

Packet Filtering Hooks Design Specification

Copyright 2006 Sun Microsystems, Inc. All rights reserved.

Darren Reed

Mike Ditto

Network Virtualisation Team

Solaris Software

Version 2.2

10. May 2006

Abstract

The packet filtering hooks project aims to introduce the capability for a party interested in intercepting packets to achieve this without needing to push a STREAMS module between IP and a network interface (NIC) driver, with the aim of facilitating packet filtering between zones. To achieve this we will be introducing a new interface for packet filtering inside the Solaris kernel. Through this interface, other modules in the kernel will be able to interact with network layer traffic using a stable, modular interface.

This document describes a new approach to providing the capability to intercept packets in Solaris. This approach is more inline with what is found in other operating systems today.

Table of Contents

1. Background.....	3
IPFilter in Solaris 10.....	4
2. Requirements.....	4
2.1.Boundaries.....	5
3. Proposed Architecture.....	5
3.1.Diagram of proposed structure.....	6
3.2.Network Interface Information.....	6
3.3.Network Protocol Configuration.....	7
3.4.Network Interface Events.....	7
3.5.Packet Interception.....	8
3.6.Packet Injection.....	8
3.7.Future Extension.....	9
4. Implementation.....	9
4.1.Network Protocol Interface.....	9
4.2.Solaris Changes.....	13
4.3.Infrastructure.....	15
5.Detailed Design.....	15
5.1.Structures.....	16
5.1.Events available with this project.....	17
5.3.Functions.....	19
5.3.Defining a network protocol within netinfo.....	20
Appendix A. Interface Stability Table.....	23
A.1.Exported Interfaces – Stable.....	23
Appendix B. Comparative Review of Interception Interface.....	24
Appendix C.Good and bad examples of using netinfo.....	25
C.1. Example 1.....	25
C.2. Example 2.....	25
Appendix D. Illustrations of IP code path changes.....	27

1. Background

From the inception of SunOS 5 and the introduction of STREAMS for the foundations of networking, the method supported by Solaris for intercepting packets has been to insert a STREAMS module at the correct location in the protocol stack. This was adequate for a long time, as there was little extra cost involved with inserting extra STREAMS modules, apart from the associated cost of extra work to do.

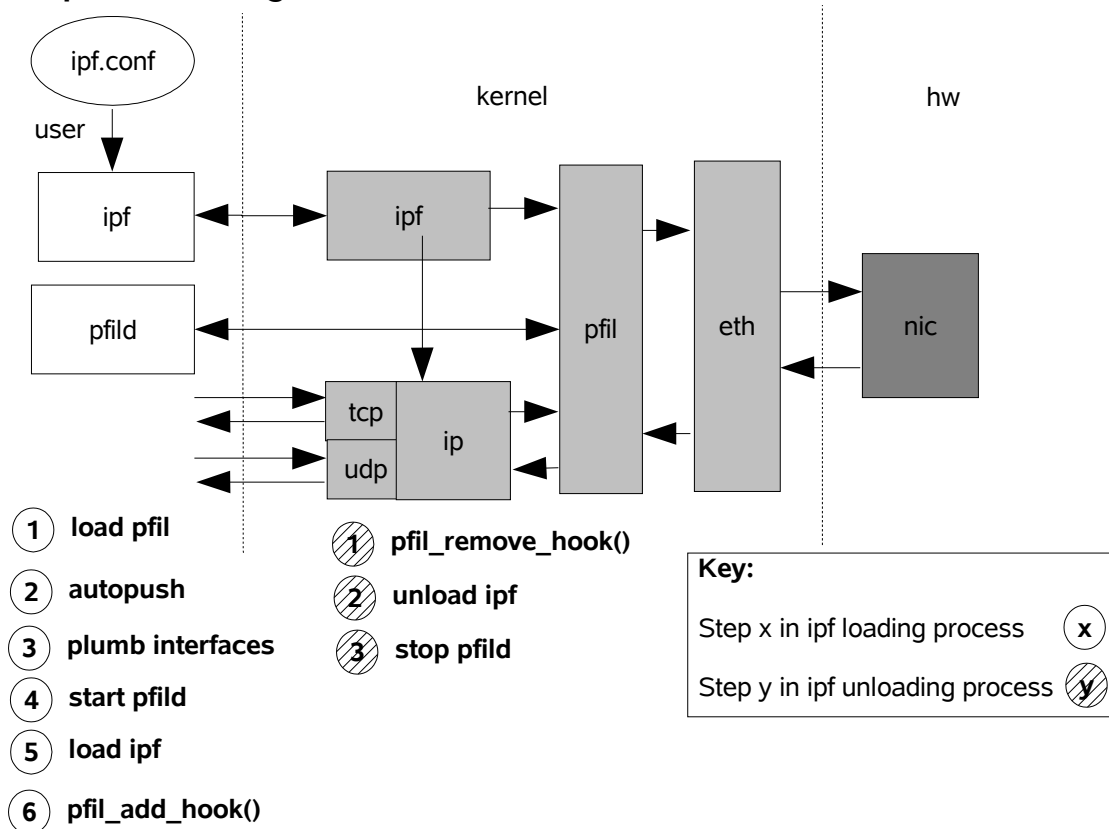
The specific STREAMS message protocol between IP and network device drivers has not been entirely stable. Although the interface is nominally DLPI there have been many incremental enhancements with varying compatibility. A variation known as *fast-path* messaging is recognized by essentially all NIC drivers but is not formally documented, even within Sun. Support for hardware offload of checksum processing [PSARC/1996/173] changes the appearance of some packets as they move between IP and the driver. TCP multi-data transmission [PSARC/2002/276] allows multiple data segments to be sent in a single STREAMS message. Each of these enhancements introduced changes in the format of messages passed between IP and device drivers. Consequently, inserting a STREAMS module that did not understand the messaging, such as a firewall, meant there was a high likelihood of incorrect behavior. A capability negotiation mechanism [PSARC/2001/070, PSARC/2003/263] was put in place to allow negotiation of incompatible enhancements, disabling those that are not understood by an interposed STREAMS module. Since then, there have been further enhancements that use this mechanism [PSARC/2003/264], [PSARC/2004/106], [PSARC/2004/174], [PSARC/2004/630], including a radically different data path between IP and device drivers, which give substantial performance improvements when used. Thus the insertion of a STREAMS module between IP and the device driver, today, can have a significant negative impact on networking performance because it causes these performance-enhancing features to be disabled for correct interaction with the STREAMS module.

Another problem with the STREAMS module approach is that there is no reliable way to cause a module to be present on every IP interface stream. The approach taken in Solaris 10 is to instruct the administrator to configure the autopush facility for each NIC driver that will be used. This is error-prone, user-unfriendly, and does not support all types of network interfaces.

There is one more nail in the coffin for using a STREAMS module for packet interception: zones. Network traffic between zones on the same system is all processed as loopback traffic. On Solaris, traffic sent to any IP address of the local system, even in a different zone, is delivered internally within IP and does not pass through any device driver. (This means that the network interface, lo0, is purely there as an administration interface and is never actually used to send packets.) There is no stream on which a packet filter STREAMS module can be pushed to gain access to this loopback traffic.

IPFilter in Solaris 10

S10 packet filtering



2. Requirements

To support IPFilter running inside the kernel, the operating system needs to provide the following interfaces to its network stack. Other firewall and firewall-like products have requirements very similar to these.

- access to all packets for a particular protocol that enter the system from the *wire*;
- access to all packets for a particular protocol that exit the system out onto the *wire*;
- allow the firewall to transform packets entering/exiting the system in any way;
- delay receipt or transmission of a packet, i.e., cause its processing to be asynchronous;
- feeding a packet generated by the firewall into IP for routing and delivery;
- emitting a packet generated by the firewall directly out of the system on a specific interface and to a specific next-hop router;
- resolve network interface names into a unique token;
- access some information that is specific to each network interface;
- read/write access to some network protocol configuration options;
- provide notification of changes to network interface configuration.

2.1. Boundaries

The above list of requirements comes close to filling all of the stated requirements from firewall software vendors - there are still a few requirements that need more discussion and investigation before being solved. For the rest of this document, we're confining our problem space to what appears above.

3. Proposed Architecture

To address these requirements, we are putting forward a design for a new interface inside the Solaris kernel to provide access to network data and functionality. The intention of this design is to focus on solving the problem for firewall/NAT software today but with the understanding that over time its scope may be increased to provide further access to network data/functionality.

To achieve this goal, we are proposing a new interface for the kernel – netinfo(9f). This interface is primarily intended for access to network-layer protocols (IPv4, IPv6) and not transport-layer protocols (TCP, UDP, etc) or those considered to be higher in the stack than TCP. An unfortunate artifact of only having Internet based protocols to work with in designing the interface is it requires the non-Internet protocols to present relevant information using the same model. The only area where this could be seen as a problem is in dealing with network interfaces: IP in Solaris has a physical/logical interface model that other network protocols might choose not to adopt.

A consideration that needs to be taken into account is that it is possible for Solaris to support more than one implementation of IP within the system at any given time, due to its use of STREAMS as the means to implement IP. By providing a generic interface, we're allowing other implementations of both IP protocols and other network protocols to make available relevant functionality. That there is a small penalty when using the model being proposed vs using direct calls into IP is understood, however, if we are to look at other subsystems within Solaris such as SCSI, the interface presented is through an abstraction layer and there is no direct interaction with the driver for the hardware. In following this model, IPv4 and IPv6 are being treated in the same fashion as specific SCSI card drivers.

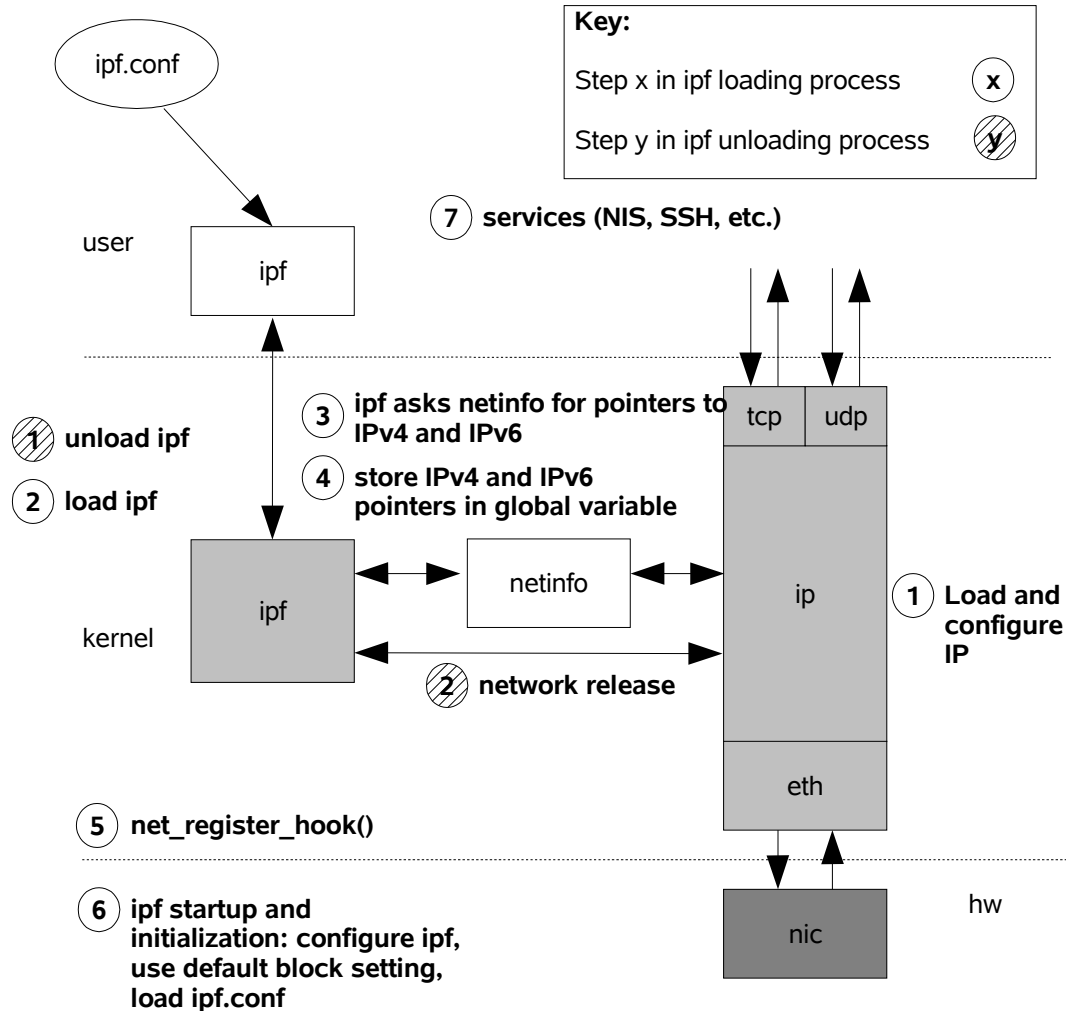
The components that make up the interface netinfo(9f) can be grouped as follows:

- network interface information
- network protocol configuration
- notification of events concerning network interfaces
- packet interception
- packet transmission

When dealing with network interfaces it is important to understand that in Solaris there is a distinction between physical and logical interfaces. What we see with output from ifconfig(1m) is a list of logical interfaces with names that match the driver used to interact with the physical interface. Thus, while this proposal makes it possible to match up packets traversing through Solaris with physical interfaces, other interface properties, such as addresses, must be accessed with an additional reference to a logical interface.

3.1. Diagram of proposed structure

S10 packet filtering hooks and GLDv3



3.2. Network Interface Information

For each network interface on Solaris, there's a collection of information stored relating to addresses as well as the interface's state. Not all of this information is useful for the purposes of writing software that is involved in either firewall or NAT activities. The network interface information is made available in a manner that mirrors the way in which Solaris stores it per logical network interface.

3.2.1. Network Addresses

A critical piece of information that belongs to each interface is that related to addressing. In Solaris today, a logical interface normally has the following addresses data associated with it:

- network/host address (with netmask/prefix length available separately)

- broadcast address (for broadcast links)
- peer address (for point to point links)

Because the number of address parameters and their sizes will vary, we see it as a disadvantage if we were to try and provide the means to fetch all address information at once via a static structure.

3.2.2. Other network interface details

The other details associated with a network interface are:

- maximum transmission unit size (MTU)
- network interface state (up or down)
- interface name

And finally, for each network interface, we need an identifier that can be used to refer to it when describing a packet's ingress or egress interface.

3.3. Network Protocol Configuration

A key problem faced by people wishing to write modules that interact with TCP/IP inside Solaris is getting access to the functionality it has and can therefore provide as well as information in its datastructures. To address this limitation, we're proposing to introduce a new kernel module that acts a layer between the implementation of network protocols (such as IPv4/IPv6) and those that might interact with them, typically firewall or address translation software. By creating a middle layer we hope to preserve the ability of developers to change, at will, the way TCP/IP is implemented in Solaris without impacting 3rd party module stability.

We recognise that in setting out to create this interface that perhaps not all functionality will be implemented or relevant to each network protocol. In light of this, the documentation for the interface will need to spell out both success, failure and “*not applicable*” results.

Network protocols may have some features that are configurable and impact the way in which certain packets are formed to be sent out over the network. For example, with IPv4 one such feature is path MTU discovery. In this case it is important for code within the kernel that constructs its own packets outside of IP to be aware of the current setting of this parameter so that it can fill in its own generated IPv4 headers to match the rest of the IP traffic on the system.

3.4. Network Interface Events

The details associated with a network interface can change over the course of its life. To keep track of what these changes are, we provide a mechanism by which a kernel module can register a callback to be invoked each time one of these events occurs. An event is generated each time one of the following occurs within IP:

- physical network interface is attached
- physical network interface is detached
- logical network interface is created
- logical network interface is destroyed
- a network address is changed on a logical interface
- a logical network interface is enabled

- a logical network interface is disabled

There is no event specifically for an address initially being set on a logical interface as this is treated as a change from an unspecified address (0).

3.4.1. Network Interface Event Concurrency

It is possible that multiple events could be activated simultaneously. The handler of these callbacks must therefore be written to be MP-safe.

3.5. Packet Interception

The immediate problem that we need to solve for Solaris is providing access to packets coming into and exiting out of a node. There are two prevalent models for packet interception today:

1. Defining specific points in the flow of packets through IP that a module needs to register with, individually (the Netfilter model adopted by Linux);
2. Defining a *firewall* hook and allowing a module to register with that and thereafter receiving both incoming and outgoing packets (this model is used by BSD based operating systems, including Tru64 and AIX.)

For Solaris, the model we've chosen to use for this implementation is (1). We have chosen this approach because

- we believe it offers a better path for future enhancements than does (2);
- it makes more sense to use a generic method for intercepting packets to achieve firewalling than it does to use a firewall API to implement packet interception for something else (e.g NAT).

Our current concerns, when it comes to providing support for packet filtering, are:

- access to packets coming into the system
- access to packets leaving the system
- access to *packets* moving internally within the system.

At present the first two can be easily achieved using a STREAMS module, albeit with a performance cost. The last is currently not possible and because of this, we're defining a separate hook for it, allowing the developer to easily choose not to be involved in intercepting packets that they have not touched previously.

Note that at this point in time, we are only supporting a single callback being registered for each packet interception hook at a given time.

3.5.1. Packet Interception Concurrency

It is possible for multiple packets to be processed simultaneously for either input, output or both. It is therefore a requirement that the callback function that intercepts packets must be written to be MP-safe.

3.6. Packet Injection

An important function of a firewall is to be able to generate its own packets, on behalf of either the system it resides on or other systems, to send as replies to the origin of a packet. The most common use of this facility is to generate *fake* TCP reset (RST) packets or *fake* ICMP error messages. When generating packets such as these, it is desirable to send them directly out of a network interface, without any further

processing.

In addition to this, a firewall may wish to delay processing of a packet or generate its own packet for processing by the system. In this regard, the firewall may wish to inject a new packet that looks like it has been received from outside of the system or it may wish to present a packet to the network protocol as if it had come from within.

This design will support three different modes of packet delivery, for packets that have been fully constructed (i.e. checksums computed) :

- (1) direct transmission of packets
- (2) queueing packets for input as if they had been received on a network interface
- (3) queueing packets for output as if they had been generated by the operating system

To implement (1), we have developed our own function to prepend the required layer 2 information (if there is any) and place the resulting message on the outbound queue for that network interface¹.

The implementation of (2) and (3) uses a task queue (see `taskq(9f)`) for delayed handling of the packet.

3.7.Future Extension

As the future unfolds future projects may come into being that wish to extend the interface being provided. Some areas where we see this as being likely are:

- implementing an equivalent of Linux's `NF_IP_LOCAL_IN` and `NF_IP_LOCALOUT`;
- providing information about zones;
- access to other network interface information, such as tunnel endpoint addresses.

4. Implementation

Given the proposed design, it is now time to explore what will change as a result of adopting this design and how code would interface with it.

4.1.Network Protocol Interface

To provide an interface to networking information within the kernel, we're providing a new interface to export network protocol functionality and data. The delivery method for this interface is in the form of a new kernel module called *netinfo*. This requires that both the network protocols that wish to publish functionality and modules that wish to access this information be linked against the *netinfo* module.

For a network protocol to make available functionality through this interface it needs to register itself with the *netinfo* framework by calling `net_register()`. The single argument to this function is a pointer to a `net_info_t` structure that contains a set of pointers to functions that implement the *netinfo* interface plus some fields that describe the structure itself (a version and protocol name.) We require that all functions be implemented and that each should return a value as indicated by the corresponding man page. If it is the intention of the interface being provided for a piece of functionality to not be provided, the man page will provide direction on what value should be returned to indicate this. The value returned by `net_register()` is to be passed back into `net_unregister()`. If `net_register()` is being called during the loading (`_init()`) phase of a module then `net_unregister()` should be called as it is being unloaded (`_fini()`). Be aware that the call to `net_register()` may fail with `EBUSY` if there are still outstanding references

¹ Surya will be providing a new function call to take care of this for us. This new function is being developed by surya as this function is intimately tied to the storage of data in IRE structures.

and in these situations, the calling module should fail the unload request.

For those that wish to make use of the functionality being exported through this interface, we provide access to the netinfo repository through **net_lookup()**. This function returns a token value that is then passed back through all subsequent interactions with this interface. The return value from this function can be safely cached for use, up until the call of **net_release()**, after which it should be considered as being invalid. It is possible to enumerate through all of the registered network protocols using **net_walk()**, however the value returned by **net_walk()** is considered to be invalid after it has been passed in on a subsequent call.

Each implementation of a network protocol is given the capability of providing a set of events that interested parties may listen in on. These events are provided asynchronously and may also be delivered simultaneously. The simultaneous nature applies not just to events of a different type but also to those of the same type. The implementor of this interface indicates its willingness to provide an event by calling **net_register_event()** with the return value from a successful **net_register()** and a suitably constructed **hook_event_t** structure. An event that has been registered can be removed – if there are no current users of it – by calling **net_unregister_event()** with the same parameters as used for **net_register_event()**.

This project delivers two types of events: those that are related to packets flowing through the kernel and those that are related to information pertaining to network interfaces. For more details on the individual events available with this project, please see “Packet Event List” on page 12.

4.1.1. Network Interfaces

In contrast with the implementation of TCP/IP networking found elsewhere, Solaris uses a model where there are data structures that represent both physical network interfaces and logical network interfaces. To be clear about what is a logical network interface, all output observed from ifconfig relates to logical interfaces. Even though the logical name of an interface may be presented in manner that makes it identical to that of its physical partner, it still remains a reference to a logical interface. In order for this project to access data that is visible with ifconfig, we must therefore provide an interface that mirrors this implementation.

Network interfaces in this project are presented as a physical¹ entity that has a number of logical entities attached to it. The physical entity does not have any address information associated with it, only the logical interfaces do. To some this may seem strange at first, due to the appearance from ifconfig(1m) output of these two being one and the same – especially in the absence of any extra logical interfaces.

To discover which physical network interfaces are available, it is possible to walk through those provided by a network protocol using **net_phygetnext()**. The start of the walk is indicated by passing in 0 as the current value for the second argument and the end of the walk is indicated by **net_phygetnext()** returning 0. If the name of a physical network interface is known, in advance, it can be searched for using **net_phylookup()**.

Using the **phy_int_t** value returned from either of **net_phygetnext()** or **net_phylookup()**, we can further use functions in this interface to learn about some of its properties, including assigned network addresses.

The current queries and set operations supported with a physical interface are:

- retrieving the name of the interface and storing it in a buffer using **net_getifname()**;
- returning the MTU value for the interface using **net_getmtu()**.

¹ Even though some physical interfaces may in fact be software based – such as loopback (lo0) – for the purposes of this document, we consider them to be physical interfaces.

As mentioned previously, a physical interface in Solaris does not have any addresses associated with it – they're only associated with a logical interface. Therefore, to retrieve network address information you must first acquire a handle for a logical interface. To do this, **net_lifgetnext()** must be called. This function behaves in the same manner as **net_phygetnext()** but with the addition of a **phy_int_t** for context.

The interception of packets is only associated with a *physical* network interface. For zones this means that if it is assigned an IP address on “hme0:2”, the packet will be seen to be associated with “hme0”. The reason for this is both historical (prior to this project it has never been possible to discover the logical interface) and technical – packet interception is not always performed when we know which logical interface a packet will belong to.

To recap on the presentation of information provided about network interfaces, a network interface can be discovered either by walking through all of those present (**net_phygetnext()**) or attempting to find a physical interface by name (**net_phylookup()**). Once we have a valid handle returned by one of these function calls, we can then use it to obtain information about the network interface. To retrieve address information associated with a physical interface requires that a reference to a logical interface be acquired first. This is done with **net_lifgetnext()** and the result returned fed into **net_getlifaddr()**. The model of having both physical and logical interface identifiers has been chosen because it reflects the actual architecture of Solaris's networking.

Using **net_getlifaddr()**, it is possible to return one or more components of network address information associated with the logical interface in a single call.

The interface used by Solaris to send a packet out to a specific destination can be determined using **net_routeto()**. Packets can either be sent out of an interface directly using **net_inject()** or they can be queued up for system processing on either the input or output side of IP using the same function.

4.1.2.Packet Interception

To register a function to be called for intercepting packets, **net_register_hook()** needs to be called. This function will only succeed if there is no other function currently registered for this purpose¹. Similarly, to stop a function from being called for packets being intercepted, **net_unregister_hook()** must be called with matching arguments to the original call.

When the registered callback is executed, it is passed a structure containing the following information:

- value indicating which interface the packet has been received on for inbound packets (a value of “0” indicates no relevant interface);
- value indicating which interface the packet is being sent out on for outbound packets (a value of “0” indicates no relevant interface);
- token to be passed back into the netinfo framework that indicates which protocol the packet belongs to;
- pointer to the start of the network protocol header;
- length of the network protocol header;
- pointer to the mblk that contains the start of the network protocol header;
- pointer to the start of the mblk chain containing the packet.

¹ This aspect of the design will be revisited in the future when we have a better understanding of the requirements.

4.1.2.1. Packet Event List

The list of events for which packets may be intercepted is as follows:

Event	Description
NH_PHYSICAL_IN	External packets that have been received by a network protocol but have not yet been processed for either local delivery or forwarding.
NH_PHYSICAL_OUT	Packets that have been passed through the routing infrastructure and are destined to transmitted externally.
NH_FORWARDING	Packets that have been received from an external source but are not destined for the local machine but for an address connected to one of its other network interfaces.
NH_LOOPBACK_IN	The receiving side of packets that have been sent by the system to an address that corresponds to one of its logical interfaces.
NH_LOOPBACK_OUT	The sending side of packets that are being sent internally to the system (their destination address corresponds to one of its logical interfaces.)

For packets that are intercepted while being forwarded, a value for both the received and intended outbound interface is passed through to the callback function.

4.1.3. Network Interface Events

The current status of networking in the operating system often changes, from unplugging a system from the network temporarily, to an interface's IP address changing as a result of DHCP. In light of this, it is far more effective to be able to receive some sort of notification that these events are happening rather than trying to walk through all of the network configuration to discover it.

4.1.3.1. Network Interface Event List

The list of network interface events that are being delivered by this project is as follows:

Event Name	Description
NE_PLUMB	This event is generated when a network interface is added to the configuration of a network protocol.
NE_UNPLUMB	This event is generated when a network interface is removed from the configuration of a network protocol.
NE_UP	After receiving this event a consumer of packet events can now expect to see packets associated with the physical interface with which this event is associated.
NE_DOWN	After receiving this event a consumer of packet events should now no longer expect to see packets associated with the physical interface with which this event is associated.
NE_ADDRESS_CHANGE	This event is generated when the address associated with a logical interface changes.

4.1.4. Callback Entitlements

Processing being done as part of the packet interception callback must adhere to one very important rule:

it should not, ever, block. A simple rule for programming callbacks is to treat them the same as you would code executing in interrupt context for a driver. For example, a callback is not permitted to call `kmem_alloc(9f)` with the `KM_SLEEP` flag nor `sleep_use_delay(9f)`. A callback is also not allowed to interface with user space, through the use of `copyin(9f)`, `copyout(9f)` or other similar functions.

NOTE: A callback is **not** allowed to make any function calls into the netinfo interface that involve the callback part of the interface. That is, it may not remove itself or add another while executing. In short, a callback **must not** call `net_register_event()`, `net_unregister_event()`, `net_register_hook()` or `net_unregister_hook()`.

4.1.5. Checksums

Due to the location of the interception points for packets in IPv4/IPv6, it is possible that the callbacks for these protocols will see incomplete checksums that *look* wrong. For packets generated by the system, on interfaces where hardware checksumming is supported only a partial checksum may be computed for TCP and UDP.

Where the TCP/UDP checksum is being offloaded, if the callback wishes to modify the network or transport layer header, checksums should be adjusted, rather than recomputed, in line with changes to the header fields. If there is also a change to the data portion of the packet, the callback should call `net_ispartialchecksum()` to discover whether or not partial checksumming is being used for this packet. The return value indicates the hardware checksumming capability provided: layer 3 full, layer 3 partial, layer 4 full, layer 4 partial, or any combination of them.

On inbound packets, it is possible that the network interface card has already validated some or all of the checksums found in the packet. As can be seen below, the locations for intercepting an inbound packet do not correspond to being *inside* of a point where the checksum is validated for IPv4 and IPv6. If the callback wishes to ascertain whether or not the checksums in the header are correct, it can call `net_isvalidchecksum()`. This function will return true or false, indicating whether the network and transport headers are correct.

4.2. Solaris Changes

4.2.1. Drivers

This project introduces two new kernel modules into Solaris, netinfo and hooks. Both of these drivers are classed as miscellaneous IO drivers and are installed in the `/kernel/misc` directory.

With the introduction of these two new modules comes the removal of another module – pfil. Previously this module was used to tap into packets as they moved between IP and the device driver for a NIC. With the callbacks being made from tap points inside of IP, there is no longer any need for this driver. The pfil driver also had a role in providing information inside of IP to IPFilter. The APIs being developed with this project eliminate the need for this role of the pfil module.

As a result of changes to IP to support the netinfo interfaces, both the IP and IPFilter modules must now be linked against a new driver called “neti” when being built. At present the hooks driver remains private and is not available for public consumption.

4.2.2. User space

Similarly, the presentation of data and functionality inside of IP through the netinfo interface allows us to remove pfil as its primary role was to feed data back into the pfil driver about routing and interface configuration so that IPFilter could access this information without using private interfaces inside of the

IP module.

With the removal of the pfil device driver and daemon, the svc:/network/pfil SMF service is also removed.

4.2.3.Changes in IP

To support this project, we need to make the following changes:

- (1) add code to IP's loading/unloading to correctly register and unregister the appropriate events within the netinfo framework;
- (2) insert code in the packet processing path that causes the execution of callbacks to occur for packet interception events if there is a callback registered for this purpose and
- (3) add a collection of functions that provide the required functionality to enumerate the netinfo interface

We achieve the first task, (1), by declaring a separate global structure for each of IPv4 and IPv6, initialised appropriately, and calling **net_register()** with each structure when the module is loaded. Following this, the startup code for IP will call **net_register_event()** for each event being supplied for IPv4 and IPv6. See “Packet Event List” on page 12 for a list of the events being delivered by this project. Whilst it is highly unlikely for IP to ever be unloaded, the matching code has been added to call **net_unregister_event()** and **net_unregister()**, just in case.

To implement (2) we've analysed the code paths taken to process a packet using visual code inspection as well as dtrace. Our aim is to instrument interception points for inbound and outbound locations as close to the boundary of IP as is reasonably possible. To minimise the impact on the execution of code when there are no callbacks registered, we use an internal field of a hook event structure as an indication of whether or not to activate the hook and call the registered function(s). With each one of these structures allocated from BSS rather than on the heap, no pointers need to be dereferenced to find out if an extra jump should be made. More details on the actual locations of the changes can be found below, in “Illustrations of IP code path changes” on page 27. We do not, however, expect that this extra if() statement will be free (in terms of CPU cost) and recognise that there will be a small performance hit. Our goal here is a performance drop of less than 1% and hopefully less than 0.5%. If we can later remove code from the IP processing path for optional features that can be modified to use the infrastructure provided by this project then we should be able to realise increases enough to at least offset this drop.

For (3), we need to add a set of functions that export the required data or functionality out of IP. The most important aspect to the implementation of these functions is that they must manage any internal state or locks for IP in a manner that is transparent to the caller. There is an implied expectation that code within IP that might hold locks will never use these functions. By way of example, we would not expect code inside IP to grab a lock on `ill_g_lock` and then call **net_phylookup()**. For each function call in the netinfo interface, at least one new one is added where that functionality is desired.

4.2.4.Changes in TCP

In Solaris's TCP code, there are a number of *shortcuts* that are taken when it is recognised that both endpoints of the connection are local to the kernel. The use of these shortcuts is such that full packets to deliver information from one endpoint to the other are never constructed once the connection has been established. This prevents an interested party in being able to see a full flow of TCP packets on the local machine – i.e it prevents stateful filtering from working properly.

To address this problem, the use of shortcuts inside of TCP is disabled when there are any active callbacks present for either NH_LOOPBACK_IN or NH_LOOPBACK_OUT are present.

Additionally, TCP can send packets directly to a network interface driver, rather than via IP, when sending IPv4 packets. In locations where this takes place, we prefix this action with a callout to the NH_PHYSICAL_OUT hook.

4.2.5.Changes in UDP

With the Yosemite project [PSARC 2005/082], a number of changes to the UDP project were made, bringing with it a substantial increase in UDP performance. As with the TCP driver, callouts needed to be added for NH_PHYSICAL_OUT, prior to delivering the packet out of the interface.

4.2.5.Changes in IPFilter

To bring IPFilter into line with this project, we need to replace all of the private interfaces it was using from the pfil module. These changes do not result in any loss or change of functionality in IPFilter.

IPFilter will use the physical-in (input) and physical-out (output) filter taps for controlling packet flow into and out of a computer running Solaris.

To enable filtering of loopback traffic it is necessary to tell IPFilter this in its configuration file. This setting must be placed before other *command* line. The syntax of this line is:

```
set intercept_loopback true;
```

4.2.6.Boot time

As we're removing the pfil module, the corresponding SMF service (svc:/network/pfil) is also being removed. Consequently the SMF service responsible for IPFilter (svc:/network/ipfilter) will no longer be dependent on the pfil service and will no longer start the pfil daemon. There is no reason to add any new SMF services, despite the addition of extra kernel modules. The linking in of IP and IPFilter against the new modules will cause them to be loaded through dependency resolution.

4.3.Infrastructure

This project does not introduce any new infrastructure, in the form of files or commands to run, that must be configured, maintained or executed, with the exclusion of enabling loopback filtering.

5.Detailed Design

This section presents the design in detail, showing the structures available by code that wishes to interact with this interface and the functions they use to do so.

5.1.Structures

```
typedef uintptr_t phy_if_t;  
typedef uintptr_t lif_if_t;
```

```

typedef struct net_info {
    int          neti_version = 1;
    char         *neti_protocol;
    int          (*neti_getifname)(phy_if_t ifp, lif_if_t lif,
                                   char *buffer,
                                   const size_t buflen);
    int          (*neti_getmtu)(phy_if_t ifp, lif_if_t lif);
    int          (*neti_getpmtuenabled)(void);
    int          (*neti_getlifaddr)(phy_if_t ifp, lif_if_t lif,
                                   size_t nelem,
                                   net_ifaddr_t type[],
                                   struct sockaddr storage[]);

    phy_if_t    (*neti_phygetnext)(phy_if_t ifp);
    phy_if_t    (*neti_phylookup)(const char *name);
    lif_if_t    (*neti_lifgetnext)(lif_if_t lif);
    int         (*neti_inject)(inject_t, net_inject_t *);
    phy_if_t    (*neti_routeto)(struct sockaddr *address);
    int         (*neti_ispartialchecksum)(mblk_t *mb);
    int         (*neti_isvalidchecksum)(mblk_t *mb);
} net_info_t;

typedef enum net_ifaddr {
    NA_ADDRESS = 1,
    NA_PEER,
    NA_BROADCAST,
    NA_NETMASK
} net_ifaddr_t;

typedef enum inject {
    NI_QUEUE_IN = 1,
    NI_QUEUE_OUT,
    NI_DIRECT_OUT
} inject_t;

typedef struct net_inject {
    mblk_t      *ni_packet;
    phy_if_t    ni_physical;
} net_inject_t;

#define        HOOK_VERSION        1

typedef void *net_hook_t;
typedef void *hook_event_token_t;

typedef int (*hook_func_t)(hook_event_token_t token, hook_data_t info);

typedef struct hook {
    int32_t     h_version;                /* Currently set to 1 */
    hook_func_t h_func;                  /* Pointer to callback */
    char       *h_name;                  /* Name of this hook */
    int        h_flags;                  /* Extra hook properties */
} hook_t;

typedef struct hook_event {
    int32_t     he_version;              /* Currently set to 1 */
    char       *he_name;                 /* Name of this hook list */
    int        he_flags;                 /* 1 = multiple entries allowed /
    boolean_t  he_interested;           /* true if registered callback exists */
} hook_event_t;

```

```

typedef void *net_data_t;
typedef void *nic_event_data_t;

typedef enum nic_event {
    NE_PLUMB = 1,
    NE_UNPLUMB,
    NE_UP,
    NE_DOWN,
    NE_ADDRESS_CHANGE,
} nic_event_t;

typedef struct hook_nic_event {
    net_data_t      hne_family;
    phy_if_t       hne_nic;
    lif_if_t       hne_lif;
    nic_event_t    hne_event;
    nic_event_data_t hne_data;
    size_t         hne_datalen;
} hook_nic_event_t;

typedef struct hook_pkt_event {
    phy_if_t       hpe_ifp;
    phy_if_t       hpe_oip;
    void           *hpe_hdr;
    mblk_t         **hpe_mp;
    mblk_t         *hpe_mb;
} hook_pkt_event_t;

```

5.1.Events available with this project

5.2.1Network events

```

/*
 * Names of network events available through netinfo with this project.
 */
#define NH_PHYSICAL_IN    "PRE_ROUTING"
#define NH_FORWARDING    "FORWARDING"
#define NH_PHYSICAL_OUT  "POST_ROUTING"
#define NH_LOOPBACK_IN   "LOOPBACK_IN"
#define NH_LOOPBACK_OUT  "LOOPBACK_OUT"
#define NH_NIC_EVENTS    "NIC_EVENTS"

```

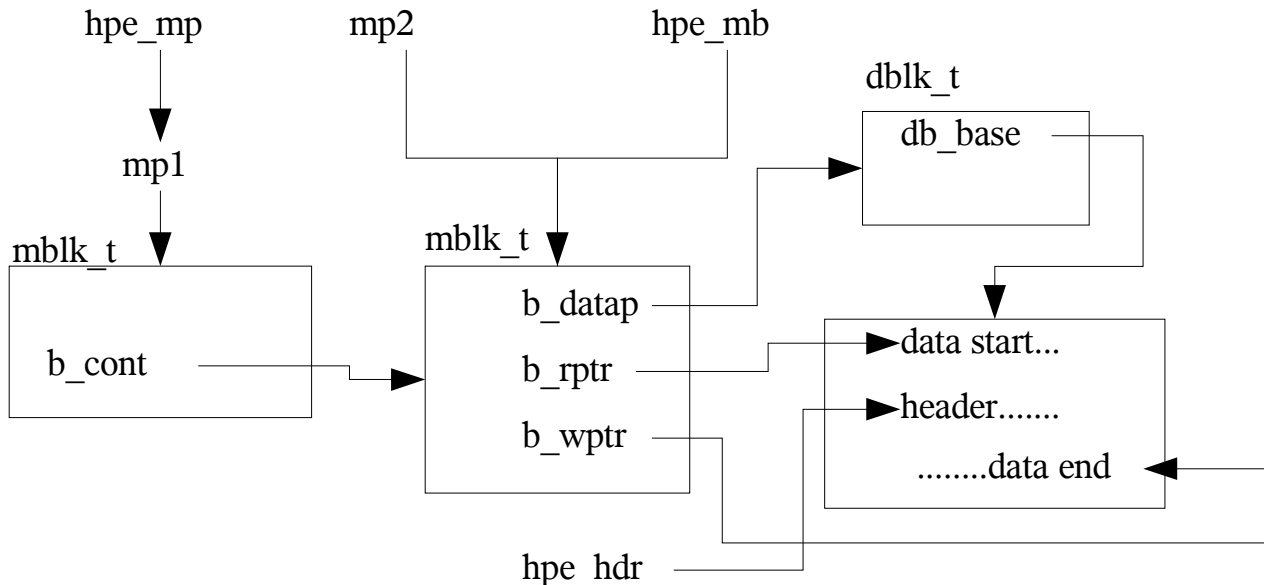
5.2.2Packet events

For callbacks that occur due to one of the packet events below, a pointer to a `hook_pkt_event` structure is passed in as the `hook_data_info_t` field. The validity of the fields in the `hook_pkt_event` structure is as described in the table below. The last row of this table indicates which fields made be changed by a callback and which fiields may not be changed.

Event	hpe_ifp	hpe_ofp	hpe_hdr	hpe_mp	hpe_mb
NH_PHYSICAL_IN	Yes	No	Yes	Yes	Yes
NH_PHYSICAL_OUT	No	Yes	Yes	Yes	Yes

Event	hpe_ifp	hpe_ofp	hpe_hdr	hpe_mp	hpe_mb
NH_FORWARDING	Yes	Yes	Yes	Yes	Yes
NH_LOOPBACK_IN	Yes	No	Yes	Yes	Yes
NH_LOOPBACK_OUT	No	Yes	Yes	Yes	Yes
<i>Allows to Change</i>	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>

The relationship between hpe_hdr, hpe_mp and hpe_mb is best described with the diagram below.



5.2.3 Network Interface events

In each of these events, the following fields in `nic_event_t` are always set as follows:

- `hne_family` contains a valid reference for a network protocol and will match that returned from `net_lookup()`, `net_walk()`, etc;
- `hne_nic` represents the physical network interface to which an event belongs to;
- `hne_event` is set to a value in the first column

Event	Field behaviour
NE_PLUMB	-
NE_UNPLUMB	-
NE_UP	-
NE_DOWN	-
NE_ADDRESS_CHANGE	<code>hne_lif</code> logical interface to which this event belongs to <code>hne_data</code> pointer to sockaddr structure with address in it <code>hne_data len</code> size of sockaddr structure

5.3.Functions

```
/*
 * Data management functions
 */
net_data_t net_lookup(const char *protocol);
net_data_t net_register(const net_info_t *);
int net_release(net_data_t);
int net_unregister(net_data_t);
net_data_t net_walk(net_data_t);

/*
 * Accessor functions
 */
hook_event_token_t net_register_event(net_data_t, hook_event_t *);
int net_unregister_event(net_data_t, hook_event_token_t);
net_hook_t net_register_hook(net_data_t, char *event, hook_t *);
int net_unregister_hook(net_data_t, net_hook_t);

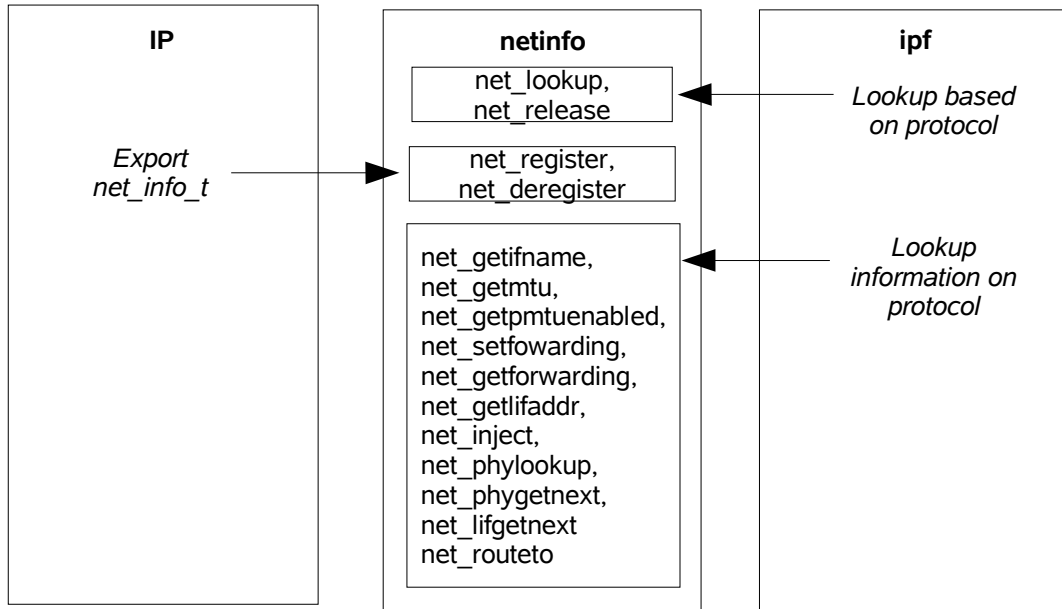
int net_ispartialchecksum(net_data_t, mblk_t *);
int net_isvalidchecksum(net_data_t, mblk_t *);

int net_getifname(net_data_t, phy_if_t, lif_if_t,
                  char *buffer, const size_t buflen);
int net_getmtu(net_data_t, phy_if_t, lif_if_t);
int net_getpmtuenabled(net_data_t *);
int net_getlifaddr(net_data_t, phy_if_t, lif_if_t,
                   int nelem, net_ifaddr_t type[],
                   struct sockaddr storage[]);

phy_if_t net_phygetnext(net_data_t, phy_if_t);
phy_if_t net_phylookup(net_data_t, const char *);
lif_if_t net_lifgetnext(net_data_t, lif_if_t);

int net_inject(net_data_t, inject_t, net_inject_t *);
phy_if_t net_routeto(net_data_t, struct sockaddr *);
```

IP, netinfo, and ipf interaction



5.3. Defining a network protocol within netinfo

5.3.1. Introduction

There are several steps involved in creating netinfo support for a new layer 3 protocol. This section details the framework that is required to support IPv4 in netinfo.

5.3.2. Declaration of `net_info_t`

A `net_info_t` structure must be registered upon protocol startup so that netinfo has a reference to the implementation of features available in that protocol. In the case of IPv4, the structure is declared as:

```
net_info_t ipv4info = { NETINFO_VERSION,
    AF_INET,
    ip_inject,
    ip_getifname,
    ip_getmtu,
    ip_getpmtuenabled,
    ip_getlifaddr,
    ip_iflookup,
    ip_ifgetnext,
    ip_routeto};
```

The first field of the `net_info_t` structure, `neti_version`, is initialized as `NETINFO_VERSION`. This could safely be omitted as this initialization is also done in the public header, `neti.h`. The second field, `neti_family`, is set to `AF_INET` to indicate IPv4. This is the same `neti_family` that is passed in and matched during a call to `net_lookup(9f)`.

5.3.3. Selecting an indexing scheme

Consumers of netinfo will utilize the function `net_lookup(9f)` to obtain a particular `lif_if_t` reference

to the interface on which an operation is to be done. The `lif_if_t` value returned by `net_lookup(9f)` is left up to the developer's discretion. However, by carefully selecting `lif_if_t`, development can be simplified.

For example, the IP netinfo implementation relies on the fact that the Solaris IP module has an internal tree of interfaces. The `lif_if_t` corresponds to the index of the interface node in the AVL tree that IP uses internally. By reusing the indexing in this tree, the data lookup process in IP netinfo becomes faster and easier.

5.3.4. Registering and deregistering `net_info_t`

In the IPv4 netinfo implementation, the `ipv4info` structure is declared inside of the IP kernel module. Upon loading of the IP module, it is registered with netinfo via the call:

```
net_register(&ipv4info)
```

At the unload of the IP module, `ipv4info` is deregistered via:

```
net_deregister(ipv4);
```

5.3.5. The accessor functions of `net_info_t`

The remaining fields after `neti_version` and `neti_family` correspond to the accessor functions as described in the netinfo specification. As a guide for writing these accessor functions for other layer 3 protocols, we will examine the implementation for the `neti_getmtu(9f)` function.

From `neti.h` and the netinfo specification, the `neti_getmtu(9f)` function pointer has the prototype:

```
int (*neti_getmtu)(phy_if_t);
```

In this example, we have registered a function called `ip_getmtu()` with the specified prototype. The function is shown below:

```
int
ip_getmtu(phy_if_t phyif)
{
    int ret;

    if (phyif == 0) {
        ret = ip_g_forward;    /* return the global property */
    } else {
        /* get the property from an ill */
        ill_t *ill;

        ill = ill_lookup_on_ifindex((uint_t)phyif, 0, NULL, NULL, NULL, NULL);
        if (ill != NULL) {
            ret = ill->ill_mtu;
            ill_refrele(ill);
        } else
            ret = -1;
    }

    return (ret);
}
```

This implementation is specific to the Solaris IPv4 implementation. It reads the IP module global variable `ip_g_forward` to determine whether or not forwarding is enabled for a specific node. In this case, `ip_g_forward` is global for IP, so all interfaces will have the same forwarding state.

5.3.6.Summary

This example is specific to the Solaris IP module implementation, however it illustrates the steps one would need to execute to implement netinfo support for any layer 3 protocol.

Appendix A. Interface Stability Table

A.1. Exported Interfaces – Consolidation Private

net_lookup()
net_register()
net_release()
net_unregister()
net_walk()
net_register_event()
net_unregister_event()
net_register_hook()
net_unregister_hook()
net_ispartialchecksum()
net_isvalidchecksum()
net_getifname()
net_getmtu()
net_getpmtuenabled()
net_getlifaddr()
net_phygetnext()
net_phylookup()
net_lifgetnext()
net_inject()
net_routeto()
phy_if_t
lif_if_t
net_info_t
net_ifaddr_t
net_inject_t
net_hook_t
hook_func_t
hook_t
hook_event_t
net_data_t
hook_event_token_t
nic_event_data_t
hook_nic_event_t
NH_PHYSICAL_IN
NH_FORWARDING
NH_PHYSICAL_OUT
NH_LOOPBACK_IN
NH_LOOPBACK_OUT
NH_NIC_EVENTS
NE_PLUMB
NE_UNPLUMB
NE_UP
NE_DOWN
NE_ADDRESS_CHANGE

Appendix B. Comparative Review of Interception Interface

Compared to Linux, what does this interface offer to developers wishing to intercept packets moving through the Solaris networking? The table below provides a direct list comparing the hooks available in Linux today with those being delivered by this project for Solaris. Whereas the symbol names representing the hooks in Linux are bound to the protocol family they are used for, e.g. `NF_IP_PRE_ROUTING`, this distinction is not present and is not necessary in the model we are proposing for Solaris. Two new hooks are presented here and extending the interface presented in Solaris is trivial as this service is provided through a generic event mechanism. This makes it trivial to add a new network protocol and/or network protocol event, without any need to recompile or impact on binary compatibility. At this point we are not adding analogues for `NF_*_LOCAL_IN` and `NF_*_LOCAL_OUT` as they are not required as part of this project to support IPFilter.

Linux	Solaris
<code>NF_*_PRE_ROUTING</code>	<code>NH_PHYSICAL_IN</code>
<code>NF_*_LOCAL_IN</code>	-
<code>NF_*_FORWARD</code>	<code>NH_FORWARDING</code>
<code>NF_*_LOCAL_OUT</code>	-
<code>NF_*_POST_ROUTING</code>	<code>NH_PHYSICAL_OUT</code>
-	<code>NH_LOOPBACK_IN</code>
-	<code>NH_LOOPBACK_OUT</code>
-	<code>NH_NIC_EVENTS</code>

Appendix C. Good and bad examples of using netinfo

C.1. Example 1

Good

```
net_data_t x;

_init()
{
    x = net_lookup(foo);
}

_fini()
{
    if (net_release(x) != 0)
        return EBUSY;
    return mod_remove(&modlinkage);
}
```

Bad

```
net_data_t x;

_init()
{
    x = net_lookup(foo);
}

_fini()
{
    net_release(x); // Return value should never be ignored
    return mod_remove(&modlinkage);
}
```

C.2. Example 2

Good

```
net_info_t foo[3];
function()
{
    foo[0] = net_lookup(PROTO1);
    foo[1] = net_lookup(PROTO2);
    foo[2] = net_lookup(PROTO3);
}
```

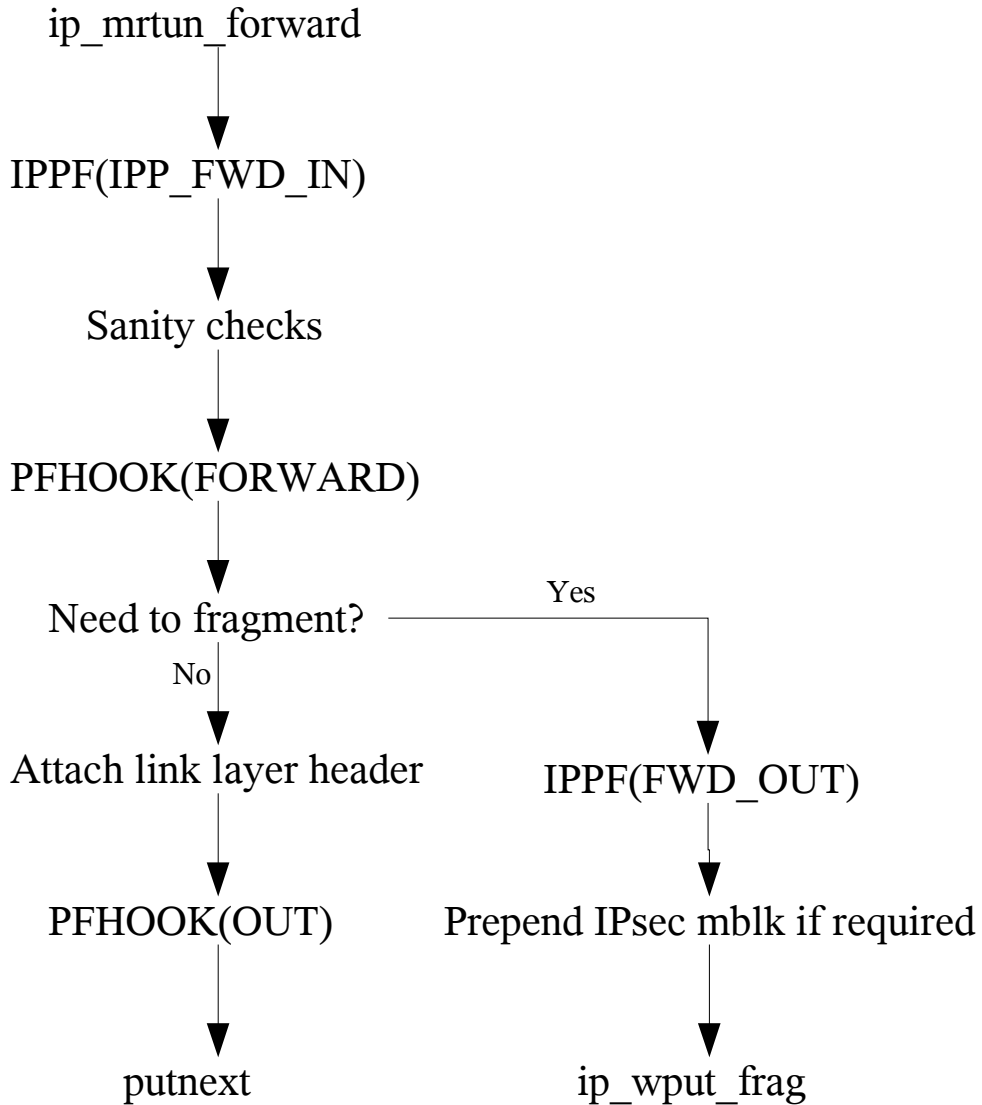
Bad

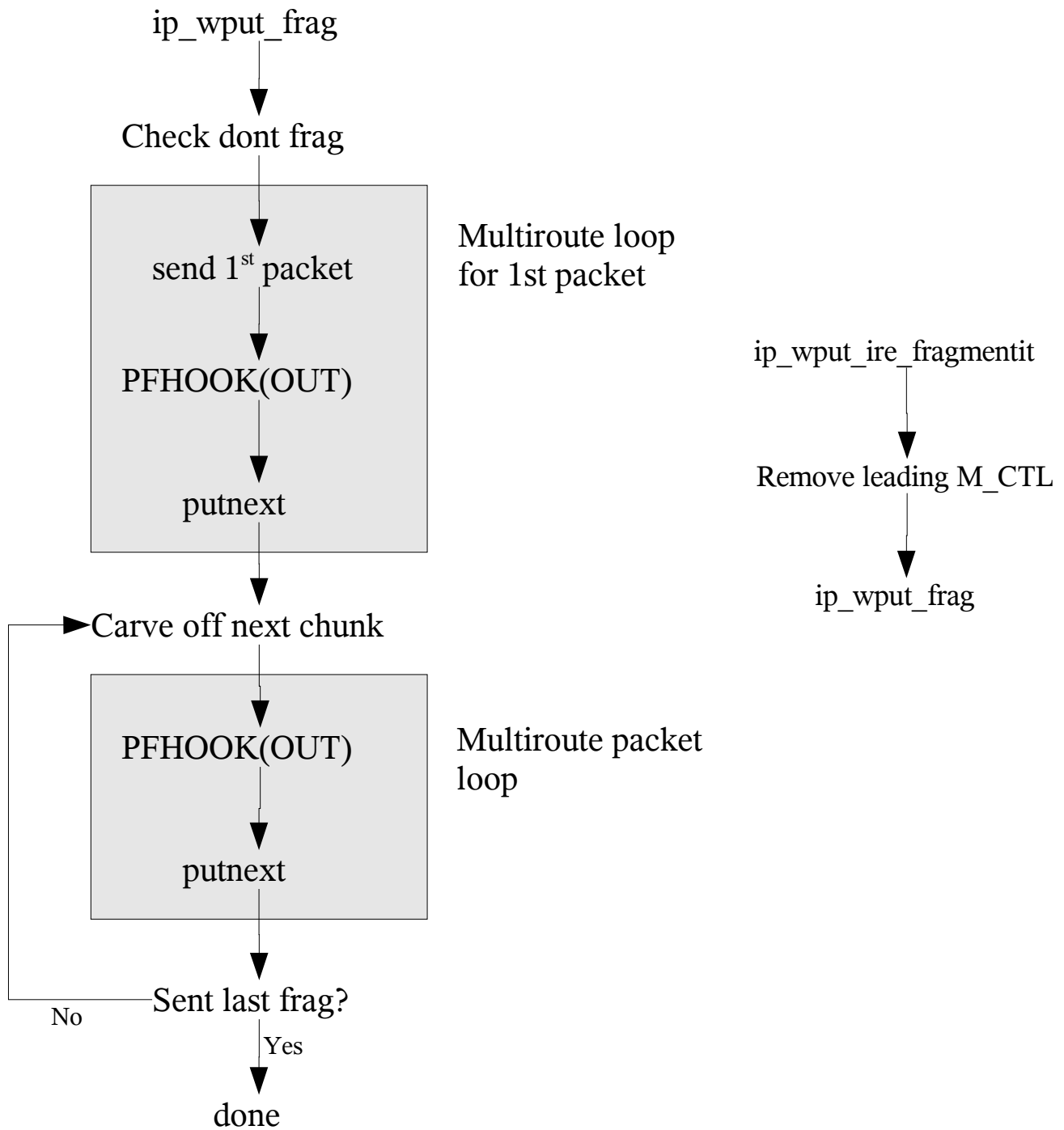
```
net_info_t foo[3];
function()
{
    net_info_t *x;

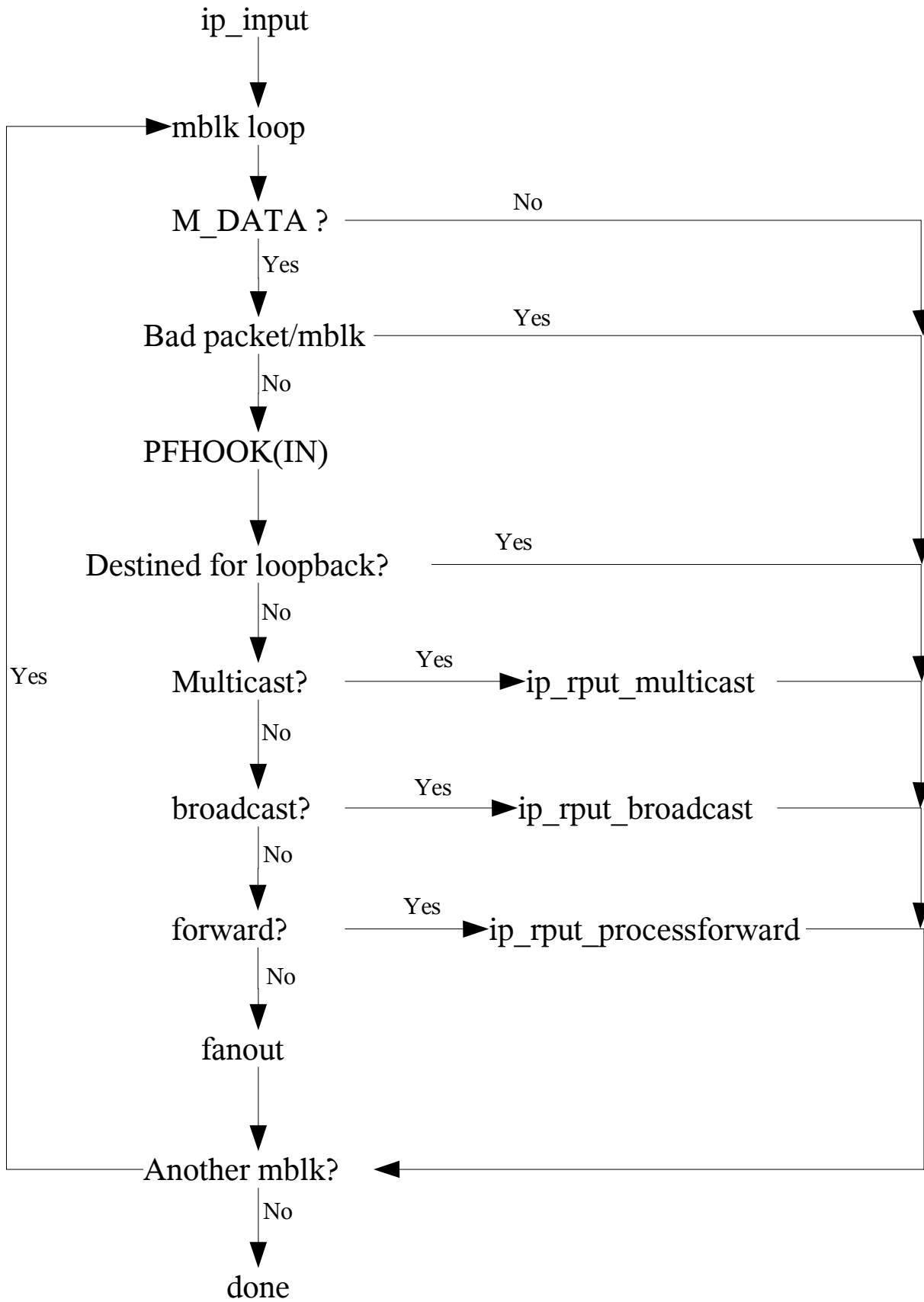
    for (i = 0, x = net_walk(NULL);
        (i < 3) && (x != NULL);
        i++, x = net_walk(x))
        foo[i] = x;
    for (; i < 3; i++)
        foo[i] = NULL;
}
```

// Return values from net_walk() are not valid after they
// have been passed back into net_walk()

Appendix D. Illustrations of IP code path changes







ip_rput_forward_multicast

