Crossbow Network Virtualization Architecture

Nicolas Droux
nicolas.droux@sun.com

Solaris Core OS
Sun Microsystems, Inc.

Revision 3
November 21, 2007

Document History

Date	Revision	Author	Changes
7/31/07	1	Nicolas Droux	First version
8/28/07	2	Nicolas Droux	Misc updates to MAC API
11/21/07	3	Nicolas Droux	VNIC pass-through, updated MAC client API, other misc changes

Table of Contents

1 Introduction

Virtualization allows multiple virtual machines or Solaris zones to be executed simultaneously on a single physical machine. Virtualization technologies provide advantages such as providing isolation between multiple virtual machines, allowing for multiple instances of an operating system to co-exist on a single machine for high availability, and last but not least allowing improved return on hardware investment by consolidating multiple machines as multiple virtual machines running on a single host.

In a virtualized environment, multiple virtual machines or zones share the physical resources available on the machine. These physical resources include processor(s), memory, I/O, and network interfaces.

This document describes in details the Crossbow network virtualization architecture, which provides virtualization capabilities at the MAC layer of the Solaris stack, and the ability to leverage this MAC virtualization using virtual NICs (VNICs), which can be assigned to zones or virtual machines.

1.1 Functional Specification Summary

This section provides an overview of the functionality provided by the Crossbow network virtualization architecture. These various items are described in more details later in this document.

1. Allow a hardware NIC or link aggregation to be virtualized at the MAC layer, and the instantiation of multiple virtual NICs on top of such data-link entities.

2. Provide an administration command-line interface integrated with the existing Solaris tools for the management (creating, deletion, modification, display) of VNICs.

3. Provide an administration library allowing the management of VNICs programatically.

4. Allow the VNIC configuration to persist across reboots.

5. Allow each VNIC to appear to the Solaris host as a regular NIC which can be plumbed by IP, accessed by MAC clients, having its own MAC address, and exposing its own set of statistics.

6. Allow VLANs to be created on top of VNICs, and at a later time allow VLANs to be implemented at the VNIC and MAC layer instead of DLS.

7. Allow VNICs to be plumbed by Solaris zones.

8. Provide connectivity between MAC clients such as VNICs and IP accessing the same physical NIC.

9. Allow a NIC to be used by VNIC and plumbed by IP directly simultaneously, and provide connectivity between VNICs and that plumbed interface.

10. Distribute broadcast and multicast packets to MAC clients, and the physical wire when applicable.

11. Allows the global zone or Xen dom0 to observe all traffic sent and received by the physical NIC, as well as all the inter-VNIC traffic.

12. Allows each VNIC to be snooped, in this case do not show the traffic pertaining to the other VNICs or directly plumbed interface.

13. Allow MAC clients such as VNICs to be assigned factory MAC addresses, a random MAC addresses, or allow the administrator to specify a MAC address value.

14. Allow MAC clients to use hardware MAC address slots, when available.

15. Allow hardware resources (TX, RX, interrupts), to be assigned to MAC clients.

16. Allow a bandwidth limit, guarantee, or priority to be associated with a VNIC.

17. Allow a set of CPUs to be used for processing the network traffic of a VNIC, and if multiple CPUs are present, fanout receive traffic to this set of CPUs.

18. Allow the stack to poll traffic from a VNIC.

19. Allow inbound GLDv3 unicast checking to be bypassed for VNIC traffic.

20. Allow sub-flows to be defined on top of a VNIC.

21. Allow IPMP on pairs of VNICs assigned to zones or virtual machines. Allow MAC clients such as VNICs to be defined on top of link aggregations.

2 Network Virtualization Architecture Overview

This chapter describes the the GLDv3 architecture in general terms, and introduces the Crossbow network virtualization architecture, and its major components.

2.1 A Brief Introduction to GLDv3

Solaris 10 Update 1 introduced the GLDv3 (Generic LAN Driver v3) framework, a.k.a. Project Nemo. GLDv3 provides a simple but powerful interface that network drivers use to interface with the network stack. This interface is provided by the MAC layer of GLDv3. It is designed to allow the network drivers to focus on managing the hardware they control, and provide added-value such as link aggregation and VLAN capabilities transparently to the drivers.

Above the MAC layer, Nemo provides a DLS layer which is responsible to provide data-link services such as VLANs, on top of MAC drivers. Above the DLS layer can be found the DLD layer, which implements the logic needed to interface with IP and DLPI clients. The overall GLDv3 architecture is represented by Figure 2.1.

In the remaining of this chapter, we show how the GLDv3 framework can be used for network virtualization.

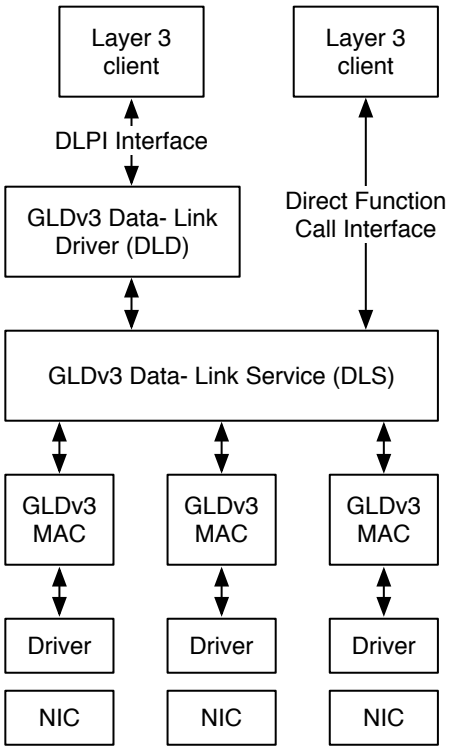

Figure 2.1: The GLDv3 architecture.

2.2 NIC Virtualization

As described in Section 2.1, at the bottom of the MAC layer, the GLDv3 framework provides to the drivers a simple function-based interface, also called the *GLDv3 driver interface*. At the top of the MAC layer, GLDv3 implements a *MAC client interface*. The MAC client interface can be used by kernel components which need access to a MAC device, without having to be concerned about the hardware implementation details of that interface. DLS as well as the aggregation driver are examples of MAC clients shipping with Solaris today.

The Crossbow network virtualization is implemented by changes to the MAC layer, and the introduction of a new *VNIC pseudo driver*. The VNIC pseudo driver allows the creation of virtual network devices, VNICs, which appear to the rest of the system as if they were regular NICs that can be plumbed, snooped on, have their own statistics, etc.

The MAC layer is modified by Crossbow to allow multiple MAC clients to access the underlying network hardware. Each MAC client is associated with a client handle and one or more MAC addresses. When a VNIC is created, the VNIC driver creates a new

MAC client which uses the MAC client interface to gain access to the underlying NIC for receiving and transmitting data. The VNIC allows specifies how the primary MAC address should be assigned to the VNIC, and optionally the value of that MAC address. IP can be plumbed on top of a NIC which is already accessed by a VNIC. In this case, the MAC address assigned to IP will be the primary MAC address of the underlying NIC.

The MAC layer therefore has knowledge of the various MAC clients which access the underlying physical NIC, their assigned MAC addresses, their registered callback functions, etc. When packets are sent or received by the underlying NIC or by one of the MAC client, the MAC layer is responsible to distribute packets to the various MAC clients or to the physical NIC.

Together, the VNIC driver and the MAC layer provide the services needed for the virtualization of network interface cards. The VNIC driver also exports an ioctl interface which is used by the libvnicadm library. libvnicadm maintains the persistent VNIC configuration information. libvnicadm is used by dladm(1M), the data-link management command line interface tool, to create, delete, modify, or show VNICs.

These various components are represented in Figure 2.2, and described in more details in the rest of this document.

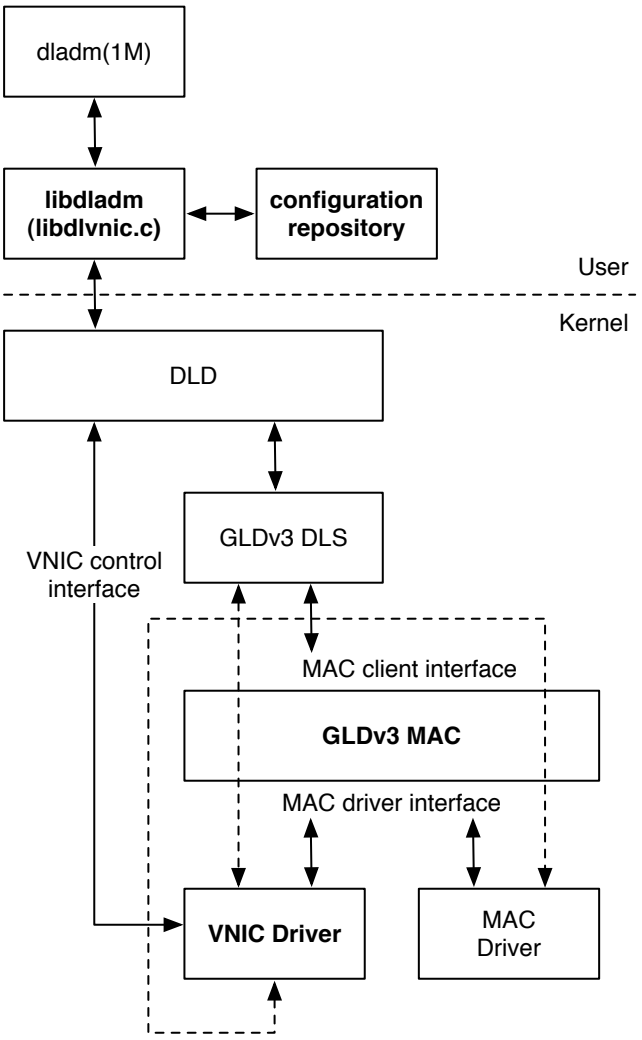

Figure 2.2: Network virtualization components.

3 Administration

VNICs are data-link objects managed through the dladm(1M) command line interface. The VNIC specific subcommands provided by dladm(1M) use the libdlvnic component of libdladm to manage VNICs. libdlvnic is a private library which maintains the persistent VNIC configuration, and provides interfaces with the VNIC driver to perform VNIC operations. These components are shown in Figure 2.2, and described in more details in this chapter.

3.1 VNIC Configuration with dladm(1M)

dladm(1M) was introduced as part of Project Nemo for the management of data-link interfaces. It is extended by Crossbow to create, delete, modify, and show VNICs. These sub-commands are summarized in this section and described in more details in the draft man page found in Section TBD.

3.1.1 create-vnic

```
create-vnic [-t] [-R <root-dir>] [-d <dev>]
    [-m <value> | auto | factory [-n <slot-identifier>]} |
    {random [-r <prefix>]}]
    [-p propname=value ...] <vnic-id>
```

Create a new VNIC with the specific <vnic-id> on the device specified by <dev>. Each VNIC must be assigned a VNIC id between 1 and 899. The VNIC ids ranging from 900 to 999 are reserved for automatic id selection, which is provided by libvnicadm for use by Xen[1]. The command will cause a new data-link of name vnic<vnic-id> to be created. That data-link can then be plumbed by IP, snooped on, etc. The following example creates and plumbs a VNIC for use by IP.

```
# dladm create-vnic -d bge0 1
# ifconfig vnic1 plumb
```

The -m option allows the administrator to specify the MAC address to be assigned to the VNIC. There are four ways to select a MAC address for a VNIC:

1 When the Nemo vanity naming project delivers, we expect that instead of specifying vnic-id, the administrator will specify a link name, and the id will be picked automatically.

- **Address by value**: The MAC address is specified by the administrator as the <value> argument of the -m option. For example -m 0:8:3:4:5:9. The administrator must provide a valid unicast MAC address. The length of the MAC address may vary depending on the MAC type of the underlying NIC. For ethernet it must be 48 bits long.

- **Factory address**: Some of the more advanced NICs such as Sun's Neptune adapter support multiple MAC addresses from the factory. If the “factory” keyword is specified as an argument of the -m option, the VNIC driver will attempt to reserve one of these factory MAC addresses and assign it to the VNIC being created. Optionally, the user can specify a specific slot number corresponding to the factory MAC address to be used. That slot number can be specified through the -n option. The list of factory MAC address slots can be displayed using the dladm(1M) show-dev -m option, see section 3.1.7.

- **Random MAC addresses**: If the “random” keyword is specified as an argument of the -m option, the MAC address is picked randomly by the VNIC driver. A default MAC address prefix consisting of a Sun IEEE OUI with the "local bit" set is used by default. The administrator can specify its own prefix with the -r option. The random MAC address is stored in the VNIC configuration repository. This allows the same MAC address to be assigned to a VNIC after a reboot of the host machine.

If the -m option is not specified, the VNIC driver will attempt to first reserve a factory address and assign it to the new VNIC. If the underlying hardware doesn't support factory MAC addresses, or all factory addresses have been exhausted, a random MAC address is picked by the VNIC driver.

The factory address slot number, random MAC address value, or specified MAC address values, are stored in the persistent VNIC configuration file to ensure that the same MAC address will be assigned to a VNIC after a restart of the host machine. Note also that a VNIC cannot be assigned the primary MAC address of the underlying NIC. This is to allow the underlying NIC to be plumbed by IP and simultaneously, as described in Chapter 5.

If the -d option is omitted, an anchor VNIC is created. Anchor VNICs allow fully virtual networks to be constructed, and are described in more details in Section 6.3.

The -p option allows the administrator to assign properties to the VNIC during a creation operation. The following properties are currently supported:

- **maxbw**: Assigns a maximum bandwidth limit with the VNIC. The bandwidth is specified by a value followed by a unit. Valid units are “k”, “m”, or “g”.

- **cpu_list**: Assigns a set of CPUs to the VNIC. If the VNIC has its own interrupt, it will be bound to one of these CPUs. The remaining CPUs will be used to process the traffic associated with the VNIC being created. This is done by mapping the soft rings kernel threads associated with the VNIC to the specified CPUs.

Once a VNIC is created, its properties can be changed using the set-linkprop dladm(1M) subcommand.

3.1.2 delete-vnic

```
delete-vnic [-t] [-R <root-dir>] <vnic-id>
```

Delete the VNIC specified by <vnic-id>.

3.1.3 modify-vnic

```
modify-vnic [-t] [-R <root-dir>] [-m <value> | auto |
    {factory [-n <slot-identifier>]} |
    {random [-r <prefix>]}] <vnic-id>
```

This subcommand allows the administrator to change the MAC address of an existing VNIC specified by `<vnic-id>`. The semantics of the -m option are the same as when used from the create-vnic subcommand described in Section 3.1.1.

3.1.4 move-vnic

```
move-vnic [-t] [-R <root-dir>] [-d <src-dev>] -D <dst-dev>
    [<vnic-id>]
```

Move a VNIC or a set of VNICs between devices or anchor VNICs. If the `<vnic-id>` argument is ommitted, the source device `<src-dev>` must be specified through the -d option, and all VNICs defined on top of `src-dev` will be moved to `<dst-dev>`. If a VNIC is explicitly specified by its `<vnic-id>`, the -d option should not be specified, and the source device is determined from the current VNIC configuration.

The move-vnic sub-command can be used to move all VNICs to another device or to an anchor VNIC if the underlying NIC must be unconfigured, for example as part of a DR (dynamic reconfiguration) operation.

If one of the VNICs being moved is using a factory MAC address of the underlying NIC, the operation will fail unless the -F option is specified. This option will cause such VNIC to be disassociated from the factory MAC address of the underlying VNIC, and the factory MAC address value will be assigned by value to the VNIC. Conversely, if -F is specified, and the destination NIC contains an available factory MAC address which is equal to the MAC address value of one of the VNICs being moved, the VNIC MAC address will be associated with that factory MAC address slot.

If one of the VNIC MAC addresses is already in use on the destination NIC, the move operation will fail.

3.1.5 show-vnic

```
show-vnic [-p] [-d <dev>] [-s [-i <interval>]] [<vnic-id>]
```

Shows information about the configured VNICs. If `<vnic-id>` is specified, displays the information about the corresponding VNIC, otherwise shows the information about every VNIC. If -d is specified, display the information about every VNIC defined on top of the specified device.

The -s option can be used to display the VNIC statistics. It can be combined with the -i option to display the statistics continuously at the specified interval.

3.1.6 up-vnic

```
up-vnic [<vnic-id>]
```

Brings up, or instantiates, the specified VNIC, or all VNICs of `<vnic-id>` is not specified, according to the persistent VNIC configuration. This command is invoked by the net-physical script when a global zone is booted. After the command is executed, configured VNICs will be available for plumbing by IP or use by other MAC clients.

3.1.7 show-link improvements

As shown in the previous sections, the VNIC can be assigned a factory MAC address when the underlying adapter supports that feature. A new -m option was added to the dladm(1M) show-link command and allows the administrator to obtain information related to the use of factory MAC addresses and MAC address slots of a data-link, in particular:

- The factory MAC addresses provided by the device, whether these MAC addresses are used or available.
- The number of MAC address filter slots available on the device, and how many are still available

```
# dladm show-dev -m nxge0
DEV             SLOT         MAC ADDRESS          INUSE
nxge0           <primary>    be:ef:11:22:33:44    yes
nxge0           1            be:ef:11:22:33:45    yes
nxge0           2            be:ef:11:22:33:46    no
FILTER_SLOTS_TOTAL: 16
FILTER_SLOTS_USED: 1
```

3.2 The VNIC Administration Library

The dladm library is currently used to implement data-link administration for all GLDv3 data-links, including link aggregation and wireless. The dladm library is further expanded to provide a management interface for VNICs. The VNIC specific functionality is implemented by the libdlvnic.c, and its interface is provided via libdlvnic.h.

The VNIC administration library provides a consolidation private API which can be used by consumers such as dladm(1M) to create, delete, modify, and query VNICs. It also maintains a persistent repository which allows VNIC configuration to be stored across reboots. The following sections describe the entry points provided by the VNIC administration library.

3.2.1 dladm_vnic_create()

```
dladm_status_t
dladm_vnic_create(uint_t vnic_id, char *dev_name,
```

```
    dladm_vnic_mac_addr_type_t mac_addr_type,
    uchar_t *mac_addr, int mac_len, int *mac_slot,
    uint_t mac_prefix_len,
    uint_t *vnic_id_out,
    uint32_t flags, const char *root)
```

Creates a VNIC with the specified `vnic_id` on top of the specified device `dev_name`. This function is invoked by the dladm(1M) create-vnic subcommand. The `mac_addr_type` specifies the type of MAC address to be assigned to the VNIC, and corresponds to the keywords allowed by the dladm(1M) create-vnic sub-command, see Section 3.1.1.

When a new random MAC address is requested, the desired prefix is passed through mac_addr, the prefix length through `mac_prefix_len`, and `mac_len` is set to zero. The generated MAC address is passed back through the `mac_addr` argument.

Since bandwidth limit, priority, level of parallelism, and CPUs are generic properties of data-links, they are set via the `dladm_{set, get}_prop()` calls of the libdladm library.

The following flags are supported:

- **`DLADM_VNIC_OPT_TEMP`**: The VNIC is created temporarily, and its configuration is not saved in the persistent repository. The VNIC will not be recreated automatically if the host is rebooted.

- **`DLADM_VNIC_OPT_AUTOID`**: The VNIC id is determined automatically, and return through the vnic_id_out argument.

- **`DLADM_VNIC_OPT_NODUPCHECK`**: The VNIC driver will not enforce that the VNIC MAC address is unique on the same physical NIC.

3.2.2 dladm_vnic_delete()

```
dladm_status_t
dladm_vnic_delete(uint_t vnic_id, uint32_t flags,
    const char *root);
```

Delete the specified VNIC. The `DLADM_VNIC_OPT_TEMP` flag can be specified and is described in Section 3.2.1.

3.2.3 dladm_vnic_modify()

```
dladm_status_t
dladm_vnic_modify(uint_t vnic_id, uint32_t modify_mask,
    dladm_vnic_mac_addr_type_t mac_addr_type,
    uint_t mac_len, uchar_t *mac_addr, uint_t mac_slot,
    uint32_t flags, const char *root)
```

Modify the attributes of the specified VNIC. The `DLADM_VNIC_OPT_TEMP` flag can be specified and is described in Section 3.2.1. The attributes to be modified are specified by `modify_mask`, and can be a combination of the following:

- **`DLADM_VNIC_MODIFY_ADDR`**: Change the MAC address of the VNIC. The new MAC address is specified by the `mac_addr_type`, `mac_len`, `mac_addr`, and `mac_slot`, which are described in Section 3.2.1.

As with dladm_vnic_create(), the bandwidth limit, priority, level of parallelism, and CPUs are set via the `dladm_{set, get}_prop()` calls of the libdladm library.

3.2.4 dladm_vnic_move()

```
dladm_status_t
dladm_vnic_move(int vnic_id, char *src_dev, char *dst_dev,
    uint32_t flags, const char *root)
```

Move one or move VNICs from between devices, aggregations, or anchor VNICs. The destination must be specified by `dst_dev`. If `vnic_id` is set to -1, all VNICs defined on top of the specified device `src_dev` are moved, and the source device, aggregation, or anchor VNIC must be specified by `src_dev`. Otherwise, the specified VNIC is moved, and the `src_dev` argument is ignored.

3.2.5 dladm_vnic_walk_sys()

```
vnicadm_status_t vnicadm_walk_sys(vnicadm_status_t
    (*fn)(void *, vnicadm_attr_t *), void *arg);
```

For each existing VNIC, invokes the specified `fn()` callback function. The first argument will be set to arg, and the second argument will contain a pointer to the attributes of the VNIC.

3.2.6 dladm_vnic_up()

```
vnicadm_status_t
vnicadm_up(uint_t vnic_id, const char *root);
```

Brings up the specified VNIC, or all VNICs if `vnic_id` is set set to 0. This function is invoked via dladm(1M) at boot time to configure all VNICs.

3.2.7 dladm_vnic_mac_addr_str_to_type()

```
static const char *
dladm_vnic_mac_addr_type_to_str(
    dladm_vnic_mac_addr_type_t type);
```

Returns a text string corresponding to the specified MAC address type.

3.3 VNIC Configuration Repository

The VNIC administration library maintains a persistent repository which captures the VNIC configuration. This information is used by libdlvnic.c to bring up all configured VNICs at boot time. The persistent repository is currently implemented as a flat text filed saved at /etc/dladm/vnic.conf. When project Clearview makes available a generic data-link repository, which could be implemented as a SMF repository, Crossbow VNICs will be updated to take advantage of that unified repository.

4 VNIC Driver

The VNIC driver allows the creation of multiple virtual MAC interfaces on top of a single physical NIC or link aggregation. These interfaces can then be plumbed, opened by MAC clients, and are associated with their own set of statistics, bandwidth control parameters, etc. The VNIC driver in tandem with the MAC layer to provide the virtualization of the underlying NIC. A description of the new MAC client interface provided by Crossbow and used by VNIC is described in details in Chapter 5.

4.1 Driver Architecture Overview

The VNIC driver is implemented as a pseudo driver. At the top, it registers multiple MACs with the MAC layer, and at the bottom, it uses the MAC client API to gain access to the underlying MAC. The VNIC driver also registers a set of ioctls with the GLDv3 framework. These ioctl roughly map to the VNIC management library entry points that are described in Section 4.8, and are used to create, delete, modify, and query VNICs.

The VNIC driver is responsible to create the virtual MAC devices that can then be subsequently plumbed by zones, or accessed by a virtual machine. This includes assigning a valid MAC address to the VNIC, and implementing the MAC driver entry points invoked by the MAC layer. The inter-VNIC data-paths and demultiplexing of data-traffic between VNICs, the physical interface, and a directly plumbed interface, are handled by the MAC layer.

The VNIC source can be found in the usr/src/uts/common/io/vnic directory of the ON consolidation, where vnic_ctl.c implements the control interface invoked by ioctls through dld, and vnic_dev.c implements entry points which are needed to expose VNIC MAC interfaces. The VNIC header files can be found in usr/src/uts/common/sys, where vnic.h contains definitions needed by the VNIC library administration library, and vnic_impl.h contains VNIC private definitions.

The main VNIC driver data-structure is the vnic_t. An instance of this data structure is created for each VNIC. It maintains the state of the VNIC, and its use by the various VNIC driver functionality is described in the following sections.

4.2 Access to Lower MAC, Pass-through

The VNIC driver accesses the underlying physical NIC or link aggregation through the MAC client interface which is described in Chapter 5. When a VNIC is created, a new instance of the `vnic_t` structure is allocated. The name of the underlying NIC is specified during VNIC creation, and it is used to open the MAC via `mac_open()` and `mac_client_open()`. Another important data structure of the VNIC driver is `vnic_mac_t`. Both MAC handle and MAC client handle are associated with a VNIC instance are saved in the corresponding `vnic_t`.

In order to simplify the VNIC and MAC layer architecture and for best performance, the VNIC driver implements a **pass-through** between its MAC client and the underlying NIC. In the following discussion, the MAC corresponding to the VNIC is refered to as the **upper MAC**, and the MAC corresponding to the underlying NIC is refered to as the **lower MAC**.

When the upper MAC client invokes `mac_client_open()` on a VNIC, the MAC layer detects that the MAC being open is a VNIC, and invokes the VNIC driver directly to retrieve the lower MAC client handle which was obtained by the VNIC driver and saved in the `vnic_t` when it opened the underlying NIC. That lower MAC client handle is returned to the MAC client of the VNIC directly. Any operation done by the upper MAC client of the VNIC will then be done directly on the lower MAC client corresponding to the VNIC. Note that a VNIC can only have one primary MAC client, and that primary MAC client maps to the lower MAC client associated with the VNIC.

Since the upper MAC client uses the MAC client interface to set its receive entry point and to transmit data, all of these operations will be done directly on the lower MAC client, allowing a complete bypass of the VNIC driver in the data path.

4.3 Hardware Resources

In order to achieve virtualization efficiently, Crossbow allows the assignment of one or more hardware rings to the MAC clients corresponding to VNICs or primary MAC clients (for example DLS). This section discusses how this sharing is done by Crossbow.

4.3.1 Rings and Classification for Virtualization

The unicast addresses associated with a MAC client are used to program the hardware classifier of the underlying NIC to steer traffic to the hardware receive rings associated with that client. This allows the hardware to do the heavy lifting of packet parsing and classification, and allows the host to simply pickup the packets from the hardware rings when they are available. Figure 4.1 shows an example where a physical NIC is shared by two VNICs and one primary MAC client. The figure shows the allocation of the hardware rings to these components.

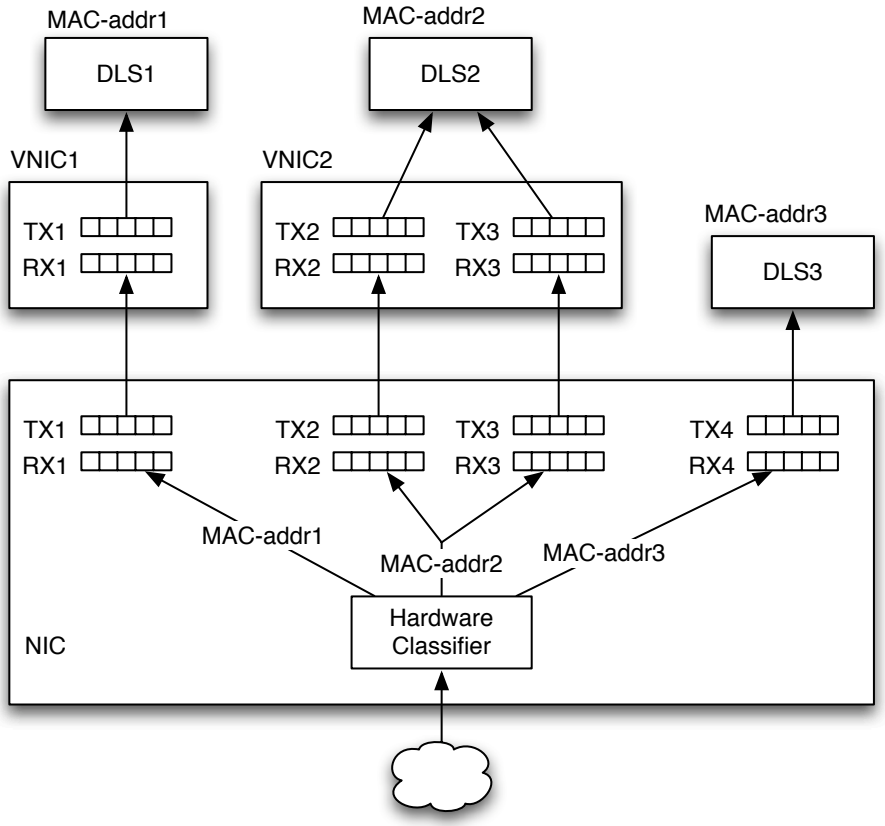

Figure 4.1: Assignment of hardware rings between VNICs and a primary MAC client.

When a driver registers with the MAC layer, it advertises its available hardware rings. The MAC layer distributes these hardware rings among the MAC clients and flows. The assignment of hardware rings to MAC clients and flows can change according to the number of MAC clients and flows accessing the underlying NIC. For example, when the primary MAC client opens the underlying NIC, it can be assigned all available transmit and receive rings, and the underlying classifier can be instructed to load balance traffic for the primary MAC address among these hardware rings. When a new MAC client opens the same underlying NIC, some of these hardware rings can then be reassigned to the new MAC client, and the primary MAC client is notified of this change.

When the hardware rings are exhausted, the MAC layer will created software rings and associate them to the MAC clients. This is done transparently to the MAC clients, i.e. a MAC client will see a set of rings, which can map to software rings, or hardware rings with no difference for the MAC client.

Since the upper MAC client of a VNIC maps directly to a lower MAC client of the underlying NIC, the assignment of resources and steering of traffic to the various rings can be done directly by the lower MAC layer in a centralized way.

The rings will be exposed to a MAC client asynchronously using the API described in Section 5.3.

4.3.2 Ring Selection

The use of rings for virtualization requires one or more rings to be assigned to a MAC client. There is a desire to make that selection as simple as possible, and let the MAC layer perform the allocation of hardware resources to MAC clients and flows. The MAC client API allows MAC clients to specify a hint consisting of the degree of parallelism needed to process the traffic associated their traffic, and optionally a set of CPUs to be used for that processing.

According to the level of parallelism specified, and the available of hardware rings, one or more hardware rings may be assigned to the MAC client. If the hardware does not have enough rings, multiple MAC clients may share a single hardware ring.

If the number of hardware rings does not allow the level of parallelism desired in hardware, the MAC layer will do software based fanout to dispatch received traffic to multiple processors, and in this case software rings will be exposed to the MAC client.

4.3.3 Hardware Fanout to Multiple Rings

As previously discussed in this section, one or more rings can be assigned with the MAC unicast address(es) of a client. When more than one ring is associated with a set of MAC address, the traffic associated with these addresses is fanned out by the hardware to the multiple rings.

4.4 Changes to Ring Allocation

The set of rings (software or hardware) exposed to a MAC client can change dynamically, for example after changing the desired level of parallelism, after the creation of another MAC client or flow which requires a repartitioning of the hardware resources. The API described in Section 5.3 allows the VNIC or primary MAC clients to be notified of these changes dynamically.

4.5 Polling and VNICs

Crossbow allows MAC clients to poll the rings for received traffic. Since each MAC client has its own set of rings, polling will be possible on the primary MAC client of a NIC as well as on VNICs. Since the MAC client of a VNIC maps to a lower MAC client, polling a VNIC ring will invoke the polling entry point of the lower MAC ring associated with the VNIC.

Polling a ring is done by invoking the polling entry point passed by the MAC layer to the MAC client during ring registration. That polling entry point must be invoked by the MAC client to poll the ring. The MAC layer knows how to get packets from the ring depending on whether the ring maps to a hardware ring, a software ring, whether bandwidth control is enabled, etc. The APIs used for advertising and polling rings are described in Section 5.3.

4.6 Statistics

The MAC layer maintains common per-MAC client statistics such as the number of packets sent and received, and the number of bytes sent and received. These statistics can be obtained through the `mac_client_stat_stat_get()` entry point, described in Section 5.2.10.

Other statistics returned by that entry point include the link duplex and link speed. The link duplex returned is the link duplex of the underlying NIC. The link speed is the minimum of the link speed of the underlying NIC, and the bandwidth limit associated with the VNIC, if one has been defined.

4.7 MAC Address

The VNIC driver determines a MAC address to associate with a VNIC according to the argument passed from the VNIC administration library. It then makes a call to `mac_unicast_add()` (see Section 5.2.8) to associate that address with the MAC client handle corresponding to the VNIC being created.

The MAC address value passed to mac_unicast_add() command can vary depending on the type of MAC address requested by the user.

- If the user specified a MAC address value, that address is used directly.
- If the user requested a factory MAC address, the VNIC driver will attempt to reserve that factory MAC address using the mac_addr_factory_reserve() call, described in Section 5.2.6.
- If the user requested a random MAC address, the VNIC driver will generate a new MAC address using the mac_addr_random() call, described in Section 5.2.7.

The primary MAC client of the VNIC itself will request the primary MAC address of the VNIC to be assigned to its `mac_client_handle_t`. The MAC layer will detect that the `mac_client_handle_t` corresponds to a VNIC, and will instead use the address value which was specified by the VNIC driver during VNIC creation.

If the user requested an automatic address assignment, the VNIC driver will first attempt to reserve a factory MAC address, and if none are available, it will generate a random MAC address value.

The MAC layer also makes use of the multiple MAC addresses capability offered by the underlying NIC, and puts the underlying in promiscuous mode when the MAC address slots are exhausted, or the underlying NIC does not support that capability.

4.8 Promiscuous Mode

When a VNIC is put in promiscuous mode, it should show the same traffic that would be shown by a physical NIC connected to a physical switch when it is put in promiscuous mode, which can be summarized as follows:

- The unicast traffic for the MAC address associated with the VNIC.
- All multicast and broadcast traffic sent by other VNICs, and a directly plumbed interface, if applicable, defined on top of the physical NIC.
- All multicast and broadcast traffic received by the underlying physical NIC.

The client of the VNIC uses the `mac_promisc_add()` function to put the VNIC to put the VNIC in promiscuous mode and define the callback function to which these packets should be delivered. The lower MAC is then responsible to dispatch copies of the packets to the promiscuous callbacks of the various MAC clients.

4.9 Bandwidth Attributes

Each VNIC can be assigned its own bandwidth limit and priority. Since Crossbow allows any GLDv3 MAC to be assigned its own bandwidth limit and priority, and VNICs are fundamentally MAC instances, VNICs take advantage of that common functionality to avoid code duplication.

4.10 Dynamic Reconfiguration

In the case of failure it might be necessary to service the physical NIC used by one or more VNICs. If the NIC to be serviced is part of an aggregation, it can be removed from the aggregation and serviced without impact to the configuration of the VNIC(s) defined on top of the aggregation.

If VNIC are defined on top of a device which needs to be serviced, these VNICs can be moved to another device, an anchor VNIC, or an aggregation using the move-vnic dladm(1M) subcommand.

4.11 High Availability

Since VNICs can be used as the primary network interface of a non-global zone or virtual machines, it should be able to leverage existing high availability technologies already available in Solaris from VNICs. In this section, we discuss how IEEE 802.3ad link aggregation and IPMP (IP Multipathing) can be combined with VNICs.

4.11.1 Link Aggregation and VNICs

The Solaris link aggregation implementation is compliant with IEEE 802.3ad and allows a set of MAC ethernet devices to be grouped together to form a highly available with higher throughput. Since link aggregations appear to the Solaris host as regular NICs, VNICs can be created on top of aggregations as they are created on top of physical devices or anchor VNICs. If one of the ports of the aggregation is detached or removed from an aggregation because or a hardware failure, or because it must be serviced, the VNICs will not be impacted, as long as there are remaining functioning ports in the aggregation.

Since virtualization is done at the MAC layer, the high availability and performance benefits of link aggregations can be made available to non-global zones and virtual machines completely transparently.

4.11.2 IPMP and VNICs

IPMP (IP Multipathing), as its name indicates, is implemented at the IP layer. It allows IP interfaces to be grouped, and provides fail-over capabilities between members of a group. IPMP is popular in environment which require failover between connections to multiple switches. Since IPMP is done in IP, it executes above the virtualization done by the MAC layer. For this reason, multiple VNICs need to be created on all physical NICs that need to be monitored by IPMP, and the resulting VNICs themselves need to be grouped by the non-global zone or virtual machines.

Since the MAC layer provides virtual switching semantics between VNICs, the inter-VNIC data-path is local, and VNICs defined on top of a NIC should have their connectivity preserved if the link of the underlying NIC goes down. For this reason, the link state of a VNIC is always up, since it reflects its connectivity to the virtual switch and other MAC clients on top of that virtual switch. This means that IPMP on top of VNIC must use probe-based detection failure.

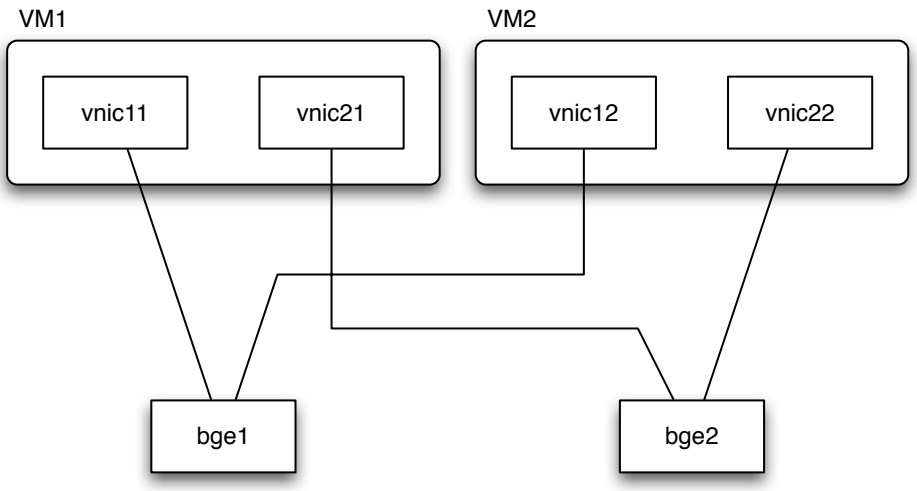

Figure 4.2: Assignment of VNICs for IPMP between VNICs from virtual machines.

When mapping this case to a physical equivalent, this is similar to having multiple hosts connected to a physical switch, and that switch connected to another switch or router on the network through a dedicated link. When that dedicated link goes down, the physical hosts continue to talk to each other, and IPMP running on these hosts needs to use probe based detection failure to detect problems on that dedicated link.

5 MAC Virtualization

The Crossbow network virtualization architecture is done partly by the VNIC driver, which exposes MAC devices which can be plumbed or used by MAC clients as if they were regular Solaris interfaces, and by the MAC layer, which allows a single NIC or aggregation to be shared by multiple MAC clients such as VNICs. The MAC layer implements the logic needed to classify and demultiplex outbound packets to the various MAC clients or to the hardware, distribute broadcast and multicast packets to interested parties, and send copies of packets to promiscuous listeners. The MAC layer also enables the path between a directly plumbed interface and VNICs defined on top of the same NIC.

5.1 Principles of Operation

Before Crossbow, GLDv3/Nemo provided a single handle per registered MAC instance. That handle was used by MAC clients as well as MAC providers to identify the MAC instance. In order to support multiple clients and the semantics needed by VNICs, it becomes necessary to identify the various MAC clients which access the same physical NIC. This is needed for example to avoid broadcast traffic sent by a MAC to be sent up as duplicate packets to the same MAC client. A primary MAC client allows IP to plumb the interface while the underlying NIC is in use by VNICs, and the MAC layer provides the data-path between that plumbed interface and the VNICs.

Crossbow extends the MAC client interface to allow multiple explicit MAC clients to share the same underlying NIC. Each MAC client is associated with its own set of unicast addresses and list of multicast addresses. Crossbow also provides a new transmit entry point which does not require the MAC clients to handle the change of transmit function pointers.

The new MAC client API is divided into two parts, a consolidation-private API, and a project-private API. Section 5.2 describes the **consolidation-private APIs** that are available to kernel modules of the ON consolidation, such as layered drivers, etc.

Section 5.3 describes the Crossbow **project-private MAC APIs** which are to be used exclusively by the Crossbow project, i.e. DLS, IP, and VNICs. The project-private APIs must not be used by any other kernel module in ON or other consolidations.

This distinction allows the MAC clients to be isolated from the details of the implementation, which will enable the Crossbow architecture and implementation to be

evolved in the future with minimal impact to MAC clients, and also allows the core framework to control the allocation of hardware resources to MAC clients and flows. The MAC clients can be hidden from these details and express high level requirements instead.

It is expected that in the long term, the MAC client consolidation private APIs will be promoted to be committed APIs available to any kernel module outside of the ON consolidation.

5.2 Consolidation-Private MAC API

This section describes in details the changes to the MAC layer API that are made by Crossbow in order to support network virtualization.

5.2.1 mac_open(), mac_close()

```
int mac_open(const char *macname, uint_t ddi_instance,
    mac_handle_t *mhp);
void mac_close(mac_handle_t);
```

Opens the specified MAC. Returns a handle uniquely identifying that MAC.

5.2.2 mac_client_open(), mac_client_close()

```
int mac_client_open(mac_handle_t *mh, char *name,
    mac_client_handle_t *mchp);
void mac_client_close(mac_client_handle_t mch);
```

`mac_client_open()` must be called by a MAC client before being able to send a receive data through a MAC. Each call to `mac_client_open()` returns a new opaque mac_client_handle_t which is associated with:

- one or more unicast MAC addresses, specified using `mac_addr_*()`
- a set of multicast addresses, specified using `mac_multicast_add()` and `mac_multicast_remove()`.

- optionally, a receive function callback, which is used by the MAC layer to deliver packets to the MAC client.

- optionally, a set of promiscous callback functions, which can be used by the MAC client to capture traffic associated with the `mac_client_handle_t`.

Typically, unique `mac_client_handle_t` will be associated with the primary interface associated with a MAC (e.g. through plumbing by IP), as well as each VNICs defined on top of the same MAC. The MAC layer is able to distribute received and sent traffic to the callbacks associated with the various `mac_client_handle_t`.

The MAC layer is also capable of identify the sender of a packet so that it doesn't send back a copy of broadcast packet to the receive entry point of the client which sends them. The client MAC handle specified by the transmit entry point is also used to ensure that packets sent by a MAC client are not passed to the promiscuous callbacks registered by the same client.

Each MAC client is also associated with a set of CPUs and network resource parameters, as described in Sections 5.2.3 and 5.2.4, respectively.

5.2.3 mac_cpu_set(), mac_cpu_get()

```
int mac_cpu_set(mac_client_handle_t mch,
    mac_bind_cpus_t *cpus, uint16_t flags);
void mac_cpu_get(mac_client_handle mch,
    mac_bind_cpus_t *cpus);
```

Sets or retrieves the list of CPUs used to process the traffic associated with the specified MAC client. The `cpus` argument allows MAC clients to specify the degree of parallelism, i.e. either a number of CPUs or a specific set of CPUs. The structure is defined as follows:

```
typedef struct mac_bind_cpus_s {
    uint_t     mbc_ncpus;
    uint32_t   *mbc_cpus;
} mac_bind_cpus_t;
```

`mbc_ncpus` defines the number of CPUs to associate with the MAC client. If `mbc_cpus` is NULL, the MAC layer will pick the CPUs. If `mbc_cpus` is non-NULL,

the MAC layer will use the CPUs specified in the array `mbc_cpus`. A CPU must be listed only once in the `mbc_cpus` array. If the CPU is removed from the system as part of a DR operation, the MAC layer will move any thread or interrupt bound to that CPU to a different CPU of the specified set.

The MAC layer also attempts to allocate a number of hardware rings which matches the number of CPUs requested. If there are still hardware rings available, but not as many as requested, the MAC layer allocates one hardware ring to the client, and performs fanout to software rings bound to the specified set of CPUs. If there are no rings available, a shared default ring is used, and the MAC layer performs fanout to software rings. The `flags` argument can be used by the MAC client can control the behavior of the `mac_cpu_bind()` call when the desired number of hardware rings cannot be allocated:

- If there are less than `mbc_ncpus` receive or transmit hardware rings available and MAC_FLAG_MULTI_RX_RINGS or MAC_FLAG_MULTI_TX_RINGS, respectively, are set, the call will fail instead of attempting to do fanout to software rings.

- If there are no hardware receive or transmit rings available and MAC_FLAG_ONE_RX_RING or MAC_FLAG_ONE_TX_RING, respectively, are set, the call will fail instead of attempting to allocate one hardware receive or transmit ring to the MAC client.

5.2.4 mac_set_resources()

```
void mac_resource_ctl_set(mac_client_handle_t mch,
    net_resource_ctl_t *resource_ctl);
```

This function allows network resource to be allocated[2] for a particular MAC client. The following type of resources can be specified:

- Bandwidth limit: set the NET_BANDWIDTH_LIMIT bit of the nr_mask mask of resource_ctl. The limit is specified through the nr_bw_limit flag of resource_ctl.

2 XXX Need to add priority

5.2.5 mac_exclusive_set(), mac_exclusive_clear()

```
int mac_exclusive_set(mac_client_handle_t mch);
void mac_exclusive_clear(mac_client_handle_t mch);
```

Sets the specific `mac_client_handle_t` to have exclusive access to the MAC. This does not prevent other clients to call `mac_open()` and `mac_client_open()`, however the MAC layer will ensure that only one MAC client can have exclusive access to the underlying MAC at a time[3].

5.2.6 mac_addr_factory_reserve(), mac_addr_factory_release()

```
int mac_addr_factory_reserve(mac_client_handle_t mch,
    int *slot, uchar_t *addr);
void mac_addr_factory_release(mac_client_handle_t mch,
    int slot);
```

Reserves one of the factory MAC addresses of the underlying NIC. If *slot is -1, an available factory MAC address is reserved, and the reserved slot is set in *slot before the call returns. If all slots are used, an ENOSPC value is returned. If a value different than -1 is specified, and the slot is already used, a EBUSY error is returned.

Upon successful execution, the value of the reserved MAC address is returned via the addr argument.

A MAC client must release its factory addresses before it closes its MAC client handle.

5.2.7 mac_addr_random()

```
int mac_addr_random(mac_client_handle_t mch,
    uint_t prefix_len, uchar_t *mac_addr);
```

Generates a random MAC address value. The MAC address prefix must be specified by the caller in the array pointed to by `mac_addr`, and its length must be specified by the `prefix_len`. The generated MAC address s returned in the array pointed to by `mac_addr`.

3 XXX Could this be done implicitly when adding the primary address to a MAC client?

5.2.8 mac_unicast_*()

```
mac_unicast_handle_t mac_unicast_add(
    mac_client_handle_t mch, mac_addr_type_t addr_type,
    uchar_t *mac_addr, uint32_t flags);
void mac_unicast_remove(mac_unicast_handle_t);
void mac_unicast_get(mac_unicast_handle_t mah,
    uchar_t *mac_addr);
void mac_unicast_update(mac_unicast_handle_t mah,
     uchar_t *mac_addr);
```

`mac_unicast_add()` adds a MAC address for use by a MAC client identified by mch. A MAC address must be set before the MAC client can send or receive traffic through the MAC. The MAC address can be specified in various ways according to the value of the `addr_type` argument:

- **`MAC_UNICAST_PRIMARY`**: Indicates that the primary MAC address of the NIC must be used. The value of the factory MAC address can be read and set by a MAC client using the `mac_unicast_get()` and `mac_unicast_update()` calls.[4]

- **`MAC_UNICAST_VALUE`**: Indicates that the MAC address is specified by value via the `mac_addr` argument.

If one is available, the MAC layer will allocate a MAC address slot to the address. This allows multiple unicast MAC addresses to be associated with the underlying NIC without requiring the NIC to be put in promiscuous mode. If no MAC address slots are available, and the `MAC_UNICAST_FLAG_HW` is not set, the NIC is put in promiscuous mode automatically by the MAC layer transparently for the MAC clients. When the NIC was put in promiscuous mode because of a shortage of MAC address slots, and a MAC address slot is freed, the slot will be assigned to one of the MAC addresses without slot. If after that operation all MAC addresses are assigned their own slot, the MAC layer will turn off promiscuous mode automatically.

Each MAC unicast address is associated with an opaque `mac_unicast_handle_t` which is allocated by the MAC layer. That handle is specified when performing an operation on a MAC address. The `mac_unicast_remove()` disassociates the specified MAC address from the MAC client. `mac_unicast_update()` allows an

4 Having a separate MAC address type for this address allows the MAC layer itself to react to changes of the primary MAC address of the underlying NIC, which simplifies the MAC client logic.

existing MAC address to be changed without being removed. The `mac_unicast_get()` entry returns the current value of the specified MAC address.

The following flags are supported by `mac_unicast_add()`:

- **`MAC_UNICAST_FLAG_HW`**: Causes the operation to fail if there are no available factory MAC address slots, instead of putting the underlying NIC in promiscuous mode.

5.2.9 mac_rx_set(), mac_rx_clear()

```
typedef void (*mac_rx_t)(void *arg,
    mac_resource_handle_t ring, mblk_t *mp);
int mac_rx_set(mac_client_handle_t mch, mac_rx_fn_t rx_fn,
    void *arg);
int mac_rx_clear(mac_client_handle_t mch);
```

Set the receive function to be invoked for the MAC client specified by mch. The client must have previously set a MAC address with `mac_addr_set()` before calling this function. The set of packets that will be received by the client are:

- packets for the unicast address specified by `mac_addr_set()`
- packets for multicast addresses assocated with the MAC (see Section 5.2.8.)
- packets for the broadcast address.

For each one of these packets, the specified receive function callback `rx_fn()` will be invoked with the specified argument and the chain of packets received.

When there is only one MAC client which uses the primary MAC address and snoop is not enabled, the `mac_rx()` calls from the driver, and the `mac_tx()` calls from the MAC client, are basically pass-through, i.e. no classification is done on the outbound or inbound paths. This allows us to have very efficient default data-path for the cases where only IP is accessing the interface.

`mac_rx_set()` automatically creates a flow entry for the unicast MAC address associated with the MAC client. It also creates a flow entry for the broadcast address of the underlying MAC type. If such an entry already exists, i.e. there are more than one

MAC clients which have set a MAC address and specified a receive callback, the specified callback function will be added to the list of callbacks associated with the flow.

In order to support multiple MAC clients interested in receiving traffic for the same broadcast and possibly multicast addresses, the flow structure supports multiple callbacks per flow. Each such callback will be given a copy of the packet, and is added to a flow during a call to `mac_rx_set()` or `mac_multicast_add()`. Each such flow callback contains the MAC handle of the client that created it. This is then used by `mac_tx()` to ensure that broadcast and multicast packets sent by a MAC client are not looped-back to their sender.

`mac_rx_clear()` unregisters the receive entry point associated with the specified MAC client. `mac_rx_set()` can be called either only after `mac_client_open()`, or after `mac_rx_clear()`.

The receive callback must be of type `mac_rx_t`. The first argument of that function is the value of `arg` passed to `mac_rx_set()`. The second argument is the ring which received the packet, that argument is currently project private. The last argument is a chain of the packets that have been received.

5.2.10 mac_client_stat_get()

```
uint64_t mac_client_stat_get(mac_client_handle_t mch,
    uint_t stat);
```

This function returns per MAC client, as opposed to per MAC instance statistics. The following statistics are tracked on a per MAC client basis:

- `MAC_STAT_LINK_STATE`: returns `LINK_STATE_UP` if there are multiple clients defined on top of the underlying NIC, or otherwise the link state of the underlying NIC.
- `MAC_STAT_LINK_UP`: returns whether the link state (as described by `MAC_STAT_LINK_STATE`) is equal to `LINK_STATE_UP`.
- `MAC_STAT_PROMISC`: returns whether the underlying NIC is in promiscuous mode.

- `MAC_STAT_IFSPEED`: returns the minimum of the underlying NIC speed and the bandwidth limit associated with the MAC client.

In addition, the MAC layer also returns statistics for the number of received and transmitted number of bytes, number of packets, multicast packets, and broadcast packets (`MAC_STAT_MULTIRCV`, `MAC_STAT_BRDCSTRCV`, `MAC_STAT_MULTIXMT`, `MAC_STAT_BRDCSTXMT`, `MAC_STAT_OBYTES`, `MAC_STAT_OPACKETS`, `MAC_STAT_OERRORS`, `MAC_STAT_IPACKETS`, `MAC_STAT_RBYTES`, `MAC_STAT_IERRORS`.)

The MAC layer will return the appropriate default values for all other statistics queried via this interface.

5.2.11 mac_multicast_add(), mac_multicast_remove()

```
int mac_multicast_add(mac_client_handle_t mch,
    const uchar_t *addr);
int mac_multicast_remove(mac_client_handle_t mch,
    const uchar_t *addr);
```

Add or remove a multicast address to or from a MAC client's list of multicast addresses. Packets for the specified multicast address will be sent to the receive callback specified by `mac_rx_set()`, see Section 5.2.9.

Internally, the MAC layer will create a new flow entry for the specified multicast address and add the receive callback of the client to the list of callbacks to the flow. If the same multicast address is used by multiple MAC clients, the corresponding flow entry will point to a list of callback functions.

5.2.12 mac_tx()

```
mblk_t *mac_tx(mac_client_handle_t *mch, mblk_t *mp,
    uint64_t hint);
```

Sends the specified packet chain from the MAC client associated with the specified handle. The function will send the packets to the appropriate destination as follows:

- Unicast packets which match a local MAC client will be looped-back to that MAC client, otherwise they will be send out through the underlying NIC.

- Broadcast packets will be sent through the underlying NIC, as well as to all MAC clients, except the sender MAC client.

- Multicast packets will be sent through the underlying NIC, as well as to all MAC clients which have added that multicast address, except the sender MAC client.

The MAC client handle passed as argument is used by `mac_tx()` to ensure that a broadcast or multicast packet is not looped-back to its sender. This is possible by ignoring flow callback entries which have the same MAC handle as the sender.

The `hint` argument allows the MAC client to give a hint for the selection of the transmit ring that will be used to send the chain of packets. The hint must be the same for packets of the same connection. If `hint` is set to `NULL`, `mac_tx()` will determine the outbound transmit ring according to the payload of the packets being transmitted.

5.2.13 mac_promisc_add(), mac_promisc_remove()

```
int mac_promisc_add(mac_client_handle_t mch,
    mac_promisc_type promisc_type,
    mac_promisc_fn_t promisc_fn, void *arg,
    mac_promisc_handle_t *php);
int mac_promisc_remove(mac_client_handle_t mch,
    mac_promisc_handle_t *ph);
```

Register a promiscous mode callback for the MAC client specified by mch. The `promisc_fn()` function will be given a copy sent and received packets according to the specified `promisc_type`:

- **`MAC_CLIENT_PROMISC_ALL`**: Deliver a copy of all packets, including packets sent and received between the MAC clients defined on top of the same underlying NIC.

- **`MAC_CLIENT_PROMISC_FILTERED`**: Deliver a copy of the unicast packets sent and received by the MAC client, as well as a copy of all multicast and broadcast packets sent and received on the wire and exchanged between MAC clients defined on top of the same NIC.

- **`MAC_CLIENT_PROMISC_MULTI`**: deliver a copy of multicast and broadcast packets only.

5.3 Project-Private MAC API

5.3.1 mac_resource_callback_set()

```
typedef mac_resource_handle_t (*mac_resource_add_t)
    (void *, mac_resource_t *);
typedef int (*mac_resource_bind_t)
    (void *, mac_resource_handle_t, processorid_t);
typedef void (*mac_resource_remove_t)
    (void *, mac_resource_handle_t);

void mac_resource_callback_set(mac_client_handle_t mch,
    mac_resource_add_t add_fn,
    mac_resource_remove_t remove_fn,
    mac_resource_bind_t bind_fn,
    void *client_arg);
```

This function is called by a MAC client to define the callbacks to be invoked by the MAC layer when the resources associated with that client have changed. Resources correspond to software or hardware rings. After setting its callback pointers points with `mac_resource_callback_set()`, a MAC client must enable polling by calling `mac_poll_enable()`, which is described in Section 5.3.2.

The specified `client_arg` argument passed to `mac_resource_callback_set()` is passed as the first argument of each callback. The callbacks supported are:

- `add_fn`: Invoked when a new resource is being added to the client. The resource description is passed as second argument, and described in more details below. The MAC client returns a `mac_resource_handle_t` opaque handle, which is later passed as an argument to the receive callback set by `mac_rx_set()`, see Section 5.2.9.

- `remove_fn`: Invoked when a resource is being removed from the client.

- `bind_fn`: Invoked when the binding of a resource to a CPU changes.

When a resource is advertised to a MAC client via the add_fn callback, it is described by a mac_resource_t data structure which contain the pointer to the polling function to be invoked by the MAC client, as well as the CPU id to which this resource is bound.

5.3.2 mac_poll_enable()

```
void mac_poll_enable(mac_client_handle_t mch);
```

Enables polling on the rings associated with the specified MAC client. This function causes the MAC resources to be advetised using the entry points defined by mac_resource_callback_set() which is described in Section 5.3.1.

6 Virtual Switching

6.1 Introduction

Crossbow's MAC virtualization and virtual NICs, also known as VNICs, allow physical NICs to be shared by multiple zones or virtual machines. VNICs appear to the rest of the system as regular NICs. The MAC layer allows a NIC to be accessed by multiple MAC clients. Each MAC clients correspond to either a primary MAC client (for example IP), or a VNIC, and can be can be assigned a subset of the hardware resources (interrupts, rings, etc) made available by the underlying hardware.

In order to provide connectivity between the multiple zones or virtual machines sharing a single physical NIC, the new MAC layer provided by Crossbow also provides a data-path between the MAC clients defined on top of the same underlying NIC. This data-path is needed since switches will not loop a packet back to its originating port.

The MAC clients sharing the same underlying NIC appear to be part of the same segment, i.e. connected to the same *virtual switch*. The mapping between physical and virtual network components is illustrated by Figure 6.1.

The following sections describe the concept of virtual switches, the semantics they implement, and how they can be used in practice.

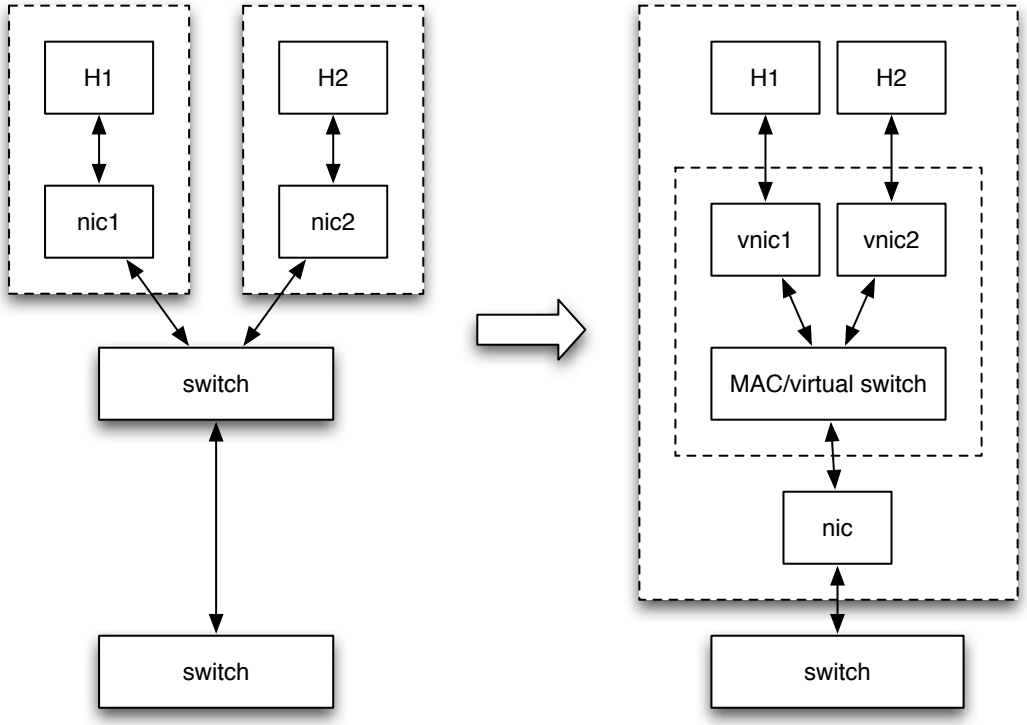

Figure 6.1: Mapping between switches and virtual switches.

6.2 Virtual Switching at the MAC layer

Virtual switches are never directly accessed or visible by the user or system administrator. Instead, they are implicitly created when the first MAC client is defined on top of a NIC. In this discussion, a MAC client corresponds to a VNIC, or DLS if the NIC is directly plumbed by IP. Virtual switches provide the same semantics expected by physical switches:

- MAC clients (for example VNICs) defined on top of the same underlying NIC can send ethernet packets to each other using their MAC addresses.

- Broadcast packets received by the underlying NIC are distributed to every MAC client defined on top of that NIC. Similarly, broadcast packets sent by one of the MAC client is sent to all MAC clients defined on top of the same NIC (except of course the originating MAC client), and to the underlying NIC for transmission. A broadcast packet is never sent back to the sender MAC client to avoid duplication of broadcast packets.

- Multicast group membership is tracked, and used to distribute multicast traffic to the appropriate MAC clients.

Connectivity is only enabled between MAC clients defined on top of the same underlying NIC. This is similar to having multiple hosts connected to multiple physical switches, as represented by Figure 6.2, which shows how VNIC MAC clients can be used to construct a virtual network based on the virtual switch.

Conceptually, each MAC client is part of a group, and MAC clients are part of the same group if they share the same underlying NIC, and the connectivity is allowed only between MAC clients that are part of the same group.

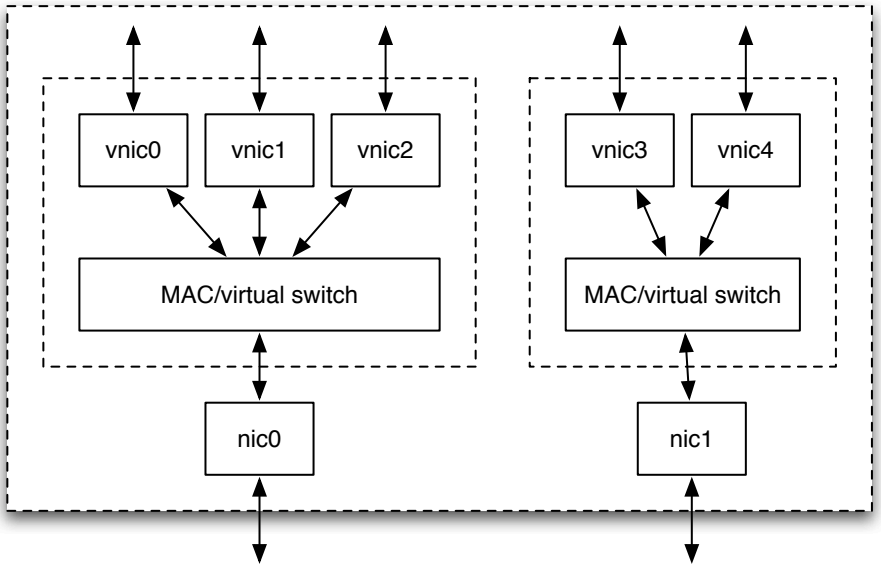

Figure 6.2: VNIC and virtual switches.

In practice, the virtual switch semantics are implemented by the transmit and receive entry points of the MAC layer, i.e. `mac_tx()` and `mac_rx()`. On the transmit path, classification is done from the MAC transmit entry point using the destination MAC address of the packet. The result of the classification is used to direct the traffic to the appropriate destination, which can be another MAC client defined on top of the same underlying NIC, or a group of MAC clients in the case of a multicast or broadcast packet. If there is no classification match on the outbound packet, the packet is for an external host and it is passed to the underlying NIC for transmission.

On the receive path, packets are classified by the hardware classifier of the underlying NIC, or by the MAC layer software classifier. After classification, the callback function associated with the matching entry is invoked. The callback is responsible to pass the packet to a single MAC client in the case of a unicast MAC address, or to a group of MAC clients in the case of a broadcast or multicast destination MAC address.

In order to simplify the administration of network virtualization, the MAC layer allows the underlying NIC to be plumbed while being used by MAC clients such as VNICs. For example, if a NIC is plumbed by the global zone, it is possible to create VNICs on top of the same NIC without unplumbing the NIC in the global zone. When such configuration is used, the plumbed interface is referred to as the primary MAC clients, and VNICs correspond to other MAC clients. From the point of view of the MAC framework, connectivity is provided between all MAC clients defined on top of the same NIC. This allows the plumbed interface to communicate with VNICs.

For example, a NIC can be used to create VNICs and assign them to non-global zones, while allowing the underlying NIC to be plumbed directly by the global zone, and connectivity is provided between the vNICs and the plumbed interface.

6.3 Anchor VNICs

As described in the previous sections, the MAC layer provides the virtual switching capabilities that allow the MAC clients such as VNICs to communicate with each other. The VNICs correspond to MAC clients which are always created on top of an underlying NIC, and these associations define the allowed local data paths between the VNICs. These local data paths provide the same semantics as a layer 2 switch to which the VNICs are virtually connected.

In some cases, it is desired to be able to create virtual networks on the same machine without the use of a hardware NIC. Some examples are described in section 6.4. Since a virtual switch is implicitly created when a VNIC is created on top of a NIC, one solution would be to have the ability to create pseudo NICs, and define VNICs on top of these pseudo NICs.

Since VNICs implement the behaviors of regular NICs, it is possible to implement such pseudo NICs as a special type of VNIC. These special VNICs are called *anchor VNICs*.

Anchor VNICs are created and deleted using the same interfaces as those used to manage regular VNICs.

VNICs can be created on top of either physical NICs, or anchor VNICs. When multiple VNICs are implemented on top of the same anchor VNIC, they can send traffic to each other through the virtual switch corresponding to the anchor VNIC. Anchor VNICs are different than regular VNICs in a number of ways:

- Anchor VNICs are not associated with an underlying NIC.
- Anchor VNICs do not send or receive data.
- Anchor VNICs do not have a MAC address.

Since anchor VNICs provide a subset of the VNIC functionality, they can be easily supported by the common VNIC layer. Figure 6.3 shows how VNICs (vnic0, vnic1, vnic2) can be connected to a virtual switch on top of an anchor VNIC (vnic5) without the need of dedicated hardware.

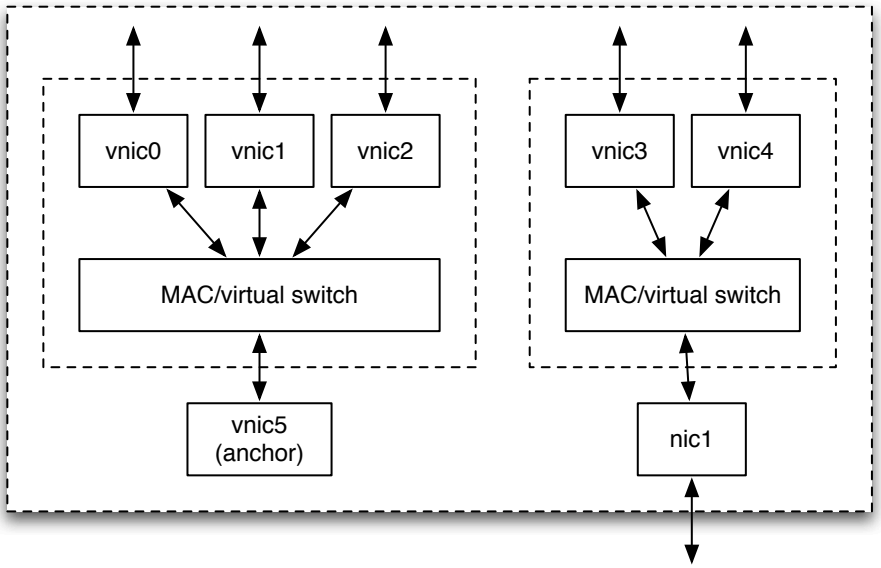

Figure 6.3: VNIC and virtual switches with an anchor VNIC.

6.4 Example

Figure 6.4 represents a zones environment where a virtual network is created between zone1-zone4. That virtual network is defined by creating VNICs vnic1-vnic4 on top of the same anchor VNIC vnic5. In addition, the physical NIC bge0 is assigned to zone1. With this configuration, the virtual network anchored by vnic5 is kept separate from the physical network. zone1 can use IPfilter, NAT, and DHCP to assign the addresses to vnic2-vnic4, and allow the permitted traffic to be exchanged between zone2-zone4 and the physical network attached to bge0.

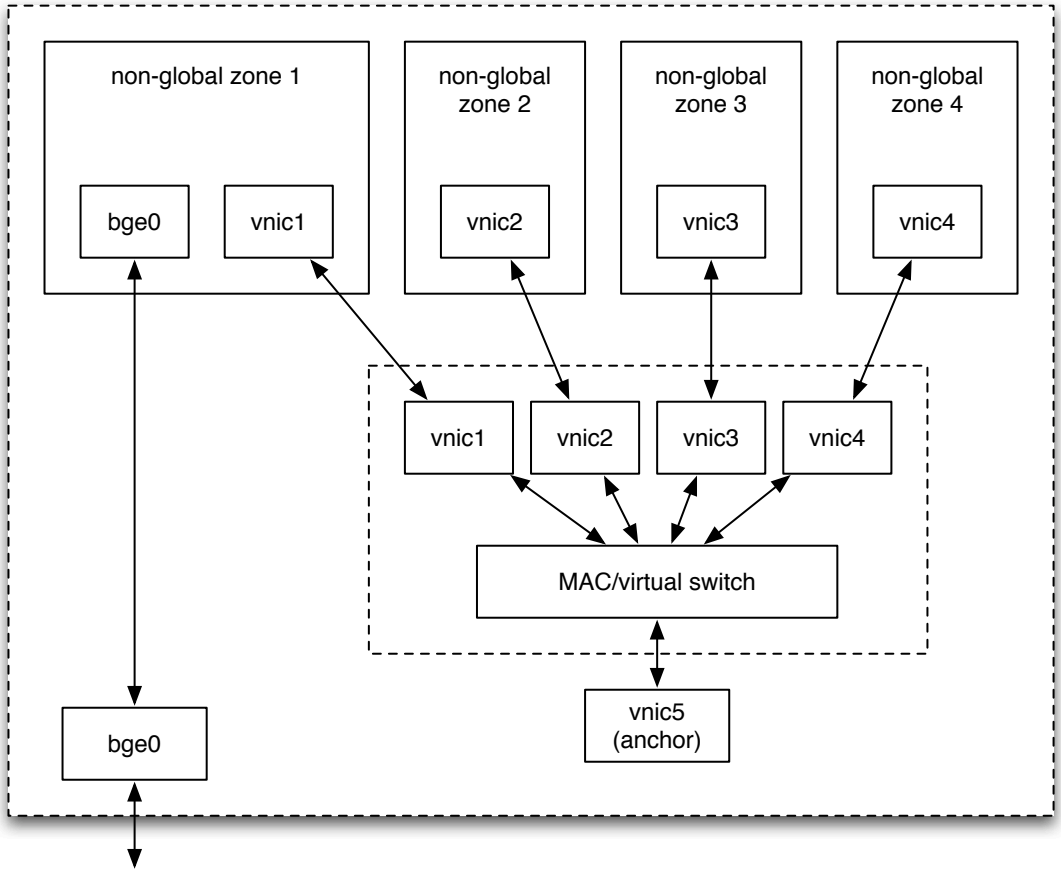

Figure 6.4: VNIC and virtual switches with an anchor VNIC.