Vanity Naming and Nemo Unification

High-Level Design Specification

Cathy Zhou & Daniel Groves

Project Clearview I-Team
clearview-iteam@sun.com

Network Approachability
Sun Microsystems, Inc.

Revision 1.6
Jul. 11, 2007

Contents

1 Introduction

This component of the Clearview project will provide a consistent administrative model for all network interfaces, and will also introduce the ability to name each individual network interface.

Specifically, while Nemo, aka GLDv3, provides a new network driver framework with enhanced features such as link aggregation, it also introduces a confusing administrative model, since only Nemo-based devices can be managed by `dladm(1M)`. For instance, only Nemo-based devices can be aggregated by `dladm create-aggr`.

Functionally, this component will also address another inconsistency of the existing network interface administrative model: currently, only a small set of network devices have Ethernet VLAN support. This component will address this by introducing a mechanism that transparently adds VLAN support to any deficient Ethernet driver.

Another long-standing issue with current network interface administration is the inability to choose network interface names. Today, the network interface names are tied to the underlying network hardware (e.g., `bge0`, `ce0`). Because configuring the system always requires network interface names to be stored in a wide range of configuration files, being able to give a meaningful and consistent *vanity name* to network interfaces will make network administration much easier. More importantly, vanity naming will prove especially useful for machine migration, Zones migration and Dynamic Reconfiguration.

2 Requirements Analysis

The requirements of this component are:

1. Must provide a consistent model for network interface administration:

- *All* network interfaces must be administrated by the same set of commands.
- *All* network interfaces must support *all* features that specific hardware can support. In particular, *all* Ethernet network devices must have aggregation and VLAN support[1].

2. Must be able to configure a vanity name for any network interface:

- The administrator must be able to name network interfaces, and also be able to administrate the interfaces based on their vanity names.
- Must provide a vanity-name DLPI device node for each network interface, allowing applications to interact with interfaces by their vanity names.

3. Must preserve backward compatibility:

- Must preserve the existing syntax of network administrative commands, and configuration files. In addition, administrative scripts must not need to be changed.
- Must not require any application code change in order to use existing DLPI device nodes (e.g. `/dev/bge`).
- Must minimize application code changes in order to use vanity-name DLPI device nodes.

4. Must not degrade network performance for data fast-path:

- This component must not have any measurable negative impact on network performance for data fast-path. We will run benchmark tests such as `netperf` and `specweb99` to make sure we meet this requirement.

5. Must not require changes to legacy network device drivers[2].

1 There are a few exceptions due to the implementation restriction, which will be discussed respectively in section 6.1.4 and 6.1.5.

2 Legacy devices are physical devices whose drivers are not written to the Nemo framework. This component will require some changes to Nemo network device drivers. However, as Nemo is not a published interface, it is not a key issue.

3 Conceptual Overview

3.1 Vanity Naming

3.1.1 Vanity Naming Fundamentals

The vanity naming component of the Clearview project will allow administrators to give interfaces meaningful names, separating the system's network configuration from the system's actual networking hardware. For example, the administrator will be able to assign the `bge0` interface the name `net0`, and then specify other network configuration such as firewall policy based on `net0`, or even create other interfaces such as aggregations over `net0`. Therefore, the `bge0` hardware can be easily replaced with other compatible hardware (for example `ce0`) by simply inserting `ce0` and assigning it the name `net0` — no changes to the existing network configuration will be needed.

As shown from the example, the goal of vanity naming is to be able to insulate the network configuration from changes to the underlying networking hardware. While vanity naming can also be used to rename links for other reasons, considerable follow-on work will be needed to allow such renames to be performed seamlessly.

Specifically, while this component will ensure that the interface name will be consistent across interface's link configuration, updating network interface names embedded in other network configuration (such as IP `/etc/hostname.*` files, firewall policy) is a broader issue which must be addressed by a tool and associated framework that operates at all levels of the networking stack. Providing such a framework to support tools to update interfaces names across all configuration on the system is out of scope of this component. However, the vanity naming component will be a part of such framework and will take care of updating all link configuration associated with the renamed interface.

Further, while administrators will be able to name most network interfaces in Solaris, including physical network devices, VLANs, aggregations and IP tunnels, this component will not support the vanity naming for network interfaces that do not exist below the IP layer, such as `vni`, `lo`, and `cgtp` interfaces.

3.1.2 Interface Naming Convention

Before proceeding, one must first understand the current network interface naming model, and the naming conventions used in this document.

Specifically, from the perspective of an administrator, each network interface has a *link name*. A *link* is an entity that represents a data-link object at layer 2 of the OSI model. The link name is shown through `dladm show-link`, and exposed in the `/dev` namespace, therefore applications like `ifconfig` or `snoop` can access the link by opening its link name in `/dev`[3].

A *physical link* is a link directly associated with physical hardware. Other than the link name, each physical link also has a device name which can be shown through `dladm show-dev` and passed via the `-d` option to various `dladm` subcommands. The *device name* is essentially the *device instance name* which is composed of the driver name and the device instance number (e.g., `bge0`). Today, the link name of a physical link is the same as its device name.

Interface vanity naming refers to the ability to rename the interface's link name. The device name, by definition, will not be able to be changed.

In order to simplify the administrative model, in the future, administrators will be able to do all the `dladm` operations using link names. As such, the `dladm show-dev` subcommand and the `-d <dev>` option will be marked as obsolete (section 4).

3 Strictly speaking, not all network links have DLPI `/dev` node names matching their links names. Some network links only have DLPI style-2 nodes, which applications must first open, and then attach to the appropriate PPA, in order to use.

No matter how we present the interface naming model to the administrator, in the kernel, each link will be identified using *four* means:

- A data-link name[4]

  The *data-link name* identifies a `dls_vlan_t` structure representing a unique data-link in the kernel. The data-link name is the same as the name displayed by `dladm show-link`.

  The data-link name is also the name of the data-link at the DLPI layer, and, when plumbed, the name of its associated IP interface. Thus, the data-link name is the name used in the majority of administrative interactions.

- A linkid

  Because data-link names must change when administrators rename links, in the future, each kernel data-link structure will be uniquely identified by a linkid which will not be changed during the link's lifetime — even across reboots. Further, all link configuration and GLD kernel structures will be updated to refer to linkids so that they will not need to be updated when the link name is changed.

- A device instance name

  The *device instance name* identifies a `dev_info_t` structure (or a `dip`) representing a unique device in the kernel. Each physical device has an associated `dev_info_t`, whose name is visible via `dladm show-dev`. Pseudo-devices also have associated `dev_info_t` structures, though typically a single `dev_info_t` is used to represent the entire pseudo-device since it is not tied to any actual physical hardware. For instance, the aggregation driver has a single `aggr0` `dev_info_t` for all of its links. Note that `dladm show-dev` only shows physical devices, and thus does not display pseudo-device instance names such as `aggr0`.

- A MAC name

  In Nemo, the `mac_impl_t` structure represents an individual unit that may be used to send and receive raw packets. For physical devices, there is one `mac_impl_t` for each device instance. For pseudo-devices, there is one `mac_impl_t` for each network link configured over the pseudo-device. For example, a system with two aggregations will have a single `aggr0` pseudo-device instance, but two aggregation `mac_impl_t` data structures. Because pseudo-devices may have only one device instance name but multiple `mac_impl_t`'s, the device instance name cannot be used to identify a `mac_impl_t`. Thus, each `mac_impl_t` is uniquely identified by a *MAC name* (e.g., `bge0` and `aggr100`). MAC names are never exposed to administrators, and are currently only used by kernel MAC clients. For instance, the aggregation MAC client aggregates several `mac_impl_t`'s into a single aggregation `mac_impl_t`.

  Today, for physical devices, their MAC names are the same as their device instance names; for pseudo-devices, the driver is responsible for mapping a MAC name to a given network link (e.g., the `aggr` driver uses `aggr`*`key`* as the aggregation's MAC name). Note this naming convention will be changed as part of this component. Although a MAC name is still able to uniquely identify a MAC, there will be no direct connection between a MAC name and a device instance name. As MAC names will become unpredictable, a kernel MAC client will have to first convert a link name or a linkid to the MAC name, then use the MAC name to access MAC services (see 7.2.4).

Note that although a linkid always maps to a single MAC, the reverse is not true: for some MACs, the packets that flow over them can be further demultiplexed into multiple data-links (subnetworks) based on the packet characteristics. Currently, the only demultiplexing approach is "VLAN", which demultiplexes packets into multiple VLAN subnetworks based

4 For clarity, this document always refers to the name of a link in the administrative model as the link name and the name of a `dls_vlan_t` in the kernel as the data-link name. Because a link's link name and data-link name are the same as each other, they can be used interchangeably.

on the VLAN ID. Therefore, the MAC name and the subnetwork identifier (VLAN ID) are necessary information to identify a specific subnetwork, which we call a "*subnetwork point of attachment*" (SPA). There is one-to-one mapping between linkids and SPAs[5].

3.1.3 Vanity Naming Rules and Constraints

- All links will have vanity naming support.
- Link names follow the current style-1 DLPI provider naming convention. That is: a composition of the driver name obeying the constraints identified in the `NOTES` section of `dlpi(7P)` and a PPA number. Note that link names can be any valid style-1 DLPI provider name.
- Each link must have only one vanity name at one time.
- Each link must have a unique vanity name.
- For backward compatibility, each physical network device will start with a link name the same as its well-known name, for example, `bge0` or `ce1`[6]. The administrator will be able to rename it later.

3.1.4 Vanity Naming Administrative Model

The `dladm(1M)` command was introduced by Nemo to administrate network link attributes at the data-link layer. Because vanity naming provides the ability to rename a link's link name, new `dladm` subcommands will be introduced, and existing `dladm` subcommands will be enhanced.

Thus, `dladm` will be used to specify a link name when the link is created[7], to rename the link's name, or to issue link operations[8] based on its link name. VLANs and virtual links (aggregations and IP tunnels) will also be created over links by specifying their link names[9].

For example, currently, aggregations are created over device names rather than link names:

```
# dladm create-aggr -d bge0 -d bge1 100
```

In the future, the `-d` option will be marked as obsolete and an aggregation will be able to be created over link names by specifying the `-l` option:

```
# dladm rename-link bge0 foo0
# dladm rename-link bge1 bar0
# dladm create-aggr -l foo0 -l bar0 100
```

In the above example, the administrator first renames the link names using `dladm rename-link`, then creates the aggregation over the two links using their link names.

Specifying link names in the `dladm` subcommands makes network administration consistent. However, the flexibility also means that some invalid commands will be able to be specified, such as attempting to create an aggregation over a VLAN. Therefore, the system will check each configuration request and fail the operation if the request is not valid for the specified link.

Further, all network administrative commands (such as `ifconfig(1M)`, `snoop(1M)`) and configuration files (such as `/etc/hostname.*`) will make use of link names. Specifically, because IP interfaces will now be created with link names, the link name will be propagated up to the IP administrative and programmatic interfaces. Thus, this allows administrators and applications to shield themselves from the hardware characteristics of network interfaces on the system.

---

5 Because VLAN is currently the only way to subclassify a MAC into multiple data-links, the only supported SPA will be the [MAC name, VLAN ID] pair. The design does not prohibit new SPA forms if new demultiplexing approaches are later introduced into Solaris.

6 If a physical link's link name conflicts with an existing link name, the system will choose another link name (section 6.2.5).

7 For links that will be created using `dladm`. Physical links will be created by the system and will then be able to be renamed using `dladm`, as previously described.

8 For example, administrators will be able to modify, display, and destroy links by specifying their link name.

9 The `dladm` subcommands are summarized in section 4.

Vanity Naming and autopush

The `autopush(1M)` command is used to set up the `autopush` configuration for a stream device based on the device's major number (or the device driver name) and a range of the minor numbers. When the device is opened, the kernel automatically pushes a pre-configured list of modules onto the stream. Currently, `autopush` settings are tightly tied to the physical device name of the underlying network hardware (e.g., `bge0`). Further, because the network devices are self-cloning, currently the `autopush` configuration is actually set *per-driver*, rather than based on the device instance (as it appears to be).

In the future, the administrator will be able to configure `autopush` based on the link's name (*per-link* `autopush` configuration) using the existing `dladm set-linkprop` subcommand with the `-p autopush` option (section 4.3.4), separating `autopush` configuration from the physical device name. For example, the administrator could tell the system to push the `vpnmod` module onto a link named `net0`. The `net0` name could then be associated with a `bge0` device when at home, or a wireless device `ath0` when on the road. The administrator only needs to rename either `bge0` or `ath0` to `net0` without changing the `autopush` configuration.

Note that the new `dladm autopush` mechanism will coexist with the old `autopush(1M)` command, but they will operate on different namespaces: `autopush(1M)` will continue to configure modules based on a device's major and minor numbers, whereas `dladm autopush` will configure modules based on a link's name. When a link is opened, the per-link `autopush` configuration is always preferred, and thus if it exists, any per-driver `autopush` configuration will be ignored. Otherwise, the per-driver `autopush` configuration will still apply (section 6.1.3).

Vanity Naming and Dynamic Reconfiguration (DR)

To better understand the impact of vanity naming, let's examine one particularly complicated area: Dynamic Reconfiguration. Below is an example of a typical network configuration today:

```
# cat /etc/hostname.bge0
myhost netmask 255.255.255.0 up
# cat /etc/iu.ap
bge     -1      0       vpnmod
# ipfstat -io
empty list for ipfilter(out)
block in on bge0 proto tcp from 129.138.34.1/32 to any
# ifconfig bge0
bge0: flags=1004843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
       inet 129.146.56.144 netmask ffffff00 broadcast 129.146.56.255
       ether 0:3:ba:50:9a:d
```

To keep this example understandable, only the configuration of one network interface (`bge0`) is shown. The configuration includes `hostname.bge0`, the `ipfilter(5)` rules and the `autopush(1M)` rules applied to the system. Let's assume we need to remove `bge0` and replace it with `ce0` via DR. The following lists the current configuration steps of the DR process in order to preserve the existing configuration[10] (assume `ap_id`[11] is `PCI8`):

10 Note that this might not be a complete list. Ensuring that we have updated all configuration that refers to `bge0` is problematic.
11 See `cfgadm(1M)`.

```
# cfgadm -c disconnect PCI8
remove bge0 and insert ce0
# mv /etc/hostname.bge0 /etc/hostname.ce0
# ipf -Fa
# echo "block in on ce0 proto tcp from 129.138.34.1/32 to any" | ipf -f -
# vi /etc/iu.ap
update bge0 to be ce0
# cat /etc/iu.ap
ce      -1      0       vpnmod
# cfgadm -c configure PCI8
```

In the future, if the link is given a meaningful name, say `net0`, the DR process will be much simpler. First, the configuration before DR will be set up like the following:

```
# dladm rename-link bge0 net0
# dladm set-linkprop -p autopush=vpnmod net0
# cat /etc/hostname.net0
myhost netmask 255.255.255.0 up
# ifconfig net0
net0: flags=1004843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
      inet 129.146.56.144 netmask ffffff00 broadcast 129.146.56.255
      ether 0:3:ba:50:9a:d
# ipfstat -io
empty list for ipfilter(out)
block in on net0 proto tcp from 129.138.34.1/32 to any
```

After the configuration is setup, the DR process will be simpler: the administrator will only need to rename the newly installed hardware to `net0` (section 6.2.7):

```
# cfgadm -c disconnect PCI8
remove bge0 and insert ce0
# cfgadm -c configure PCI8
# dladm rename-link ce0 net0
```

Note that the order of the `cfgadm` and `rename-link` operations will not matter, i.e., following configuration steps will achieve the same result:

```
# cfgadm -c disconnect PCI8
remove bge0 and insert ce0
# dladm rename-link ce0 net0
# cfgadm -c configure PCI8
```

Alternatively, if `bge0` is removed but the administrator does not expect any new device to inherit its configuration, the administrator will be able to explicitly delete all the link configuration associated with `bge0` using `dladm delete-phys` (section 4.3.5), permitting the name `net0` to be reused (section 6.2.7):

```
# dladm delete-phys net0
```

3.1.5 Vanity-name Device Node

Today, applications operate on a network data-link by opening its associated `/dev` node and issuing DLPI primitives. The `/dev` node is symbolically linked to the physical path of the device under `/devices`, which in turn contains the major and minor number needed to open the appropriate device driver.

In order to avoid namespace pollution in `/dev` and also to avoid a possible namespace collision with the existing `/dev` nodes, we will introduce a new `/dev/net` directory to

place vanity-name nodes: in the future, each link will have a DLPI style-1 `/dev/net` node with the same name as its link name. Applications can access the link by its link name using the `/dev/net` node. Note that except for opening the `/dev/net` node instead of the `/dev` node, all of the other DLPI operations (such as `DL_INFO_REQ` or `DL_BIND_REQ`) will be the same as today.

For each physical link on the system, a /dev/net node will be created automatically with its well-known name (for example, bge0). The administrator will be able to rename the link using the dladm rename-link subcommand:

```
# ls -l /dev/net
crw-------   1 root     sys       99,   1 Aug 31 16:20 bge0
# dladm rename-link bge0 net0
# ls -l /dev/net
crw-------   1 root     sys       99,   1 Aug 31 16:20 net0
```

The libdlpi[12] library will also be changed to open the /dev/net node first, and if that fails, fall back to opening the /dev node as it does today, so that the namespace change will be transparent to applications that use libdlpi (for example, ifconfig).

3.1.6 Network Link Enumeration

Today, applications enumerate network links on the system by traversing the DDI_NT_NET nodes in the devinfo tree (see di_walk_minor(3DEVINFO)). Unfortunately, some applications assume that the link name is the same as the device instance name, and therefore derive link names directly from the DDI_NT_NET node information. After vanity naming is introduced, a link name will not be tied to the underlying hardware. Therefore, applications may behave unexpectedly when vanity names are used.

While di_walk_minor(3DEVINFO) is a documented interface, the above assumptions made by the applications are not. However, to mitigate the risk, a public libdlpi function dlpi_walk() will be provided to enumerate network links on the system:

```
typedef boolean_t dlpi_walkfunc_t(const char *, void *);
void dlpi_walk(dlpi_walkfunc_t *func, void *arg, uint_t flags);
```

The dlpi_walk() function will call dladm_walk() (section 5.2.1) to enumerate all network links on the system. One will also be able to set the flags parameter to DLPI_DEVIPNET to enumerate all the /dev/ipnet nodes on the system (see PSARC 2006/475).

Note that some applications use di_walk_minor(DDI_NT_NET) to enumerate network devices rather than network links, therefore we cannot blindly change all places calling di_walk_minor(DDI_NT_NET) to call dlpi_walk() instead. To detect possible incorrect usage, we could choose a "break it a lot" approach and change di_walk_minor(DDI_NT_NET) to return failure, forcing applications to to call di_walk_minor() with a new argument (say DDI_NT_NET_DEV) to enumerate network devices. Therefore, problems in such applications will show up immediately, forcing the developer to choose either dlpi_walk() or di_walk_minor(DDI_NT_NET_DEV).

At this point, we do not have enough data on how di_walk_minor(DDI_NT_NET) is used, in order to determine whether the “break it a lot” approach is worthwhile. Therefore, we will not use that approach for now, and will only examine the impact. Because the default name of physical links will not be changed, the issue described above will not introduce any regressions if vanity naming is not used. Once we have enough data and change applications to call the correct function, we could safely change a physical link's default name to be completely disassociated from its hardware (for example, of the form of net*N*).

3.1.7 Backward Compatibility

- /dev nodes

  In order to preserve administrative expectations and provide backward compatibility for applications that cannot access the /dev/net node, we will preserve the legacy /dev

12 As part of Nemo, libdlpi was introduced to centralize all DLPI operations performed by applications. It has been made public as a part of the Clearview project (PSARC 2006/436).

nodes for all current and future physical devices. Further, because aggregations also have `/dev` nodes, these will be preserved as well[13]. Therefore, while the `/dev/net` directory will list a DLPI style-1 node with its link name for every network link on the system, (including physical links - Nemo or legacy, VLANs, aggregations and IP Tunnels), the `/dev` directory will appear the same as today: physical links and aggregations will have their DLPI style-1 or/and style-2 nodes (depends on the driver) but only with their legacy names.

- VLAN access

  Backward compatibility will also be provided to access a VLAN by its "PPA hack"[14] name. A VLAN `/dev/net` node will be implicitly created when an application attempts to open it using the VLAN PPA hack name (e.g., `/dev/net/bge5001`) and will be deleted when the VLAN is no longer in use by any applications (section 6.2.3).

- `autopush`

  We will make sure that the `autopush` modules on a device configured by the traditional `autopush` syntax will be pushed automatically when opening a link, if the link is associated with that device. As an example:

```
# dladm show-phys
LINK      STATE   SPEED   DUPLEX   DEVICE
net0      up      1000    full     ce0
# cat /etc/iu.ap
# major      minor  lastminor    modules
  ce         -1     0            vpnmod
```

  When an application opens `/dev/net/net0`, the `vpnmod` module will be pushed automatically(section 6.1.3).

3.2 Nemo Unification

One limitation of `dladm` today is that it only configures links for network drivers written to the Nemo framework, leading to a frustrating administrative experience. This component of Clearview will alleviate this problem by allowing *all* network links to be administrated with `dladm`. This work is called *Nemo unification*.

In particular, today:

- Many legacy Ethernet devices simply do not have VLAN support.
- Many legacy Ethernet devices do not have a bundled link aggregation support[15].

Nemo unification will fix both problems. It will provide a generic solution for all Ethernet devices to have both VLAN and link aggregation support, without the need to change the device drivers. In fact, after Nemo unification, features or performance improvements Nemo provides today or in the future will automatically be applied to all compatible network links.

Table 1 shows the future functionality matrix for each type of network link[16]:

13 Note that as we will not be able to determine the legacy name of an aggregation created using a vanity name, only aggregations created using the "`-k`" option will have their `/dev/` nodes (in the form of `aggr<key>`).

14 Today, applications access a VLAN by opening a style-2 DLPI device node and attaching to [VLAN ID*1000+PPA]. This is called the VLAN PPA hack. For example, an application accesses VLAN 5 over `bge1` by opening `bge` and attaching to PPA 5001. While this will still work, we will recommend that an application access this VLAN by opening the DLPI style-1 device node `net/bge5001`.

15 The unbundled Sun Trunking product provides link aggregation support for a few legacy devices; see section 6.3.2.

16 The `show-iptun` subcommand in this table will be introduced by the IP Tunnel Device Driver Clearview component.

	Nemo device	**Legacy Device**	**IP Tunnel**	**Aggregation**	**VLAN**
Existing DLPI support	Y	Y	N/A	Y	Y
Vanity Naming support	Y	Y	Y	Y	Y
Generic VLAN support	Y*	Y*	N/A	Y	N/A
Link aggregation support	Y*	Y*	N/A	N/A	N/A
Specific info (dladm *subcommand*)	show-phys	show-phys	show-iptun	show-aggr	show-vlan
General info (dladm *subcommand*)	show-link	show-link	show-link	show-link	show-link

Note: Y* means it only applies to Ethernet devices.

Table 1

4 dladm Changes

The following is a summary of the changes to the dladm subcommands which will be made by this component. For existing subcommands, we only highlight the changes that will be introduced.

4.1 Show Configuration

4.1.1 show-link

```
dladm show-link [-pP] [<link>]
dladm show-link [-s [-i <interval>]] [-p] [<link>]
```

As is currently the case, administrators will be able to monitor network data-link information using the dladm show-link subcommand[17]. If the link argument is specified, show-link will display the data-link information for the specified link, otherwise it will display the information for all links.

For backward compatibility, the -p option will still specify that the information should be displayed in a machine parseable format. By default, show-link will show available links on the running system. A link will be *available* unless it was temporarily deleted using the -t flag (e.g., delete-vlan -t), or the hardware associated with it has been removed but the hardware configuration has not been deleted (section 4.3.5):

```
$ dladm show-link
LINK        CLASS   MTU     STATE    OVER
eth0        phys    1500    up       --
ib0         phys    4092    down     --
aggr0       aggr    1500    up       eth0 eth2
vlan0       VLAN    1500    up       aggr0
tun0        iptun   1452    up       --
...
```

As can be seen from the example, for each link, show-link will show the name, the class, the MTU, the link state and the network topology (OVER) associated with the specified link.

If a link will be recreated during system reboot, it will be considered *persistent*. A link will be persistent unless created using the -t flag (e.g., create-vlan -t). If the -P option is specified, show-link will show links persistent on the system:

```
$ dladm show-link -P
LINK      CLASS     OVER
eth0      phys      --
eth1      phys      --
ib0       phys      --
aggr0     aggr      eth0 eth2
vlan0     vlan      eth0
vlan1     vlan      aggr0
vlan2     vlan      eth1
...
```

In the above example, vlan0 is only shown in the show-link -P output because it has been deleted by dladm delete-vlan -t (section 4.3.3); eth1 and vlan2 are no longer available because the hardware associated with the eth1 link has been removed from the system.

The link configuration associated with removed hardware can subsequently be reassociated with other compatible hardware on the system (section 6.2.7). Alternatively, the link configuration can be explicitly deleted (section 4.3.5), which will cause show-link -P to

17 Note that although all options currently supported by show-link will still be supported and will have similar semantics, the output format of show-link will be changed significantly and is still a work in progress.

no longer display it. Any non-persistent links associated with hardware will be automatically deleted when the hardware is removed (section 6.2.7).

As is currently the case, the `-s` option will be used to show the statistics of each link. The `-i interval` option will be used with `-s` to specify an interval, in seconds, at which statistics will be displayed. If the `-i` option is not specified, statistics will only be displayed once:

```
$ dladm show-link -s
LINK      IPACKETS   RBYTES     IERRORS    OPACKETS     OBYTES       OERRORS
eth0      161486528  581096008  0          80323486     10948428787  0
vlan0     0          0          0          0            0            0
ib0       0          0          0          0            0            0
aggr0     4850278    352964848  0          7288030      5939684768   0
vlan1     156636250  228131160  0          73035456     5008744019   0
tun0      0          0          0          0            0            0
...
```

Note that the statistics of `eth0` in the above example will show the traffic statistics of all links `OVER eth0`, including VLAN and virtual link traffic that ultimately goes through `eth0`.

If the `link` argument is not specified to `show-link`, `dladm_walk_datalink_id()` will be called to get all available links (see 5.2.1), and `dladm_info()` will be called to get the generic attributes of a specific link (section 5.2.2). If the specific link is a VLAN or an aggregation, `dladm_vlan_info()` or `dladm_aggr_info()` will be called respectively to get the link's topology information (section 5.2.2).

4.1.2 show-phys

```
dladm show-phys [-pP] [<phys_link>]
```

The `show-phys` subcommand will be used to show the device name and physical attributes of a specific physical link. The `-p` option will specify that the information should be displayed in a parseable format. If `-P` is not specified, `show-phys` will only show available physical links.

```
$ dladm show-phys
LINK        MEDIA         STATE    SPEED    DUPLEX    DEVICE
eth0        ether         up       100Mb    full      bge0
ib0         ib            down       0Mb    --        ibd0

$ dladm show-phys -P
LINK        DEVICE     MEDIA       FLAGS
eth0        bge0       ether       -----
ib0         ibd0       ib          -----
eth1        ce0        ether       r----
```

The `r` flag in the `show-phys -P` output will indicate that the physical link is `REMOVED`, that is, the hardware associated with the link has been removed. Additional `FLAGS` may be defined in the future if necessary.

The `show-phys -P` subcommand will be very useful for DR operations: administrators will be able to identify the removed hardware by looking for the links marked with the `r` flag. They can then issue `delete-phys` or a DR operation together with `rename-link` in order to either delete or inherit all configuration associated with the removed hardware (section 6.2.7).

4.1.3 show-dev

```
dladm show-dev [-s [-i <interval>]] [-p] [<dev>]
```

As is currently the case, the `show-dev` subcommand is used to show the physical attributes

of the device. However, `show-dev` operates on a device name rather than a link name, which is inconsistent with the future network administrative model. Therefore, the `show-dev` subcommand will be reclassified as obsolete, and `show-phys` will display all network information currently shown by `show-dev`. Note that another possibility would be to change `show-dev` to accept a link name, but that option was eventually not selected because it would lead to inconsistencies between `show-dev` and the `-d` option to subcommmands such as `create-aggr`.

Although legacy devices are currently displayed by the `show-dev` subcommand, their information is not always correct: some legacy devices provide statistics using different names from what `show-dev` expects. For example, some devices have `duplex` instead of `link_duplex` statistics; further, `link_state` is not provided by GLDv2 drivers.

Because this problem will be easily addressed as a part of the Nemo unification work (section 6.1), in the future, the `show-dev` subcommand will be able to provide correct physical information for all physical links including those of legacy network devices.

4.1.4 show-vlan

```
dladm show-vlan [-pP] [<vlan_link>]
```

The `show-vlan` subcommand will be introduced to show VLAN specific information. Again, by default, `show-vlan` will show available VLANs on the running system, and with the `-P` option, `show-vlan` will show persistent VLANs:

```
$ dladm show-vlan
LINK        VID     OVER     FLAGS
vlan1       2       aggr0    -i---
...
$ dladm show-vlan -P
LINK        VID     OVER     FLAGS
vlan0       30      eth0     -----
vlan1       2       aggr0    -----
vlan2       11      eth1     f----
...
```

The `i` flag in the `show-vlan` output will indicate that the VLAN was implicitly created by an application using the VLAN PPA hack name (footnote 14), and the `f` flag will indicate that the VLAN was forcefully created using the `-f` option to `create-vlan` (section 4.3.2). Additional `FLAGS` may be defined in the future if necessary.

4.1.5 show-aggr

```
dladm show-aggr [-LvpP] [-s [-i interval]] [<key> | <aggr_link>]
```

The `show-aggr` subcommand was introduced by Nemo to show aggregation-specific information. There will be several changes made to the `show-aggr` subcommand:

- A new option, `-x`, will be added to display detailed aggregations information, including the aggregation port information on the system.
- A new option, `-P`, will be added to display persistent aggregations on the system. Note that the previous `-L` option (showing aggregation LACP information) and the new `-v` option cannot be used together with the `-P` option.
- The aggregation's link name (`aggr_link`, section 4.2) will be able to be used to specify the aggregation to display. The `key` argument will no longer be necessary and will be marked as obsolete.
- Link names rather than device names associated with this aggregation will be displayed in the output, even if the aggregation is created over devices using the `-d` option.

– The format of the output will also be changed to be consistent with other `dlam show-*` subcommands.

```
# dladm show-aggr
LINK      POLICY  ADDRPOLICY                LACPACTIVITY LACPTIMER  FLAGS
aggr1     L4      auto                      off          short      -----
aggr2     L2      fixed (0:3:ba:50:9a:50)   off          short      -----

# dladm show-aggr -P
LINK      POLICY  ADDRPOLICY                LACPACTIVITY LACPTIMER  FLAGS
aggr1     L4      auto                      off          short      -----
aggr2     L2      fixed (0:3:ba:50:9a:50)   off          short      -----
aggr3     L4      auto                      on           short      f----

# dladm show-aggr -L
LINK   PORT   AGGREGATABLE SYNC COLL DIST DEFAULTED EXPIRED
aggr1  eth0   yes          no   no   no   no        no
       eth1   yes          no   no   no   no        no
aggr2  eth2   yes          no   no   no   no        no

# dladm show-aggr -x
LINK   PORT   SPEED   DUPLEX   STATE    ADDRESS           PORTSTATE
aggr1  --     1000Mb  full     up       0:3:ba:50:9a:e    --
       eth0   1000Mb  full     up       0:3:ba:50:9a:e    attached
       eth1   1000Mb  unknown  unknown  0:3:ba:50:9a:f    attached
aggr2  --        0Mb  unknown  unknown  0:3:ba:50:9a:50   --
       eth2      0Mb  unknown  unknown  0:3:ba:50:9a:10   standby

# dladm show-aggr aggr1 -s
LINK   PORT   IPACKETS  RBYTES      OPACKETS  OBYTES      %IPKTS %OPKTS
aggr1  --     880       80718       28        3278        --     --
       eth0   412       37835       13        1536        46.8   46.4
       eth1   468       42883       15        1742        53.2   53.6
```

The `f` flag in the `show-aggr` output will indicate that the aggregation was forcefully created using the `-f` option to `create-aggr` (section 6.1.5). Additional `FLAGS` may be defined in the future if necessary.

4.2 Virtual Link Configuration

```
dladm create-aggr [-tf] [-R <root-dir>] [-p <policy>]
       [-L <mode>] [-T <time>] [-u <address>] [-d <dev1> | -l <link1>]
       [-d <dev2> | -l <link2>] ... <key | aggr_link>

dladm delete-aggr [-t] [-R <root-dir>] <key | aggr_link>

dladm add-aggr [-tf] [-R root-dir] [-d <dev1> | -l <link1>]
       [-d <dev2> | -l <link2>] ... <key | aggr_link>

dladm remove-aggr [-t] [-R <root-dir>] [-d <dev1> | -l <link1>]
       [-d <dev2> | -l <link2>] ... <key | aggr_link>

dladm modify-aggr [-t] [-R <root-dir>] [-P <policy>]
       [-L <mode>] [-T <time>] [-u <address>] <key | aggr_link>
```

In the future, virtual links such as aggregations and IP tunnels will be able to be administered using their link names. Using aggregation configuration as an example, the aggregation's link name (`aggr_link`) will also be used to specify the aggregation being configured.

If an aggregation is created using the `key` argument, the system will give the aggregation a default link name of *`aggr`<key>*. The administrator will be able to operate on the aggregation using either `key` or *`aggr`<key>* afterwards. On the other hand, if an aggregation is created using the `aggr_link` argument, the system will choose the number greater than

1000[18] as the aggregation's key but this key will only be used internally, and administrators will not be able to administrate the aggregation using key (section 6.2.6). Because the aggregation will be able to be configured using its link name, the key argument will no longer be necessary and will be marked as obsolete.

In addition, administrators will be able to aggregate physical links (the -l option) instead of devices (the -d option). The latter will also be marked as obsolete.

Note that the -l option is used in the create-aggr and modify-aggr subcommands today to specify the LACP mode. To avoid confusion with the -l <link> option, a new -L option[19] will be introduced to specify the LACP mode. As link names and LACP mode values have different formats and can be easily differentiated, we will be able to keep the -l <mode> option for backward compatibility, but it will be marked as obsolete.

Further, because of implementation restrictions, legacy devices which do not have DL_NOTE_LINK_UP/DL_NOTE_LINK_DOWN notifications support will only be able to be aggregated using the -f option with the dladm create-aggr or the dladm add-aggr subcommands (section 6.1.5).

The dladm subcommands for IP tunnel configuration are described in section 4 of the IP Tunnel Device Driver document.

4.3 Vanity Naming Configuration

Below are the proposed vanity naming administrative commands. By default, the configuration will be applied to the running system *and* be applied persistently across reboots. However, if the -t option is specified, the configuration will only be applied to the running system, and will not survive the next reboot.

4.3.1 rename-link

```
dladm rename-link [-R <root-dir>] <link1> <link2>
```

There will be three types of link rename requests:

1. Rename an existent link to a link that does not exist.

   This will be the most common usage of rename-link. Upon success, link1 will be renamed to link2, and all link configuration based on link1 will be updated to be based on link2. The rename will fail if link1 is currently in use (section 6.2.6).

   Note that a rename operation may cause potential confusion. For example, if the administrator plumbs bge1000 (VLAN 1 over link bge0) and then renames the bge0 link to net0, the rename will not affect the bge1000 link, and an attempt to plumb net1000 will fail because the VLAN already exists under another name. If bge1000 is then unplumbed and replumbed, the replumb will fail because bge0 no longer exists. At that point, administrators will have to plumb net1000 to access the VLAN.

2. Rename a non-existent link to a REMOVED physical link.

   This usage (and the third one) will be used for network configuration inheritance — that is, to make newly added hardware inherit the configuration of a REMOVED link (section 6.2.7).

   Assume net0 is the link name of the bge0 device, and bge0 is disconnected by DR. Administrators will be able to run dladm rename-link ce0 net0 *before* they connect the new ce0 device to the system, and the new device will inherit all configuration based on the link name net0 once it is connected to the system.

18 Due to the VLAN PPA hack, aggregation keys are required to be less than 1000 today.

19 Note that this semantics is consistent with the -L option used in the show-aggr subcommand.

3. Rename an available[20] link to a `REMOVED` physical link.

   Administrators will also be able to run `dladm rename-link ce0 net0` *after* `ce0` is connected. Upon success, the `ce0` link will be renamed to `net0`, and all inactive configuration associated with the disconnected `bge0` device will be restored.

   Note that the rename will fail if `ce0` is already open, or if there is any VLAN or aggregation created over `ce0`. Further, the rename will fail if any configuration associated with the `REMOVED` physical link does not apply to the newly added link. For example, if the `REMOVED` physical link was associated to an Ethernet device, and the newly added link is an Infiniband device.

In either case, administrators must take care not to refer to `link1` in any non-`dladm` configuration (for example, `/etc/hostname.*`), otherwise the results may be unexpected (section 3.1.1).

This subcommand will call into `libdladm`'s `dladm_rename_link()` function (section 5.2.4).

4.3.2 `create-vlan`

```
dladm create-vlan [-t] [-f] [-R <root-dir>] -l <link> -v <vid> [<vlan_link>]
```

As discussed in footnote 14, VLANs are currently created implicitly when applications open a physical Ethernet link (or an aggregation) and attach to a VLAN hack PPA. In this case, the VLAN name is tied to the underlying device name (for instance, `bge1000`). Further, because links in use will not be able to be renamed, and implicitly created VLANs only exist while in use, implicitly created VLANs will not be able to be renamed. Therefore, a new `dladm create-vlan` subcommand will be introduced to allow VLANs to be created with meaningful names.

The `dladm create-vlan` subcommand will be used to explicitly create a VLAN with the name specified by `vlan_link`. If `vlan_link` is not specified, the VLAN will be given a default name using the VLAN PPA hack rule[21]. For example, `dladm create-vlan -l net0 -v 1` will create a VLAN with the name of `net1000`.

Unfortunately, some network drivers currently have an MTU issue (section 6.1.4) which makes generic VLAN deployment problematic. To prevent accidental misconfiguration, the system will not allow the creation of VLANs over such links unless the `-f` option is specified.

This subcommand will call into `libdladm`'s `dladm_vlan_create()` function (section 5.2.5).

4.3.3 `delete-vlan`

```
dladm delete-vlan [-t] [-R <root-dir>] <vlan_link>
```

The `delete-vlan` subcommand will be used to delete a VLAN. Note that an attempt to delete a VLAN which is currently in use will fail.

This subcommand will call into `libdladm`'s `dladm_vlan_delete()` function (section 5.2.5).

4.3.4 `autopush linkprop`

20 Physical links become unavailable when their associated hardware is either disconnected by DR or removed across a system reboot.

21 The PPA hack rule only works correctly for PPAs less than 1000. Thus, if the specified PPA is 1000 or greater, `vlan_link` must be specified.

```
dladm set-linkprop -t [-R root-dir] -p autopush=<modlist> <link>
dladm show-linkprop -cP -p autopush <link>
dladm reset-linkprop -t [-R root-dir] -p autopush <link>
```

The existing `set-linkprop` subcommand will be used to configure the per-link `autopush` settings. Specifically, the `autopush` link property will be used to represent the lists of automatically pushed STREAMS modules when a specified link is opened. Upon success, the previous `autopush` link property setting configured on the link will be overwritten.

The `modlist` argument will be a dot ('`.`') separated list of modules. Its syntax will be the same as `autopush(1M)`:

- At most eight modules will be able to be specified over a link.
- The modules will be pushed in the order they are specified.
- An optional special character sequence `[anchor]` indicates that a STREAMS anchor needs to be placed on the stream at the module previously specified in the list.
- It will be an error to specify more than one anchor or to have an anchor first in the list.

Note that `autopush` link property settings configured by `dladm set-linkprop` will only take effect when the specified link is next opened. That is, the autopush modules will not be pushed onto the stream if the specified link is opened before the `dladm set-linkprop` subcommand is issued.

The `reset-linkprop` subcommand will be used to remove the `autopush` link property settings for specific link, and `show-linkprop` will be used to list the `autopush` link property settings.

This subcommand will call into `libdladm`'s `dladm_set_prop()` and `dladm_get_prop()` functions (section 6.3.5).

4.3.5 `delete-phys`

```
dladm delete-phys [<phys_link>]
```

Administrators will be able to use the `delete-phys` subcommand to inform the system that the specific physical hardware which was removed will never be inserted back, thus deleting all link configuration associated with it. Thus, the names of the deleted links (including the physical link itself and all associated VLANs) will be able to be reused. The `phys_link` argument must be a `REMOVED` physical link, otherwise the `delete-phys` operation will fail. If the `phys_link` argument is not specified, the link configuration associated with all `REMOVED` physical links on the system will be deleted.

This subcommand will call into `dladm_phys_delete()` function (section 5.2.6).

5 Library Changes

5.1 libdladm

The libdladm library, together with several other libraries (libwladm and liblaadm), were first introduced by Nemo and provided APIs used by the dladm utility. As result of PSARC 2007/140, they were merged into a single libdladm library. Since vanity naming will also be administered using dladm, we will enhance libdladm to support vanity naming, as described below. As with all existing libdladm routines, the proposed functions will return DLADM_STATUS_OK upon success and a DLADM_STATUS_XXX error code upon failure. Additionally, the new functions will follow the current naming convention: dladm_<class>_<operation>.

Further, aside from each dladm_<class>_create() function, where the linkid is passed as the output argument, all other libdladm interfaces will take linkid (datalink_id_t) instead of link names as the input arguments. Although this might require an extra step to convert the link name into linkid before libdladm functions can be called, this approach has the following advantages:

- Because link name to linkid conversion is needed, libdladm API users will be forced to evaluate the impact of a link name change, and take necessary actions.
- Applications that do not care about link name changes might simply keep linkids instead of link names in their configuration, and not worry about the impact of a link renaming operation.

5.2.1 Walking Links

```
dladm_status_t dladm_walk_datalink_id(int (*fn)(datalink_id_t, void *),
    void *arg, dladm_class_t class, datalink_mediareq_t media,
    uint32_t flags);
dladm_status_t dladm_walk(int (*fn)(const char *, void *), void *arg,
    dladm_class_t class, uint32_t media, uint32_t flags);
```

Currently the dladm_walk() function is used to to enumerate all network links on the system, by walking all DDI_NT_NET device nodes. Because the link name will no longer be directly derived from the link's DDI_NT_NET node, in the future, a new dladm_walk_datalink_id() function will be used to walk the linkids of all network links on the system. This new function will take additional arguments (class, media and flags) to restrict the set of links to enumerate:

- The class argument will be used to specify the class(es) of links.
- The media argument will be used to specified the media types of links
- The flags argument will be a combination of DLADM_OPT_ACTIVE and DLADM_OPT_PERSIST, to indicate whether to enumerate the active links, persistent links, or both.

The dladm_walk() function will still be supported to enumerate link names on the system. It will call into the dladm_walk_datalink_id() function, to get the list of link names.

5.2.2 Querying Link Information

```
dladm_status_t dladm_info(datalink_id_t linkid, dladm_attr_t *dap);

typedef struct dladm_attr {
        uint_t        da_max_sdu;
} dladm_attr_t;
```

Today, the `dladm_info()` function is used to return the attributes of the specified link by issuing a `DLDIOCATTR` ioctl to the `dld` driver. To make the ioctl constants easier to read, all of the proposed ioctls are of the form `DLDIOC_`*operation*. For consistency, the only remaining original ioctl, `DLDIOCATTR`, will be changed to `DLDIOC_ATTR`. Since `DLDIOCATTR` is not a public interface, this change will have no impact on backward compatibility.

Further, the `dladm_attr_t` structure will be changed to only return general link information (see above). Other class-specific link information (for example, VLAN IDs for VLANs), will be returned by the new `dladm_phys_info()`, `dladm_vlan_info()` and `dladm_aggr_info()` functions:

```
dladm_status_t dladm_phys_info(datalink_id_t linkid,
        dladm_phys_attr_t *dpap, uint32_t flags);
dladm_status_t dladm_vlan_info(datalink_id_t vlanid,
        dladm_vlan_attr_t *dvap, uint32_t flags);
dladm_status_t dladm_aggr_info(datalink_id_t linkid,
        dladm_aggr_grp_attr_t *attrp, uint32_t flags);

typedef struct dladm_phys_attr {
        char    dp_dev[DLPI_LINKNAME_MAX];
        char    dp_path[MAXPATHLEN];
} dladm_phys_attr_t;

typedef struct dladm_vlan_attr {
        uint16_t            dv_vid;
        datalink_id_t       dv_linkid;
        boolean_t           dv_force;
        boolean_t           dv_implicit;
} dladm_vlan_attr_t;

typedef struct dladm_aggr_grp_attr {
        datalink_id_t               lg_linkid;
        uint32_t                    lg_key;
        uint32_t                    lg_nports;
        dladm_aggr_port_attr_t      *lg_ports;
        uint32_t                    lg_policy;
        uchar_t                     lg_mac[ETHERADDRL];
        boolean_t                   lg_mac_fixed;
        boolean_t                   lg_force;
        aggr_lacp_mode_t            lg_lacp_mode;
        aggr_lacp_timer_t           lg_lacp_timer;
} dladm_aggr_grp_attr_t;
```

5.2.3 Validating Link Names

```
boolean_t dladm_valid_linkname(const char *link);
```

The `dladm_valid_linkname()` function will be used to validate a link name. Allowed characters in a valid link name are: alphanumeric (a-z, A-Z, 0-9), the dot ('.') and the underscore ('_'). In addition, a valid link name cannot start with a digit, and must end with a digit.

5.2.4 Renaming Links

```
dladm_status_t dladm_rename_link(const char *link1, const char *link2);
```

The `dladm_rename_link()` function will be used to rename a link from `link1` to `link2`. It will issue a `DLDIOC_RENAME` ioctl to the `dld` driver.

5.2.5 Creating and Deleting VLANs

```
dladm_status_t dladm_vlan_create(const char *vlan, datalink_id_t linkid,
            uint16_t vid, uint32_t flags);
dladm_status_t dladm_vlan_delete(datalink_id_t vlanid, uint32_t flags);
```

The `dladm_vlan_create()` function will be used to create a VLAN over the given link, using the given VLAN name and VLAN ID.

The `dladm_vlan_delete()` function will be used to delete a given VLAN.

The `flags` parameter in the above functions can be `DLADM_OPT_ACTIVE` or `DLADM_OPT_PERSIST` to indicate whether or not the operation only applies to the running system and will not survive (across reboot). In addition, the `flags` parameter can be set to `DLADM_OPT_FORCE` for `dladm_vlan_create()` to force creation of a VLAN over a link that has the MTU issue discussed in section 6.1.4.

Each function will issue an ioctl to the `dld` driver: either `DLDIOC_CREATE_VLAN` or `DLDIOC_DELETE_VLAN`.

5.2.6 Deleting Physical link configuration

```
dladm_status_t dladm_phys_delete(datalink_id_t linkid);
```

The `dladm_phys_delete()` function will be used to delete all link configuration associated with the given `REMOVED` physical link. If the `linkid` argument is `DATALINK_INVALID_LINKID`, this function will delete the link configuration associated with all `REMOVED` physical links on the system (section 6.2.7).

5.2.7 Conversion between Linkids and Device Names

```
dladm_status_t dladm_dev2linkid(const char *devname,
      datalink_id_t *linkidp);
dladm_status_t dladm_linkid2legacyname(datalink_id_t linkid,
      char *legacy, size_t len);
```

The `dladm_dev2linkid()` function will be used to query the linkid of the specified physical device, and the `dladm_linkid2legacyname()` function will be used to query the legacy name of a given linkid. The legacy name is the device instance name for a physical link, the PPA hack name for a VLAN or `aggr<key>` for an aggregation.

5.2.8 Link Management API

There will be two sets of APIs:

- Linkid management API

  This API will be used to manage the (linkid, link name) mapping. Changes made using this API take effect immediately and are temporary (i.e. will not exist past the next reboot) unless persisted.

- Link configuration management API

  The (linkid, link name) mappings and other link configuration information will be persisted using the configuration management API.

You can find more details of these two APIs in section 3 of Vanity Naming Link ID and Persistent Configuration Management document. A brief overview is provided below.

The following functions make up the first API:

```
dladm_status_t dladm_create_datalink_id(const char *name,
      datalink_class_t class, uint_t media, uint32_t flags,
      datalink_id_t *id);
dladm_status_t dladm_destroy_datalink_id(datalink_id_t id,
      uint32_t flags);
dladm_status_t dladm_remap_datalink_id(datalink_id_t id,
      const char *name);
dladm_status_t dladm_name2info(const char *name, datalink_id_t *id,
      uint32_t *flags, datalink_class_t *class, uint32_t *media);
dladm_status_t dladm_datalink_id2info(datalink_id_t id, uint32_t *flagp,
      dladm_class_t *classp, uint_t *media, char *name, size_t
      name_length);
dladm_status_t dladm_up_datalink_id(datalink_id_t id);
```

`dladm_create_datalink_id()` will create a [link name, linkid] mapping. `dladm_destroy_datalink_id()` will remove the mapping and will make the linkid available for reassignment to another mapping. `dladm_remap_datalink_id()` will change the name a linkid is assigned to. `dladm_name2info()` and `dladm_datalink_id2info()` will provide information about a link based on the given link name or the given linkid. `dladm_up_datalink_id()` will mark the given link as active.

The following make up the second API:

```
dladm_status_t dladm_create_conf(const char *name, datalink_id_t id,
datalink_class_t class, uint_32_t media, dladm_conf_t *conf)
dladm_status_t dladm_read_conf(datalink_id_t id, dladm_conf_t *conf)
dladm_status_t dladm_get_conf_field(dladm_conf_t conf, const char
*field, dladm_datatype_t *type, void *value, size_t vallen)
dladm_status_t dladm_set_conf_field(dladm_conf_t conf, const char
*field, dladm_data_t type, void *value)
dladm_status_t dladm_unset_conf_field(dladm_conf_t conf, const char
*field)
dladm_status_t dladm_rename_conf(datalink_id_t id, const char *name)
dladm_status_t dladm_write_conf(dladm_conf_t conf)
void dladm_remove_conf(datalink_id_t id)
void dladm_destroy_conf(dladm_conf_t conf)
```

`dladm_create_conf()` will create a reference to a link configuration structure. `dladm_read_conf()` will read a link's configuration into that structure. `dladm_get_conf_field()` and `dladm_set_conf_field()` will get and set, respectively, the given field in the given configuration structure. `dladm_unset_conf_field()` will remove the given field from the given configuration structure. `dladm_rename_conf` will change a link's name in the flat file repository. `dladm_write_conf()` will write the given configuration structure to the flat file repository. `dladm_remove_conf()` will remove the given configuration from the repository. `dladm_destroy_conf()` will return the given configuration structure to the heap.

The functions in the second API will all work on a flat file repository for the system they execute on. The API will have no support for alternate root environments (i.e. JumpStart). In

order to change configuration in such an environment, the administrator must use the `dladm` command with the `-R` argument. This will update the `var/svc/profile/upgrade` script in the alternate root with `dladm` commands that will execute on the next reboot.

6 Architectural Design

We will enhance the existing Nemo framework to provide both a consistent administrative model and the vanity naming support. Those who are not familiar with the existing Nemo architecture are encouraged to read through the Nemo Interface Specification before proceeding. The enhanced architecture is illustrated below. (The new blocks added by this component are shown in grey.)

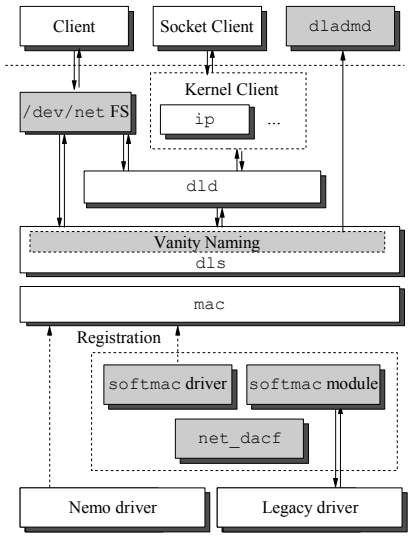

Figure 1 Enhanced Nemo Architecture

6.1 Nemo Unification

6.1.1 Overview

This new architecture will introduce two new modules called `net_dacf` and `softmac` respectively. The `net_dacf` module will be used to track the existence of legacy devices, and the `softmac` module will register "soft" MAC service providers (*softmacs*) to the `mac` (Nemo MAC Services) module on behalf of underlying legacy devices, e.g. `ce`, making each legacy device appear as a Nemo MAC.

The benefit of the softmac approach is obvious – all legacy devices will indirectly become Nemo devices. Therefore, all existing and future features provided by Nemo will be automatically applied to legacy devices[22]. Furthermore, the `dladm` subcommands which administrate Nemo devices will then work automatically for *all* devices, providing a consistent and scalable model to administrate all network links.

6.1.2 `net_dacf`

In order to register the softmac for each legacy device on the system, the `softmac` module must track the current set of legacy devices on the system. A DACF (Device Autoconfiguration Framework) module called `net_dacf` will be used to perform this tracking.

Specifically, the DACF mechanism allows post-attach and pre-detach configuration operations (or "actions") to be performed automatically by the Solaris DDI framework as devices are added or removed from the system (PSARC 1998/212). To use the DACF mechanism, one must provide functions in a DACF module to perform configuration

22 Obviously, the legacy devices will need to have the appropriate attributes to support the functionality Nemo provides. For example, only legacy Ethernet devices will have link aggregation support.

operations, and specify a set of binding rules to bind these configuration operations to specific devices. The binding rules can be based on the device's node path, node type, or device instance name.

The net_dacf module will be the DACF module which implements the post-attach and pre-detach functions for network devices. Further, the DACF bind rules will be updated to bind these functions to DDI_NT_NET devices. As a result, net_dacf will be able to track the attach and the detach of each network device, and hence will know the current set of legacy devices on the system.

6.1.3 softmac

The softmac module will be a shim layer sitting between the existing mac module and the underlying legacy devices. It will:

- Register and unregister softmacs

  The softmac module will create a softmac during the post-attach process of each legacy device, and will register the softmac with the Nemo framework. A dls_vlan_t will then be created and be associated with a dev_t for each legacy device during post-attach. As a result, one will be able to access the specific legacy device as a Nemo MAC, either using the softmac's MAC name, or through the dls_vlan_t by opening the dev_t.

  Likewise, when the device is detached from the system, the softmac module will unregister the softmac from Nemo and the dls_vlan_t will be deleted during the pre-detach process.

  Note that legacy device nodes under /dev (for example, /dev/ce) will be kept to preserve backward compatibility.

- Provide Nemo MAC service

  The Nemo framework requires a MAC being registered to provide its specific MAC callback functions (section 7.1.3) and invoke mac functions when certain events occur (section 7.1.2). The softmac module will provide all MAC services Nemo requires for softmacs just like any other MAC drivers do.

  Specifically, on the receive-side, the softmac module will monitor all data and control DLPI messages that come from legacy devices, and invoke mac functions to inform Nemo accordingly. For example, softmac will call mac_link_update() when it receives link up/down notifications from the underlying device, and will call mac_rx() when M_DATA messages are received.

  On the send-side, the softmac module will process requests from the Nemo framework which come through callback functions registered by softmac, transform them into DLPI requests that the underlying devices understand, and lastly transform the DLPI replies from underlying devices back into return values which can be returned to Nemo. For example, when Nemo calls the mc_promisc() callback function to set the promiscuity of the softmac, softmac_m_promisc() will interpret this request and issue the DL_PROMISCON_REQ to the underlying device, and lastly return success or failure based on the reply from the device[23].

Multiple or Single Lower Streams

Obviously, in order to provide MAC service on behalf of the legacy device, softmac will need to setup one or more streams to the legacy device to send and receive DLPI messages. A stream setup by softmac is referred to as a lower stream, whereas the stream used by the

23 The MAC service offering process will be different based on the type of MAC service DLS clients request. It is either the client-shared lower stream service or the per-stream lower stream service, see more details below. What we described above is the former.

DLS client to operate on the legacy device is referred to as an upper stream. Figure 2 shows the relationship between the two types of streams.

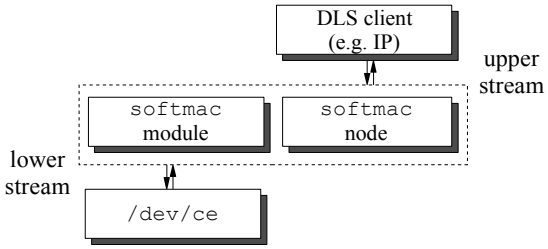

Figure 2 Upper Stream and Lower Stream

There are three possible ways to setup the lower stream(s) to the underlying device:

1. Multiple per-stream lower streams

   In this approach, a lower stream will be created for each upper stream opened by a DLS client. Each lower stream will process the DLPI requests for each associated DLS client.

   This seems straightforward, but cannot provide additional functionalities to the legacy devices. For example, an upstream VLAN PPA access attempt will fail if `softmac` simply passes the `DL_ATTACH_REQ` message to a VLAN-incapable lower stream.

2. Single lower stream

   In this approach, a single lower stream to the underlying device will be shared by multiple upper streams, and will provide the MAC service required by Nemo for the softmac. This single lower stream will be the only stream to the device plumbed by the Nemo framework[24] and all DLPI control and data messages will be exchanged between `softmac` and the device through this stream. Because most legacy DLPI drivers only allow one stream to bind to one SAP, therefore, in order to get all inbound packets of interest, the lower stream will be set to `DL_PROMISC_SAP` promiscuous mode.

   This provides necessary functionalities, but performance analysis has shown that throughput and latency seriously suffer from the extra data demultiplexing and filtering processing in the GLDv3 DLS layer.

3. Combination of both

   In this approach, for each non-VLAN DLS client, or each VLAN DLS client on VLAN-capable legacy devices[25], a lower stream will be set up to exchange the control and data messages with the upper stream. This will avoid the DLS processing overhead because all the data processing will be done by the lower stream, the outbound traffic will be directly passed to the lower stream, and the inbound traffic will be directly passed to the DLS clients. On another hand, for aggregations, or VLAN DLS clients on VLAN-incapable legacy devices, `DL_PROMISC_SAP` lower streams will still need to be established, as described in the second approach.

   This combination provides the necessary functionality, and addresses[26] the performance issues of the second approach. Therefore, the combination approach will be chosen.

   Figure 3 shows how the streams will be set up in order for the `softmac` module to provide Nemo MAC services on behalf of the underlying legacy devices.

24 Others will also be able to open streams to the underlying device by directly accessing the legacy device node. Note that the DLPI messages on those streams will not go through Nemo and hence will not go through `softmac`.

25 Legacy devices that allows VLAN PPA access , see `VLAN_FULL_CAPABLE` capability described in section 6.1.4.

26 Note that `DL_PROMISC_SAP` lower streams are only used in cases that are unsupported today, and therefore do not cause performance regressions.

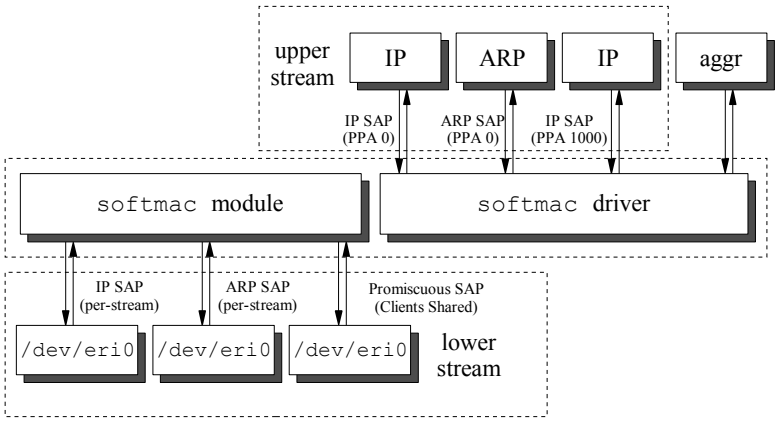

Figure 3 Streams Configuration for softmac

As shown in the figure above, when a DLS client attempts to open a stream over a legacy device, depending on whether it is a VLAN stream and whether the underlying device can support VLAN, Nemo will either share the "single" lower stream with other clients or set up a per-stream lower stream in order to access the softmac. Both types of lower streams will be set up by:

1. Opening the underlying device.
2. For DLPI style-2 devices, attaching to the appropriate PPA.
3. Pushing the `softmac` module onto the stream. The `softmac` module will receive messages from the underlying devices and then pass the messages to the upper streams or to the Nemo framework for further processing.

– When setting up the clients-shared stream, Nemo will bind the stream to a specific SAP[27] and set the `DL_PROMISC_SAP` promiscuous mode, in order to get all data messages of interest (see 7.1.3).

– When setting up the per-stream lower stream for a specific upper stream, a handle of the upper stream will be passed down to `softmac`, and `softmac` will return a handle of the lower stream to Nemo. Therefore, the `dls` and `softmac` modules will be able to make associations between these two streams (for more details, see 7.1.4).

MAC clients like the `aggr` driver will also need to have access to all packets received by a specific device, and hence will also use the clients-shared lower stream to access the device.

DLPI Control-Path

Currently, Nemo processes the DLPI control message sent by the DLS client, and requests MAC service from the MAC if it is required in order to process the message. For example, if a DLS client sends a `DL_PROMISCON_REQ` message to set the device to `DL_PROMISC_PHYS` mode, Nemo processes the `DL_PROMISCON_REQ` and calls the `mc_promisc()` callback function of the corresponding MAC, to request the device to turn on `DL_PROMISC_PHYS` mode.

This control path will not change for upper streams if the corresponding lower streams are of type clients-shared. In particular, the `softmac_m_promisc()` function will send a `DL_PROMISCON_REQ` request down to the underlying device to turn on the device's

27 The SAP value does not matter because the stream will be set to the `DL_PROMISC_SAP` promiscuous mode.

DL_PROMISC_PHYS mode, and wait for the reply from the device for a certain time interval. The softmac_m_promisc() function will return success if the device replies DL_OK_ACK before the timer expires, or otherwise return failure.

On another hand, if the corresponding lower streams are of type per-stream, Nemo will simply pass down the control messages to the legacy devices through the lower streams.

The processing of other DLPI control messages for softmacs will be similar. The details are discussed in section 7.1.3.

DLPI Data-Path

There are several DLPI data-path modes currently supported by Solaris DLPI devices:

- Standard DLPI mode
- Fastpath mode[28]
- Poll mode[29]
- DLIOCRAW mode[30]
- Multidata Transmit (MDT) mode[31]

Different data-path modes define different message formats for passing inbound and outbound network data; Currently Nemo supports the first four data-paths[32]. Based on what data-path mode the upper stream is in, Nemo transforms the DLPI data messages sent by the DLS clients into the "raw" packets and requests the MAC to send packets out on the wire. Similarly, on the receive-side, Nemo processes the "raw" packets that come from the MAC, transforms them into appropriate DLPI data message format, and passes them up to the DLS client.

Note that "raw" means that the Nemo framework expects to operate on the complete packet as part of send and receive processing. Specifically:

- On the send-side, Nemo constructs the full M_DATA packet (including the link layer header) before requesting the MAC to send it out on the wire.
- On the receive-side, Nemo also expects to see the full packet the device receives.

In the future, how to process the messages will be determined by the type of the lower stream (Section 7.1.5):

- Clients-shared lower stream mode

If the lower streams are of type clients-shared, the current Nemo data-path will not change.

On the send-side, the softmac module will enable DLIOCRAW mode on the lower stream. This is to make sure legacy devices do not strip off the link layer header for both inbound and outbound packets. Therefore, softmac will be able to request the lower stream to send the full M_DATA packet sent from Nemo, and pass the full packet it receives from the lower stream to Nemo.

On the receive-side, Nemo provides an interrupt coalescing feature, which requests the network device to send multiple packets in one single interrupt to avoid interrupt storms flooding the system. The softmac module could provide similar functionality by supporting soft interrupt coalescing for legacy devices. That is, if the interrupt coalescing state is enabled by the DLS client, when softmac receives M_DATA messages from the lower stream, it could put the messages into an internal queue instead of passing them up to the Nemo framework. The queued messages would then be passed up to the Nemo framework when the interrupt coalescing timer expires or the queued messages reach an established limit.

28 For a detailed description of Fastpath, see Solaris Network "Fastpath" Technical Description.

29 This DLPI data-path was introduced by Nemo, see appendix A of the Nemo Interface Specification document.

30 DLIOCRAW is described in gld(7D) or dlpi(7P) depending on the release of Solaris.

31 See the Multidata Transmit home page.

32 The Nemo support for MDT is discussed further in section 7.1.4.

However, we compared the performance data with and without soft interrupt coalescing and did not find any performance benefit. Therefore, on the receive side, the softmac module will not do any packet queueing. Instead, it will immediately pass `M_DATA` messages up to the Nemo framework for further processing.

- Per-stream lower stream mode

On another hand, if the lower stream is of type per-stream, most of the Nemo processing will be skipped. The lower stream will be set to the same DLPI data-path mode as the upper stream has, and because there is one-to-one mapping between the lower stream and upper stream, data messages will be able to be passed directly between the two streams.

`kstats`

Nemo currently provides two different sets of `kstats` for Nemo links: one is created for Nemo MACs (MAC `kstats`) and another is created for Nemo data-links (data-link `kstats`). Nemo updates both the MAC and data-link `kstats` by querying the MAC of the statistics.

When any DLS client queries the statistics, Nemo will call through to the softmac's `mc_getstat()` callback function, which retrieves the statistics by querying the underlying driver (see `softmac_m_stat()`). Specifically, the `softmac` module will first query the underlying device using the statistic names defined in the Nemo MAC statistic names array[33]. If that fails, because different legacy drivers implement statistics in very different ways, and because there is no standard naming scheme, `softmac` will also query each device using some commonly used statistic names, which are listed below:

Nemo `kstats`	**Legacy `kstats`**
`ifspeed`	`link_speed`
`norcvbuf`	`rx_no_buf`
`noxmtbuf`	`No Txpkt`
`align_errors`	`alignment_err`
`fcs_errors`	`crc_err`
`tx_late_collsions`	`late_collisions`
`ex_collsions`	`excessive_collisions`
`toolong_errors`	`length_err`
`macrcv_errors`	`Rx Error Count`
`link_duplex`	`duplex`

Fortunately, common statistics such as `ipackets`, `opackets`, `ierrors`, `oerrors` and `collisions` are required by `netstat(1M)`, and thus are supported by all drivers.

Note that we will not be able to change the `kstat` support in legacy drivers. Further, for GLDv3 devices, o keep backward compatibility, we will keep the current support of data-link `kstats`, which are of the form <`driver`, `instance`, `device_name`>. Additionally, we will provide another set of data-link `kstats` named after the link names. To avoid potential naming conflicts with the old form of data-link `kstats`, the new set of data-link `kstats` will be of the form <"`link`", `0`, `link_name`>.

33 The statistics names are listed in `i_mac_si[]`.

autopush Backward Compatibility

For legacy devices, backward compatibility requires that the autopush modules configured using the traditional autopush(1M) syntax are pushed automatically when the upper stream opens a legacy device's softmac node. Specifically, using the following autopush configuration as an example:

```
$ dladm show-phys net0
LINK        STATE    SPEED    DUPLEX    DEVICE
net0        up       1000     full      ce0
# cat /etc/iu.ap
# major      minor  lastminor     modules
   ce         -1     0             vpnmod
```

In the future, assuming no changes are made to the current autopush logic in stropen(), when an application opens /dev/net/net0 (whose major number is softmac instead of ce), the softmac driver would open the underlying device ce0, which would cause all the specified autopush modules (vpnmod in this case) to be pushed between the underlying ce driver and softmac module. Although this might be able to provide backward compatibility, this approach will not work if autopush-ed modules assume that the ip module can be found by walking upstream through q_next.

To solve this, the stropen() autopush logic when opening a network device will be changed to:

1. First, find the associated link of the device based on the dev_t passed to stropen(), then find its per-link autopush configuration. Because the per-link autopush configuration is preferred, if it exists, apply it to the stream.

2. Otherwise, find the link's phy_dev_t (the phy_dev_t is different from the dev_t only if dev_t is a softmac device. In that case, phy_dev_t is the dev_t of the underlying legacy device). We can then find the per-driver autopush configuration based on phy_dev_t. If it exists, apply it to the stream.

Further, when setting up the lower stream, pop out all the autopushed modules, as these modules are already pushed above the associated softmac device in step 2.

6.1.4 Generic VLAN Support

Today, most legacy Ethernet devices do not have VLAN support. With softmac, a generic VLAN solution will be automatically provided for all devices of type DL_ETHER (Ethernet), since softmac will register all legacy devices with Nemo, allowing the Nemo framework to provide the VLAN support.

When deploying a VLAN network using legacy devices, there is one issue that may cause problems: some legacy devices simply assume that the maximum frame size they can handle is 1514 bytes, and if they receive (or are requested to send) any packet whose size is greater than that, the drivers will silently drop the packet.

While the packets on the send-side can be clamped by the upper layer to a smaller size, the issue is particularly problematic on the receive-side. Specifically, because the other systems on the same VLAN think that the neighbors have an MRU of 1500, plus the Ethernet header (14 bytes) and the VLAN tag (4 bytes), the packets sent to this VLAN would be too big for the problematic driver and will be dropped silently. In order to make the VLAN deployment functional, the administrator would have to carefully configure the MTU of each node on the VLAN to be lower.

To address this problem, we will introduce a DLIOCVLANINFO ioctl for network devices. The underlying driver will acknowledge this ioctl with three different values: VLAN_INCAPABLE, VLAN_SIZE_CAPABLE and VLAN_FULL_CAPABLE:

- VLAN_INCAPABLE

  The driver cannot handle packets of bigger size, i.e., cannot receive or send packets 4 bytes longer than the driver's advertised maximum SDU.

- VLAN_SIZE_CAPABLE

  The driver can handle packets of bigger size.

- VLAN_FULL_CAPABLE

  The driver is VLAN_SIZE_CAPABLE and can also handle the VLAN ppa hack DL_ATTACH_REQ itself. This capability will be used to support VLANs over legacy devices using the per-stream lower stream scheme which performs better (see 7.1.5).

Devices that fail to acknowledge the DLIOCVLANINFO ioctl will be considered as VLAN_INCAPABLE.

By default, creating VLANs over devices which are VLAN_INCAPABLE will fail. If administrators are aware of the issue described above and are willing to configure the MTU carefully on each system on the VLAN, or are willing to assume the risk of incorrect VLAN deployment, they will be able to explicitly force a VLAN to be created (section 4.3.2).

We will coordinate to update all Sun supported legacy drivers to be able to handle larger size packets, and acknowledge DLIOCVLANINFO ioctl correctly. Until the fixes are available, the administrator will have to explicitly force to create VLANs over such devices.

6.1.5 Generic Aggregation Support

Similarly, most existing legacy Ethernet devices do not have 802.3ad aggregation support, with a few exceptions (see Sun Trunking support, section 6.3.2). After the Nemo unification work, all DL_ETHER devices will be make known to the Nemo framework. Therefore, theoretically Nemo will be able to provide the aggregation support for all DL_ETHER devices.

Unfortunately, the 802.3ad standard requires support of DL_LINK_NOTE_UP/DOWN notifications in order to attach/detach specific ports from an aggregation. Therefore, legacy devices which do not have such support will only be able to be forcefully aggregated using create-aggr with the "-f" option, and the link state of such devices will always be considered as up. This implementation restriction is unfavorable but will not cause any regressions, as DL_LINK_NOTE_UP/DOWN notifications are also required by the current Sun Trunking product.

6.1.6 Support of Other Media Types

The Clearview project has already integrated a MAC plugin architecture, allowing adoption of various media types into the Nemo framework (see the MAC Type Independent Nemo Architecture document). So far, the Nemo framework can support the DL_ETHER and the DL_WIFI media types.

Work to support the DL_IB (Infiniband) MAC type plugin and implement the GLDv3 ibd (IPoIB) driver is already under way. Therefore, we will not support DL_IB as part of this component. Support of other media types (such as FDDI and Token Ring) will not be considered in our initial delivery, and can be done as part of follow-up work provided that hardware required by the development and testing work become available. Today, devices using use other media types is very rare. Further, the legacy /dev nodes of those devices will remain accessible, thus no regression will be introduced.

6.2 Vanity Naming

6.2.1 Overview

We will introduce a new network/datalink-management SMF service, a new

dlmgmtd daemon, and will also enhance the existing Data-link Services Module (dls) to provide the name mapping functionality required by vanity naming (section 6.2.2). A new directory namespace, /dev/net, will be introduced and will contain all of the available links on the system. This namespace will be managed using a new general-purpose filesystem mechanism called Filesystem Driven Device Naming (devname). Each link will have exactly one /dev/net node, with the file name matching its link name. When applications access the link by its link name using the /dev/net node, the devname filesystem will invoke a name resolution routine to map a link name to the link's dev_t (section 6.2.4) .

6.2.2 Vanity Name Mapping Information

Linkid Management

As we have described in section 3.1.2, network administration will be based on link names, and each link will be assigned a linkid which persists until the link is deleted. Further, all link configuration and kernel structures will refer to linkids so that they will not need to be updated when the link name is changed.

The dlmgmtd daemon will be used to manage the [link name, linkid] mapping for each link on the system. It will allocate linkids and assign them to link names. dlmgmtd will also act as an interface to the flat file repository for storing [link name, linkid] mappings and configuration. For physical links, it will also keep the associated device instance name (section 6.2.5). The administrator will interact with dlmgmtd through dladm.

Note that the mapping will include both the links currently available on the system, and the links which are not accessible but are persistent in the flat file. All links will share the same namespace in order to avoid any potential naming conflicts. Specifically, assume that there is a physical link net0 whose hardware was removed: while this physical link is not available now, the net0 name must not be taken by any other link, so that the configuration associated with net0 will be able to be restored when the hardware is reinserted.

The dlmgmtd daemon will create a door, and will receive door calls from libdladm or the dls module to:

- Allocate a new linkid and associated it with a specific link name.
- Disassociate a link name with its linkid, and free this linkid for future use.
- Make a [link name, linkid] mapping persistent.
- Change a [link name, linkid] mapping so that it has a different name.
- Update configuration information for a particular link.

We will use door calls to simplify the locking model and to simplify management of the available linkids. There are several problems:

- How do we determine the next available linkid?
- How do we check for a name collision?

Having the dls module and dladm allocate linkids themselves causes problems. While there are system-wide locks that we could use, there is a problem of what would happen if one process terminated while holding the lock. With a daemon there is only one set of locks, contained within one process. This makes coordinating access and preventing problems such as deadlock easier.

Preventing a collision, either a name collision or an ID collision (where two links have the same ID) is also easier with a daemon. Without a daemon, dladm and the dls module would have to search through the flat file repository to see what linkids are available, and what link names are in use. A daemon can keep a list in memory and search it quickly using an AVL tree.

The downsides of a daemon are that the daemon could become a single point of failure and/or a bottleneck. However, we will run the daemon under SMF so that if it terminates, SMF will

restart it. Of course, this can only mitigate temporary failures (such as an administrator accidently killing `dlmgmtd`). A hard failure (such as a corrupted repository) cannot be fixed by restarting the daemon. In fact, a hard failure would also prevent a distributed approach from working. A daemon would actually not be a bottleneck, because as stated earlier the daemon can (using a cache of the repository stored in an AVL tree) perform collision detection quicker than a solution that relies on `dladm` and the `dls` module performing linkid management requests.

Kernel Data-link Structures

Currently, Nemo creates a kernel data-link structure (`dls_vlan_t`) for each available link on the system. This will not be changed, but the kernel data-link structure will be modified to use the linkid as its unique identifier. Further, in addition to the SPA, the kernel data-link structure will also be updated to keep the `dev_t` of the link's `/dev/net` node, which will be used to access the `/dev/net` node (section 6.2.4).

Once a kernel data-link structure is created, it will be added to the kernel data-link hash tables. Note that compared to a single hash table `i_dls_vlan_hash` today, in the future, there will be three hash tables created to keep all the kernel data-links:

- `i_dls_vlan_hash`

  Keyed by data-link's linkid.

- `i_dls_vlan_hash_by_dev`

  Keyed by data-link's `dev_t`. This will be used to lookup a kernel data-link once the `dev_t` is resolved by the `devname` filesystem (see 6.2.4).

- `i_dls_vlan_hash_by_spa`

  Keyed by data-link's SPA (mac name and VID). This will be used to check the existence of a specific VLAN to avoid VLAN name conflict.

Below is an example of the information that might be contained in these hash tables on a running system:

linkname	linkid	link-over	SPA(MAC/VID)	dev_t(major/minor)
ce0	1	1	softmac1201/0	softmac/1202
aggr1	2	2	aggr1/0	aggr/1
vlan1	3	2	aggr1/2	aggr/2301
tun0	4	4	tun1536/0	tun/1537

Note that a MAC name will not be obviously tied to either the link name or the device name, but it must be unique system-wide.

Link Configuration Repository

Link configuration information will be stored in `/etc/dladm/datalink.conf` for persistent links and `/var/run/datalink.conf` for temporary links. This will be a flat file whose entries are made up of name-value pairs. This will include vanity name mapping information as well as some link configuration information. As a result, the existing `/etc/dladm/aggregation.conf` and `/etc/dladm/linkprop.conf` are will no longer be needed and will be removed once their contents have been converted to the flat file (described later).

`datalink.conf` will have one entry per line. The format will be as follows:

```
link-id      prop0=type,value;prop1=type,value;...;propN=type,value;
```

A sample `datalink.conf` is:

```
1       name=string,bge0;class=int,1;
2       name=string,bge1;class=int,1;
3       name=string,aggr5;port=string,1.2;class=int,2
4       name=string,vlan0;over=int,1;class=int,3
```

`datalink.conf` is a private file and must not be modified by the administrator. The only supported way to change the file is by using `dladm`.

The reason for two files is to support recovering temporary link information when `dlmgmtd` is restarted, but the system has not rebooted. We can leverage the fact that `/var/run` is cleared out on reboot to ensure that temporary links are removed from the flat file repository without adding special logic to `dlmgmtd`.

Note that having separate configuration files will still be allowed, and will be required to store sensitive configuration that requires different permission (for example, WEP keys for a wireless link). However, any other configuration files will only keep linkids.

Converting `/etc/dladm/aggregation.conf` and `/etc/dladm/linkprop.conf`

The contents of both `/etc/dladm/aggregation.conf` and `/etc/dladm/linkprop.conf` will be converted to the new flat file repository when an administrator upgrades a system running a pre-vanity-naming version of Solaris to a version of Solaris that includes vanity naming. A script will run during upgrade that will read each line of these two files and generate the appropriate `dladm` commands from them. These commands are then appended to `/var/svc/profile/upgrade`. Each line of `aggregation.conf` will correspond to one `dladm create-aggr` command. Each link property listed in `linkprop.conf` will correspond to one `dladm set-linkprop` command.

6.2.3 DLPI Device Nodes

As a result of this component, Nemo will create a DLPI style-1 node for each network link on the system. The `devname` filesystem will resolve the vanity-name node under `/dev/net` to this device node. Note that for legacy devices, the `/dev/net` node will resolve to the `softmac` device node instead of the legacy physical device node.

DLPI Style-1 Node

Only style-1 vanity-name nodes will be supported in the future. Therefore, attempting to open a DLPI style-2 node such as `/dev/net/bge` will fail. We decided not to support DLPI style-2 nodes for the following reasons:

- Having DLPI style-2 nodes will complicate the vanity-name node resolution process. In particular, a style-2 node does not refer to any specific link until an application issues a `DL_ATTACH_REQ` for a PPA. For example, the DLPI style-2 node `/dev/net/net` could have PPA `0` refer to `bge0`, PPA `1` refer to VLAN 1 over `e1000g1`, and PPA `2` refer to a link of non `DL_ETHER` type!

  In addition, some applications issue `DL_INFO_REQ` before attaching to any specific PPA. The acknowledgement of this request will be particularly problematic if the DLPI style-2 node can refer to links of different media types. Specifically, because the media type will not be known until the application attaches to a PPA, a `DL_INFO_REQ` received before that time cannot be acknowledged correctly.

- Currently, opening a DLPI style-2 node forces all possible device instances to be attached, and the stream does not associate to a specific `dip` until `qassociate(9F)` is called when `DL_ATTACH_REQ` is processed. This is feasible today because a DLPI style-2 node is always associated with one driver. However, this would become very problematic for a DLPI style-2 vanity-name node: since any device instance could be associated with a

vanity-name node, all network device instances would need to be held whenever the vanity-name node was opened.

VLAN Creation

Because there will only be DLPI style-1 nodes in the `/dev/net` namespace, applications will not be able to open a VLAN by opening a DLPI style-2 `/dev/net` node and attaching to a specific VLAN hack PPA. Instead, applications will be able to open a VLAN by either of the following two approaches:

- Explicit VLAN creation

  Administrators will be able to explicitly create a VLAN and assign it a vanity name using `dladm create-vlan`. Once created, the VLAN can be opened using the vanity name.

  For example, `dladm create-vlan -l bge0 -v 1 vlan0` will create VLAN 1 over `bge0`, and name it `vlan0`. The `dladm create-vlan` subcommand will fail if the VLAN already exists. An explicitly created VLAN will be able to be deleted using the `dladm delete-vlan` subcommand.

- Implicit VLAN creation

  Applications will also be able to directly open VLANs by opening the style-1 `/dev/net` node using the VLAN's PPA hack name. For example, an application will be able to open `/dev/net/bge1000` to access VLAN 1 over `bge0`. This operation will fail if this VLAN has already been explicitly created with another name. An implicitly created VLAN will be deleted when it is no longer in use by any application.

Note that implicit VLAN creation is provided only for backward compatibility. Because the VLANs created implicitly will not be able to be renamed, administrators will be encouraged to use explicit VLANs and assign them meaningful names.

In addition, both forms of VLAN creation will be affected by a link rename operation. For example, when the link `bge0` is renamed to `net0`, administrators will run `dladm create-vlan -l net0` to create the VLAN explicitly, or applications will open `net/net1000` rather than `net/bge1000` to access the VLAN 1. Although VLAN creation will be based on link names, once a VLAN is created, it will not be affected by a rename operation (section 6.2.6).

6.2.4 `devname` Overview

As the word *implicitly* implies, when applications create a VLAN by opening its `/dev/net` node, the `/dev/net` node does not exist yet for this VLAN. The system must create the `/dev/net` node as a part of the open operation. Creating `/dev/net` nodes on demand is not possible today, but will be possible in the future using the `devname` filesystem.

Specifically, the `devname` filesystem is a project which will deliver a new directory-based filesystem to manage the `/dev` namespace. From the `devname` project design specification:

> *The primary deliverable of this project is a filesystem implementation of `/dev`. This file system is characterized by the features listed here:*
>
> - *exports the `/dev` namespace;*
> - *mounts on `/dev` in the global zone;*
> - *supports multiple mount points;*
> - *utilizes a local backing store, for each mount point, on the underlying filesystem to persist filesystem state across system reboots;*
> - *provides an infrastructure to support (filesystem) directory-based device name resolution.*

In particular, the last feature will be required by vanity naming.

Dynamic Name Resolution

The `devname` filesystem will provide a Directory-Based device Name Resolution (DBNR) infrastructure to resolve a node name to its `dev_t`: DBNR will allow each `/dev` subdirectory it manages to specify a name resolution routine. When an application accesses a `/dev` node, the `devname` filesystem's `VOP_LOOKUP()` routine will be invoked, and in it DBNR will call the directory-specific name resolution routine to resolve the node name into either a symbolic link to the `/devices` node or a specific `dev_t`. With DBNR, if an application attempts to access a non-existent `/dev` node, the invoked name resolution function will be able to create the `/dev` node dynamically.

Specifically for network link vanity naming, a name resolution function for the `/dev/net` subdirectory will be implemented.

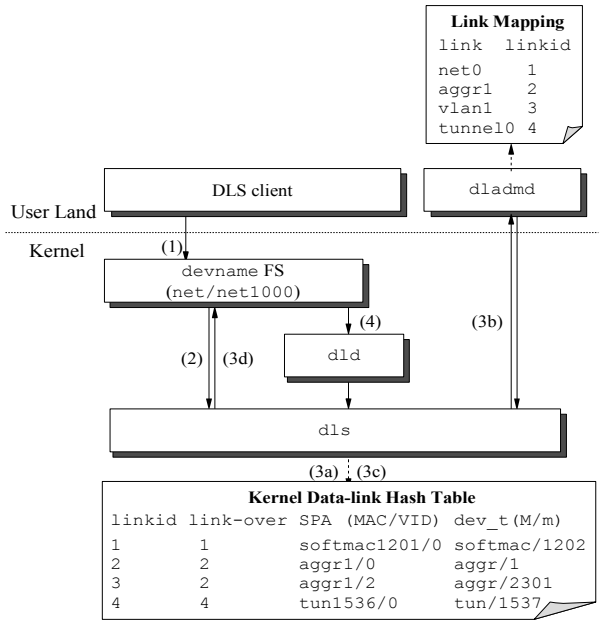

Figure 4 Vanity-name Node Access

Figure 4 shows an example of the name resolution process when a DLS client opens a vanity-name node in `/dev/net`:

1. The DLS client attempts to access `/dev/net/net1000`.
2. The `devname` filesystem's `VOP_LOOKUP()` lookup routine is invoked, which calls the `/dev/net` name resolution function. The name resolution function queries the `dls` module, which in turn queries the `dlmgmtd` daemon to obtain the vanity name mapping for `net1000`.
3. To resolve `net1000` to its `dev_t`, `dls`:
   a. Requests the `dlmgmtd` daemon to looks up `net1000` in its link mapping table.
      → If it is found, then there is currently a link with the name `net1000` on the running system. With `net1000`'s linkid, `dls` looks up the kernel data-link hash table.
      → If the kernel data-link is found and its zoneid is the same as the current zoneid, return its `dev_t`.
      → If the kernel data-link is not found, and the current zoneid is non-global zone, then return failure.

→ Otherwise, dls checks whether the link name should be treated as an implicit VLAN (PPA > 999):

→ If it should not be, then return failure.

b. Makes a door upcall to dlmgmtd to get net0's linkid. If it does not exist, return failure.

b. Makes a door upcall to dlmgmtd to allocate a new linkid and associate it with net1000.

c. Looks up net0's corresponding MAC in the kernel data-link hash table using its linkid, creates a VLAN link for net1000, using allocated linkid and net0's MAC information.

d. Returns the dev_t of net1000.

4. The devname filesystem continues to open the driver with the dev_t of the /dev/net node, so that the DLS clients are able to proceed with other DLPI operations.

Note that unlike the /dev node, which is symbolically linked to the physical /devices node, the /dev/net node will be destroyed when it is no longer open. Each time a DLS client accesses an /dev/net node that is not currently open, the /dev/net name resolution function will be invoked, resolving the /dev/net name to its dev_t.

Further, in order to resolve all /dev/net nodes when the directory is read (e.g., by ls), a similar /dev/net name resolution function will need to be provided for the devname filesystem's VOP_READDIR() routine.

6.2.5 Data-link Creation

Today, kernel data-links are created:

- by mac_register() when physical device instances get attached to the system.
- by mac_register() when virtual device drivers successfully process virtual link creation requests.
- by dls_vlan_create() when VLANs are implicitly created as a result of DLS clients attaching to the VLAN hack PPA.

In the new code, mac_register() will only register MACs with the Nemo framework: The kernel data-link creation will be moved outside of mac_register(). This will allow the MAC layer to be decoupled from the data-link layer, so that mac_register() will not need to take additional arguments associated with data-link creation (for example, link names and linkids).

Instead, data-links will be created:

- As part of the post-attach process of the physical device instance.
- Upon request from a virtual device driver after the driver successfully calls mac_register() (section 6.2.6).
- Upon request from dls as part of either implicit or explicit VLAN creation.

The existing dls_vlan_create() function will continue to be used to create kernel data-links, but its name will be changed to dls_vlan_alloc() to avoid the confusion between it and the dls_create_vlan() function, which will be introduced to process an explicit-VLAN-creation request. The core logic of dls_vlan_alloc() will remain unchanged, except it will create the /devices minor node for the link, which is currently done in mac_register(). To perform this creation, dls_vlan_alloc() will be updated to take a linkid, SPA and the device node's instance number as the arguments.

Likewise, kernel data-links will be deleted:

- during the pre-detach process of a physical device instance.
- upon explicit request from the virtual device driver before the driver calls `mac_unregister()`.
- as a part of deleting a VLAN.

The `/devices` node will be deleted as a part of the kernel data-link deletion process.

Physical Data-links

For convenience, the `net_dacf` post-attach and pre-detach functions described in section 6.1.2 will also be used to call into the `dls` module to create and delete kernel data-links for physical network devices.

As a result, the `net_dacf` post-attach function will:

- if necessary, request `softmac` to register the softmac for the legacy device.
- request `dls` to create the kernel data-link for the physical device.

By default, the link name of a physical link will be its traditional name, i.e., its device instance name. If this default name has been used by other links, a name `net`*N* will be chosen instead. In order to do this, a door upcall will be first issued to `dlmgmtd` to map the device instance name to a linkid. If the device instance name does not exist, `dlmgmtd` will generate a new linkid, and associate it with the device instance name and a data-link name which will be either:

- the device instance name if it is available.
- the name `net`*N*, where *N* is lowest available number, e.g., `net0`[34].

The `dlmgmtd` daemon will then return the linkid to the `softmac` module, which calls `dls_vlan_alloc()`.

Likewise, the `net_dacf` pre-detach function will:

- request `dls` to delete the data-link for the physical device.
- if necessary, unregister the softmac.

As a result, the kernel data-link will be deleted with the physical link's linkid. Because the detachment of a physical device instance only indicates that the corresponding physical link is currently not available, the persistent [link name, linkid] mapping for the physical link will still be kept by the `dlmgmtd` daemon.

Note that when the device instance fails to detach, the DACF framework ensures that the pre-detach operations will be reversed by calling the post-attach function.

Virtual Data-links

The virtual device driver will explicitly request `dls` to create the kernel data-link after it successfully registers the MAC, and request `dls` to delete the kernel data-link before it unregisters the MAC. The details are discussed in section 6.2.6.

6.2.6 `dladm` Subcommand Processing

Virtual Link Operations

As a result of this component, virtual links will be administered using their link names, which must internally be mapped to linkids by `dladm` or `libdladm`. The mapping method will depend on the type of operation:

- For operations that create a new link, `libdladm` will issue a door call to `dlmgmtd` to generate a new linkid and associate it with the new link.

34 Attach failure will be very rare, so we choose to generate another name for the physical device if the device instance name is not available. A message will be logged to indicate the name collision.

– For operations that use an existing link, dladm will issue a door call to dlmgmtd to obtain its linkid.

The libdladm library will then send the virtual link operation requests (ioctls) to the virtual device driver using the linkid.

Using aggregation operations as an example:

– Creating Aggregations

  For create-aggr, libdladm will issue a door call to dlmgmtd to get the linkids of the aggregated links from either the link name (if create-aggr -l was used) or the device name (if create-aggr -d was used). Then, libdladm will request dlmgmtd to generate a new linkid and associate the linkid with the given aggregation name. This request may fail if the given link name already exists, but if it succeeds, libdladm will send a LAIOC_CREATE ioctl to the aggr driver to create the aggregation.

  The laioc_create_t structure used by LAIOC_CREATE will be changed to carry the the linkid of the aggregation, and the linkids of the aggregated links.

  The aggr driver will process the LAIOC_CREATE ioctl. It will first create the aggregation group. After that succeeds, aggr will register the MAC with Nemo, and pick <key> (if the key argument was specified) or -1 as the Mac's instance number. As a result, the mac module will pick aggr<key> or the lowest available aggr<1000+*N*> as the aggregation's MAC name. Then aggr will request dls to create the aggregation's kernel data-link.

  When libdladm receives a successful acknowledgement from the aggr driver, it will update the link persistent repository with the persistent aggregation.

  Note that the aggregation's link name will not be kept by the aggr driver. Instead, the linkid will be used in any dladm aggregation operation (e.g., show-aggr). The dladm daemon will first query dlmgmtd to get the linkid of the aggregation, and then pass it to libdladm, which will send the specific ioctl to the aggr driver with the linkid.

Creating and Deleting VLANs

VLANs will be created explicitly by the dladm create-vlan subcommand, or created implicitly by accessing the VLAN PPA hack name. The latter is described in section 6.2.4.

When a VLAN is created explicitly, in addition to looking up the linkid of the link on which the VLAN is created, libdladm will also request dlmgmtd to generate a new linkid and associate that linkid with the given VLAN name. Upon success, libdladm will send a DLDIOC_CREATE_VLAN ioctl to the dld driver.

The dld driver will pass the ioctl to the dls module, which will create the kernel data-link for this VLAN and insert it into three kernel data-link hash tables (see 6.2.2). Failing to insert into i_dls_vlan_hash_by_spa means the given VLAN already has another name, and the VLAN creation will fail. When libdladm receives a successful acknowledgement from the dld driver, it will request dlmgmtd to update the link configuration repository with the persistent VLAN.

If the VLAN is deleted by dladm delete-vlan, dladm will lookup the linkid of the given VLAN name. If the linkid exists, libdladm will send a DLDIOC_DELETE_VLAN ioctl to the dld driver to delete the VLAN.

Renaming Links

As described in section 4.3.1, there are three types of link rename requests. Here, the first type is discussed; the latter two will be discussed in section 6.2.7:

– Rename an existent link to a link name that does not exist on the system.

The libdladm library will first send a DLDIOC_RENAME ioctl to the dld driver, which

will then be passed to the dls module. The dls module will return success if the specified link is not currently held open. The libdladm library will then inform the dlmgmtd daemon to update the [link name, linkid] mapping for this specific link. Finally, if the link is persistent, its link name will be updated in the link configuration repository.

Note that VLANs and aggregations associated with the renamed link are created over the link's corresponding MAC in the kernel, and their persistent link configuration will only refer to the linkid, which will never change. Therefore, the rename operation will not affect the traffic over the relevant VLANs or aggregations, nor their persistent link configuration.

6.2.7 Dynamic Reconfiguration

RCM Network Device Namespace

RCM (Reconfiguration Coordination Management framework) is a generalized framework which uses plugin modules to do all of the higher level policy analysis and user land preparation associated with doing DR operations.

Today, the network_rcm plugin module tracks the current set of physical network devices. In the network_rcm module, each physical network device is a *network resource* which is identified by an abstract name in the form of SUNW_network/<*device_instance_name*> (for example, SUNW_network/bge0). The network_rcm module maps a device's physical path to its SUNW_network name, and propagates the network device DR operation using its SUNW_network name. Therefore, other RCM plugin modules such as ip_rcm can process the DR operation based on the SUNW_network name.

In the future, this component will change the network resource namespace to SUNW_network/<*linkid*>[35]. As a result, the propagation of network resources to other RCM plugin modules will not be affected by a link rename operation. The network_rcm module will require an additional step to map the <*device_instance_name*> to the <*linkid*>.

Further, currently DR does not support aggregations. A DR disconnect operation simply fails if there is any aggregation created on the device. This issue will be addressed as part of this component.

Link Configuration Reestablishment

When a physical device is disconnected by DR, the network_rcm offline routine is invoked, and it propagates an offline event with the corresponding network resource name to each RCM plugin module interested in this specific network resource. Therefore, each plugin module can process this offline event, and make offline decisions for particular set of configuration. Specifically, the IP RCM plugin module (ip_rcm) is currently the only module which is interested in the network resources. When ip_rcm is informed of a network resource offline event, it checks the IP configuration (including implicitly created VLAN configuration) and determines whether to fail this offline request, or instead to update IP configuration to allow the corresponding device instance to be detached. The ip_rcm module then returns the decision to network_rcm.

The network_rcm online routine (also known as undo_offline()) is used to reverse the offline operation. Likewise, an online event is propagated to each relevant RCM plugin module, and is processed by each plugin module to restore configuration which was removed during the offline process.

When a device is successfully disconnected, the RCM framework loses the device's network resource information. Therefore, when the physical device is reattached, devfsadmd generates a RCM_RESOURCE_NETWORK_NEW sysevent with the device name of the new device. Currently this sysevent only gets consumed by the ip_rcm module, which in turn updates all relevant IP configuration based on the /etc/hostname.<if> configuration file. In the future, this RCM_RESOURCE_NETWORK_NEW sysevent will be changed to be

35 The SUNW_network prefix must be kept because while the linkid is unique within the network subsystem, it might not directly usable in the RCM framework which spans both network and non-network resources.

consumed by the network_rcm module, which will map the device name to its linkid and propagate a RCM_RESOURCE_LINK_NEW network resource event with this linkid. This event will in turn be consumed by different RCM plugin modules to update its particular set of associated configuration.

Specifically, in the future, two new RCM plugin modules will be implemented, which will process the network resource events propagated from network_rcm:

- The aggregation RCM plugin module (aggr_rcm)

  This RCM module will make decisions based on the current aggregation configuration. Specifically, if an offlined network resource is associated with any aggregations, it will be removed from the aggregation by aggr_rcm, unless this device is the last device left in the aggregation. In that case, aggr_rcm will initiate another network offline event for this aggregation network resource, and propagate this event to other RCM modules, such as ip_rcm and vlan_rcm (described later). If other RCM modules agree with this offline event, the aggregation will be deleted by aggr_rcm. Otherwise, administrators will have to force the device disconnect operation, which will in turn force the aggregation to be deleted[36].

  When a network resource online event arrives, aggr_rcm will restore the aggregation configuration, by either adding this physical device back to the aggregation, or recreating the aggregation over again.

  When a RCM_RESOURCE_LINK_NEW RCM event arrives, since it associated aggregation configuration has been lost by aggr_rcm during the DR disconnect process, aggr_rcm will restore aggregation configuration based on the persistent aggregation configuration.

- The VLAN RCM plugin module (vlan_rcm)

  This RCM module will make decisions based on the current VLAN configuration. Specifically, when vlan_rcm receives a network resource offline event, it will first check whether there are any VLANs associated with the given offlined link, and propagate new sets of network offline events for the VLANs. Any module that does not agree to offline these VLANs will result in failure of the original network resource offline request. Otherwise, vlan_rcm will continue the offline process, and delete those explicitly created VLANs. Note that implicitly created VLAN should have already been deleted as a result of the ip_rcm processing.

  When a network resource online event arrives, vlan_rcm will undo all the offline process by recreating all explicitly created VLANs over that given link, and propagating network online events for all VLANs, in order to let other modules undo their offline process.

  The RCM_RESOURCE_LINK_NEW RCM event handling process will be similar to the online process, but because of the loss of the associated VLAN configuration during the DR disconnect process, vlan_rcm will have to restore VLAN configuration based on the persistent VLAN configuration. Finally, vlan_rcm will propagate a RCM_RESOURCE_LINK_NEW event for any new created VLANs.

  Note that currently the VLAN configuration (implicitly-created VLAN only at this time) is management by ip_rcm, and will be managed as part of vlan_rcm in the future.

DR and delete-phys

When a physical device is disconnected by DR, all VLANs associated with the device will be deleted, and this physical link will be removed from all associated aggregations. However, all persistent VLAN and aggregation configuration associated with the device will be kept, and will be used to restore the VLANs and aggregations if the device is later reconnected. Once the device has been disconnected, the administrator may delete the persistent configuration with dladm delete-phys; this is primarily useful if the device will not be

36 This mirrors the behavior of offlining the last IP interface in an IPMP group.

reattached. Once the persistent configuration has been removed, the link names will again be available for use.

When the administrator deletes a physical link, `libdladm` will:

- Request `dlmgmtd` to remove the [link name, linkid] mappings for all associated VLANs, and free their linkids for future use.
- Remove all associated persistent VLAN configuration.
- Update all associated persistent aggregation configuration to not be associated with this link.

DR and `rename-link`

With the new `dladm rename-link` subcommand, we will provide a way to associate a new DR connected device with an existing link configuration, making DR more flexible to use. For example, assume there is a link called `net0` which maps to a device `bge0`, and `bge0` is then disconnected. The administrator then inserts another network device, `ce0`, and connects it using DR. By renaming `ce0` to `net0`, the administrator tells the system that `ce0` is replacing `bge0`, and that all link configuration associated with `bge0` should now be associated with `ce0`. The rename operation can happen before or after `ce0` is connected:

- `rename-link` *before* connecting `ce0`

  When the administrator runs `dladm rename-link ce0 net0` before `ce0` is connected, `libdladm` will request `dlmgmtd` to remap `net0` to the device name `ce0`. Because all of the link configuration associated with `net0` is based on its linkid, this remap operation will be sufficient to cause the `ce0` device to inherit all of the configuration that had been associated with `bge0`. The link configuration repository will also be updated accordingly.

  Once the `ce0` device is connected, it will automatically inherit all configuration associated with `net0`. All VLANs and aggregations associated with `net0` will be recreated automatically by the `vlan_rcm` and `aggr_rcm` modules as a part of link configuration reestablishment. As is currently the case, other configuration such as IP interface plumbing will be restored by `ip_rcm`.

- `rename-link` *after* connecting `ce0`

  When the administrator runs `dladm rename-link ce0 net0` after `ce0` is connected, `libdladm` will first check whether `ce0` is in use by any applications, or if there are any VLANs or aggregations created over that link. If so, the rename operation will fail.

  Otherwise, the `libdladm` library will change the link name in the [link name, linkid] mapping and the link configuration repository from `ce0` to `net0`. A `RCM_RESOURCE_LINK_NEW` sysevent will then be generated, and consumed by various RCM plugin modules, which will in turn restore associated data-link and IP configuration as a part of link configuration reestablishment.

Why not Rename using `cfgadm`?

It would be possible to extend `cfgadm` to support a network-specific option that informs the system that a newly connected device should inherit all configuration associated with a previously disconnected device. This option would eliminate the need for an extra renaming step. For example, administrators would only be required to run `cfgadm -c configure -o "replace,net0" PCI8` command (where `PCI8` is the device attachment ID of the old device).

Although this option seems preferable from the administrative perspective, we chose the `rename-link` approach for the following reasons:

- Consistency with the rest of the DR user experience for controlling other DR subsystems

(such as file systems, swap and dump devices and Solaris Volume Manager). In general, the DR subsystems do some automatic configuration after a DR attach operation based on their own feature-specific policies (stored in feature-specific configuration files, such as `/etc/vold.conf` or `/etc/hostname.*`), without relying upon any options provided by the administrator. It's very common to have to run commands after doing a DR attach operation in order to get the new resources fully utilized.

- The `cfgadm` approach requires the "`replace,net0`" option to be passed to `network_rcm`, so that the name mapping information can be updated accordingly. Currently, there is no interface between `cfgadm` and the RCM plugin modules for device attach operations. Instead, RCM modules learn about the attached device by themselves, for example, by getting a sysevent generated when the device node is created. However, this approach does not allow for an "`-o`" option to be passed. As a result, entirely new interfaces would need to be created, and the RCM plugins would need to be recast to use them.
- In the current `cfgadm` architecture, each class of removable hardware is supported by a different `cfgadm` plugin (e.g, Infiniband or PCI). Further, each `cfgadm` plugin performs its own parsing of `cfgadm` "`-o`" options. Since network devices can be located on many classes of removable hardware, this means that each plugin module for each separate class would have to be changed to process the additional "`-o`" option.
- Furthermore, `cfgadm` is not the only administrative command used to perform DR. For instance, the `rcfgadm`, `addboard`, `deleteboard`, an `moveboard` commands are available on the service processors of larger systems. Most of these commands would have to be enhanced to support "`-o`". To further complicate matters, SunMC and other browser-based utilities layer on top of the service processor commands, and therefore would also require significant changes to support "`-o`".

6.2.8 System Boot Process

Multiple SMF services will be needed to reconfigure the link configuration at boot time. Reconfiguration of the `autopush` settings (together with other `linkprops`), aggregations and VLANs will be part of `svc:/network/physical` service, and they will be configured before the service plumbs any IP interfaces. On the other hand, the IP tunnel configuration will need to be done after both the `svc:/network/physical` and `svc:/network/initial` services run, as discussed in the IP Tunnel Device Driver document.

In the future, a `svc:/network/datalink-management` service will be created. It will start the `dlmgmtd` daemon, and manage the [link name, linkid] mapping information and the configuration persistence that will be required by the `svc:/network/physical` service. Therefore, `svc:/network/physical` will depend on the `svc:/network/datalink-management` service.

The `network/physical` service will walk through the link configuration and call `dladm up-aggr` and `dladm up-vlan` to recreate aggregations and VLANs.

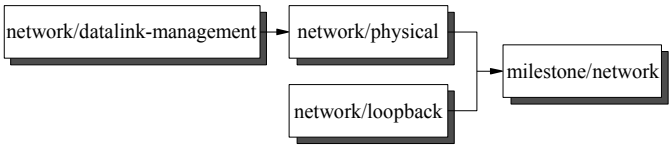

Figure 5 Network SMF Dependencies Graph

Note that because the `filesystem/root` service which is used to mounts `/usr` has an indirect dependency on the `network/physical` service, if `/usr` is a separate filessystem

from the root filesystem, it will not be mounted when dlmgmtd is started. Further, because of the indirect dependency between dlmgmtd and librcm (dlmgmtd depends on libdladm, and libdladm depends on librcm, see section 6.2.7), we will move librcm from /usr/lib to /lib, and leave a symlink behind in /usr/lib.

6.3 Miscellaneous

6.3.1 /dev Link Generation

Today, the devfsadmd daemon processes the /devices node creation event by calling into a specific devfsadm link generator module, which creates the /dev link for the /devices node based on the link creation rules. The link creation rules can be based on the node type, the driver name, or both. Currently, the rules specify that all nodes of type DDI_NT_NET will be created by the misc_link link generator module.

To preserve the backward compatibility, in addition to the DDI_NT_NET nodes of legacy devices which are created by the legacy drivers, the following DDI_NT_NET nodes will be created by the GLDv3 framework:

- style-2 and style-1 /devices nodes for physical links for GLDv3 devices;
- style-2 and style-1 /devices nodes for VLANs for GLDv3 devices;
- style-1 /devices nodes for aggregations if they are created using the "-k" option.

Note that aggregations created using names and softmacs will not have their /devices nodes created, and hence will not have their /dev links generated.

6.3.2 Sun Trunking

Sun Trunking is another link aggregation solution which provides the ability to aggregate multiple devices to work in parallel as if they were a single device. The nettr(1M) utility is used by Sun Trunking to configure aggregations. Currently, Sun Trunking provides aggregation support for the qfe, gem and ce devices. Unlike Nemo, Sun Trunking strictly requires that only the same type of devices can be aggregated. For example, four ce devices can be aggregated but one ce and one qfe device cannot be aggregated. Nemo also differs from Sun Trunking in the way aggregations are accessed. Specifically: with Sun Trunking, the administrator must define one of the aggregated links as a "trunk head". For instance, the administrator might configure ce0 as the trunk head for a Sun Trunking aggregation consisting of ce0 and ce1. After that point, ce0 represents the whole aggregation, rather than just the ce0 link – e.g., ifconfig ce0 plumb will plumb the aggregation. This is quite different from Nemo, which creates a new DLPI device that represents the whole aggregation as part of dladm create-aggr.

In order to support Sun Trunking, drivers need to support the Solaris-specific DLPI primitives: DL_AGGR_REQ, DL_UNAGGR_REQ, and DL_AGGR_IND, and the DL_NOTE_AGGR_AVAIL, DL_NOTE_AGGR_UNAVAIL notifications.

Because we encourage administrators to use Nemo aggregations instead of Sun Trunking, none of the above DLPI messages will be supported by the Nemo framework. Therefore, nettr will still need to access devices using their legacy /dev nodes instead of the /dev/net nodes, and administrators preferring Sun Trunking will not be able to benefit from vanity naming. For example, if ce0 is renamed to net0, administrators will have to use ce0 instead of net0 when using nettr to create Sun Trunking aggregations.

Note that administrators will still be able to give a vanity name to the Sun Trunking aggregation link (the trunk head). For example, the administrator will be able to create an aggregation over ce0 using nettr, and rename ce0 to net0. After the rename operation, administrators will be able to operate on the aggregation using its vanity name net0.

Although this will work, using the device name in the `nettr` configuration, but the link name in other administrative commands, will be confusing and therefore discouraged.

Because Nemo aggregation provides an equivalent but more complete aggregation solution, we would like to pursue the possibility to EOF Sun Trunking and start the transition to the Nemo aggregation solution. Unfortunately, there might be performance regression when using Nemo aggregations over softmacs, as compared to the Sun Trunking solution. Until that is resolved, we will not be able to EOF Sun Trunking.

6.3.3 Netinstall and Diskless Boot

During a netinstall or diskless client boot process, a boot interface needs to be plumbed by the kernel. As a result, the specific network device (along with other network devices, especially those using the same driver) will be attached before the `dlmgmtd` daemon is present. As is described in 6.2.5, the `net_dacf` post-attach routine will request to create the kernel data-links (`dls_vlan_t`) of these physical links. But because the `dlmgmtd` daemon is unavailable, the <link name, linkid> information kept in it is also unavailable, those kernel data-links will be created with no associated linkid. As a result, they will only be able to be inserted into two of the three hash tables: `i_dls_vlan_hash_by_dev` and `i_dls_vlan_hash_by_spa`. Therefore, each kernel data-link is associated with a `dev_t`, so that it can be opened and then used to exchange packets. But no vanity naming operations can be issued at this point.

Later, when the `dlmgmtd` daemon is started, the above kernel data-link creation process will be retried for all existing physical links. At this time, the linkid information for the physical link can be found and will be associated to a specific kernel data-link, allowing the kernel data-link to be inserted into the `i_dls_vlan_hash` hash table. The physical link creation is now complete.

6.3.4 Data-link Consumers and `libdlpi(3LIB)`

As part of this component, we will change all data-link applications in ON (for example, `ifconfig` and `snoop`) to access a link by its `/dev/net` node instead of its legacy `/dev` node. Applications will still be able to access all physical links and aggregations using their `/dev` nodes, but will not be encouraged to do so. Accessing the legacy physical link by its `/dev` node may have some unexpected side effects: because we cannot change all existing legacy drivers, there is no way for the Nemo framework to know of the operations issued on the legacy device nodes. For example, the framework will never know whether any active operations (see `dlpi(7P)`) have already been issued on the `/dev` node, and hence can not prohibit the link from being aggregated via its `/dev/net` node.

In order to update all data-link applications to access the `/dev/net` nodes, another Clearview component, PSARC 2006/436, has provided a new `libdlpi` programming interface and migrated data-link applications to use it. Specifically, `libdlpi` introduces a new `dlpi_open()` interface , which abstracts the logic to open a data-link. As part of this component, `dlpi_open()` will be changed to open the `/dev/net` nodes first, and fallback to open the `/dev` nodes if that fails. The fallback is not only necessary for backward compatibility, but also because:

- the IPoIB driver (`ibd`) has not been ported to GLDv3 yet, and will not be unified to GLDv3 as part of this component;
- some interfaces (such as `vni`) only exist in IP layer, and will not have nodes created under `/dev/net`;

6.3.5 `autopush linkprop`

Today, a `set/show-linkprop` subcommand is processed by `libdladm` in two steps. First `libdladm` processes the generic `linkprops` (currently only the `zone linkprop`), and then it processes the `wifi linkprops` operation if the `linkprop` is a wifi property.

The two steps are separate but they share almost the same processing logic. Specifically, each step has a table describing the attributes and the callbacks of each `linkprop`, and has a separate `dladm_xxx_set_prop()` function which calls into specific callback defined in the table. Although the two tables are structured a little differently, they both describe each `linkprop`'s default value, and its `setprop()`, `getprop()`, `checkprop()` callbacks.

In the future, we will generalize the table structure and `linkprops` processing logic to make it suitable for all `linkprops`, and will define a single set of `dladm_set/get_prop()` functions.

We will also introduce a new `linkprop`: `autopush` (see 4.3.4). When an administrator sets the `autopush linkprop` using the `dladm set/get-linkprop -p autopush` command, `libdladm` will look up the `autopush linkprop`'s `setprop()/getprop()` callback, which will send the `DLDIOC_SETAUTOPUSH`, `DLDIOC_GETAUTOPUSH` or `DLDIOC_CLRAUTOPUSH` ioctl to the `dld` driver. The `dld` driver will then pass the ioctl to the `dls` module, which will either allocate, retrieve, or free the kernel link `autopush` structure using the provided linkid. This kernel link `autopush` structure will be used to keep the link `autopush` settings (see 7.2.1).

When a network link is opened, STREAMS will look up the link `autopush` settings, and push the configured `autopush` modules onto the network device stream.

7 Implementation Details

7.1 Nemo Unification

7.1.1 Legacy Device Tracking

As discussed in section 6.1.2, the net_dacf module will be implemented to provide the post-attach and pre-detach functions (net_postattach() and net_predetach()) which will be invoked during the post-attach and pre-detach process of each network device instance. The net_postattach() and net_predetach() functions will call into the softmac module, as described below:

net_postattach()

The net_postattach() function will check whether the attached device is a legacy device. If so, it calls softmac_create() to create a softmac for the legacy device.

net_predetach()

When the net_predetach() function is called during the pre-detach process of the legacy device instance, softmac_destroy() will be called to delete the softmac from the system.

DACF Configuration File

The dacf.conf(4) file will be updated to specify binding rules to bind net_postattach() and net_predetach() to the DDI_NT_NET node type.

Note that an consolidate private DACF interface will be added to get the dev_t of interest. It will be used by softmac to open the underlying device.

```
/*
  * Given a dacf_infohdl_t, obtain the dev_t the instance
  * being configured.
 */
dev_t dacf_get_dev(dacf_infohdl_t info_hdl)
```

7.1.2 mac Functions

The mac module provides a set of "mac functions" in order to implement a Nemo-based network driver. This subsection describes each of the mac functions that will be called by the softmac implementation. Section 4 of the Nemo Interfaces Specification document gives the description of each mac function.

mac_register() and mac_unregister()

As we discussed, during the post-attach process of a legacy network device instance, net_postattach() will call softmac_create() to create a softmac, which in turn calls mac_register() to register the softmac with the Nemo framework. Specifically, softmac will send DL_INFO_REQ, DL_NOTIFY_REQ and DL_CAPABILITY_REQ messages to the underlying device, in order to have knowledge of all the device-specific attributes and MAC capabilities required by the Nemo framework. Although these DLPI requests should succeed, in the case of failure, softmac_create() will abort the softmac creation process, log the error to syslog, and return success, so the attach process of the legacy device instance will be able to proceed[37].

The softmac will also need to fill in its mac_callbacks_t callback functions before registering to the Nemo framework. The definition of softmac_m_callbacks is:

37 In this case, the device will not be able to be administered by dladm, and the device will not have a /dev/net node. But its legacy /dev node will be usable.

```
#define SOFTMAC_M_CALLBACK_FLAGS        \
        (MC_RESOURCES | MC_IOCTL | MC_GETCAPAB | MC_OPEN | MC_CLOSE)

static mac_callbacks_t softmac_m_callbacks = {
        SOFTMAC_M_CALLBACK_FLAGS,
        softmac_m_stat,
        softmac_m_start,
        softmac_m_stop,
        softmac_m_promisc,
        softmac_m_multicst,
        softmac_m_unicst,
        softmac_m_tx,
        softmac_m_resources,
        softmac_m_ioctl,
        softmac_m_getcapab,
        softmac_m_open,
        softmac_m_close
};
```

Note that two new MAC callbacks (shown in bold above) will be introduced specifically for softmac (more details will be discussed in section 7.1.3):

```
typedef struct mac_callbacks_s {
        ...
        mac_open_t      mc_open;        /* Open the device */
        mac_close_t     mc_close;       /* Close the device */
} mac_callbacks_t;
```

During the pre-detach of a legacy network device instance, `net_predetach()` will call `softmac_destroy()` to unregister the softmac from the Nemo framework, and destroy the softmac.

`mac_link_update()`

The `softmac` module will listen for `DL_NOTE_LINK_UP/DL_NOTE_LINK_DOWN` messages sent from the underlying device, and will call `mac_link_update()` accordingly.

`mac_rx()`

As discussed in section 6.1.3, `mac_rx()` will be called when the lower stream is of type clients-shared, and when softmac receives `M_DATA` messages from the underlying device. More data-path details will be discussed in section 7.1.5.

`mac_tx_update()`

The `mac_tx_update()` function is called by the MAC to notify the Nemo framework of a change in its packet transmission resources. The `softmac` module will call this function whenever flow control is relieved on the lower stream, to inform Nemo that it is ready to accept more data for transmission (section 7.1.5).

7.1.3 MAC Callbacks

To provide the MAC service for the softmacs registered with the Nemo framework, `softmac` will also provide all MAC callbacks required by Nemo. For example, it will provide `softmac_m_promisc()` as its `mc_promisc()` callback function. PSARC 2006/249 describes each Nemo callback function.

`mc_open()` and `mc_close()`

The `mc_open()` callback will be called when the MAC is accessed in the clients-shared mode for the first time, and the `mc_close()` callback will be called when the MAC is no longer shared by any MAC clients.

When the softmac_m_open() function is called, it will setup the client-shared lower stream to the underlying legacy device, as discussed in section 6.1.3. This lower stream will be closed when the last client-shared lower user releases its access to the softmac.

These two callbacks will be introduced specific for softmacs. It is due to the reason that the setup of the lower stream can not be deferred to the mc_start() function: in the case of an aggregation, the aggr driver might call other MAC callback functions (for example mc_unicst()) before calling mc_start(). At that time, the lower stream must be ready in order for the softmac to process the request.

mc_start() and mc_stop()

The mc_start() callback is called by the framework to enable a MAC to be able to transmit or receive packets and the mc_stop() callback is called when the framework is sure that a MAC no longer needs to transmit or receive packets. As the lower stream has already been established in softmac_m_open() and already been ready to transmit packets, softmac_m_start() function will simply return success, so will the , softmac_m_stop() function.

mc_promisc()

The mc_promisc() callback is used to turn on or off the promiscuous mode of the MAC. The softmac_m_promisc() function will send a DL_PROMISCON_REQ or DL_PROMISCOFF_REQ request down to the underlying device to turn on or off DL_PROMISC_PHYS mode, and return the result when it gets the reply from the device or the timeout occurs.

When DL_PROMISC_PHYS is enabled on a stream, all packets received by the DLPI device are passed up the stream, and all packets sent to the DLPI device are looped back. To implement the receive-side semantics, we will have to enable DL_PROMISC_PHYS for the underlying driver so that all packets can be received. However, this presents two problems:

- Traditionally, Nemo performs send-side loopback inside the MAC layer, before the packet is sent to mc_tx() (mac_txloop()). However, because DL_PROMISC_PHYS is enabled on the underlying DLPI driver, it will also perform send-side loopback.
- All mac-sharing DLS clients share the lower stream to the driver, but not all of them will have DL_PROMISC_PHYS enabled. Therefore, additional logic is necessary to ensure that only DLS clients that have DL_PROMISC_PHYS enabled receive the additional packets.

To address the first problem, Nemo will be changed to specifically disable its send-side loopback mechanism when using softmac. That is, when DL_PROMISC_PHYS is enabled but the underlying MAC is softmac, Nemo will register its transmit callback function to be mac_tx() instead of mac_txloop().

To address the second problem, the existing b_flag value MSGNOLOOP will be used, and Nemo will set this flag on all transmitting-side messages when passing them down to the softmac module. Further, Nemo's filtering will be enhanced on the receive-side to filter out the messages looped back from the underlying driver for DLS clients that are not DL_PROMISC_PHYS. Specifically, currently Nemo's dls_accept() routine already filters out packets not destined to the device's hardware address, broadcast address, or joined multicast addresses (unless DL_PROMISC_MULTI is enabled). This logic will be extended to also filter out packets with the MSGNOLOOP flag set, causing all packets looped back by the underlying driver to be filtered out.

mc_multicst() and mc_unicst()

The mc_multicst() callback is used to enable or disable reception of multicast packets for an individual multicast address. The softmac_m_multicst() function will send

either a DL_ENABMULTI_REQ or a DL_DISABMULTI_REQ down to the underlying device, and return the result when it gets the reply from the device or the timeout occurs.

The mc_unicst() callback is used to set a new physical address on the MAC. The softmac_m_unicst() function will send a DL_SET_PHYS_ADDR_REQ to the underlying device, and return the result when it gets the reply or the timeout occurs.

mc_getstat()

The mc_getstat() callback is called to retrieve the statistics of the MAC. The softmac_m_stat() function will query the underlying driver to retrieve the appropriate statistics. As described in 6.1.3.

mc_ioctl()

Today, the mc_ioctl() callback is only used to pass through M_IOCTL messages to the MAC for processing. In the future, it will pass all messages that are used during STREAMS ioctl() processing (M_IOCTL, M_COPYIN, M_COPYOUT, M_IOCDATA, M_IOCACK and M_IOCNAK messages).

The softmac_m_ioctl() function will forward all passed messages to the underlying device, and pass back any responses from the underlying device. This will allow underlying devices using "transparent" STREAMS ioctls to work as expected.

mc_tx()

The mc_tx() callback is called by Nemo to request the MAC to transmit the provided packets. In the future, the Nemo framework will directly transmit packets using the queue pointer of the underlying legacy driver if the underlying MAC is a softmac, and the softmac_m_tx() function will simply return failure. See more details in section 7.1.5.

mc_resources()

The mc_resources() callback is called to request the MAC to register its individual receive resources.

Unfortunately, the current Nemo framework ties the upper-layer-protocol-direct-transmission feature together with the interrupt coalescing support, that they share the same capability - DL_CAPAB_POLL. Therefore, to make use of the performance gain of the direct transmitting function, although softmac will not have interrupt coalescing support (see 6.1.3), softmac_m_resources() will have to resister an empty softmac_blank() function.

7.1.4 MAC Capabilities

The Nemo framework uses the mc_getcapab() callback function to obtain the MAC capabilities and capability associated data from a specific MAC. Specifically, the softmac_m_getcapab() function is used by softmac to inform Nemo of the MAC capabilities of each underlying legacy device. Table 2 shows each MAC capability, its corresponding DLPI capability. Note that apart from four existing MAC capabilities, we will add more MAC capabilities into the Nemo framework (shown in bold in the table).

MAC Capabilities	DLPI Capabilities	Comments
MAC_CAPAB_HCKSUM	DL_CAPAB_HCKSUM	
MAC_CAPAB_POLL	DL_CAPAB_POLL	
MAC_CAPAB_MULTIADDRESS	N/A	Used by GLDv3 to set multiple MAC addresses. N/A
MAC_CAPAB_LSO	DL_CAPAB_LSO	N/A

MAC Capabilities	DLPI Capabilities	Comments
MAC_CAPAB_NOZCOPY	DL_CAPAB_ZEROCOPY	The MAC does not support zero-copy.
MAC_CAPAB_TX_LOOPBACK	N/A	The MAC loops back the transmit-side packets itself.
MAC_CAPAB_LIMITED_VLAN	N/A	The MAC's VLAN capability.
MAC_CAPAB_PUTNEXT_TX	N/A	The MAC supports putnext() fastpath on the transmit-side.
MAC_CAPAB_NOLINK_UPDATE	N/A	The MAC does not support link down/up notification.
MAC_CAPAB_MDT	DL_CAPAB_MDT	The MAC supports MDT. Used for per-stream lower stream mode.
MAC_CAPAB_PERSTREAM	N/A	The MAC supports perstream lower stream scheme.
MAC_CAPAB_IPSEC	DL_CAPAB_IPSEC_AH/ESP	s10-update only.

MAC_CAPAB_HCKSUM

MAC_CAPAB_HCKSUM represents the MAC's hardware checksum off-load capability, and this capability is used to advertise the DL_CAPAB_HCKSUM DLPI capability.

The softmac module will check whether the underlying driver supports the hardware checksum off-load capability. It will keep the information in the private data of the softmac and inform the framework when the softmac_m_getcapab() is called. Nemo will then do the hardware checksum off-load operation according to the MAC's hardware checksum capability.

MAC_CAPAB_POLL

MAC_CAPAB_POLL actually indicates two different capabilities, which including support for both interrupt coalescing and the upper-layer-protocol-direct-transmission feature. Nemo will query MAC of this capability and advertise the DL_CAPAB_POLL DLPI capability accordingly. As previously described, softmac will have the MAC_CAPAB_POLL capability, and provide an empty softmac_blank() function in order to make use of the direct-transmission feature.

MAC_CAPAB_MULTIADDRESS

MAC_CAPAB_LSO

The above two MAC capabilities will not be supported by any legacy devices, hence will not advertised by any softmac.

MAC_CAPAB_NOZCOPY

Today, all Nemo MACs support the zero-copy capability. However, not all legacy devices support zero-copy. Therefore, a new MAC capability MAC_CAPAB_NOZCOPY will be introduced. The softmac will check whether zero-copy is supported by the underlying driver and will return B_TRUE if it is not supported. The Nemo framework will check this capability when negotiating the DL_CAPAB_ZEROCOPY capability with DLPI clients.

MAC_CAPAB_NOZCOPY will have no associated data.

MAC_CAPAB_TX_LOOPBACK

This capability will have no associated data. The MAC will return B_TRUE if it loops back the packets when the promisc mode is enabled at the physical level. Only softmacs will return B_TRUE for this capability.

The Nemo framework will check this capability and disable its own send-side loopback mechanism to avoid duplicate loopback.

MAC_CAPAB_LIMITED_VLAN

The MAC will return B_TRUE if it cannot support VLAN by itself. The data associated with this capability will point to either VLAN_INCAPABLE or VLAN_SIZE_CAPABLE.

A MAC is VLAN_INCAPABLE if it meets any of below conditions:

- The MAC does not allow packet of size greater than 1514.
- The MAC claims to have MAC_CAPAB_HCKSUM capability but does not adjust the checksum offset for VLAN packets.
- The MAC claims to have MAC_CAPAB_IPSEC capability (see below) but does not adjust the IP header offset for VLAN packets.

The Nemo framework will not advertise the DL_CAPAB_HCKSUM capability and DL_CAPAB_IPSEC_AH/ESP capabilities for any VLAN streams on a VLAN_INCAPABLE MAC, to avoid incorrect hardware offload behavior.

Further, Nemo will not allow the creation of VLANs over VLAN_INCAPABLE MACs unless the -f option is specified.

A MAC is VLAN_SIZE_CAPABLE if it meets all the above conditions but its legacy driver does not support VLAN PPA access. In this case, Nemo will support VLANs on such MACs using the clients-shared lower stream scheme.

MAC_CAPAB_PUTNEXT_TX

This capability will be introduced specifically for softmacs to improve the transmit-side data-path performance. The Nemo framework will call putnext() directly instead of calling the mc_tx() callback function to send packets. The data associated with MAC_CAPAB_PUTNEXT_TX will point to the queue used to send packets.

MAC_CAPAB_NOLINK_UPDATE

The MAC will return B_TRUE if it cannot support either the DL_NOTE_LINK_UP/DL_NOTE_LINK_DOWN or DL_NOTE_SPEED notifications. The data associated with this capability will point to the notifications the MAC cannot support.

This capability will be used when acknowledging the DL_NOTIFY_REQ request. Further, a device will not be able to be added into an aggregation if its MAC cannot support any of the mentioned notifications (see 6.1.5).

As we discussed in 7.1.2, in the softmac_create() function, softmac will send DL_NOTIFY_REQ to the underlying device, and determine notification types supported by the legacy device. The softmac module will return the MAC_CAPAB_NOLINK_UPDATE capability based on this information.

MAC_CAPAB_PERSTREAM

This capability will be introduced specifically for softmacs to support the per-stream lower stream scheme. The data associated with MAC_CAPAB_PERSTREAM will point to a mac_perstream_capab_t structure which keeps the pointers of the MAC callback functions to open and close the per-stream lower streams:

```
typedef struct mac_perstream_open_arg_s {
        /* Fields filled by the upper layer. */

        /* Upper stream handle. */
        void *mpo_hdl;

        /* Upperstream rx() callback, used by softmac to passup datas */
        mac_perstream_rxinfo_t *mpo_rxinfo;

        /*
         * Upper stream txupdate() callback, used by softmac to inform
the
         * avalibility of the tx resource.
         */
        mac_txupdateinfo_t *mpo_txupdateinfo;

        /* Fields returned by the specific MAC. */

        /* Lower stream handle. */
        void *mpo_perstream_hdl;

        /*
          * Lower stream callback for the upper stream to request lower-
stream
    * control messages processing.
         */
        mac_perstream_ctl_tx_t mpo_ctl_tx;

        /*
         * Lower stream m_resources callback. Called by the upper stream
to
         * update the tx resources for the DL_CAPAB_POLL capability.
         */
        mac_perstream_resources_t mpo_resources;

        /*
         * Lower stream set_rxinfo() callback. Called by the upper
stream to
         * update the rx callback then DL_CAPAB_POLL or
DL_CAPAB_SOFTRING is
         * enabled or disabled.
         */
        mac_perstream_set_rxinfo_t mpo_set_rxinfo;

        /*
         * Lower stream write-side queue, used for the upper stream to
         * transimt data messages.
         */
        queue_t *mpo_wq;
} mac_perstream_open_arg_t;

typedef int (*mac_perstream_open_t)(void *mac_private,
    mac_perstream_open_arg_t *perstream_arg, uint16_t vid);
typedef void (*mac_perstream_close_t)(void *perstream_handle);

typedef struct mac_perstream_capab_s {
        mac_perstream_open_t     mpc_open;
        mac_perstream_close_t    mpc_close;
} mac_perstream_capab_t;
```

When an upper stream is attached, the `dls` module will call the `mac_perstream_open()` function, which in turn will call the `mpc_open()` callback function, to request to open the per-stream lower stream. The `dls` module will fill in the `mac_perstream_open_arg_t` argument with the handle of the upper stream instance, and the pointers to the upper layer's callbacks. This information and the VID of the upper stream will be finally passed to the `mpc_open()` callback (the

softmac_perstream_open() function). The softmac module will then process this request, establish the lower stream, fill in the mac_perstream_open_arg_t argument with the handle of the lower stream and the pointers to the lower stream's callbacks, then return it to the upper layer.

The lower stream handle will be kept in the upper layer and will be passed to the mpc_close() callback (the softmac_perstream_close() function) to close the lower stream when the upper streams is unattached.

With the 1-1 upper stream and lower stream mapping established, the DLPI processing will be skipped in Nemo and will be performed by the underlying MAC.

MAC_CAPAB_MDT

Currently, Nemo does not support the DL_CAPAB_MDT (Mutidata Transmit) capability. However, some legacy devices do support MDT[38]. In order to make use of the legacy MDT capability, especially when the per-stream lower stream scheme is used, a new MAC capability MAC_CAPAB_MDT will be introduced. The softmac will return B_TRUE if MDT is supported by the underlying driver. The Nemo framework will advertise the DL_CAPAB_MDT capability if the MAC is MAC_CAPAB_MDT capable.

The data associated with MAC_CAPAB_MDT will point to a mac_mdt_capab_t structure which keeps the MDT related information of the specific MAC.

```
typedef struct mac_mdt_capab_s {
        t_uscalar_t mdt_flags;
        t_uscalar_t mdt_hdr_head;
        t_uscalar_t mdt_hdr_tail;
        t_uscalar_t mdt_max_pld;
        t_uscalar_t mdt_span_limit;
} mac_mdt_capab_t;
```

The Nemo framework will also need to be changed to recognize the M_MULTIDATA message on the transmit data-path and will send such messages using the same path as the M_DATA message. Ultimately, the M_MULTIDATA message will be sent down to the MAC either through the mc_tx() callback function or the putnext() function if the underlying MAC support MAC_CAPAB_PUTNEXT_TX.

In addition, we will introduce a new MDT attribute PATTR_NOLOOP to indicate the M_MULTIDATA messages can be looped back from the underlying legacy driver. The mmd_transform() function is usually called by the underlying driver to transform a M_MULTIDATA message to multiple M_DATA messages, in order to do the send-side loopback. This function will be changed to check the existence of the PATTR_NOLOOP attribute in an M_MULTIDATA message, and will mark each resulted M_DATA message using the MSGNOLOOP b_flag.

MAC_CAPAB_IPSEC

Currently, some legacy drivers have IPsec hardware authentication and encryption support, but Nemo does not yet provide the DL_CAPAB_IPSEC_AH/ESP capabilities required by IPsec hardware acceleration. Note that this IPsec acceleration support will be obsoleted in Nevada. Therefore, we will only add the MAC_CAPAB_IPSEC capability and the negotiation of the DL_CAPAB_IPSEC_AH/ESP capabilities to the Nemo framework if we ever port this component into S10 update.

The MAC_CAPAB_IPSEC capability will be introduced for the MACs capable of IPsec hardware acceleration. The data associated with MAC_CAPAB_IPSEC will point to a mac_ipsec_capab_data_t structure. More details will be discussed in section 7.1.6.

38 For now, only ce and ibd driver support DL_CAPAB_MDT.

7.1.5 Data Path

In the future, the data packets processing will be different based on the lower stream types:

Clients-shared lower stream

When a softmac is accessed in the clients-shared mode for the first time, its `mc_open()` callback (`softmac_m_open()`) will be called, which will set up the clients-sharing lower stream (see 7.1.3). The `mac` module will then be able to get the lower stream's write `queue` pointer by querying the softmac's `MAC_CAPAB_PUTNEXT_TX` capability.

Further, instead of the specific `mc_tx()` callback, the `mac_tx()` function will be called to send data messages. It will call `canputnext()` and `putnext()` to transmit as many messages as can be sent to the lower stream's write `queue`. Same as today's flow control scheme, messages fails to be sent will be returned and queued in the `dld` module, and will be sent later when `mac_tx_update()` is called to inform the availability of the transmitting resource. A softmac will call `mac_tx_update()` in the the `softmac_wsrv()` function, to indicate that the lower stream is able to handle more messages.

On the receive side, when `softmac_rput()` receives `M_DATA` messages, it will call `mac_rx()` to commit the received messages to the Nemo framework.

Further, in the clients-shared lower stream mode, the implementation of the `DL_CAPAB_POLL` and `DL_CAPAB_SOFTRING` capability support will be the same as today, and will not involve any `softmac` changes.

Per-stream lower stream

When a softmac is accessed in the per-stream mode, its `mac_perstream_capab_t.mpc_open()` callback (`softmac_perstream_open()`) will be called, which will set up the per-stream lower stream (see 7.1.3). As a result, the `dld` module will register `dld_perstream_rx()` and `dld_tx_update()` respectively to `softmac`, as the callbacks to receive packets and update the transmitting resource. On the other hand, `softmac` will inform `dld` of the lower stream write `queue` pointer, and register `softmac_perstream_resources()` and `softmac_perstream_set_rxinfo()` to `dld` , as the callbacks to add `rx` resources and update the softmac's `rx` callback function.

When the `DL_CAPAB_POLL` capability is enabled, the `dld` module will call the `softmac_perstream_resources()` callback to add the `rx` resources, and then call the `softmac_perstream_set_rxinfo()` function to update the softmac's `rx` callback to be `dld_perstream_poll_rx()`.

Likewise, when the `DL_CAPAB_SOFTRING` capability is enabled, the `dld` module will call the `softmac_perstream_set_rxinfo()` function to update the softmac's `rx` callback to be `dld_perstream_sofring_rx()`.

When the `softmac_rput()` receives M_DATA messages, it will pass the messages up using the `rx` callback pointer.

Specifically, the `dld_perstream_rx()` callback will simply call `canputnext()` and `putnext()` to pass the messages up to the upper layer protocal, so that most of the Nemo data-path processing will be skipped. The `dld_perstream_poll_rx()` function and the `dld_perstream_softring_rx()` function will call `dls_impl_t.di_rx()` function pointer[39], which is set when the `DL_CAPAB_POLL` or `DL_CAPAB_SOFT_RING` capabilities is negotiated. As a result, messages will be directly passed to the upper layer using function calls.

On the send-side, in order to commit the data messages to the underlying device to be sent out, the `dld` module will call the `dld_wput_perstream_callback()` function, which

39 Because some upstream clients (e.g. IP) have the assumption that the `db_ref` of the received message is not greater than `1`, messages will first be made a copy in that case.

will directly call putnext() with the pointer of the lower stream queue, when the queue is not flow controlled. Because a canputnext() check could be expensive, and because the softmac module is pushed right above the underlying device and _I_INSERT/_I_REMOVE is not processed in the lower stream, it will be safe to decide the flow control status of the lower stream by the following check:

```
#define CANPUTNEXT(q)   (!((q)->q_next->q_nfsrv->q_flag &
QFULL) || \
                        canput((q)->q_next))
```

If the check fails, the dld module will stop trying to send messages down to the lower stream, but instead put them into a queue. The queued messages will be resent after the dld_tx_update() callback is called. The softmac module will call dld_tx_update() (registered by dld in softmac_perstream_open()) in the softmac_rsrv() function, when the flow control on the lower stream is relieved.

The same dld_wput_perstream_callback() function will also be registered to the upper layer protocol as the tx callback when the DL_CAPAB_POLL capability or the DL_CAPAB_SOFT_RING capability is enabled. Therefore, the upper layer protocol (TCP/IP) will be able to directly send the data messages to the underlying device using putnext().

7.1.6 IPsec Cipher Hardware Acceleration

The process of supporting hardware accelerated IPsec encryption and authentication includes:

- DL_CAPAB_IPSEC_AH/ESP capability negotiation

  IPsec negotiates with a device to determine its IPsec hardware acceleration capabilities. Then, based on those capabilities, it requests the device to enable or disable specific ciphers.

- Security Association (SA) downloading

  Based on the negotiated capabilities, IPsec downloads the SA information to the device by sending a DL_CONTROL_REQ. Different types of the control operations are used to add, delete, flush, update, or query the SA information of the device.

- Per-packet cipher processing

  IPsec must be notified about the success of cipher processing for each inbound packet. Additionally, IPsec provides hints for each outbound packet to let the network device know whether cryptographic acceleration is needed. This is achieved by prepending the packets with an M_CTL containing a da_ipsec_t structure.

Today, none of the above is supported by Nemo. However, it is quite likely that future Nemo-based drivers will support hardware that is capable of hardware acceleration. Further, existing legacy devices already support hardware acceleration. Therefore, to fully support these devices, we will add hardware acceleration support both to the Nemo framework and to the softmac driver.

`softmac` Support

- The `MAC_CAPAB_IPSEC` capability

```
enum ipsec_capab_type {
       IPSEC_CAPAB_TYP_AH,
       IPSEC_CAPAB_TYP_ESP,
       IPSEC_CAPAB_TYP_MAX
};

typedef int (*mac_sadb_modify_t)(void *, uint_t, uint_t,
       dl_ct_ipsec_key_t *, dl_ct_ipsec_t *);

typedef struct mac_ipsec_capab_s {
       uint_t                  mic_nciphers;
       dl_capab_ipsec_alg_t    *mic_algs;
} mac_ipsec_capab_t;

typedef struct mac_ipsec_capab_data_s {
       mac_sadb_modify_t       mid_sadb_modify;
       mac_ipsec_capab_t       *mid_ipsec[IPSEC_CAPAB_TYP_MAX];
} mac_ipsec_capab_data_t;
```

  The `softmac` module will check the underlying device's IPsec hardware acceleration support and inform the Nemo framework of its `MAC_CAPAB_IPSEC` capability with the associcated data pointed to the `mac_ipsec_capab_data_t` structure. The specific IPsec acceleration capability data of each device will be filled in the `mid_ipsec` field and will then be used by the Nemo framework to negotiate the IPsec hardware capabilities with DLS clients.

- Security Association downloading

  The `mid_sadb_modify()` function will point to the `softmac_m_sadb_modify()` function. In this function, softmac will request to add, remove, update or flush the SA information by sending `DL_CONTROL_REQ` messages to the underlying device. It will then return success or failure based on the device's reply.

- Per-packet cipher processing

  The `soft_m_tx()` entry point will process the passed `M_CTL` message the same as the M_DATA message – simply calls `putnext()` to send it down to the underlying legacy device.

  On the receive-side, the processing of an `M_CTL` message from the underlying device will be the similar to the processing of an `M_DATA` message but with one exception: `softmac` will examine the `da_ipsec_t` structure carried in the `M_CTL` message, and drop it if its `da_type` is not `IPHADA_M_CTL`. Otherwise, pass the message upstream the same as a normal `M_DATA` message.

Nemo Support

In order to support hardware accelerated IPsec encryption and authentication, Nemo will be enhanced to provide:

- `DL_CAPAB_IPSEC_AH/ESP` capability negotiation

  The `mc_getcapab()` callback function can be used to check the `MAC_CAPAB_IPSEC` capability of certain MAC. Nemo will then use the returned information as part of performing IPsec hardware acceleration capability negotiation.

- Security Association downloading

  Nemo will maintain an SA table which will be hashed using a `[type, key]` pair. It will

process the DL_CONTROL_REQ messages from IPsec. If the message is a valid SA modify request, Nemo will modify the SA table accordingly, and then call the mid_sadb_modify() callback (embedded in the MAC_CAPAB_IPSEC data) to inform the MAC to modify its SA information. If the message is a valid SA query request, Nemo will look up the SA in the SA table and, if found, return its data.

- Per-packet cipher processing

  The dld driver will be enhanced to process per-packet cipher information.

  Transmit side

  When dld receives an M_CTL message from upstream, it will examine the da_ipsec_t structure carried in the M_CTL message. If its da_type is IPHADA_M_CTL[40], dld will know that it is a data packet which requires hardware cipher processing. The subsequent packet processing in dls layer will be mostly unchanged, except that each function on the data send-path will need to recognize the M_CTL message. Ultimately, this message will be passed to MAC by the m_tx() function. The MAC will then request the hardware to perform the required cipher processing and send the processed packet out.

  Receive side

  If the MAC has performed hardware acceleration on an incoming IPsec packet, it will prepend a M_CTL message (fill in the da_ipsec_t structure with the computed packet ICV[41]) to the data message, before passing it to mac_rx(). The message will then be passed up to the dld_str_rx_fastpath(), dld_str_rx_unitdata(), and i_dls_link_rx() functions and lastly to upstream DLS clients. The mac_rx(), dld_str_rx_fastpath(), dld_str_rx_unitdata(), and i_dls_link_subchain() functions[42] will all accept the M_CTL message as the input message parameter.

40 The IPHADA_M_CTL message is prepended to the normal data message, which is either a DL_UNITDATA_REQ message or an M_DATA message.

41 If hardware cipher processing is successful, the da_ipsec_t structure will be filled in with the computed packet Integrity Check Values (ICV). Otherwise, it will be filled in with an invalid ICV.

42 The i_dls_link_ether_rx_promisc() function will not require changes because hardware cryptographic acceleration is always disabled whenever promiscuous mode is enabled.

7.2 Vanity Naming

7.2.1 Type Definitions and Structures

Type Definitions

```
typedef uint32_t datalink_id_t;
typedef enum {
        DATALINK_CLASS_PHYS         = 1,
        DATALINK_CLASS_VLAN         = 2,
        DATALINK_CLASS_AGGR         = 4,
        ...
} datalink_class_t;
```

Kernel Data-link Structures

As previously described, each kernel data-link (`dls_vlan_t`) represents an available link on the system. A number of changes will be made to this structure, shown in bold:

```
struct dls_vlan_s {
        datalink_id_t dv_vlanid; /* Unique linkid of this link */
        datalink_id_t dv_linkid; /* linkid of the link VLAN create on */
        uint16_t      dv_vid;
        dev_t         dv_dev;    /* dev_t of the link's /dev/net node */
        dev_info_t    dv_dip;    /* dip used to create devfs minor node */
        uint_t        dv_ref;
        dls_link_t    *dv_dlp;
        kstat_t       *dv_ksp;
        kstat_t       dv_legacyksp; /* legacy kstats */
} dls_vlan_t;
```

Link `autopush` Settings Structure

A kernel link `autopush` structure will be introduced to keep the `autopush` settings of a given link:

```
struct dlautopush {
        uint_t  dap_anchor;
        uint_t  dap_npush;
        char    dap_aplist[MAXAPUSH][FMNAMESZ+1];
};
typedef struct dld_ap {
        datalink_id_t           da_linkid;
        struct dlautopush       da_ap;

#define da_anchor               da_ap.dap_anchor
#define da_npush                da_ap.dap_npush
#define da_aplist               da_ap.dap_aplist
} dld_ap_t;
```

7.2.2 `dlmgmtd` Door Service Routine

As described in section 6.2.2, the `dlmgmtd` daemon will manage linkids and link configuration, and will receive linkid management and link configuration requests from the `libdladm` library and the `dls` module through door calls. In pseudo-code, the `dlmgmtd` service routine will look like:

```
void dlmgmtd_door_service(linkmap)
{
    /*
     * linkmap carries the name mapping information of a link,
     * including linkname, linkid, linkclass and devname
     */
    switch (request) {
    case DLMGMT_CMD_CREATE_LINKID:
        /* Generate a linkid and associate it with given link info. */
        /* Input: linkname, linkclass, devname. Output: linkid */
        if (link_lookup(linkmap->linkname) != NULL)
            linkmap->linkid = LINKID_NONE; /* name conflict */
        else
            linkmap->linkid = link_id_create(linkmap);
        break;
    case DLMGMT_CMD_GETLINKID:
        /* Query link info. */
        /* Input: linkname. Output: linkid, linkclass, devname */
        link_entry = link_lookup(linkname);
        fill in linkmap with the returned link information;
        break;
    case LINK_LOOKUP_BY_DEV:
        /* Query link info by device instance name. */
        /* Input: devname. Output: linkname, linkid, linkclass */
        link_entry = link_lookup_by_dev(devname);
        fill in linkmap with the returned link information;
        break;
    case DLMGMT_CMD_GETNEXT:
        /* Get the lowest available netN name. */
        /* Input: None. Output: linkname */
        linkmap->linkname = link_gen_name();
        break;
    case DLMGMT_CMD_DESTROY_LINKID:
        /* Disassociate linkid with its linkname and free linkid */
        /* Input: linkid. Output: None */
        link_free(linkmap->linkid);
        break;
    case DLMGMT_CMD_REMAP_LINKID:
        /* Find the mapping of the given linkid, and remap it
           with the given [linkname, devname] information.
        /* Input: linkid, linkname, devname. Output: None */
        link_remap(linkmap);
        break;
    case DLMGMT_CMD_WRITECONF:
        /* Persist link mapping information to flag file */
        /* Input: linkname, linkid, linkclass, devname. Output: None */
        link_write(linkmap);
        break;
    case DLMGMT_CMD_CREATECONF:
        /* Return a reference to a configuration structure */
        link_create_conf(linkmap);
        break;
    case DLMGMT_CMD_SETATTR:
        /* Set a link's configuration field. */
        link_set_field(linkmap);
        break;
    case DLMGMT_CMD_RENAMECONF:
        /* Rename a link in SMF */
        link_rename(linkmap);
        break;
    case DLMGMT_CMD_GETATTR:
        /* Get a link's configuration field. */
        link_get_field(linkmap);
        break;
    case DLMGMT_CMD_UNSETATTR:
        /* Clear the given field from the link's configuration */
        link_unset_field(linkmap);
        break;
    }
```

7.2.3 `/dev/net` Lookup Routine

As described in section 6.2.4, `devname` will allow a name resolution routine (`devnet_lookup()`) to be associated with the `/dev/net` directory. This will be invoked to resolve a `/dev/net` node to the corresponding `dev_t`. In pseudo code, `devnet_lookup()` will look like:

```
/* Resolve a link name to link's dev_t */
/* Check if the link exists. Set linkmap->linkname to the link name */
door_upcall_to_dlmgmtd(DLMGMT_CMD_GETLINKID, linkmap);
if (linkmap->linkid != LINKID_NONE) {
       if (linkmap->devname != NULL) {
              /* Hold the device so that the kernel data-link can
                 be created during the device's post-attach process
                 This device will then be released in the /dev/net
                 filesystem's fop_inactive callback function */
              hold_device(devname);
       }
       data_link = dls_vlan_hold(linkmap->linkid);
       return data_link->dv_dev;
}
/* Implicit VLAN creation? */
if (!is_valid_vlan_name(linkname, &linkname_over, &vid))
       return FAILURE;
/* Lookup the link mapping info of linkname_over */
/* Set linkmap_over's linkname to linkname_over */
dlmgmtd_door(DLMGMT_CMD_GETLINKID, linkmap_over);
if (linkmap_over->linkid != LINKID_NONE) {
       /* Generate a new linkid and associate it with the VLAN. */
       /* Set linkmap->linkname to linkname */
       dlmgmtd_door(DLMGMT_CMD_DLS_CREATE, linkmap);
       /* Create data-link for this VLAN */
       data_link = i_dls_vlan_create(linkmap->linkid,
              linkmap_over->linkid, vid);
       return data_link->dv_dev;
}
return FAILURE;
```

7.2.4 `mac_open()`

As we already discussed, a MAC (`mac_impl_t`) represents an basic unit that used to transmit raw packets. Each MAC is uniquely identified by its MAC name.

In the future, although a MAC name will still be in the form of `<drv>N`, it will not be as predictable as of today. The reason is for softmacs (for legacy devices) and MACs of virtual link with a vanity name, `N` will be allocated internally by the Nemo framework, hence their MAC names will not be known by administrators.

Fortunately, except VLANs, there will be one-to-one mapping between a MAC and a physical link or a virtual link (e.g., aggregation), and the [linkid, MAC name] mapping will be managed by the `dlmgmtd` daemon. Specifically, when an virtual link is requested to be created by an administrator, or a physical link is created during post-attachment of an physical device, the link's linkid will be passed upto the `dlmgmtd` daemon together with the MAC name (see 7.2.2, the `DLMGMT_CMD_DLS_UPDATE` upcall). Therefore, applications will be able to request the `dlmgmtd` daemon to convert a link name or a linkid to its MAC name, and then use the returned MAC name to access the MAC.

7.2.5 Creation and Deletion of Explicit VLANs

As described in section 6.2.6, when administrators request to create VLANs, `libdladm` will issue a door call to `dlmgmtd` with a `DLMGMT_CMD_CREATE_LINKID` request, which will associate a newly allocated linkid with the link name. Upon success, `libdladm` will then send a `DLDIOC_CREATE_VLAN` request to the `dld` driver, which will call `dls_create_vlan()` to create the data-link.

Similarly, when administrators request to delete VLANs, `dladm` will issue a `DLMGMT_CMD_GETLINKID` door call to `dlmgmtd` to lookup the VLAN's linkid, and `libdladm` will then send a `DLDIOC_DELETE_VLAN` request to the `dld` driver. The dld driver wil call `dls_destroy()` to delete the data-link. Upon success, `libdladm` will issue a `DLMGMT_CMD_DESTROY_LINKID` door call to `dlmgmtd` to disassociate the [link name, linkid] mapping and free the linkid for future use.

Note that both the `dls_create_vlan()` function and the `dls_destroy()` function will take the linkid instead of the link name as the argument.

7.2.6 Creation and Deletion of Physical Data-links and Virtual Data-links

As described in section 6.2.5, as a result of the attachment of a physical device instance or the creation of a virtual data-link, mac_register() will be called, which will call dls_create() to create the data-link. Likewise, The detachment of a physical link or the deletion of a virtual link, will eventually call mac_unregister(), which will call dls_destroy() to delete the data-link.

`dls_create()`

As shown in the pseudo code below, the `dls_create()` function will issue a `DLMGMT_CMD_DLS_UPDATE` door call to `dlmgmtd`, if the given data-link's linkid is already determined.

```
dls_create(mac_handle_t mh, datalink_id_t linkid)
{
        /*
         * Update the macname of the given linkid. Two cases:
         *
         * a. A virtual link such as aggr1, iptun0, whose linkid is
         * already allocated by dladm.
         *
         * b. A physical link, whose linkid is already allocated and
         * associated with the device name before this function is called.
         */
        linkmap.mapname = mac_name(mh);
        linkmap.linkid = linkid;
        if (linkid != DATALINK_INVALID_LINKID)
            dlmgmtd_door(DLMGMT_CMD_DLS_UPDATE, linkmap);

        dls_vlan_alloc();
}
```

In the case of a physical link, a `DLMGMT_CMD_DLS_CREATE` request will be sent to `dlmgmtd` prior to `dls_create()` in order to assign the physical link a linkid. When `dlmgmtd` processes this request, it will first determine whether this is a reattachment of an existing physical link. If this is an attachment of a new link, `dlmgmtd` will check the availability of the specified link name, and generate a new link name in the form of `net`*`N`* in case the given name is already used, then allocate a new linkid and associate it with the link name:

```
/* Check to see whether it is the reattachment of an existing device */
if ((linkp = lookup_link_by_dev(linkmap.devname)) != NULL) {
        /*
         * It is the reattachment. But it might be a DR of another new
         * physical device. Update the link's macname, and phy_dev_t
         *'
        update_link(linkp, linkmap.macname, linkmap.phy_dev);
} else {
        /* A completely new device. */
        if (create_link(linkmap.devname, &linkp) == EEXIST) {
                /*
                 * The device name (default link name) has already be used.
                 */
                generate_name("net", linkname);
        } else {
                linkname =  linkmap.devname;
        }
        create_link_mapping(linkname, &linkp);
        linkmap.linkid = linkp.linkid;
}
persist_link(linkp);
```

dls_destroy()

In the dls_destroy() function, dls will call dls_vlan_free() to delete the data-link for the specific physical link or the virtual link. Because virtual links' [link name, linkid] mappings will be requested to be deleted by libdladm, and physical link will not be gone until it is DRed out, dls_vlan_free() will not request to delete any [link name, linkid] mapping.

7.2.7 Renaming Links

As described in section 5.2.4, libdladm will process link rename requests, which will be classified into one of the three type of rename requests:

- Rename an existent link name (link1) to a non-existent link name (link2).

  This type of rename request will be the most common one.

  The libdladm library will first send the DLDIOC_RENAME ioctl down to the dls module. As is described in section 6.2.6, the rename operation will proceed even if there are VLANs or aggregations created over link1. Therefore, the dls module will only check whether link1 is opened by any DLS clients. If not, dls will return success:

```
/* linkid1 is an available link and link2 does not exist yet */
int dls_rename_link_1(datalink_id_t linkid1)
{
        dvp = dls_vlan_lookup(linkid1);
        if (dls_vlan_is_hold(dvp))
                return (EBUSY);
        else
                return (0);
}
```

  The libdladm library will then issue a DLMGMT_CMD_REMAP_LINKID door call to dlmgmtd to remap link2 to link1's linkid. If the remapping succeed, libdladm will then issue a DLMGMT_CMD_RENAMECONF door call to request to update the mapping in the link configuration repository as well.

- Rename a non-existent link (link1) to a REMOVED physical link (link2).

```
int dladm_rename_link_1(const char *link1, const char *link2)
{
    send the DLDIOC_RENAME ioctl to the dld driver;
    if (dld acknowledges success) {
       /* set linkmap's linkname to be link1, query its linkid mapping */
       door_call_to_dlmgmtd(DLMGMT_CMD_GETLINKID, linkmap);
       /* Then change linkmap->linkname to link2 */
       door_call_to_dlmgmtd(DLMGMT_CMD_REMAP_LINKID, linkmap);
       door_call_to_dlmgmtd(DLMGMT_CMD_RENAMECONF, linkmap);
    }
}
```

The second type of rename request will be used before `link1` is connected to the system. Administrators issuing this request usually expect `link2`'s configuration to be automatically associated with `link1`, when `link1`'s device is attached (section 6.2.7).

Note that when `link1`'s device gets attached to the system, the attachment process actually implies two steps:

1. The device's link gets created with the link name `link1`.
2. The `link1` link is then renamed to `link2`, so that `link1` can inherit all configuration associated with `link2`.

As described before, when a physical network device gets attached to the system, its physical link will be created with a link name matching its device instance name. Specifically in the above case, the device instance name of the attached device must be `link1`, in order for a link named `link1` to be created in step `(1)`.

Therefore, the `libdladm` library will issue a `DLMGMT_CMD_SETATTR` door call to `dlmgmtd` to update the `link2`'s device instance name to be `link1`, so that when a device named `link1` is attached, its link will automatically inherit `link2`'s configuration:

```
int dladm_rename_link_2(const char *link1, const char *link2)
{
       /* Set linkmap->linkname to link2, qeury its link mapping */
       door_call_to_dlmgmtd(DLMGMT_CMD_GETLINKID, linkmap);
       door_call_to_dlmgmtd(DLMGMT_CMD_SETATTR, linkmap,
           FDEVNAME, link1);
}
```

– Rename an available link (`link1`) to a `REMOVED` physical link (`link2`) .

The third type of rename request will be used to cause a new link (`link1`) to inherit all link configuration associated with a `REMOVED` link (`link2`), after `link1` is connected to the system (section 6.2.7). This request will fail link1 is currently in use, i.e., if there is any DLS clients have `link1` open, or if there are any aggregations or VLANs created over this link.

In order for `link1` to inherit all link configuration associated with `link2`, we need to delete the <linkname, linkid> mapping for the old `link2`, and update `link1`'s mapping to be associated with `linkid2`. Further, because `link1` is currently available, its kernel state will also be updated to be associated with `linkid2`.

Spcifically, the `libdladm` library will first send the `DLDIOC_RENAME` ioctl down to the `dls` module. The `dls` module will request the mac module to check whether the operation can proceed, and update the kernel state of `link1` to be associated with `linkid2`:

The `libdladm` library will then issue a `DLMGMT_CMD_REMAP_LINKID` door call to update `link1`'s mapping to be associated with `linkid2`, which implies deletion of the mapping of the old `linkid2`. A `DLMGMT_CMD_RENAMECONF` request will then be sent

```
/* linkid1 is an available link and linkid2 is a REMOVED LINK */
int dls_rename_link_3(datalink_id_t linkid1, datalink_id_t linkid2)
{
       dvp = dls_vlan_lookup(linkid1);
       if (mac_is_used(dvp->dl_dlp->dl_mh) == EBUSY)
              return FAILURE;
       dls_vlan_update_linkid(dvp, linkid2);
}
```

to update the the link mapping in the persistent link configuration repository as well. Finally, `libdladm` will generate a `RCM_RESOURCE_LINK_NEW` sysevent to inform the RCM modules of the existence of the "new" link (`linkid2`), causing `vlan_rcm`, `aggr_rcm` and `ip_rcm` to restore the link and IP configuration associated with `link2`.

```
/* link1 is an available link and link2 is a REMOVED physical LINK */
int dladm_rename_link_3(const char *link1, const char *link2)
{
       send the DLDIOC_RENAME ioctl to the dld driver;
       if (dld acknowledges failure)
              return FAILURE
       walk all VLAN and aggregation persistent configuration to see
              whether link2 is associated with any of them, if so,
              return failure.

       /* Set linkmap1->linkname to link1, query its linkid */
       door_call_to_dlmgmtd(DLMGMT_CMD_GETLINKID, linkmap1);
       /* Remap link1's linkid to linkid2 */
       door_call_to_dlmgmtd(DLMGMT_CMD_REMAP_LINKID,
              linkmap1,linkid2);
       /* Update the persistent configuration */
       door_call_to_dlmgmtd(DLMGMT_CMD_RENAMECONF, linkmap1);
       send RCM_RESOURCE_LINK_NEW sysevent to inform linkid2;
}
```