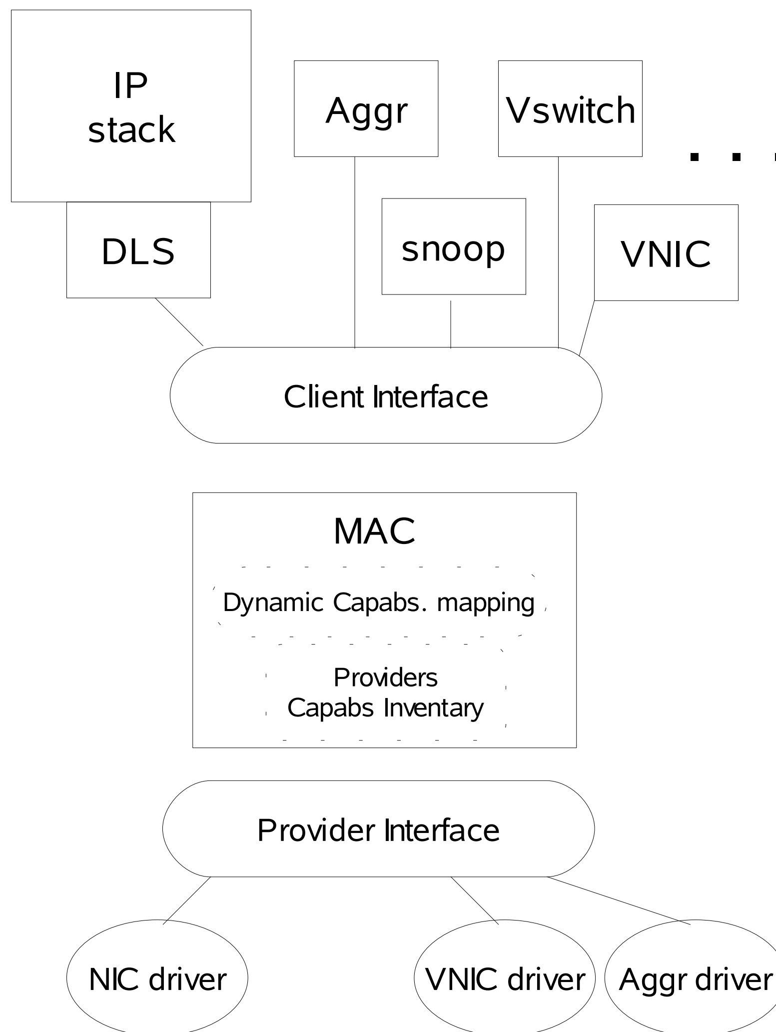
Crossbow Provider Model and Interface

Kais Belgaied & Roamer Lu

{kais.belgaied,roamer.lu}sun.com

Provider Model

This section describes the MAC provider model and interface. The MAC acts as an abstraction layer that presents a common behavior to its client, while dealing with the heterogeneous set of providers. MAC client API accesses the providers' capabilities as well as internal MAC information to handle the client's calls.



The first paragraph surveys the major NIC capabilities that are relevant to the virtualization work. Our abstract view of these capabilities is what makes up the provider model that we expect device driver writers to map their hardware capabilities to. The second paragraph describes the changes and extensions proposed to the GLDv3 interface introduced by the Nemo project [1] and its later amendments [2], in order to express the new and advanced capabilities. The design choices made in the second paragraph are explained by the features and constraints presented in the first. We include at the end a few examples as a guidance for device driver writers desiring to port to the provider interface introduced here.

NIC capabilities

Though heterogeneous in terms of capabilities and performance, NICs may be classified into three categories:

- DLPI NIC

This is a driver interfacing the TCP/IP stack using the Streams based DLPI interface. This is intended to be ultimately accessed by the stack via a shim-layer called ***softmac*** [3], so it could be looked at as NIC of the second type: Basic GLDv3.

- Basic GLDv3 driver.

We sometimes refer to this category as "dumb Nemo NICs). Drivers of this type offer GLDv3 MAC interface for data transfer and basic control functions. This is the case for all the MAC drivers shipped with Solaris Nevada and since Solaris 10U1.

- Virtualization-ready GLDv3

All NIC features that have to do with the ability to partition the NIC resources are relevant to virtualization. Virtualization-ready GLDv3 drivers may be equipped with:

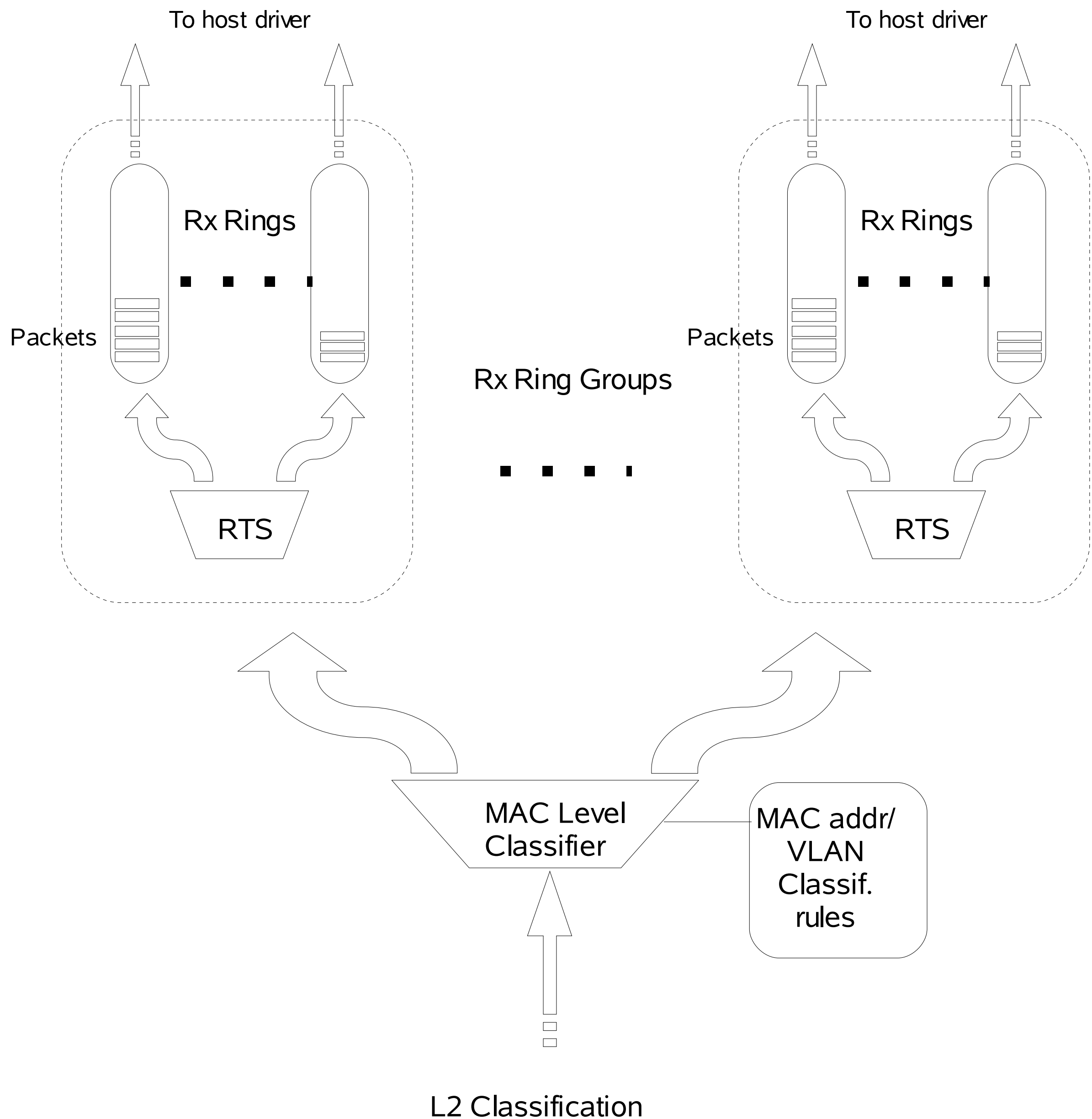
  - Multiple receive rings

  On the receive path, some drivers may also support a fixed or configurable number of receive rings. The term historically refers to the software structure that maps to a ring of DMA buffer descriptors. Incoming packets can be deposited into these rings in parallel. Packets can be picked off each ring independently.
  Receive rings have a finite capacity. When full, it is possible to specify the desired behavior, whether to drop packets destined to that ring, or to overflow to another ring.

  - Receive Ring Groups

  One of the advanced virtualization features is the ability to bundle multiple Receive Rings in a single group. One or more MAC addresses may be assigned to a group. Incoming packets destined to the a group's MAC address are delivered to any ring member, according to a programmable or predefined RTS policy (Receive Traffic Steering) (see next two paragraphs). Note that member rings can still be polled individually. Rx ring groups can come with a predefined set of member rings, or they are programmable by adding and removing rings to/from them.

  The following figure describes an abstraction of the ring groups



Within a group, packets are delivered to individual Rx rings according to a Receive Traffic Steering policy. The choice of recipient ring can be made based on a Receive Load Balancing, a programmed High Level Classification, or a cascade of both.

○ Receive Load Balancing

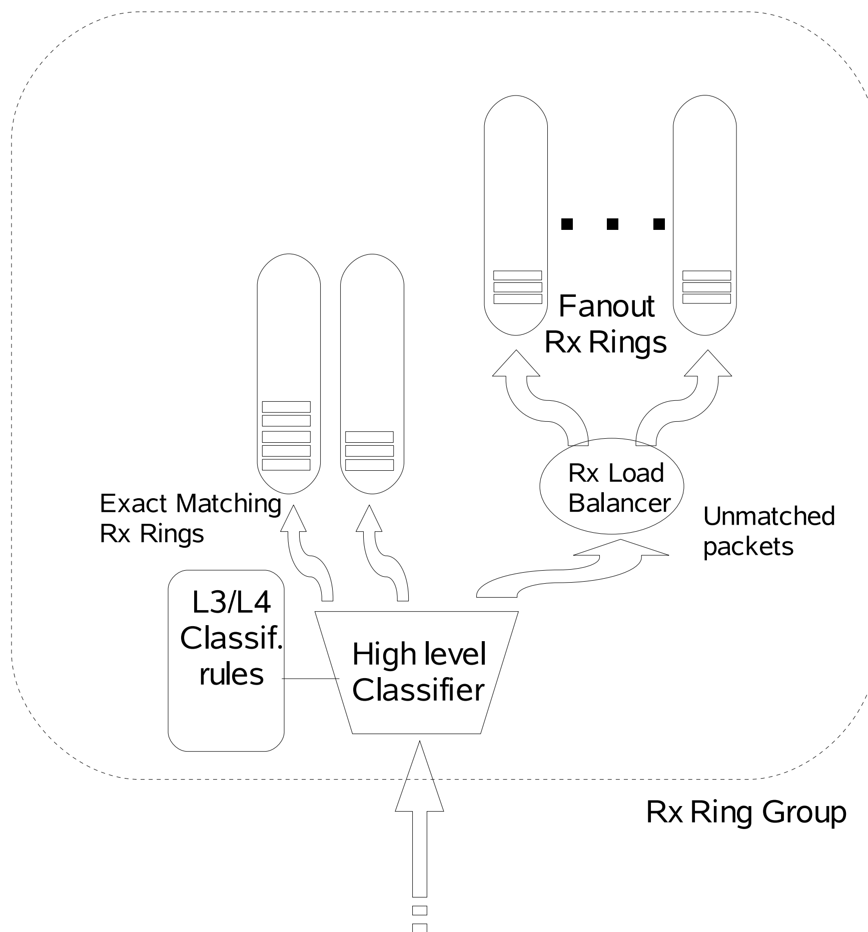
Some NICs are able to balance the incoming flow of packets between Receive Rings based on a predefined or programmed policy. This is a feature particularly useful when a single CPU is not capable handling the full line speed. Spreading the inbound traffic to multiple channels allows multiple processor to collaborate in receiving the packets. The ring selected for depositing an incoming packets is typically based on the value of a hash function computed over L3, L4 or other sections of the packet headers.

- High Level Classification

Classification of incoming packets may be based on:

- L3 criteria (IP source and destination, IPv6 flow ID, protocol number ...)
- L4 criteria (e.g. transport port number)
- Combination of both.

The outcome of the classification is depositing the packet that matches the criterion (also referred to as rule in some NIC specifications) into the programmed Receive Ring. Packets that do not match any classification rule are deposited into the default ring of the ring group.



RTS (Receive Traffic Steering)

The device has a Default Receive Ring (DRR). All unknown unicast or multicast packets and all broadcast packets are delivered to that ring. The group that includes that ring is called the

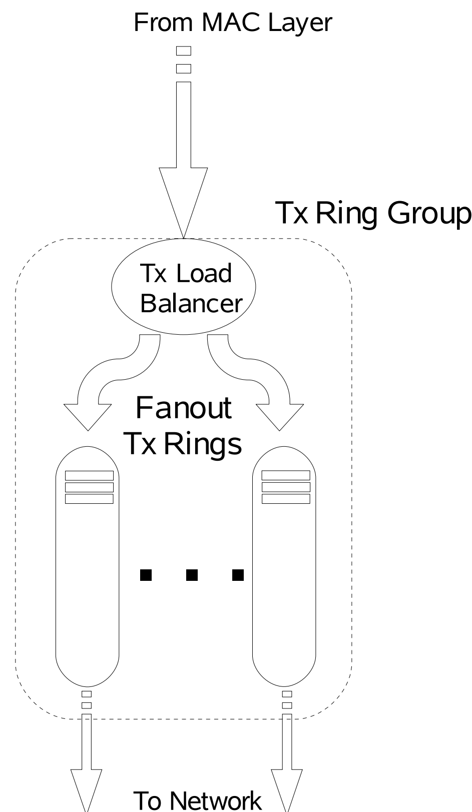
Default Receive Ring Group (DRRG).
Within a group, there is also a default ring. Packets that do not match any L3/L4 classification rule are deposited in that ring.

- Multiple transmit rings.

Multiple rings (also referred to in some NICs specifications as Tx FIFOs or Tx Queues) allow sending multiple packet flows through the same device in parallel. A Transmit Ring is a software concept that refers to a set of transmit descriptors used for outgoing packet(s). The device driver notifies the hardware to begin reading the descriptors from system memory when the packets are ready to be sent. Hardware may DMA the content of descriptors to hardware descriptor queue and eventually DMA the packets data and start the transmission.
Drivers supporting multiple transmit rings register the list of their opaque handles.

- Multiple Transmit Rings Groups

Similarly to Rx rings, Transmit Rings can be bundled together. A Transmit Rings Group is a set of transmit rings with same capabilities (hardware checksum, LSO, etc.) and a Transmit Load Balancer.
At the time of writing this document, only a few vendors announced future hardware support of Transmit Ring Grouping, and only a couple actually implemented a software version of it in the driver.



Outbound Load Balancing

- Multiple Interrupts Support:

  When the device driver is capable of MSI-X, multiple interrupt sources (packets coming, transmit completion, link status change, etc.) can be mapped to different MSI-X vector, so that interruptions for each event may be handled independently, and can be assigned to a different CPU. We envision three levels of organizing the interrupts assignment by a NIC driver. The choice of each level depends on multiple factors, including hardware implementation, interrupts availability, CPU speed, etc.

  - Per-ring level: Interrupts are allocated and associated with each ring, and may be toggled on and off independently from one another. Packets can be received in parallel from multiple rings. On the receive side, packet may be delivered by the device driver from the per-ring interrupt handler, or may be polled off a specific ring at a later time by the software stack. On the transmit side, the transmit completion routine may be invoked for an individual ring for faster recycling of the transmit descriptors.

  - Per-group level. In case the per-ring level assignment is not possible, interrupts can be allocated on a per-group basis. Events relevant to ring group members are reported through common interrupt(s).

  - Per device interrupt. Events from multiple groups are reported via common interrupt(s).

- Multiple MAC addresses

  In addition to the default vendor specified MAC address, a NIC may have room for listening on multiple unicast MAC addresses. Unlike the promiscuous mode of operation, NICs capable of this feature admit and possibly generate an interruption to the host for packets destined to those addresses only. All other packets are filtered out in hardware.
  Multiple MAC addresses are assigned to Receive Ring Groups. The driver and the hardware needs to guarantee that at least one MAC address is set for the group. It may optionally support adding more.
  A MAC address cannot be shared with other groups.

In addition to the virtualization related features, NICs have advanced features that are suited for performance, especially in the 10Gbps. These include:

- Large Send Offload (LSO)

  Large segments (up to 64KB currently) may be submitted in a single shot to the driver and hardware, which will deal with the segmentation into multiple MTU-sized packets including the generated headers, compute the checksum and send the frame out.

- Header-payload splitting for received packets.
- Large Receive Offload (LRO)
- Receive and transmit Hardware Checksum

Although beyond the scope of virtualization, we mention these these features in this context because

they will exposed as attributes of rings and ring groups.

Provider Interfaces

This document explains new driver interfaces introduced by Crossbow Virtualization Architecture.
In the following design, we make a distinction between the interfaces used to advertise the features and those that implement specifics of features. For example, both receive rings and transmit rings are resources delivering traffic, and some features of them are common and need common handling by the framework. So we adopt generic interfaces for drivers to advertise either type of resources for code redundancy avoidance. Therefore, we use a common data structure mac_capab_rings_t to describe multiple ring capability of either receive or transmit direction. The backing functions that fill in the list of rings and groups will use a mac_ring_type_t as a differentiator to indicate the type. The specific details of rings are described in the mac_rx_ring_info_t and mac_tx_ring_info_t, and mac_rx_ring_group_info_t and mac_tx_ring_group_info_t are for groups.

Multiple Rings Capability

Initially, drivers need to report the multiple rings capability to the framework as queried. Two new capabilities, MAC_CAPAB_RX_RINGS and MAC_CAPAB_TX_RINGS, are introduced to allow the MAC layer to get the number of hardware rings, ring groups, and per-ring/per-group function pointers.

The multiple rings capability is implemented through the mac_getcapab_t callback.

```
typedef enum {
     MAC_RING_TYPE_RX = 1,    /* Receive ring */
     MAC_RING_TYPE_TX = 2     /* Transmit ring */
} mac_ring_type_t;

typedef enum {
     mac_rx_ring_info_t,
     mac_tx_ring_info_t
} mac_ring_info_t;

typedef   void (*mr_get_ring_t)(void *, mac_ring_type_t,
     mac_ring_grp_drv_t, const int, mac_rx_info_t *,
     mac_ring_handle_t);
typedef   void (*mr_get_ring_group_t)(void *, mac_ring_type_t,
     const int, void *, mac_ring_group_handle_t);
typedef   void (*mr_add_ring_t)(void *, void *, mac_ring_type_t);
typedef   void (*mr_rem_ring_t)(void *, void *, mac_ring_type_t);

typedef   struct mac_capab_rings_s {
     uint_t         mr_rnum;       /* Number of rings */
     mr_get_ring_t  mr_get_ring;   /* Get ring from driver */
     uint_t         mr_gnum;       /* Number of ring groups */
     mr_get_group_t mr_get_group;  /* Get ring groups from driver */
     mr_add_ring_t  mr_add_ring;   /* Add ring into a group */
     mr_rem_ring_t  mr_rem_ring;   /* Remove ring from a group */
} mac_capab_rings_t;
```

- mr_rnum: the number of rings to be exposed

- mr_get_ring: the function used to get ring information from driver
  The first argument, void *, is the pointer of driver handle registered in mac_register().
  The second argument, mac_ring_type, indicates the type of ring (Rx or Tx).
  The third argument is the index identifying the ring. It is a number between 0 and (mr_rnum – 1).
  Depending on the mac_ring_type, the NIC driver will interprete the forth argument "mac_ring_info_t" as either mac_rx_ring_info_t or mac_tx_ring_info_t . the (*mr_get_ring)() routine will fill the ring information structure, and return.
  mac_ring_handle_t is the framework handle for the ring. It is expected that the driver uses this handle for all subsequent calls (e.g. Notifications of ring status change) to he framework referring to this ring.

- mr_gnum: the number of ring groups to be exposed

- mr_get_group: the function used to get ring group information from driver
  This function is similar to (*mr_get_ring)() except that it gets ring group instead of ring

- mr_add_ring: if present, any ring exposed separately could be added into any exposed group. The first argument is a exposed ring group handle from driver, and the second argument is a ring handle.

- mr_rem_ring: paired with mr_add_ring, this function is used to remove a previously added ring from a group.

Depending on the NIC hardware, a driver needs to decide how to organize and expose its multiple ring capabilities. The framework will query about the presence of ring groups first then separate rings. Within a ring group, if a fix number of ring is reported with the registration of ring group handle, the framework will create a appropriate data structure, *_ring_info_t, and ask the driver to fill it up according to the index, 0 ~ N-1, and then bind it to the assigned ring group. If no ring is pre-assigned to this group, it hints that the group is capable to be programed dynamically. After having gotten groups and their pre-assigned rings, the framework will register all un-assigned rings one by one if any. Two functions are optionally implemented for driver to group those free rings if capable.

As explained early, drivers are required to organize receive rings to groups following the previous descriptions, so both (*mr_get_ring)() and (*mr_get_ring_group)() are required to be implemented for receive side.

Unless the hardware supports transmit ring grouping, the driver must register all transmit rings directly by setting mr_gnum to zero.

It is worth noting that a driver can have configuration parameters for altering the initial grouping of Rx or Tx rings. The administrator may choose to maximize the number of groups, in order to increase the number of virtual NICs that can be built over this NIC. Alternatively, he/she may want to bring up only the primary interface and wishes to maximize the load spreading in order to achieve a higher throughput. Setting the initial layout to a single ring group encompassing all available rings on the NIC would be a suitable choice for this case. This initial configuration deals with deployment choices, it is

beyond the scope of this document, and is tackled by project Brussels [6]

Rx Rings

The data structure defined for driver to register receive ring is mac_rx_ring_info_t:

```
typedef   int  (*mac_ring_start_t)(mac_ring_drv_t);
typedef   void (*mac_ring_stop_t)(mac_ring_drv_t);
typedef   mblk_t    *(*mac_rx_ring_poll_t)(mac_ring_drv_t, int);

typedef struct mac_rx_ring_info_s {
     mac_ring_drv_t mr_driver;
     mac_intr_t          mr_intr;
     mac_ring_start_t    mr_start;
     mac_ring_stop_t     mr_stop;
     mac_rx_ring_poll_t  mr_poll;
} mac_rx_ring_info_t;
```

- mr_driver: ring handle registered by driver
- mr_intr: optimized per-ring interrupt handle, so the framework will be able to disable it and switch to precise polling mode
- mr_start: optional per-ring function that start the hardware ring . In this function, driver does hardware initialization of this channel, allocate DMA buffers for incoming packets, etc.

  Returning failure indicates the framework to consider another ring or retry later
- mr_stop: optional per-ring function that stop the hardware ring
- mr_poll: polling function that picks up a number of packets from driver
  This function take the first argument as the ring handle implemented by driver, and the second arugument is the how many bytes that will be polled out.

When the registered function mr_get_ring() is called, if the ring type is `MAC_RING_TYPE_RX`, driver casts the forth argument, mac_ring_info_t *, to `mac_rx_ring_info_t` and fill in all applicable information. If a pre-ring interrupt is supported and successfully allocated, mr_intr is used to pass the interrupt handle to the framework. Otherwise the mr_intr should be set to NULL. Drivers are suggested to support per-ring mr_start function and implement DMA buffer allocation in it. mr_poll() is a required function if that per-ring interrupt is supported. Based on bandwidth control or the pressure of too many interrupts, the framework may to disable interrupt and start to poll the receive ring. The implementation of mr_poll() returns a chain of messages which has the specified number of packets.

The data structure defined for driver to register receive ring group is mac_rx_ring_group_info_t:

```
typedef   int  (*mac_ring_grp_start_t)(mac_ring_grp_drv_t);
typedef   void (*mac_ring_grp_stop_t)(mac_ring_grp_drv_t);
typedef   int  (*mac_set_mac_addr_t)(mac_ring_grp_drv_t, uint8_t *);
typedef   int  (*mac_add_mac_addr_t)(mac_ring_grp_drv_t, uint8_t *);
typedef   int  (*mac_rem_mac_addr_t)(mac_ring_grp_drv_t, uint8_t *);
```

```
typedef   int  (*mac_set_lb_policy_t)(mac_ring_grp_drv_t,
     mac_lb_policy_t);
typedef   int  (*mac_add_rule_t)(mac_ring_grp_drv_t,
     mac_ring_drv_t *, mac_rts_rule_t);
typedef   int  (*mac_rem_rule_t)(mac_ring_grp_drv_t,
     mac_ring_drv_t *, mac_rts_rule_t);

typedef struct mac_rx_ring_group_info_s {
     mac_ring_grp_drv_t  mrg_driver;
     mac_intr_t               mrg_intr; /* Optional per-group
                                           interrupt */
     mac_ring_grp_start_t     mrg_start;     /* Start the group */
     mac_ring_grp_stop_t mrg_stop; /* Stop the group */
     uint_t              mrg_rnum; /* Number of ring in the group */
     mac_set_mac_addr_t  mrg_setmac;/* Set the default MAC addr */
     mac_add_mac_addr_t  mrg_addmac; /* Add a MAC address to the
                                        group */
     mac_rem_mac_addr_t  mrg_remmac;    /* Remove a pre-added MAC
                                           address */
     mac_set_lb_t        mrg_setlb;     /* RX load balancing
                                           Policy */
     mac_add_rule_t      mrg_addrule;/* Add a classification rule */
     mac_rem_rule_t      mrg_remrule;/* Remove a classif'ion rule */
} mac_rx_ring_group_info_t;
```

- mrg_driver: ring group handle registered by driver
- mrg_intr: “nice to have” per-group interrupt if per-ring interrupts are not guaranteed by hardware or system
- mrg_start: optional function that starts the ring group to function
- mrg_stop: optional function that stops the ring group
- mrg_rnum: number of pre-assigned rings. If 0, the group is capable to be regrouped later
- mrg_setmac: function that sets a MAC address to a ring group. This function should be called before starting this group.
- mrg_addmac: optional function that adds an extra MAC address to the ring group
- mrg_remmac: paired with mrg_addmac, this function removes an pre-added MAC address from the ring group. Both functions are callable anytime
- mrg_setlb: optional function that change receive side scaling policy
- mrg_addrule: optional function that could steering a specific type of packets into the selected ring in this group

- mrg_remrule: paired with mrg_addrule, the function is used to remove a pre-added classification rule

Before calling (*mr_get_ring)(), the framework calls (*mr_get_ring_group)() as explained above. Driver casts the forth argument, void *, to `mac_rx_ring_group_info_t` and fill in all available information. mrg_rnum is used to decide how many times the framework will call (*mr_get_ring)(). The set of mrg_*mac() functions are used to replace existing multiple MAC addresses interfaces introduced by PSARC/2006/265 by a simpler way. mrg_setmac() is called to set the default MAC address for this receive ring group. mrg_addmac() and mrg_remmac() are optional functions paire that add/remove extra MAC address in try-and-fail way.

Tx Rings

The data structure defined for driver to register transmit rings is mac_tx_ring_info_t:

```
typedef   int  (*mac_ring_start_t)(mac_ring_drv_t);
typedef   void (*mac_ring_stop_t)(mac_ring_drv_t);
typedef   mblk_t    *(*mac_tx_ring_send_t)(mac_ring_drv_t, mblk_t *);

typedef struct mac_tx_ring_info_s {
     mac_ring_drv_t      mr_driver;
     mac_intr_t               mr_intr;
     mac_ring_start_t         mr_start;
     mac_ring_stop_t          mr_stop;
     mac_tx_ring_send_t  mr_send;
} mac_tx_ring_info_t;
```

- mr_driver: ring handle registered by driver
- mr_intr: per-ring interrupt handle if any
- mr_start: optional per-ring function that start the hardware ring
- mr_stop: optional per-ring function that stop the hardware ring
- mr_send: transmit functions that send the packet in the passed down message out

Transmit rings are organized in the similar way used by receive rings, but the grouping support is optional. mr_send() send a single message, packet, out instead of sending a chain of messages.

MAC Driver Event Update

```
typedef enum {

     MAC_EVENT_DEV_LINK_UP = 1,
     MAC_EVENT_DEV_LINK_DOWN = 2,
     MAC_EVENT_RING_TX_AVAILABLE = 3
} mac_event_type_t;
```

```
typedef   void *mac_event_obj_t;
typedef   void *mac_event_info_t;

void mac_event_update(mac_event_type_t, mac_event_obj_t,
     mac_event_info_t);
```

For accurate flow control on the transmit ring, the framework needs to know the the ring's status changes immediately. The driver is required to call mac_event_update() to notify the availability of tx descriptors. The call can be made in interrupt context. New hardware will be able to reset/restart single ring or ring group independently, and the mac_event_type will be extended to handle more types of driver/hardware events.

Examples of writing various driver types into this framework

The new interfaces supporting multiple rings and ring groups are introduced to utilize advanced hardware virtualization features that are implemented differently among 10GbE NIC vendors. Can the new framework virtualize various hardwares without impacting performance? We'll investigate two different types of 10GbE NIC to see the flexibilty of new interfaces. To simplify the model, only major relevant features are listed below.

#Example 1: 10GbE NIC “X”

- 8 independent transmit channels (without grouping)
- 8 independent receive channels (up to 4 groups)
  - Dynamic reconfiguration of channel grouping
  - Packet Classification within a ring group (configurable for receive load balance)
- 16 MAC addresses that can be associated with any ring group
- MSI-X support based on interrupt group

Without transmit ring grouping, the driver of NIC X can simply register all transmit rings through the MAC_CAPAB_TX_RINGS capability. The framework will group rings and assign the virtual transmit ring groups to different MAC clients. In every virtualized group, a software transmit load balancer is adopted to distribute traffic to different rings.

Grouped receive rings are organized and registered through MAC_CAPAB_RX_RINGS capability interface. To maximize the flexibilty of regrouping receive rings, the driver may register all 4 ring groups without assigning any ring. All 8 channels are organized as free channels, that will be grouped into any selected ring group afterwards through mr_add_ring()/mr_rem_ring().

16 MAC addresses are shared by 4 channel groups. It's up to driver whether to reserve multiple MAC address slots for every ring group.

#Example 2: 10GbE NIC “Y”

- 8 independent transmit channels (without grouping)
- 8 independent receive channels (without grouping)

- Priorilized Receive Traffic Steering Engine
    - Socket Pair Direct Match Hashing
    - TCP/UDP Port Steering
    - MAC addresses classification (receive ring based)
    - up to 32 MAC addresses supported per ring
- MSI-X based on either transmit or receive channel

The different implementation of receive channels/rings make the driver implementation different from the one for NIC “X”. Driver need to decide how to group independent ring explicitly. Considering the priorilized hardware classification, there is no clear classification layers, driver may either group all receive rings to one group, in which the Socket Pair Hashing or Port Steering are used to do load balance among all rings, or to register 8 single-ring groups with multiple MAC addresses support, up to 32 addresses per ring. The first approach is to maximize the receive side performance, but the latter is to maximize its virtualization capabilities.

#Example 3: Legacy NIC “Z”

(TBD)

Expected Hardware Implementations

What's the ideal NIC model that will fit in the new virtualization framework perfectly? During designing the interfaces, we investigated many latest 10GbE hardwares and summarized our expected features here:

- 16 transmit channel groups with independent DMA channels (at least 4)
    - Nice to have 4 independent tranmit channels per group, or dynamically configurable
    - Hardware implementation of load balancer
    - Per-ring based LSO and Full Hardware Checksum support
    - Per-ring or per-group MSI-X interrupts
- 16 receive ring groups with independent DMA channels (at least 4)
    - At least 4 independent receive channels per group, or dynamically configurable
    - Layered packet filter/classification
        - L2 Destination Address Filter/Classfier based on 8 different MAC addresses
        - L2 VLAN tag based classification that can be programed to be used together with destination MAC addresses together or not
        - Configurable L3,L4 Header based classification that steers packet to any selected ring in the group, then
        - Configurable L3,L4 Header based load balancer that fanout rest traffic among all other rings in the group
    - Per-ring MSI-X itnerrupts
    - Per-ring LRO support and Full Hardware Checksum support

Other omitted features doesn't mean those are not important, but these listed features are

desired to have to support the virtualization, energy saving without compromising performance of 10GbE. Crossbow Provider Model organizes hardware resources and do flexible virtualization assignment, and it unifies various hardware implementations to common interfaces for MAC clients.

References

[1] Nemo, http://www.opensolaris.org/os/community/networking/nemo-design.pdf

[2] Extended capabilities, http://www.opensolaris.org/os/project/clearview/nemo-binary-compatibility.txt

[3] Clearview UV, http://www.opensolaris.org/os/project/clearview/uv-design.pdf

[4] LSO (Large Send Offload) PSARC/2006/190 (To be published)

[5] LRO (Large Receive Offload)

[6] Brussels http://www.opensolaris.org/os/project/brussels/files/brussels.pdf