

IP Tunneling Device Driver Design Specification

Sebastien Roy

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

Network Approachability
Sun Microsystems, Inc.

Revision 1.2
December 5, 2005

Contents

1	Introduction	1
2	Requirements	1
3	Architectural Overview	2
4	Configuration Using <code>dladm(1M)</code>	3
4.1	Naming of IP Tunnel Links with <code>dladm(1M)</code>	3
4.2	New <code>dladm(1M)</code> subcommands	3
4.2.1	<code>create-iptun</code>	3
4.2.2	<code>modify-iptun</code>	5
4.2.3	<code>delete-iptun</code>	5
4.2.4	<code>show-iptun</code>	5
4.2.5	<code>up-iptun</code>	6
4.2.6	<code>down-iptun</code>	6
4.3	Impact on Existing <code>dladm(1M)</code> Subcommands	7
4.3.1	<code>show-link</code>	7
4.3.2	<code>show-dev</code>	7
5	Configuration Using <code>ifconfig(1M)</code>	7
6	The <code>libiptun.so</code> Configuration Library	9
6.1	Tunnel Parameters	9
6.2	Library Functions	10
6.2.1	Creating an IP Tunnel	10
6.2.2	Deleting an IP Tunnel	11
6.2.3	Modifying Tunnel Parameters	11
6.2.4	Getting a Tunnel's Parameters	11
6.2.5	Walking the Tunnel List	12
6.2.6	Re-Creating Persistently Defined Tunnels	12
6.2.7	Deleting all Tunnels	12
6.3	Configuration File	12
7	Tunnel Driver	13
7.1	Tunnel Control Device	13
7.2	Interaction With IP	14
7.3	Interaction With IPsec	16
7.4	<code>kstat</code> Implications	16
8	General DLPI Implications	17
8.1	<code>DL_INFO_ACK</code>	17
8.2	<code>DL_BIND_REQ</code>	18
8.3	<code>DL_ENABMULTI_REQ</code> and <code>DL_DISABMULTI_REQ</code>	18
8.4	<code>DL_PROMISCON_REQ</code> and <code>DL_PROMISCOFF_REQ</code>	18
8.5	<code>DLIOCRAW</code>	18
8.6	<code>DLIOCHDRINFO</code> or <code>DL_IOC_HDR_INFO</code>	18
8.7	<code>DL_NOTE_LINK_UP</code> and <code>DL_NOTE_LINK_DOWN</code>	19
9	Changes to the Nemo Framework	19

9.1	Nemo MAC Type Plugins	19
9.1.1	DL_IPV4 Plugin	19
9.1.2	DL_IPV6 Plugin	21
9.1.3	DL_6T04 Plugin	23
9.2	DL_NOTE_SDU_SIZE	24
9.3	Tunnel Source and Destination via DLPI	24
9.4	Summary of Changes to Nemo Interfaces	26
9.4.1	MAC driver changes	26
9.4.2	MAC client changes	26
9.4.3	DLD client changes	26
10	IP Tunneling SMF Service	27
11	Dependency on Network Vanity Naming	27
12	Impact on libdlpi.so	28
13	Impact on <sys/dlpi.h>	28
14	Impact on STREAMS Autopush	28
15	Impact on Mobile IP	29
16	EOL of Automatic Tunnels	29
A	Configuration Example	31
B	<libiptun.h>	34
C	<sys/iptun.h>	35
D	<sys/iptun_impl.h>	35

1 Introduction

Observability of network interfaces is one of the focal points of the Clearview project. Providing a method of accessing packets flowing through a network interface is one of the main network interface requirements described in the Clearview Overview¹. This has been a missing feature of IP tunnel interfaces since their introduction in Solaris 8.

This is especially problematic for IP tunnels that use encryption (IPsec tunnels), as the packets flowing through those tunnel interfaces are not observable at any level.

The root of the problem is that IP tunnel interfaces do not currently provide DLPI device nodes that traditional network observability tools such as `snoop(1M)`, `ethereal`, and `tcpdump` use to open and observe packets from. The current IP tunnel implementation also does not provide the full DLPI state machine required to support such observability. These issues are what this part of the Clearview project will address.

2 Requirements

The technical requirements for the IP tunnel device driver are:

1. Must be backward compatible.

Solaris currently implements four kinds of IP tunneling, all of which are configured using `ifconfig(1M)`. This design must maintain all of the functionality and administrative interfaces of the current implementation. Customers have custom scripts that configure IP tunnel interfaces on demand. To allow these scripts to continue to function unaltered, we must maintain the `ifconfig` syntax for configuring IP tunnels.

2. Must export a device node under `/dev/net`² that implements a full DLPI state machine for each IP tunnel interface.

This is required in order for tunnel interfaces to be accessible in the same manner as other network interfaces. Today, tunnel interfaces are not represented in `/dev` and no DLPI applications have access to these interfaces. Implementing this requirement will enable DLPI access to tunnel interfaces. This will allow administrators to use `snoop` to observe packets flowing over these interfaces.

3. Must be able to observe clear-text packets flowing through an IPsec tunnel configured to use ESP.

This is crucial for debugging networking related problems when using an encrypted IP tunnel. When running `snoop` on a tunnel interface, the messages passed up by the device node must be in the clear.

4. Must be able to administer the link-layer attributes of tunnel interfaces using the `dladm(1M)` command.

The link-layer attributes of an IP tunnel include its tunnel source and destination IP addresses, as well as its link name. This will allow IP tunnel interfaces to be named using the Vanity Naming feature introduced by Clearview.

¹<http://clearview.east/docs/overview.txt>

²Under the Clearview project, networking DLPI devices are found under `/dev/net`. Because the Vanity Naming functionality provided by Clearview allows for nearly arbitrary names for networking devices, isolating the devices in such a manner prevents namespace clashes with other devices on the system. For details on the Clearview Vanity Naming functionality, see http://clearview.east/wiki/index.php/Clearview:Vanity_Naming

5. Must not degrade performance of IP tunnel interfaces.

Throughput and latency of data flowing through tunnel interfaces must not be negatively impacted by this project, regardless of the number of tunnel interfaces configured.

3 Architectural Overview

The IP tunnel device driver will be a Nemo (GLDv3) network driver that exports a DLPI device node under `/dev/net` for each tunnel. This will allow observability tools like `snoop` to open tunnel devices and observe traffic.

The `dladm(1M)` command will be used to create and configure tunnel devices that IP and other DLPI consumers can then open and use as with any other DLPI link-layer provider. It will do this by calling into a tunnel configuration API implemented in the `libiptun.so` library. This library will be responsible for interacting with the tunnel control device using a set of `ioctl`s. This is analogous to the configuration of aggregations, where `dladm` calls into the `liblaadm.so` provided API, which in turn interacts with the aggregation control device.

IP tunnel interfaces will be plumbed using `ifconfig`. One major difference between this design and how `ifconfig` currently plumbs IP tunnel interfaces is that `ifconfig` will open an IP tunnel DLPI device just as it does currently with other types of network devices. Currently, to configure an IP tunnel interface, `ifconfig` opens `/dev/ip` and pushes the `tun` module on the stream, which then allows `ifconfig` to issue DLPI primitives to configure the rest of the IP interface (the `tun STREAMS` module implements enough DLPI primitives to allow `ifconfig` and `ip` to configure an IP interface on such a stream). The `ifconfig` command knows to do this dance based on the syntax of the tunnel interface name, which is `ip.tun#`. This will all be drastically simplified by the fact that this design will provide a DLPI device node that `ifconfig` can simply open as it does with all other network interfaces.

The `tun STREAMS` module along with its friends `atun` and `6to4tun` will be removed, as their functionality will be replaced by the DLPI driver mentioned above. The `tun(7M)` man page documents these modules as Evolving, but they are really an implementation detail of how IP tunnel interfaces are configured by `ifconfig`. Nothing other than `ifconfig` is believed to currently use these modules.

To allow the IP tunnel driver to hook into the Nemo framework, the framework will be altered to support the `DL_IPV4`, `DL_IPV6`, and `DL_6T04` MAC types that the tunnel driver will implement. This will be done by modifying Nemo to MAC-type independent.

4 Configuration Using `dladm(1M)`

The `dladm` command will be used to create and administer IP tunnel links. Administrators will be allowed to create IP tunnel links, configure and modify all link-layer properties of IP tunnel links, and view IP tunnel configuration information.

4.1 Naming of IP Tunnel Links with `dladm(1M)`

The `dladm` command will refer to IP tunnels by data-link name³. When creating an IP tunnel, the administrator will pick a data-link name for that tunnel (or one will be chosen automatically), and that name will also be used to modify or delete the tunnel. The name itself may also be changed using `dladm`'s `modify-link` subcommand provided by the Vanity Naming⁴ feature of Clearview.

The choice to use data-link names to refer to IP tunnels was made to simplify the administration of these objects. Requiring the administrator to deal with two namespaces (one for the IP tunnels, and another for the data-links associated with those tunnels) would be confusing. Further, this decision allows IP tunnel interfaces to meet one of the core Clearview network interface requirements⁵, which is to have the same name at any layer it is named at.

4.2 New `dladm(1M)` subcommands

New subcommands will allow an administrator to configure every aspect of a tunnel link in a temporary or persistent manner. The `-t` option (which stands for “temporary”) will allow administrators to operate on the running system, but not on the persistent system configuration. If `-t` is not used, every command will result in a persistent configuration that will be applied automatically when the system reboots.

All new IP tunnel operations will be implemented by calling into the `libiptun.so` library described in section 6.

Currently, `dladm` requires both `PRIV_SYS_NET_CONFIG` and `PRIV_NET_RAWACCESS` privileges for all of its operations. A separate fix is required to make `dladm` require specific privileges only for those operations that need them. For example, the `show-iptun` subcommand described below should not require those privileges, but the current `dladm` design does require them. There is an existing bug documenting this problem:

```
6321504 dladm(1M) show-* subcommands need to work for ordinary users
```

The following subcommands will be introduced for IP tunnel link management:

4.2.1 `create-iptun`

```
dladm create-iptun [-t] [-T type] -s tsrc [-d tdst] [-h hoplimit]  
[-e encaplimit] [name]
```

³The Vanity Naming documentation section of the Clearview web site contains a treatise on the namespaces involved within the Nemo framework. It describes how these names are generated and their relationships: http://clearview.east/wiki/index.php/Clearview:Vanity_Naming:Naming_Treatise

⁴http://clearview.east/wiki/index.php/Clearview:Vanity_Naming

⁵The core Clearview network interface requirements are contained in the Clearview overview document located here: <http://clearview.east/docs/overview.txt>

The `create-iptun` subcommand will create an IP tunnel and associated DLPI data link node in `/dev/net`. It will call into `libiptun.so`'s `iptun_create()` function.

The *type* argument can be any one of the following⁶:

- `ipv4`

A configured IPv4 tunnel is a point-to-point link between two IPv4 nodes. The term “configured” is used to refer to the fact that the tunnel destination is a pre-configured static value, as opposed to other types of tunneling for which the tunnel destination is determined automatically when encapsulating packets. Note that both IPv4 and IPv6 interfaces can be configured over an IPv4 configured tunnel. This type of tunnel requires the configuration of IPv4 tunnel source and destination addresses.

- `ipv6`

A point-to-point link between two IPv6 nodes. Like an IPv4 configured tunnel, both IPv4 and IPv6 interfaces may be configured over an IPv6 configured tunnel. This type of tunnel requires the configuration of IPv6 tunnel source and destination addresses.

- `6to4`

This tunnel type is described in [RFC 3056]. It provides an automatic tunneling mechanism for IPv6 over IPv4. It requires the configuration of an IPv4 tunnel source address.

The *type* argument can be omitted when creating a configured tunnel where both tunnel source and destination addresses are specified. For example, if both are literal IPv4 addresses, then the tunnel type will be assumed to be `ipv4`. Likewise, if both are literal IPv6 addresses, then the type will be assumed to be `ipv6`.

The command will accept literal IP addresses or host names for the *tsrc* and *tdst* arguments. Note that if using host names, those names will be resolved to IP addresses, and one of those IP addresses will be used to configure the tunnel on the running system. The address picked will be the first in the list returned by `getaddrinfo(3SOCKET)`. The host names will be stored verbatim into the configuration storage to be applied at subsequent boots (if not using `-t` to make the configuration temporary). As a result, if those names resolve to different IP addresses at subsequent boots, the tunnel configuration will change. Moreover, if the names fail to resolve or resolve to an address family that is not applicable to the tunnel type, the tunnel will fail to be created altogether.

The creation will also fail if the tunnel's addresses conflict with an existing tunnel. Such a failure will occur in the following cases:

1. If the tunnel being created is a configured tunnel and another configured tunnel with the same tunnel source and tunnel destination exists.
2. If the tunnel being created is a configured tunnel and a 6to4 tunnel with the same tunnel source exists.
3. If the tunnel being created is a 6to4 tunnel and a tunnel with the same tunnel source exists.

These error semantics will be implemented in order to fix a de-multiplexing ambiguity that exists when such tunnels are created. This problem is documented in the following bug, which will be fixed by this design:

⁶Note that these are the IP tunnel types currently implemented in Solaris. No new types are being introduced here, nor are any types being EOL'ed as part of this design.

4152864 Configuring two tunnels with the same tsrc/tdst pair works.

The *name* argument will be optional, and will be the name of the tunnel. If *name* is not specified by the caller, it will be automatically chosen based on the type of the tunnel as described in section 6.2.1, and will be printed to `stdout`.

This name will:

1. Be the user's handle to the tunnel for other `dladm` tunnel operations such as `modify-iptun` and `delete-iptun` described below.
2. Be the name used by the `libiptun.so` API described in section 6 and the `ioctl` control interface provided by the control device described in section 7.1.
3. Be the name of the data-link node created in `/dev/net`.

As such, throughout this document, the term “tunnel name” refers to this name.

Examples:

```
# dladm create-iptun -s 63.1.2.3 -d 192.4.5.6 punchin0
# dladm create-iptun -T 6to4 -s 63.4.5.6 ipv6gateway12
# dladm create-iptun -s 2001:db8:feed::1234 -d 2001:db8:beef::4321
ip6.tun0
#
```

4.2.2 modify-iptun

```
dladm modify-iptun [-t] [-s tsrc] [-d tdst] [-h hoplimit] [-e encaplimit] name
```

The `modify-iptun` subcommand will call into `libiptun.so`'s `iptun_modify()` function. This operation will fail if the tunnel source or destination address are inappropriate for the type of the tunnel, or if *hoplimit* or *encaplimit* are beyond acceptable ranges. The operation is applicable even after IP interfaces have been configured on the tunnel link, and any changes will take effect immediately.

4.2.3 delete-iptun

```
dladm delete-iptun [-t] name
```

The `delete-iptun` subcommand will call into `libiptun.so`'s `iptun_delete()` function. This operation will fail if the tunnel's DLPI device is open (by an IP interface or any other DLPI consumer).

4.2.4 show-iptun

```
dladm show-iptun [-p] [name]
```

The `show-iptun` subcommand will print tunnel configuration. If *name* is given, it will call `libiptun.so`'s `iptun_get_params()` function for the specified tunnel and print its configuration. If the *name* argument is omitted, it will call `libiptun.so`'s `iptun_walk_sys()` function using a function that prints each tunnel's configuration information as a callback.

Example:

```
# dladm show-iptun
tunnel name  type      source      destination  hoplimit  encapslimit
punchin0     ipv4      63.1.2.3    192.4.5.6   60        N/A
ipv6gateway12 6to4     63.4.5.6    N/A         60        N/A
ip6.tun0     ipv6      2001:db8:feed::1234 2001:db8:beef::4321 60        0
```

Note that the name of the tunnel printed is the name used when creating the tunnel with `create-iptun`.

The `-p` option causes `show-iptun` to print machine-parseable output. Example:

```
# dladm show-iptun -p
tunnel=punchin0
type=ipv4
source=63.1.2.3
destination=192.4.5.6
hoplimit=60

tunnel=ipv6gateway12
type=6to4
source=63.4.5.6
hoplimit=60

tunnel=ip6.tun0
type=ipv6
source=2001:db8:feed::1234
destination=2001:db8:beef::4321
hoplimit=60
encapslimit=0
```

4.2.5 up-iptun

```
dladm up-iptun
```

This project-private subcommand will be used by the `svc:/network/iptun` SMF service's start method to configure all tunnels that were previously created. It will call `libiptun.so`'s `iptun_up()` function. Its main purpose will be to re-create all persistent tunnels at boot-time.

4.2.6 down-iptun

```
dladm down-iptun
```

This project-private subcommand will be used by the `svc:/network/iptun` SMF service's stop method to temporarily delete all existing tunnels. It will call `libiptun.so`'s `iptun_down()` func-

tion. It is being provided for symmetry with the `up-iptun` command, so that tunneling as a service can be disabled either at system shutdown time or manually using `svcadm(1M)`.

4.3 Impact on Existing `dladm(1M)` Subcommands

4.3.1 `show-link`

The `show-link` subcommand will display tunnel links in addition to the links it displays today.

Example:

```
# dladm show-link
bge0          type: non-vlan  mtu: 1500    device: bge0
bge2000       type: vlan 2    mtu: 1500    device: bge0
aggr2         type: non-vlan  mtu: 1500    aggregation: key 2
punchin0      type: IP tunnel mtu: 1480
ipv6gateway12 type: IP tunnel mtu: 65515
ip6.tun0      type: IP tunnel mtu: 1452
```

4.3.2 `show-dev`

The `show-dev` subcommand displays physical network devices, which IP tunnels are not. It will thus not display tunnel devices, just as it does not show aggregation devices. A separate `show-iptun` (previously described in section 4.2.4) subcommand will be used to show tunnels.

5 Configuration Using `ifconfig(1M)`

Today, `ifconfig` is used to create tunnel links, set their link layer properties such as tunnel source and tunnel destination, and configure IP interfaces on those tunnels. As described in Section 3, these tasks will become split between `dladm` and `ifconfig`, where link-layer operations such as the creation of tunnel links and the setting of link-layer properties will be done through `dladm`, and the configuration of IP interfaces on tunnels will be done through `ifconfig`.

In order to maintain backward compatibility, the existing `ifconfig` syntax for creating and modifying tunnel link-layer properties will still function, but will call in to the `libiptun.so` library. This needs to be done so that existing configuration files such as `/etc/hostname*.ip*tun*` can be left unmodified, and customer scripts that call `ifconfig` directly can continue to work unmodified. All `ifconfig` operations done through `libiptun.so` will be temporary (by setting only the `IPTUN_APPLY_NOW` `applyflags` flag to the `libiptun.so` functions) to maintain `ifconfig`'s current behavior of not altering persistent system state.

Here is an example showing how a tunnel is created today using `ifconfig`:

```
# ifconfig ip.tun0 plumb
# ifconfig ip.tun0 tsrc 11.0.0.1 tdst 12.0.0.1
# ifconfig ip.tun0 10.0.0.1 10.0.0.2 up
```

Note that this sequence of commands is invoked automatically during boot by the `/lib/svc/method/net-init` boot method if a file named `/etc/hostname.ip.tun0` exists and contains the following lines:

```
tsrc 11.0.0.1 tdst 12.0.0.1
10.0.0.1 10.0.0.2 up
```

This design proposes to maintain backward compatibility with `ifconfig` by maintaining the above syntax. The difference lies in what `ifconfig` will do internally. Below is a break-down of what `ifconfig` will do for each command mentioned above:

1. `ifconfig ip.tun0 plumb`

The `ifconfig` command will attempt to plumb `ip` on a DLPI device named `ip.tun0`. This will be no different than what it does for every other kind of device today. The `libdlpi.so` `dlpi_if_open()` function will no longer parse the interface name to dissect STREAMS modules from the device name in order to push those modules. It will simply open the named device. This change is discussed further in section 12.

If the named device (`ip.tun0` in this case) does not exist, `ifconfig` will silently call `iptun_create()`, then plumb `ip` on the resulting device as mentioned above. The tunnel type will be determined based on the name of the tunnel interface. This will allow administrators to upgrade Solaris without having to re-create their tunnels using `dladm`. The tunnels will simply be created automatically.

The following types will be used for the given interface names:

Tunnel Name	Resulting Tunnel Type
<code>ip.tun#</code>	<code>IPTUN_TYPE_IPV4</code>
<code>ip6.tun#</code>	<code>IPTUN_TYPE_IPV6</code>
<code>ip.6to4tun#</code>	<code>IPTUN_TYPE_6T04</code>

2. `ifconfig ip.tun0 tsrc 11.0.0.1 tdst 12.0.0.1`

Today, `ifconfig` uses the `icfg_set_tunnel_[src|dst]()` functions from `libinetcfg.so` to set these addresses. Those functions open a `SOCK_DGRAM` socket and send down `SIOCSTUNPARAM` ioctls to set those tunnel properties. The ioctls are passed through `ip` to the appropriate `tun` STREAMS module, which configures the tunnel source and destination addresses.

The `libinetcfg.so` functions will no longer be needed, as the `libiptun.so` API will replace their functionality. As such, they will be removed from `libinetcfg.so`. This should be of no consequence as these interfaces were not documented. The `SIOCSTUNPARAM` and `SIOCGTUNPARAM` ioctls used by `libinetcfg.so` will also be removed for the same reason⁷.

In this design, `ifconfig` will call `iptun_modify()` to set the tunnel addresses instead.

3. `ifconfig ip.tun0 10.0.0.1 10.0.0.2 up`

This step will remain unchanged. `ifconfig` will continue using the `SIOCSLIF*` ioctls to configure the IP interface that was previously plumbed on the tunnel device.

This design will gracefully handle an upgrade without needing any fancy scripts to convert the `/etc/hostname*.tun*` contents to `dladm` commands. It will be up to the administrator to remove the unneeded contents of the `/etc/hostname*.tun*` files (`tsrc` and `tdst`), but their presence will not cause problems. It will be safe for the administrator to remove such contents from `/etc/hostname*.tun*` files when `dladm` has been used to create the tunnel link on a persistent basis.

A potential conflict could arise if an administrator has created a tunnel using `dladm` and has conflicting `tsrc` or `tdst` information in the `/etc/hostname*.*` file associated with that tunnel. Because the `/etc/hostname*.*` files will be processed after tunnels are created by `dladm up-iptun`, the information in the `/etc/hostname*.*` file will override what has been configured by `dladm`.

⁷The `SIOC*TUNPARAM` ioctls were unfortunately documented in the `tun(7M)` man page, but `ifconfig` is the only consumer.

6 The libiptun.so Configuration Library

This library will provide a project-private API for creating and administering IP tunnel links. It will be used by `dladm` and `ifconfig` to create tunnels and set their parameters. It will act as the interface between administrative utilities and the `ioctl`s exported by the tunnel control device described in section 7.1.

The operations provided by the API will apply to the running system or to a persistent configuration to be applied at next boot, or both. This will be controlled with the `applyflags` argument, which can have two possible flags set:

- `IPTUN_APPLY_NOW`
- `IPTUN_APPLY_LATER`

If neither flag is set, an `assert()` failure will be triggered.

The persistence of configuration information will be maintained by placing such information into storage that is private to the library (a file named `/etc/iptun.conf` described in section 6.3). When the system boots, `dladm` will call into the library to create all tunnels described in the persistent configuration store.

The API will also support configuring an alternate root environment. Functions will accept an `altroot` argument that, if set, will apply changes to the configuration file in the specified alternate root. The only acceptable value for `applyflags` when `altroot` is used is `IPTUN_APPLY_LATER`. Any other value will trigger an `assert()` failure.

6.1 Tunnel Parameters

Most of the functions provided by the `libiptun.so` API take a set of tunnel parameters as input. The library will define the following structure to hold these parameters:

```
typedef struct iptun_params {
    iptun_fields_t  itp_fields;
    char            itp_name[LIFNAMSIZ];
    iptun_type_t    itp_type;
    char            *itp_src;
    char            *itp_dst;
    uint8_t         itp_hoplimit;
    uint8_t         itp_encaplimit;
    ipsec_req_t     itp_secinfo;
} iptun_params_t;
```

- `itp_fields`

This will be a set of flags that will be set by the caller to specify which parameters apply to functions that take an `iptun_params_t`. The possible flag values are defined in appendix B.

- `itp_name`

This will be the data-link name of the IP tunnel. This is the same name that users of both `dladm` and `ifconfig` (the two consumers of this API) will pass to those commands.

- **itp_type**

The tunnel type will be set at tunnel creation time, and will not be modifiable once set.

The following types will be defined⁸:

- IPTUN_TYPE_IPV4
- IPTUN_TYPE_IPV6
- IPTUN_TYPE_6T04

- **itp_src**

The tunnel source address is used as the source address in the outer IP headers when packets are encapsulated by the tunnel. Every tunnel type must be assigned a tunnel source address before the interface can send or receive packets. This string can either be a literal IP address or a host name.

- **itp_dst**

Like the tunnel source address, the tunnel destination address is used as the destination address in the outer IP headers when packets are encapsulated by the tunnel. Only configured tunnels are assigned a tunnel destination address. This string can either be a literal IP address or a host name.

- **itp_hoplimit**

The tunnel hop limit is used as the TTL in outer IPv4 headers, and as the hop limit in outer IPv6 headers. Setting the hop limit will be optional, and the system will use a suitable default if left unset. The system will also use a default value if the specified hoplimit is 0.

- **itp_encaplimit**

This optional setting is only applicable to configured IPv6 tunnels. The limit controls how many levels of nested IPv6 tunneling is allowed for a given packet, and is implemented by placing a Tunnel Encapsulation Limit Destination Option in each encapsulating packet. This mechanism is described in section 4.1.1 of [RFC 2473]. The default value of 0 will not impose an encapsulation limit on tunneled packets.

- **itp_secinfo**

This optional setting will allow the caller to specify IPsec policy for the tunnel. The policy will be specified using an `ipsec_req_t`, which is the same mechanism used for configuring per-socket policy. See `ipsec(7P)` for details.

6.2 Library Functions

6.2.1 Creating an IP Tunnel

```
iptun_err_t iptun_create(tun_params_t *params, uint_t applyflags,  
                        const char *altroot);
```

The `iptun_create()` function will create a new IP tunnel given a set of tunnel parameters specified in the `params` argument. The `itp_type` field of the `params` argument is the only required `params` field.

The caller will have the option of specifying the name of the data-link for the IP tunnel by setting the `itp_name` field in the `params` argument. If `itp_name` is the empty string (one whose first

⁸See section 4.2.1 for a description of these tunnel types.

character is '\0'), the data-link name will be automatically chosen based on the tunnel type according to the following mapping:

<u>Tunnel Type</u>	<u>Tunnel Name</u>
<code>IPTUN_TYPE_IPV4</code>	<code>ip.tun#</code>
<code>IPTUN_TYPE_IPV6</code>	<code>ip6.tun#</code>
<code>IPTUN_TYPE_6TO4</code>	<code>ip.6to4tun#</code>

The PPA number (denoted by `#` above) will be chosen as the lowest available number for that tunnel type. The chosen name will be in the `itp_name` field upon function return.

A DLPI device node named `/dev/net/itp_name` will also be created.

The `itp_type` and `itp_name` fields will be the only parameters that will not be modifiable after tunnel creation time. The caller will have the option of setting all other parameters either at tunnel creation time, or at a later time using `iptun_modify()`.

Note that a tunnel will not transfer data until the tunnel source address has been set. For configured tunnels, the tunnel will not transfer data until both the tunnel source and tunnel destinations have been set.

6.2.2 Deleting an IP Tunnel

```
iptun_err_t iptun_delete(const char *name, uint_t applyflags,  
                        const char *altroot);
```

A tunnel will be destroyed by calling `iptun_delete()`. The tunnel of the name specified by `name` will be destroyed, as well as its associated `/dev/net` DLPI node.

This operation will fail if the named tunnel does not exist, or if the tunnel's DLPI node has an open reference (a plumbed IP interface for example).

6.2.3 Modifying Tunnel Parameters

```
iptun_err_t iptun_modify(const iptun_params_t *params, uint_t applyflags,  
                        const char *altroot);
```

This function will be used to modify tunnel properties of the tunnel named in the `itp_name` field of the `params` structure. The list of parameters is described in section 6.1. The only parameters that will not be modifiable are `itp_name` and `itp_type`.

Note that modifying the name of the data-link of the tunnel is outside the scope of this API. This will be handled by the `dladm modify-link` subcommand, the inner-workings of which are handled by the Vanity Naming feature of Clearview.

6.2.4 Getting a Tunnel's Parameters

```
iptun_err_t iptun_get_params(iptun_params_t *params, uint_t applyflags);
```

The parameters of the tunnel named by the `itp_name` field of the `params` argument will be returned in the `params` structure. The `applyflags` argument will determine if the parameters

returned are those of the running system (IPTUN_APPLY_NOW), or of the persistent configuration (IPTUN_APPLY_LATER).

6.2.5 Walking the Tunnel List

```
iptun_err_t iptun_walk_sys(uint_t applyflags, iptun_walk_func_t *fnp,  
    void *arg);
```

For each created tunnel, this function will call the function `fnp` and pass it two arguments: the `iptun_params_t` parameters associated with the tunnel, and the optional opaque `arg` argument passed to `iptun_walk_sys`.

The `applyflags` argument will determine if the list of tunnels walked is the list of tunnels configured on the running system (IPTUN_APPLY_NOW), or the list of tunnels configured persistently in the configuration file (IPTUN_APPLY_LATER).

6.2.6 Re-Creating Persistently Defined Tunnels

```
iptun_err_t iptun_up(const char *tunname);
```

This function will re-create a tunnel that was persistently defined. It will re-create all persistently defined tunnels if the `tunname` argument is NULL. This will be used when booting the system. The `dladm` command will provide a project-private subcommand called `up-iptun` that, when used by a boot script, will call this function resulting in previously created tunnels being configured.

6.2.7 Deleting all Tunnels

```
iptun_err_t iptun_down(void);
```

This function will temporarily delete all tunnels by calling `iptun_delete()` with the IPTUN_APPLY_NOW flag on every tunnel using the `iptun_walk_sys()` routine. It is a convenience function for the `down-iptun` `dladm` subcommand.

6.3 Configuration File

The `libiptun.so` library will keep persistent configuration information in a project-private configuration file named `/etc/datalink.conf`. The format of the file is to be determined, but it will contain all persistent information for all data-links on the system, including IP tunnels. The Vanity Naming component of Clearview will define the format of this file.

The information to be kept in the file for an IP tunnel interface will be:

- Link name
- Link id
- Tunnel type

- Tunnel source
This will be the string contained in the `iptun_params_t` structure used to configure the tunnel. If a hostname was used to configure the tunnel, that hostname will be stored here. If it was a literal IP address, that literal address will be stored here.
- Tunnel Destination (if applicable)
Similar to the tunnel source, this will be the string contained in the `iptun_params_t` structure passed in when the tunnel was configured.
- Tunnel hop-limit (if applicable)
- Tunnel encapsulation limit (if applicable)

7 Tunnel Driver

7.1 Tunnel Control Device

The tunnel control device will be used as the kernel interface for managing IP tunnels. All interfaces provided by the device will be project-private. Its only consumer will be the `libiptun.so` library. The device will be accessible by opening the file `/dev/iptunctl`.

The `/dev/iptunctl` device will have read/write permissions for everyone, but will require specific privileges for particular ioctls.

The following ioctls will be exported:

- `IPTUN_CREATE`
This ioctl will take an `iptun_create_t` structure as input. The `iptun_create_t` structure is defined in appendix C. It will create a tunnel MAC and register it with the `mac` module using `mac_register()`. The MAC name used for the tunnel will be `iptun<link-id>`. The link-id is simply a number known to be universally unique amongst all links and will thus be known to be unique amongst all tunnels.
If the `mac_register()` operation is successful, a data-link will be created for the tunnel using `dls_vlan_create()` using the link-name and link-id passed in to the ioctl.
This ioctl will fail if a tunnel with the given link-name or link-id already exists, if a tunnel with the given tunnel source and destination already exists, or if tunnel parameters malformed or invalid.
This ioctl will require the `PRIV_SYS_NET_CONFIG` privilege.
- `IPTUN_DELETE`
This ioctl will take the link-id of the tunnel to be deleted. The driver will delete the tunnel and unregister it with the `mac` module using `mac_unregister()`. The data-link associated with the tunnel will also be deleted. The driver will lookup the link-name of the tunnel by using a function provided by the `dls` module that will implement link-id to link-name lookups.
This ioctl will fail if there is no tunnel with the given id, or if the named tunnel's DLPI node has an open reference (a plumbed IP interface for example).
This ioctl will require the `PRIV_SYS_NET_CONFIG` privilege.

- **IP_TUN_MODIFY**

This ioctl will take an `iptun_kernparams_t` structure defined in appendix C. It will modify the tunnel whose link-id is `itk_linkid` using the properties specified by `itk_fields`.

This ioctl will require the `PRIV_SYS_NET_CONFIG` privilege.

- **IP_TUN_INFO**

This will return the properties of one or all tunnels. This ioctl will not require any privileges.

- **IP_TUN_SET_6TO4RELAY**

This will replace the existing undocumented `SIOCS6TO4TUNRRADDR` socket ioctl. It will take an `ipaddr_t` as its sole argument, and will set the 6to4 relay router address. The only consumer of this ioctl will be the `6to4relay(1M)` command. The `SIOCS6TO4TUNRRADDR` ioctl will be removed.

This ioctl will require the `PRIV_SYS_NET_CONFIG` privilege.

- **IP_TUN_GET_6TO4RELAY**

This will replace the existing undocumented `SIOCG6TO4TUNRRADDR` socket ioctl. It will take an `ipaddr_t` as its sole argument, and will return the current address of the 6to4 relay router in that argument. The only consumer of this ioctl will be the `6to4relay(1M)` command. The `SIOCG6TO4TUNRRADDR` ioctl will be removed.

This ioctl will not require any privileges.

7.2 Interaction With IP

IP factors into an IP tunnel interface in two places. One is above the tunnel module, and one is below. Figure 1 illustrates this model. In this design, the tunnel driver will not need to directly interact with IP above it, as this will be handled by the Nemo framework through the use of DLPI and a polling interface provided by the Nemo `d1d` and `d1s` modules⁹.

The tunnel driver will communicate with IP below it using direct function calls into the `ip` module. Each tunnel link will be represented as a separate connection (`conn_t`) in `ip`. When a tunnel is created, a new `conn_t` will also be created using `ipcl_conn_create()`. When the tunnel source is set on a given tunnel, the tunnel driver will call `ip_bind[_v6]()` directly using the `conn_t` associated with the tunnel. When sending packets, the tunnel driver will call `ip_output[_v6]()` directly.

This differs from the way that the current `tun` STREAMS module interacts with `ip`. Currently, it issues TPI STREAMS messages just like the `icmp` module does for raw sockets. It does this using the streams that were created for each tunnel by `ifconfig`¹⁰. Binding to tunnel source and destination addresses involves a `T_BIND_REQ` message, and sending a packet involves an `M_DATA` STREAMS message. As mentioned, all of this STREAMS interaction will be replaced by direct function calls.

As a result of this change, the tunnel driver will no longer need to use `IRE_DB_REQ_TYPE` STREAMS messages to obtain copies of `IRE_CACHE` entries for tunnel destinations when calculating tunnels' path MTUs. It will call `ire_route_lookup()` directly, and will obtain any IPsec overhead by calling `conn_ipsec_length()` directly. Note that the `IRE_DB_REQ_TYPE` message will not be removed entirely from Solaris, as it is still used by other transport layer STREAMS modules. The IP tunnel driver simply will not use this mechanism.

⁹For more detail on the functionality provided by Nemo, see the Nemo design documentation: <http://clearview.east/docs/phase0/nemodesign-v5.sxw>

¹⁰Remember, tunnel interfaces are currently created by pushing the `tun` module onto a `/dev/ip` stream.

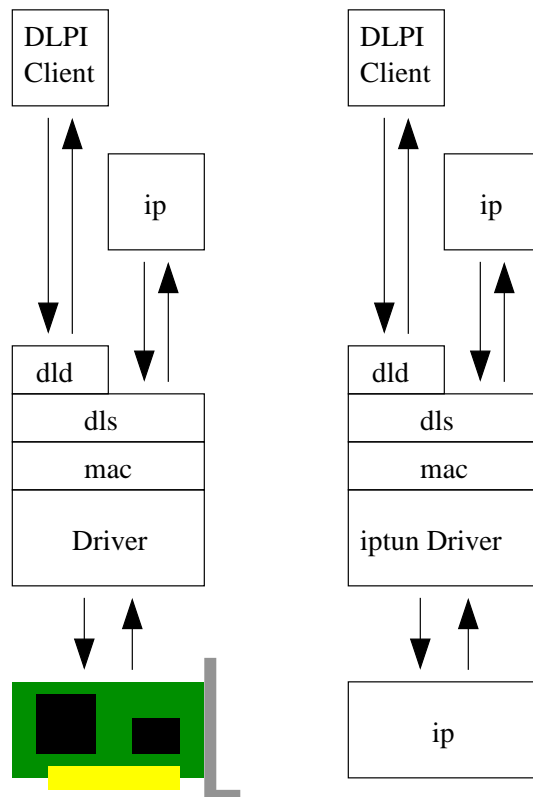


Figure 1: At left, the model of a typical Nemo Ethernet driver. The driver registers with the `mac` module above and interacts with the media hardware below. At right, how the tunnel driver uses this same model. The tunnel driver also registers with the `mac` module above, but instead of interacting with media hardware, it interacts with the `ip` module below it using function calls.

The receive path will require `ip` to fanout packets to various tunnel interfaces. A new tunnel-specific fanout will be implemented in `ip` for this purpose. This fanout will associate tunneled packets to a set of tunnel `conn_t`'s using the source address, destination address, and IP version of the outer IP header, and then call the input function in the tunnel driver. If a tunneled packet does not match any of the fanout entries, it will fall-back to the general protocol fanout so that raw sockets can receive it.

Note that tunneled packets are currently fanned out using `ip`'s generic protocol fanout in the `ip_fanout_proto[_v6]()` function. This fanout is a hash table that is keyed by IP protocol number. It was designed for raw sockets that are bound to a particular protocol, and not necessarily to IP addresses. This current fanout is inadequate for two reasons:

- When a system has a large number of tunnels, all of those tunnels end up in the same hash bucket (their IP protocol numbers are all IP!), leading to increased latency on the receive side. This current behavior limits the scalability of tunnels in Solaris. With the new fanout design described above, the fanout hash function will distribute tunnels to optimize lookups.
- The current fanout simply will not work given the constraints of this new design. A single tunnel link will support both IPv4 and IPv6 DLSAPs, and so the connection to `ip` is not bound to a specific protocol number. The fanout needs to be done using other input. The new fanout design described above proposes to use the outer IP addresses of the tunnel as well as the outer IP version.

The existing protocol fanout will remain untouched for raw sockets, it simply will not be used to fanout tunneled packets.

7.3 Interaction With IPsec

Solaris currently uses per-socket policy to implement IPsec tunnels. This is feasible, as IP tunnels are implemented as raw IP sockets with a STREAMS module (`tun`) that implements IP tunneling pushed on the socket stream, and an `ip` STREAMS module linked above that to form the IP interface. Each tunnel IP interface is represented as a raw IP socket. To configure tunnel IPsec policy, administrators put special `ifconfig` syntax in `/etc/hostname.*` files associated with their tunnel IP interfaces, and `ifconfig` sends special `SIOCSTUNPARAM` ioctls to the `tun` STREAMS module. The tunnel module then issues `IP_SEC_OPT` or `IPV6_SEC_OPT` socket options¹¹ down to the lower instance of `ip`.

The overall model for implementing IPsec tunnels will remain similar in this design. When a tunnel MAC is created through the use of the `iptun_create()` function and ultimately the `IPTUN_CREATE` ioctl, a `conn_t` will be created in `ip` that corresponds to that tunnel. Any `itp_secinfo` IPsec information configured either through `IPTUN_CREATE` or `IPTUN_MODIFY` will cause the tunnel driver to call `ipsec_set_req()` to latch the tunnel's per-socket IPsec policy in the `conn_t` associated with that tunnel. This is equivalent to what happens today when the `tun` STREAMS module issues `IP_SEC_OPT` or `IPV6_SEC_OPT` socket options on its raw socket stream to `ip` upon receiving `SIOCSTUNPARAM` ioctls. The only difference with this design is that it will not be using STREAMS, and will instead call directly into `ip` to set the IPsec policy. As mentioned previously, the `SIOCSTUNPARAM` ioctl will be removed.

Once the per-socket policy for a tunnel has been set, the IPsec processing of packets flowing through tunnel interfaces will be identical to what it is today by virtue of the policy still being per-socket policy latched to the appropriate `conn_t` within `ip`.

This design will maintain the existing `ifconfig` user interface for configuring tunnel IPsec policy for backward compatibility. The `ifconfig` command will call the `iptun_create()` and `iptun_modify()` `libiptun.so` functions, setting the `itp_secinfo` field of the `params` argument, to accomplish this configuration instead of issuing `SIOCSTUNPARAM` ioctls.

Note that the Security organization is starting a project called IPsec Tunnel Reform¹² to address other issues related to IPsec tunnels. The way forward for tunnel IPsec policy will be to migrate the policy configuration from a per-socket model configured through `ifconfig` to a global policy model configured through `ipseconf`. This is the reason why this design does not introduce a method for setting IPsec tunnel policy using `dladm`. This design will only maintain backward compatibility with `ifconfig` to allow the Tunnel Reform Project to more easily transition to this new model without having to deal with backward compatibility with `dladm`.

7.4 kstat Implications

All Nemo drivers are required to supply the set of supported statistics in the `mi_stat` field of the `m_info` structure provided to the `mac` module during `mac_register()`. The IP tunnel driver will support the statistics defined by the `MAC_STAT_MIB()` macro, except for `MAC_STAT_IFSPEED` and `MAC_STAT_COLLISIONS`. Other statistics defined in `mac_stat_t` will not be supported. This will allow `dladm` to display statistics for tunnel interfaces as it does today with other link types. This

¹¹`IP_SEC_OPT` and `IPV6_SEC_OPT` are generic socket options used to configure per-socket policy. See `ipsec(7P)` for details.

¹²<http://atlantic.east/ipsec/projects/tunnel-reform/>

set of statistics is sufficient for the proper operation of `netstat(1M)` command when used with `-i`. Note that this set of kstats will be the same as what the `tun` module provides today.

8 General DLPI Implications

The Nemo framework will provide the DLPI interfaces on behalf of the IP tunnel driver. As a result, all DLPI primitives that Nemo provides will be made available for IP tunnel devices. The following sections outline the ways in which IP tunnel DLPI devices differ from other types of DLPI devices.

8.1 DL_INFO_ACK

The `dl_info_ack_t` structure contained in `DL_INFO_ACK` primitives passed up by IP tunnel DLPI devices will be filled out as follows:

- `dl_max_sdu`

This will be set to the maximum possible size of a tunneled IP datagram (`IP_MAXPACKET` - tunneling overhead). When the tunnel discovers more about the tunnel destination (if there is a tunnel destination), it will adjust the actual SDU by sending up `DL_NOTIFY_IND` messages for `DL_NOTE_SDU_SIZE`.

- `dl_min_sdu`

This will be set to 0.

- `dl_addr_length`

This will be set to `sizeof (in_addr_t)` for IPv4 and 6to4 tunnels, and to `sizeof (in6_addr_t)` for IPv6 tunnels.

- `dl_mac_type`

The MAC type will be a function of the tunnel type using the following mapping:

Tunnel Type	MAC type
IPv4	DL_IPV4
IPv6	DL_IPV6
6to4	DL_6T04

A separate MAC type is needed for each tunnel type, as the data-link layers implemented by each type is significantly different from the others. The SAP space for 6to4 tunnels differs from the others (it only allows IPv6 SAPs), and the address sizes differs between IPv4 (and 6to4) and IPv6 tunnels. Also, only IPv4 and IPv6 tunnels support the concept of a destination link-layer address. The differing MAC types also allow the network layer (the IP layer above the tunnel) to implement special addressing that is specific to the tunnel type. Section 9.3 details how the `ip` module will use `dl_mac_type` to implement the kinds of network-layer addressing required by each tunnel type.

DLPI consumers that have enabled `DLIOCRAW` (such as `snoop`) will need to be aware that `DL_IPV4`, `DL_IPV6`, and `DL_6T04` links pass up packets that begin with an IP header. They can further distinguish between IPv4 and IPv6 based on the specific MAC type.

- `dl_provider_style`

As all DLPI device nodes under `/dev/net` will be style-1 only, this will be set to `DL_STYLE1`.

- `dl_brdcst_addr_length`

The tunnel types implemented by the tunnel driver do not have the concept of broadcast. As such, this will be set to 0.

8.2 DL_BIND_REQ

In Solaris, DLPI consumers use a well-known DLSAP number space for all DLPI devices. For example, when creating an IPv4 interface, the `ip` module issues a `DL_BIND_REQ` for the `IP_DL_SAP` address (0x0800). When creating an IPv6 interface, the `ip` module similarly issues a `DL_BIND_REQ` for the `IP6_DL_SAP` address (0x86dd). These are, in fact, the EtherTypes for IPv4 and IPv6, and the rest of the DLSAP number space is similarly assumed to be populated with EtherType values.

Because of this assumption, when receiving packets, the Nemo framework will map the IP header version number of the network layer header to these DLSAP or EtherType values in order to match DLPI consumers that have bound to those addresses. As such, binding to DLSAP values other than those associated with IPv4 and IPv6 will have no effect, as no other network layer packets will ever be passed up by the IP tunnel driver.

8.3 DL_ENABMULTI_REQ and DL_DISABMULTI_REQ

Because there is no link-layer multicast support for the tunnel types implemented by the IP tunnel driver, these DLPI primitives will be rejected with a `DL_ERROR_ACK` using a `dl_errno` of `DL_NOTSUPPORTED`.

8.4 DL_PROMISCON_REQ and DL_PROMISCOFF_REQ

All promiscuous modes will be supported, but `DL_PROMISC_MULTI` will have no effect as all packets sent and received by the tunnel driver are unicast.

8.5 DLIOCRAW

Solaris DLPI drivers support this ioctl to enable an `M_DATA` raw mode for the DLPI stream. This mode of operation requires that the consumer pass down `M_DATA` messages that include link-layer headers. It also results in received packets being passed up as `M_DATA` messages that also include link-layer headers. Applications like `snoop` use this ioctl to allow full observability of packets sent and received by the driver.

The `dld` module will implement this functionality for the `iptun` driver as it does with other drivers. For IP tunnels, the outer IP header of tunneled packets will be the link-layer header.

8.6 DLIOCHDRINFO or DL_IOC_HDR_INFO

This existing Solaris specific ioctl (it is a single ioctl with two different names) enables what is known in Solaris as fast-path. The ioctl returns a copy of the link-layer header that would be used to transmit packets for a given destination link-layer address. This allows the DLPI consumer to pre-allocate data packets that include this link-layer header, and send them as `M_DATA` messages. The ioctl also results in `M_DATA` messages being passed up for received packets, but unlike `DLIOCRAW`,

it does not include link-layer headers in those received packets. It is mainly used as an optimization for the `ip` module.

As it does with other drivers, the `dld` module will implement this functionality for the `iptun` driver. The link-layer header returned will be an IP header of appropriate version based on the MAC type.

8.7 DL_NOTE_LINK_UP and DL_NOTE_LINK_DOWN

The tunnel driver will notify the Nemo framework of link state changes via the `mac_link_update()` function, which in turn causes the generation of `DL_NOTE_LINK_UP` and `DL_NOTE_LINK_DOWN` DLPI notifications. The link state of a tunnel link will be down until the required configuration for the tunnel link is complete. For configured tunnels, the required configuration is a tunnel source and a tunnel destination. For 6to4 tunnels, the tunnel source is the only piece of required configuration.

9 Changes to the Nemo Framework

The following sections detail all changes required in the Nemo framework to support the IP tunnel driver. It is assumed that the reader is familiar with Nemo and its interfaces¹³.

This work depends on the MAC Type Independent Nemo Architecture¹⁴, which will allow Nemo to support the MAC types defined in this document via the MAC type plugins defined in section 9.1.

The Nemo framework currently only supports Ethernet MAC drivers. Changes to this framework will be required to allow an IP link-layer where the link-layer address is an IP address and link-layer headers are IP headers. In order to meet this requirement, the proposed changes will make Nemo completely MAC layer independent to allow for future adoption of other media types¹⁵.

Note that these changes are preliminary, and more changes may be required as the implementation takes shape. This document will be updated as the required changes become apparent.

Some of these changes are internal to the Nemo framework, and others will affect consumers of the framework. Those changes that affect external consumers of the framework are also summarized in section 9.4.

9.1 Nemo MAC Type Plugins

9.1.1 DL_IPV4 Plugin

This plugin will implement MAC type specific operations for IPv4 configured tunnels. The MAC type for this plugin will be `DL_IPV4`. The physical address length for this MAC type will be 4 bytes (the length of an IPv4 address). No broadcast address will be registered. The plugin will implement the following functions for each plugin callback:

¹³The Nemo framework is detailed in the Nemo Interface Specification document located at <http://sac.sfbay/PSARC/2004/571/final.materials/NemoInterfaceSpecification.pdf>. Sections 3, 4, 5, and 6 in that document detail the interfaces being modified here.

¹⁴<http://clearview.east/docs/XXX>

¹⁵Such future adoption will occur as a result of Clearview's Vanity Naming and Nemo Unification feature documented at http://clearview.east/wiki/index.php/Clearview:Vanity_Naming. In addition to implementing network vanity naming in the Nemo framework, this feature will bring all existing monolithic network drivers under the Nemo framework.

- `mtops_unicst_verify`

```
int ipv4tun_unicst_verify(const uint8_t *addr);
```

This function will return 0 if the given IPv4 address is a valid IPv4 unicast addresses, and EINVAL otherwise. A valid IPv4 unicast address is one that is not multicast nor 255.255.255.255. This function will not verify if the address is specifically an IPv4 unicast address on the system nor verify that the address is not the subnet broadcast addresses of one of the system's interfaces.

- `mtops_multicst_verify`

```
int ipv4tun_multicst_verify(const uint8_t *addr);
```

This function will return ENOTSUP as link-layer multicast is not supported by IPv4 tunnels. Multicast at the IP layer above the tunnel interface is still supported, as the IP interfaces on IPv4 tunnels are point-to-point interfaces.

- `mtops_sap_verify`

```
boolean_t ipv4tun_sap_verify(uint_t sap, uint_t *bind_sap);
```

The SAP space for IPv4 tunnels consists of IPPROTO_ENCAP (for IPv4 in IPv4) and IPPROTO_IPV6 (for IPv6 in IPv4). This function will B_TRUE if either of those protocol numbers are passed in as the `sap` argument, and B_FALSE otherwise. If non-NULL, the `bind_sap` argument will be set to be equal to the `sap` argument.

- `mtops_header`

```
mblk_t *ipv4tun_header(const uint8_t *saddr, const uint8_t *daddr, uint_t sap,
    void *mac_header_data, size_t extra_len);
```

This function will construct an IPv4 header based on the arguments passed in. Both `saddr` and `daddr` must point to buffers of 4 bytes each. The `sap` argument must be IPPROTO_ENCAP or IPPROTO_IPV6. If non-NULL, the `mac_header_data` argument must point to the following structure:

```
typedef struct ipv4tun_header_data_s {
    uint_t ipv4hd_fields;
    uint8_t ipv4hd_ttl;
} ipv4tun_header_data_t;

/* ipv4hd_fields */
enum {
    IPV4HD_TTL = 0x01
};
```

Because the IPv4 TTL is the only piece of header data currently supported, the only field in the structure is `ipv4hd_ttl`. The `ipv4hd_fields` flags field must include IPV4HD_TTL. In the future, the plugin may support additional fields.

If `extra_len` is non-zero, the function will allocate additional space of the indicated length at the end of the `mblk`.

The resulting IPv4 header will have no IP options, and its fields will be set to:

Field Name	Value
IP Version	4
Header Length	<code>sizeof (struct ip)</code>
TOS	0 (will be set on a per-packet basis by the driver)
Total Length	0 (will be set on a per-packet basis by the driver)
ID	0
Fragment Offset	0
Fragment Flags	IPH_DF (don't fragment)
TTL	<code>ipv4hd_ttl</code> if supplied, or 255
Protocol	<code>sap</code> argument
Header Checksum	0 (filled in by lower <code>ip</code> instance)
Source Address	<code>saddr</code> argument
Destination Address	<code>daddr</code> argument

- `mtops_header_info`

```
int ipv4tun_header_info(mblk_t *mp, mac_header_info_t *mhip);
```

This function will fill in the contents of `mhip` according to the IPv4 header at the beginning of `mp`. The `mhi_length` field will be the size of the IPv4 header, `mhi_daddr` will point to the IPv4 destination, `mhi_saddr` to the source, `mhi_sap` will be the protocol number (`IPPROTO_ENCAP` or `IPPROTO_IPV6`), and `mhi_dsttype` will be `MAC_ADDRTYPE_UNICAST`.

9.1.2 DL_IPV6 Plugin

This plugin will implement MAC type specific operations for IPv6 configured tunnels. The MAC type for this plugin will be `DL_IPV6`. The physical address length for this MAC type will be 16 bytes (the length of an IPv6 address). No broadcast address will be registered. The plugin will implement the following functions for each plugin callback:

- `mtops_unicst_verify`

```
int ipv6tun_unicst_verify(const uint8_t *addr);
```

This function will return 0 if the given IPv6 address is a valid IPv6 unicast addresses, and `EINVAL` otherwise. A valid IPv6 unicast address is one that is not multicast. This function will not verify if the address is specifically an IPv6 unicast address on the system.

- `mtops_multicst_verify`

```
int ipv6tun_multicst_verify(const uint8_t *addr);
```

This function will return `ENOTSUP` as link-layer multicast is not supported by IPv6 tunnels. Multicast at the IP layer above the tunnel interface is still supported, as the IP interfaces on IPv4 tunnels are point-to-point interfaces.

- `mtops_sap_verify`

```
boolean_t ipv6tun_sap_verify(uint_t sap, uint_t *bind_sap);
```

The SAP space for IPv6 tunnels consists of `IPPROTO_ENCAP` (for IPv4 in IPv6) and `IPPROTO_IPV6` (for IPv6 in IPv6). This function will `B_TRUE` if either of those protocol numbers are passed in as the `sap` argument, and `B_FALSE` otherwise. If non-NULL, the `bind_sap` argument will be set to be equal to the `sap` argument.

- `mtops_header`

```
mblk_t *ipv6tun_header(const uint8_t *saddr, const uint8_t *daddr, uint_t sap,  
    void *mac_header_data, size_t extra_len);
```

This function will construct an IPv6 header based on the arguments passed in. Both `saddr` and `daddr` must point to buffers of 16 bytes each. The `sap` argument must be `IPPROTO_ENCAP` or `IPPROTO_IPV6`. If non-NULL, the `mac_header_data` argument must point to the following structure:

```
typedef struct ipv6tun_header_data_s {  
    uint_t ipv6hd_fields;  
    uint8_t ipv6hd_hoplimit;  
    uint8_t ipv6hd_encaplimit;  
} ipv6tun_header_data_t;  
  
/* ipv6hd_fields */  
enum {  
    IPV6HD_HOPLIMIT = 0x01,  
    IPV6HD_ENCAPLIMIT = 0x02  
};
```

The `ipv6hd_fields` flags field denotes which fields are supplied. The plugin may support additional fields in the future.

If `extra_len` is non-zero, the function will allocate additional space of the indicated length at the end of the `mblk`.

The resulting IPv6 header will have its fields set to:

Field Name	Value
IP Version	6
Traffic Class	0 (to be set on a per-packet basis by the driver)
Flow ID	0 (to be set on a per-packet basis by the driver)
Payload Length	0 (to be set on a per-packet basis by the driver)
Next Header	<code>sap</code> argument or <code>IPPROTO_DSTOPTS</code>
Hop Limit	<code>ipv6hd_hoplimit</code> if supplied, or 255
Source Address	<code>saddr</code> argument
Destination Address	<code>daddr</code> argument

Note that the “Next Header” field has two possible values. If `ipv6hd_encaplimit` is supplied and non-zero, the “Next Header” will be set to `IPPROTO_DSTOPTS`, and the IPv6 header will be followed by a Destination Options Header containing a single Tunnel Limit Option. The Destination Options Header’s next header field will be set to the `sap` argument, and the Tunnel Limit Option’s `ip6ot_encap_limit` field will be set to `ipv6hd_encaplimit`.

- `mtops_header_info`

```
int ipv4tun_header_info(mblk_t *mp, mac_header_info_t *mhip);
```

This function will fill in the contents of `mhip` according to the IPv6 header at the beginning of `mp`. The `mhi_length` field will be the size of the IPv6 header plus any extension headers optionally following the IPv6 header, `mhi_daddr` will point to the IPv6 destination, `mhi_saddr` to the source, `mhi_sap` will be the next header value of either the IPv6 header or of the last extension header (`IPPROTO_ENCAP` or `IPPROTO_IPV6`), and `mhi_dsttype` will be `MAC_ADDRTYPE_UNICAST`.

9.1.3 DL_6T04 Plugin

This plugin will implement MAC type specific operations for 6to4 tunnels. The MAC type for this plugin will be `DL_6T04`. The physical address length for this MAC type will be 4 bytes (the length of an IPv4 address). No broadcast address will be registered. The plugin will implement the following functions for each plugin callback:

- `mtops_unicst_verify`

```
int tun6to4_unicst_verify(const uint8_t *addr);
```

This function will return 0 if the given IPv4 address is a valid IPv4 unicast addresses, and `EINVAL` otherwise. A valid IPv4 unicast address is one that is not multicast nor 255.255.255.255. This function will not verify if the address is specifically an IPv4 unicast address on the system nor verify that the address is not the subnet broadcast addresses of one of the system's interfaces.

- `mtops_multicst_verify`

```
int tun6to4_multicst_verify(const uint8_t *addr);
```

This function will return `ENOTSUP` as link-layer multicast is not supported by 6to4 tunnels.

- `mtops_sap_verify`

```
boolean_t tun6to4_sap_verify(uint_t sap, uint_t *bind_sap);
```

The SAP space for 6to4 tunnels consists of a single value: `IPPROTO_IPV6`. This function will return `B_TRUE` if `sap` is `IPPROTO_IPV6`, and `B_FALSE` otherwise. If non-NULL, the `bind_sap` argument will be set to be equal to the `sap` argument.

- `mtops_header`

```
mblk_t *tun6to4_header(const uint8_t *saddr, const uint8_t *daddr, uint_t sap,  
                      void *mac_header_data, size_t extra_len);
```

This function will construct an IPv4 header based on the arguments passed in. Both `saddr` and `daddr` must point to buffers of 4 bytes each. The `sap` argument must be `IPPROTO_IPV6`. If non-NULL, the `mac_header_data` argument must point to the following structure:

```
typedef struct tun6to4_header_data_s {  
    uint_t  hd6to4_fields;  
    uint8_t hd6to4_ttl;  
} tun6to4_header_data_t;  
  
/* hd6to4_fields */  
enum {  
    HD6T04_TTL = 0x01  
};
```

Because the IPv4 TTL is the only piece of header data currently supported, the only field in the structure is `hd6to4_ttl`. The `hd6to4_fields` flags field must include `HD6T04_TTL`. In the future, the plugin may support additional fields.

If `extra_len` is non-zero, the function will allocate additional space of the indicated length at the end of the `mblk`.

The resulting IPv4 header will have no IP options, and its fields will be set to:

Field Name	Value
IP Version	4
Header Length	<code>sizeof (struct ip)</code>
TOS	0 (will be set on a per-packet basis by the driver)
Total Length	0 (will be set on a per-packet basis by the driver)
ID	0
Fragment Offset	0
Fragment Flags	<code>IPH_DF</code> (don't fragment)
TTL	<code>ipv4hd_ttl</code> if supplied, or 255
Protocol	<code>sap</code> argument
Header Checksum	0 (filled in by lower ip instance)
Source Address	<code>saddr</code> argument
Destination Address	<code>daddr</code> argument

- `mtops_header_info`

```
int tun6to4_header_info(mblk_t *mp, mac_header_info_t *mhip);
```

This function will fill in the contents of `mhip` according to the IPv4 header at the beginning of `mp`. The `mhi_length` field will be the size of the IPv4 header, `mhi_daddr` will point to the IPv4 destination, `mhi_saddr` to the source, `mhi_sap` will be the protocol number (`IPPROTO_IPV6`), and `mhi_dsttype` will be `MAC_ADDRTYPE_UNICAST`.

9.2 DL_NOTE_SDU_SIZE

Solaris' DLPI implementation contains a notification mechanism that allows DLPI consumers to be asynchronously notified of particular events. The `dlpi(7P)` man page defines these notifications, one of which is `DL_NOTE_SDU_SIZE`. This notification is issued when the device alters its maximum send data unit size (aka MTU).

A configured tunnel has a dynamic MTU that is based on the Path MTU of the tunnel destination. In order to adjust the MTU of IP interfaces when underlying DLPI devices change their SDU size like IP tunnels do, IP currently registers to receive `DL_NOTE_SDU_SIZE` notifications from DLPI devices using the mechanism described above.

The `mac` module does not currently provide a way for drivers to issue such notifications. A new function, `mac_sdu_update(mac_t *)`, will be created that MAC drivers can call to notify the `mac` module of such changes. This function will in turn send a `MAC_NOTE_SDU_SIZE` notification using `i_mac_notify()`.

The `str_notify()` function in the `dld` module is the existing callback to receive such notifications, and it will issue `DL_NOTE_SDU_SIZE` notifications to interested DLPI consumers in response to `MAC_NOTE_SDU_SIZE` callbacks. Note that this mechanism mimics how existing notifications (such as `DL_NOTE_PHYS_ADDR` and `DL_NOTE_LINK_UP`) work.

9.3 Tunnel Source and Destination via DLPI

Solaris' IP implementation currently implements particular ease-of-configuration features for IPv6 tunnel interfaces. Specifically, when an IPv6 tunnel interface is configured on IPv4 or IPv6 tunnel links, the IPv6 addresses for that interface are initially automatically created based on the tunnel's underlying tunnel source and tunnel destination addresses. For 6to4 tunnels, the IPv6 address of the IP interface is core to the functionality provided by those tunnels. Examples:

```
# ifconfig ip.6to4tun0 inet6 plumb tsrc 10.8.57.3 up
# ifconfig ip.6to4tun0 inet6
ip.6to4tun0: flags=2300041<UP,RUNNING,ROUTER,NOUD,IPv6> mtu 65515 index 4
    inet tunnel src 10.8.57.3
    tunnel hop limit 60
    inet6 2002:a08:3903::1/64
```

In the example above, the tunnel source address is embedded in the IPv6 prefix used to configure the IP interface. This is core to how 6to4 functions (see [RFC 3056] for details).

```
# ifconfig ip.tun0 inet6 plumb tsrc 129.148.174.103 tdst 10.8.57.3 up
# ifconfig ip.tun0 inet6
ip.tun0: flags=2200851<UP,POINTOPOINT,RUNNING,MULTICAST,NOUD,IPv6> mtu 1480 index 4
    inet tunnel src 129.148.174.103 tunnel dst 10.8.57.3
    tunnel hop limit 60
    inet6 fe80::8194:ae67/10 --> fe80::a08:3903
```

In this example, the interface-id used to generate the IPv6 link-local source address of the interface is the IPv4 tunnel source, and the interface-id used to generate the IPv6 link-local destination address of the interface is the IPv4 tunnel destination. [RFC 2893] describes this method of generating link-local addresses in section 3.7. Note that these interface-id's are the same id's used by `in.ndpd(1M)` when generating global IPv6 addresses using the stateless address autoconfiguration mechanism.

The `ip` module currently does this automatic IPv6 address assignment by intercepting `ifconfig`'s `SIOCSTUNPARAM` ioctls¹⁶ and snooping the tunnel type, the tunnel source, and the tunnel destination from the ioctls' data. The tunnel type lets `ip` know what algorithm to use to generate the IPv6 addresses, and the tunnel addresses are used as input into that algorithm. A different mechanism will be needed to obtain this information because the configuration of tunnel link properties will be done by opening the tunnel control device, not an IP socket.

Obtaining the tunnel type will be trivial, since as described in section 8.1, each tunnel type will be represented by a different MAC type, and the MAC type will be available to the `ip` module when it receives a `DL_INFO_ACK` from the DLPI device. The `ip` module will use the MAC type to determine the algorithm to use when generating IPv6 addresses.

The tunnel addresses will be made available as the "physical addresses" of the underlying tunnel device. There are two mechanisms used in DLPI to obtain physical addresses: the `DL_PHYS_ADDR_REQ` and `DL_PHYS_ADDR_ACK` messages, and the `DL_NOTIFY_IND` `DL_NOTE_PHYS_ADDR` notification mechanism. Both mechanisms give access to different address types, one of which is `DL_CURR_PHYS_ADDR`. For tunnel devices, `DL_CURR_PHYS_ADDR` will be the tunnel source address.

No DLPI physical address type currently exists to represent the tunnel destination. A new type, `DL_CURR_DEST_ADDR`, will be defined for this purpose. The `ip` module will request this type of address in addition to `DL_CURR_PHYS_ADDR` for `IFF_POINTOPOINT` interfaces.

In order to support this, the Nemo framework will need to be updated to maintain a destination address that can be returned in response to such a request. First, the `mac` module will need to maintain a destination address for the MAC driver. The `mac_info_t` structure that is currently included in the `mac_t` provided by the driver at `mac_register()` time will be augmented to contain a field named `mi_dest_addr`. The `dld` module obtains this `mac_info_t` using the existing

¹⁶Remember, `ifconfig` currently configures tunnel properties by sending these ioctls to a `SOCK_DGRAM` socket with the expectation that the `ip` module will forward them to the appropriate `tun STREAMS` module down below.

`mac_info()` client function, so it will have access to the destination address in order to respond to `DL_CURR_DEST_ADDR` requests.

The `mi_dest_addr` field will be dynamically allocated like `mi_unicst_addr` and `mi_brdcst_addr`. If it is `NULL`, `dld` will return a `DL_ERROR_ACK` message in response to `DL_CURR_DEST_ADDR` requests for MACs that do not support the concept of a destination address.

A `mac_dest_update()` function will be provided for the MAC driver to use when a change in the destination address has been made. The `mac` module will issue `MAC_NOTE_DEST` notifications when a driver calls this function, allowing the `dld` module to in turn issue `DL_NOTE_PHYS_ADDR` notifications for `DL_CURR_DEST_ADDR` address types. This is analogous to the way other notifications such as `DL_CURR_PHYS_ADDR` are implemented.

9.4 Summary of Changes to Nemo Interfaces

Some of the changes listed in the above sections will impact consumers of the Nemo framework. Note that none of the Nemo interfaces being modified are documented, although the MAC driver interfaces will most likely be documented in the future. The changes are summarized here in terms of each type of Nemo consumer:

9.4.1 MAC driver changes

- `mac_info_t`

The `mi_dest_addr` field will be added to the `mac_info_t` structure, a structure that MAC drivers fill in as the `m_info` field of the `mac_t` that is registered using `mac_register()`. The new `mi_dest_addr` field and the existing `mi_unicst_addr` and `mi_brdcst_addr` fields will be pointers, and will be dynamically allocated by the driver.

- `mac_dest_update()`

This new driver support function will be used by drivers to update the destination address.

- `mac_sdu_update()`

This new driver support function will be used by drivers to update the maximum SDU or MTU.

9.4.2 MAC client changes

- `MAC_NOTE_SDU_SIZE`

Clients of the `mac` module will be able to register for this notification using the existing `mac_notify_add()` function. Notification callbacks that register for `MAC_NOTE_SDU_SIZE` will be called when the driver calls `mac_sdu_update()`.

- `MAC_NOTE_DEST`

Clients of the `mac` module will be able to register for this notification using the existing `mac_notify_add()` function. Notification callbacks that register for `MAC_NOTE_DEST` will be called when the driver calls `mac_dest_update()`.

9.4.3 DLD client changes

- `DL_NOTE_SDU_SIZE`

The `dld` module will support notifications for `DL_NOTE_SDU_SIZE`.

- `DL_CURR_DEST_ADDR`

DLPI clients will be able to request this type of address in `DL_PHYS_ADDR_REQ` messages. This type of address will also be included in `DL_NOTE_PHYS_ADDR` notifications when the destination address of the MAC changes.

10 IP Tunneling SMF Service

The configuration of IP tunnel interfaces at boot time needs to be done after all other IP interfaces have been configured because tunnel source addresses are actually IP addresses that have been configured on other interfaces. It also needs to be done after IPsec has been initialized so that IPsec tunnels can function properly. Because of these two dependencies, this configuration is currently done at the end of the `svc:/network/initial` SMF service, which contains IPsec configuration and runs after `svc:/network/physical` (the SMF service that configures all other IP interfaces).

The `svc:/network/physical` SMF service defers the configuration of IP tunnel interfaces by ignoring IP interface configuration files named `/etc/hostname.ip*.*` or `/etc/hostname6.ip*.*`. Because IP tunnel interface names will no longer be required to adhere to the `ip*.*` convention, the `svc:/network/physical` service will use `dladm`'s `show-iptun` command to discover which interfaces to ignore.

It is also beneficial to separate the configuration of IP tunnel interfaces from the other configuration tasks done in `svc:/network/initial`. Thus, a new service named `svc:/network/iptun` will be created, and will depend on both `svc:/network/physical` and `svc:/network/initial`. Really, the only part of `svc:/network/initial` that the configuration of IP tunnel interfaces depends on is the IPsec configuration. If the IPsec configuration is ever split into its own service, then `svc:/network/iptun` should be made to depend on that service instead of `svc:/network/initial`.

As mentioned in section 6.3, `libiptun.so` will use a text file to store persistent configuration information. This design decision was made to be consistent with other `dladm` configuration information stored in `/etc/aggregation.conf`. In the future, this configuration information can be merged into the SMF repository.

11 Dependency on Network Vanity Naming

This design depends on Network Vanity Naming¹⁷, another piece of the Clearview project. It presupposes the ability to associate an IP tunnel with an arbitrarily named `dls` link. It depends on this functionality in order to provide link names that are backward compatible with the interface names currently used for IP tunnels.

For example, an IPv4 configured tunnel is created today by plumbing an IP interface named `ip.tun0` using the `ifconfig` command. For backward compatibility, this design proposes to keep this capability by having `ifconfig` use `libiptun.so`'s `iptun_create()` to create a tunnel, passing it a link name of `ip.tun0`. This link needs to be associated with the IP tunnel MAC, but the Nemo framework today (without Clearview's Vanity Naming) does not allow these two entities (the link and the MAC) to have different names.

The Vanity Naming feature is what will decouple the link name from the MAC name entirely, giving this this design the capability to easily implement its backward compatibility requirement.

¹⁷http://clearview.east/wiki/index.php/Clearview:Vanity_Naming

12 Impact on libdlpi.so

As mentioned previously, the tunnel IP interface configuration mechanism done by `ifconfig` hinges on an obscure and undocumented interface name parsing “feature” implemented in `libdlpi.so`’s `dlpi_if_open()` function. This feature allows the configuration of STREAMS plumbing to be embedded in an IP interface name. The syntax is as follows:

```
device[.module[.module...]]#
```

Given the above interface name syntax, the `dlpi_if_open()` function opens `/dev/device`, and iteratively pushes (using `I_PUSH`) every STREAMS module denoted by *module* onto the device stream. Once all modules have been pushed, it then attaches to the PPA denoted by `#`.

For example, calling `dlpi_if_open()` with an interface name of `ip.tun0` causes the function to first open `/dev/ip`, then `I_PUSH` the `tun` STREAMS module on the `/dev/ip` stream, then `DL_ATTACH` to PPA 0.

This design will abolish the STREAMS plumbing semantics behind this syntax on the premise that they were created to implement the configuration of IP tunnel interfaces, are currently only used for that purpose, are undocumented, and are inherently flawed. The following bug which will be fixed by implementing this design further discusses the flawed nature of this feature:

```
4505468 network interface names can confuse, lie, and deceive
```

While the plumbing semantics behind the interface naming will be removed by this design, the naming syntax itself will still be allowed for backward compatibility with existing tunnel interface names. For example, for `ifconfig` to plumb an interface named `ip.tun0`, it will call `dlpi_if_open()` as it does today using `ip.tun0` as the interface name, and that function will simply open the `/dev/net/ip.tun0` style-1 DLPI device. A change to the NOTES section of `dlpi(7P)` will be needed to document that the `'.'` character is acceptable in DLPI device node names.

Inserting and removing STREAMS modules in an IP interface stream will still be possible through the `ifconfig modinsert` and `modremove` subcommands.

13 Impact on <sys/dlpi.h>

The Nemo framework will export DLPI devices for IP tunnels with the `dl_mac_type` set to `DL_IPV4`, `DL_IPV6`, or `DL_6T04` depending on the tunnel type. The first two are already defined in `<sys/dlpi.h>` to represent IPv4 and IPv6 tunnel links. `DL_6T04` will be added to this header file under the private media types already defined therein.

14 Impact on STREAMS Autopush

Because this design introduces a STREAMS-based DLPI device for tunnel interfaces, it will become possible to configure miscellaneous STREAMS modules to be automatically pushed onto IP tunnel interface streams using the `autopush(1M)` command. This is currently not possible in Solaris.

For example, the use of STREAMS-based packet filtering software such as IP filter and Checkpoint’s Firewall-1 will now be possible on IP tunnel interfaces¹⁸.

¹⁸A project called “Packet Filtering Hooks” is underway to supersede this specific use of `autopush`. This project

15 Impact on Mobile IP

The Mobile IP agent (`mipagent(1M)`) depends on a type of tunnel called a “reverse tunnel” as described in [RFC 2344]. The `mipagent` daemon implements this feature by creating a configured IPv4 tunnel between the foreign and home agents on behalf of a mobile node using the mobile node’s home-address as the IP interface’s source or destination address (depending on whether `mipagent` is running as a home or foreign agent).

There are two issues with this functionality:

1. Duplicate `ifconfig` code:

The `mipagent` daemon currently duplicates all of `ifconfig`’s code for plumbing IP tunnels. It would need to be modified to call the `libiptun.so` API instead.

2. Duplicate tunnels:

The `mipagent` daemon blindly creates duplicate tunnels. In other words, it creates tunnels between the same home and foreign agents using the same tunnel source and tunnel destinations. It does this when multiple mobile nodes roaming in the same foreign network need to tunnel packets back to the same home agent. This is problematic because there is no way for the `ip` module to distinguish between these duplicate tunnels when fanning out packets. This results in a single tunnel receiving *all* tunneled packets.

If these different tunnels have different IPsec policies, the result is that only one of the tunnels’ policy is applied to all packets regardless of which tunnel these packets were destined for.

The daemon would need to be fixed to create a single tunnel between a foreign and home agent pair, and create additional logical point-to-point IP interfaces on that single tunnel for each mobile node requiring a reverse tunnel. In order to fix the IPsec policy problem mentioned above, `mipagent` would need to use future features to be introduced by the IPsec Tunnel Reform project mentioned in section 7.3.

Given that these changes are significant and would require the extensive testing of a technology that does not work and is arguably not used by any single customer, the `mipagent` functionality will be EOL’ed as part of this project¹⁹.

16 EOL of Automatic Tunnels

Those familiar with Solaris’ existing set of IP tunneling functionality will notice that this design does not implement automatic IPv6 tunneling as described in section 5 of [RFC 2893]. This type of tunneling is being deprecated by the IETF, and will be EOL’ed as part of this project.

The ”Basic Transition Mechanisms for IPv6 Hosts and Routers” IETF document currently being edited as [draft-ietf-v6ops-mech-v2] (meant to eventually obsolete [RFC 2893]) has removed all references to automatic tunneling as a transition mechanism, and justifies the removal by stating:

RFC 2893 contains a mechanism called automatic tunneling. But a much more general mechanism is specified in [RFC 3056] which gives each node with a (global) IPv4 address a /48 IPv6 prefix i.e., enough for a whole site.

will allow IP filtering software to intercept packets as they flow through IP stack using functional callbacks instead of using STREAMS as a means to intercept data between ip and link-layer devices. See PSARC/2005/334 “Packet Filtering Hooks” for details.

¹⁹While this is the current plan, further investigation is needed to assert that this is the correct course of action. This plan may change.

The general mechanism being referred to is 6to4, which will be supported in Solaris and implemented by this design.

Automatic tunnels use special kinds of IPv6 addresses called "IPv4-compatible IPv6 addresses" defined in [RFC 3513]. This RFC is being obsoleted and replaced in the IPv6 Working Group by [draft-ietf-ipv6-addr-arch-v4], and IPv4-compatible addresses are being deprecated as part of this work:

The "IPv4-compatible IPv6 address" is now deprecated because the current IPv6 transition mechanisms no longer use these addresses. New or updated implementations are not required to support this address type.

Nothing needs to be done in Solaris to deprecate this address type other than remove automatic tunneling functionality, but the above citation serves as further justification for the EOL of this feature.

A Configuration Example

To help understand how the design pieces fit with the administrative model, here we provide an example of how an IP tunnel interface is created and configured.

We begin by creating an IP tunnel link:

```
# dladm create-iptun -s 66.1.2.3 -d 192.4.5.6 mytunnel0
```

This will create an IP tunnel link between 66.1.2.3 and 192.4.5.6. The tunnel type will default to `ipv4`. This operation will result in the creation of a DLPI device named `/dev/net/mytunnel0`. Figure 2 depicts the kernel objects that will be created as a result of this operation.

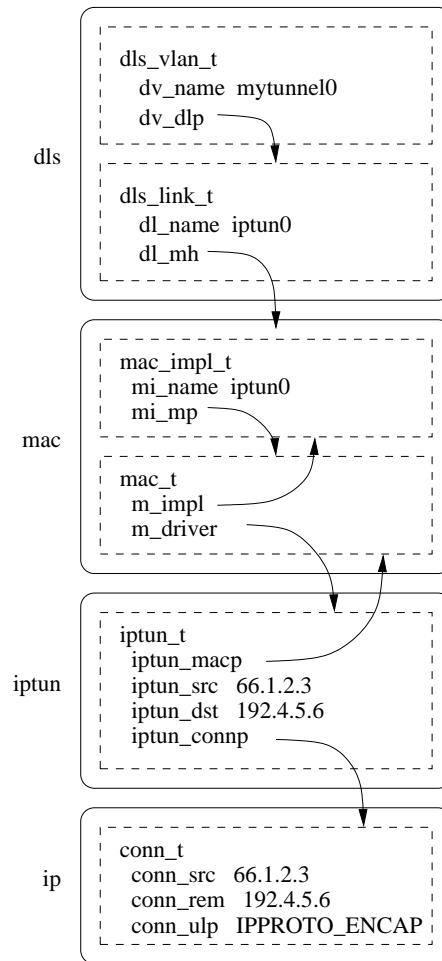


Figure 2: The creation of an IP tunnel link will result in the `iptun` driver registering a MAC with the `mac` module, which in turn will result in the creation of a Nemo link. Setting the tunnel source and destination will also cause `iptun` to create a `conn_t` in `ip`, and bind to the source and destination.

Now that a tunnel link has been created, an IP interface can be plumbed on this tunnel:

```
# ifconfig mytunnel0 plumb 10.1.0.1 11.2.0.2 up
```

This will cause `ifconfig` to open the DLPI device `/dev/net/mytunnel0` and plumb an IP interface just as it would with any other DLPI device. Figure 3 depicts the kernel objects that will be involved in this operation.

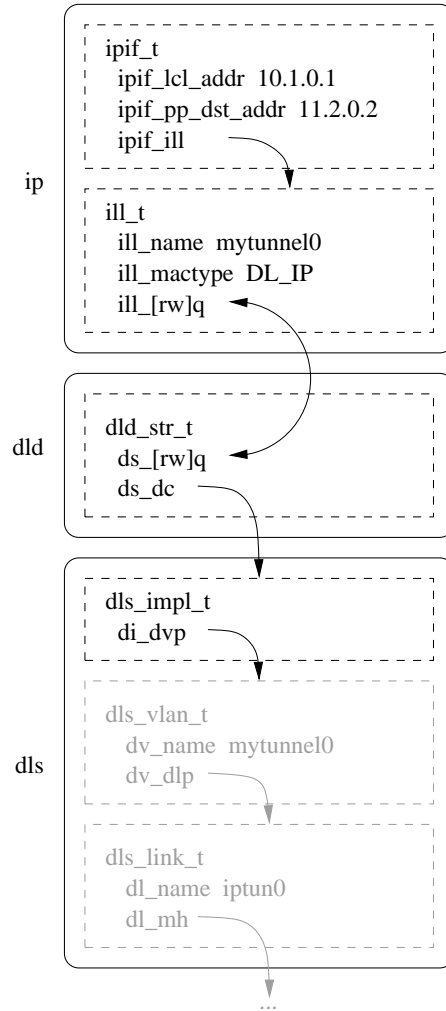


Figure 3: The grayed objects were created in the previous step. When `ifconfig` opens `/dev/net/mytunnel0`, `dld_open()` will be called which will result in the creation of a `dld_str_t` and linkage between this `dld` stream and the `dls` data-link called `mytunnel0`. An IP interface represented by an `ipif_t` and an `ill_t` will be created on this stream. Note that the STREAMS queue pair depicted here will not be the only method used by `ip` to send and receive data. There is an existing capability mechanism described as “polling” that bypasses STREAMS entirely using a direct function call interface between Nemo and `ip`. That is not depicted here.

A new benefit of this design is that administrators may use `snoop` or other DLPI-based observability tools to monitor packets flowing over the tunnel. For example:

```
# snoop -d mytunnel0
```

The `snoop` command will open the DLPI device `/dev/net/mytunnel0` as it would with any other network interface. Figure 4 depicts what will happen in the kernel when `snoop` opens `mytunnel0`.

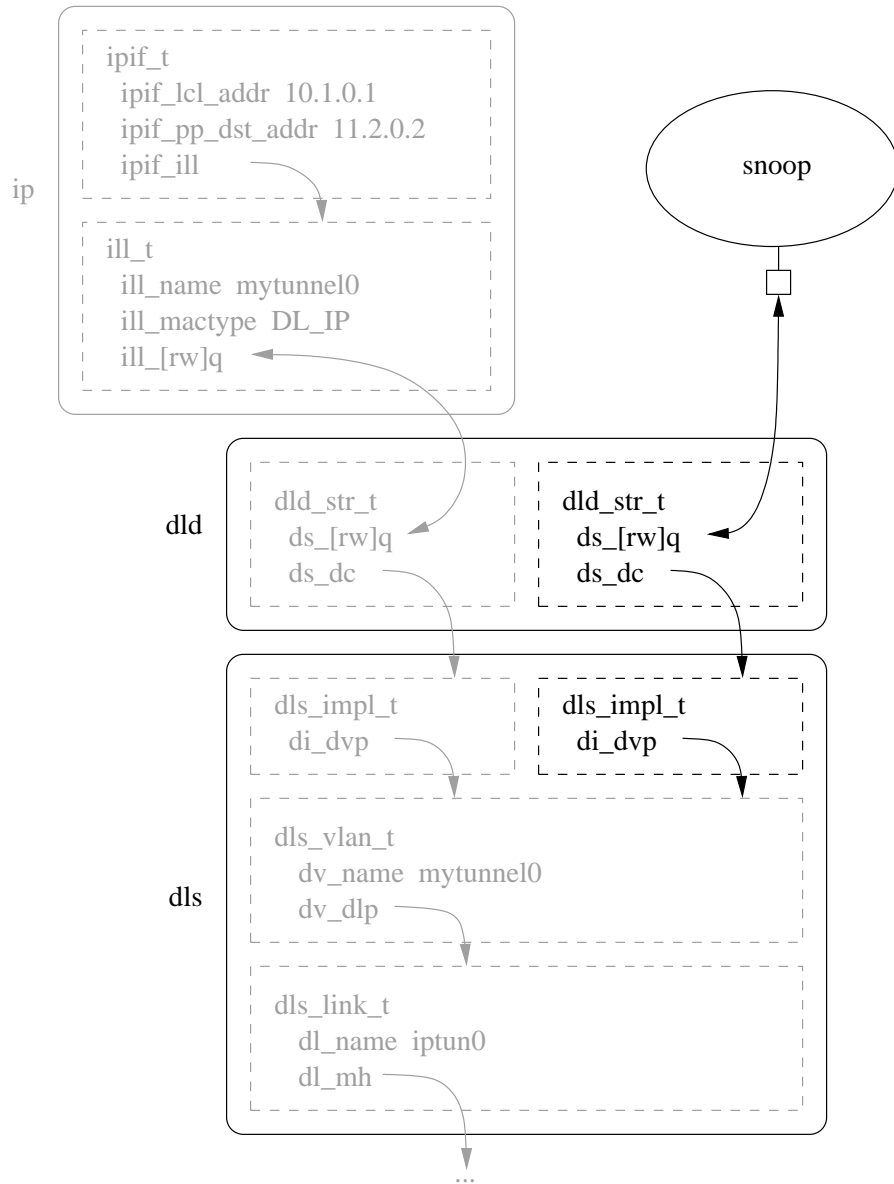


Figure 4: The `snoop` command opens `/dev/net/mytunnel0`, resulting in the creation of new `dld_str_t` and `dls_impl_t` structures.

B <libiptun.h>

The <libiptun.h> header file will contain all necessary data structures for consumers of the libiptun.so library described in section 6. Note that this information is very preliminary and will change as the implementation takes form.

```
/* applyflags */
#define IPTUN_APPLY_NOW          0x01    /* apply changes to running system */
#define IPTUN_APPLY_LATER       0x02    /* apply changes to config. file */

typedef enum {
    IPTUN_OK = 0,
    IPTUN_E_TYPE,
    IPTUN_E_SRC,
    IPTUN_E_DST,
    IPTUN_E_HOPLIMIT,
    IPTUN_E_ENCAPLIMIT,
    IPTUN_E_SECINFO
} iptun_err_t;

typedef enum {
    IPTUN_TYPE_IPV4,
    IPTUN_TYPE_IPV6,
    IPTUN_TYPE_6TO4
} iptun_type_t;

typedef struct iptun_params {
    uint_t          itp_fields;
    char            itp_name[LIFNAMSIZ];
    iptun_type_t   itp_type;
    const char      *itp_src;
    const char      *itp_dst;
    uint8_t         itp_hoplimit;
    uint8_t         itp_encaplimit;
    ipsec_req_t     itp_secinfo;
} iptun_params_t;

/* itp_fields */
#define ITP_NAME          0x00000001
#define ITP_TYPE          0x00000002
#define ITP_SRC           0x00000004
#define ITP_DST           0x00000008
#define ITP_HOPLIMIT      0x00000010
#define ITP_ENCAPLIMIT    0x00000020
#define ITP_SECINFO       0x00000040

typedef void (iptun_walk_func_t)(const char *, const iptun_params_t *, void *);
```

C <sys/iptun.h>

The <sys/iptun.h> header file will contain all necessary data structures for consumers of the IP tunnel control device described in section 7.1. Note that this information is very preliminary and will change as the implementation takes form.

```
typedef struct iptun_kernparams {
    linkid_t      itk_linkid;
    uint_t       itk_fields;
    iptun_type_t  itk_type;
    struct sockaddr_storage itk_src;
    struct sockaddr_storage itk_dst;
    uint8_t      itk_hoplimit;
    uint8_t      itk_encaplimit;
    ipsec_req_t  itk_secinfo;
} iptun_kernparams_t;

/* itk_fields */
#define ITK_TYPE      0x00000001
#define ITK_SRC      0x00000002
#define ITK_DST      0x00000004
#define ITK_HOPLIMIT 0x00000008
#define ITK_ENCAPLIMIT 0x00000010
#define ITK_SECINFO  0x00000020

typedef struct iptun_create {
    char          itc_linkname[LIFNAMSIZ];
    iptun_kernparams_t itc_params;
} iptun_create_t;
```

D <sys/iptun_impl.h>

The <sys/iptun_impl.h> header file will contain all data structures that are private to the IP tunneling kernel module. No external interfaces will be visible here. Note that this information is very preliminary and will change as the implementation takes form.

```
/*
 * There is an iptun_t structure for every IP tunnel MAC. They are
 * created by using the IPTUN_CREATE ioctl and stored in a private hash
 * table.
 */
typedef struct iptun {
    kmutex_t      iptun_lock;
    uint32_t      iptun_instance;
    mac_t         *iptun_macp; /* The thing above us */
    conn_t        *iptun_connp; /* The thing below us */
    iptun_type_t  iptun_type;
    in6_addr_t    iptun_src;
    in6_addr_t    iptun_dst;
    uint32_t      iptun_mtu;
```

```
    clock_t      iptun_pmtu_lastupdated;
    uint8_t      iptun_hoplimit;
    uint8_t      iptun_encaplimit;
    ipsec_req_t  iptun_secinfo;
    uint32_t     iptun_ipsec_overhead;
} iptun_t;
```

References

- [RFC 3056] “Connection of IPv6 Domains via IPv4 Clouds,” B. Carpenter, K. Moore, February 2001.
- [RFC 2893] “Transition Mechanisms for IPv6 Hosts and Routers,” R. Gilligan, E. Nordmark, August 2000.
- [RFC 2473] “Generic Packet Tunneling in IPv6,” A. Conta, S. Deering, December 1998.
- [RFC 2401] “Security Architecture for the Internet Protocol,” S. Kent, R. Atkinson, November 1998.
- [RFC 2344] “Reverse Tunneling for Mobile IP,” G. Montenegro, May 1998.
- [RFC 3513] “Internet Protocol Version 6 (IPv6) Addressing Architecture,” R. Hinden, S. Deering, April 2003.
- [draft-ietf-v6ops-mech-v2] “Transition Mechanisms for IPv6 Hosts and Routers,” E. Nordmark, R. E. Gilligan, March 2005.
- [draft-ietf-ipv6-addr-arch-v4] “IP Version 6 Addressing Architecture,” R. Hinden, S. Deering, May 2005.