Vanity Naming and Nemo Unification

High-Level Design Specification

Cathy Zhou

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

Network Approachability
Sun Microsystems, Inc.

Revision 1.4
Oct. 2, 2006

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 `libmicro`, `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 exeptions due to the implementaion 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 Clearview 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).

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

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.

- 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 link 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`). For physical devices, their MAC names are 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). MAC names are never exposed to administrators, and are 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`.

Note that the linkid and the MAC name do not have a one-to-one mapping: 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 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].

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.

5 Because today VLAN is 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.

3.1.3 Vanity Naming Rules and Constraints

- All links will have vanity naming support.
- Link names can be any valid style-1 DLPI provider name, which is a composition of the driver name obeying the constraints identified in the `NOTES` section of `dlpi(7P)` and a PPA number.
- 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 the `dladm rename-link` subcommand, 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.

---

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`).

In the future, the administrator will be able to configure `autopush` based on the link's link name using the `dladm autopush` subcommand (section 4.3.4), separating the `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 number, whereas `dladm autopush` will configure modules based on a link's link name. If administrators use both mechanisms at the same time, the results may be unexpected. To mitigate this problem, we will:

- Provide a `-u` option (section 4.3.4) to the `dladm autopush` subcommand that allows the `autopush(1M)` configuration for network devices stored in `autopush` file to be migrated to `dladm`.
- Run `dladm autopush -u /etc/iu.ap` command automatically during reconfiguration boot and during the DR connect operation.
- Print a warning whenever `autopush(1M)` is used to configure network devices. However, for backward compatibility, the command will still succeed.
- When a link is opened, always prefer `dladm autopush` configuration, and if it exists, ignore any `autopush(1M)` configuration. Otherwise, the `autopush(1M)` configuration will still apply (section 3.1.7).

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 autopush -m 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
lrwxrwxrwx 1 root root 36 Aug 31 16:20 bge0 ->
../devices/pci@0,0/pci1025,57@b:bge0
# dladm rename-link bge0 net0
# ls -l /dev/net
lrwxrwxrwx 1 root root 36 Aug 31 17:02 net0 ->
../devices/pci@0,0/pci1025,57@b:bge0
```

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 devinfo tree (see di_walk_minor(3DEVINFO)). There will be several issues with this behavior:

- Some applications assume that the link name is the same as the device instance name, and therefore derive link names directly from the /devices node. 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.
- Each link (including VLANs) will have its own DLPI style-1 /devices node. This might be unexpected to some applications.
- Each legacy device will have two different DLPI /devices nodes: a physical /devices node, and a softmac /devices node (created for Nemo unification; section 6.1). This might also cause some applications to behave unexpectedly.

While di_walk_minor(3DEVINFO) is a supported interface, the above assumptions made by the applications are not. However, to mitigate the risk, a library function dladm_walk() will be provided to enumerate network links on the system (section 5.2.2). In addition to changing the applications in ON to call dladm_walk() to enumerate links, a function to enumerate network links will also be published if necessary.

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 nodes for all current and future physical devices. Further, because aggregations also have /dev nodes, these will be preserved as well. 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

12 As part of Nemo, libdlpi was introduced to centralize all DLPI operations performed by applications. While it is currently private, work is underway to make it available for external use.

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"[13] 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[14].

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, any feature or performance improvement 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[15]:

13 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`.

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

15 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[16]. 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 4.3.6):

```
$ dladm show-link
LINK        CLASS     MTU       STATE    OVER
eth0        phys      1500      up       --
ib0         phys      4092      down     --
aggr0       aggr      1500      up       eth0 eth2
vlan1       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     MTU       STATE    OVER
eth0        phys      1500      up       --
eth1        phys      --        --       --
ib0         phys      4092      down     --
aggr0       aggr      1500      up       eth0 eth2
vlan0       vlan      --        --       eth0
vlan1       vlan      1500      up       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

16 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.

configuration can be explicitly deleted (section 4.3.5), which will cause `show-link -P` to 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 traffic and virtual link traffic that ultimately goes through `eth0`.

If the `link` argument is not specified to `show-link`, `dladm_walk()` will be called to get all available links (section 5.2.2), and `dladm_info()` will be called to get the attributes of a specific link (section 5.2.3). If `-P` is specified, `dladm_walk_persistent()` will be called to get all persistent links (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      STATE    SPEED    DUPLEX   DEVICE
eth0      up       100      full     bge0
ib0       down     0        --       ibd0

$ dladm show-phys -P
LINK      STATE    SPEED    DUPLEX   DEVICE    FLAGS
eth0      up       100      full     bge0      -----
ib0       down     0        --       ibd0      -----
eth1      --       0        --       ce0       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 was rejected 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   -----
...
$ dladm show-vlan -P
LINK      VID     OVER     FLAGS
vlan0     30      eth0     -----
vlan1     2       aggr0    -----
vlan2     11      eth1     f----
...
```

The `f` flag in the `show-vlan` output 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 [-L] [-s [-i interval]] [-pP] [<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, `-P`, will be added to display persistent aggregations on the system.
- Because administrators will be able to see statistics using `show-link -s`, the `-s` and `-i` options will remain supported but will be classified obsolete.
- 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.

4.2 Virtual Link Configuration

```
dladm create-aggr [-t] [-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 [-t] [-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`[17] 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 because of the implementation restriction, legacy devices which do not have `DL_NOTE_LINK_UP`/`DL_NOTE_LINK_DOWN` notifications support will not be able to be aggregated using `dladm create-aggr` (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 [-t] <link1> <link2>
```

There will be three types of link rename requests:

1. Rename an available[18] 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

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

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

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.

3. Rename an available 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 there are VLANs configured over the REMOVED physical link, and the newly added link is of type Infiniband.

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 dladm_renamelink() function (section 5.2.4).

4.3.2 create-vlan

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

As discussed in footnote 13, VLANs are currently created implicitly when applications open a physical 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[19]. 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 dladm_createvlan() function (section 5.2.5).

19 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.

4.3.3 delete-vlan

```
dladm delete-vlan [-t] <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 `dladm_deletevlan()` function (section 5.2.5).

4.3.4 autopush

```
dladm autopush -m <modulelist> <link>
dladm autopush -l <link>
dladm autopush -r <link>
dladm autopush [-d <dev>] -u <filename>
```

The `autopush` subcommand will be used to configure lists of automatically pushed STREAMS modules over the specified link.

The `autopush -m` subcommand will be used to configure the list of modules to be automatically pushed onto the stream when the specified link is opened. Upon success, the previous `autopush` setting configured on the link will be overwritten.

The `modulelist` argument will be a comma separated list of modules, its syntax will be the same as `autopush(1M)`:

- At most eight modules will be able to be pushed 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` settings configured by `autopush -m` will only take effect when the specified link is next opened. That is, the autopush modules will not be pushed on the stream if the specified link is opened before the `autopush -m` subcommand is issued. Further, the `link` argument does not need to exist on the system.

The `autopush -l` subcommand will be used to list the current `autopush` settings of the link and `autopush -r` will be used to remove the current `autopush` settings of the link.

The `autopush -u` subcommand will be used to migrate the traditional `autopush(1M)` settings for network devices stored in the specified `autopush` file to the `dladm` configuration (section 3.1.4). If the `-d` option is specified, the `autopush(1M)` settings for the specified device will be migrated to `dladm`. The `-d` option will be useful when new devices are inserted into the system.

This subcommand will call into `libdladm`'s `dladm_setautopush()`, `dladm_getautopush()` and `dladm_clrautopush()` functions (section 5.2.8).

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_deletephys()` function (section 5.2.7).

5 Library Changes

5.1 libmacadm

The libmacadm library was introduced by Nemo and macadm_walk() is the only function provided by this library. As it is unnecessary to keep a library only to house one single function, we will move macadm_walk() to the libdladm library.

```
int macadm_walk(void (*fn)(void *, const char *), void *arg,
    boolean_t use_cache)
```

In addition, several changes will be made to the macadm_walk() function: today, macadm_walk() enumerates physical MACs by walking through all DDI_NT_NET device nodes and explicitly filtering aggr driver nodes. In the future, a DLDIOC_MAC ioctl will be issued to the dld driver and dld will pass this ioctl to the mac module, which will return all the physical MACs[20]. Further, the use_cache argument, which is not currently used, will be removed[21].

5.2 libdladm

The libdladm library was also introduced by Nemo and provides APIs used by the dladm utility. 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 0 upon success and -1 upon failure.

5.2.1 Changes to Walking VLANs

Today, dladm_walk_vlan() is used to walk through all the VLANs on the system. It was introduced because currently the system does not create device minor nodes for VLANs, therefore di_walk_minor(3DEVINFO) can not be used to walk all the links on the system. In the future, all links including VLANs will be able to be enumerated by issuing a DLDIOC_LINK ioctl to the dld driver (section 5.2.2). Therefore, the need to have a separate dladm_walk_vlan() function is eliminated, and the function will be removed. The DLDIOCVLAN ioctl, which is used to support dladm_walk_vlan(), will also be removed.

5.2.2 Walking Links

```
int dladm_walk(void (*fn)(void *, const char *), void *arg);
int dladm_walk_persistent(void (*fn)(void *, const char *),
       void *arg);
```

The dladm_walk() function currently walks all DDI_NT_NET device nodes to enumerate all network links on the system. Because the link name will no longer be directly derived from the /devices node, dladm_walk() will issue a DLDIOC_LINK ioctl to the dld driver, and dld will pass the DLDIOC_LINK ioctl to the dls module, which will return all available links on the system.

Further, all persistent links will be able to be enumerated using a new function called dladm_walk_persistent(). For each persistent link, the callback function specified in the first argument will be called with the opaque pointer specified in the second argument, and the link name. The dladm_walk_persistent() function will walk all persistent links using the link configuration file (section 6.2.2).

20 Only the ioctl names used by dladm are discussed here. The ioctl structures, error conditions, and error codes will be covered in other documentation.

21 See bug 6329535.

5.2.3 Querying Link Information

```
int dladm_info(const char *link, dladm_attr_t *dap);
```

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 extended to return additional information, shown in bold below:

```
typedef struct dladm_attr {
        char          da_link[IFNAMSIZ];
        /* Device instance name */
        char          da_dev[MAXNAMELEN];
        /* DLPI media types */
        uint_t        da_media_type;
        /* DL_CLASS_PHYS, DL_CLASS_VLAN, DL_CLASS_AGGR, DL_CLASS_IPTUN */
        dl_class_t    da_class;
        uint_t        da_max_sdu;
        /* DL_LINK_UP, DL_LINK_DOWN or DL_LINK_UNKNOWN */
        uint_t        da_link_state;
        /* VLAN ID, 0 for non-VLAN */
        uint16_t      da_vid;
} dladm_attr_t;
```

5.2.4 Renaming Links

```
int dladm_renamelink(const char *link1, const char *link2, uint32_t flag);
```

The `dladm_renamelink()` function will be used to rename a link from `link1` to `link2`. The `flag` parameter can be `DL_TEMPORARY`, which indicates that the operation only applies to the running system and will not survive the next reboot.

The `dladm_renamelink()` function will issue a `DLDIOC_RENAME_LINK` ioctl to the `dld` driver.

5.2.5 Creating and Deleting VLANs

```
int dladm_createvlan(const char *vlan_link, const char *link,
       uint16_t vid, uint32_t flag);
int dladm_deletevlan(const char *vlan_link, uint32_t flag);
```

The `dladm_createvlan()` function will be used to create a specific VLAN. The VLAN will be created over the specified link name, using the specified VLAN name and VLAN ID.

The `dladm_deletevlan()` function will be used to delete a specific VLAN.

The `flag` parameter in the above functions can be `DL_TEMPORARY` to indicate that the operation only applies to the running system and will not survive the next reboot. In addition, the `flag` parameter can be set to `DL_FORCE` for `dladm_createvlan()` in order to force creation of a VLAN over a link with 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 Disconnecting Physical Hardware

```
int dladm_disconnectphys(const char *link);
```

When an administrator disconnects physical networking hardware using DR, the system must first delete all links created over it so that the DR operation can proceed. The `dladm_disconnectphys()` function will be introduced for this purpose. No ioctl will be required (section 6.2.7).

5.2.7 Deleting Physical link configuration

```
int dladm_deletephys(const char *link);
```

The `dladm_deletephys()` function will be used to delete all link configuration associated with the given `REMOVED` physical link. If the `link` argument is `NULL`, this function will delete the link configuration associated with all `REMOVED` physical links on the system. Again, no ioctl will be required (section 6.2.7).

5.2.8 Configuring `autopush`

```
typedef struct dladm_ap {
          uint_t   da_anchor;
          char     da_aplist [MAXAPUSH][FMNAMESZ+1];
} dladm_ap_t;
int dladm_setautopush(const char *link, const dladm_ap_t *ap);
int dladm_getautopush(const char *link, dladm_ap_t *ap);
int dladm_clrautopush(const char *link);
```

The three functions above will be used to get, set and clear the `autopush` settings for the specified link. Each function will issue an ioctl to the `dld` driver: either `DLDIOC_SETAUTOPUSH`, `DLDIOC_GETAUTOPUSH` or `DLDIOC_CLRAUTOPUSH`.

5.2.9 Querying and Generating Linkids

```
linkid_t dladm_getlinkid(const char *link, boolean_t generate);
```

The `dladm_getlinkid()` function will be used to query or request generation of the linkid for the specified link. If `generate` is set to false, `dladm_getlinkid()` will return the link's existing linkid, or `0` to indicate `link` does not exist on the system. If `generate` is set to true, and `link` is an existing link name, `0` will also be returned to indicate a link name conflict; otherwise, a new linkid will be generated and it will be associated with the specified link name.

5.2.10 Querying Linkids by Device Names

```
linkid_t dladm_getlinkid_bydev(const char *dev);
```

The `dladm_getlinkid_bydev()` function will be used to query the linkid of the specified physical device. It will return the corresponding link's linkid or `0` to indicate that there is no link corresponding to the specified device name. This function will only be used to support the legacy `-d` options to various `dladm` subcommands.

5.2.11 Freeing Linkids

```
int dladm_freelinkid(linkid_t linkid);
```

The `dladm_freelinkid()` function will disassociate the specified linkid with its associated link name, and then free the linkid for future use.

6 Architectural Design

We will enhance the existing Nemo framework to provide both a consistent administrative model and 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.)

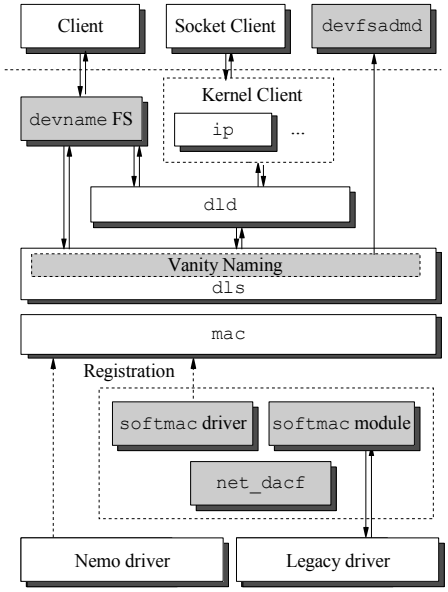

Figure 1 Enhanced Nemo Architecture

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 operations, and specify a

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.

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. Further, a `softmac` device node will be created for the legacy device during post-attach, and a `/dev/net` node will be symbolically linked to this `softmac` device node. When DLS (data-link service) clients open this `/dev/net` node, they will be able to access the specific legacy device as a Nemo MAC.

  Likewise, the `softmac` module will unregister the softmac from Nemo and the `softmac` device node together with its `/dev/net` node will be deleted during the pre-detach process, when the device is detached from the system.

  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.

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 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.

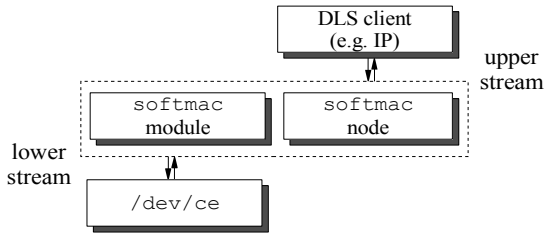

Figure 2 Upper Stream and Lower Stream

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, if an upper stream attempts to access a VLAN PPA of a VLAN-incapable legacy device, the operation will fail if `softmac` simply passes the `DL_ATTACH_REQ` message to the 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[23] 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[24], the lower stream will be set to `DL_PROMISC_SAP` promiscuous mode.

   This provides the necessary functionality, 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 the data processing is always done by the lower stream. However, 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 provides the necessary functionality, and addresses[26] the performance issues in the second approach. Therefore, we will choose this approach.

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

23 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`.

24 On the receive-side, some legacy devices incorrectly filter out VLAN-tagged packets even if the stream is set to `DL_PROMISC_SAP` mode, unless the corresponding VLAN is plumbed (6292043).

25 Legacy devices that allows VLAN PPA access .

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

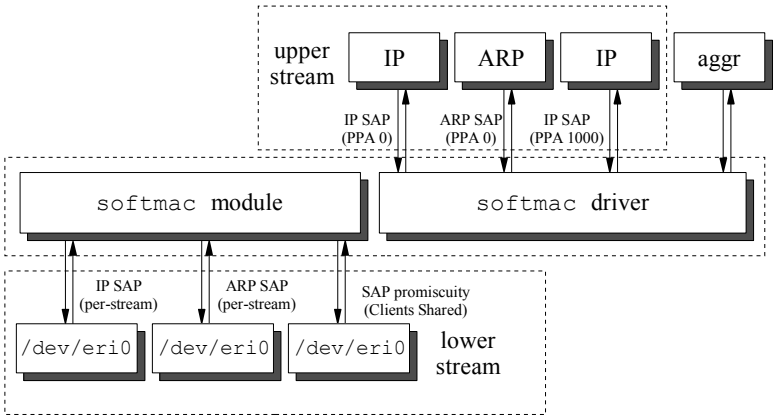

Figure 3 Lower Streams Setup

In the example, when a DLS client attempts to open a stream over a legacy device, based 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 the 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 process.

– When setting up the clients-shared stream, Nemo will be bound the stream to a specific SAP[27] and be set to 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, the handle of the upper stream will be passed down to `softmac`, and `softmac` will return the 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, hence it 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 the `DL_PROMISCON_REQ` request down to the underlying device to turn on `DL_PROMISC_PHYS` mode, and it will wait for the reply from the device for a certain time interval. The `softmac_m_promisc()` function will return success if the device replies with

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

DL_OK_ACK, or will return failure if the device replies with DL_ERROR_ACK or the timer expires.

If the corresponding lower streams are of type per-stream, Nemo will pass down the control messages to 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.

The data-path will not change if the lower streams are of type clients-shared. The softmac module will enable DLIOCRAW mode on the lower stream to make sure legacy devices do not strip off the link layer header for both inbound and outbound packets[33]. 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 lower stream to the Nemo framework.

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 the "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.

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

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.

33 Some legacy devices mishandle VLAN traffic in DLIOCRAW mode. On the receive-side, even if the stream is in DLIOCRAW mode, the VLAN tag is stripped off when the driver passes the packets to the DLS consumer (4722784, 6306794). On the send-side, some legacy devices do not keep the VLAN tags in the VLAN packets intact in DLIOCRAW mode. Instead, they replace the VLAN tag according to the VLAN PPA of the specific stream on which the packets are sent.

will not do any packet queueing. Instead, it will simply pass M_DATA messages up to the Nemo framework for further processing.

If the lower stream is of type per-stream, most of the Nemo processing will be skipped. The DLD layer will call putnext() to pass the send-side packets directly to the corresponding lower stream. Likewise, when softmac receives a packet from a per-stream lower stream, it will call putnext() to pass that packet directly to the corresponding upper stream. Further, both the receive-side and send-side will leverage the Nemo POLL capability. On the send-side, the DLS client will register the POLL callback function dld_wput_perstream_callback(), which will call putnext() to pass the packets directly to the corresponding lower stream. On the receive-side, if the POLL capability is enabled, softmac will directly pass the received packets to the input function provided by IP.

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). The MAC tells Nemo the set of statistics it supports when it registers with Nemo, and Nemo updates both the MAC and data-link kstats by querying the MAC for statistics.

When a softmac is registered with Nemo, the softmac module will inform the framework of the list of statistics the softmac can support: it will first query the underlying device using the statistics names defined in the Nemo MAC statistics names array[34] to determine what statistics will be able to be supported. 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.

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()).

Note that the kstat support of legacy devices will not be changed. However, in order to avoid potential naming conflict with the legacy device kstats, the softmac data-link kstats will be changed to prefixed with "net/" instead of only the link name. Otherwise, for a link without the vanity name, its data-link kstats would be created with the default link name (the device instance name), which would conflict with the name of the legacy device kstats. In order to ensure consistency between Nemo devices and legacy devices, all the data-link kstats (including kstats for legacy devices and Nemo devices) will be prefixed with "net/".

34 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 opening a legacy device's `softmac` node. Using the `autopush` configuration below 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
```

Although `/dev/net/net0` will point to a device node whose major number is `softmac` instead of `ce`, when an application opens `/dev/net/net0`, the `softmac` driver will open the underlying device `ce0`, which causes all the specified `autopush` modules (`vpnmod` in this case) to be pushed between the underlying `ce` driver and `softmac` driver, thus providing backward compatibility.

Note that this approach will not work if `autopush`-ed modules assume that the `ip` module can be found by walking upstream through `q_next`. However, inspecting private `ip` data structures via `q_next` itself is questionable, and has never been a supported interface.

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.

In order to address the problem, we will introduce a `DL_IOC_VLAN_CAPAB` ioctl for network devices. The underlying driver will acknowledge this ioctl if it can receive and send packets 4 bytes longer than the driver's advertised maximum SDU. By default, creating VLANs over devices which are incapable of acknowledging `DL_IOC_VLAN_CAPAB` will fail. If administrators are aware of the issue we described above and are willing to configure MTU carefully on each system of 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 process the `DL_IOC_VLAN_CAPAB` 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 aggregation support, with a few exeptions (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, legacy devices which do not support DL_LINK_NOTE_UP/DOWN notifications will not be able to be aggregated using dladm, because such notifications are required by the Nemo aggregation support (in order to attach/detach specific port from an aggregation). 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 solution.

6.1.6 Support of Other Media Types

Although the softmac approach is generic to any DLPI media type, the Nemo framework currently only supports DL_ETHER MAC drivers. Another component of Clearview, IP Tunnel device, will implement a MAC plugin architecture, allowing future adoption of other media types to the Nemo framework. Details of the MAC plugin architecture are discussed in the MAC Type Independent Nemo Architecture document[35].

This component will support DL_ETHER and DL_IB (Infiniband) legacy drivers[36]. Note that we will not implement MAC plugins for other media types (such as FDDI and Token Ring) in our initial delivery. It can be done as part of follow-up work provided that hardware required by the development and testing work become available. Today, the use of devices of other media types is very rare. Further, the legacy /dev node of those devices will remain accessible, thus no regression will be introduced.

6.2 Vanity Naming

6.2.1 Overview

The existing Data-link Services Module (dls) and the devfsadmd daemon will both be enhanced 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 being developed by the I/O group 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 /devices node (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 GLD kernel structures will refer to linkids so that they will not need to be updated when the link name is changed.

The devfsadmd daemon will be used to manage the [link name, linkid] mapping for each link on the system. For physical links, it will also keep the associated device instance name (section 6.2.5).

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

35 This document will be updated to reflect the new architecture once the MAC plugin design is finalized.

36 Note that legacy wireless devices appear as DL_ETHER devices and thus do not require special support from softmac.

- 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.

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 link name and the SPA, the kernel data-link structure will also be updated to keep:

- Link class

  Currently, `phys`, `aggr`, `iptun` or `vlan`.

- Major and minor number of the `/devices` node

  Used to resolve the `/devices` node from the link name (section 6.2.4).

Once a kernel data-link structure is created, it will be added to the kernel data-link hash table. Below is an example of the information that would be contained in this hash table on a running system:

```
link_name linkid  MAC      VID  class   major    minor
net0        1     ce0/0     0   phys    softmac  1
aggr1       2     aggr0/1   0   aggr    aggr     1
vlan1       3     aggr0/1   2   vlan    aggr     1
tunnel0     4     tun0/0    0   iptun   tun      1
```

Link Configuration File

In the future, a centralized link configuration file, `/etc/datalink.conf`, will be used to store all `dladm`-related persistent link configuration, including vanity name mapping information. Note that while the link configuration can be moved into SMF, and this seems to be more aligned with the Solaris management model, the current SMF modle does not have the facility to prevent administrators updating un-stablized properties, or read-only properties, or internal properties which should not be exposed to the end-user. As a result, an incidental update of such properties could cause serious problem to the system (for example, the linkid is fixed during the life-time of a link and must not be updated) . Therefore, the adoption of an SMF repository for data-links is outside the scope of this component.

As a result of the centralized link configuration file, the existing `/etc/aggregation.conf` file is no longer needed and will be removed. Note that the `datalink.conf` file will be automatically updated by `dladm` and must not be directly edited.

Below is an example `/etc/datalink.conf`:

```
#link_name  linkid  device_name class  dladm_configuration
 net0         1     ce0         phy    autopush=vpnmod
 aggr1        2     --          aggr   key=1, policy=L4, nports=1,
                                       ports-list=1, macaddr=auto,
                                       lacp-mode=off, lacp-timer=short
 vlan1        3     --          vlan   over=2, vid=2
 tunnel0      4     --          iptun  type=configured-ipv4,
                                       tsrc=my-addr, tdst=peer-addr
```

Using `/etc/datalink.conf`, the system will be able to recreate all persistent links during system boot, rebuilding the vanity name mapping information required for `/dev/net` node resolution in the process.

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.

6.2.3 DLPI Device Nodes

As a result of this component, Nemo will create a DLPI style-1 device node (under `/devices`) 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 be 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 link rename. 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, neither the `/dev/net` node nor the `/devices` node exist yet for this VLAN. The system must create the `/dev/net` and `/devices` nodes 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. While the `devname` project is still a work in progress and some of the design details have not been finalized, we are working with the `devname` team to make sure it meets our requirements.

Dynamic Name Resolution

The `devname` filesystem will provide a Directory-Based device Name Resolution (DBNR) infrastructure to resolve a node name to its `/devices` node: 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 its physical `/devices` node. 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.

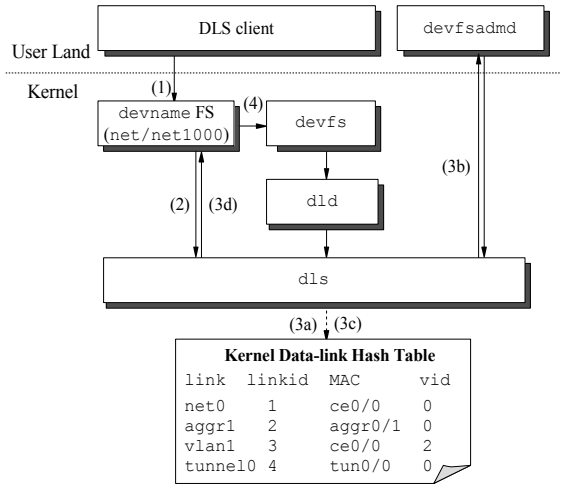

Figure 4 Vanity-name Node Access

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 to obtain the vanity name mapping for `net1000`.
3. To resolve `net1000` to its `/devices` node, `dls`:
   a. Looks up `net1000` in the kernel data-link hash table.
      → If it is found, then there is currently a link with the name `net1000` on the running system. The major and minor number (`dev_t`) associated with that link is returned.
      → If it is not found, then `dls` checks whether the link name should be treated as an implicit VLAN (PPA > `999`):
         → If it should not be, then failure is returned.
   b. Makes a door upcall to `devfsadmd` to check for a link name conflict.
      → If there is already a link with that name, then failure is returned.
      → If there is not a link with that name, then a linkid is allocated by `devfsadmd` and associated with `net1000`.
   c. Looks up `net0`'s corresponding MAC in the kernel data-link hash table, creates a VLAN link named `net1000`, which has the allocated linkid and `net0`'s MAC information.
   d. Returns the `dev_t` of `net1000`.
4. The DLS client opens the driver with the `dev_t` of the physical node, and proceeds with other DLPI operations.

Note that unlike the `/dev` node, which is symbolically linked to the physical `/devices` node, the `/dev/net` node will never be created. Each time a DLS client accesses the `/dev/net` node, the `/dev/net` name resolution function will be invoked, resolving the `/dev/net` name to its `/devices` node.

Therefore, 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 (see below).
- 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. The core logic of `dls_vlan_create()` will remain unchanged, except that it will also create the `/devices` minor node for the link. In order to perform this creation, `dls_vlan_create()` will be updated to take a link name, link class, linkid, SPA, and device node minor name as 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.

The `dls` function will first issue a door upcall to `devfsadmd` to map the device instance name to a linkid. If the device instance name does not exist, `dls` will then request `devfsadmd` to 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`[37].

The `dls` module will then call `dls_vlan_create()`.

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

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

Again, `dls` will first lookup the linkid using the device instance name, and then delete the kernel data-link with the specified linkid. Because the detachment of a physical device instance only indicates that the corresponding physical link is currently not available, the [link name, linkid] mapping for the physical link will still be kept in the `devfsadmd` 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 `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 `devfsadmd` to generate a new linkid and associate it with the new link.
- For operations that use an existing link, `libdladm` will issue a door call to `devfsadmd` 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`, `liblaadm` will issue a door call to `devfsadmd` 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, `liblaadm` will request `devfsadmd` 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, `liblaadm` 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 both the link name and the linkid of the aggregation, and the linkids of the aggregated links.

  The `aggr` driver will process the `LAIOC_CREATE` ioctl. It will choose `aggr0/<key>` (if the `key` argument is specified) or the lowest available `aggr0/1000+`*N* as the aggregation's MAC name. After `aggr` successfully creates the aggregation, it will register the MAC with Nemo, and request `dls` to create the aggregation's kernel data-link.

  When `liblaadm` receives a successful acknowledgement from the `aggr` driver, it will update the link configuration file with the persistent aggregation.

37 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.

Note that both the aggregation's link name and its linkid will not be kept by the aggr driver. However, the linkid will be used by the driver to query the SPA when it processes other dladm requests, such as the two operations below.

- Displaying Aggregations

  For show-aggr, liblaadm will send an LAIOC_INFO ioctl to the aggr driver to obtain aggregation information. The aggr driver will query the dls module to get the link names of the aggregation and the aggregated links by using their linkids.

- Operations using Aggregation Link Names

  For all other link aggregation subcommands (add-aggr, remove-aggr, delete-aggr, modify-aggr), liblaadm will first query devfsadmd to get the linkid of the aggregation, and then 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 devfsadmd 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 look up the MAC name corresponding to the link the VLAN is created on. If the given VLAN ID is already in use on the MAC, the VLAN creation will fail. When libdladm receives a successful acknowledgement from the dld driver, it will update the link configuration file with the persistent VLAN.

If the VLAN is deleted by dladm delete-vlan, libdladm 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 available link to a link name that does not exist on the system.

The libdladm library will first send a DLDIOC_RENAME_LINK ioctl to the dld driver, which it will pass to the dls module. If this specified link is currently in use by any applications, the rename operation will fail. Otherwise, dls will update the corresponding kernel data-link's link name. The libdladm library will then inform the devfsadmd 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 configuration file.

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.

autopush over Link Names

For dladm autopush, libdladm will query the linkid of the specified link, and send the DLDIOC_SETAUTOPUSH, DLDIOC_GETAUTOPUSH or DLDIOC_CLRAUTOPUSH ioctl to the dld driver.

The dld driver will pass the ioctl to the dls module, which will either allocate, retrieve, or

free the kernel link autopush structure using the specified linkid. This kernel link autopush structure will be used to keep the link autopush settings.

When libdladm receives a successful acknowledgement from the dld driver, it will update the link configuration file with the new autopush configuration.

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

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/<*link_name*>[38]. The network_rcm module will require an additional step to map the <*device_instance_name*> to the <*link_name*>. When a link is renamed, a RCM_RESOURCE_NETWORK_UPDATE sysevent will be generated by libdladm, and will be consumed by network_rcm, which will update the name of corresponding network resource.

Link Configuration Reestablishment

The network_rcm offline routine is invoked when a physical device is disconnected by DR. In the new code, this will call dladm_disconnectphys() to delete any available associated VLANs, and remove the device from any associated aggregations[39], in order to allow the corresponding device instance to be detached. As a result, the [link name, linkid] mappings for any non-persistent VLANs will be deleted from devfsadmd. If the device is the last device left in the aggregation, administrators will have to force the device disconnect operation, which will in turn force the aggregation to be deleted.

The network_rcm online routine (also known as undo_offline()) is used to reverse the offline operation. It will restore all associated persistent VLANs and aggregations based on the link configuration file.

Note that when a device is successfully disconnected, the RCM framework loses the device's network resource information. Therefore, when the physical device is reattached, devfsadmd needs to generate a RCM_RESOURCE_NETWORK_NEW sysevent with the device name of the new device. This sysevent currently gets consumed by the ip_rcm module, which in turn plumbs all relevant IP interfaces. In the future, this sysevent will be consumed by the network_rcm module, which will map the device name to the link name, and restore all associated persistent VLANs and aggregations based on the link configuration file. The network_rcm module will then propagate this event by generating a RCM_RESOURCE_LINK_NEW RCM event with the link name, which will in turn be consumed by ip_rcm to restore associated IP configuration.

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

39 Currently, the DR disconnect fails if there are any available aggregations created over the device being disconnected.

DR and `delete-phys`

When a physical device is disconnected by DR, all VLANs associated with the device will be deleted. However, all persistent VLAN configuration associated with the device will be kept, and will be used to recreate the VLANs 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 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 `devfsadmd` to remove the [link name, linkid] mappings for all associated VLANs, and free their linkids for future use.
- Remove all associated persistent configuration from the link configuration file.

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 `devfsadmd` 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 file will also be updated accordingly.

  Once the `ce0` device is connected, it will automatically inherit all configuration associated with `net0`. All VLANs associated with `net0` will be recreated automatically by the `network_rcm` module 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, the `dls` module will first lookup `ce0` in the kernel data-link hash table. It will then 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 `dls` module will update the link name of the kernel data-link structure from `ce0` to `net0`. The [link name, linkid] mapping in `devfsadmd` and the configuration file will also be updated. A `RCM_RESOURCE_NETWORK_UPDATE` sysevent will then be generated, and consumed by the `network_rcm` module. The `network_rcm` module will update the name of the network resource, restore all relevant link configuration and propagate this event by notify the `RCM_RESOURCE_LINK_NEW` RCM event to the `ip_rcm` module, which will in turn restore associated IP configuration.

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. The `autopush` settings, aggregations and VLANs will need to be configured before the `svc:/network/physical` service runs, since the service plumbs all 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.

Therefore, a `svc:/network/link-init` service and a `svc:/network/iptun` service will be created. The former will have the `svc:/network/physical` service depend on it, while the latter will depend on the `svc:/network/initial` service[40].

40 Note that the `svc:/network/initial` service depends on the `svc:/milestone/network:default` service, which in turn depends on the `svc:/network/physical` service.

The svc:/network/link-init service will first issue door calls to provide devfsadmd with the vanity name mapping information of each physical link. It will then walk through the link configuration and call dladm create-aggr and dladm create-vlan to recreate aggregations and VLANs. The svc:/network/iptun service will call dladm up-iptun to request to recreate IP tunnels.

Figure 5 Network SMF Dependencies Graph

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.

As a result of this component, each class of link (phys, aggr, iptun or vlan) will have a DLPI style-1 DDI_NT_NET /devices node created. However, some of the /devices nodes will not need a corresponding /dev link:

- VLANs and IP Tunnels

  Since there is no backward compatibility issue, VLANs and IP Tunnels will only be accessible from the /dev/net namespace.

- softmac /devices nodes

  Each legacy device already has a /dev node linked to its legacy physical /devices node. The softmac /devices nodes will only be used for Nemo unification and will be accessed using the /dev/net nodes.

Therefore, in the new code, misc_link will check both the link class and the driver name of the /devices node, and will only create the /dev link if the class is aggr or phys, and if the driver name is not softmac.

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 solution that Sun Trunking provides, we are working with the SSG/NSG group to EOF Sun Trunking and start the transition to the Nemo aggregation solution. The transition process may also include a automated configuration conversion utility to aid customers to upgrade their Sun Trunking configuration to the new one seamlessly.

6.3.3 Diskless Boot

During the boot process of a diskless client, a boot interface needs to be plumbed by the kernel. This happens before the devfsadmd daemon is present. Further, because one cannot rename a link if it is in use, the boot interface's name will not be able to be renamed later after the boot.

Because of the above issues, supporting vanity naming on the network interface used for diskless boot would require significant additional engineering resources. As such, this will not be supported by this project. However, note that vanity naming of any (non-boot) interfaces will work as expected.

6.3.4 Data-link Consumers

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.

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 just 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` 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[41].

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:

41 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_NOTIFY_IND` 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 one of the below conditions are met:

1. Softmac receives `M_DATA` messages from the underlying device, and
   - Interrupt coalescing is disabled.
   - Interrupt coalescing is enabled, but the interrupt coalescing message queue reaches its high-water mark.
2. The interrupt coalescing timer expires.

`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 (see `soft_m_tx()`).

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_start()` as its `mc_start()` 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 by any MAC client for the first time, and the mc_close() callback will be called when the MAC is no longer used by any MAC clients.

When a softmac is accessed by any MAC client for the first time, the softmac_m_open() function 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 introduces specific for softmacs. It is due to the reason that the lower stream can not be defered 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. The softmac_m_start() and softmac_m_start() function will simply return success.

mc_tx()

The mc_tx() callback is used by the Nemo framework to request the MAC to transmit the provided packets. The softmac_m_tx() function will transform the provided packets and send them down the lower stream. If the lower stream is flow controlled, softmac_m_tx() will return failure.

mc_promisc() and mc_multicst()

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(). However, because DL_PROMISC_PHYS is enabled on the underlying DLPI driver, it will also perform send-side loopback.
- All 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.

To address the second problem, the existing b_flag value MSGNOLOOP will be used, and softmac will set this flag on messages when passing them down to the underlying driver. 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.

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.

mc_unicst()

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.

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_resources()

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

The softmac module will virtualize a receive ring for the softmac, and register the softmac_blank() coalescing function. The softmac_blank() function will be used by Nemo to start the soft interrupt coalescing timer and put the softmac into the interrupt coalescing state.

7.1.4 MAC Capabilities

The Nemo framework uses the mc_getcapab() callback function to obtain the MAC capabilities and 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.

Currently, only two MAC capabilities are defined by the Nemo framework:

MAC_CAPAB_HCKSUM

MAC_CAPAB_HCKSUM represents the MAC's hardware checksum off-load 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 indicates the MAC can support interrupt coalescing. As previously

described, softmac will provide softmac_blank() interrupt coalescing function, therefore, all softmacs will have the MAC_CAPAB_POLL capability.

Apart from the above two MAC capabilities, we will add more MAC capabilities into the Nemo framework:

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 zero-copy 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 to send packets instead of calling the mc_tx() callback function. 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).

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 int (*mac_perstream_open_t)(void *, mac_perstream_open_arg_t *);
typedef void (*mac_perstream_close_t)(void *);

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 pointer to the upper layer's receiving callback function. This information will be finally passed to softmac. The softmac module will then process this request, open the lower stream, fill in the mac_perstream_open_arg_t argument with the handle of the lower stream and the softmac's transmitting callback function pointer, and 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() function to close the stream when the upper streams is unattached.

MAC_CAPAB_MDT

Currently, Nemo does not support the DL_CAPAB_MDT (Mutidata Transmit) capability. However, some legacy devices support MDT[42]. In order to make use of the legacy MDT capability, 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 then negotiate the DL_CAPAB_MDT capability with DLPI clients.

The data assoicated 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 additon, 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

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

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. We will add the negotiation of the DL_CAPAB_IPSEC_AH/ESP capabilities to the Nemo framework.

The MAC_CAPAB_IPSEC capability will be introduced for the MACs capable of IPsec hardware acceleration. The data assoicated with MAC_CAPAB_IPSEC will point to a mac_ipsec_capab_data_t structure. More details will be discuessed in section 7.1.5.

7.1.5 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[43], 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[44]) 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[45] will all accept the M_CTL message as the input message parameter.

7.1 Vanity Naming

7.2.1 Type Definitions and Structures

Type Definitions

```
typedef id_t linkid_t;
typedef enum dl_class {
       DL_CLASS_PHYS,
       DL_CLASS_VLAN,
       DL_CLASS_IPTUN,
       DL_CLASS_AGGR,
} dl_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:

43 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.

44 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.

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

```
struct dls_vlan_s {
        char            dv_name[IFNAMSIZ];
        linkid_t        dv_linkid;    /* Unique linkid */
        dl_class_t      dv_class;     /* The link class */
        dev_t           dv_dev;       /* dev_t of the link's minor node */
        uint_t          dv_ref;
        dls_link_t      *dv_dlp;
        uint16_t        dv_vid;
        kstat_t         *dv_ksp;
} 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 dls_autopush_s {
        linkid_t        da_linkid;
        uint_t          da_anchor;
        uint_t          da_modcnt;
        char            da_modlist[MAXAPUSH][FMNAMESZ+1];
} dls_autopush_t;
```

7.2.1 `devfsadmd` Door Service Routine

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

```
void devfsadm_link_door_service(linkmap)
{
    /*
     * linkmap carries the name mapping information of a link,
     * including linkname, linkid, linkclass and devname
     */
    switch (request) {
    case LINK_ID_CREATE:
        /* 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 LINK_LOOKUP:
        /* 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 LINK_GEN_NAME:
        /* Get the lowest available netN name. */
        /* Input: None. Output: linkname */
        linkmap->linkname = link_gen_name();
        break;
    case LINK_FREE:
        /* Disassociate linkid with its linkname and free linkid */
        /* Input: linkid. Output: None */
        link_free(linkmap->linkid);
        break;
    case LINK_REMAP:
        /* 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 LINK_PERSIST:
        /* Persist link mapping information to the config file */
        /* Input: linkname, linkid, linkclass, devname. Output: None */
        start an asynchronous task to write link mapping information;
        break;
    }
}
```

7.2.2 Creation and Deletion of Virtual Data-links and Explicit VLANs

As described in section 6.2.6, when administrators request to create virtual links or VLANs, `libdladm` will issue a door call to `devfsadmd` with a `LINK_ID_CREATE` request, which will associate a newly allocated linkid with the link name. Upon success, `libdladm` will then send a link creation request to either the `dld` driver or the virtual device driver, which in turn will call `dls_vlan_create()` to create the data-link.

Similarly, when administrators request to delete virtual links or VLANs, `libdladm` will issue a `LINK_LOOKUP` door call to `devfsadmd` to lookup the link's linkid, and then send a link delete request to either the `dld` driver or the virtual device driver. The driver will call `dls_vlan_delete()` to delete the data-link. Upon success, `libdladm` will issue a `LINK_FREE` door call to `devfsadmd` to disassociate the [link name, linkid] mapping and free the linkid for future use.

The `dls_vlan_create()` and `dls_vlan_destroy()` functions called above will be changed to take a linkid instead of a link name. Further, `dls_vlan_create()` will take an additional argument specifying the minor name of the link's `/devices` node:

```
int dls_vlan_create(const char *linkname, linkid_t linkid,
          const char *macname, uint16_t vid, const char *nodename);
int dls_vlan_destroy(linkid_t linkid);
```

7.2.3 `/dev/net` Lookup Routine

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

```
/* Resolve "linkname" to its /devices node */
/* Check if the link exists. Set linkmap->linkname to linkname */
door_upcall_to_devfsadmd(LINK_LOOKUP, 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_lookup(linkname);
       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 */
devfsadmd_door(LINK_LOOKUP, linkmap_over);
if ((linkmap_over->linkid != LINKID_NONE) &&
    (data_link_over = dls_vlan_lookup(linkname_over)) != NULL)
{
       /* Generate a new linkid and associate it with the VLAN. */
       /* Set linkmap->linkname to linkname */
       devfsadmd_door(LINK_ID_CREATE, linkmap);
       /* Create data-link for this VLAN */
       data_link = dls_vlan_create(linkname, linkmap->linkid,
              data_link_over->mac, vid, linkname);
       return data_link->dv_dev;
}
return FAILURE;
```

7.2.4 Creation and Deletion of Physical Data-links

As described in section 6.2.5, in addition to registering and unregistering softmacs for legacy devices, the same `net_dacf` functions (`net_postattach()` and `net_predetach()`) will be used to create and delete kernel data-links for physical network devices.

`net_postattach()`

As shown in the pseudo code below, before `net_postattach()` requests `dls` to create the data-link for the attached device instance, it will first issue a `LINK_LOOKUP_BY_DEV` door call to `devfsadmd` to check whether the device instance name exists, and if so, get its linkid and link name. Otherwise, `dls` will derive the physical link's data-link name, and request to associate this name with a newly generated linkid. Finally, the kernel data-link for this physical link will be created.

```
/* A device named as devname is attached, first check if it is new */
/* Set linkmap->devname to devname */
devfsadmd_door(LINK_LOOKUP_BY_DEV, linkmap);
if (linkmap->linkid != LINKID_NONE) {
       /* Not a new device, create data-link, vid=0 */
       dls_vlan_create(linkmap->linkname, linkmap->linkid, macname,
           0, devname);
       return;
}

/*
 * A new physical device, derive its data-link name, First check if
 * the device instance name conflicts with existing names
 */
linkname = devname;
/* Set linkmap->linkname to linkname */
devfsadmd_door(LINK_LOOKUP, linkmap);
if (linkmap->linkid != LINKID_NONE) {
    /* Name conflicts, put a new netN name in linkmap->linkname */
    devfsadmd_door(LINK_GEN_NAME, linkmap);
}

/*
 * Generate a linkid and associate it with linkmap->linkname
 * Set linkmap->linkname to linkname, linkmap->devname to devname
 */
devfsadmd_door(LINK_ID_CREATE, linkmap);

/* Create data-link, vid=0 */
dls_vlan_create(linkmap->linkname, linkmap->linkid, macname,
       0, devname);
/* Persist link information */
devfsadmd_door(LINK_PERSIST, linkmap);
```

net_predetach()

In the net_predetach() function, dls will delete the data-link for the specific device instance without deleting its [link name, linkid] mapping in devfsadmd.

7.2.5 Renaming Links

As described in section 5.2.4, libdladm will process link rename requests from administrators and send DLDIOC_RENAME_LINK ioctls to the dld driver, which will be passed to dls to be classified into one of the three type of rename requests:

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

  This type of rename request will be the most common one. 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 by calling the link_is_open() function. If not, dls will update the link name of the corresponding kernel data-link structure:

```
/* link1 is an available link and link2 does not exist yet */
int dls_rename_link_1(const char *link1, const char *link2)
{
       data_link = dls_vlan_lookup(link1);
       /* check whether the link is opened by any DLS client */
       if (link_is_open(data_link))
              return FAILURE;
       dls_vlan_update_name(data_link, link2);
}
```

  The libdladm library will then issue a LINK_REMAP door call to devfsadmd to remap link2 to link1's linkid. As the result, the link mapping in the persistent link

configuration file will also be updated. Finally, a RCM_RESOURCE_NETWORK_UPDATE sysevent will be generated to inform the network_rcm module to update the name of the corresponding network resource:

```
int dladm_rename_link_1(const char *link1, const char *link2)
{
    send the DLDIOC_RENAME_LINK ioctl to the dld driver;
    if (dld acknowledges success) {
       /* Set linkmap->linkname to link1, return link1's linkname
          and devname */
       door_call_to_devfsadmd(LINK_LOOKUP, linkmap);
       /* Then change linkmap->linkname to link2 */
       door_call_to_devfsadmd(LINK_REMAP, linkmap);
       send RCM_RESOURCE_NETWORK_UPDATE sysevent;
    }
}
```

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

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 LINK_REMAP door call to devfsadmd 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 */
       door_call_to_devfsadmd(LINK_LOOKUP, linkmap);
       /* Then change linkmap->devname to link1 */
       door_call_to_devfsadmd(LINK_REMAP, linkmap);
}
```

– 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).

The dls module will check whether any DLS clients have link1 open, or whether there are any aggregations or VLANs created over this link. Since aggregations and VLANs are all MAC clients, when a DLS client opens a link, the corresponding MAC will in turn be held. Therefore, the mac_is_held() function will be used to perform this check. If mac_is_held() returns success, the dls module will update the link name of the corresponding kernel data-link from link1 to link2:

```
/* link1 is an available link and link2 is a REMOVED physical LINK */
int dls_rename_link_3(const char *link1, const char *link2)
{
       data_link = dls_vlan_lookup(link1);
       /*
        * Check whether the corresponding MAC is in use by any
        * aggregations or VLANs
        */
       if (mac_is_held(data_link->dv_dlp->dl_mac)
              return FAILURE;
       dls_vlan_update_name(data_link, link2);
}
```

The `libdladm` library will then issue a `LINK_FREE` door call followed by a `LINK_REMAP` door call to `devfsadmd` to disassociate `link1` from its linkid, and reassociate `link1`'s device instance name with `link2`. As the result, the link mapping in the persistent link configuration file will also be updated. Finally, `libdladm` will generate a `RCM_RESOURCE_NETWORK_UPDATE` sysevent to inform the `network_rcm` module of the updated link name, causing `network_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_LINK ioctl to the dld driver;
    if (dld acknowledges success) {
       /* Set linkmap1->linkname to link1 */
       door_call_to_devfsadmd(LINK_LOOKUP, linkmap1);
       /* Set linkmap2->linkname to link2 */
       door_call_to_devfsadmd(LINK_LOOKUP, linkmap2);
       /* Free linkmap1->linkid */
       door_call_to_devfsadmd(LINK_FREE, linkmap1);
       /* Set linkmap2->devname to linkmap1->devname */
       door_call_to_devfsadmd(LINK_REMAP, linkmap2);
       send RCM_RESOURCE_NETWORK_UPDATE sysevent;
}
```