# Crossbow Architecture Overview

## PSARC/2006/357 Inception Review

## 1. Motivation

Project Crossbow extends Solaris reach in several markets:

### a) OS/Network/Server Consolidation:

The application, network and server consolidation environments where both OS and network virtualization play a big role. This market is typically driven by the cost of owning and managing physical machines and physical networks. The sweet spot for these horizontally scaled environment are the 2-4 socket machines which appear as 4-8 CPU machines in case of x86/x64 systems and 32-64 CPU machines in case of SUN's new Niagara based servers. From total cost of ownership perspective, these blades have only one physical NIC (1Gb or 10Gb) but are trying to run multiple virtual machines (Xen, Containers, LDoms) which have to share the NIC resources and the available bandwidth.

The problem gets worse because for 3 decades we have been designing applications to go as fast as possible and congestion control is the job of the transport layer (if at all). So if one virtual machine is using UDP based traffic, then other virtual machines on the same system using TCP traffic will suffer badly. Even within same transport (TCP for instance), bulk throughput applications like ftp/http etc will have a very negative impact on interactive traffic and latency sensitive applications.

The goal of the project Crossbow is to different virtual machines share the common NIC in a fair manner and allow system administrators to set preferential policies where necessary (e.g. the ISP selling limited B/W on a common pipe) without any performance impact.

### b) Traditional QOS and application consolidation:

Existing host based QOS mechanisms are very complex to setup and typically come with a sizable performance penalty and increase in latency. The big part of the problem is the interrupt based delivery mechanism for inbound packets and the QOS being implemented by a separate layer (typically between NIC driver and IP). The network and transport layer of the host stack are unaware of the QOS layer. Packets are already delivered to the host memory by means of interrupts and the QOS layer needs to classify the packets to various queues before it can apply the policies. In case the packet can not be processed because the bandwidth usage for that class is exceeded, it sits in a queue while still consuming system memory.

Project Crossbow integrates stack virtualization and QOS as part of the stack architecture itself to offer a large subset of QOS type functionality at zero performance penalty and simple administrative interfaces. It also integrates diffserv [1] with the stack where a virtual NIC can set and read the diffserv based labels. Since Crossbow architecture is limited in differentiating the traffic based on layer 2, 3, and 4 headers only i.e. the VLAN tag, local mac address, local IP address, protocol, and ports; the functionality offered is a subset of existing QOS mechanism although it covers the vast majority of the use cases without any performance penalty. This is the prime reason why project Crossbow refers to

the bandwidth related policies as 'Bandwidth resource control' instead of QOS.

### c) Horizontally scaled markets:

This is the market segment made up of low priced volume servers (typically 2-4 socket machines) which offer services that require little or no sharing of data between them. The small servers can be standalone machines in a rack or blades in a chassis. Grids are another way to use volume servers to achieve the output of the traditional large SMP machines or mainframes.

In case of blades which share a common 10Gb NIC to the chassis, Crossbow again provides the sharing of bandwidth in a fair manner. In addition, the Crossbow provided APIs for network management, virtualization and bandwidth resource control can be used by 3rd party management software to propagate the common policy throughout the server farm or all the blades in the chassis. In a Solaris based homogenous environments, its very easy to mark an application or a virtual machine (based on port or IP address) as critical and propagate the same policy through all the machines. The diffserv labels can be added appropriately such that the policy is honoured by all machines and network element in the center.

## 2. Technical problems in existing architectures

As mentioned earlier, the host based QOS systems work as a layer between the network stack and as such are pretty inefficient in providing the QOS services required of them. But that is not all.

The existing interrupt driven packet delivery model precludes any kind of policy enforcement and fair sharing. When a NIC interrupt is raised, it is at a high priority and the CPU has to context switch whatever processing to deal with the interrupt. Very often, the processing of a critical packet is interrupted to deal with the arrival of a non critical packet.

The anonymous packet processing in the kernel is another major problem in virtualizing the stack and enforcing any kind of bandwidth resource control (including fairness). 80% of the work is already done for an incoming packet when the stack determines that no one is actually interested in the packet and it needs to drop it. In other words, the cost of dropping unwanted packets is too high.

Everything in the host flows through common queues and is processed by common threads which make enforcing policies based on traffic type very difficult. Receiving or transmitting each packet impacts processing on any other packet on that particular CPU.

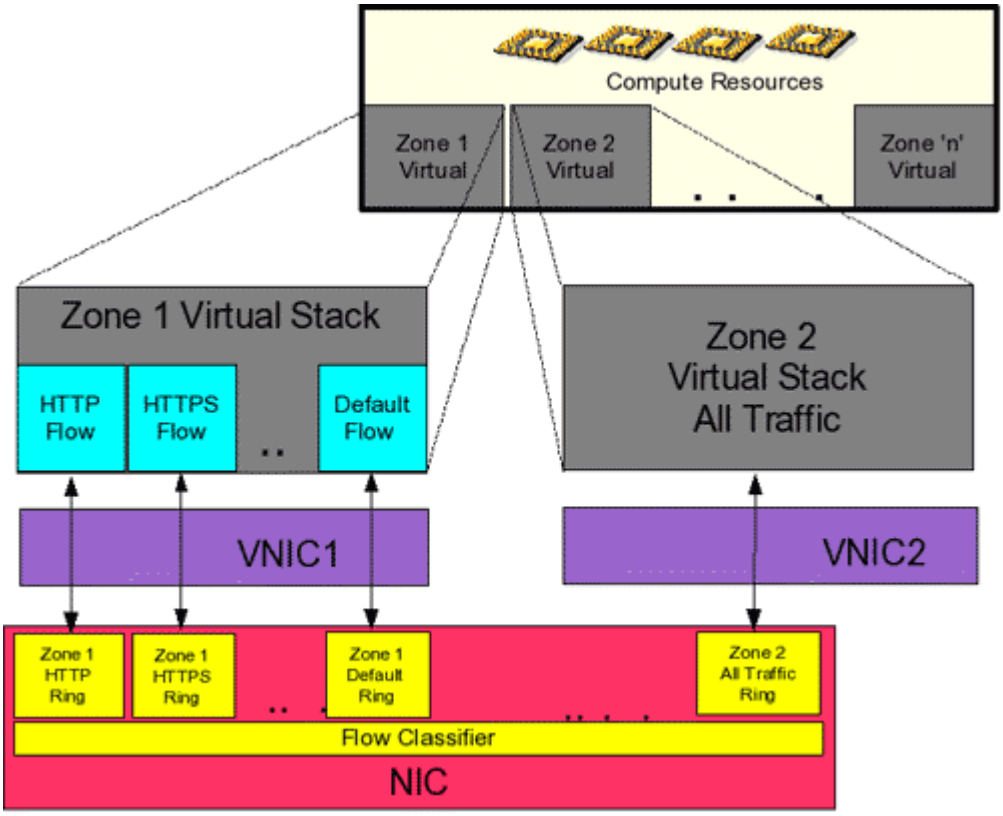
In most of the virtualized environments, the pseudo NIC in the virtual machines has no way of knowing about the hardware capabilities of the real hardware (even simple things like hardware checksum). Additionally, there is no mechanism to share the NIC in a fair manner. The transition of typical packet from the dom0 to domU also causes severe performance problems.

Modern NICs vendors have been aware of the advent of virtualization. Most 10GNICs offer multiple transmit and receive queues, hardware level packets classification, multiple interrupts `and multiple MAC addresses, etc. The networking stack is lagging behind. It has not evolved to discover and utilize those capabilities.

# 3. Crossbow Architecture

## a) High Level Architecture

At a very high level, the Crossbow architecture consists of the following major components: Virtual NICs (VNICs), network resource control using the network stack serialization queue (squeues [2]), and hardware support for flow classification. These components and their interaction are represented by the following figure:



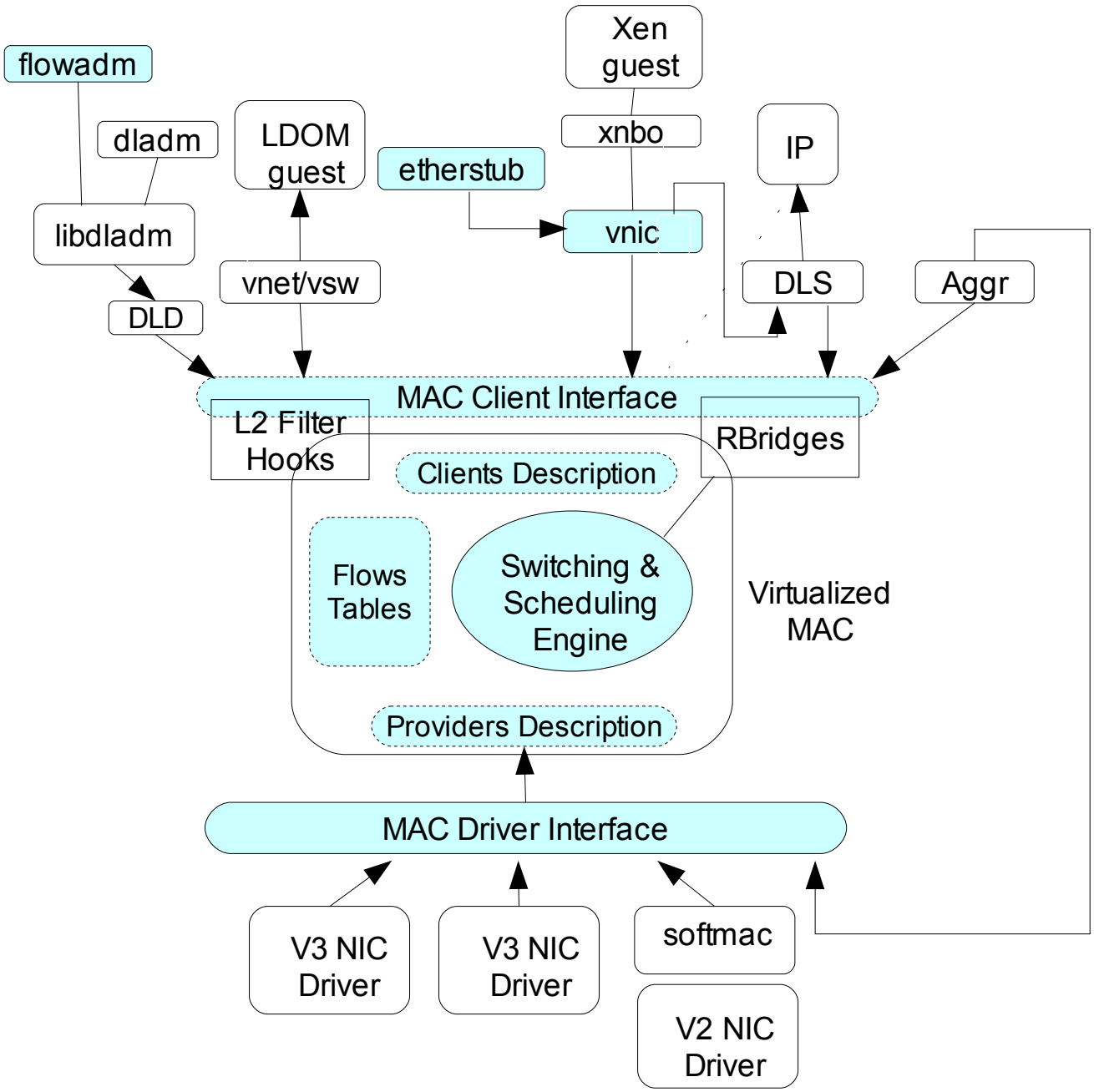

The Crossbow architecture starts out by integrating network virtualization and resource control as part of the stack architecture. The Solaris 10 network stack [2] has already been designed for the next decade where the connection to CPU affinity is maintained and the upper stack has tight control over the NIC resources.

Crossbow builds on top of that by pushing the classification of packets based on services, protocols or virtual machines as far below as possible. If the NIC hardware itself has ability to divide onboard memory into segments/queues (know as Rx and Tx rings) which can preferably have their own DMA channels and MSI-X interrupts, the stack programs the NIC classifier to classify packets based on configured policies to different Rx rings. Each Rx/Tx ring is owned by a CPU and a separate kernel queue know as serialization queue which controls the rate of packet arrival into the system based on configured bandwidth.

The Rx/Tx ring, the associated DMA channel, MSI-X interrupt, the serialization queue, the CPU, and

processing threads are all unique for the service, protocol or virtual machine in question and can be assigned a unique MAC address and a Virtual NIC which becomes the administration entity that can be administered like a normal NIC. The NIC classifier drives the incoming packets to the correct RX ring from where the Squeue owning the Rx ring (and VNIC) will pull the packets via polling mode based on fair sharing of resources or configured bandwidth. The interrupt mode is used only when the Squeue has no packets to process and the Rx ring is empty. Each individual Rx ring is dynamically switched between interrupt and polling mode. Incoming packets that exceed the configured bandwidth limit remain in the NIC itself in their corresponding Rx ring and are pulled in the system only when they are ready to be processed.

## b) Scope and Components of the Architecture



In the picture above, the components in pale green are newly developed by the project. The components in grey/marble boxes refer to existing modules, libraries and executables ported to the new interfaces introduced or modified by the project. The components in orange boxes refer to projects under development that will be ported to Crossbow.

**Virtualized MAC**

MAC layer interfaces network device drivers with the stack. It was introduced as part of Project Nemo a.k.a. GLDv3 [3]

The MAC layer is virtualized by Crossbow. It allows access by multiple MAC clients. Each MAC client is assigned its own hardware resources (rings, interrupts, etc), bandwidth limit, priority, and set of flows.

The MAC layer programs the hardware classifier to steer traffic to the clients, thus ensuring fanout to multiple CPUs. If the NIC does not support hardware classification, it uses a software classifier and fans out to MAC software rings.

**Vnics**

Vnics are Implemented as a pseudo Nemo/GLDv3 MAC driver and are managed through dladm(1M).

A Vnic is seen by the system as a virtual network devices which can be used as any regular NIC It can be assigned to a zone or a Virtual Machine (e.g. XVM).

Vnics are based on the virtualized MAC layer, each Vnic corresponds to a MAC client. Note that a Vnic is an administrative entity, and not an extra layer. It is pass-through for data path and most of the control path for better performance

**MAC Clients**

Today, there's a single mac_handle_t used by both clients and drivers to identify a device
With Crossbow, multiple MAC clients have their own mac_client_t data structure.
Each MAC client is associated with

- Receive callback function
- Set of unicast and multicast addresses, VLAN id
- Statistics
- Link state
- Promiscuous callbacks
- Rings, bandwidth limit, priority, set of CPUs, fanout

**Switching Engine**

The MAC layer provides packet switching semantics, equivalent to an ethernet switch:
A virtual switch is created implicitly each time more than two MAC clients are created on a NIC.
The switching code

- provides a local data path at the MAC layer between MAC clients (e.g. VNICs, plumbed NIC) defined over the same NIC.
- Provides connectivity between MAC clients and external network
- Distributes broadcast and multicast packets to local MAC clients and external hosts

**Etherstubs**

VNICs can be created on top of ether stubs instead of regular NICs, thus overcoming the limitation in number of physical NICs available. It also allows the implementation of completely virtual switches (aka network in a box). An ether stub is a special NIC with no underlying hardware. Like with physical NICs, Vnics created on top of the same ether stub are connected through a virtual switch. Unlimited number of ether stubs can be created.

**Scheduling Engine**

As mentioned above the VNIC is just an administrative entity. If the classification has already been done by the NIC to a particular Rx ring, the packets as delivered directly to IP layer by means of function calls when Rx ring is interrupt mode or the squeue residing in IP layer pulls the packet chain directly from the Rx ring when in the polling mode. In essence, the entire data link layer is bypassed resulting in improved performance and lower latencies.
The entire layered architecture is built on function pointers know as 'upcall_func' and 'downcall_func' with corresponding 'upcall_arg' and 'downcall_arg' for context. Every layer provides a pointer of its recv function as 'upcall_func' and a context as 'upcall_arg' to the layer below. This is how the packet path is constructed. Any layer can short circuit itself out by providing the 'upcall_func' and 'upcall_arg' of the layer above to layer below (and same for transmit side if needed). All context cookies for a layer work on reference based system when each layer pointed to it gets a reference and ensure that data structures don't get freed till all references are dropped.
In case, the NIC hardware does not have classification capability or have run out of the classification capability, the architecture provides a classification capability in the mac layer and employs soft rings which are similar to functionality as NIC hardware classifier and RX rings. The NIC hardware layer coupled with lower MAC layer and soft rings are termed as 'Pseudo Hardware layer'. A request by administrator to create a new VNIC or flow will always return successful from the pseudo hardware layer. The pseudo hardware layer manages the hardware and software classification capability and Rx rings and soft rings transparently from upper layers.

**Flows**

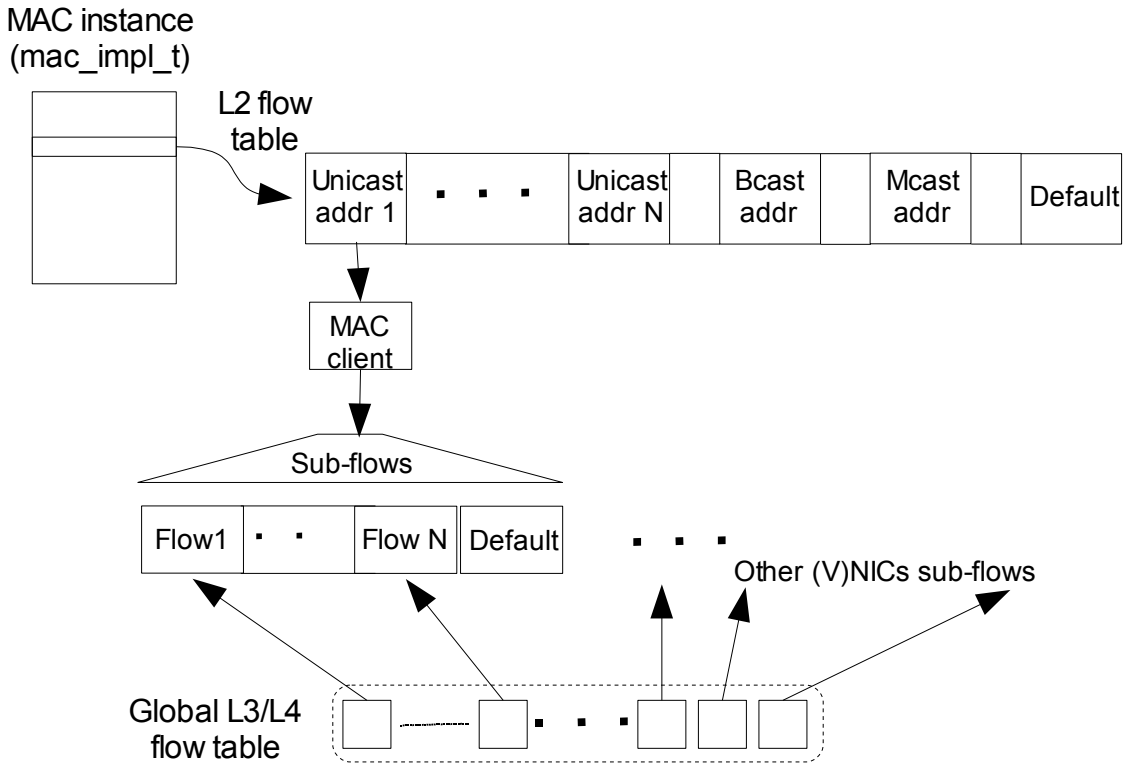
A flow entry is the internal data structure that maps to a flow. It contains mainly:

- information about the criteria describing the flow (what headers, what values)
- information about the treatment for the packets matching the flow
    - function pointer and argument to invoke in case of marching
    - resource controls (scheduling info, priority, bandwidth limit, etc.)

The virtualized MAC maintains a flow table for each MAC device instance (represented by a mac_impl_t structure). The table has an entry for each unicast MAC address (and/or VLAN) per client open over that instance (DLS as a primary MAC, or Vnic), as well as an entry for each broadcast and multicast flow. This is the L2 flow table. In case of hardware classification, the unicast entries mirror the hardware classification rules. They are needed in this case for proper steering in inter-client communication, or for possible delivery of packets to the default ring instead of a dedicated ring, in case of hardware limitation (packets mis-parsed by the hardware classification due to fragmentation, presence of IP options, bugs. Etc).

The outcome of the L2 classification is a flow entry that delivers the packet to a MAC client.

Additional flows may have been created over the MAC client, using flowadm in order to set higher level resource controls based on IP addresses, transport or services. These flows are kept in tables associated with each MAC client.




Flow Table Organization

**Providers and MAC driver Interface**

See Provider Model section below.

## c) Changes to Other Components

The main two changes are

- on the consumer side of MAC.

  The introduction of the notion of virtualized MAC layer mandated that consumers now need to access MAC exclusively and in a controlled way from the MAC client interface. Previous direct accesses using driver exposed capabilities are not allowed any more, since they bypass the

ability to virtualize the NICs.

Crossbow ported the DLS, DLD, LDom vnet/vswitch code, and the back end Xen driver to the new MAC client API.

- On the provider side

  GLDv3 drivers, including softmac are modified to use the new driver interface.

RBdiges [5] and L2 Filter hooks [6] are two other projects that are affected by Crossbow's re-writing of the MAC layer. They both have an architectural dependency on Crossbow.

# 4. Provider Model

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.

This paragrap 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. A future detailed design will describe the changes and extensions proposed to the GLDv3 interface and its later amendments [7], in order to express the new and advanced capabilities. The design choices made in that document are explained by the features and constraints presented here. The detailed design will include a few examples as a guidance for device driver writers desiring to port to the provider interface.

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***, 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:

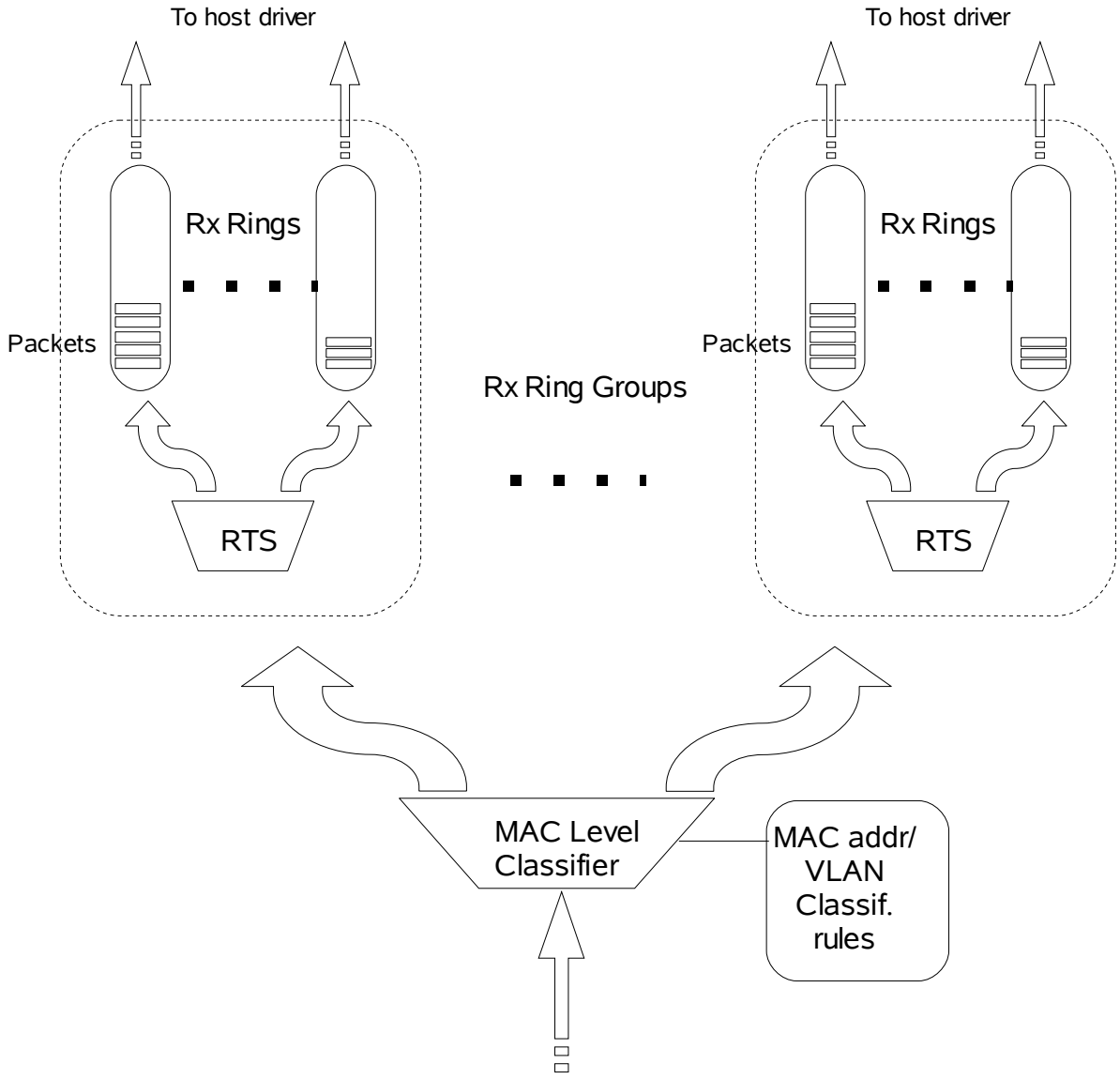
  - Multiple receive rings

    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

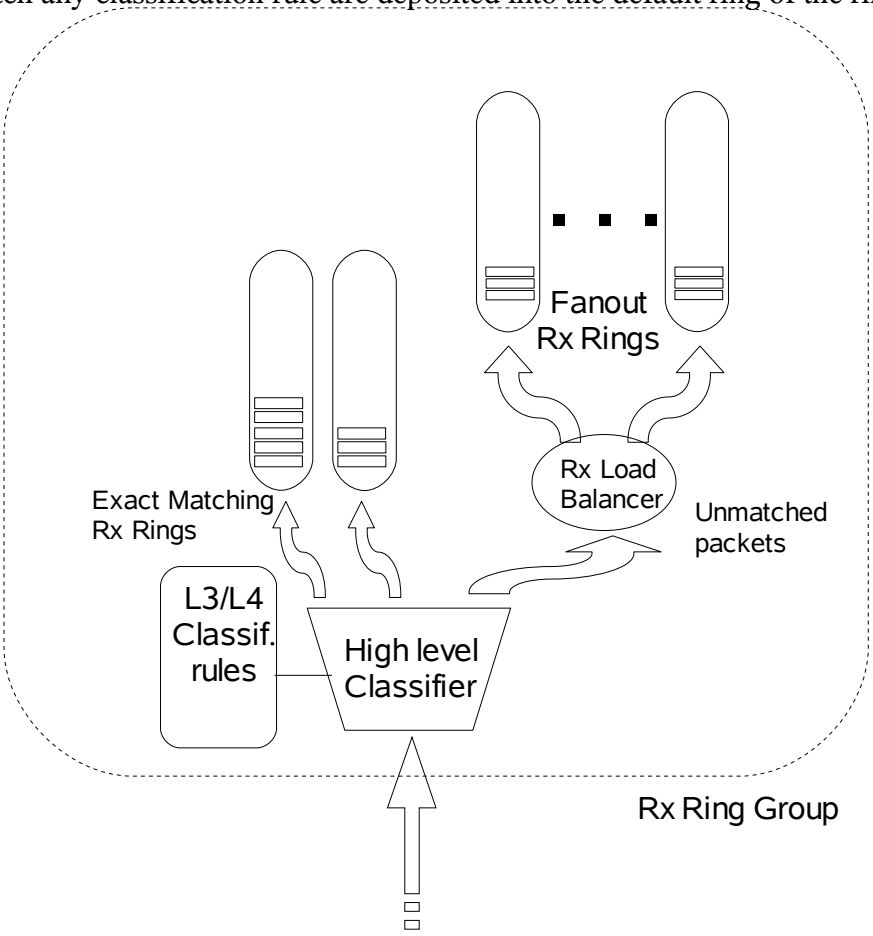
Some NICs are able to balance the incoming flow of packets between Receive Rings based on a predefined or programmed policy. This is a feature particularly useful when a single CPU is not capable handling the full line speed. Spreading the inbound traffic to multiple channels allows multiple processor 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)

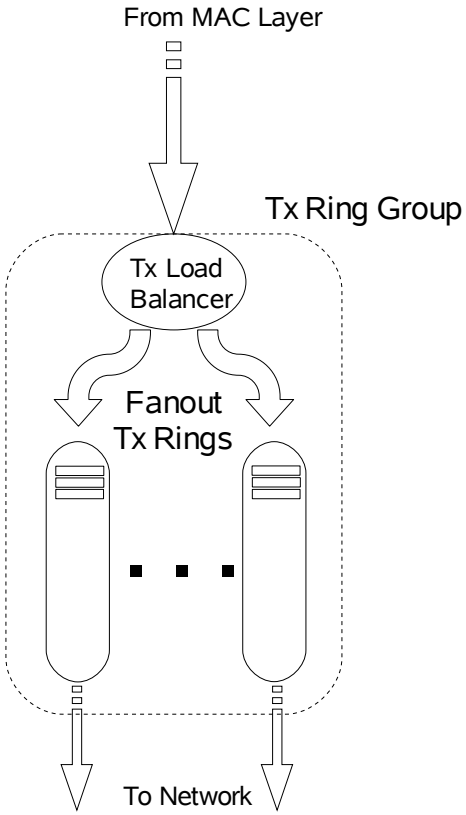
The device has a Default Receive Ring (DRR). All unknown unicast or multicast packets and all broadcast packets are delivered to that ring. The group that includes that ring is called the Default Receive Ring Group (DRRG).
Within a group, there is also a default ring. Packets that do not match any L3/L4 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.

# 5. Administrative Model

Crossbow introduces a new command called flowadm(1m) and further augments dladm(1m) which was introduced as part of Nemo.
dladm (1M)- This is primarily used to create, modify and destroy VNIC based on mac addresses or VLAN. The created VNIC is visible and managed by ifconfig(1m) just like any other NIC and can get its IP address assigned via DHCP if necessary.

flowadm(1M) manipulates flows based on IP or transport level criteria over a given datalink.

Both command set the resource control properties such as bandwidth limits, priority, CPU binding to a data link. They also offer observability features for real time bandwidth usage, or to gather historic usage data for post mortem analysis.

See attached draft man pages.

# 6. Obsoleted Interfaces

With Crossbow, the following two features are obsoleted:

. PPA “hack” for expressing VLANs introduced with Nemo.

. IPQoS

## References:


[1] RFC 2474 http://www.ietf.org/rfc/rfc2474.txt

[2] FireEngine: A new architecture in Networking. PSARC/2002/433

[3] Nemo - a.k.a. GLD v3: PSARC/2004/571

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

[5] RBridges: ssh://anon@hg.opensolaris.org/hg/rbridges/rbridges-doc

[6] L2 Filter Hooks http://jurassic.sfbay/~zf203873/presentation/layer2-summit-20080521.pdf

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


## Credits


Sunay Tripathi, Nicolas Droux, Roamer Lu and Kais Belgaied contributed to the content of this document