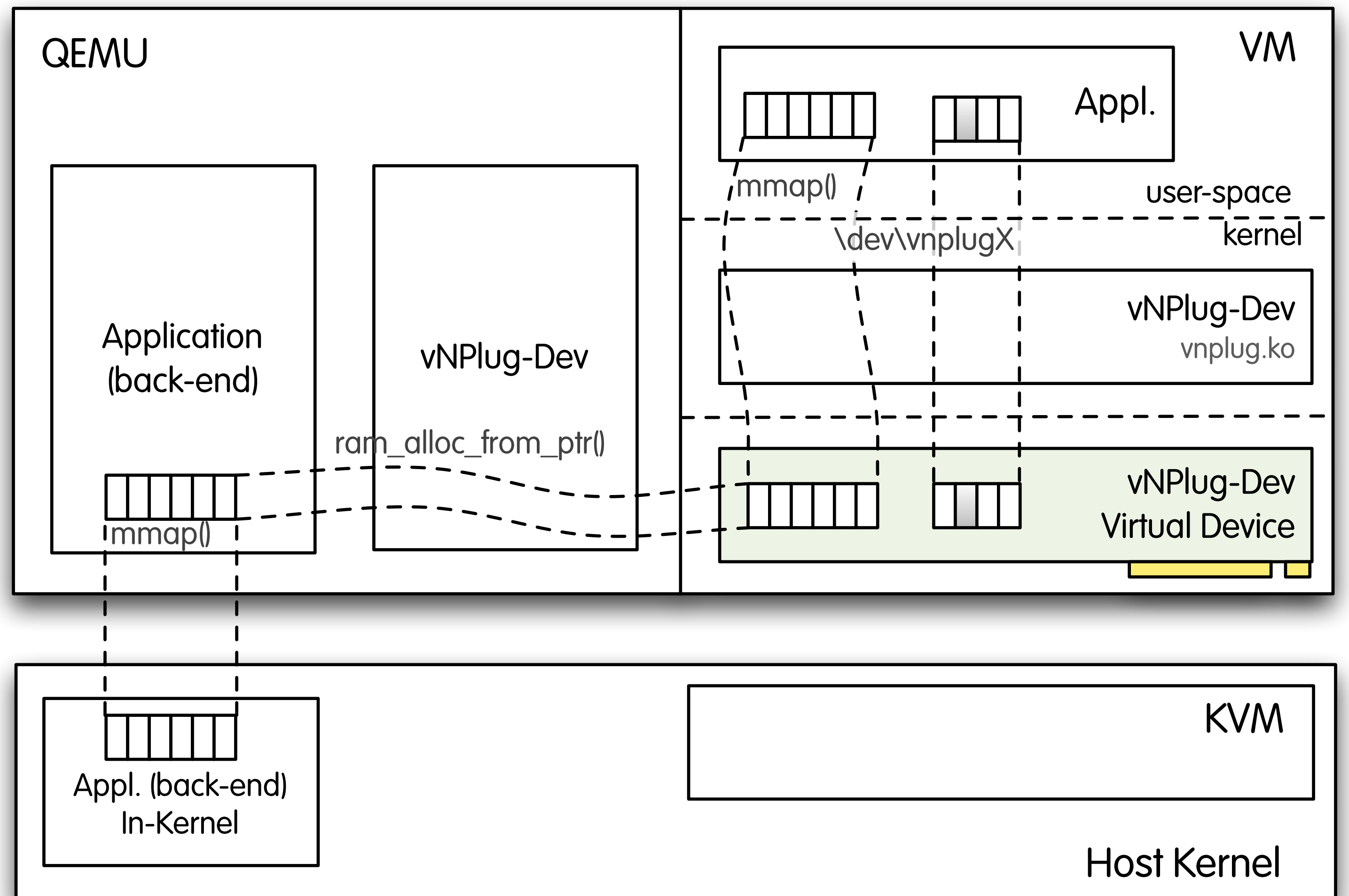
Open Issues

- Long packet journey from NIC to the VM.
- Various packet copies are involved.
- Packets replicated on all VMs.
- Overhead due to the abstraction level.

Goal

- Implement straight capture to the VM.

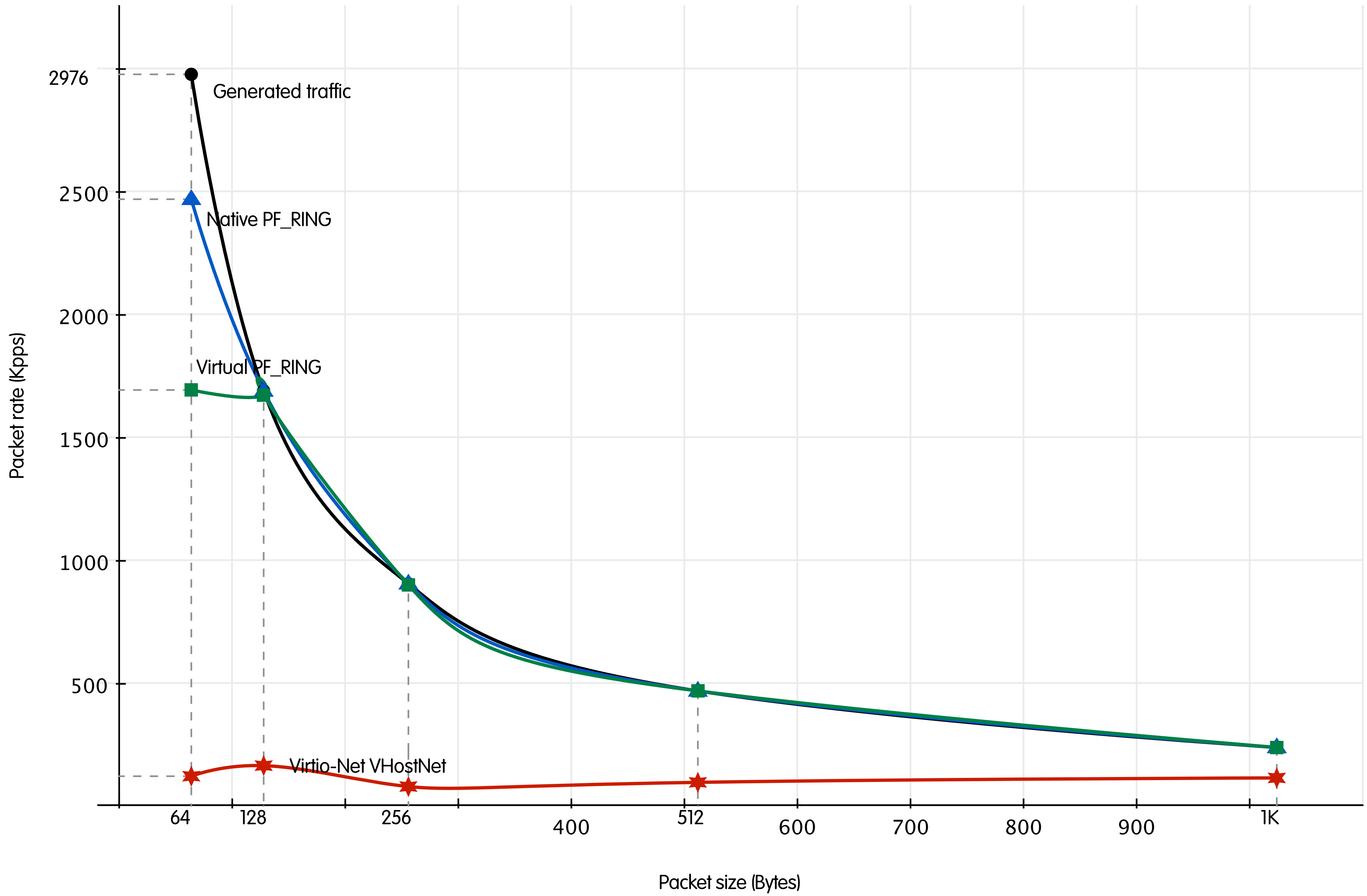
PF_RING on VMs [2/4]



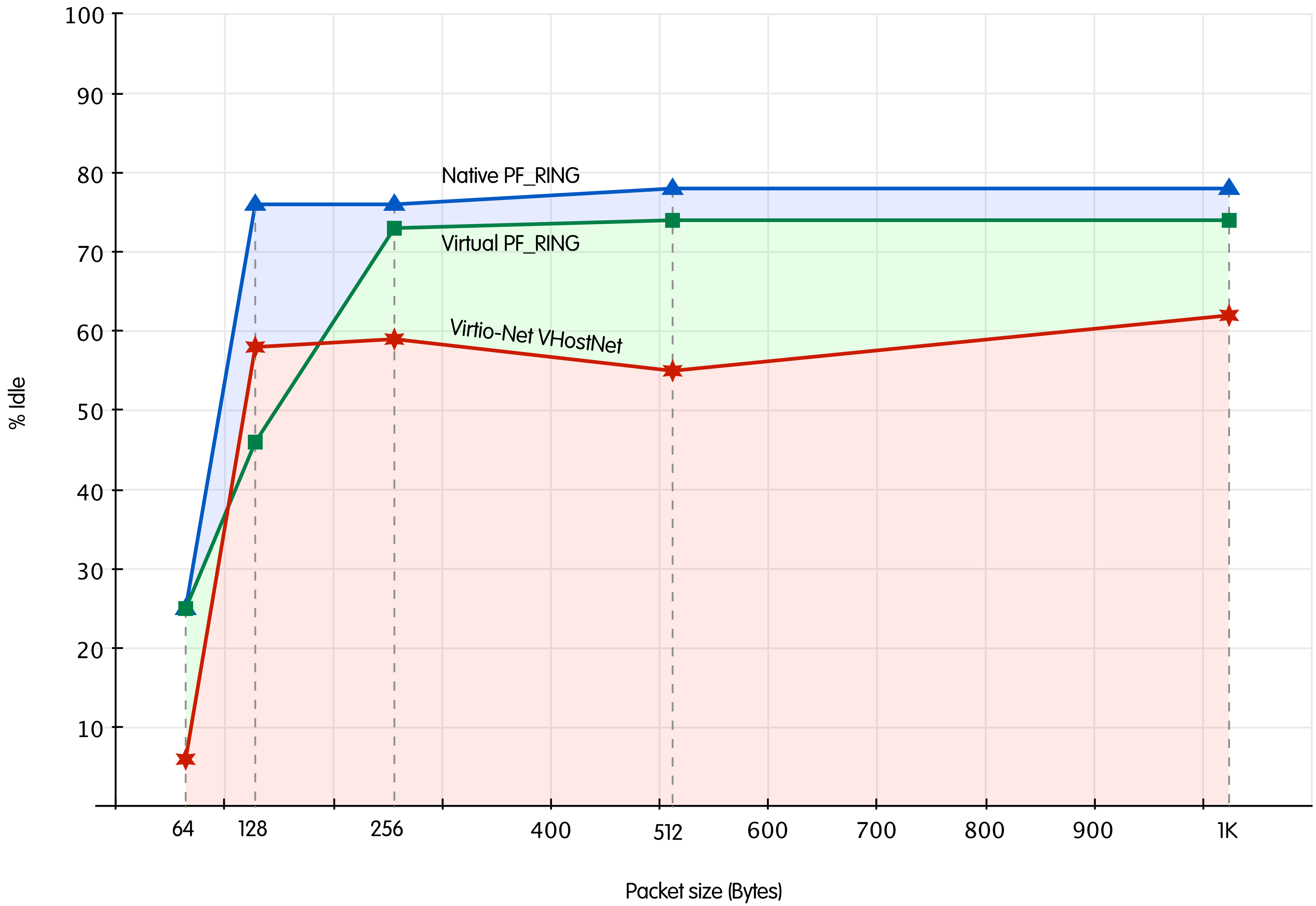
vNPlug

- It implements a shared memory area (host <->VM) that is mapped as a dummy PCI device
- Host->Guest signaling by emulating interrupts.
- Guest->Host signaling by writing on PCI registers that are monitored via ioeventfd.

PF_RING on VMs [3/4]



PF_RING on VMs [4/4]



References

- http://www.ntop.org/
- http://www.intel.com/cd/network/connectivity/emea/eng/226275.htm
- http://www.tilera.com

Email: Luca Deri <deri@ntop.org>

High-Speed Traffic Capture and Analysis Using Open-Source Software and Commodity Hardware

Part 2: Traffic Monitoring

Luca Deri <deri@ntop.org>

Monitoring Goals

- Analysis of LAN and WAN Traffic
- Unaggregated raw data storage for the near past (-3 days) and long-term data aggregation on selected network traffic metrics (limit: available disk space)
- Data navigation by means of a web 2.0 GUI
- Geolocation of network flows and their aggregation based on their geographical source.
- Integration with routing information in order to provide accurate traffic path analysis.

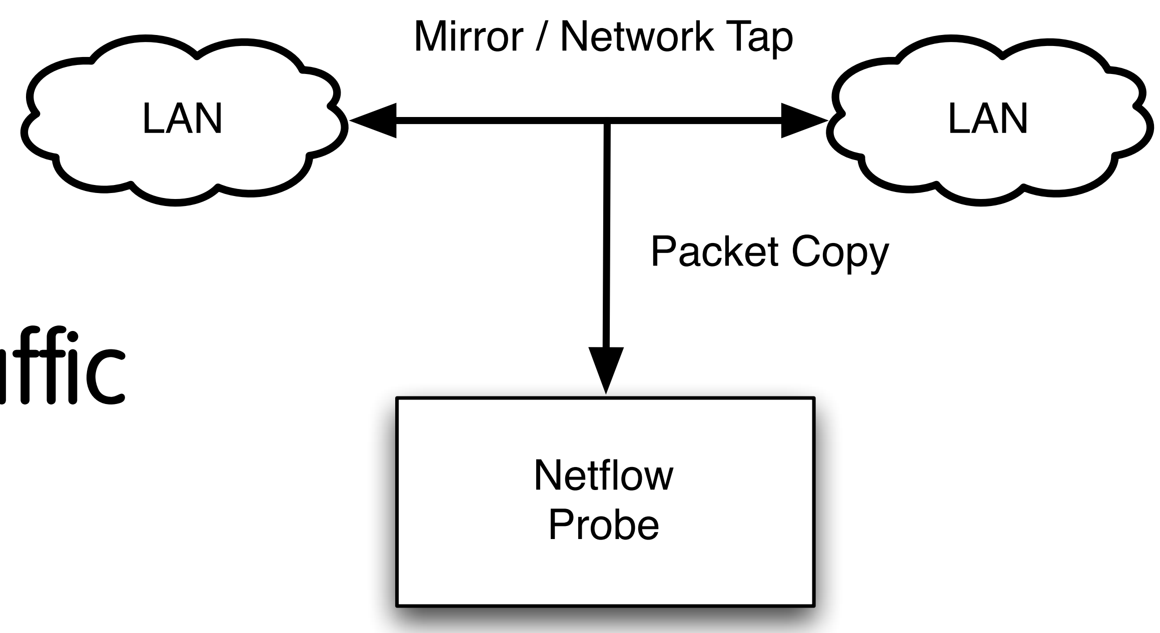
Traffic Collection Architecture [1/2]

- Available Options
  1. Exploit network equipment (routers and switches)
  - Advantages:
    - Maximize investment.
    - Avoid adding extra network equipment/complexity in the network.
    - No additional point of Failure
  - Disadvantages:
    - Often is necessary to buy costly netflow engines
    - Have to survive with bugs (e.g. Juniper have issues with AS information)

Traffic Collection Architecture [2/2]

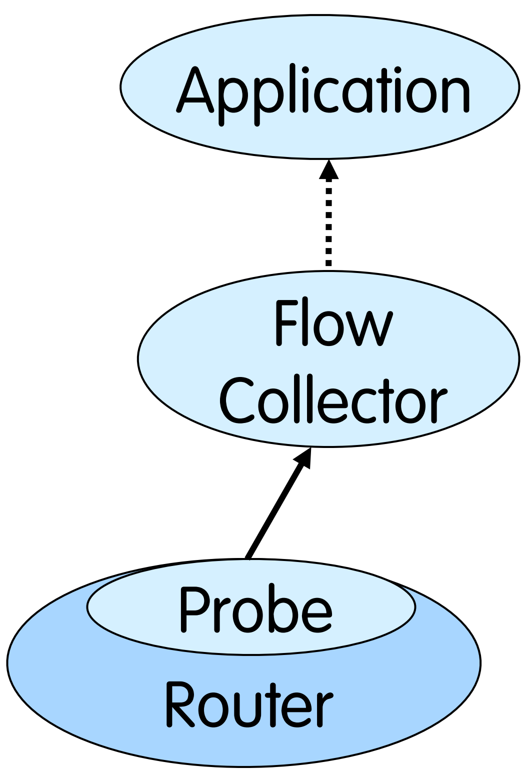
2. Custom Network Probes

- Advantages
  - Ability to avoid limitations of commercial equipment
  - (Often) Faster and more flexible than hw probes
- Disadvantages
  - Add complexity to the net
  - Need to mirror/wiretap traffic

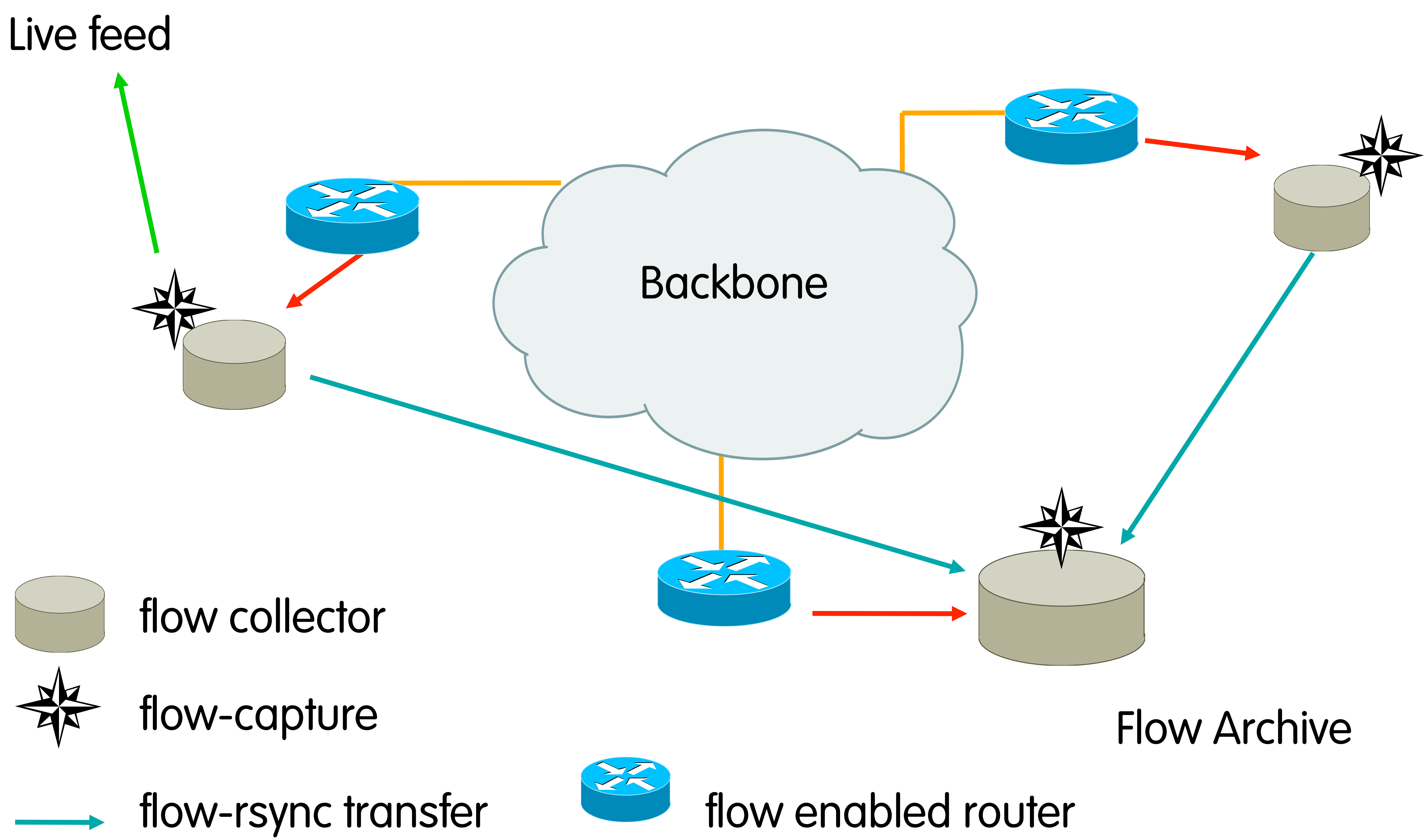


Introduction to Cisco NetFlow

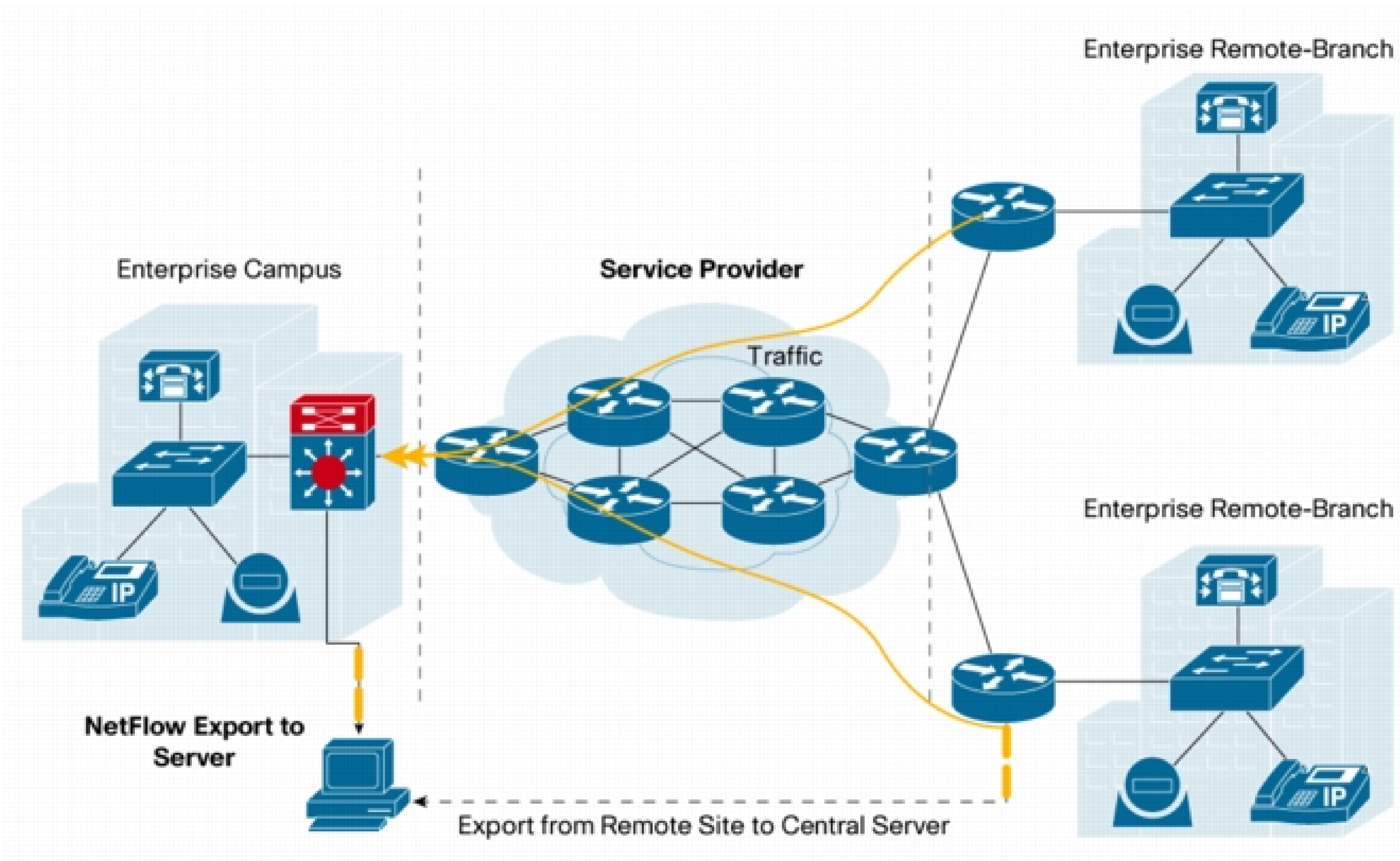
- Flow: "Set of network packets with some properties in common". Typically (IP src/dst, Port src/dst, Proto, TOS, VLAN).
- Network Flows contain:
  - Peers: flow source and destination.
  - Counters: packets, bytes, time.
  - Routing information: AS, network mask, interfaces.



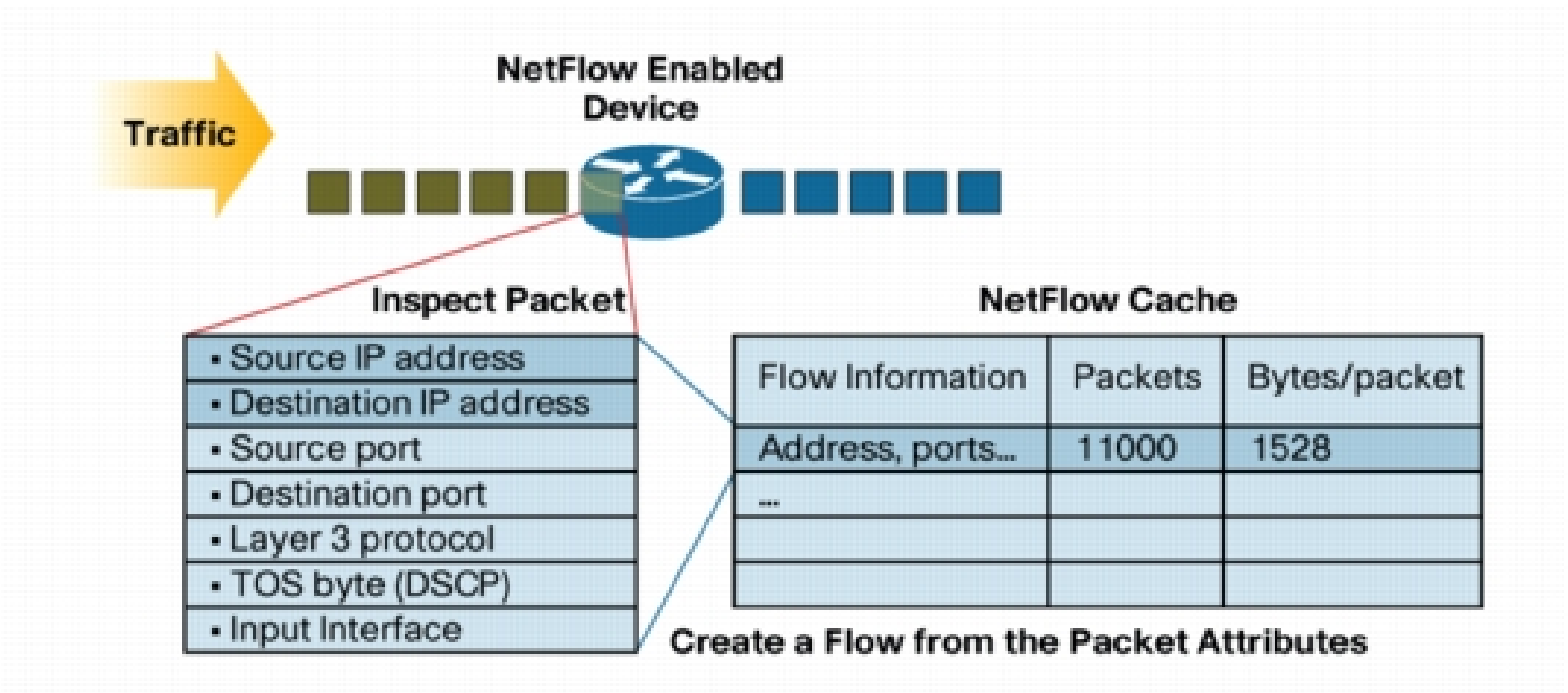
Collection Architectures [1/2]



Collection Architectures [2/2]



Flow Journey: Creation



Flow Journey: Export

1. Flow Cache—The First Unique Packet Creates a Flow

SrcIf	SrcIPadd	DstIf	DstIPadd	Protocol	TOS	Flgs	Pkts	Src Port	Src Msk	Src AS	Dst Port	Dst Msk	Dst AS	NextHop	Bytes/Pkt	Active	Idle
Fa1/0	173.100.21.2	Fa0/0	10.0.227.12	11	80	10	11000	162	/24	5	163	/24	15	10.0.23.2	1528	1745	4
Fa1/0	173.100.3.2	Fa0/0	10.0.227.12	6	40	0	2491	15	/26	196	15	/24	15	10.0.23.2	740	41.5	1
Fa1/0	173.100.20.2	Fa0/0	10.0.227.12	11	80	10	10000	161	/24	180	10	/24	15	10.0.23.2	1428	1145.5	3
Fa1/0	173.100.6.2	Fa0/0	10.0.227.12	6	40	0	2210	19	/30	180	19	/24	15	10.0.23.2	1040	24.5	14

2. Flow Aging Timers

- Inactive Flow (15 sec is default)
- Long Flow (30 min (1800 sec) is default)
- Flow ends by RST or FIN TCP Flag

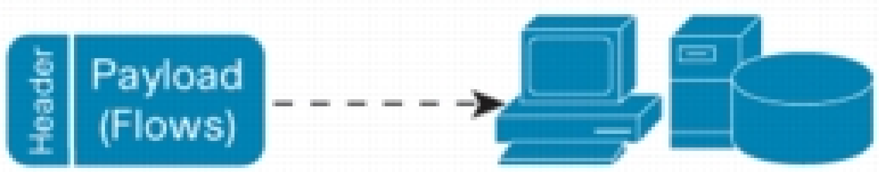
SrcIf	SrcIPadd	DstIf	DstIPadd	Protocol	TOS	Flgs	Pkts	Src Port	Src Msk	Src AS	Dst Port	Dst Msk	Dst AS	NextHop	Bytes/Pkt	Active	Idle
Fa1/0	173.100.21.2	Fa0/0	10.0.227.12	11	80	10	11000	00A2	/24	5	00A2	/24	15	10.0.23.2	1528	1800	4

3. Flows Packaged in Export Packet

Non-Aggregated Flows—Export Version 5 or 9

4. Transport Flows to Reporting Server

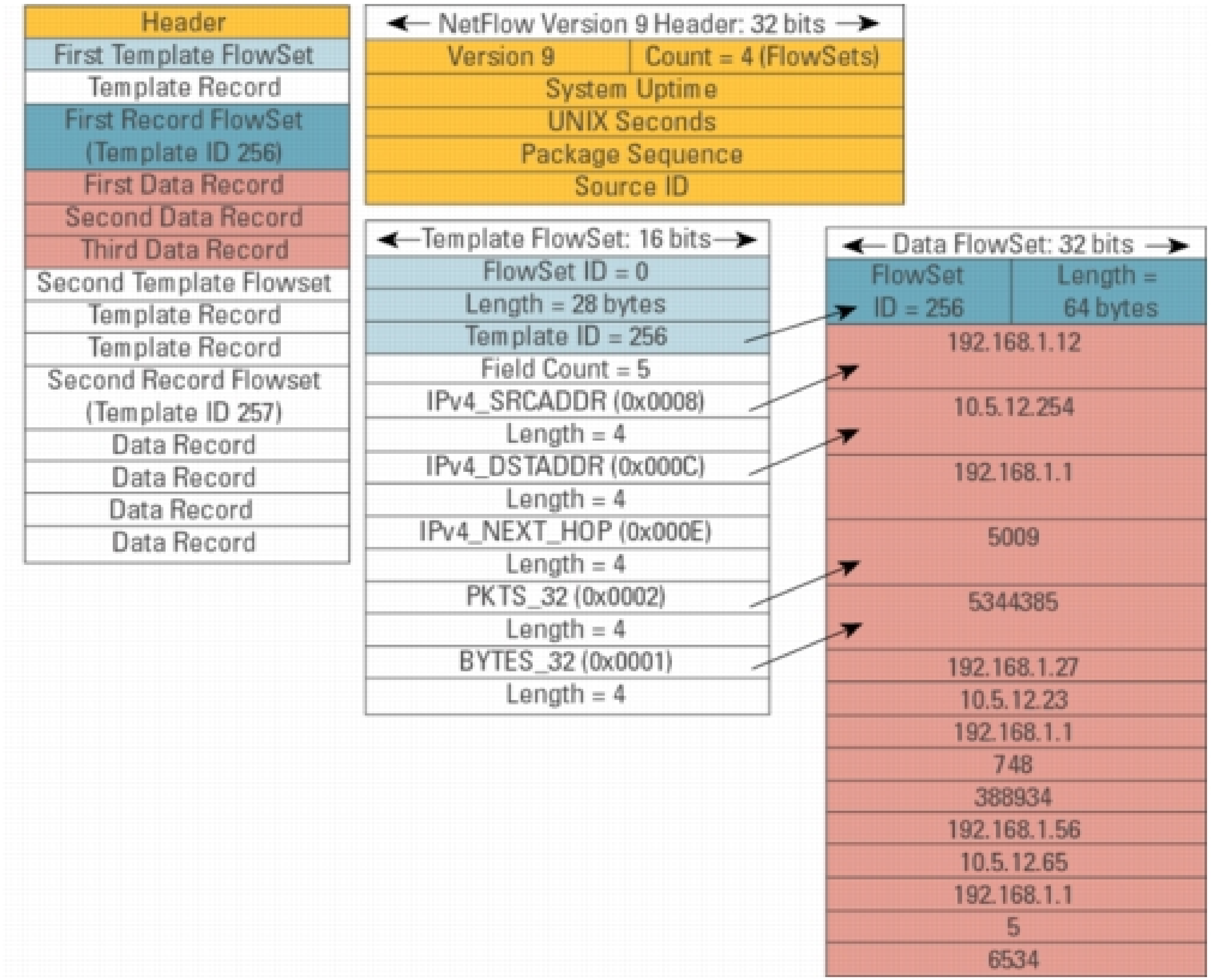
Export Packet



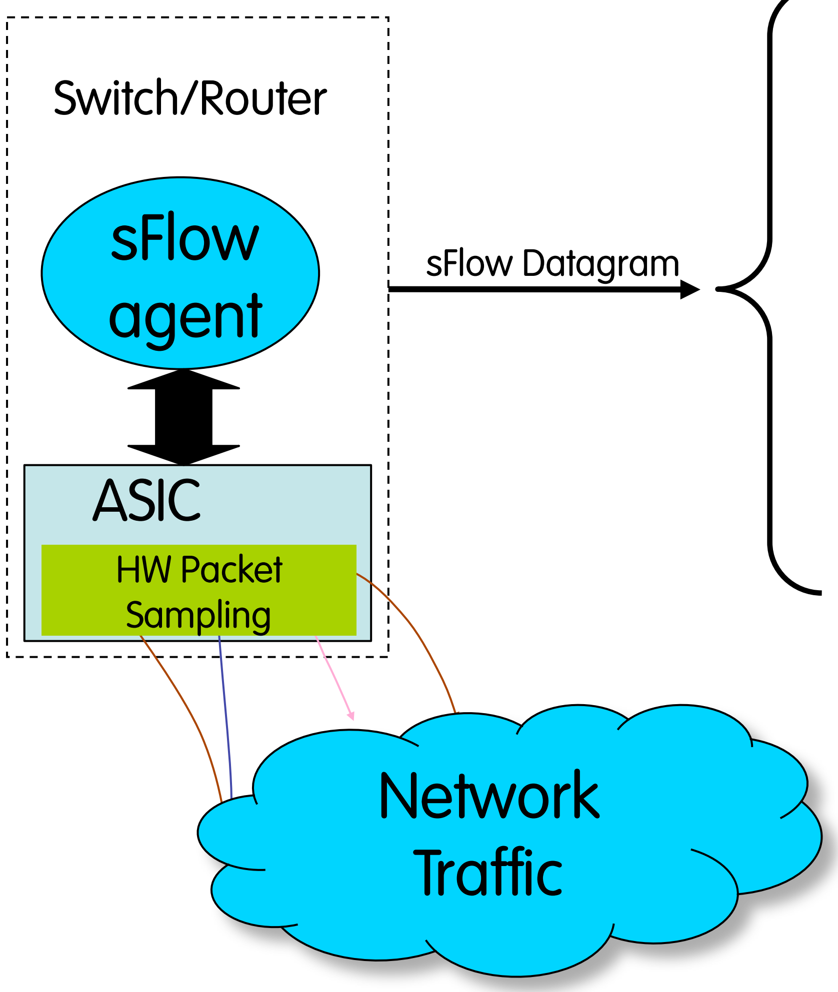
Flow Format: NetFlow v5 vs v9

	v5	v9
Flow Format	Fixed	User Defined
Extensible	No	Yes (Define new FlowSet Fields)
Flow Type	Unidirectional	Bidirectional
Flow Size	48 Bytes (fixed)	It depends on the format
IPv6 Aware	No	IP v4/v6
MPLS/VLAN	No	Yes

Flow Format: NetFlow v9/IPFIX



InMon sFlow



- Packet header (e.g. MAC,IPv4,IPv6,IPX,AppleTalk,TCP,UDP, ICMP)
- Sample process parameters (rate, pool etc.)
- Input/output ports
- Priority (802.1p and TOS)
- VLAN (802.1Q)
- Source/destination prefix
- Next hop address
- Source AS, Source Peer AS
- Destination AS Path
- Communities, local preference
- User IDs (TACACS/RADIUS) for source/destination
- URL associated with source/destination
- Interface statistics (RFC 1573, RFC 2233, and RFC 2358)

% Sampling Error <= 196 * sqrt(1 / number of samples)
[http://www.sflow.org/packetSamplingBasics/]

Integrated Network Monitoring

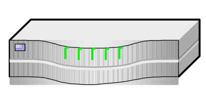
sFlow enabled switches

Core network switches

RMON enabled switches

L2/L3 Switches

NetFlow enabled routers



sFlow

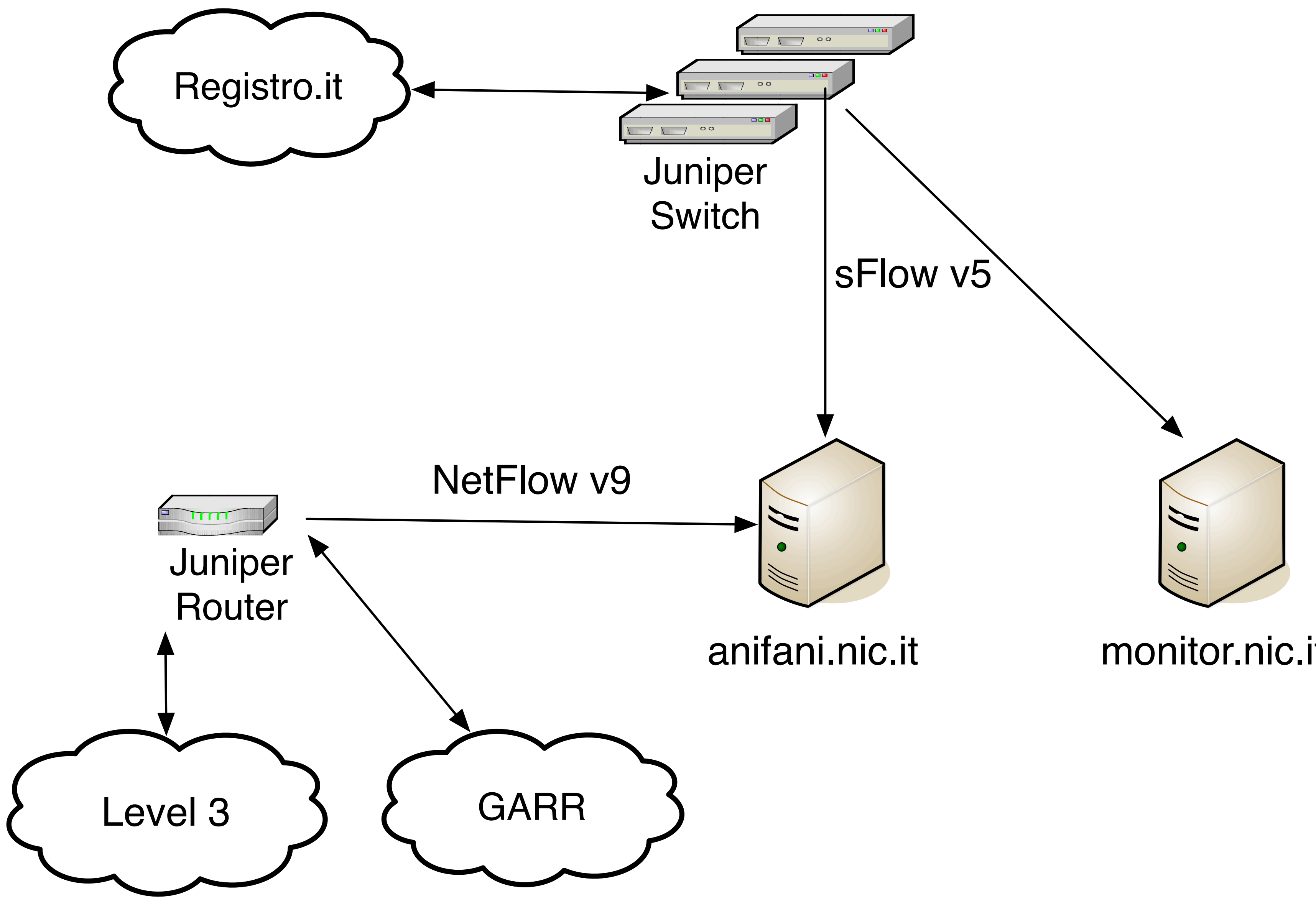
RMON

NetFlow

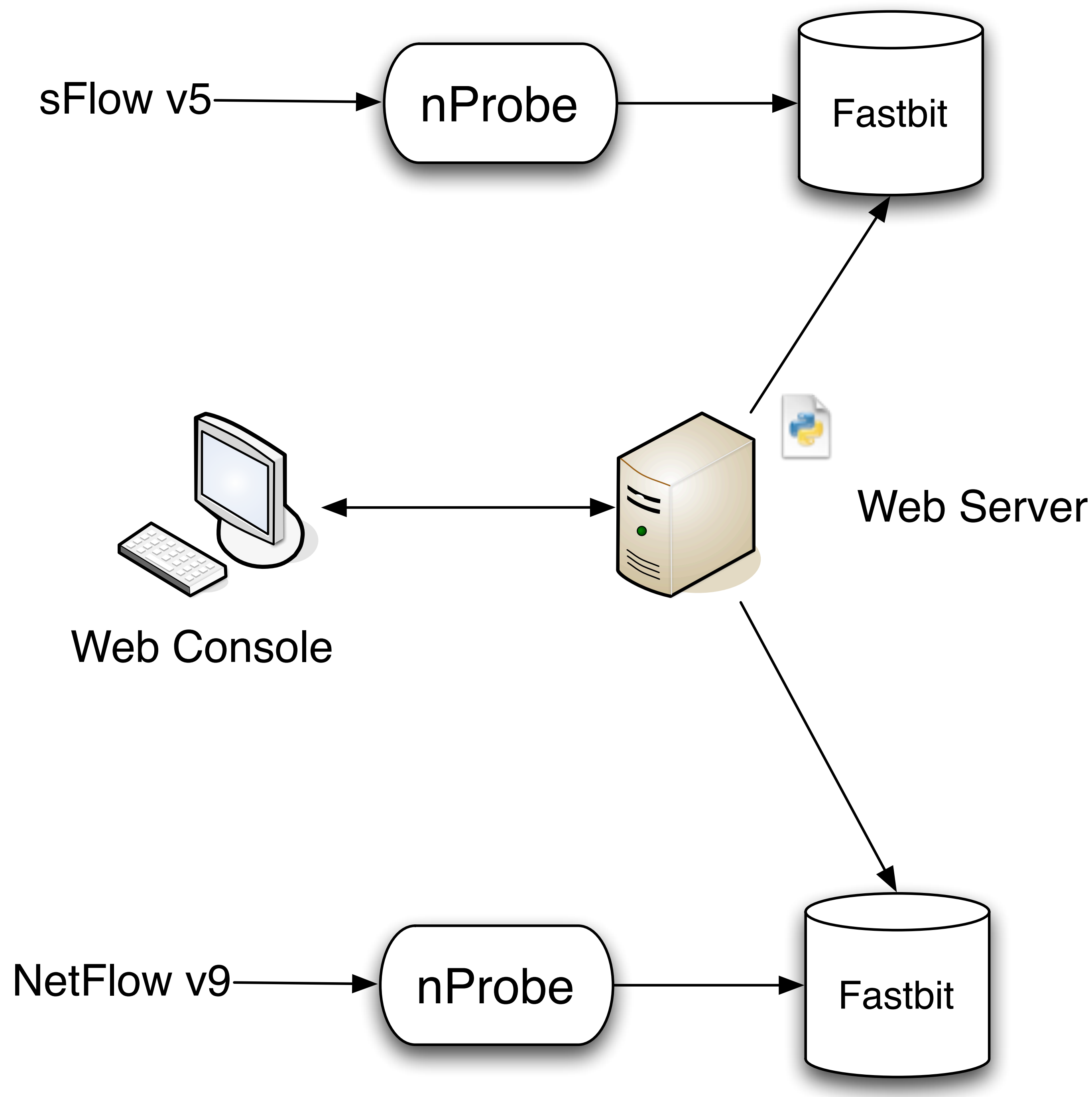
Traffic Analysis & Accounting Solutions

- Network-wide, continuous surveillance
  - 20K+ ports from a single point
- Timely data and alerts
  - Real-time top talkers
  - Site-wide thresholds and alarms
- Consolidated network-wide historical usage data

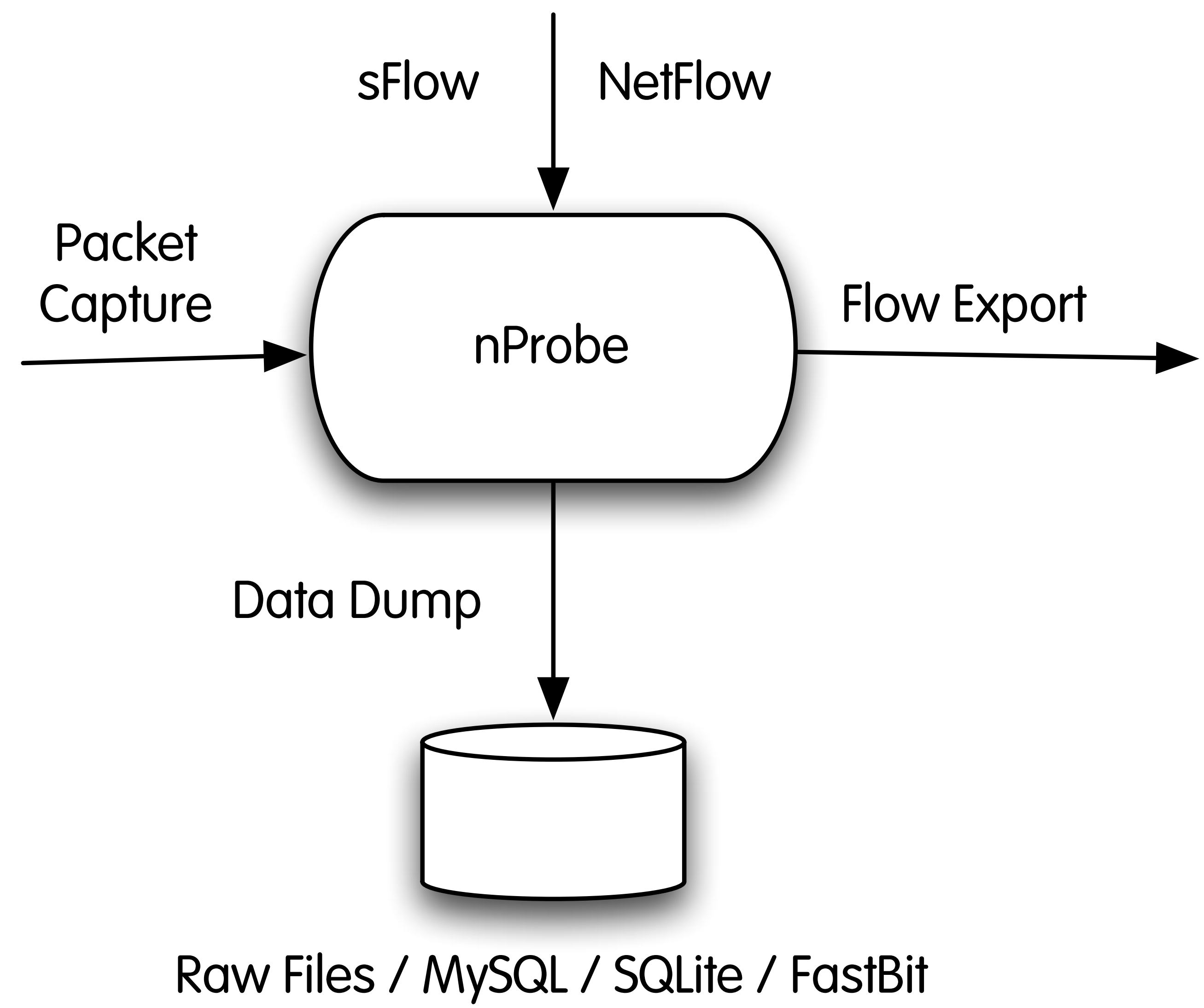
Traffic Collection: A Real Scenario



Heterogeneous Flow Collection



nProbe: sFlow/NF/IPFIX Probe+Collector



Problem Statement [1/2]

- NetFlow and sFlow are the current state-of-the-art standard for network traffic monitoring.
- As the number of generated flows can be quite high, operators often use sampling in order to reduce their number.
- Sampling leads to inaccuracy so it cannot always be used in production networks.
- Thus network operators have to face the problem of collecting and analyzing a large number of flow records.

Problem Statement [2/2]

Where to store collected flows?

- Relational Databases
  - Pros: Expressiveness of SQL for data search.
  - Cons: Sacrifice flow collection speed and query response time.
- Raw Disk Archives
  - Pros: Efficient flow-to-disk collection speed (> 250K flow/s).
  - Cons: Limited query facilities as well search time proportional to the amount of collected data (i.e. no indexing is used).

Towards Column-Oriented Databases [1/3]

- Network flow records are read-only, shouldn't be modified after collection, and several flow fields have very few unique values.

- B-tree/hash indexes used in relational DBs to accelerate queries, encounter performance issues with large tables as:
  - need to be updated whenever a new flow is stored.
  - require a large number of tree-branching operations as they use slow pointer chases in memory and random disk access (seek), thus taking a long time.

- Thus with relational DBs it is not possible to do live flow collection/import as index update will lead to flow loss.

Towards Column-Oriented Databases [2/3]

- A column-oriented database stores its content by column rather than by row. As each column is stored contiguously, compression ratios are generally better than row-stores because consecutive entries in a column are homogeneous to each other.
- Column-stores are more I/O efficient (than row stores) for read-only queries since they only have to read from disk (or from memory) those attributes accessed by a query.
- Indexes that use bit arrays (called bitmaps) answer queries by performing bitwise logical operations on these bitmaps.

Towards Column-Oriented Databases [3/3]

- Bitmap indexes perform extremely well because the intersection between the search results on each value is a simple AND operation over the resulting bitmaps.
- As column data can be individually sorted, bitmap indexes are also very efficient for range queries (e.g. subnet search) as data is contiguous hence disk seek is reduced.
- As column-oriented databases with bitmap indexes provide better performance compared to relational databases, the authors explored their use in the field of flow monitoring.

nProbe + FastBit

- FastBit is not a database but a C++ library that implements efficient bitmap indexing methods.
- Data is represented as tables with rows and columns.
- A large table may be partitioned into many data partitions and each of them is stored on a distinct directory, with each column stored as a separated file in raw binary form.
- nProbe natively integrates FastBit support and it automatically creates the DB schema according to the flow records template.
- Flows are saved in blocks of 4096 records.
- When a partition is fully dumped, columns to be indexed are first sorted then indexed.

Performance Evaluation: Disk Space

MySQL	No/With Indexes	1.9 / 4.2
FastBit	Daily Partition (no/with Indexes)	1.9 / 3.4
	Hourly Partition (no/with Indexes)	1.9 / 3.9
nfdump	No indexes	1.9

Results are in GB

Performance Evaluation: Query Time [1/2]

nProbe+FastBit vs MySQL

Query	MySQL		nProbe + FastBit Daily Partitions		nProbe + FastBit Hourly Partitions	
	No Index	With Indexes	No Cache	Cached	No Cache	Cached
Q1	20.8	22.6	12.8	5.86	10	5.6
Q2	23.4	69	0.3	0.29	1.5	0.5
Q3	796	971	17.6	14.6	32.9	12.5
Q4	1033	1341	62	57.2	55.7	48.2
Q5	1754	2257	44.5	28.1	47.3	30.7

Results are in seconds

Performance Evaluation: Query Time [2/2]

nProbe+FastBit vs nfdump

nProbe+FastBit	45 sec
nfdump	1500 sec

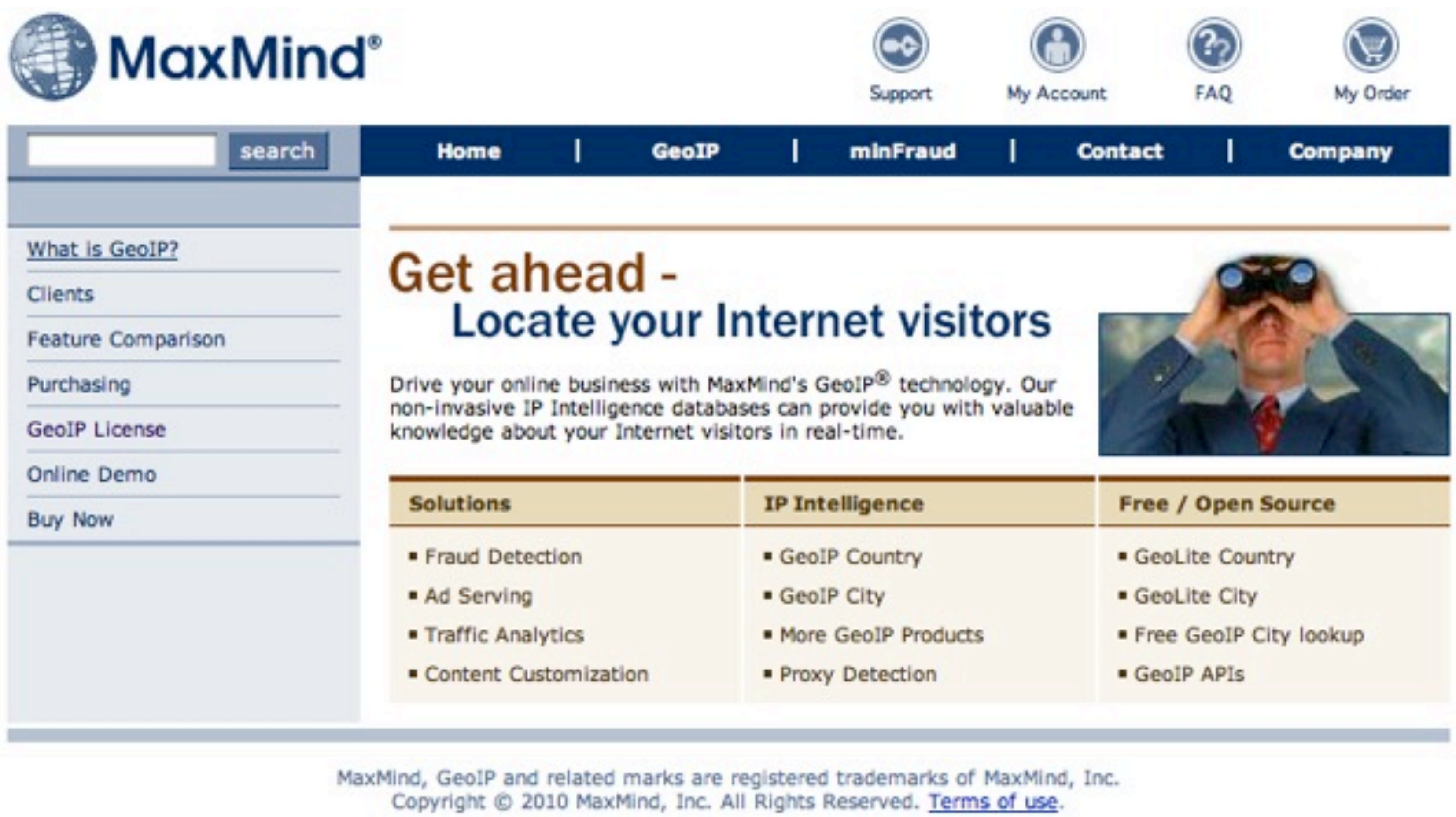
SELECT IPV4_SRC_ADDR, L4_SRC_PORT, IPV4_DST_ADDR, L4_DST_PORT, PROTOCOL FROM NETFLOW WHERE IPV4_SRC_ADDR=X OR IPV4_DST_ADDR=X

worth 19 GB of data (14 hours of collected flows)

nfdump query time = (time to sequentially read the raw data) + (record filtering time)

Host Geolocation [1/3]

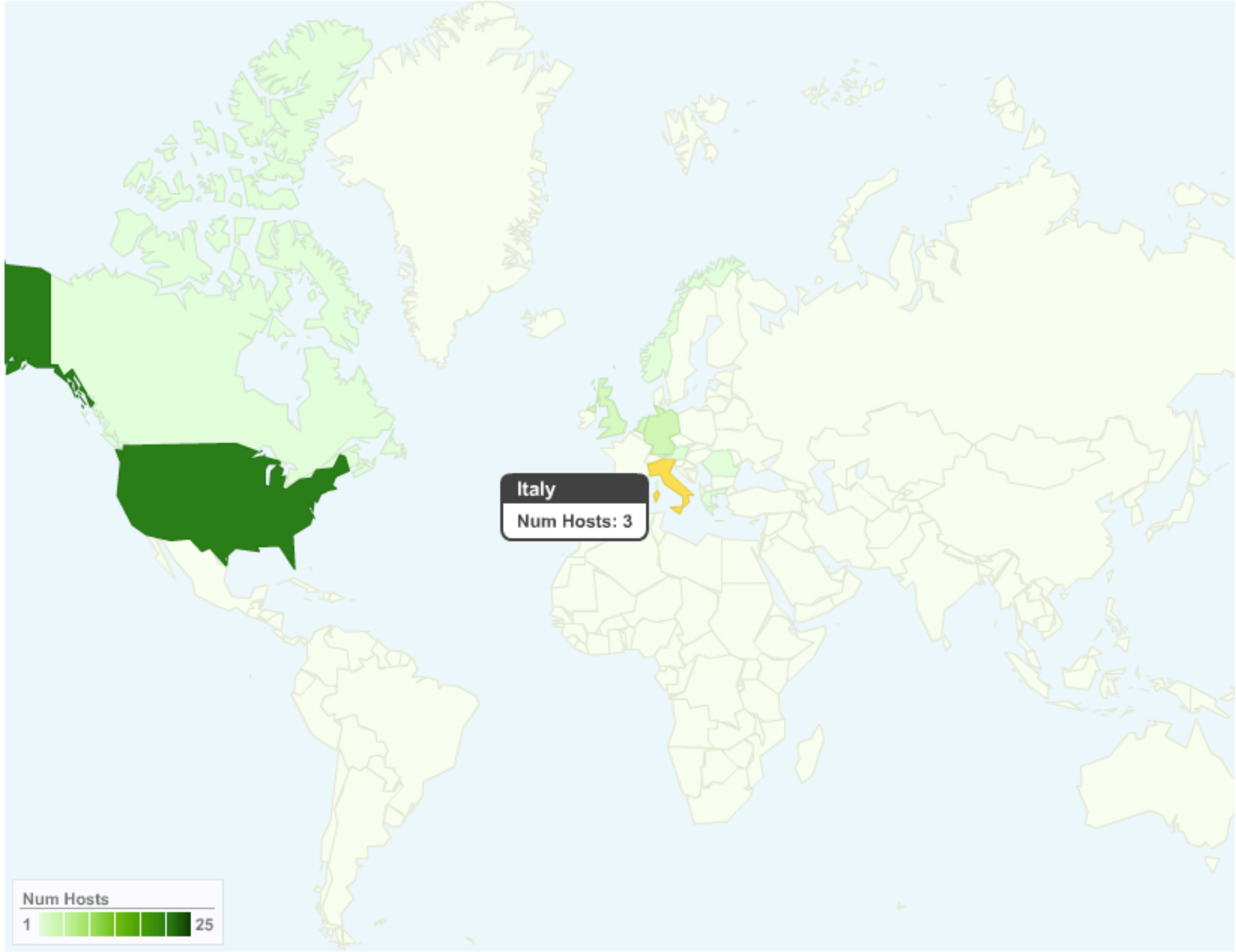
- Host geolocation is a known problem (vd http://en.wikipedia.org/wiki/Geoip)
- Need to handle thousand flows/sec (no inline internet query)
- Requirements: IP -> Location e IP -> ASN



Host Geolocation [2/3]

- Interactive Flash™ world map, that displays hosts distribution by country and by cities of a selected country
- nProbe + GeoIP + Python + Google Visualization. The script
  - Cycles through all the hosts seen by ntop
  - Gets their GeoIP info
  - Counts them based on their location.
- Google GeoMap and Visualization Table
- Ajax/JSON communications with web server for updated data

Host Geolocation [3/3]



How to Add Geolocation Data [1/3]

- Routers are unable to export any geolocation information.
- NetFlow/IPFIX flows do not contain any information about geolocation into standard flow formats.
- Solution:
  - Let the collector add geolocation information to flows received by routers
  - Let the softprobe export this information to collectors.

How to Add Geolocation Data [2/3]

- nProbe takes advantage of GeoIP library (GPL) to
  - Add geolocation information to flows
  - Map IP addresses to ASN (Autonomous System Numbers) for adding ASN awareness.

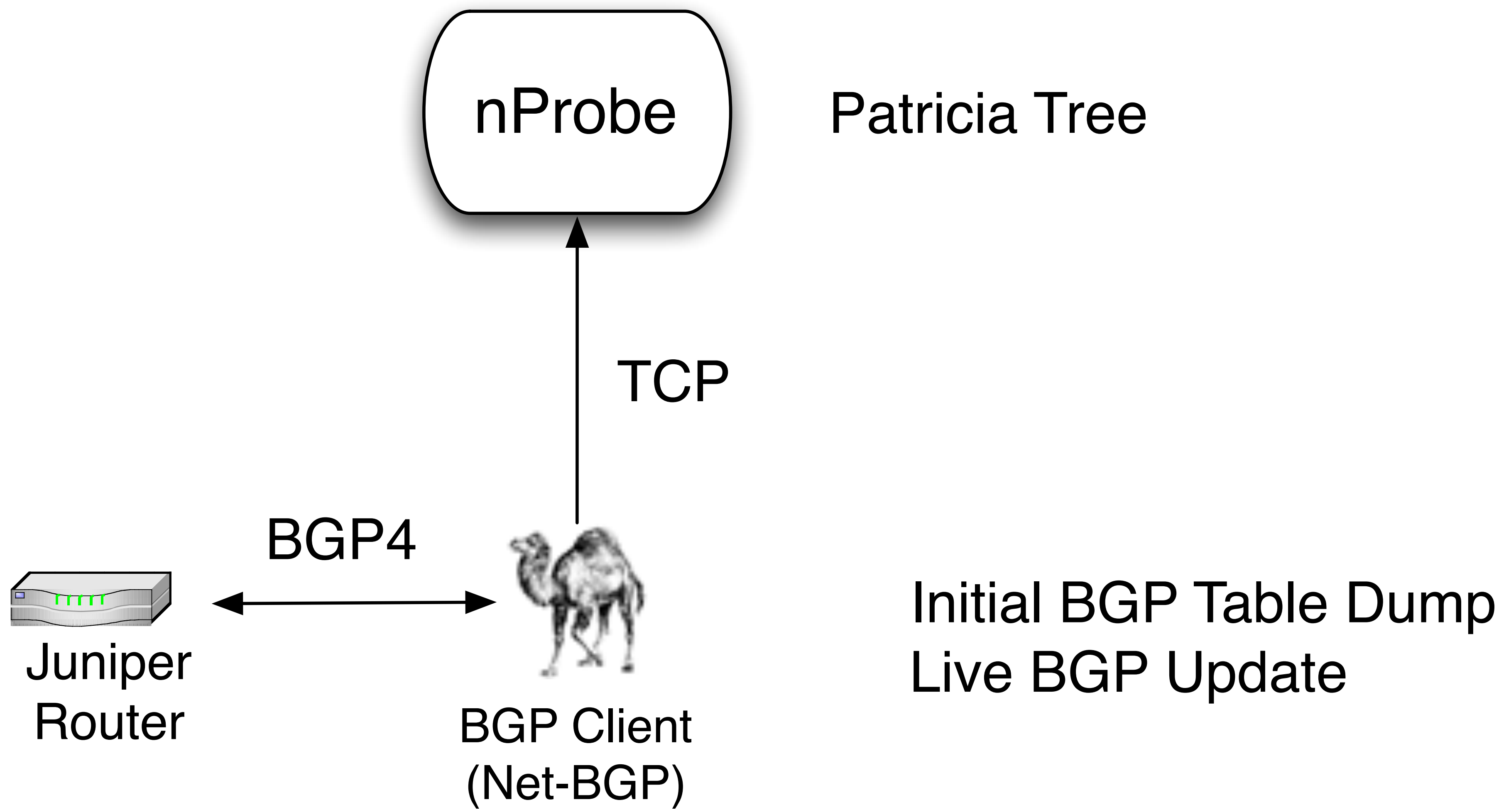
  - GeoIPASNum.dat (ASN)
  - GeoLiteCity.dat (GeoLocation)

How to Add Geolocation Data [3/3]

```
  if(host->ipVersion == 4)
    return(GeoIP_record_by_ipnum(readOnlyGlobals.geo_ip_city_db, host->ipType.ipv4));
#ifdef INET6
  else if(host->ipVersion == 6)
    return(GeoIP_record_by_ipnum_v6(readOnlyGlobals.geo_ip_city_db, host->ipType.ipv6));
#endif
```

```
    char *rsp = NULL;
    u_int32_t as;

    if(ip.ipVersion == 4)
      rsp = GeoIP_name_by_ipnum(readOnlyGlobals.geo_ip_asn_db, ip.ipType.ipv4);
    else {
#ifdef INET6
      rsp = GeoIP_name_by_ipnum_v6(readOnlyGlobals.geo_ip_asn_db, ip.ipType.ipv6);
#endif
    }

    as = rsp ? atoi(&rsp[2]) : 0;
    free(rsp);
```

BGP Data Integration [1/2]



BGP Data Integration [2/2]

```
# Constructor
$update = Net::BGP::Update->new(
    NLRI            => [ qw( 10/8 172.168/16 ) ],
    Withdraw        => [ qw( 192.168.1/24 172.10/16 192.168.2.1/32 ) ],
    # For Net::BGP::NLRI
    Aggregator      => [ 64512, '10.0.0.1' ],
    AsPath          => [ 64512, 64513, 64514 ],
    AtomicAggregate => 1,
    Communities     => [ qw( 64512:10000 64512:10001 ) ],
    LocalPref       => 100,
    MED             => 200,
    NextHop         => '10.0.0.1',
    Origin          => INCOMPLETE,
);
```

What if you have no BGP Router? [1/3]

RIPE NCC | LIR Portal | RIPE | About RIPE NCC | Contact | Search | Sitemap

RIPE NCC

Routing Information Service

Quick Links | GO

you are here: home -> RIPE NCC Projects -> RIS

RIS:

- RIS Home Page
- RIS Raw Data
- Documentation
- Analysis using RIS
- Contact Us
- Send Feedback

RIS Raw Data

This page links to the raw data collected by the RRCs using Quagga routing software, stored in MRT format. This format is described in an IETF draft. These files can be read using libbgpdump, a library written in C by Dan Ardelean, currently maintained by the RIPE NCC. A Python library also exists, PyBGPdump which provides access to MRT files via Python.

Please note that the raw data format recently changed to include support for 4-byte ASNs.

For more information see: http://www.ripe.net/projects/ris/docs/asn.html

Two sets of files are available for each of the RRC's:

- All BGP packets, created with the Zebra command *"dump bgp all ..."*. The filenames start with *updates* and are created every five minutes.
- The entire BGP routing table, created with the Zebra command *"dump bgp routes-mrt ..."*. These files are created every eight hours, the filenames start with *bview*.

BGP Timer settings since 23 November 2006:

- Keepalives:
  - 60 seconds
- Holddown:
  - 180 seconds

BGP Timer settings between 17 October 2002 and 23 November 2006:

- Keepalives:
  - 0 (disabled)
- Holddown:
  - 0 (disabled)

BGP Timer settings before 17 October 2002:

- Keepalives:
  - 60 seconds
- Holddown:
  - 180 seconds

Please note that this machine is connected to the Internet over a link that is loaded during Amsterdam office hours. If you want to download a large number of files, please do so at a spread-out rate, preferably at a time *outside Amsterdam office hours* (08:00 - 18:00 UTC).

Click on a collector to get a list of available files. Files are grouped collector by collector, then month by month.

- rrc00.ripe.net at RIPE NCC, Amsterdam, collects default free routing updates from peers. From October 1999.
- rrc01.ripe.net at LINX, London. Collects route updates announced by LINX members. From July 2000.
- rrc02.ripe.net at SFINX, Paris. Collected route updates announced by SFINX members from March 2001 until October 2008.

RELATED TOPICS

- DNS Monitoring
- DISI
- K-root Name Server
- RRCC
- TTM

What if you have no BGP Router? [2/3]

Index of /rrc10/2010.06

Name	Last modified	Size	Description
Parent Directory		-	
bview.20100601.0759.gz	01-Jun-2010 08:00	8.6M	
bview.20100601.1559.gz	01-Jun-2010 16:00	8.6M	
bview.20100601.2359.gz	02-Jun-2010 00:00	8.6M	
bview.20100602.0759.gz	02-Jun-2010 08:00	8.7M	
bview.20100602.1559.gz	02-Jun-2010 16:00	8.7M	
bview.20100602.2359.gz	03-Jun-2010 00:00	8.7M	
bview.20100603.0759.gz	03-Jun-2010 08:00	8.7M	
bview.20100603.1559.gz	03-Jun-2010 16:00	8.7M	
bview.20100603.2359.gz	04-Jun-2010 00:00	8.6M	
bview.20100604.0759.gz	04-Jun-2010 08:00	8.6M	
bview.20100604.1559.gz	04-Jun-2010 16:00	8.6M	
bview.20100604.2359.gz	05-Jun-2010 00:00	8.6M	

bview.20100613.2359.gz	14-Jun-2010 00:00	8.6M
bview.20100614.0759.gz	14-Jun-2010 08:00	8.7M
bview.20100614.1559.gz	14-Jun-2010 16:00	8.7M
updates.20100601.0000.gz	01-Jun-2010 00:04	15K
updates.20100601.0005.gz	01-Jun-2010 00:10	14K
updates.20100601.0010.gz	01-Jun-2010 00:15	19K
updates.20100601.0015.gz	01-Jun-2010 00:20	17K
updates.20100601.0020.gz	01-Jun-2010 00:25	12K
updates.20100601.0025.gz	01-Jun-2010 00:29	13K
updates.20100601.0030.gz	01-Jun-2010 00:35	17K
updates.20100601.0035.gz	01-Jun-2010 00:40	19K
updates.20100601.0040.gz	01-Jun-2010 00:45	11K

What if you have no BGP Router? [3/3]

- libbgpdump can be used to read BGP dump and updates.
- Periodically poll the RIPE RIS directory searching for full dumps or updates.

```
TIME: 06/15/10 15:59:58
TYPE: TABLE_DUMP_V2/IPV4_UNICAST
PREFIX: 12.51.167.0/24
SEQUENCE: 1321
FROM: 217.29.66.65 AS12779
ORIGINATED: 06/15/10 13:20:28
ORIGIN: IGP
ASPATH: 12779 1239 3356 19343 19343 19343 19343
NEXT_HOP: 217.29.66.65
COMMUNITY: 12779:1239 12779:65098
```

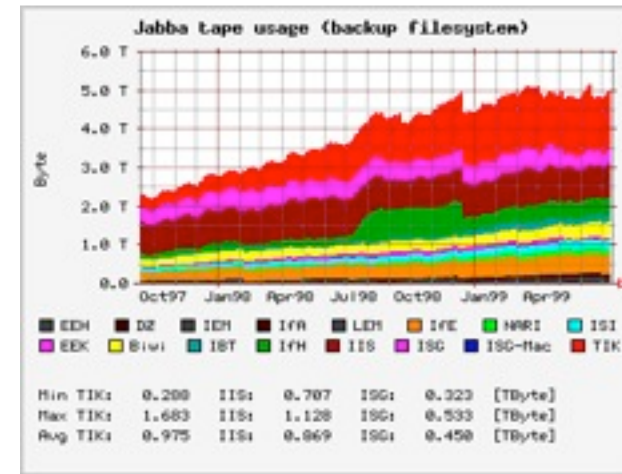
- Connect to the probe and refresh the routes according to the values being read.
- NOTE: always use the BGP dumps for a location near to you in order to have your view of the Internet.

Implementing a Web 2.0 GUI

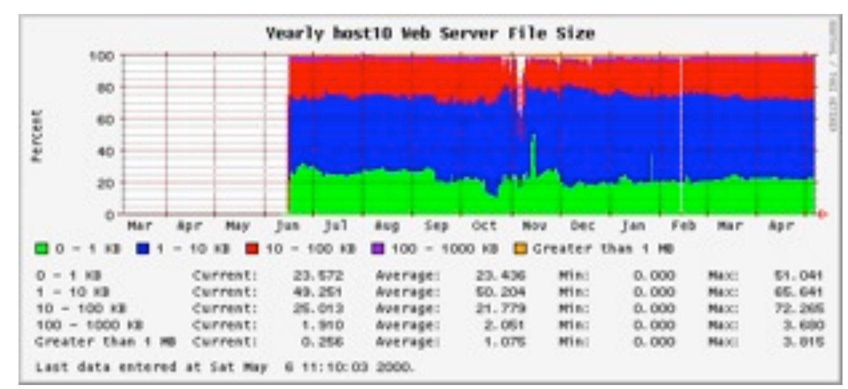
- Web server: Lighttpd (easy and fast), avoid Apache.
- Ajax: use established frameworks such as jQuery or Prototype.
- Implement class libraries used to read your monitoring data. Python is used for speed, ease of use and script compilation.
- Use templates (e.g. Mako) for generating (XML-free) pages.
- Web frameworks are perhaps easier to use, but you will be bound to them forever (pros and cons).

Storing Historical Data [1/2]

- RRD is the de-facto standard for permanently storing numerical data.




```
$rrd = "$dataDir/$agent-$ifIndex.rrd";
if(! -e $rrd) {
    RRDs::create ($rrd, "--start",$now-1, "--step",20,
      "DS:bytesIn:COUNTER:120:0:10000000",
      "DS:bytesOut:COUNTER:120:0:10000000",
      "RRA:AVERAGE:0.5:3:288");
    $ERROR = RRDs::error;
    die "$0: unable to create `$rrd': $ERROR\n" if $ERROR;
}
    RRDs::update $rrd, "$now:$ifInOctets:$ifOutOctets";
   if ($ERROR = RRDs::error) {
    die "$0: unable to update `$rrd': $ERROR\n";
}
```



Storing Historical Data [2/2]

- RRD has several limitations:
  - Only one (quantity one) numerical data can be stored at each time interval (e.g. # of bytes received).
  - You must know 'in advance' what you want to store. For instance you can't store anything like 'the name and amount of traffic sent by the top host': the top host changes overtime, so you need an rrd per top host and this is not what you want.
  - Sets or lists of data (e.g. top protocols with bytes on interval X) cannot be stored in RRD.

Beyond RRD

- Requirements:
  - Store network values are tuples (list of <name>:<value>, where <value> can also be a list).
  - Ability to aggregate tuples using a user-defined function (i.e. not just max/min/average).
  - Manipulate values as RRD does: create, update, last, export, fetch and graph.
  - Graph: images are not enough as we have tuples (not just one value) and also because the user must be able to interact with data, not just look at it.

pSWTDB [1/4]

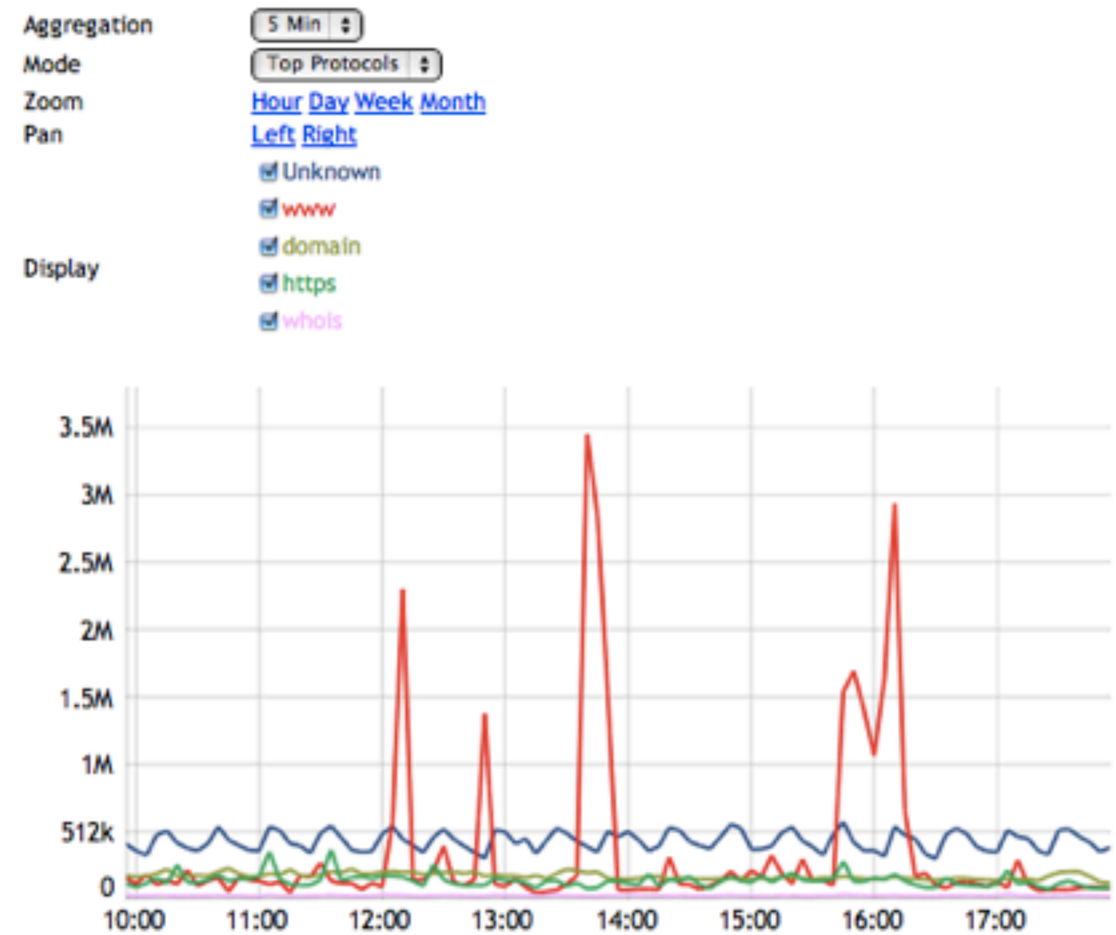
- pSWTDB (Sliding Window Tuple DB).
- python class used to store tuples on disk using data serialization (called pickle on python).
  - Pros:
    - native in python
    - portable across datatypes (i.e. no need to define the type)
  - Cons:
    - Slow as RRD (deserialize/update/serialize at each update)
- Same principle of RRD with the exception that here we use <u>tuples</u> and <u>not</u> numerical values.

pSWTDB [2/4]

- It comes with aggregation functions such as:
  - Each time interval has a list of (key, value).
  - Sum values with same key.
  - Sort values
  - Discard values ranking after position X (e.g. take the top/bottom X values).
- Examples
  - Top X protocols (list of <proto>:<value>)
  - Top X hosts (list of <host>:(<proto>:<value>,...))

pSWTDB [3/4]

- Data are plotted using SVG/JavaScript.
- Users can interact with data (pan, zoom, move).
- Multiple criteria can be plotted at the same time (e.g. top X hosts and Y protocols).
- Clicking on data can be used to trigger GUI updates



Query time: 2.35686707497 sec

IPV4_SRC_ADDR	IPV4_DST_ADDR	SUM(IN_BYTES)	SUM(IN_PKTS)	COUNT(*)
192.12.192.5	89.149.209.158	153585348	309352	1
89.149.209.158	192.12.192.5	122275527	310295	1
192.12.193.21	81.23.251.109	43858112	0	1
78.46.47.139	192.12.192.242	25821168	166459	1
78.46.47.139	192.12.192.229	23806373	154724	1
192.12.192.242	78.46.47.139	23759119	70706	1
192.12.193.12	146.48.98.155	21837485	0	1
192.12.192.229	78.46.47.139	21767757	69770	1
192.12.193.56	74.125.170.208	19601351	0	1
192.12.192.242	178.63.254.125	18829579	41672	2

pSWTDB [4/4]

```
deri@MacLuca.local 233> cat pcreate.py
#!/usr/bin/python

import pSWTDB

t = pSWTDB.pSWTDB('ptest.pkl')
# Hearbeat is 5 min
t.create(300)

# Keep 60 samples, one per minute
t.add_base_aggregation('1min', 60, 60)

# Keep 50 samples, each aggregating 5 samples
# of the base aggregation
t.add_aggregation('5min', 5, 50, pSWTDB.sum, '')

# Keep 60 samples, each aggregating 24 samples
# of the 5min aggregation
t.add_aggregation('hour', 24, 60, pSWTDB.sum, '5min')

# Keep 30 samples, each aggregating 12 samples
# of the hour aggregation
t.add_aggregation('day', 12, 30, pSWTDB.sum, 'hour')
```

```
deri@MacLuca.local 234> cat pupdate.py
#!/usr/bin/python

import pSWTDB

t = pSWTDB.pSWTDB('IT.pkl')
t.update('now',
         { 'keys' : ['APPL_PROTOCOL'],
           'values' : ['SUM_PKTS'],
           'data' : {
            'das' : ( 4522726 ),
            'domain' : ( 1706286 ),
            'whois' : ( 62838 ),
            'www' : ( 28699 ),
            'smtp' : ( 16149 ),
            'https' : ( 10892 ),
            'Unknown' : ( 4934 ),
            }
           })
```

```
deri@MacLuca.local 238> cat pfetch.py
#!/usr/bin/python

import pSWTDB
import pprint

t = pSWTDB.pSWTDB('IT.pkl')
ret = t.fetch('', 'now-1h', 'now')
print t.plot(ret)
```

Traffic Data Analysis [1/4]

Flow collection and storage
in FastBit Archive Format
(5 min timeframe partition)

↓

Column data sort and
data indexing

↓

Partition data analysis

↓

Metrics persistent storage

```
deri@anifani 203> ls -lL
total 24
4 -rwxr-xr-x 1 deri deri 1377 Mar 27 12:06 cities.py*
4 -rwxr-xr-x 1 deri deri  950 Mar 23 23:21 flows.py*
4 -rwxr-xr-x 1 deri deri 2162 May 22 13:49 top_n_flows_countries.py*
4 -rwxr-xr-x 1 deri deri 2106 Mar 25 15:48 top_n_l7_protocols.py*
8 -rwxr-xr-x 1 deri deri 4565 May 22 14:32 top_n_proto_countries.py*
deri@anifani 204> pwd
/home/deri/nProbe/fastbit/python/partition_scripts
```

Traffic Data Analysis [2/4]

```
deri@anifani 208> ls -l
total 24
16 drwxr-xr-x 3 root root 16384 May 25 08:21 aggregations/
 4 drwxr-xr-x 4 deri deri  4096 Mar 27 12:07 queries/
 4 drwxr-xr-x 6 deri deri  4096 Mar 18 19:37 rrd/
deri@anifani 209> ls -l *
aggregations:
total 34000

 20 -rw-r--r-- 1 root root  18768 May 25 16:12 A1.pkl
164 -rw-r--r-- 1 root root 167641 May 25 16:12 A2.pkl
152 -rw-r--r-- 1 root root 154778 May 25 16:12 AD.pkl
216 -rw-r--r-- 1 root root 219872 May 25 16:13 AE.pkl
148 -rw-r--r-- 1 root root 148012 May 25 16:13 AF.pkl
152 -rw-r--r-- 1 root root 152841 May 25 16:13 AG.pkl
100 -rw-r--r-- 1 root root 100615 May 25 16:12 AI.pkl
...
152 -rw-r--r-- 1 root root 154259 May 25 16:13 YE.pkl
 12 -rw-r--r-- 1 root root  10101 May 25 15:13 YT.pkl
200 -rw-r--r-- 1 root root 201469 May 25 16:12 ZA.pkl
148 -rw-r--r-- 1 root root 151246 May 25 16:12 ZM.pkl
156 -rw-r--r-- 1 root root 156071 May 25 16:12 ZW.pkl
308 -rw-r--r-- 1 root root 315311 May 25 16:13 all_countries.pkl
  4 -rw-r--r-- 1 root root    791 May 15 23:55 ne.pkl
 24 drwxr-xr-x 2 root root  20480 May 22 13:57 top_hosts/


queries:
total 8
4 drwxr-xr-x 7 deri deri 4096 May  1 00:05 2010/

rrd:
total 144
48 -rw-r--r--   1 root root 47128 May 25 16:13 bits.rrd
12 drwxr-xr-x   2 root root 12288 May  6 02:06 bytes/
12 drwxr-xr-x 475 root root 12288 May 16 19:26 country/
12 drwxr-xr-x   2 root root 12288 May 24 23:36 flows/
48 -rw-r--r--   1 root root 47128 May 25 16:13 flows.rrd
12 drwxr-xr-x   2 root root 12288 May 12 20:42 pkts/
```

Traffic Data Analysis [3/4]

```
rrd/country/CH/mandelspawn.rrd
rrd/country/CH/gds_db.rrd
rrd/country/CH/dircproxy.rrd
rrd/country/CH/rmtcfg.rrd
rrd/country/CH/ssh.rrd
rrd/country/CH/isisd.rrd
rrd/country/CH/cfinger.rrd
rrd/country/CH/gris.rrd
rrd/country/CH/daap.rrd
rrd/country/CH/x11.rrd
rrd/country/CH/postgresql.rrd
rrd/country/CH/amanda.rrd
rrd/country/CH/zephyr-hm.rrd
rrd/country/CH/gsigatekeeper.rrd
rrd/country/CH/fax.rrd
rrd/country/CH/netbios-ssn.rrd
rrd/country/CH/afs3-fileserver.rrd
rrd/country/CH/cvspserver.rrd
rrd/country/CH/ospf6d.rrd
rrd/country/CH/bpcd.rrd
rrd/country/CH/proofd.rrd
rrd/country/CH/afs3-errors.rrd
rrd/country/CH/ggz.rrd
rrd/country/CH/tproxy.rrd
rrd/country/CH/cfengine.rrd
rrd/country/CH/x11-6.rrd
rrd/country/CH/msp.rrd
rrd/country/CH/rje.rrd
rrd/country/CH/sane-port.rrd
rrd/country/CH/smtp.rrd
```

```
deri@anifani 213> ls queries/2010/05/25/16/00/
total 1172
1164 cities.pkl      8 top_n_l7_protocols.pkl
```

Traffic Data Analysis [4/4]

```
deri@anifani 215> ~/nProbe/fastbit/python/dump.py cities.pkl |m
{'city': [['SRC_COUNTRY',
           'SRC_CITY',
           'SRC_LATITUDE',
           'SRC_LONGITUDE',
           'SRC_REGION',
           'COUNT'],
          ['', '', '', '', '', 15079],
          ['IT', 'Rome', 41.899999999999999, 12.4832, 'Lazio', 1427],
          ['KR',
           'Seoul',
           37.566400000000002,
           126.9997,
           "Seoul-t'ukpyolsi",
           1250],
          ['RU',
           'Moscow',
           55.752200000000002,
           37.615600000000001,
           'Moscow City',
           1243],
          ['IT',
           'Milan',
           45.466700000000003,
           9.1999999999999993,
           'Lombardia',
           936],
```

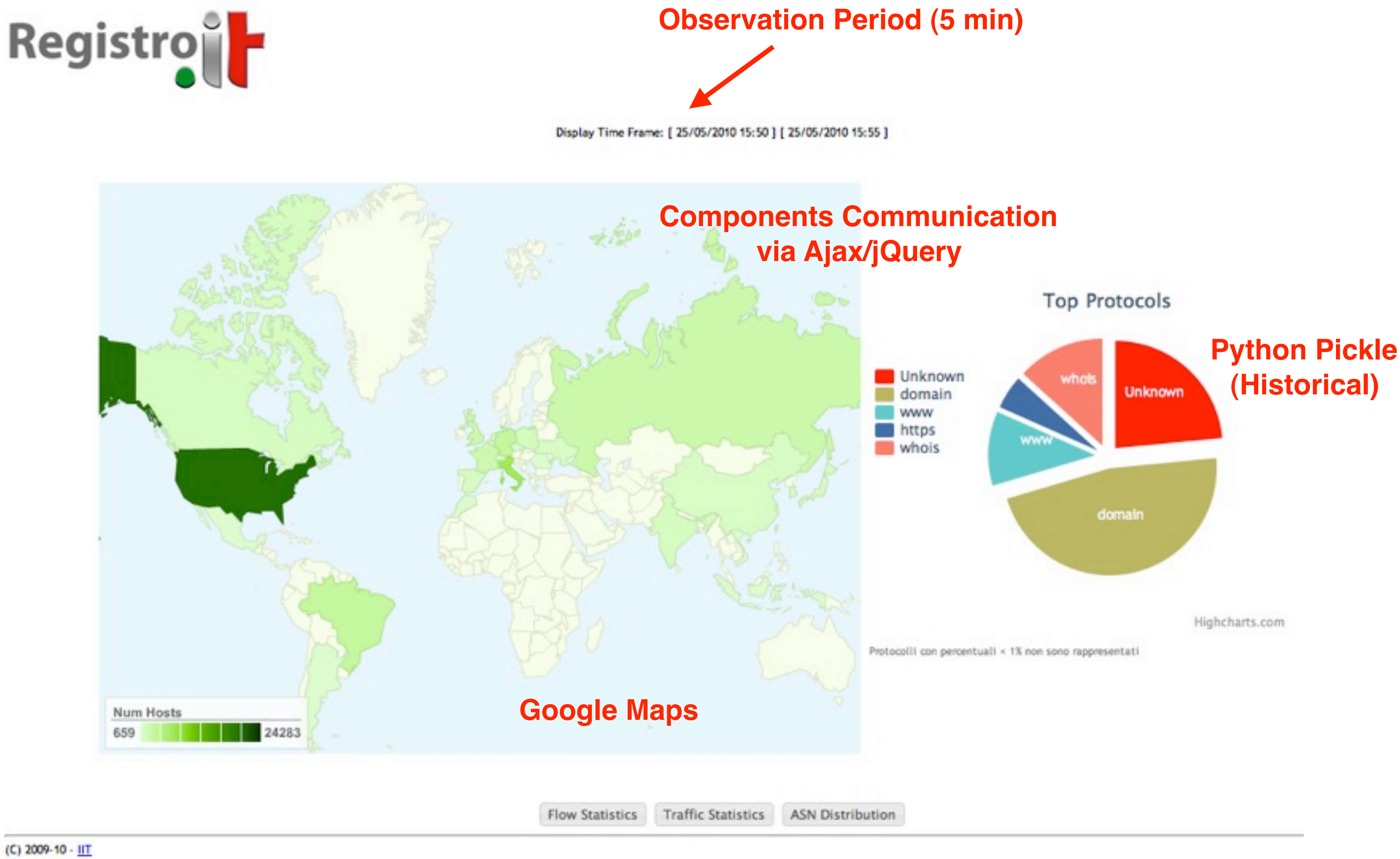
Remote Probe Deployment [1/2]

- In order to monitor a distributed network it is often necessary to deploy remote probes.
- Exporting flows towards a central location is not always possible:
  - Limited bandwidth available.
  - Need to have a separate/secure network/tunnel as flows contain sensitive data.
  - Interference with other network activities.
  - Export of raw flows is much more costly than exporting the metrics we're interested in.

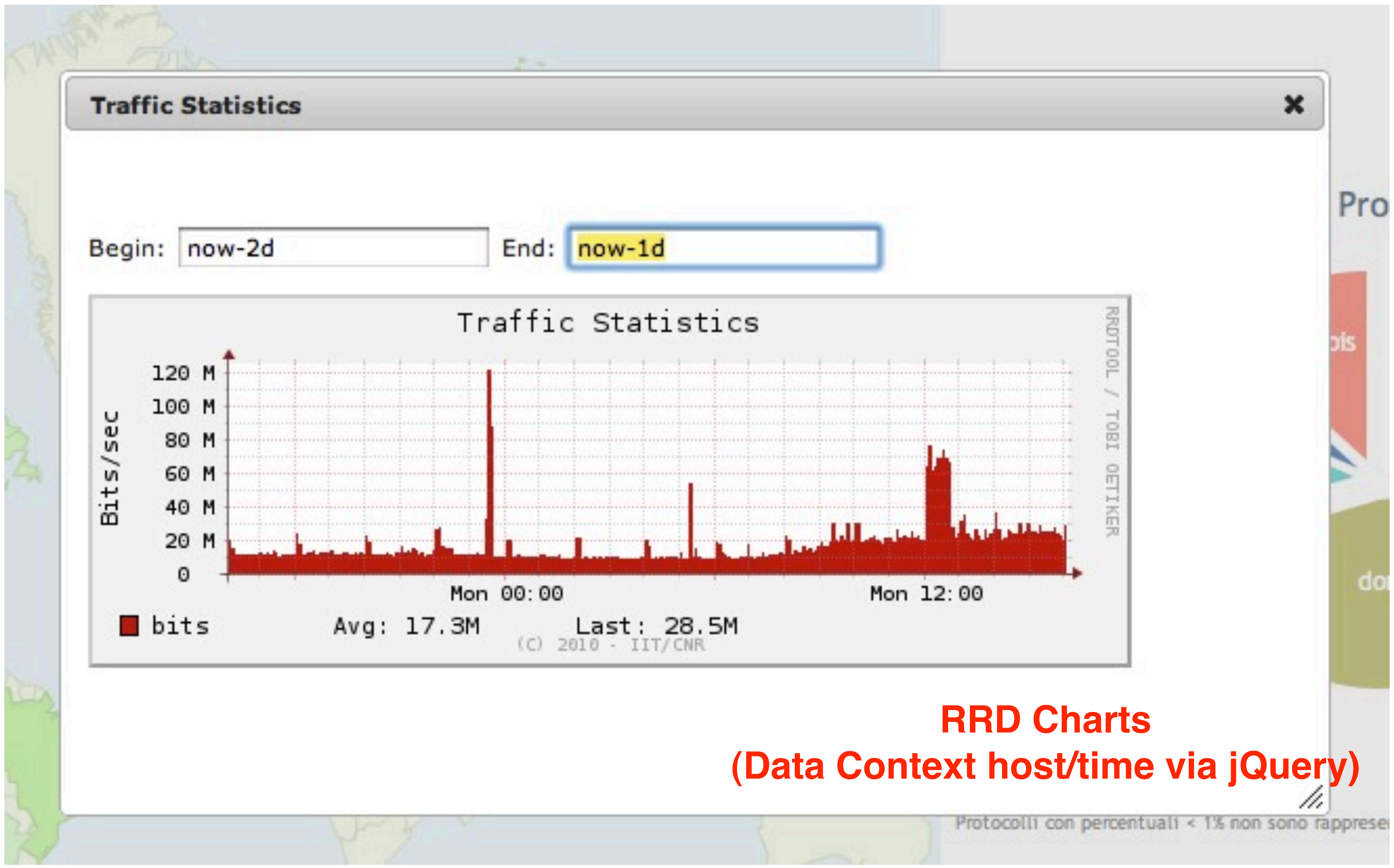
Remote Probe Deployment [2/2]

- Exporting data on off-peak times is not an option:
  - We would introduce latency in data consumption.
  - The amount of data to transfer is not significantly reduced (zip flows) with respect to live data export.
  - Unable to use the system for near-realtime analysis and alarm generation.
- Better solution
  - Create a web service for querying data remotely in realtime
  - Export aggregated metrics (e.g. .pkl files)

Web Interface: Internals [1/3]



Web Interface: Internals [2/3]



RRD Charts
(Data Context host/time via jQuery)

Web Interface: Internals [2/3]

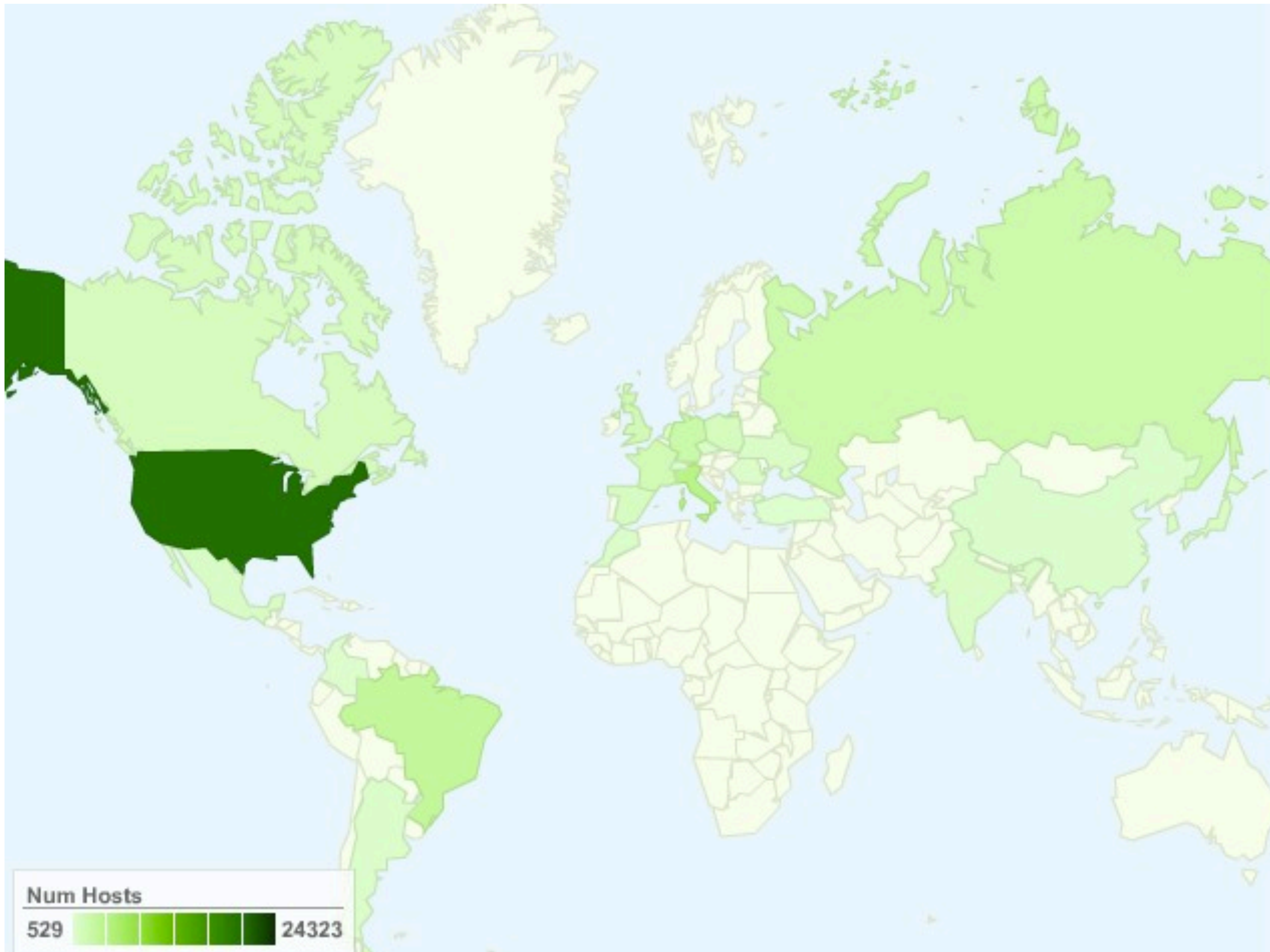
Partition /home/deri/fastbit/netflow/2010/05/25/16/00

Protocol All Protocols

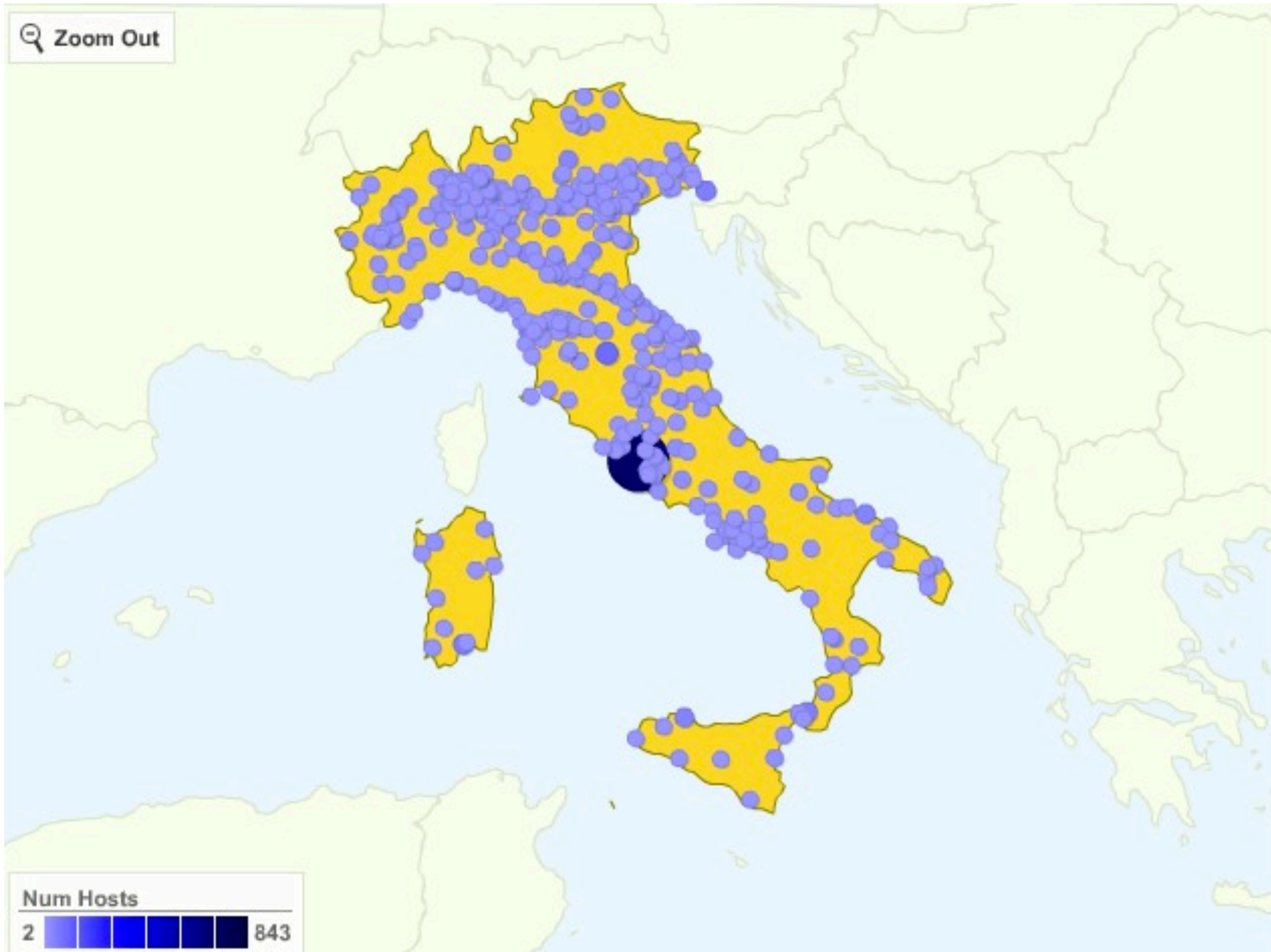
Live FastBit Query+Aggregation
Python Glue Software

Distance: 1					
ASN	AS Name	Traffic		Flows	Path
2597	REGISTRO CCTLD IT	1.0 GB	100.0 %	1367295	
Distance: 2					
ASN	AS Name	Traffic		Flows	Path
3356	Level 3 Communications, LLC	358.3 MB	95.7 %	633432	
137	GARR Italian academic and research network	12.0 MB	3.2 %	6951	
12637	Seeweb Srl	3.1 MB	0.8 %	2949	
21056	Welcome Italia S.p.A.	468.0 KB	0.1 %	1183	
21309	ACANTHO SPA	434.8 KB	0.1 %	284	
64862	??	216.0 KB	0.1 %	270	
16004	MIX S.r.L.	9.3 KB	0.0 %	20	
15469	Warinet NOC AS	365.0 bytes	0.0 %	1	
Distance: 3					
ASN	AS Name	Traffic		Flows	Path
9035	Wind Telecomunicazioni spa	35.5 MB	14.3 %	34395	
702	UUNET - Commercial IP service provider in Europe	25.9 MB	10.4 %	31974	
6762	Telecom Italia international high speed,	21.6 MB	8.7 %	36706	
3549	Global Crossing	17.7 MB	7.1 %	22630	
24940	Hetzner Online AG RZ-Nuernberg	16.0 MB	6.4 %	8078	
6453	Teleglobe Inc.	15.6 MB	6.3 %	22171	
12956	Telefonica Data Autonomous System	12.4 MB	5.0 %	28594	
1239	Sprint	11.2 MB	4.5 %	17075	

Using Geolocation Data [1/2]



Using Geolocation Data [2/2]



Disk and Memory Usage

- Collection of ~5k flows netflow/sec
- Each 5 min partition takes ~150 MB in FastBit format (32 GB/day)
- Partitions with raw data stay 3 days on disk (limited by available disk space)
- Each tuple archive in pickle format takes up to 400 KB (112 MB in total, almost constant).
- BGP patricia tree (inside the probe) of all routing tables takes about ~100 MB

Final Remarks

- NetFlow and sFlow are the two leading monitoring protocols.
- nProbe is an open-source software probe that can efficiently act as a probe/collector/proxy
- Storing and analyzing large volume of data is challenging but there are solutions available for doing it efficiently.
- Geolocation and routing information are useful for mapping traffic with users.