IP Observability Device Design Specification

Phil Kirk

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

Solaris Networking
Sun Microsystems, Inc.

Revision 1.1
October 25, 2007

Contents

1 Introduction

The ability to observe packets flowing through a network device is crucial in diagnosing many networking problems. While this is possible at the data link layer[1], it is not possible at the IP layer. Providing observability at the IP layer is one of the main network interface requirements documented in the Clearview Overview[2].

This inability to observe packets at the IP layer means that there is currently no way in Solaris to observe locally destined packets as they flow through the system using common packet sniffing tools such as `snoop` and `ethereal`. This is due to `ip` recognizing that packets are destined for local delivery, and short-circuiting transmission of the packet to the network interface. Locally destined packets consist of traffic destined for localhost, as well as traffic between IP addresses hosted on the system.

With the introduction of zones, this lack of observability has become a major problem as there is no way to observe inter-zone traffic. In fact, right now there is no way of observing IP traffic at all from within a zone. Already, several major customers have raised this issue as a significant concern. Prior to zones, this lack of observability had also proved problematic when trying to resolve several high severity escalations. By providing the ability to observe traffic flowing through `ip`, this problem will be resolved.

In addition, there is no way of observing traffic flowing over an IPMP group. Specifically, an administrator can observe packets flowing through the individual interfaces that make up a group, but there is currently no way of viewing the group as a whole. By providing IP observability, along with the IPMP rearchitecture,[3]this long standing problem will be resolved.

In the future should a socket layer loopback become available then this design would also support observation of this traffic in the same way that loopback at the TCP will be handled (see section 3.2 for detail).

2 Requirements Analysis

From the description in Section 1, the requirements for IP layer observability are:

1. Must be able to view IP traffic sent to an IP address local to the system. This includes all local addresses, including those hosted on non-loopback interfaces.
2. From the global zone, it must be possible to view all IP traffic on the machine. This means all loopback IP traffic, traffic from remote machines, traffic sent from this machine, forwarded traffic and inter/intra-zone traffic.
3. From a local zone, it must be possible to view all IP traffic sent to or from this zone, including traffic local to the zone. It must not be possible to see any other traffic.
4. Must allow an administrator to use the same tools he currently uses to observe network traffic, e.g. `snoop`. We must also provide consistency with other Unix variants. [4]

In the above requirements IP traffic means IPv4 and IPv6 traffic only. Other protocol traffic for example ARP traffic is not captured.

[1] Even for the data link layer this is not true for tunnels. That problem is being addressed by the IP Tunnel Device, part of the Clearview project
[2] http://clearview.east/docs/overview.txt
[3] http://opensolaris.org/os/community/networking/ipmp-highlevel-design.pdf
[4] Providing a consistent administration experience is one of the key Clearview goals

The solution arrived at will introduce a new IP layer DLPI style-1 device for each IP interface on the system. These devices will provide access to all packets with IP addresses local to the system which includes inter-zone and intra-zone traffic. In addition to these devices we will also provide a `/dev/lo0` device that will provide the same behaviour as other Unix variants.

These IP layer devices will exist in a different directory, `/dev/ipnet` instead of `/dev`, avoiding naming conflicts with DLPI data link devices. The end-result will be a consistent and familiar experience: just as now an administrator runs `snoop -d bge0` to capture packets flowing over the `bge0` data link, he will be able to run `snoop -I bge0`[5] to view all IP traffic associated with `bge0`.

The requirement that an administrator must be able to use the same tools he currently uses to observe traffic at different layers, means that a solution implemented entirely using DTrace is not appropriate. Specifically, an administrator must not need to use different tools to observe traffic at different layers. With a solution based entirely on DTrace this would not be possible. However, using DTrace to implement part of the solution is potentially possible and is discussed in Section 3.2.

Since the devices in `/dev/ipnet` will provide access to all packets local to the system, including inter-zone and intra-zone traffic, the design satisfies requirements 1 and 2, namely:

- Must be able to view IP traffic sent to an IP address local to the system. This includes all local addresses, including those hosted on non-loopback interfaces.
- From the global zone, it must be possible to view all IP traffic on the machine. This means all loopback IP traffic, traffic from remote machines, traffic sent from this machine, forwarded traffic and inter/intra-zone traffic.

Further by making `/dev/ipnet` accessible from within a zone, and restricting the IP traffic received to only traffic sent to or from the zone, the design satisfies requirement 3, namely:

- From a local zone, it must be possible to view all IP traffic sent to or from this zone, including traffic local to this zone. It must not be possible to see any other traffic.

As the devices will provide a DLPI interface, this means that common network monitoring tools such as `snoop` can still be used to observe this layer. By providing a `/dev/lo0` device we will also provide consistency with other Unix variants. This satisfies requirement 4, namely:

- Must allow an administrator to use the same tools he currently uses to observe network traffic, e.g. `snoop`. We must also provide consistency with other Unix variants.

[5] The exact command line invocation to tell `snoop` to use the `ipnet` devices is still being considered

3 Architectural Design

3.1 /dev/ipnet

This project will introduce a `/dev/ipnet` name space, which will contain devices providing access to packets passing through `ip`. For every IP interface on on the system, there will be a node created in `/dev/ipnet`. As an example, supposing `ifconfig -a` returns:

```
# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
lo0:1: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        zone Z1
        inet 127.0.0.1 netmask ff000000
lo0:2: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        zone Z2
        inet 127.0.0.1 netmask ff000000
bge0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 192.168.0.1 netmask ffffff00 broadcast 192.168.0.255
        ether 8:0:20:f0:5a:8
eri0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 3
        inet 10.0.0.1 netmask ff000000 broadcast 10.255.255.255
        ether 8:0:20:f0:5a:8
eri0:1: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 3
        zone Z1
        inet 10.0.0.2 netmask ff000000 broadcast 10.255.255.255
eri0:2: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 3
        zone Z2
        inet 10.0.0.3 netmask ff000000 broadcast 10.255.255.255
```

With this configuration, the following DLPI style-1 devices will be created in `/dev/ipnet`:

```
/dev/ipnet/lo0
          /bge0
          /eri0
```

Opening these devices will provide access to all IP packets with addresses associated with the interface. This includes both IPv4 and IPv6 traffic, and addresses hosted on logical interfaces. For this reason, there is no `/dev/ipnet/eri0:1`; instead opening `/dev/ipnet/eri0` will provide all traffic that is destined for, or originating from, any address hosted on `eri0`. If administrators want to see packets from a particular address, then they will be able to use filters, just as they can now when running `snoop` on a data link.

To try to avoid any confusion for users observing traffic on these devices, we initially tried to mirror the semantics of those currently defined by the data link devices. However this led to ambiguities in the semantics for the `/dev/ipnet` devices. Given this, the semantics for the `/dev/ipnet` devices will be:

- A packet will be passed to a consumer if:
  - The packet passes `ip`'s checks for validity **AND** either (1) or (2) are true.

1. The source or destination address of the packet is one of the addresses assigned to the interface[6] **OR**,
2. `DL_PROMISC_PHYS` is enabled and the interface was used for input or output of the packet to or from the link-layer **AND** the consumer is in the global zone or in a non-global zone to which the packet is destined.

The following table shows how received packets will be processed.

[c]

Rule	Received
(2) & (1)	Yes
(2) & !(1)	No unless promiscuous mode is set
!(2) & (1)	Yes

Table 1: Processing of received packets

To determine which interface was used for input or output the interface index will be tracked.

Using the `ifconfig` output at the start of section 3.1, summarized in Table 2, Table 3 shows what packets a user would receive when running `snoop` on a device in `/dev/ipnet` from the global zone, and from within a zone. In the table, G represents the global zone and Z1 and Z2 represent the local zones. The table is based on the rules shown in table 1.

Interface	Address	Zone
`lo0`	127.0.0.1	G
`lo0:1`	127.0.0.1	Z1
`lo0:2`	127.0.0.1	Z2
`bge0`	192.168.0.1	G
`eri0`	10.0.0.1	G
`eri0:1`	10.0.0.2	Z1
`eri0:2`	10.0.0.3	Z2

Table 2: System IP configuration

Two interesting cases for reception of packets are broadcast and multicast. In the multicast case, the ip module does not by default receive all multicast traffic, it will only receive traffic for those applications that have bound to specific multicast addresses. To ensure all multicast traffic is seen when a user opens an ipnet device and issues a DL_ENABMULTI_REQ, the ipnet driver will send a DL_ENABMULTI_REQ down to the underlying ill_t associated with the ipnet device that is being opened. Code already exists in ip to achieve this so the same code will be used by the ipnet device. When the ipnet is closed, assuming no other user has also issued a DL_PROMISC_MULTI request, the ipnet device will send a DL_DISABMULTI_REQ to turn promiscuous mode off.

Enabling promiscuous mode will have a slight overall performance impact, and therefore a user with the privileges needed to use ipnet devices from inside a zone can potentially impact the performance outside their zone. However, this is already true for multicast applications operating in a zone today, and therefore is not a new concern. Broadcast traffic will be seen as expected.

[6] The addresses assigned to the interface are those listed by an invocation of `ifconfig -a` in the zone of the consumer. From the global zone, this is all of the addresses assigned to the interface. From a non-global zone, this is only the addresses assigned to that non-global zone.

[c]

Source ->Destination	Direction	Interface[1]	Zone	`snoop -P`[2]	`snoop`
127.0.0.1 ->127.0.0.1(G)	Outbound on `lo0`	`/dev/ipnet/lo0`	G	Y	Y
127.0.0.1 ->127.0.0.1(Z1)	Outbound on `lo0`	`/dev/ipnet/lo0`	Z1	Y	Y[3]
10.0.0.2 ->10.0.0.3	Inbound on `eri0`	`/dev/ipnet/eri0`	G	Y	Y
10.0.0.1 ->10.0.0.2	Inbound on `eri0`	`/dev/ipnet/eri0`	Z2	N	N
192.168.0.1 ->192.168.0.2	Outbound on `bge0`	`/dev/ipnet/eri0`	G	N	N
192.168.0.1 ->192.168.0.2	Outbound on `bge0`	`/dev/ipnet/bge0`	G	Y	Y
192.168.0.4 ->10.0.0.10	Inbound on `bge0`	`/dev/ipnet/bge0`	G	N	Y
192.168.0.4 ->10.0.0.10	Outbound on `eri0`	`/dev/ipnet/bge0`	G	N	N
192.168.0.4 ->10.0.0.10	Outbound on `eri0`	`/dev/ipnet/eri0`	G	N	Y
10.0.0.10 ->192.168.0.1	Inbound on `eri0`	`/dev/ipnet/eri0`	G	N	Y
10.0.0.10 ->192.168.0.1	Inbound on `eri0`	`/dev/ipnet/bge0`	G	Y	Y

[1] The interface `snoop` is been run against.
[2] Running `snoop -P` means that `DL_PROMISC_PHYS` is not set.
[3] A zone will only see its own inter-zone and intra-zone traffic.

Table 3: Packet visibility

3.2 The Observability Device (`ipnet`)

We will introduce an observability device named `ipnet`, responsible for the creation and deletion of nodes in `/dev/ipnet`, and for passing up IP traffic to listeners using these device nodes. No actual control node for `ipnet` will be created, devices in `/dev/ipnet` are created dynamically. The devices created in `/dev/ipnet` will include the proposed `ipmp` device described in the IPMP Rearchitecture document. Monitoring an `ipmp` device in `/dev/ipnet` will allow one to see all IP packets sent to and received from the IPMP group as a whole.

The `ipnet` device will need the following to perform its tasks:

- To be able to query `ip` for the current list of physical interfaces on the system.
- To receive notification of interface events, such as plumbing a new physical interface.
- To receive packets sent from and received by `ip`.

The Packet Filtering Hooks team is already in the process of defining similar interfaces [7] and we did discuss the possibility of using the Packet Filtering Hooks work to receive packets sent from and received by `ip`. However, during the discussions it became clear that there were potential problems making the Hooks Framework generic at this time so we will implement our own specific hooks in `ip`. Although we will not be using the Packet Filtering Hook framework for receiving packet from `ip` we still plan on using their proposed `netinfo (9f)` interfaces for interface events.

Using DTrace as a provider of the data required by the DLPI device was considered but adds more complexity than adding simple callback functions int `ip`. In the future, using DTrace may happen, and potentially provides some exciting possibilities such as allowing administrators to choose what data they wish to receive. Nonetheless, the proposed semantics for DLPI nodes in `/dev/ipnet` are not tied to the underlying data provider architecture, and could be changed to be based atop DTrace providers in the future.

The introduction of TCP fusion[8] means that local TCP is handled entirely within `tcp` and IP never

[7] http://sac.sfbay/arc/PSARC/2005/334/
[8] See bug 4821256.

sees this traffic. This means that even with the ability to observe IP layer traffic these connections would not be visible. This problem will be resolved so that when a user is observing IP traffic, `tcp` will no longer short-circuit the connection and packets will be passed to `ip`.

3.3 `/dev/lo0`

On other Unix variants loopback traffic is observed by opening `/dev/lo0`. Loopback traffic in this case includes **all** traffic whose source and destination IP addresses are local to the system - `127.0.0.1` or otherwise. Although the new `ipnet` devices provide a more correct model for viewing traffic we also recognise the need for consistency with other Unix variants. This is part of requirement 4 in the Requirements Analysis section. For this reason the project will also implement a `/dev/lo0` device. The semantics for this device will mirror other Unix variants and will provide access to all local traffic.

3.4 `DL_IPNET`

For the `ipnet` device we will introduce a new `dl_mac_type`, `DL_IPNET`. Initially we had planned on using the same data link type as the IP Tunnel device. Architecturally, this seemed to be a good design choice; both devices exist at the IP layer.

However, these devices will be very different in nature. The `ipnet` device will differ from a traditional DLPI device in that it:

- Will not allow transmission of packets
- Will not have a link state
- Will not have a single address format – both IPv4 and IPv6 will be represented

The `ipnet` device exists to provide access to packets at the IP layer. It will only respond to the subset of DLPI requests (see section 5.5) necessary for an application such as `snoop` to open it and receive packets from it. For these reasons, it makes sense to introduce a new mac type to represent the devices in `/dev/ipnet`.

The SAP values supported by the `DL_IPNET` will be `IP_DL_SAP`, `IP6_DL_SAP` and 0. As the device only provides access to IP packets supporting other types is not necessary. We will support 0 so that applications such as `snoop` which bind to SAP 0 will still work.

3.5 Handling IP Ancillary Data

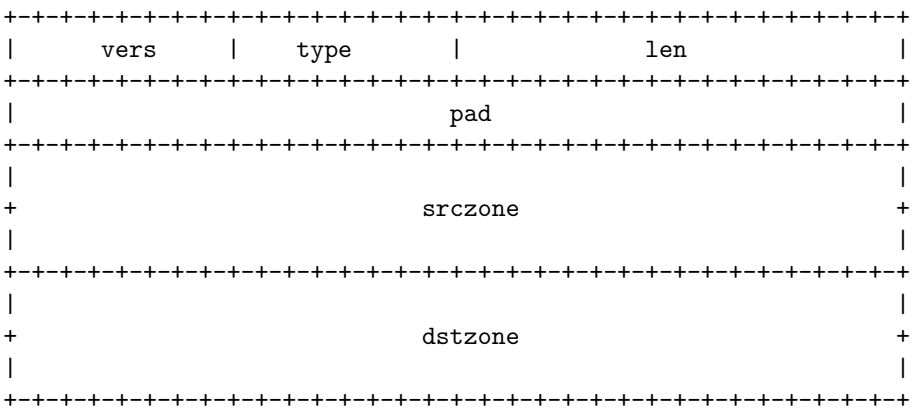
In order to provide intra-zone communication, each zone has its own loopback interface configured to 127.0.0.1. This means that a user telnetting to 127.0.0.1 results in a connection being established over the zone's loopback interface, back into the same zone. As stated in our requirements, using the `ipnet` device will allow all local traffic which includes intra zone traffic to be seen in the global zone. Given that all intra zone traffic uses 127.0.0.1 this presents the problem that there is no way to determine which zone the intra-zone traffic is related to.

Given this, a mechanism to pass ancillary data, specifically zone data, to consumers such as `snoop` is required. As such, for each IP packet, the source and destination zoneid will be provided if known. For IP packets where the destination address is a remote machine, the destination zoneid

will be unknown.The value for an unknown zoneid will be -1, ALL_ZONES. Similarly, where the source address of a packet is a remote machine the source zoneid will be unknown.

Being able to pass zone information up to a consumer such as snoop is only one part of the problem. This ancillary data may also need to be stored in a file for later processing. The ipnet device will be of type DL_IPNET and these devices may pass up an ancillary data header. Consumers such as snoop will be modified to understand the new DL_IPNET mac type and also to handle the ancillary data. By attaching the ancillary data to the mac type we avoid having to make snoop generically handle ancillary data. This was considered and is discussed below. For snoop there is still a potential problem: there is no way to indicate in the capture file that the ancillary data has been captured. This problem is dealt with by snoop always capturing ancillary data for DL_IPNET devices. With this approach, there does not need to be any change to the snoop file format described by RFC 1761. The changes to snoop will be discussed in a separate document which will be finished shortly.

The ancillary data header proposed for the ipnet device will be of fixed length, and contain a version number to aid expansion in the future as needed. The structure of the header is:



and in C expressed as:

```
typedef struct dl_ipnetinfo {
        uint8_t  dli_version; /* Current version */
        uint16_t dli_len;     /* sizeof dl_ipnetinfo_t */
        uint32_t dli_pad;     /* Padding for alignment */
        uint64_t dli_srczone; /* Source of zoneid where known. */
        uint64_t dli_dstzone; /* Destination zoneid where known. */
} dl_ipnetinfo_t;
```

The current version will be held in the following constant:

```
#define DL_IPNETINFO_VERSION 1
```

The header will be stored in big endian which matches the existing snoop header format. By default, the ipnet device will only pass up packet data. To get the ancillary data header, an application will need to issue the new DL_IOC_IPNETINFO ioctl. The new ioctl will take a boolean parameter to enable or disable the facility. Although the header will be of fixed size, the actual length will be dependent on the version. An application will issue a DL_IOC_IPNETINFO request passing in 1 to enable the facility. The device will return a non-zero value indicating the current DL_IOC_IPNETINFO version.

Other design choices for handling ancillary data within `snoop` were considered and are listed below, along with the reason we decided not to use it.

- Using `M_PROTO` messages to pass up the ancillary data from the device. This option was dismissed, as using `M_PROTO` for this reason would mean using `M_PROTO` differently to the semantics already defined by the DLPI specification. Given this is a DLPI device this seems inappropriate.
- Extending the `snoop` packet record format to generically handle ancillary data. Although revising the `snoop` format would potentially be beneficial, this was not something we wished to undertake as part of this project, as the size of the work did not match our time scales for the project. Also it is not clear whether parts of the work are feasible as existing device drivers could require modification.

A complete discussion of these various options can be found on the Clearview wiki[9].

3.6 Security implications

For these devices the project will introduce a new privilege, PRIV_IP_OBSERVABILITY. This will just allow access to the ipnet devices where packets can only be observed not sent. Although the new privilege only allows observation of traffic it does present additional risks to those associated with PRIV_NET_RAWACCESS. Local connections have always been more trusted than connections on the wire. For connections on the wire there is always the possibility of anonymous eavesdroppers, the same is not true for local connections. Given this, traffic on the wire might be subject to encryption whilst local traffic may not under the assumption there is no problem of someone eavesdropping. Allowing observation of local traffic clearly breaks this assumption.

Given the new risks exposing observation of local traffic brings, the proposed new privilege may be refined in the future, adding finer grained privileges. As an example of how this privilege might be refined, there could be separate privileges for observing intrazone, interzone, and inter-machine traffic. However, for now we plan on keeping one high level privilege, which only uid 0 will be granted by default.

Given that Trusted Extensions architecture is built upon zones it is clear that the ipnet architecture must also respect labelling information if the trusted extensions facility is being used. Where hooks are placed we check whether the system is lableled and if it is check what the zoneid should be based on the labelling information.

[9] http://clearview.east/wiki/index.php/Handling_ancillary_data_within_snoop

4 Interaction with ip

This section describes the proposed interaction between the ipnet device and ip. The ipnet device will register a callback function with ip. Where a hook is inserted there will be a check for whether the callback function is set. In the case where the callback function has been set a data structure containing all the information required by the ipnet device will be constructed and passed to it via the callback function.

The structure passed to the ipnet device will contain a copy of the IP header, the IP payload, source and destination zoneid and the interface index the packet arrived on/was been sent out on. Access to the IP header and IP payload will be read-only.

The callback function will only be set if there are active users of the devices created by the ipnet device. On the last close the ipnet device will cancel the callback function in ip.

As well as transmit and receive events we also require notification of interface events. For this we will use the proposed netinfo interface which will be introduced by the Packet Filtering Hooks project.

4.1 Requirements on the Hook Framework

As mentioned in section 3.2, this project relies on the Packet Filtering Hook Framework, specifically the netinfo(9f) interface that the Packet Filtering Hooks team is implementing. Our requirements on the netinfo interface are below:

1. Must be able to get an initial view of all network interfaces configured on the system.
2. Must be able to receive updates as and when the network interface changes. As well as addition and removal of interfaces, we also require notification when address configuration is changed. Specifically when an address on an interface changes or when it is configured up or down.

We are working with the Packet Filtering Hooks team to ensure our requirements are met.

4.2 Placement of ipnet Hooks in ip

When placing hooks in ip there is a need to make them symmetrical. If the hook placement is not symmetrical then a very confusing view would be presented to an administrator. Consider the asymmetrical case where on the inbound path hooks are placed after processing, and on the outbound path after processing. In the case of packets protected by IPsec packets would be seen after decryption on the inbound path, and after encryption on the outbound path.

For this reason hooks will be placed at the top of the inbound path before ip has processed the packet and at the bottom of the outbound path after ip has processed the packet. In addition to these, we will also hook in the local forwarding path.

The placement of the inbound and outbound hooks essentially mirrors the view at the DLPI layer. Having multiple tap points in ip would be ideal but introduces much more complexity including how an administrator chooses at which point they want to observe traffic. Whether tools like snoop are even the right choice for this observability also comes into question. However, the choice of hook placement still offers benefits namely viewing an IPMP group becomes possible.

One thing to note here is the behaviour in the case where checksum offload is in use on any of the interfaces. In this case the checksum seen when observing traffic through the associated `/dev/ipnet/` device will appear incorrect. This mirrors the current behaviour when observing traffic at the DLPI layer. Additionally in the case where `ip` loops packets back internally checksumming is turned off. This means when viewing local traffic via a device in `/dev/ipnet/` the checksum here will also appear incorrect. There is an open bug 4821256 which suggests adding the facility to enable/disable checksumming of loopback packets. Clearly if this feature is implemented then it will be possible to see correct checksums on the loopback.

5 IP Observability Device

The following subsections introduce the `ipnet` device, and how it will interact with the system. In all the following subsections, the example configuration from section 3.1 will be used. This is summarized in Table 4.

Interface	Address	Zone
`lo0`	127.0.0.1	G
`lo0:1`	127.0.0.1	Z1
`lo0:2`	127.0.0.1	Z2
`bge0`	192.168.0.1	G
`eri0`	10.0.0.1	G
`eri0:1`	10.0.0.2	Z1
`eri0:2`	10.0.0.3	Z2

Table 4: IP configuration for the `ipnet` device examples

5.1 Initialization

When the `ipnet` device is loaded, it will build an initial list of all the IP interfaces on the system using the `netinfo(9f)` interface provided by the Packet Filtering Hooks project, and then populate `/dev/ipnet`. Using the example configuration, this means the system will have the following nodes in `/dev/ipnet`:

```
/dev/ipnet/lo0
          /eri0
          /bge0
```

Internally, the `ipnet` device will manage the devices in `/dev/ipnet` using two data structures[10]: `ipolink_t` and `ipoaddr_t`. The `ipolink_t` structure will contain details regarding the data link, while `ipoaddr_t` structures containing IP address details for each address associated with a data link. The `ipoaddr_t` will include the zoneid that the address is associated with. The `ipolink_t` structure will act as a container for `ipoaddr_t`'s. Each `ipolink_t` structure will map to an entry in `/dev/ipnet`, while each `ipoaddr_t` will map to an address on the data link. Figure 1 shows these structures and how they are linked after the creation of the nodes in `/dev/ipnet`.

5.2 Network Interface Events

After the `ipnet` device has been initialized, and the initial set of devices in `/dev/ipnet` created we need a way of keeping this information updated. The update process will be handled by registering with the network interface event provider. The network interface event provider is still a work in progress, however common network events such as plumb/unplumb, up/down, address related changes will be supported. In terms of the data structures `ip` maintains, the `ipnet` device will be able to maintain its internal data structures by seeing changes to `ipif_t`'s, as the devices are really managing IP addresses configured on the system and the data links they are associated with.

[10] The netinfo interface will provide a more structured view of interface configuration. The structures here will change once more detail about the netinfo interface becomes available.

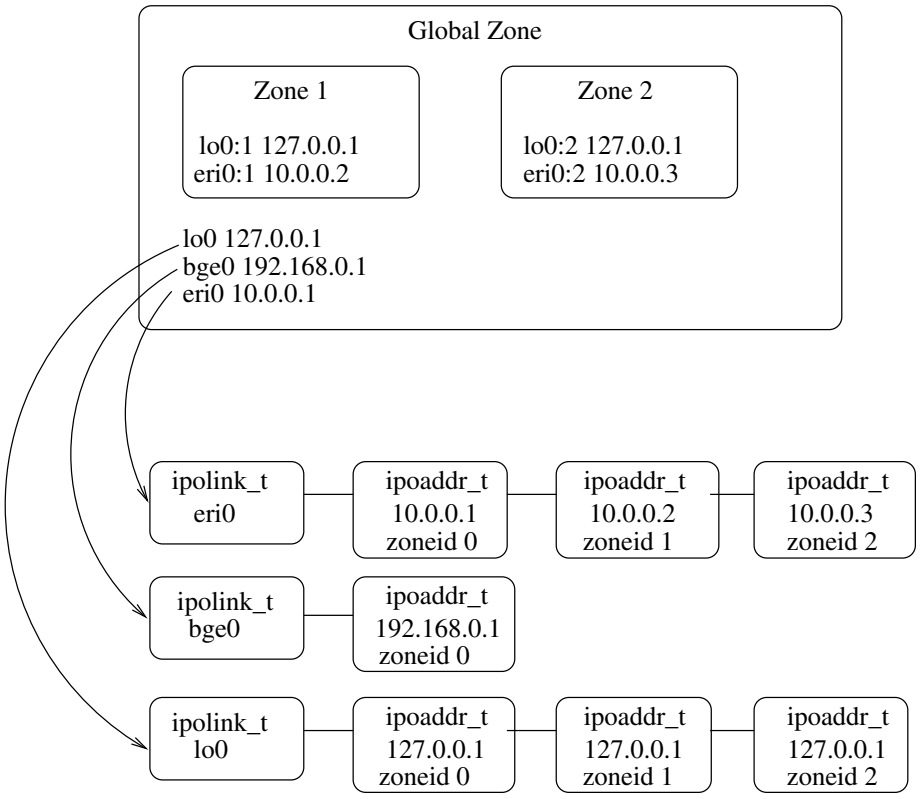


Figure 1: Data structures after initialization

5.3 Open

Handling per-stream instance data will be done through the `ipostr_t` data structure. Contained in the `ipostr_t` will be a pointer to an `ipolink_t`. Only users with `PRIV_NET_RAWACCESS` privilege will be able to open the devices in `/dev/ipnet`. On opening a device in `/dev/ipnet`, a new `ipostr_t` instance structure will be allocated and populated. The `ipolink_t` member will be populated by calling `ddi_get_soft_state()` to return the state associated with this minor device. The zoneid that the device has been opened from will be identified through calling `getzoneid()` and stored in the `ipostr_t` as well. The open of a device in `/dev/ipnet` will also trigger the device to register with the hook framework to receive observability events. As there will only one device in `/dev/ipnet` representing the data link, determining which `ipoaddr_t` instances a user can observe, will be achieved by calling `getzoneid()` and matching this to an `ipoaddr_t` contained in the `ipolink_t` structure for this device. Figure 2 shows the proposed interaction between the `ipnet` device and `ip` upon a user opening a device in `/dev/ipnet.` The Figure uses the example configuration.

5.4 Observability Events

When the `ipnet` device receives a packet via a transmit or receive event, it will fan out the message to all `ipostr_t` instances which should receive a copy of it. Leaving aside promiscuous mode handling the packet well be sent upstream if:

1. The source or destination address of the packet matches the IP address contained in the

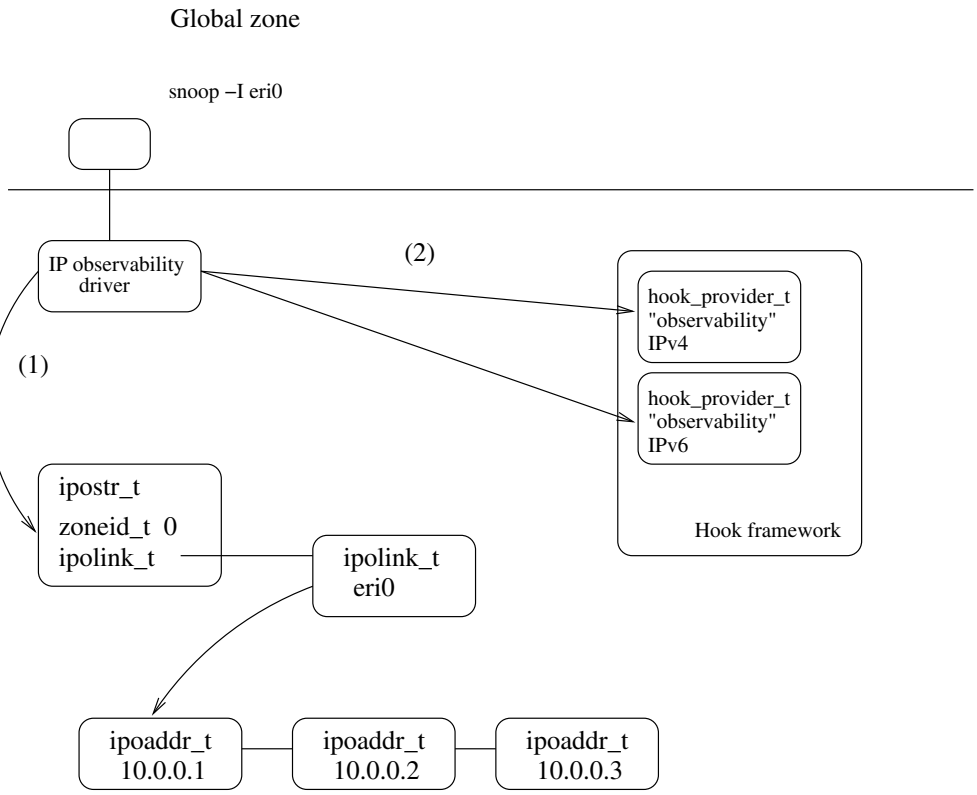


Figure 2: Device open

`ipoaddr_t` structures linked to the `ipolink_t` for the device open

2. The device was opened in the global zone **OR** the zoneid of the zone the device was opened in matches the zoneid contained in the `ipoaddr_t`.

In pseudo code, the basic processing path, leaving aside the additional checks that the promiscuity logic adds, is:

```
foreach ipostr_t {
        ipoaddr_t *ipoaddrp = ipostr_t->ipolink_t->ipoaddr_t;

        while (ipoaddrp != NULL) {
                if (packet->srcaddr == ipoaddrp->addr ||
                    packet->dstaddr == ipoaddrp->addr) {
                        if (ipostr_t->zoneid == GLOBAL_ZONEID)
                                pass packet up
                        else if (ipostr->zoneid == ipoaddrp->zoneid)
                                pass packet up
                }
                ipoaddrp = ipoaddrp->next
        }
}
```

Based on this example, packets will be passed up in the following way. Figure 3 illustrates the processing of a received packet. Step (1) shows the hook framework passing the `ipnet` device a

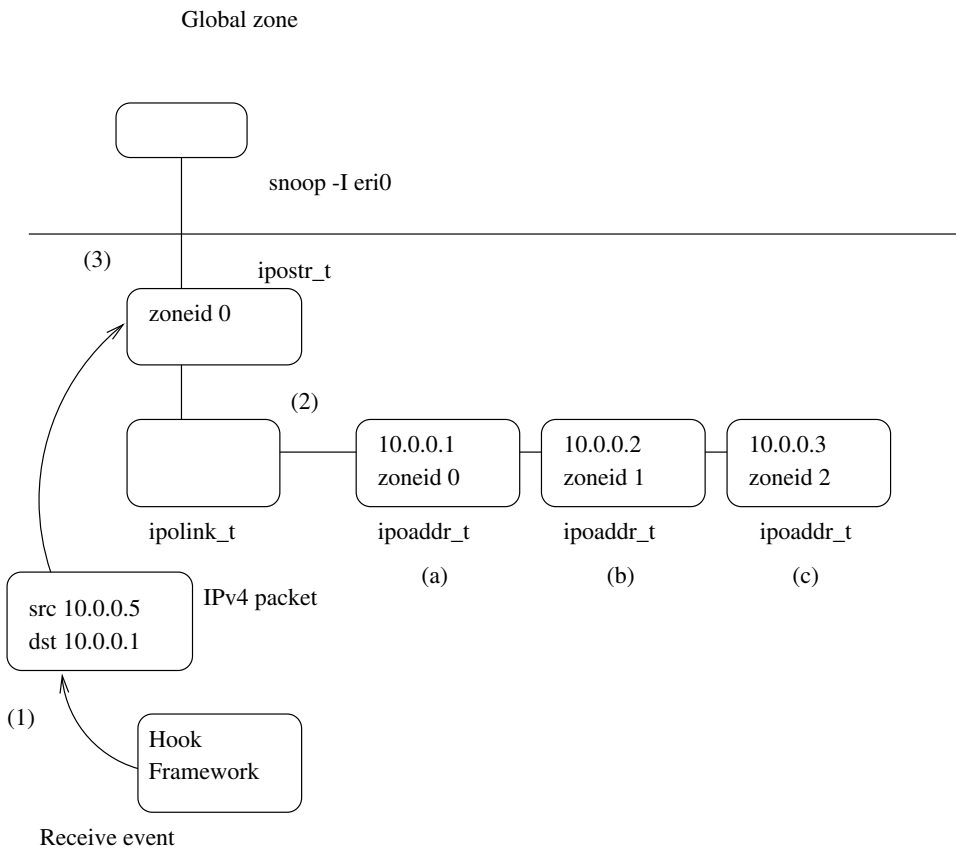


Figure 3: Receive event processing for an observer in the global zone

receive event. In step (2) the `ipnet` device checks whether this `ipostr_t` instance should receive the packet. The packet has a destination address of 10.0.0.1 which matches `ipoaddr_t` (a). As there is a matching `ipoaddr_t`, the final step (3) is to check that the zoneid in the `ipoaddr_t` matches the zoneid in the `ipostr_t`. As it matches, we pass the packet up.

Using the same example but moving the observer into Z1 the packet will not be passed upstream. As in the above example the packet will match `ipoaddr_t` (a) but the zoneid will not match and so the packet will not be passed up. This is as expected, a user in a non-global zone should only be able to see traffic associated with that zone.

5.5 DLPI Processing

Although the device will appear to consumers as a data link device, it is clearly not. While DLPI provides a convenient mechanism to capture packets, it is not reasonable for this device to become a generic DLPI device. Given this only the following DLPI request types will be supported:

- `DL_INFO_REQ` Get information
- `DL_BIND_REQ` Bind DLSAP address
- `DL_UNBIND_REQ` Unbind DLSAP address
- `DL_PROMISCON_REQ` Turn on promiscuous-mode
- `DL_PROMISCOFF_REQ` Turn off promiscuous-mode
- `DLIOCRAW` Turn on/off raw mode
- `DL_IOC_IPNET_INFO` Turn on/off the IP info ancillary data header

This subset provides the request types needed for a consumer such as `snoop`. The `ipnet` device will be a "style 1" data link service provider and users will not have to explicitly attach. The list below details how the device will respond to the above DLPI requests.

- `DL_INFO_REQ`
  On receiving a `DL_INFO_REQ` the device will reply with a `DL_INFO_ACK`, containing the following `dl_info_ack_t` structure:

```
dl_info_ack_t ipoinfoack = {
    DL_INFO_ACK,                /* dl_primitive */
    UINT_MAX,                   /* dl_max_sdu */
    0,                          /* dl_min_sdu */
    0,                          /* dl_addr_length */
    DL_IPNET,                   /* dl_mac_type */
    0,                          /* dl_reserved */
    0,                          /* dl_current_state */
    -2,                         /* dl_sap_length */
    DL_CLDLS,                   /* dl_service_mode */
    0,                          /* dl_qos_length */
    0,                          /* dl_qos_offset */
    0,                          /* dl_range_length */
    0,                          /* dl_range_offset */
    DL_STYLE1,                  /* dl_provider_style */
    0,                          /* dl_addr_offset */
    DL_VERSION_2,               /* dl_version */
    0,                          /* dl_brdcst_addr_length */
    0,                          /* dl_brdcst_addr_offset */
    0                           /* dl_growth */
};
```

- `DL_BIND_REQ`
  The `ipnet` device will only allow binding to `IP_DL_SAP`, `IP6_DL_SAP` and 0 when handling the bind request. Binding to anything other than `IP_DL_SAP`, `IP6_DL_SAP` or 0 will cause a `DL_ERROR_ACK` in response. The error returned will be `DL_BAD_SAP`.

- `DL_PROMISCON_REQ`
  The `ipnet` device will support all promiscuous modes:

  - `DL_PROMISC_PHYS`
  - `DL_PROMISC_SAP`
  - `DL_PROMISC_MULTI`

  Enabling `DL_PROMISC_SAP` will mean the device receives both IPv4 and IPv6 packets. Enabling `DL_PROMISC_PHYS` will mean receive all packets. If this is not set then only the packets with addresses matching any of the configured addresses on the data link will be received. Enabling `DL_PROMISC_MULTI` will mean receive all multicast group addresses.

- `DL_IOC_IPNET_INFO`
  The `DL_IOC_IPNET_INFO` enables ancillary data to be passed up. The ioctl takes a boolean to enable/disable the facility. When the facility is enabled a non-zero integer reflecting the current `DL_IOC_IPNETINFO` will be returned.

- `DLIOCRAW`
  Issuing the `DLIOCRAW`[11] ioctl to the device will return successfully but have no effect. Making

[11] The `DLIOCRAW` ioctl was added by Sun, so that applications can receive packets that include the link layer header. Normally, received packets would be passed up without the link layer header.

DLIOCRAW cause the device to just pass up the IP payload and treating the IP header as the link layer header was considered. However, this would break the bind semantics. Being able to bind to receive IPv4 packets but actually receiving TCP packets does not seem correct.

In the case where the ipnet device receives a request it does not support it will respond with a DL_ERROR_ACK containing DL_UNSUPPORTED.

Everything in this section will be documented in the ipnet(7D) manpage for the device.

A Design of the /dev/ipnet Name Space

Very early on in the project the issue of how the IP observability devices should be named was raised. There were various proposals each with advantages and disadvantages and these are listed below and use the following `ifconfig` output to provide examples:

```
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
eri0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 129.156.173.47 netmask ffffff00 broadcast 129.156.173.255
eri0:1: flags=1000842<BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 10.0.0.5 netmask 0
lo0: flags=2002000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv6,VIRTUAL> mtu 8252 index 1
        inet6 ::1/128
eri0: flags=2000841<UP,RUNNING,MULTICAST,IPv6> mtu 1500 index 2
        inet6 fe80::203:baff:fe2d:39b3/10
```

A.1 Device Name Based on IP Address

In this scheme devices are created in `/dev/ipaddr/` based on the IP addresses configured on the system. Using the example `ifconfig` output above this would mean we would have:

```
/dev/ipaddr/127.0.0.1
           /129.156.173.47
           /10.0.0.5
           /::1
           /fe80::203:baff:fe2d:39b3
```

Advantages of this approach are that it is very explicit. If an administrator wants to see packets on the 10 network they just run `snoop -d ipaddr/10.0.0.5`. This frees them from being tied to an interface name which may of course change. Network administrators think in terms of IP addresses not devices. From an administration perspective though this naming scheme seems un-natural. Administrators expect to use tools such as `snoop` on devices returned for example from `ifconfig`. With this naming scheme it would also be hard to incorporate the capture of forwarded traffic. Specifically what device should you open to see forwarded traffic? Additionally devices based on an IPv6 address look strange.

A.2 Device Name Based on Logical Interface Name

In this scheme devices are created based on the logical interface name. Using the example `ifconfig` output we would have:

```
/dev/ipnet/lo0
          /eri0
          /eri0:1
```

Advantages to this are that the naming scheme should be very familiar to an administrator. There are disadvantages this with design though. By treating logical devices as real devices we are adding to the confusion that already exists, namely that logical interfaces are not real devices but really address aliases in disguise. Given the way logical interfaces are implemented there is no guarantee

that a particular address will be assigned to the same logical interface from one boot to another. For example, there is no guarantee that zones will boot in the same order twice, nor that IPv6 stateless address auto configuration will assign prefixes in the same order twice. Running `snoop` on `eri:1` could therefore give you different results from one invocation to the other, making it difficult to script. Defining the device as being associated with an IP address is more deterministic.

A.3 Device Name Based on the Data Links in the System.

In this scheme devices are created based on the data links on the machine. Using the example `ifconfig` output we would have:

```
/dev/ipnet/lo0
          /eri0
```

This approach makes sense from an administration point of view, it should feel comfortable for an administrator to type `snoop -d ipnet/eri0`. Initially the thought was that `snoop`'ing on one of these devices would give you all packets sent from/to the device by `ip`. This would mean you would miss packets that are looped back inside `ip`, broadcast/multicast packets and forwarded packets. This design choice was extended to take in parts from the first two design options in that it would provide packets sent and received by `ip` to and from the data link of the same name but also IP packets looped back that have a source or destination equal to an IP address hosted on that IP interface.

Given the choices the final option seems to give the best result.

B Design Review Concerns

During the design review there were several concerns raised over the design. The concerns are listed below followed by a detailed discussion for each point.

1. The proposed new DLPI mac type will cause problems for third party applications.
2. The proposed `-I` option may cause confusion.
3. The proposed hierarchy should contain zone information.
4. The proposed design extends too far by providing inter-machine traffic as well as intra-machine traffic.
5. There is no discussion about other designs considered by the team.

In the case of (1) this is new functionality so existing applications will still work, they just won't understand `snoop` files produced from the new DLPI mac type. The alternative would be to re-use an existing mac type such as `DL_ETHER`. Making the new observability devices look like Ethernet devices just isn't correct. We also want to capture zone information in the header so that it's easy to see which zones the traffic is associated with. One can think of workarounds for this but none that are foolproof. We could overload the pseudo Ethernet header with this information but again this is just wrong and would potentially lead to confusion for someone looking at the raw hex data from a trace. Given we will be adding support for the new mac type to the major third party applications `ethereal`, `libpcap` and `tcptrace` we do not consider that third party products that

have been produced in house present a major problem. Adding support for DL_IPNET will be a fairly simple operation.

Concern (2) we believe is subjective. Given that this feature will be fully documented we believe any confusion will be mitigated.

Regarding concern (3), the hierarchy was something the team spent a considerable amount of discussing. We believe the current proposal provides a consistent and intuitive view. The namespace could be implemented to include zone information but the benefit is unclear. Furthermore on a system with many zones the `ipnet` directory would quickly become very cluttered. The device names would also have no direct mapping to the output of `ifconfig`, this inconsistency is something we are trying to avoid. It was proposed that by encoding zone information in the device name that there would be no need for zone information in the packet record capture header. However this still does not solve the problem of someone sending in a `snoop` file and not providing the device it was captured on. By capturing the information in the packet capture record this problem is solved.

Regarding concern (4), the requirements were extended to allow more than just intra-machine traffic primarily to observe an IPMP group without having to run `snoop` on the individual interfaces. We recognise that this is only a small piece towards `ip` observability and don't claim that this is a full `ip` observability solution. However, we don't see that implementing this now causes any problems for future work in adding a more complete `ip` observability solution. We do believe there is a clear benefit to this functionality now.

Looking at concern (5), other designs were considered, primarily around the `/dev/ipnet` namespace.

While we recognise these concerns we believe the proposed design will provide a consistent and intuitive model for IP traffic on the `ip` layer.

C Update Release Considerations

For this component of Clearview the only constraint for back porting the feature to an update release will be the availability of the Packet Filtering Hooks Framework. The Packet Filtering Hooks team are already targeting an update release so this does not present a problem.