IP Tunneling Device Driver Design Specification

Sebastien Roy

Project Clearview I-Team
clearview-discuss@opensolaris.org

Solaris Networking
Sun Microsystems, Inc.

Revision 1.6
August 5, 2009

Contents

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 Overview[1]. 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. It also does not interact with any common framework where such observability will be implemented in the future[2]. 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)`. With the exception of IPv6 automatic tunneling which is being removed, 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.

   A sub-requirement for backward compatibility is the ability to create, delete, and modify IP tunnels from within non-global zones. This capability was introduced by IP instances, and must be preserved by this project.

2. Must export a device node under `/dev/net`[3] 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.

[1] http://clearview.east/docs/overview.txt

[2] See PSARC/2009/232 Packet Capture for OpenSolaris

[3] With the integration of the Clearview UV component, networking DLPI devices are found under `/dev/net`. For details on the Clearview Vanity Naming functionality, see `http://www.opensolaris.org/os/project/clearview/uv/`

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.

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 High-Level Architectural Overview

The IP tunnel device driver will be a Nemo (GLDv3) network driver that registers tunnel links with the GLDv3 framework. Each tunnel link will thus have a DLPI device node under `/dev/net`. This will allow observability tools like `snoop` and `wireshark` 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 `libdladm.so` library. This library will be responsible for interacting with the `iptun` kernel module using a set of ioctls through the `dld` control device.

IP interfaces over 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 (this is done within the `dlpi_open()` implementation described in section 9.9), 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 `dlpi_open()` implementation that `ifconfig` depends on 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.

For backward compatibility, plumbing IP interfaces named `ip.tun#`, `ip6.tun#`, or `ip.6to4tun#` with ifconfig will implicitly result in the creation of IP tunnel links with those names. In addition, the IP tunneling specific `ifconfig` subcommands `tsrc`, `tdst`, `thoplimit`, and `encaplimit` will continue to behave as before. This will be implemented by having `ifconfig` call `libdladm.so` functions instead of using `libinetcfg.so` as it does today.

It will be possible to create IP tunnel links and IP interfaces from within non-global zones. This is also done for backward compatibility, as it is possible today to plumb IP tunnels from non-global zones. The architectural pieces necessary are introduced by PSARC/2009/410. This design introduces a `sys_iptun_config` privilege that can be delegated to non-global zones, allowing them to create, modify, and delete IP tunnel links.

It will also be possible to assign IP tunnel links to exclusive stack non-global zones as described in the following RFE:

```
6218826 need to be able to tunnel into a zone
```

The `tun` STREAMS module along with its friends `atun` and `6to4tun` will be removed, as their

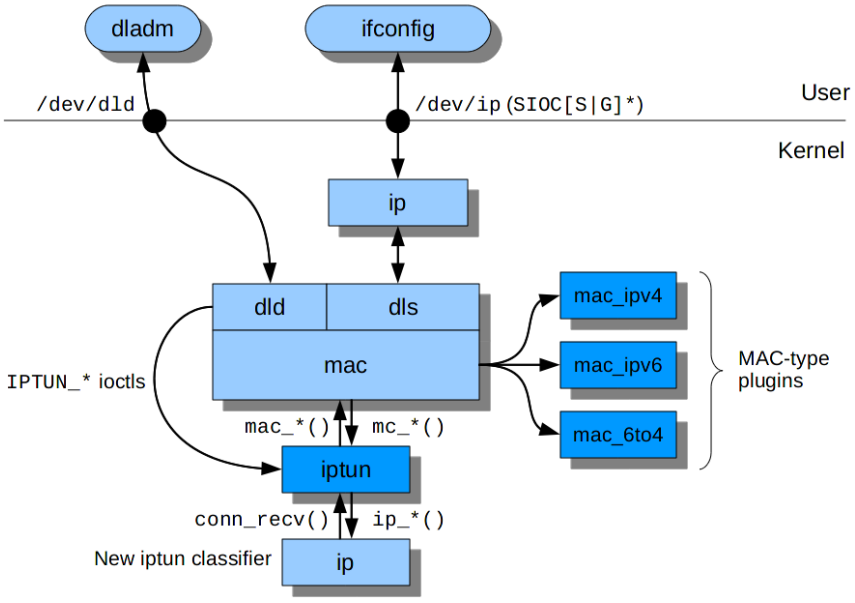

Figure 1: High-level relationship between the system components that are involved in the Clearview IP tunneling design.

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, three new MAC-Type plugin modules will be implemented to support the `DL_IPV4`, `DL_IPV6`, and `DL_6TO4` MAC types that the tunnel driver will implement.

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.

IP tunnel links will represent a new "iptun" class of data-link[4], and new subcommands will allow the administrator to configure every aspect of this class of data-link.

These dladm subcommands will refer to IP tunnel data-links by name, in-line with how dladm handles other classes of data-links. 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. IP tunnel links will be displayed by dladm show-link, and tunnel link names may also be changed using dladm rename-link as with any other class of link.

4.1 New dladm Subcommands

This section describes the new dladm subcommands introduced by this project.

All new IP tunnel operations will be implemented by calling into the libdladm.so library. The changes being made to libdladm.so are described in section 6.

For each new subcommand, 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, the command will result in changes to both the running system and to persistent configuration that will be applied automatically when the system reboots.

Many dladm subcommands take a -R {root-dir} argument which specifies an alternate root in which to apply persistent configuration changes. The subcommands introduced for IP tunneling also have this capability, and this is implemented in the same way as existing dladm subcommands[5].

4.1.1 create-iptun

dladm create-iptun [-t] [-R *root-dir*] -T *type* [-s *tsrc*] [-d *tdst*] *name*

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

The mandatory *type* argument can be any one of the following[6]:

- 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.

[4] See the dladm(1M) man page for the definition of "class"

[5] Alternate root support for dladm is implemented by inserting dladm commands to the /var/svc/profile/upgrade_datalink script which is run by svc:/network/physical when the alternate root is next booted

[6] Note that no new types are being introduced by this project. These are existing IP tunnel types currently implemented in Solaris.

- `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 command will accept literal IP addresses or hostnames for the *tsrc* and *tdst* arguments. Note that if using a hostname, that name will be resolved to IP addresses. In order to minimize accidental misconfigurations, the command will fail if that name resolves to more than one IP address. Hostnames will be stored verbatim into the configuration storage to be applied at subsequent boots (if not using `-t` to make the configuration temporary). Because IP tunnels are created before naming services have been brought online during the boot process, it's important that any hostnames used in the persistent configuration be included in `/etc/hosts`. Also note that 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. The tunnel is of type `ipv4` or `ipv6` and another tunnel exists with the same source and destination addresses.
2. The tunnel being created is a 6to4 tunnel and another 6to4 tunnel exists with the same tunnel source address.

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:

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

The *name* argument is mandatory, and will be the name of the tunnel data-link. This name will be the user's handle to the tunnel for other `dladm` tunnel operations such as `modify-iptun` and `delete-iptun` described below. It will also be the name of the data-link node created in `/dev/net`. Throughout this document, the term "tunnel name" refers to this name.

Examples:

```
# dladm create-iptun -T ipv4 -s 63.1.2.3 -d 192.4.5.6 vpn0
# dladm create-iptun -T 6to4 -s 63.4.5.6 ipv6gateway12
# dladm create-iptun -T ipv6 -s 2001:db8:feed::1234 \
-d 2001:db8:beef::4321 ip6.tun0
```

4.1.2 `modify-iptun`

`dladm modify-iptun [-t] [-R` *root-dir*`] [-s` *tsrc*`] [-d` *tdst*`]` *name*

The modify-iptun subcommand will be used to modify the source or destination address of the tunnel. It will call into libdladm.so's dladm_iptun_modify() function. This operation will fail if the tunnel source or destination address are invalid or inappropriate for the type of the tunnel. The operation is applicable even after IP interfaces have been configured on the tunnel link, and any changes will take effect immediately.

4.1.3 delete-iptun

```
dladm delete-iptun [-t] [-R root-dir] name
```

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

4.1.4 show-iptun

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

The show-iptun subcommand will print tunnel configuration. By default (without -P), it prints the configuration of the running system. With -P, it prints the persistent configuration.

Example:

```
# dladm show-iptun
LINK         TYPE  FLAGS  SOURCE              DESTINATION
vpn0         ipv4  s-     63.1.2.3            192.4.5.6
ipv6router0  6to4  --     63.4.5.6            --
ipv6site2    ipv6  --     2001:db8:feed::1234 2001:db8:beef::4321
ip.tun0      ipv4  -i     10.0.0.1            10.0.0.2
# dladm show-iptun vpn0
LINK         TYPE  FLAGS  SOURCE              DESTINATION
vpn0         ipv4  s-     63.1.2.3            192.4.5.6
```

Note that only two flags are defined for the FLAGS column. The s flag denotes that IPsec policy for this tunnel link exists as displayed by the ipsecconf(1M) command. The i flag denotes that the tunnel was implicitly created, and will be implicitly deleted on last close (see sections 5 and 8.2).

The -p option causes show-iptun to print machine-parsable output using the existing dladm parsable output format.

4.1.5 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 persistently created. It will call libdladm.so's dladm_iptun_up() function. Its main purpose will be to re-create all persistent tunnels at boot-time.

4.1.6 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 libdladm.so's dladm_iptun_down() function. 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).

4.2 IP Tunnel Link Properties

The dladm command provides the set-linkprop, reset-linkprop, and show-linkprop subcommands to manage link properties. Two new properties specific to IP tunnel links are being introduced.

4.2.1 hoplimit

The hoplimit property is applicable to all types of IP tunnels. For ipv4 and 6to4 tunnels (tunnel types with outer IPv4 headers), this specifies the TTL of the encapsulating IPv4 header. For ipv6 tunnels, this specifies the hop limit of the encapsulating IPv6 header. The default value is 64, and valid values range from 1 to 255.

```
# dladm show-linkprop -p hoplimit vpn0
LINK         PROPERTY        VALUE          DEFAULT        POSSIBLE
vpn0         hoplimit        64             64             1-255
```

4.2.2 encaplimit

The encaplimit property is only applicable for ipv6 tunnels. This sets the tunnel encapsulation limit, which controls how many levels of nested tunneling a packet may enter before it is dropped. The default value is 4, and valid values range from 0 to 255. A value of 0 indicates no limit is imposed by the tunnel.

```
# dladm show-linkprop -p encaplimit ipv6site2
LINK         PROPERTY        VALUE          DEFAULT        POSSIBLE
ipv6site2    encaplimit      4              4              0-255
```

4.3 Impact on Existing dladm(1M) Subcommands

4.3.1 show-link

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

Example:

```
# dladm show-link
LINK        CLASS     MTU    STATE    OVER
```

```
bge0         phys     1500   up       --
bge1         phys     1500   unknown  --
vpn0         iptun    1480   up       --
ipv6router0  iptun    65515  up       --
ipv6site2    iptun    1452   up       --
vlan3        vlan     1500   up       bge0
```

Note that the CLASS column denotes IP tunnels with a class of iptun.

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 libdladm.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 libdladm.so will be temporary (by setting only the DLADM_OPT_ACTIVE flag as the flags argument to the libdladm.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_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 9.9.

   If the named device (ip.tun0 in this case) did not exist prior to the ifconfig plumb operation, the kernel will implicitly create an IP tunnel link with the appropriate name during

the `open` system call as detailed in section 8.2, allowing `ifconfig` to successfully open the resulting `/dev/net` DLPI device.

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 `libdladm.so` IP tunneling 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 reason[7].

   In this design, `ifconfig` will call `dladm_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`.

6 `libdladm.so` IP Tunneling API

The `libdladm.so` library constitutes a Consolidation-Private API for manipulating networking objects at the data-link layer. This project will add support for IP tunnel links to this API, along with a new `libdliptun.h` header file. 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 ioctls exported by the tunnel driver 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 `flags` argument, which can have two possible flags set:

- `DLADM_OPT_ACTIVE`
- `DLADM_OPT_PERSIST`

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

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

The persistence of configuration information is a private implementation detail of the API, but is done by using the linkid management API introduced by PSARC/2006/499, which results in data being stored in `/etc/dladm/datalink.conf`. When the system boots, `dladm up-iptun` will call into the library to create all tunnels described in the persistent configuration store.

6.1 Tunnel Parameters

Three of the IP tunneling functions provided by the `libdladm.so` API (`dladm_iptun_create()`, `dladm_iptun_modify()`, and `dladm_iptun_getparams()`) take a set of tunnel parameters as input. The `libdliptun.h` header file defines the following structure to hold these parameters:

```
typedef struct iptun_params {
        datalink_id_t   itp_linkid;
        uint_t          itp_flags;
        iptun_type_t    itp_type;
        char            itp_src[NI_MAXHOST];
        char            itp_dst[NI_MAXHOST];
        ipsec_req_t     itp_secinfo;
} iptun_params_t;
```

- `itp_linkid`

  The linkid of the IP tunnel link.

- `itp_flags`

  A set of flags that denote which `iptun_params_t` fields are set in the structure, and for other state associated with the tunnel link. Possible values are:

  ```
  #define ITP_TYPE        0x00000001 /* itp_type is set */
  #define ITP_SRC         0x00000002 /* itp_src is set */
  #define ITP_DST         0x00000004 /* itp_dst is set */
  #define ITP_SECINFO     0x00000008 /* itp_secinfo is set */
  #define ITP_IMPLICIT    0x00000010 /* implicitly created IP tunnel */
  #define ITP_IPSECCONF   0x00000020 /* IPsec policy exists in ipsecconf(1M) */
  ```

- `itp_type`

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

  The following types will be defined[8]:

  - `IPTUN_TYPE_IPV4`
  - `IPTUN_TYPE_IPV6`
  - `IPTUN_TYPE_6TO4`

- `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.

[8] See section 4.1.1 for a description of these tunnel types.

- `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_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 New `libdladm.so` Functions

6.2.1 `dladm_iptun_create()`

```
dladm_status_t dladm_iptun_create(dladm_handle_t handle, const char *name,
    iptun_params_t *params, uint32_t flags);
```

The `dladm_iptun_create()` function will create a new IP tunnel link 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 on input. The function will create a new linkid for the tunnel link and return the new linkid in the `itp_linkid` field of the `params` argument.

The caller will have the option of specifying the name of the data-link for the IP tunnel by passing in a non-NULL `name` argument. If `name` is NULL, the data-link name will be automatically chosen based on the tunnel type according to the following mapping:

Tunnel Type	Tunnel Name
`IPTUN_TYPE_IPV4`	`ip.tun#`
`IPTUN_TYPE_IPV6`	`ip6.tun#`
`IPTUN_TYPE_6TO4`	`ip.6to4tun#`

The PPA number (denoted by `#` above) will be chosen as the lowest available number for that tunnel type. The chosen name can be retrieved by calling `dladm_datalink_id2info()` using the linkid returned.

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

The `itp_type` field 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 `dladm_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 `dladm_iptun_delete()`

```
dladm_status_t dladm_iptun_delete(dladm_handle_t handle, datalink_id_t linkid,
    uint32_t flags);
```

A tunnel will be destroyed by calling `dladm_iptun_delete()`. The tunnel with the given `linkid` will be destroyed, as well as its associated `/dev/net` DLPI node.

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

6.2.3 `dladm_iptun_modify()`

```
dladm_status_t dladm_iptun_modify(dladm_handle_t handle,
    const iptun_params_t *params, uint32_t flags);
```

This function will be used to modify tunnel parameters of the tunnel whose linkid is specified as `itp_linkid` in the supplied `params` structure. The list of attributes is described in section 6.1. The only parameter that is not modifiable is `itp_type`.

6.2.4 `dladm_iptun_getparams()`

```
dladm_status_t dladm_iptun_getparams(dladm_handle_t handle, iptun_params_t *itp,
    uint32_t flags);
```

This function will get the parameters of the tunnel whose linkid is specified as `itp_linkid` in the supplied `params` structure. The function will fill-in the supplied `params` structure, and set `itp_flags` to denote which fields are set in the structure, as well as whether or not the tunnel was implicitly created or has IPsec policy associated with it.

As with other `libdladm.so` functions, the `DLADM_OPT_ACTIVE` and `DLADM_OPT_PERSIST` flags denote whether the caller is interested in the parameters of a tunnel on the running system, or a tunnel defined in the persistent system configuration.

6.2.5 `dladm_iptun_up()`

```
dladm_status_t dladm_iptun_up(dladm_handle_t handle, datalink_id_t linkid);
```

This function will re-create tunnel links that were previously persistently created. If `linkid` is `DATALINK_ALL_LINKID`, then all persistent tunnel links are brought online, otherwise only the tunnel with the matching `linkid` will be brought online.

The main purpose of this function is to create tunnels 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.

6.2.6 `dladm_iptun_down()`

```
dladm_status_t dladm_iptun_down(dladm_handle_t handle, datalink_id_t linkid);
```

This function will temporarily delete all tunnels by calling `dladm_iptun_delete()` with the `DLADM_OPT_ACTIVE` flag on every active tunnel. It is a convenience function for the `down-iptun` `dladm` subcommand.

6.2.7 dladm_iptun_set6to4relay()

```
dladm_status_t dladm_iptun_set6to4relay(dladm_handle_t handle,
    struct in_addr *addr);
```

This function will set the system-wide 6to4 relay router address. The sole consumer of this function is the `6to4relay(1M)` command.

6.2.8 dladm_iptun_get6to4relay()

```
dladm_status_t dladm_iptun_get6to4relay(dladm_handle_t handle,
    struct in_addr *addr);
```

Obtain the currently configured 6to4 relay router address and return the result in the `addr` argument. If `addr` is `INADDR_ANY` upon return, then no 6to4 relay router is currently configured. The sole consumer of this function is the `6to4relay(1M)` command.

7 iptun IP Tunneling Driver

At the heart of this design is a new GLDv3 iptun driver. This driver replaces the tun, atun, and 6to4tun STREAMS modules that comprised the previous IP tunneling implementation. The driver registers tunnel links with GLDv3 using the MAC driver API, and implements IP communication below it by interacting with the ip module via function calls. In this way, the iptun driver is both a MAC driver and an IP client, as illustrated in Figure 2.

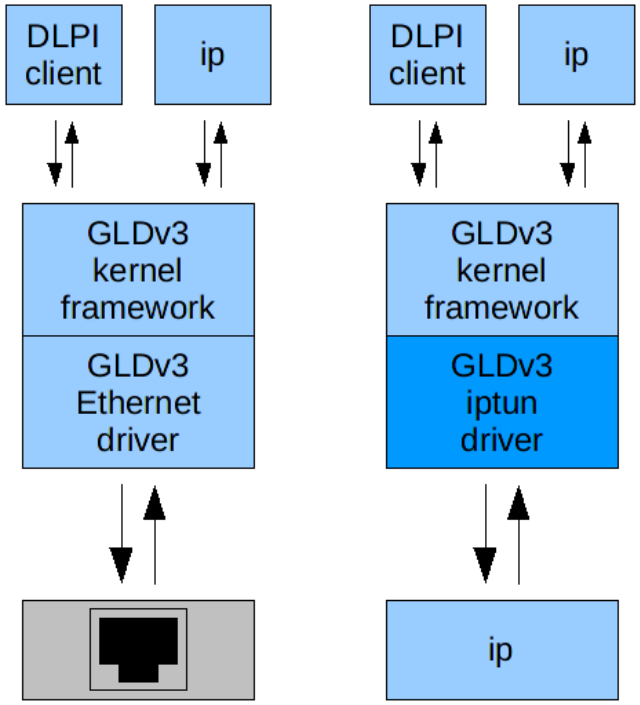

Figure 2: At left, the model of a typical Nemo Ethernet driver. The driver registers with the GLDv3 framework above and interacts with the media hardware below. At right, how the tunnel driver uses this same model. The tunnel driver also registers with the GLDv3 framework above, but instead of interacting with media hardware, it interacts with the ip module below it using function calls.

The libdladm.so API functions configure IP tunnel links by issuing ioctls to the /dev/dld control device, which dispatches them to the iptun driver.

7.1 iptun ioctls

The iptun driver supports the set of Project-Private ioctls described in this section. The ioctls and the data structures they use are defined in a new <inet/iptun.h> header file which is intended to only be consumed by libdladm source code.

Note that the previous IP tunneling implementation used a set of socket ioctls documented in tun(7M) (SIOCSTUNPARAM and SIOCGTUNPARAM). These ioctls were used by ifconfig (indirectly

through libinetcfg). Although these were documented as "Evolving" (i.e. Committed according to the ARC interface taxonomy), they were in practice implementation details and not used beyond the internal implementation of libinetcfg. Support for these ioctls don't make sense with the new implementation, and they will be removed by this project.

7.1.1 IPTUN_CREATE

This ioctl creates a tunnel and registers it with the mac module using mac_register(). If the mac_register() operation is successful, a /dev/net node will be created for the tunnel using dls_devnet_create().

This ioctl will fail if a tunnel with the given linkid 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 sys_iptun_config privilege.

7.1.2 IPTUN_DELETE

This ioctl takes the linkid of the tunnel to be deleted. The driver deletes the tunnel and unregisters it with the mac module using mac_unregister(). The data-link associated with the tunnel is also deleted.

This ioctl will fail if there is no tunnel with the given linkid, or if the named tunnel's DLPI node has an open reference (a plumbed IP interface for example).

This ioctl requires the sys_iptun_config privilege.

7.1.3 IPTUN_MODIFY

This ioctl modifies an existing tunnel's source, destination, or IPsec policy. It requires the sys_iptun_config privilege.

7.1.4 IPTUN_INFO

This unprivileged ioctl returns configuration information for a single IP tunnel.

7.1.5 IPTUN_SET_6TO4RELAY

This replaces the existing undocumented SIOCS6TO4TUNRRADDR socket ioctl. It takes an ipaddr_t as its sole argument, and sets the 6to4 relay router address. The only consumer of this ioctl is the dladm_iptun_set6to4relay() function. The SIOCS6TO4TUNRRADDR ioctl will be removed.

This ioctl requires the sys_iptun_config privilege.

7.1.6 IPTUN_GET_6TO4RELAY

This replaces the existing undocumented SIOCG6TO4TUNRRADDR socket ioctl. It takes an ipaddr_t as its sole argument, and returns the current address of the 6to4 relay router in that argu-

ment. The only consumer of this ioctl is the `dladm_iptun_get6to4relay()` function. The `SIOCG6TO4TUNRRADDR` ioctl will be removed.

This ioctl does 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 2 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 `dld` and `dls` modules[9].

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 addresses are set on a given tunnel, the tunnel driver will call the appropriate `ip_proto_bind_*()` function 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. 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.

Aside from the periodic calls to `ire_route_lookup()` to obtain path MTU information for tunnel destinations, the `iptun` driver will also adjust its tunnel MTUs based on ICMP fragmentation needed and ICMPv6 packet too big messages received as a result of sending tunneled packets that may be larger than some segment of the path to the destination. This ICMP processing and resulting MTU adjustment remains unchanged from how the `tun` module previously behaved. The main difference being that the `iptun` module, being a GLDv3 driver, will notify the GLDv3 framework of MTU changes by calling `mac_maxsdu_update()`.

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:

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

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

- 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.

Note that the `ip` module still requires an `ip` stream to be set in each tunnel's `connp->conn_wq` and `connp->conn_rq` (used for flow-control when an underlying link uses STREAMS. As such, the `iptun` module needs some STREAMS queue to the `ip` module for this purpose.

The solution is to introduce a dummy driver called `iptundummy` (akin to the dummy `udp` driver and its dummy friends used for similar purposes) that is really `ip` in disguise. The `iptun` module opens the `iptundummy` module using `ldi_open_by_name()` to create one STREAMS queue per stack for the purpose of fulfilling `ip`'s addiction to STREAMS. It is expected that the IP Datapath Refactoring project will subsequently cure this addiction and remove the `iptundummy` driver as it will no longer be needed.

Below is a breakdown of each `ip` module function that the `iptun` module uses. Note that none of the following require that the caller be holding any locks within the `ip` module or be using an `ip`-specific synchronization mechanism such as an `ipsq`.

- `ipcl_conn_create()`

  This is used to create the `conn_t` associated with a given `iptun_t` when the tunnel is created.

- `ip_quiesce_conn()`

  When a tunnel is deleted, this is called to quiesce activity on the tunnel's `conn_t` to ensure its safe destruction by calling `CONN_DEC_REF()` while holding the last reference.

- `ip_proto_bind_connected_[v4|v6]()`

  When both source and destination addresses have been configured on an IPv4 or IPv6 tunnel, `iptun` binds to them using these functions. The `udp`, `tcp`, and raw socket implementations use these functions in an identical way. This inserts the tunnel's `conn_t` in the IP tunnel fanout within `ip`.

- `ip_proto_bind_laddr_v4()`

  This is used to bind to the source address of a 6to4 tunnel. This inserts the 6to4 tunnel's `conn_t` in the IP tunnel fanout within `ip`.

- `ire_route_lookup[_v6]()`

  The `iptun` module regularly obtains path MTU information for tunnel destinations in order to dynamically calculate tunnel MTUs for tunnels that have fixed destination addresses. The IRE for the destination is obtained, and its `ire_max_frag` is used to calculate the tunnel MTU. The reference to the IRE is released using `ire_refrele()`.

- `ip_output[_v6]()`

  This is used to transmit tunneled IP packets. It is called indirectly through the `conn_send` pointer set within the `iptun_t`'s `conn_t` pointer to transmit data packets, and is also used directly to transmit ICMP errors when needed.

7.3 Interaction With IPsec

The overall model for implementing IPsec tunnels is similar as today in this design. IPsec tunnel policy is stored within the tunnel instance data structure, and policy is applied by the `iptun` module in-line with data processing.

When a tunnel link is created, the `iptun` module tries to determine if there is existing IPsec policy (previously configured and loaded into the kernel using `ipsecconf(1M)`) for this tunnel by calling the `get_tunnel_policy()` function which consults the kernel security policy database. If yes, it immediately takes effect by virtue of the `iptun` module storing the resulting policy pointer in its internal tunnel data structure.

If `ipsecconf(1M)` is invoked to create tunnel policy for an existing IP tunnel link, the kernel policy code in `spdsock` calls into the `iptun` module to have it automatically store that policy.

For backward compatibility with existing configuration that sets IPsec tunnel policy on an IP interface using `ifconfig(1M)`, the `iptun` module allows an `ipsec_req_t` to be included in the data passed in to the `IPTUN_MODIFY` ioctl, and the `libdladm dladm_iptun_modify()` function thus also allows this. `ifconfig(1M)` uses `dladm_iptun_modify()` to set "simple" IPsec policy using this mechanism on the tunnel link.

7.4 Tunnel Link Statistics

The `iptun` driver's `mi_getstat()` callback will support the generic `MAC_STAT_IERRORS`, `MAC_STAT_OERRORS`, `MAC_STAT_RBYTES`, `MAC_STAT_IPACKETS`, `MAC_STAT_OBYTES`, and `MAC_STAT_OPACKETS` statistics. This is backward compatible with the set of kstats that the `tun` module being replaced implements.

8 Changes to the GLDv3 Framework

The following sections detail all changes required in the GLDv3 framework to support the IP tunnel driver. It is assumed that the reader is familiar with GLDv3 and its interfaces.

8.1 New MAC Type Plugins

8.1.1 `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:

- `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	`sizeof (struct ip)`
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	`ipv4hd_ttl` if supplied, or 255
Protocol	`sap` argument
Header Checksum	0 (filled in by lower `ip` instance)
Source Address	`saddr` argument
Destination Address	`daddr` 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`.

8.1.2 `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	`sap` argument or `IPPROTO_DSTOPTS`
Hop Limit	`ipv6hd_hoplimit` if supplied, or 255
Source Address	`saddr` argument
Destination Address	`daddr` 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`.

8.1.3 DL_6TO4 Plugin

This plugin will implement MAC type specific operations for 6to4 tunnels. The MAC type for this plugin will be DL_6TO4. 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 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 {
          HD6TO4_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 HD6TO4_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	`sizeof (struct ip)`
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	`ipv4hd_ttl` if supplied, or 255
Protocol	`sap` argument
Header Checksum	0 (filled in by lower `ip` instance)
Source Address	`saddr` argument
Destination Address	`daddr` 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`.

8.2 Implicit IP Tunnel Creation

This design includes a mechanism for implicit tunnel creation in order to maintain backward compatibility with previous versions of Solaris in which `ifconfig` is responsible for creating IP tunnel links. The general idea is that when an application such as `ifconfig` opens a device in `/dev/net` which doesn't exist, and which has a special IP tunnel name, the `devnet` implementation will create an IP tunnel as part of the `open()` operation and allows the `open()` to succeed.

The sequence of events within `dls_mgmt.c` when opening a `/dev/net` node is as follows:

1. `dls_devnet_open()` is called by the `/dev/net` subsystem, which in turns calls `dls_devnet_hold_by_name()`.
2. If the named `devnet` node exists, then `dls_devnet_hold_by_name()` calls `dls_devnet_hold()` on that node and returns success.
3. If, however, the link does not exist, then `dls_devnet_hold_by_name()` verifies if the name of the `devnet` node being opened matches one of three special IP tunnel link names. If so, it creates a linkid by calling `dls_mgmt_create()`, then creates an IP tunnel link by calling the `iptun` driver's `iptun_create()` function. The type of the tunnel created depends on the link name according to the following table:

Tunnel Name	Resulting Tunnel Type
`ip.tun#`	`IPTUN_TYPE_IPV4`
`ip6.tun#`	`IPTUN_TYPE_IPV6`
`ip.6to4tun#`	`IPTUN_TYPE_6TO4`

   Note that the `iptun_create()` function will create a `devnet` node for the tunnel link by calling `dls_devnet_create()`.

4. After the IP tunnel link and associated `devnet` node have been created in the previous step, `dls_devnet_hold_by_name()` calls `dls_devnet_hold()` on the newly created node and returns success. The `open()` then succeeds.

When the last reference to the `devnet` node created implicitly as described above is released in `dls_devnet_rele()` (as a result of the last process calling `close()` on the associated file descriptor), `dls_devnet_rele()` automatically calls the `iptun` driver's `iptun_delete()` function to delete the IP tunnel link.

8.3 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.

8.4 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,NONUD,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,NONUD,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 4213] 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` ioctls[11] and snooping the tunnel type, the tunnel source, and the tunnel destination from the ioctl's 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 9.1, 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 `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.

[11] Remember, `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.

9 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.

9.1 DL_INFO_ACK

The `dl_info_ack_t` structure contained in `DL_INFO_ACK` primitives passed up by IP tunnel DLPI devices will contain the following notable field values:

- `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_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_6TO4`

  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 8.4 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_6TO4` 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_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.

9.2 DL_BIND_REQ

Today 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.

As described in section 8.1, however, the SAP spaces for the various IP tunnel links is not comprised of Ethertype values, but of IP protocol values (e.g. IPPROTO_ENCAP and IPPROTO_IPV6). DLPI consumers such as the ip module who wish to bind to a specific IP SAP will need to use these values with DL_BIND_REQ.

The ip module's ip_m_tbl media table will be augmented to contain MAC-Type specific IPv4 and IPv6 SAP values so that it can bind using the correct value regardless of MAC-Type. Entries for DL_IPV4, DL_IPV6, and DL_6TO4 will be included in the media table.

9.3 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.

9.4 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.

9.5 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.

9.6 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.

9.7 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.

9.8 DL_NOTE_SDU_SIZE

The tunnel driver will notify the Nemo framework of link MTU changes resulting from tunnel destination path-MTU fluctuations using `mac_maxsdu_update()`, which will in turn cause the generation of `DL_NOTE_SDU_SIZE` DLPI notifications.

9.9 Impact on libdlpi.so

The existing tunnel IP interface configuration mechanism done by `ifconfig` hinges on an obscure and undocumented interface name parsing "feature" implemented in `libdlpi.so`'s `dlpi_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_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_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_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.

9.10 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_6TO4` depending on the tunnel type. The first two are already defined in `<sys/dlpi.h>` to represent IPV4 and IPv6 tunnel links. `DL_6TO4` will be added to this header file under the private media types already defined therein.

10 IP Tunnel Management in Non-Global Zones

As a side-effect of the IP Instances project (PSARC/2006/366 Stack instances: Exclusive IP stack per zone), it is possible today to create an IP tunnel from within a non-global zone. This is due to zones with exclusive IP stacks having the `sys_ip_config` privilege, which allows them to plumb IP interfaces. This is the only privilege currently required to have the "`ifconfig ip.tun0 plumb`" command complete successfully.

Because one of this project's goals is backward compatibility, that same "`ifconfig ip.tun0 plumb`" command must also succeed in this design. PSARC/2009/410 defines the architecture necessary to allow for datalink management in non-global zones in general. This design fills in the class-specific architecture to make this work for IP tunnel links.

This design introduces a new `sys_iptun_config` privilege. A process with this privilege is allowed to create, modify, and delete IP tunnel links. This new privilege is a subset of the existing `sys_dl_config` privilege, meaning that a process that has the `sys_dl_config` privilege is allowed to perform the operations on IP tunnel links.

The `sys_iptun_config` privilege will be delegated to exclusive-stack non-global zones, but `sys_dl_config` will not. This will result in such zones only being able to modify IP tunnel links, but no other kinds of links.

As described in PSARC/2009/410, class-specific kernel modules must participate in the network stack shutdown process in order to automatically delete links that were created when the zone was running. The `iptun` module will do this for IP tunnel links. This involves inserting a `NS_IPTUN` netstack entry in `netstack_t` as shown here:

```
/*
 * One for each module which uses netstack support.
 * Used in netstack_register().
 *
 * The order of these is important for some modules both for
 * the creation (which done in ascending order) and destruction (which is
 * done ine in decending order).
 */
#define NS_ALL          -1      /* Match all */
#define NS_DLS          0
#define NS_IPTUN        1
#define NS_STR          2       /* autopush list etc */
#define NS_HOOK         3
#define NS_NETI         4
#define NS_ARP          5
#define NS_IP           6
...
```

During stack shutdown, `iptun_stack_shutdown()` will automatically delete IP tunnels that were

created in the zone for the given stack.

11 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 policy 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 also 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:default` and `svc:/network/ipsec/policy:default`. The dependency on `svc:/network/physical:default` will be a `require_all` dependency specifically for the `default` instance, as the `nwam` instance of the `svc:/network/physical` service does not handle IP tunnels configured at boot time.

12 Impact on STREAMS Module 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 `dladm autopush` link property. This is currently not possible in Solaris.

13 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 has been deprecated by [RFC 4213], and will be EOL'ed as part of this project. [RFC 4213] 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.

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 was obsoleted by [RFC 4291], and IPv4-compatible addresses were deprecated in the new RFC:

> 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.

14 Modifications to libpcap

To make applications like `wireshark` open and decode IP tunnel link devices, some modifications are needed within the `libpcap` library. Specifically, a mapping from the DLPI MAC-types used by IP tunnels (`DL_IPV4`, `DL_IPV6`, and `DL_6TO4`) to a data-link type used by `libpcap` consumers is needed.

`libpcap` consumers determine how to decode packets read from a `pcap` device based on the data-link type of the device, as returned by `libpcap`'s `pcap_datalink()` function. These data-link types are actually `DLT_*` values modeled after the Berkeley Packet Filter's `bpf.h` data-link types.

Within the `libpcap` implementation, there is a mapping from `DLPI` MAC-types to BPF data-link types. A data-link type suitable for IP tunnels already exists (`DLT_RAW`[12]), but a mapping from the IP tunneling DLPI MAC-types to `DLT_RAW` is needed.

The `libpcap` library is in the SFW consolidation, and this change will be done as a separate integration to SFW. The existing `libpcap` patch in SFW will be modified to add this mapping, and the change will be pushed upstream to the `tcpdump` community.

[12] `DLT_RAW` indicates that packets read from the device begin with an IP header

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 -T ipv4 -s 66.1.2.3 -d 192.4.5.6 mytunnel0
```

This will create an IPv4 tunnel link between `66.1.2.3` and `192.4.5.6`. This operation will result in the creation of a DLPI device named `/dev/net/mytunnel0`. Note that the operation above is persistent, meaning that the `mytunnel0` link will be reinstantiated when the system reboots.

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. Persistence is achieved just as with any other IP interface; by placing the desired ifconfig commands (in this case "`10.1.0.1 11.2.0.2`") into `/etc/hostname.mytunnel0`.

An IPv6 interface can be created over this same tunnel link:

```
# ifconfig mytunnel0 inet6 plumb up
# ifconfig mytunnel0 inet6 addif 3000::1 3000::2 up
```

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 link. For example:

```
# snoop -d mytunnel0
```

The `snoop` command will open the DLPI device `/dev/net/mytunnel0` as it would with any other network interface.

B Future Work Related to IP Tunneling

Enumerated here are potential future enhancements and features related to IP tunneling.

B.1 Tunnel Link State Enhancements

The link state of an IP tunnel link is currently a function of whether or not tunnel addresses are configured. Specifically, if the tunnel source and tunnel destination addresses (or tunnel source in the case of 6to4) are successfully configured, then the link state is up. If the tunnel source address is subsequently brought down or removed, the tunnel is not aware of this and the link state remains up even though the tunnel is inoperable. Similarly, if a route to the destination is removed and the destination is no longer reachable, the link state remains up. This is also true for today's `tun`

implementation. Once the tunnel has successfully bound to the tunnel addresses, it does not keep track of the state of those addresses nor the reachability of the destination.

A potential future enhancement would be to add heuristics in the kernel to keep track of the status of IP tunnel source addresses, and perhaps also of the reachability of tunnel destination addresses, in order to reflect this in the link state of tunnel links.

B.2 Tunnel Creation Dependency Enhancements

The tunneling configuration code currently requires that tunnel source addresses exist when they're configured. For example, if one configures IP address A as a tunnel source address, that IP address must already be plumbed for the configuration to succeed. This is an existing limitation in the current implementation which is not addressed by this design. As a result of this limitation, IP tunnels must be created after the IP addresses that they depend on have been plumbed.

The reason for this limitation is an implementation detail. The `iptun` module attempts to bind to the tunnel addresses in the context of their configuration, and if the bind fails, the configuration fails. The `ip` module does not have a kind of bind that allows a client to bind to a non-existent address.

This could potentially be addressed in a number of different ways. One would be to implement a special kind of bind in `ip` that allows a client to bind to a non-existent address. When that address is plumbed, `ip` could have logic to actually bind pending clients.

Another method would be to have the `iptun` module keep track of which IP addresses exist, and delay binding until it detects that IP addresses are available.

B.3 Point-To-Multipoint Architecture and 6to4

6to4 tunnel links are essentially point-to-multipoint links, yet there is no point-to-multipoint architecture in the `ip` module. Because there is no fixed link-layer destination for a point-to-multipoint link and there is no external resolver that can obtain a destination link-layer address for a given IP destination, IP fast-path header generation cannot function as with other types of links [13]. The current 6to4 implementation functions by having an empty destination link-layer address used for fast-path header generation, and the `iptun` module fills in the destination on a packet-by-packet basis.

In order to fully utilize the fast-path functionality in `ip`, a point-to-multipoint architecture is needed in the stack. One such architecture could involve modifying the `ip` module to pass down the IP next hop address in its fast-path ioctls instead of the link-layer destination for special point-to-multipoint links. The GLDv3 modules would need to allow for a `DL_UNITDATA_REQ` message containing a `dl_dest_addr_length` different from the actual link-layer address length (6to4 link-layer addresses are 4-byte IPv4 addresses, and IPv6 addresses of IP interface above the link are 16-bytes).

GLDv3 functions such as `dls_header()`, `mac_header()`, and `mtops_header()` plugin callbacks would all need to be modified to accept addresses of lengths different from the registered link-layer address length for the link in question.

The `mac_6to4` plugin (or other plugins for point-to-multipoint media) could then be modified to

[13] An IP fast-path link-layer header is obtained by sending down a `DLIOCHDRINFO` ioctl with an embedded `DL_UNITDATA_REQ` message. The `DL_UNITDATA_REQ` message contains the destination link-layer address that IP wishes to be used to generate the link-layer header

generate proper link-layer headers based on the passed-in network-layer address "hint".

To make this scheme function with relay routers, the configuration of relay routers would require the addition of a special default IPv6 route with an IPv6 next-hop corresponding to the relay router's IPv4 address. For example, 6to4-related routes on a 6to4 router with configuration for a relay of 10.11.12.13 would look like this:

```
Destination       Gateway               Interface
2002::/16         2002:0102:0304::1     ip.6to4tun0
default           2002:0a0b:0c0d::1     ip.6to4tun0
```

The first route is the interface route for the `ip.6to4tun0` interface. The 6to4 link's IPv4 source is 1.2.3.4. The second route is an off-link route with a gateway corresponding to the relay router's IPv4 address. Use of routes in this way reinforces the idea of a 6to4 link on which all 2002::/16 addresses are conceptually on-link, and all other addresses are off-link.

Using the default route's gateway address in fast-path message requests would result in fast-path headers containing the relay router's IPv4 destination address in the outer IPv4 header's destination field. The `iptun` module would then not need to be involved at all with the configuration of a relay router, as the `6to4relay` command could do everything it needs by manipulating the system's routing table.

Regardless of whether this proposed architecture is implemented, the lack of such architecture today prevents 6to4 from being fully-integrated `ip` fast-path, and this should be addressed in the future.

B.4 GRE and Other Tunneling Mechanisms

The tunneling architecture being introduced by this project lends itself well to the future implementation of additional tunneling mechanisms. Specifically, the use of GLDv3 and MAC-Type plugins defines clear interfaces where additional types of tunnels can be implemented.

The GRE tunneling mechanism described in [RFC 2784] is one such mechanism that is missing from OpenSolaris and is oft-requested. Such an implementation would involve implementing a new MAC-Type plugin and support for a new GRE tunnel type in `dladm`, `libdladm`, and `iptun`.

References

[RFC 3056] "Connection of IPv6 Domains via IPv4 Clouds," B. Carpenter, K. Moore, February 2001.

[RFC 4213] "Basic Transition Mechanisms for IPv6 Hosts and Routers," E. Nordmark, R. Gilligan, October 2005.

[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 3513] "Internet Protocol Version 6 (IPv6) Addressing Architecture," R. Hinden, S. Deering, April 2003.

[RFC 4291] "IP Version 6 Addressing Architecture," R. Hinden, S. Deering, February 2006.

[RFC 2784] "Generic Routing Encapsulation (GRE)," D. Farinacci, T. Li, S. Hanks, D. Meyer, P. Traina, March 2000.