IPMP Rearchitecture: High-Level Design Specification

Peter Memishian (meem)

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

Solaris Networking
Sun Microsystems, Inc.

Revision 1.14
November 13, 2008

Contents

1 Introduction

1.1 Brief Overview of IPMP

IPMP is a popular Solaris-specific multipathing technology which operates at the IP layer. Specifically, IPMP attempts to insulate IP-based networking applications from changes to the underlying networking hardware on the system and the system's connectivity to the network as a whole. For instance, if two networking interfaces are put into an "IPMP group", then the failure or removal of either interface from the system will not affect applications using the IP addresses hosted on those networking interfaces. Further, the inbound[1] and outbound networking traffic using those IP addresses will be load spread across the networking interfaces in the group, providing greater network utilization.

As networking availability is a widespread concern, several other multipathing technologies exist in Solaris today: 802.3ad (link aggregation), Sun Trunking, Routing (OSPF-MP), CGTP, and SCTP. While a detailed comparison of these technologies is beyond the scope of this overview, it's worth summarizing where IPMP fits among them in order to separate the key goals of IPMP from the current implementation details.

Specifically, IPMP is intended to provide high-availability and improved network utilization for IP-based networking applications[2], while also satisfying the following requirements:

- Must work with all networking cards supported on Solaris.
- Must not require changes to existing hosts and hardware on the network.
- Must not require changes to existing applications.
- Must support all IP-based protocols.

The first two requirements are not met by 802.3ad and Sun Trunking[3], both of which require an 802.3-based link-layer and hardware (switch) support. Further, due to implementation limitations, only a few Solaris network interfaces currently support 802.3ad or Trunking. The second requirement is also not met by a routing-based solution, which requires that all participating systems run a routing protocol, and also does not support all IP-based applications (e.g., multicast-based applications are not supported). Likewise, CGTP[4] requires that all participating systems run CGTP (e.g., in order to filter duplicate packets), and is generally only suited to traffic between hosts on the same link. Finally, SCTP does not meet the third and fourth requirements, since applications must be recoded to use it and must use SCTP as the transport protocol.

Unfortunately, IPMP's high-availability and network utilization features currently come at a cost. As detailed in the following section, IPMP is hard to administer, often impossible to observe, poorly integrated with other core Solaris features, and quite complicated in implementation. As we will see, these problems stem from an improper relationship between IP interfaces, IP addresses, and IPMP groups. Since Project Clearview[5] is specifically focussed on rationalizing the Solaris network interface abstractions, and moreover, since the current IPMP implementation violates several of the network interface requirements[6] imposed by Clearview, it is both opportune and necessary to rearchitect IPMP as part of Clearview.

While brief background material is provided herein, **those who have not regularly used IPMP are *strongly* encouraged to read through the IPMP chapters[7] of the Solaris 10 System**

[1] Inbound load-spreading has never been explicitly documented but many customers are aware of the feature.
[2] This includes Java-based networking applications as well.
[3] Sun Trunking is an add-on product – not part of base Solaris.
[4] CGTP is also an add-on product – not part of base Solaris.
[5] http://opensolaris.org/os/project/clearview/
[6] http://opensolaris.org/os/project/clearview/overview.txt
[7] http://docs.sun.com/app/docs/doc/816-4554/6maoq027c

Administration Guide before proceeding. For those who need a quick refresher on IPMP terminology, a brief glossary is provided at the end of this document.

1.2 Problems with IPMP

While IPMP has been quite popular with customers, its adoption has been seriously hampered by a significant number of implementation and design flaws. Specifically, IPMP currently:

- **Provides a frustrating and overwhelming administrative experience.**

  For instance, `ifconfig -a` on a host using IPMP can yield a bedazzling amount of output[8], but does not provide easy answers to common questions such as "What interfaces are currently active in IPMP group `a`?", "What IP addresses are currently available for use in group `a`?" or "Has IPMP group `a` failed completely?". In addition, despite that many IPMP configurations use a significant number of IP addresses, it is not possible to configure these addresses through DHCP. Further, key questions such as "What probe targets are being used on group `a`?" can only be answered through monitoring network traffic.

- **Promotes a misunderstanding of the relationship between addresses and interfaces**

  Because `ifconfig` shows specific IP addresses hosted on specific IP interfaces, the administrator is lulled into a fundamental misunderstanding that IP addresses are actually "owned" by a particular IP interface. This is not true: a packet may be sent over *any* active interface in the IPMP group, regardless of its source address. Despite attempts to document this issue, this remains a source of constant confusion and a call generator.

- **Migrates IP addresses between interfaces during interface failure, repair, offline, and undo-offline operations.**

  This migration confuses applications that operate at the network-interface level (such as routing daemons), and prevents the use of DHCP to manage the addresses. For instance, `dhcpagent(1M)` has no way to realize that an address that it was managing on `ce0` is now on `ce1`. Further, DHCP uses the network interface as the administrative handle to manage a lease (e.g., `ifconfig ce0 dhcp extend`), but with IPMP, this handle is not stable over the lifetime of the lease.

  It also frequently startles and confuses Solaris administrators, since it spreads the misconception that the addresses are tied to specific interfaces (in truth, all the interfaces in an IPMP group share a pool of *all* of the group's addresses).

  Similarly, when operating in a zone, IP interfaces seem to shift around as if by magic. For instance, suppose `ce0` and `ce1` are together in a group in the global zone, and the IP address used by a shared-stack local zone is initially hosted on `ce0:1`. If `ce0` fails, a subsequent `ifconfig -a` invocation in the local zone will no longer show `ce0:1`, but instead show the logical interface on `ce1` that the address migrated to (e.g, `ce1:2`).

- **Provides no facility for interacting with the traffic flowing through the IPMP group as a whole – instead, each network interface in the group must be operated on individually.**

  Since any interface in the IPMP group may be used to send or receive traffic, debugging network problems using tools like `snoop` is made much more complicated: each of the interfaces in the group must be individually monitored, and the resulting packet traces must be carefully merged to get a complete view of the relevant traffic.

[8] http://opensolaris.org/os/project/clearview/ipmp/jurassic-ifconfig.txt

Similarly, this introduces problems for packet filters and other facilities that associate policy with network interfaces: since administrators cannot apply or observe policies associated with the group as a whole, they must instead carefully apply the policy to each of the group's underlying interface[9]. Further, this policy must be manually updated whenever interfaces are added or removed from the group.

A more serious problem exists with stateful firewalls: although packets may be sent and received using any interface in the group, the firewall is unaware of the relationship between the interfaces and thus cannot share state across the interfaces in the group. This renders stateful firewalls unusable with IPMP[10].

- **Introduces confusing and non-deterministic routing table semantics.**

  Routes are added to the routing table using the underlying network interfaces that make up the IPMP group. However, to provide reasonable IPMP semantics, those routes actually apply to the whole group, rather than just the specified network interface. Besides being generally misleading to administrators and applications, this leads to a number of ugly corner cases, such as what to do when the interface is removed from the group (or from the system!)

  Additionally, one is not allowed to add routes that use a failed interface – even though the route applies to the group as a whole and though there are other functioning interfaces in the group. Since there is no way to predict when an interface will fail, this introduces a non-deterministic point of failure.

- **Provides error-prone and incomplete routing socket (`route(7P)`) semantics.**

  Applications such as routing daemons that make use of routing sockets get an incomplete and confusing picture. For instance, if an address migrates from `ce0` to `ce1` as a result of a failure on `ce0`, then an application will first see the addresses for `ce0` disappear (and potentially have to notify other systems that the address that was on `ce0` is no longer reachable), only to later see the address reappear on `ce1` (and potentially re-notify systems that the address is again reachable). For routing, this odd behavior can trigger failsafe mechanisms (e.g., in Cisco's IOS), such as route flap dampening and hold-downs, can lead to extended outages.

  Further, there is no way to observe the IPMP group as a whole using routing sockets – only the underlying interfaces that comprise it can be observed.

- **Exposes IPMP test addresses and `INACTIVE`, `OFFLINE`, and `FAILED` IP interfaces to unsuspecting applications.**

  Applications that need to discover and use IP addresses or IP interfaces on the system must be explicitly updated to ignore IPMP test addresses and IP interfaces that IPMP has deemed unusable. Specifically, the well-known `SIOCGLIFCONF ioctl` returns *all* IP addresses and interfaces, even interfaces that have been marked `OFFLINE`, `INACTIVE`, or `FAILED`, and also returns IPMP test addresses. The latter is particularly troublesome, because the application appears to work normally until the interface hosting the test address fails.

- **Comprises thousands of lines of complicated, subtle, and fragile kernel code.**

  The code to support address migration (failover and failback) in the kernel is extremely complicated, involving more than 7500 lines of subtle and fragile logic that exists solely to support IPMP. This support impacts the core data structures, such as `ipsq_t`, which have special support for merging with and splitting from other `ipsq_t` structures just for IPMP address migration.

[9] Though the `ipmp_hook_emulation ndd` tunable provides short-term relief.
[10] Again, the `ipmp_hook_emulation ndd` tunable provides short-term relief.

2 Requirements

Simply put, the requirements are to address all of the aforementioned flaws without affecting backward compatibility. Specifically:

1. IP addresses in an IPMP group must never migrate as part of failure, repair, offline, or undo-offline operations.
2. IP addresses in an IPMP group must be administered using the same tools and in the same manner as IP addresses not placed in an IPMP group. As a corollary, address autoconfiguration through DHCP must work.
3. The IPMP group's core configuration and current state must be able to be easily and concisely obtained.
4. The IPMP group as a whole must be observable through packet monitoring tools such as `snoop(1M)`. It must also still be possible to observe individual interfaces as well.
5. The IPMP group as a whole must be able to administered as a whole – e.g., specified directly to `route(1M)`, and visible directly through `netstat(1M)`.
6. IPMP test addresses must not be visible to applications unless explicitly requested.
7. Migration of `UP` addresses must be obviated (to prevent complex migration code in the kernel).
8. The documented IPMP administrative commands (`if_mpadm`, `ifconfig`, and `cfgadm`) must continue work as before.
9. The documented IPMP semantics, e.g., for failure detection, repair detection, outbound interface selection and source address selection must continue to operate as before.
10. The documented IPMP configuration files (`/etc/hostname.`*if* files) must continue to operate as before.

All of these requirements can be met by changing the core IPMP architecture such that each IPMP group is modeled as an IP interface (rather than as a collection of loosely federated IP interfaces), and introducing a new tool to easily and concisely obtain the core configuration and current state of each group.

Note that because each IPMP group will be modeled as an IP interface, Clearview's list of interface requirements must be followed[11] as well. This adds one additional requirement: vanity naming. That is, **the name of the IP interface representing each IPMP group must be configurable by the administrator.**

The remainder of this document describes the proposed new model, and contrasts it to the existing model where possible. The following section provides an overview of how core IPMP concepts and operations will be represented in the new model. The next two sections discuss the administrative and programmatic impact of the new model. Finally, the final section revisits the requirements and summarizes how they will be satisfied by the proposed rearchitecture. Note that since this is a high-level design document, a detailed design of each specific change will not be provided.

[11] http://opensolaris.org/os/project/clearview/overview.txt

3 IPMP Rearchitecture: Basic Operation

3.1 IPMP Network Interface

As previously discussed, the lion's share of the problems with IPMP stem from not treating each IPMP group as its own IP interface. As an example, a typical two-interface IPMP group today looks like:

```
ce0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 2
       inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
       groupname a
ce0:1: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
       inet 129.146.17.55 netmask ffffff00 broadcast 129.146.17.255
ce1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 3
       inet 129.146.17.58 netmask ffffff00 broadcast 129.146.17.255
       groupname a
ce1:1: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 3
         inet 129.146.17.57 netmask ffffff00 broadcast 129.146.17.255
```

The above output shows `ce0` and `ce1` with test addresses, and `ce0:1` and `ce1:1` with data addresses[12]. If `ce0` subsequently fails, the data address on `ce0:1` will be migrated to `ce1:2`, but the test address will remain on `ce0` so that the interface can continue to be probed for repair.

In the future, this will instead look like:

```
ce0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 2
         inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
         groupname a
ce1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 3
         inet 129.146.17.58 netmask ffffff00 broadcast 129.146.17.255
         groupname a
ipmp0: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
         inet 129.146.17.55 netmask ffffff00 broadcast 129.146.17.255
         groupname a
ipmp0:1: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
         inet 129.146.17.57 netmask ffffff00 broadcast 129.146.17.255
```

That is, all of the IP data addresses associated with the IPMP group will instead be hosted on an IPMP IP interface, such as `ipmp0`[13]. With this new model, **data addresses will no longer be associated with any specific physical underlying interface, but instead will belong to the IPMP group as a whole**. As will become clear, this addresses many outstanding problems and vastly simplifies the implementation.

There will be a one-to-one correspondence between IPMP groups and IPMP interfaces. That is, each IPMP group will have exactly one IPMP interface. By default, each IPMP interface will be named `ipmp`*N*, but administrators will be encouraged to specify a name of their choosing, as described in section 4.1.5. Since an IPMP interface's name will not be fixed, the system will set a new `IPMP` flag on all IPMP interfaces to indicate that the interface has special properties and semantics, as detailed throughout this document.

For clarity, this document uses the `ipmp`*N* naming convention for IPMP interfaces unless another format is necessary to explain a given topic.

[12] Where feasible, this configuration will be used for examples throughout the document.

[13] In the new model, had the data addresses been hosted on the physical interfaces, then the test addresses would have appeared on `ce0:1` and `ce1:1`, with `ce0` and `ce1` containing `0.0.0.0` placeholder addresses. On a configuration without test addresses, there would *only* be `0.0.0.0` placeholder addresses.

3.2 Outbound Load-Spreading

Currently, the IP module load-spreads outbound traffic for each IP destination address across any *active* interface in the group, where *active* is defined as an interface that is not marked `FAILED`, `INACTIVE`, or `OFFLINE`, and has at least one `UP` address[14]. This will be unchanged in the new model.

3.3 Source Address Selection

Currently, if the application does not bind to a specific source address, the IP module will select a source address from the set of addresses in the IPMP group. The purpose of varying the source address is to affect inbound load-spreading. Specifically, in the current model, all addresses hosted on a given interface are associated with that interface's hardware address. Thus, currently, if one connection uses a source address on `ce0`, and another a source address on `ce1`, then each connection will use a different interface for inbound traffic. It's important to note that **choice of outbound interface is independent from choice of source address**. Thus, it is quite common to have a connection which sends packets through `ce0`, using a source address hosted on `ce1`, and thus receives the response on `ce1` (this is necessary to spread outbound load when the set of available source addresses is smaller than the set of available outbound interfaces).

Since data addresses will now be hosted on the IPMP interface, this will change slightly in the new model. Specifically, the IP module itself will have the responsibility of associating each available IP address in the IPMP group with a specific active underlying interface, and ensuring that the set of addresses remain evenly distributed over the active interfaces. In particular, source addresses may need to be redistributed as available addresses are added or removed from the group, or as a result of an active interface becoming unusable or `INACTIVE`.

As before, all packets to a specific destination will use the same source address. Both the IPv4 and IPv6 source address selection algorithms will remain unchanged. The IPv6 algorithm is described in `inet(7P)`. The IPv4 algorithm is undocumented, but will remain:

1. Non-deprecated source addresses on the same subnet as the next-hop destination[15].
2. Non-deprecated source addresses on a different subnet.
3. Deprecated source addresses on the same subnet as the next-hop destination.
4. Deprecated source addresses on a different subnet.

Also as before, if after performing the algorithm more than one IP address remains, the IP module will "randomly"[16] pick from the remaining set of IP addresses.

An implementation artifact currently allows test addresses to be considered as source addresses. However, since all IPMP groups are required to have at least one data address, they will never be chosen by the above algorithm (unless all data addresses are marked `deprecated`, in which case the current code erroneously selects a test address). In the new model, the test addresses will *never* be considered because underlying interfaces will be ignored during source address selection. However, since the administrator may still choose to mark some of the IPMP data addresses as deprecated, the source address selection algorithm will continue to prefer non-deprecated addresses to meet the guarantees stated in `ifconfig(1M)`.

As before, only `UP` non-zero addresses in the same zone will be considered.

[14] `UP` is an address property rather than an interface property, but the IP module has always assumed that an interface must have at least one `UP` address before it can send or receive packets.

[15] *next-hop destination* is the next-hop router if the destination is off-link, or the destination itself if it is on-link.

[16] We use the term loosely as the implementation need not actually use an RNG; anything that varies the IP source address will suffice.

3.4 Interface Failure and Repair

Currently, when `in.mpathd` detects that an interface can no longer be used to access the network, it marks it `FAILED`, *and* triggers a failover operation to move the addresses to another functioning interface in the group. However, with the IPMP interface, the address migration step will no longer be necessary. For instance, if `in.mpathd` detects that `ce0` has failed, it will just need to mark `ce0` as `FAILED`[17]:

```
ce0: flags=19040843<UP,BROADCAST,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,FAILED> mtu 1500 index 2
        inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ce1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 3
        inet 129.146.17.58 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
        inet 129.146.17.55 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0:1: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
        inet 129.146.17.57 netmask ffffff00 broadcast 129.146.17.255
```

If `ce1` were to then also fail, it would also be marked `FAILED` (and of course no migration would occur). In addition, the group itself will be marked `FAILED`, making it clear that the entire IPMP group is unusable (note that the new `ipmpstat` tool discussed in section 4.2 will bring more appreciable administrative improvements, including fine-grained IPMP group and interface state information using established FMA terminology.)

```
ce0: flags=19040843<UP,BROADCAST,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,FAILED> mtu 1500 index 2
        inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ce1: flags=9040843<UP,BROADCAST,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,FAILED> mtu 1500 index 3
        inet 129.146.17.58 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0: flags=801000843<UP,BROADCAST,MULTICAST,IPv4,IPMP,FAILED> mtu 1500 index 4
        inet 129.146.17.55 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0:1 flags=801000843<UP,BROADCAST,MULTICAST,IPv4,IPMP,FAILED> mtu 1500 index 4
        inet 129.146.17.57 netmask ffffff00 broadcast 129.146.17.255
```

As one would suspect, once `in.mpathd` detects that the interface can be used again, it will only need to clear the appropriate `FAILED` flags – as with interface failure, address migration will no longer be needed.

3.5 Interface Offline and Undo-Offline

Currently, data addresses are also migrated to another functioning interface as part of an administratively requested offline or undo-offline operation (if no interface is functioning, the operation fails). Following the migration, the `OFFLINE` flag is set to indicate that the interface is now unusable, and any test addresses are brought down to complete the offline operation (the opposite is done on an undo-offline).

Again, since data addresses will no longer be hosted on the physical interfaces, migration step will no longer be necessary. Instead, an offline operation will consist of setting the `OFFLINE` flag and bringing down the test addresses[18]. For instance, if `ce0` in the previous example is offlined, the resulting configuration will be:

[17] If the interface was active, an `INACTIVE` interface in the group may also be activated – see sections 3.7 and 3.8.
[18] As with interface failure, an `INACTIVE` interface in the group may also be activated – see sections 3.7 and 3.8.

```
ce0: flags=9040843<BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER, OFFLINE> mtu 1500 index 2
        inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ce1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu 1500 index 3
        inet 129.146.17.58 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
        inet 129.146.17.55 netmask ffffff00 broadcast 129.146.17.255
        groupname a
ipmp0:1: flags=801000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500 index 4
        inet 129.146.17.57 netmask ffffff00 broadcast 129.146.17.255
```

3.6 Dynamic Reconfiguration

Dynamic Reconfiguration support is provided by the Solaris Reconfiguration Coordination Manager (RCM) framework and built upon the aforementioned offline and undo-offline operations. Specifically, when removal of a network card is requested, RCM first offlines the interface in the manner previously described. Next, any test addresses assigned to the interface are removed, and finally the interface is unplumbed from the system.

A card insertion operation does the reverse: RCM replumbs the interface, rebuilds the IP configuration of the interface from the system's `/etc/hostname.`*if* file, and then finally performs an undo-offline operation to restore the interface to its previous state.

While this sounds straightforward, the current implementation is complicated significantly by the need to handle data address migration. In particular, because the kernel uses interface indices to track which data addresses belong to which interfaces, and because the interface index of the inserted interface may be different than the removed interface, RCM must interact with `in.mpathd` and the kernel to change the "original" interface index of each data address to the inserted interface's index. In addition, the RCM code duplicates a significant piece of the `ifconfig` command-line interpreter in order to parse the `/etc/hostname.`*if* file.

This complexity will be eliminated in the new model. In particular, since the card being replaced will no longer host data addresses, there will be no need to reassociate any addresses with the interface index of the replacement card. This will allow all of the interface index tracking logic to be removed from the kernel and IPMP utilities, along with the removal of the undocumented `SIOCGLIFOINDEX` and `SIOCSLIFOINDEX` ioctls, and the private `MI_SETOINDEX` message interface between `in.mpathd` and RCM. In addition, the hefty parsing engine inside RCM will be able to be replaced with a simple call to the existing `ifparse(1M)` command[19] to obtain the IPMP-related information from `/etc/hostname.`*if*. Finally, to allow DHCP-managed test addresses to work with DR, RCM will be enhanced to check for `/etc/dhcp.`*if*, and if found, use `ifconfig` to start DHCP on the interface. To mirror boot functionality, the contents of `/etc/dhcp.`*if* will be passed as command-line arguments to `ifconfig`.

The RCM code also has special handling for undoing an offline operation for a card that was never actually removed. This code is usually exercised to restore the system's configuration when a remove request fails part-way through. In this case, the "current" IP interface configuration is restored from a cache inside RCM, rather than being rebuilt from the persistent IP interface configuration in the `/etc/hostname.`*if* file. RCM builds creates its cache using the undocumented `ifconfig configinfo` subcommand. While this is questionable, this will not be changed since it is orthogonal to the IPMP Rearchitecture and since customers may depend on the current behavior.

[19]The `ifparse` command post-dates the original RCM work, which is why it wasn't used originally.

3.7 `STANDBY` Interaction

Currently, interfaces can optionally be marked `STANDBY` by an administrator. This flag indicates that the interface must not be used unless some other interface in the group fails. Thus, when an interface is marked `STANDBY`, the kernel also sets `INACTIVE` to indicate that the interface must not be used for outbound traffic, and `in.mpathd` moves all data addresses to another interface so that the interface will not be used for inbound traffic. If an active interface subsequently fails, `in.mpathd` will prefer to failover its addresses to another `INACTIVE STANDBY` interface in the group. As part of the failover operation, the IP module clears the `INACTIVE` flag, allowing the interface to be used. When the failed interface is eventually repaired, the IP module restores the `INACTIVE` flag as part of the failback operation.

Since the concept of address failover and failback will no longer exist, the behavior of a `STANDBY` interface will change slightly in the new model. Specifically, in the new model, `in.mpathd` will be responsible for updating the `INACTIVE` flag to activate `STANDBY` interfaces as necessary. Thus, if there is an `INACTIVE STANDBY` interface in the group when an *active* interface fails or is taken offline, instead of triggering a failover to the `INACTIVE STANDBY`, `in.mpathd` will simply clear its `INACTIVE` flag, allowing the `STANDBY` to be used.

An interface repair or undo-online may allow an active `STANDBY` to be made `INACTIVE` again. Specifically, if an interface not marked `STANDBY` is repaired or brought back online, and that interface causes there to be more active interfaces than interfaces not marked `STANDBY`, `in.mpathd` will mark one of the active `STANDBY` interfaces `INACTIVE` again. Similarly, if a `STANDBY` interface is repaired or brought back online, and there are more active interfaces than interfaces not marked `STANDBY`, then the repaired interface will be marked `INACTIVE`.

3.8 `FAILBACK=no` Interaction

The `FAILBACK /etc/default/mpathd` tunable gives administrators another way to control IPMP's interface selection. Specifically, as long as there are other active interfaces in the group, setting `FAILBACK=no` instructs the system to keep a previously-failed interface unused even after repair. However, if another interface subsequently fails, then the unused interface will again be used. This behavior is desirable in certain environments where changing the binding of addresses to incoming interfaces must be kept to a bare minimum – for instance, in order to avoid needless invalidation of other hosts' ARP caches.

The support for this functionality in the existing model has been a challenge (see CRs 5084072 and 4796820 – and the undocumented `SIOCSIPMPFAILBACK` ioctl). In the new model, support for this will be straightforward: on interface repair, `in.mpathd` will mark it `INACTIVE` as long as another interface in the group is also functioning. Thus, in the common case, the repaired interface will remain unused for data traffic. If another active interface in the group subsequently fails, and there are no interfaces marked `INACTIVE STANDBY`, then `in.mpathd` will search for interfaces marked just `INACTIVE`, and clear the `INACTIVE` flag on the first one found. Since kernel involvement will no longer be needed, the `SIOCSIPMPFAILBACK` ioctl will be removed.

Note that `FAILBACK=no` has a subtle interaction with `STANDBY`. Specifically, if `FAILBACK=no` causes a repaired interface to be marked `INACTIVE` (as described above), then by definition the number of active interfaces in the group will remain unchanged. Thus, the repair will not cause an active `STANDBY` interface to be marked `INACTIVE`.

3.9 IPMP Interface Flags

As is clear from the preceding discussion, a wide range of IPMP attributes are currently managed through interface flags, visible through `ifconfig(1M)`:

flag	meaning
`STANDBY`	Interface is administratively assigned to only be used if another interface fails
`INACTIVE`	Interface is functioning, but not currently being used according to administrative policy
`OFFLINE`	Interface is an unknown state and unavailable for use
`FAILED`	Interface is not functioning and must not be used
`RUNNING`	If not set, interface is not functioning and must not be used
`NOFAILOVER`	Address is administratively assigned as a test address and must not be used
`DEPRECATED`	Address is administratively not preferred for source address selection
`UP`	If not set, address is administratively unavailable for source address selection

Although the semantics of all of these flags must be supported for backward compatibility, administrative and architectural improvements will greatly reduce the programmatic and administrative reliance on these flags. Most notably, `ipmpstat -i` (section 4.2.3) will allow the administrator to easily see how each underlying interface will be used by IPMP without having to consider the flags, and only the following subset of the above flags will be visible on an IPMP interface:

flag	meaning
`FAILED`	Interface is not functioning and must not be used
`RUNNING`	If not set, interface is not functioning and must not be used
`DEPRECATED`	Address is administratively not preferred for source address selection
`UP`	If not set, address is administratively unavailable for source address selection

Of these, `FAILED` and `RUNNING` are mutually exclusive on the IPMP interface, and one will or the other will always be set (strictly speaking, `FAILED` is not necessary, but will be preserved for symmetry with the way underlying interfaces are managed). Further, `DEPRECATED` will only be set if explicitly enabled on an address (e.g., through `ifconfig ipmp0:1 deprecated`). Thus, in practice, only the `RUNNING` and `UP` flags will prove relevant to understanding the behavior of an IPMP interface and its set of addresses. Since both of these flags have existed for decades and are well-understood, they will be properly handled by administrators and applications alike.

While all of the flags have been preserved, a few backward-compatible revisions to their semantics have been made for administrative simplicity or for correctness. Specifically, `NOFAILOVER` will now always imply `DEPRECATED` (section 4.1.3), `FAILED` will now always imply `~RUNNING` on the IPMP interface (section 5.3), and `INACTIVE` will be controlled by `in.mpathd` (section 3.7).

3.10 IPMP Interface MTU

Currently, each interface in an IPMP group maintains its own MTU, as does each address on each interface. When an address migrates between interfaces, the MTU of the address is checked against the MTU of the destination interface. If the interface's MTU is smaller, the original MTU of the address is saved, and the MTU of the address is lowered to match the interface's MTU. If the address later migrates again (either as a result failback, or another failure), the original MTU is consulted, and the MTU of the address is increased, if possible.

In the new model, the each interface in an IPMP group will still maintain its own MTU, but that MTU will only directly affect probe packets sent over the interface. The MTU of data packets will be controlled by the *IPMP interface's MTU*, which will be set to the smallest MTU supported by *all* of the interfaces currently in the group. Thus, if an IPMP group is comprised of two interfaces, one with a 1400 byte MTU, and another with a 1500 byte MTU, then the IPMP interface will have a 1400 byte MTU. If the first interface is removed from the group, the MTU will be automatically raised to 1500 bytes. As a degenerate case, an IPMP interface with no underlying interfaces will have the minimum MTU supported by the IP module (currently 68 bytes for IPv4, and 1280 bytes for IPv6).

Because applications cannot tie themselves to a specific interface in an IPMP group, this change will not have any affect on applications. That is, because any packet could be sent over any interface in the group, an application already had to assume that the MTU was the smallest supported by all interfaces in the group.

3.11 Failure and Repair Detection

Currently, `in.mpathd` can use *probe-based* failure detection, *link-based* failure detection, or both forms of failure detection to track the health of an interface. *Probe-based* failure detection actively probes the send and receive paths of each network interface using a method similar to `ping(1M)`. Since performing probe-based failure detection on an interface requires a test address to be configured on it, the administrator can deactivate probe-based failure detection by *not* configuring a test address.

In contrast, *link-based* failure detection passively monitors the link status of the interface, as reported by the underlying driver. This requires no explicit configuration and provides instantaneous failure detection, but requires support from each underlying network driver, and only monitors the health of the link itself, rather than the entire send and receive path of the network interface. Unfortunately, it is not currently possible to administratively tell if an interface supports link-based failure detection, limiting the adoption of the feature.

Because each form of failure detection has its strengths and weaknesses, both will remain. However, the administrator will now be able to use `ipmpstat` to easily see which failure detection method is enabled on a given interface, including whether the underlying interface supports link-based failure detection. To preserve compatibility, for interfaces that support link-based failure detection, the `RUNNING` flag will be set when the link is up, and clear when the link is down.

3.12 Responsibilities of `in.mpathd` and the IP module

Currently, the IPMP implementation is split between two components: the `in.mpathd` daemon, and the kernel's IP module. This split will remain, but some of the responsibilities will change.

Specifically, `in.mpathd` will remain responsible for failure and repair detection, including the selection of probe targets, the sending and receiving of ICMP probes, and the tracking of each interface's link status. Additionally, the kernel will remain responsible for outbound load-spreading and source address selection. However, `in.mpathd` will no longer need to perform address migration, simplifying its implementation. Since the inbound interface that will receive a packet is dictated by the IP interface hosting the IP address the packet is sent to, this change will have a subtle impact on inbound load-spreading. Specifically, since `in.mpathd` will no longer perform address migration, it will no longer have any control over how the IP addresses are distributed across the IP interfaces in the group, and the IP module will assume complete responsibility for inbound load-spreading. As part of transitioning that responsibility, `in.mpathd` will be responsible for marking interfaces `INACTIVE`, rather than the IP module.

3.13 Multicast Interaction

Currently, because there is no concept of an IPMP interface, all multicast membership information is associated with the underlying interfaces in a group (or the addresses hosted on those interfaces). This introduces a significant amount of implementation complexity, since the original multicast membership associations must be maintained across interface failure and repair. Moreover, nothing prevents a multicast group from being joined on more than one underlying interface in the same IPMP group, leading to duplicate traffic. Further, established technologies such as IGMP snooping have not been well-supported in an IPMP environment[20].

In the future, multicast memberships will be associated with the IPMP interface[21]. In turn, the IP module will nominate one active interface to handle *all* multicast traffic for the IPMP group (provided an active interface exists)[22]. Since the memberships themselves will remain tethered to the IPMP interface, and since only one interface in the group will handle all multicast traffic, the handling is simplified considerably. Specifically, if the nominated interface fails, then another active interface will be nominated (if one exists), and memberships will be transferred to it. As part of the nomination process, IGMP or MLD messages will be generated to alert any IGMP or MLD snoopers. Since the memberships themselves are tied to the IPMP interface, no action is needed upon interface repair (unless it is the first active interface in the IPMP group).

Another problem concerns the handling of multicast ICMP echo requests. Specifically, if the `ip_respond_to_echo_multicast ndd` tunable is enabled, then an `all-hosts` multicast ICMP echo request will be answered using the first `UP` address on the interface, even if it is a test address. This is problematic because it allows other hosts to accidentally discover and potentially use the test address, which is not highly available. In the future, `all-hosts` multicast ICMP echo requests will be answered using an arbitrary data address in the IPMP group, fixing the problem.

3.14 Broadcast Interaction

Currently, in order to avoid duplicate broadcast traffic, an active interface in the group is nominated to receive all broadcast traffic for the IPMP group. If the interface subsequently fails, the nomination process is repeated. In the future, the interface nominated to handle multicast traffic will also be used to receive broadcast traffic. However, administratively, the two concepts will remain decoupled (see section 4.2.3).

As with multicast, broadcast ICMP echo requests are currently mishandled. Specifically, if the `ip_respond_to_echo_broadcast ndd` tunable is enabled, then a broadcast ICMP echo request will be answered using the first `UP` address on the interface, even if it is a test address. In the future, broadcast ICMP echo requests will be answered using an arbitrary data address in the IPMP group, fixing the problem.

3.15 Data-link Interaction

As the name implies, an IPMP group exists at the IP layer. Below this layer, an IPMP group is simply a collection of individual data-link interfaces that have the same "interface type" as per the `IFT_*` values in `<net/if_types.h>`. That is, each data-link interface in a group maintains its own

[20] See bug 6220619.

[21] There is one exception: to support neighbor discovery for IPv6 test addresses, the IP module will internally create solicited-node multicast memberships for each IPv6 test address on the underlying IP interface hosting each address. Since test addresses remain tethered to their underlying IP interface, the implementation remains straightforward.

[22] This also means that multicast traffic will not be load-spread. If this becomes a problem, we could enhance the implementation to load-spread by multicast group – at the cost of code complexity.

identity, including its own DLPI state machine, hardware address and link speed. Because each data-link interface in an IPMP group maintains its own identity, it is not semantically feasible to provide data-link access to the IPMP interface as a collective whole, and thus no such access will be provided.

While this may seem like a significant limitation, IP-layer support for packet monitoring (section 4.11) tools, packet filtering (section 4.12), and DHCP (section 4.13), will significantly lessen its impact. Moreover, data-link access to an IPMP group as a whole is not currently provided, so this will not be a regression. Further, this rearchitecture will not add any new barriers to implementing such an abstraction in the future. Finally, although it not part of this rearchitecture, restricted data-link access could be added on a case-by-case basis to address specific limitations, such as packet monitoring of the “group” at the data-link layer.

Of course, data-link access to each underlying physical interface in a group will still be provided.

3.16 Naming Considerations

Currently, each IPMP group has two sets of names: the name of the group itself (e.g., `a`), and the names of the interfaces in the group (e.g., `ce0` or `hme0`). With the IPMP rearchitecture, a third naming abstraction will be introduced: the IPMP interface name (e.g., `ipmp0`).

Since each IPMP interface name represents an IPMP group, it would have been ideal to use the IPMP group name itself as the interface name. Unfortunately, this is not possible because the naming rules associated with interface names are more strict than those associated with group names (most notably, an interface name *must* end in a number – but there are other differences as well). While it would be possible to translate group names into a format acceptable as an interface name, we decided it was simpler to implement parallel namespaces, but to provide an administrative model that encouraged administrators to use the same name in both (e.g., by defaulting to a group name that matched the chosen IPMP interface name).

While both namespaces will continue to exist, the IPMP interface name will become the core administrative naming abstraction for IPMP. This is necessary in order to allow IPMP to be smoothly integrated with other core networking abstractions, such as the routing and ARP tables, which operate exclusively in terms of IP interface names. By encouraging administrators to choose the same name for their IPMP interface name and IPMP group name, we will shield them from this subtlety, and eventually permit group names to be phased out as a separate namespace.

While the `dladm` enhancements provided by Clearview Vanity Naming[23] allow one to rename the data-links associated with the underlying interfaces in an IPMP group, the IPMP interface does not exist at the data-link layer and thus cannot be renamed through that facility. As such, an IPMP-specific mechanism, described in section 4.1.5, will be provided through `ifconfig` to enable the IPMP interface name to be set. While the asymmetry of this is unfortunate, it is unavoidable given that IPMP simply does not exist below the IP layer.

3.17 Sun Cluster Considerations

Sun Cluster makes use of IPMP to provide high-availability between nodes of a cluster. In addition to supporting the typical IPMP configurations, Sun Cluster also uses a special configuration called *IPMP Singleton*. In this configuration, only a single network interface is placed into an IPMP group. Because there is only one interface in the IPMP group, any addresses on the interface cannot migrate to another interface, and thus any test addresses can also serve as IPMP data

[23]See PSARC/2006/499 and http://opensolaris.org/os/project/clearview/uv/

addresses. IPMP Singleton is used by Sun Cluster to monitor interface failures and then trigger a higher-level failover operation, such as transitioning tasks to another node in the cluster, without requiring the explicit configuration of test addresses.

Currently, addresses in an IPMP Singleton group that are not marked `NOFAILOVER` will only be used as data addresses, whereas those marked `NOFAILOVER` will be used as both data and test addresses. The default configuration for Sun Cluster is to configure a single address on each IP interface in a singleton group, and mark it `NOFAILOVER` so that it can be used for both data and test traffic.

Since the new model requires that all data addresses be on an IPMP interface and all test addresses be on an underlying interface, and an address can only be on one interface, allowing a single IP address to be both a data address and a test address is incompatible with the model. As such, this will no longer be supported, and an explicit test address must be configured to perform probe-based failure detection. However, given that link-based failure detection is now widespread and sufficient for many Sun Cluster configurations, and that a dedicated test address can always be configured if necessary, we expect this limitation will have minimal impact.

The Sun Cluster software also has internal Sun contracts to use the IP interface flags `IFF_FAILED`, `IFF_INACTIVE`, `IFF_STANDBY` and `IFF_NOFAILOVER`, and the IPMP ioctls `SIOCGLIFGROUPNAME` and `SIOCSLIFGROUPNAME`. For the most part, these interfaces are unaffected by the IPMP Rearchitecture. However, as per section 5.17, `SIOCSLIFGROUPNAME` will enforce several new restrictions which will complicate direct use by Sun Cluster. As such, we will work with the Sun Cluster team to have the impacted software use `ifconfig` to change the group name[24].

3.18 IPMP Anonymous Group Support

Along similar lines, IPMP also supports an obscure *Anonymous Group* feature. This feature is disabled by default, but can be enabled by setting `TRACK_INTERFACES_ONLY_WITH_GROUPS` to `no` in `/etc/default/mpathd`. When this feature is enabled, `in.mpathd` tracks the health of *all* network interfaces in the system, regardless of whether they belong to an IPMP group. Just like the IPMP Singleton case, each interface tracked in this manner is alone in a group and thus no migration is possible, allowing a data address on the interface to also be used as a test address. However, unlike IPMP Singleton, these interfaces are never placed into an IPMP group, and thus no restriction is needed on whether any of their addresses are marked `NOFAILOVER`. That is, from IPMP's perspective, these interfaces are not part of *any* IPMP group and thus will be visible to applications regardless of the setting of `NOFAILOVER`.

3.19 Interaction with Other Network Availability Technologies

As previously mentioned, Solaris currently supports a range of network availability technologies. The proposed rearchitecture will slightly modify their interaction and integration with IPMP, as summarized in the following subsections.

3.19.1 Interaction with 802.3ad and Sun Trunking

As is currently the case, it will remain possible to layer IPMP on top of 802.3ad aggregations or Sun Trunking trunks. However, just as IPMP is unaware of the underlying speed of each underlying interface in an IPMP group, it is also unaware of the capacities of each aggregation or trunk, and

[24]Unfortunately, all IP interface configuration logic is hardwired into `ifconfig` rather than factored out into a library, precluding a C API. Properly refactoring `ifconfig` is outside the scope of this work.

thus is unable to optimize the load-spreading across them. This will remain the case in the new model, though programming interfaces can be added in the future to allow each underlying link's capacity to be obtained and monitored, if there is sufficient customer interest.

Since the IPMP interface will not exist at the DLPI layer, it will not be possible to place an IPMP interface into an aggregation or trunk.

3.19.2 Interaction with CGTP

Currently, IPMP is mutually exclusive with CGTP: the CGTP virtual interface cannot be placed into an IPMP group, and none of the physical interfaces used by CGTP can belong to an IPMP group. This behavior is required because CGTP must guarantee that each packet sent using a particular source address is always sent over a specific interface, which is directly at odds with the outbound load-spreading feature of IPMP. For this reason, this exclusivity will be preserved, and attempts to create CGTP routes to the IPMP interface will be rejected by the kernel.

3.19.3 Interaction with SCTP

As is currently the case, SCTP-based applications will be able to use addresses that belong to an IPMP group. Since all SCTP applications are IP-based, they will transparently benefit from the proposed changes.

3.19.4 Interaction with OSPF-MP

Currently, the system does not allow OSPF-MP to be used with IPMP. Specifically, because `vni(7D)` is virtual and does not represent a single link, it makes no sense to place it into an IPMP group. Further, while it is semantically reasonable to allow `usesrc` to be applied consistently to all interfaces in an IPMP group (see below), there is currently no administrative way to enforce this usage. That is, with the current implementation of IPMP, it is not possible to ensure that the value of `usesrc` is identical for each interface in an IPMP group – and indeed, it is quite likely `usesrc` will be misused.

The introduction of the IPMP group interface will fix the usability problems with `usesrc`. Specifically, by allowing `usesrc` to be set on an IPMP interface, we could ensure that each IPMP group had only one `usesrc` value, and that the value applied to all interfaces in the group. When set, this would override the group's source address selection algorithm. Thus, the addresses hosted on each IPMP group interface would be used to route packets to the addresses hosted on the appropriate `vni` interface. Further, the source address would always be an address on the selected `vni` interface.

Consider a configuration with two IPMP groups together in an OSPF-MP group, where each IPMP group had two underlying interfaces. This would provide twice the network utilization versus placing all four interfaces into an OSPF-MP group[25] (since both interfaces in the selected IPMP group will be able to send and receive traffic) and require two fewer subnets (since each pair of interfaces in an IPMP group would be on the same subnet). Further, this could provide end-to-end availability that would not be available by simply placing all four interfaces into an IPMP group.

While this is an intriguing opportunity afforded by the new architecture, we would like to gauge customer need before implementing such a feature. Moreover, once ECMP support is available,

[25] This is because ECMP is not currently available on Solaris.

OSPF-MP will likely provide a much simpler and more compelling story for end-to-end availability. As such, we will not yet allow `usesrc` to be applied to an IPMP group. However, this restriction may be lifted in the future.

3.20 SMF Considerations

Currently, IPMP is not tied into the SMF framework (though Sun Cluster provides a restarter for `in.mpathd` as an interim solution). While integrating IPMP into the SMF is desirable, it is not necessary to satisfy the core requirements specified in section 2. Further, there are other complicating factors:

- IPMP configuration is not cleanly separable from other IP interface configuration currently done via `/etc/hostname.`*if*. Thus, migration of IPMP configuration should be done as part of a broader effort to migrate `/etc/hostname.`*if* information to the SMF.
- The `in.mpathd` daemon itself is an awkward fit for the SMF model. In particular, it is usually started on-demand by `ifconfig` when an IP interface is placed in a group (and never self-terminates). However, it needs to be started explicitly (typically by the boot scripts) if IPMP anonymous groups are being used and no named IPMP groups are configured.
- Some third party applications, such as Symantec VCS, are known to explicitly start and stop `in.mpathd`. Since that will no longer work as expected once `in.mpathd` is under SMF control, we will need to work with them to understand their needs and provide an alternative (likely simply a restart of the SMF service).

Therefore, this proposal will not integrate IPMP into the SMF. However, the introduction of the IPMP group interface will simplify the future representation of IPMP into SMF, since a special construct will not be required to represent IPMP groups. For instance, many of the existing tunables in `/etc/defaut/mpathd` should be per-IPMP-group rather than global; tying these to IP interface properties which could be set on each IPMP group interface would provide more flexibility and improve administrative consistency. Finally, this proposal is careful not to introduce any new administrative commands or tunables which might complicate or conflict with future integration of IPMP into the SMF.

4 IPMP Rearchitecture: Administrative Model

4.1 Configuration with `ifconfig`

4.1.1 Adding and Removing Data Addresses

Since the IPMP interface will become the core administrative abstraction for IPMP, we will recommend that administrators directly configure addresses on the IPMP interface. For instance, to add an additional address to `ipmp0`, one will do:

```
# ifconfig ipmp0 addif 129.146.17.59 up
```

To maintain compatibility with previous releases of Solaris, data addresses will also be able to be added to the underlying physical interface:

```
# ifconfig ce0 addif 129.146.17.59 up
```

However, once the address is brought up, it will actually appear in the next available logical interface on the IPMP interface associated with `ce0` – e.g., `ipmp0:2`. Note that since the address will not usable prior to bringing it up, there will be no need to implement complex address migration to support this operation – the address will simply be changed to be associated with the IPMP interface and brought up.

Once a data address has been associated with an IPMP interface, its interface name binding (e.g., `ipmp0:2`) will never change except by explicit administrative action. To remove a given data address from a group, the administrator will be able to specify either the logical interface or the address. That is, the following will be equivalent:

```
# ifconfig ipmp0 removeif 129.146.17.59
# ifconfig ipmp0:2 unplumb
```

Alternatively, marking an address `down` will temporarily remove it from system use:

```
# ifconfig ipmp0:2 down
```

4.1.2 Configuring Data Addresses with DHCP

As with physical network interfaces, it will be possible to use DHCP to obtain addresses on the IPMP interface:

```
# ifconfig ipmp0:1 plumb
# ifconfig ipmp0:1 dhcp start primary
```

A summary of the changes to DHCP is provided in section 4.13.

4.1.3 Adding and Removing Test Addresses

Since test addresses remain associated with each underlying interface, they will still be configured on the physical interface:

```
# ifconfig ce0 129.146.17.56 -failover deprecated up
```

However, since no test addresses should ever be used as a source address, setting `-failover` will now imply `deprecated`. Thus, this will also configure a test address:

```
# ifconfig ce0 129.146.17.56 -failover up
```

Test addresses will be prohibited on IPMP interfaces, and attempting to set `-failover` on an address hosted on an IPMP interface will fail.

As is currently the case, test addresses will be able to be removed using the `unplumb` or `removeif` `ifconfig` subcommands. One will also be able to prevent the use of a test address by marking it `down`. It will also be possible to convert the test address to a data address via the `failover` subcommand, though this will rarely be useful in practice[26]. Of course, if `failover` is issued on an `UP` address, the address will migrate from the physical interface to the appropriate IPMP interface.

4.1.4 Adding and Removing Underlying Interfaces

As is currently the case, underlying interfaces will be able to be added or removed from a group using the `group` subcommand. For instance, to replace `ce1` with `ce2` in group `a`:

```
# ifconfig ce2 group a
# ifconfig ce1 group ""
```

While we currently document a number of requirements that need to be met in order to place an interface into a group, none of these are currently enforced by the system when an interface is placed into a group. In the future, several of these requirements will be enforced by the kernel, resulting in diagnostic messages being printed by `ifconfig`. Specifically:

- The interface type must be the same as any other interfaces in the group (e.g., Ethernet).
- The interface type must be broadcast-capable (e.g., no point-to-point interfaces)[27].
- The interface must not have any data addresses that are controlled by other applications (currently `dhcpagent(1M)` and `in.ndpd(1M)`), as migration of those addresses may confuse those applications. (Since there is no convenient way to control `in.ndpd(1M)`, `ifconfig` will bring down the link-local IPv6 address which `in.ndpd(1M)` will use as a hint to tear down its addresses on that IPv6 interface. However, if `in.ndpd(1M)` does not tear down its addresses within a short period of a time, a diagnostic message will still be printed.)

[26] As per section 5.3, changing `IFF_NOFAILOVER` on an `IFF_UP` address will not be permitted. Accordingly, `ifconfig` will bring the address `down` first, and back `up` afterwards; the `up` will trigger migration.

[27] In principle, point-to-point could be made to work, but only if the peer was also IPMP-aware. Moreover, Multilink PPP already fills this need.

Note that as before, every active interface in a group must have a unique hardware address. However, since a hardware address may become a duplicate *after* an interface has joined a group, this will not become a requirement to join a group. See section 4.1.10 for details.

To meet the requirement that `UP` addresses never migrate, the kernel will fail requests to place interfaces with `UP` addresses into a group. However, `ifconfig` will automatically bring any `UP` addresses `down` prior to placing an interface into a group, and bring them back `up` afterwards, triggering migration as appropriate[28]. As such, administrative compatibility will be preserved.

4.1.5 Creating IPMP Interfaces

IPMP interfaces will be able to be created using either *implicit creation* or *explicit creation*. Explicit creation will provide optimal control over the configuration of the IPMP interface, at the expense of requiring the administrator to learn a new administrative procedure. In contrast, implicit IPMP interface creation will require no new administrative knowledge, but will limit the administrative control over the name of the IPMP interface and interfaces it will be associated with. While our documentation (e.g., on `docs.sun.com`) will describe both, we will encourage administrators to use explicit creation both because it provides optimal control and because it makes the semantic split between the IPMP and underlying IP interfaces apparent.

Implicit IPMP Interface Creation

Implicit creation will occur by placing a physical interface into an IPMP group that does not currently exist. In this case, the system will create the lowest available `ipmp`N interface and migrate any `UP` addresses from the physical interface to the IPMP interface. For instance, if `ipmp0` already exists as group `a`, then `ifconfig ce0 group b` will cause the system to create `ipmp1`.

Unfortunately, because data addresses are automatically migrated when brought `UP`, the following misconfiguration will now always lead to unexpected results:

```
# ifconfig ce0 group b 10.0.0.1 up
# ifconfig ce0 -failover
```

Specifically, since `10.0.0.1` will not be on `ce0` by the time the `-failover` subcommand is issued, `-failover` will not be applied to it. Note that the above sequence of commands has always been incorrect, but has usually worked as long as `ce0` has not failed. To reduce the risk of this being a call generator, a warning will be placed in both the IPMP documentation and the release notes.

Explicit IPMP Interface Creation with `ipmp`

Explicit creation will enable the administrator to choose the name of the interface, and optionally associate a specific group name with it. IPMP interfaces will be explicitly created using the new `ipmp` subcommand. This subcommand is analogous to `plumb`, but will instruct the system to create an IPMP group interface rather than a traditional IP interface. For instance, to explicitly create an IPMP interface named `ipmp0`:

```
# ifconfig ipmp0 ipmp
```

As with `plumb`, by default `ipmp` will create an IPv4 instance, but an IPv6 instance will be able to be created instead by using `inet6 ipmp`. However, in practice, this will rarely be needed since `IPv6` will be instantiated on demand if IPv6 addresses are added to a given IPMP group interface.

[28] Of course, test addresses will not migrate as part of being brought `up`. Nonetheless, the kernel will requires that even test addresses be brought down to ensure that an interface is entirely quiesced prior to joining a group.

Attempting to use the ipmp subcommand to create an IP interface that already exists will fail. Similarly, attempting to use the ipmp subcommand to create an IPMP group that already exists will fail.

By default, the group name of the interface will match its interface name. Thus, above, ipmp0 will have a group name of ipmp0. We hope that this behavior, coupled with recommendations in our documentation, will convince administrators to use identical IPMP group and interface names.

Because the ipmp subcommand will allow ifconfig to differentiate the creation of an IPMP group from the plumbing of a physical interface, the administrator will be able to associate a semantically meaningful name with an IPMP group interface:

```
# ifconfig outside0 ipmp
```

As justified in section 3.16, we will still support the ability to associate an arbitrary group name with an IPMP interface. Unsurprisingly, this will be done with the group subcommand. Thus, to move outside0 from group outside0 to group b:

```
# ifconfig outside0 group b
```

Of course, the two former commands will be able to be merged together:

```
# ifconfig outside0 ipmp group b
```

See section 4.1.8 for restrictions associated with changing the group name.

4.1.6 Destroying IPMP Interfaces

Like any other interface, the unplumb subcommand will cause an IPMP group interface to be destroyed. However, destruction will only be possible if there are no physical interfaces currently part of the relevant IPMP group. For instance:

```
# ifconfig ipmp1 ipmp group b
# ifconfig ce0 group b
# ifconfig ipmp1 unplumb
ifconfig: ipmp1: cannot unplumb: IPMP group is not empty
# ifconfig ce0 group ""
# ifconfig ipmp1 unplumb
```

As with a traditional physical interface, any IP addresses associated with the IPMP interface will be removed from the system as part of the unplumb operation.

4.1.7 IPMP Interface Extent

As discussed in section 4.1.5, the initial placement of a physical interface into an IPMP group will create an IPMP interface for the given group (if it does not yet exist), and move any UP data addresses to the IPMP interface. To avoid having to preserve the complicated address migration code, the move will instead be modeled as a set of SIOCLIFREMOVEIF operations (to remove the addresses from the physical interface), followed by a set of `SIOCLIFADDIF` operations (to add the addresses to the IPMP interface).

At this point, the data addresses will be "owned" by the IPMP interface – **removing the interface from the IPMP group will not cause the addresses to "migrate back"**[29]. Similarly, placing the interface into a different IPMP group will not trigger address migration because the physical interface will have no data addresses prior to the group change operation.

Even if all physical interfaces are removed from an IPMP group, the IPMP interface associated with the group will continue to exist (this will be necessary to support explicit IPMP group creation). Thus, the IPMP group can be explicitly destroyed using `unplumb` if it is no longer desired.

4.1.8 Configuring IPMP Group Names

As previously mentioned, the administrator will be able to use the `group` subcommand to set the group name of an IPMP interface. Any interfaces in the group will be changed to have the new group name. That is:

```
# ifconfig ipmp0 ipmp
# ifconfig ce0 group ipmp0
# ifconfig ipmp0 group b
# ifconfig ce0
ce0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> ...
     inet 129.146.17.56 netmask ffffff00 broadcast 129.146.17.255
     groupname b
```

Attempts to use an existing group name or to clear the group name will fail:

```
# ifconfig ipmp0 ipmp
# ifconfig ipmp1 ipmp
# ifconfig ipmp1 group ipmp0
ifconfig: ipmp1: SIOCSLIFGROUPNAME: already exists
# ifconfig ipmp1 group ""
ifconfig: ipmp1: SIOCSLIFGROUPNAME: invalid argument
```

4.1.9 Configuring `STANDBY` Interfaces

Since `STANDBY` is inherently tied to a given physical interface, this attribute will still be configured on it using:

```
# ifconfig ce0 standby
```

Since it makes no semantic sense, attempting to set `standby` on an IPMP interface will fail.

4.1.10 Configuring Hardware Addresses

IPMP requires that only one interface in a group may use a given hardware address. This is usually the case, unless the `eeprom` variable `local-mac-address?` is set to `false`[30], or the underlying

[29] Unfortunately, this also means that if an interface is placed into the wrong group, any migrated addresses must be manually "migrated back" via `removeif` and `addif`.

[30] See CRs 4868371 and 6504375 for more on this unfortunate variable.

interface does not support a unique hardware address. In the former case, `local-mac-address?` can simply be set to `true`. In the latter case, the `ether` subcommand must be used to configure the interface with a unique hardware address.

This will remain unchanged in the new model. However, in the new model, `in.mpathd` will actively enforce that only one interface in a group may use a given hardware address. Specifically, if an interface with a duplicate hardware address is added to a group, or an existing interface in a group is changed to have a duplicate hardware address, `in.mpathd` will log a diagnostic message and take the interface offline[31]. If subsequent changes to the group configuration cause the interface's hardware address to become unique again, `in.mpathd` will log another diagnostic message and bring the interface back online. Finally, `in.mpathd` will fail attempts to online an interface with a duplicate hardware address (regardless of whether the interface was explicitly offlined by `in.mpathd`, or merely became a duplicate while it was administratively offline).

Since the IPMP interface has no hardware address, attempting to use `ether` on it will fail.

4.1.11 Miscellaneous `ifconfig` Operations

Since the IPMP interface will not exist at the DLPI layer, the `auto-revarp`[32], `modlist`, `modinsert` and `modremove` `ifconfig` subcommands will fail with an appropriate error. Additionally, each IPMP interface has its own interface metric (see section 5.7), which will be able to be changed with the `metric` subcommand.

4.2 Observability with `ipmpstat`

The IPMP rearchitecture will include a new command in `/usr/sbin` called `ipmpstat`. This tool will concisely display information about the IPMP subsystem that is currently only available by sifting through the output of several existing commands, or simply not available at all.

By sifting through open bug reports and posts to IPMP mailing lists over the past several years, and drawing on our own internal experience with IPMP, we generated a list of common questions regarding the IPMP subsystem[33]. While the amount of IPMP-related information is surprisingly large, it can be broken down into five subcategories:

- IPMP group information
- IPMP address information
- IPMP underlying interface information
- IPMP target information
- IPMP probe information

Accordingly, we propose five different output formats for the `ipmpstat` command, each selectable through a command-line flag. The following subsections explain each output format through a collection of sample output. Please note that:

- By default, hostnames rather than numeric IP addresses will be output (provided a hostname exists). As with `netstat` and other existing commands, an `-n` option will cause numeric IP addresses to be output instead. The following examples use `-n` where necessary to clarify the relationship between the proposed `ipmpstat` output and the `ifconfig` output shown earlier.

[31] As covered in section 4.2.3, `ipmpstat` will also report this misconfiguration.

[32] While we could change `ifconfig` to perform RARP on an active underlying interface in the group, DHCP has mostly obsoleted RARP, making such a wart hard to justify.

[33] http://opensolaris.org/os/project/clearview/ipmp/ipmp-obs-questions.txt

- As implied by the examples, in general, no special privileges will be needed to invoke `ipmpstat`. One exception is `ipmpstat -p`, since limitations[34] in the sysevent framework prevent unprivileged processes from receiving sysevents.
- The primary purpose of `ipmpstat` will be to interactively debug problems. As such, the standard output format will be `unstable`. However, for each output format, an `-o` option will allow the set of displayed fields to be controlled, and a `-P` option will produce a machine-parsable format; see sections 4.2.6 and 4.2.7 for details.
- In order to properly showcase each of the formats, the sample configuration shown in the earlier examples has been expanded significantly. Subsequent sections of this document will revert to the simpler configuration first shown in section 3.1.

4.2.1 Observing IPMP groups with `ipmpstat -g`

The state of all IPMP groups on the system will be viewable using the `-g` flag to `ipmpstat`:

```
$ ipmpstat -g
GROUP      GROUPNAME   STATE      FDT       INTERFACES
ipmp0      outside     ok         10.00s    ce0 ce1
ipmp1      inside      failed     --        [hme0 hme1]
ipmp2      service     degraded   20.00s    qfe0 qfe3 (qfe2) [qfe1]
```

As can be seen above, for each IPMP group, `ipmpstat` will show its name, associated group interface, group state, failure detection time (`--` if probe-based failure detection is disabled), and underlying interfaces. If IPMP anonymous group support is enabled, the anonymous group will be shown with `--` for its `GROUP` and `GROUPNAME` fields.

The group state will be one of:

- `ok`: All interfaces in the group are usable.
- `degraded`: Some – but not all – interfaces in the group are usable.
- `failed`: No interfaces in the group are usable.

The terms `degraded`[35] and `ok` were selected to ease future integration between IPMP and the Solaris Fault Management Architecture (FMA)[36], which provides compatible definitions for these terms. Although FMA prefers `faulted` to indicate failure, we propose `failed` in order to remain consistent with existing IPMP terminology and exposed interfaces such as the `IFF_FAILED` flag.

Finally, closer examination of the interface list will allow experienced administrators to quickly tell which interfaces are active, `INACTIVE`, and unusable. Specifically:

- Active interfaces will be listed first and not enclosed in any brackets or parenthesis. Active interfaces are those currently being used by the system to send or receive data traffic.
- `INACTIVE` interfaces will be listed next and enclosed in parenthesis. `INACTIVE` interfaces are those that are functioning, but not currently being used according to administrative policy.
- Unusable interfaces will be listed last and enclosed in brackets. Unusable interfaces are those unable to be used at all in their current configuration (e.g., `FAILED` or `OFFLINE`).

[34] See CR 4791900.
[35] The concept of a "degraded" group is new with the IPMP rearchitecture.
[36] http://opensolaris.org/os/community/fm/

4.2.2 Observing IPMP data addresses with `ipmpstat -a`

The state of all IPMP data addresses on the system will be viewable using the `-a` flag to `ipmpstat`:

```
$ ipmpstat -an
ADDRESS                 STATE   GROUP      INBOUND    OUTBOUND
129.146.17.55           up      ipmp0      ce0        ce0 ce1
129.146.17.57           up      ipmp0      ce1        ce0 ce1
192.0.0.100             up      ipmp1      --         --
192.0.0.101             up      ipmp1      --         --
128.0.0.100             up      ipmp2      qfe0       qfe0 qfe3
128.0.0.101             up      ipmp2      qfe3       qfe0 qfe3
128.0.0.102             down    ipmp2      --         --
```

For each data address, the administrator will be able to quickly see what group it belongs to, whether it is available for use (as toggled with `ifconfig up/down`), and what inbound and outbound interfaces it can be used on. Note that this will allow administrators to finally see that a given data address is not tied to a specific outbound interface, but rather may be used on *any* active interface in the group. Likewise, this output will make it explicit that each data address can be associated with at most one inbound interface at a time, and allow administrators to observe the current inbound interface binding chosen by the system.

4.2.3 Observing IPMP interfaces with `ipmpstat -i`

The state of all interfaces using IPMP on the system will be viewable using the `-i` flag to `ipmpstat`:

```
$ ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
ce0         yes     ipmp0       --mb---   up        ok        ok
ce1         yes     ipmp0       -------   up        disabled  ok
hme0        no      ipmp1       -----d-   up        disabled  offline
hme1        no      ipmp1       is-----   down      unknown   failed
qfe0        yes     ipmp2       --mb---   unknown   ok        ok
qfe1        no      ipmp2       i------   up        ok        ok
qfe2        no      ipmp2       -------   up        failed    failed
qfe3        yes     ipmp2       --mb---   up        ok        ok
```

For each interface, the administrator will be able to quickly tell whether an interface is active and what group it belongs to (`--` for interfaces in the anonymous group). The remaining information will allow the administrator to ascertain the reason the interface is or is not active:

- The `link` field will provide the current link status, and will be one of `up`, `down`, or `unknown`. The `unknown` state will indicate that the driver does not support link up/down notification.
- The `probe` field will provide the current probe status, and will be one of `ok`, `failed`, `unknown`, or `disabled`. The `unknown` state will indicate that no probe targets could be found. The `disabled` state will indicate that no IPMP test address is configured on the interface, disabling probe-based failure detection.
- The `state` field will provide the overall state of the interface, and will be one of `ok`, `failed`, `offline`, or `unknown`. If the interface has been offlined, then its state will be `offline`. For

online interfaces, if the link status is `down` or the probe status is `failed`, then the interface state will be `failed`. Otherwise, if the probe status is `unknown` (indicating that no probe targets could be found), the overall state will be `unknown`. If none of the above apply, it will be `ok`.

- The `flags` field will concisely display whether the interface is:
  - `i`: `INACTIVE`
  - `s`: `STANDBY`
  - `m`: Nominated to send and receive IPv4 multicast traffic (see sections 3.13 and 5.15).
  - `b`: Nominated to receive broadcast traffic (see section 3.14).
  - `M`: Nominated to send and receive IPv6 multicast traffic (see sections 3.13 and 5.15).
  - `d`: Unusable due to being marked `down`.
  - `h`: Unusable due to having a duplicate hardware address (see section 4.1.10).

  Additional flags may be defined in the future, if needed.

4.2.4 Observing IPMP probe targets with `ipmpstat -t`

The status of all probe targets associated with each interface using IPMP will be viewable using the `-t` flag to `ipmpstat`:

```
$ ipmpstat -tn
INTERFACE MODE      TESTADDR          TARGETS
ce0       routes    129.146.17.56     129.146.17.1 129.146.17.3
ce1       disabled  --                --
hme0      disabled  --                --
hme1      routes    192.1.2.200       192.1.2.1
qfe0      multicast 128.0.0.200       128.0.0.1 128.0.0.2
qfe1      multicast 128.0.0.201       128.0.0.2 128.0.0.1
qfe2      multicast 128.0.0.202       128.0.0.1 128.0.0.2
qfe3      multicast 128.0.0.203       128.0.0.1 128.0.0.2
```

For each interface, the administrator will be able to see the method used to obtain the probe targets, the test address used to send and receive probes, and the current list of probe targets. All IP interfaces configured to use IPMP will be displayed, but those that have probe-based failure detection disabled will show a target mode of `disabled`, and `--` will be shown for the remaining fields. Although unusual, the administrator may choose to configure an IP interface with both IPv4 and IPv6 test addresses, in which case two lines of output will be displayed (for the IPv4 and IPv6 probe target configurations, respectively).

If probe-based failure detection is enabled, the target mode will be set to either `routes` or `multicast`, depending on whether the system routing table or a multicast probe was used to obtain the target list. In addition, the test address associated with the interface will be displayed, along with a list of current probe targets, in firing order (i.e., the first one listed will be the next target to be probed). Note that if multicast is used, all interfaces in a group will not necessarily have the same ordering or set of probe targets.

4.2.5 Observing IPMP probes with `ipmpstat -p`

All of the IPMP probes being sent on the system will be viewable using the `-p` flag to `ipmpstat`:

```
# ipmpstat -pn
TIME     INTERFACE   PROBE   TARGET              NETRTT    RTT       RTTAVG
0.11s    ce0         589     10.8.57.248         0.51ms    0.76ms    0.76ms
0.17s    hme1        612     192.1.2.1           --        --        --
0.25s    qfe0        602     128.0.0.1           0.61ms    1.10ms    1.10ms
0.26s    qfe1        602     128.0.0.2           --        --        --
0.25s    qfe2        601     128.0.0.1           0.62ms    1.20ms    1.00ms
0.26s    qfe3        603     128.0.0.1           0.79ms    1.11ms    1.10ms
1.64s    ce0         590     10.8.57.248         0.49ms    0.50ms    0.73ms
1.66s    hme1        613     192.1.2.1           --        --        --
1.70s    qfe0        603     128.0.0.1           0.63ms    1.10ms    1.10ms
^C
#
```

Unlike other `ipmpstat` command-line flags, `-p` will run until explicitly terminated by the administrator. While running, `ipmpstat` will retrieve information from the IPMP subsystem regarding acknowledged or lost probes, and output any new probe activity.

For each probe, `ipmpstat -p` will be able to report the following information:

- An identifier representing the probe. The identifier will start at zero and will monotonically increment for each probe sent by `in.mpathd` over a given IP interface. To facilitate more detailed analysis by packet monitoring tools, this identifier will match the `icmp_seq` field of the ICMP probe itself.
- The interface the probe was sent on. Administrators interested in probes only from specific interfaces can post-process the output based on this field.
- The time the probe was sent, relative to when `ipmpstat` was invoked. If the displayed probe was sent prior to starting `ipmpstat`, the time will be shown as negative, relative to when `ipmpstat` was invoked.
- The target the probe was sent to. Administrators interested in probes only to specific targets can post-process the output based on this field.
- The network round-trip-time for the probe, in milliseconds. If `in.mpathd` has concluded that the probe has been lost, `--` will be displayed instead. This is the time between when the IP module sends the probe and when the IP module receives the ack.
- The total round-trip-time for the probe, in milliseconds.If `in.mpathd` has concluded that the probe has been lost, `--` will be displayed instead. This is the time between when `in.mpathd` starts executing the code to send the probe, and when `in.mpathd` completes processing the ack. Spikes in the total round-trip time that are not present in the network round-trip time indicate that the system itself is overloaded.
- The current average and standard deviation for the round-trip-time to the specified target on the specified interface. This will allow easy identification of slow or jittery targets. As implied by the above example, the standard deviation (`RTTDEV`) will not be reported by default because of insufficient screen width.

4.2.6 Controlling Output with `ipmpstat -o`

Any of the aforementioned output formats will be able to be combined with `-o` to produce a customized and deterministic output format. The `-o` option will take a comma-separated list of field names to display. For example:

```
$ ipmpstat -g -o group,fdt,interfaces
GROUP      FDT       INTERFACES
ipmp0      10.00s    ce0 ce1
ipmp1      --        [hme0 hme1]
ipmp2      20.00s    qfe0 qfe3 (qfe2) [qfe1]
```

The special field name `all` will cause all fields to be output, rather than the default behavior of displaying the fields that will fit in 80 columns. This distinction is only important for output formats that have more fields than will fit in 80 columns (currently only `ipmpstat -p`). However, since new fields may be added that cause an output format to overflow 80 columns, `all` will ensure that all fields are always output even as `ipmpstat` is enhanced.

4.2.7 Machine-Parsable Output with `ipmpstat -P`

Any of the aforementioned output formats will be able to be combined with `-P` to produce a machine-parsable format. The format will be based on `dladm`'s parsable format, and thus will differ from the normal `ipmpstat` output format in the following ways:

- The header will be omitted.
- Fields will be separated by `:` rather than whitespace.
- If multiple fields are requested, and a literal `:` or \ occur in a field's value, they will be escaped by prefixing them with \.
- Fields with empty values will yield no output, rather than `--`.

Thus, adding `-P` to the previous example:

```
$ ipmpstat -g -P -o group,fdt,interfaces
ipmp0:10.00s:ce0 ce1
ipmp1::[hme0 hme1]
ipmp2:20.00s:qfe0 qfe3 (qfe2) [qfe1]
```

As with `dladm`, this format is intended to be easy to use from shell scripts. For instance, to get an FDT for a specific group:

```
getfdt() {
        ipmpstat -gP -o group,fdt | while IFS=: read group fdt; do
                if [[ "$group" = "$1" ]]; then
                        echo "$fdt"
                        return
                fi
        done
}
```

To ensure that the output remains predictable as the `ipmpstat` feature set evolves, the command will fail if an output field list is not specified, or if `-o all` is specified:

```
$ ipmpstat -gP
ipmpstat: output field list (-o) required in parsable output mode
$ ipmpstat -gP -o all
ipmpstat: "all" not allowed in parsable output mode
```

4.3 System Boot

4.3.1 IPMP Configuration at Boot

Administratively, the persistent IPMP configuration of each IP interface is stored in a corresponding `/etc/hostname.`*if* file. As part of system boot, the contents of these files are passed to `ifconfig` to configure the interface's IPMP-related attributes. In addition, `in.mpathd` is started to begin probing interface health and perform any necessary address migrations. At startup, `in.mpathd` reads `/etc/default/mpathd`, and adjusts its behavior accordingly.

Because the IPMP rearchitecture maintains `ifconfig` compatibility, existing `/etc/hostname.`*if* files will continue to work, and no changes to `/etc/hostname.`*if* files will be required as part of system upgrade. However, by analogy with `ifconfig`, our documentation (e.g, on `docs.sun.com`) will encourage administrators to put all IPMP data addresses in new `/etc/hostname.`*ipmpif* files, rather than in the underlying interface's `/etc/hostname.`*if* file. In addition, administrators who wish to associate a particular IPMP interface with a particular groupname will need to create a corresponding `/etc/hostname.`*ipmpif* file containing `ipmp group` *groupname*.

For instance, the current way to configure the configuration used throughout this document is:

```
/etc/hostname.ce0:
   group a
   129.146.17.55 netmask + broadcast + up
   addif 129.146.17.56 deprecated -failover netmask + broadcast + up
/etc/hostname.ce1:
   group a
   129.146.17.57 netmask + broadcast + up
   addif 129.146.17.58 deprecated -failover netmask + broadcast + up
```

That will continue to work, but this will be preferred:

```
/etc/hostname.net17:
  ipmp group a
  129.146.17.55 netmask + broadcast + up
  addif 129.146.17.57 netmask + broadcast + up
/etc/hostname.ce0:
  group a 129.146.17.56 -failover
/etc/hostname.ce1:
  group a 129.146.17.58 -failover
```

4.3.2 Interface Creation Order at System Boot

For the above example to work, `net17` must configured at boot prior to the configuration of `ce0` and `ce1` – otherwise, the system would implicitly create an `ipmp0` interface and assign *it* (rather than `net17`) to group `a`. To address this, any `/etc/hostname.`*ipmpif* files (easily identified since their first token will be `ipmp`) will be configured prior to configuring any underlying interfaces.

4.3.3 Missing Interfaces at System Boot

While boot appears straightforward, there is currently an edge case that significantly complicates matters. In particular, suppose an `/etc/hostname.`*if* file exists for an interface that is part of an IPMP group, but the interface cannot be plumbed. In this case, the IP addresses in that `/etc/hostname.`*if* file *must* be brought up on another interface in the group, so that the system remains completely available. Unfortunately, since the interface cannot be plumbed, `in.mpathd` is unaware of the "failed" interface, and thus the system boot scripts must implement failover on `in.mpathd`'s behalf. Because selecting the optimal interface to failover to is complicated, this logic significantly complicates system boot.

Since the concept of address failover no longer exists, this logic will be able to be simplified considerably. Specifically, in the future, the boot scripts will still try to plumb each interface, and will still build a list of interfaces that could not be plumbed. However, once the list of failed interfaces has been constructed, the boot scripts will only need to determine what IPMP group each interface was supposed to be in, and add its data addresses directly to the appropriate IPMP group interface (potentially creating IPMP groups along the way).

A pleasing side effect of the new model is that a system will be able to boot properly even when *all* of the group's underlying interfaces cannot be plumbed, and will automatically make use of any underlying interfaces that are subsequently dynamically reconfigured into the system.

4.3.4 Boot Environment Changes

As part of handling missing interfaces, the boot scripts must check whether a given IPMP group already exists. This can easily be done using `ipmpstat`, but will require that `ipmpstat` be in `/sbin` since `/usr` may not be available yet. An `ipmpstat` symlink will be provided in `/usr/sbin` for convenience and consistency with other administrative programs used during boot, such as `mount(1M)`. Since `ipmpstat` makes use of `libipmp` (see section 5.21), `libipmp` will be moved from `/usr/lib` to `/lib`.

Since `ipmpstat` queries live state within `in.mpathd`, it must also be running and thus must also be in `/sbin`. Long ago, `in.mpathd` resided in `/sbin` but was moved to `/usr/lib/inet` to make use of `libsysvent.so` back before dynamic linking was possible in `/sbin`. However, programs in `/sbin` are now dynamically linked and thus `in.mpathd` can be moved back to `/sbin`, and a compatibility symlink put in `/usr/lib/inet`. In addition, the undocumented `SUNW_NO_MPATHD` environment variable – which prevented `in.mpathd` from starting until `/usr` was mounted – will be removed. However, to ensure that `in.mpathd` is started on systems only using the IPMP anonymous group, the `net-init` script will continue to invoke `in.mpathd` with the undocumented `-a` (adopt) flag if it is not already running. As is currently the case, `in.mpathd` will automatically exit if there are no IP interfaces to track.

4.4 Dynamic Reconfiguration

Since dynamic reconfiguration is performed at the hardware-level, the administrative model for DR will be unchanged. That is, the administrator will continue to refer to specific physical devices when using `if_mpadm`, and specific attachment points when using `cfgadm`. Since it is not associated with any specific piece of hardware, it will not be possible to offline or remove an IPMP interface.

4.5 Routing Table Administration

Understanding and manipulating the routing table will become significantly simpler with the new model. Specifically, the core issues will be fixed since a routing table entry will now be able to refer to the IPMP group as a whole. This is semantically identical to the current behavior, but suffers from none of the existing corner cases, such as what to do when the interface associated with a given route is removed from the IPMP group. It also makes it clear to the administrator that the route is associated with the group as a whole, rather than a specific interface in a group.

For instance, the routing table for the configuration used throughout currently looks like:

```
129.146.17.0        129.146.17.56        U          1      0  ce0
129.146.17.0        129.146.17.58        U          1      0  ce1
129.146.17.0        129.146.17.55        U          1   6582  ce0:1
129.146.17.0        129.146.17.57        U          1    191  ce1:1
default             129.146.17.1         UG         1  15273  ce0
```

Specifically, we see interface routes for the data addresses on `ce0` and `ce1`, and that `ce0` has been arbitrarily picked to represent the IPMP group for the `default` route. With the IPMP interface, this will look like:

```
129.146.17.0        129.146.17.56        U          1      0  ce0
129.146.17.0        129.146.17.58        U          1      0  ce1
129.146.17.0        129.146.17.55        U          1   6582  ipmp0
129.146.17.0        129.146.17.57        U          1    191  ipmp0:1
default             129.146.17.1         UG         1  15273  ipmp0
```

As one would expect, routes will be added using the IPMP interface – for instance, to add another default route:

```
# route add -net default 129.146.17.2 -ifp ipmp0
```

For backward compatibility, this will also be supported:

```
# route add -net default 129.146.17.2 -ifp ce0
```

Specifically, the kernel will automatically convert this to a request to add a route on the corresponding IPMP group interface; see section 5.5 for details. (The first two routes in the preceding routing table examples are interface routes installed by the kernel when the test addresses are brought up and are used by `in.mpathd` for probe traffic.)

4.6 IPv6 Configuration

4.6.1 IPv6 Link-Local Test Addresses

Currently, when an interface is plumbed for IPv6, an IPv6 link-local address is automatically generated and brought up. Thus, each physical interface in an IPMP group has its own IPv6 link-local address. While these addresses do not fail over as part of interface failure, they will only be used as test addresses by `in.mpathd` if explicitly marked `IFF_NOFAILOVER` by the administrator.

To continue to provide failure and repair detection for IPv6 hosts, the existing semantics of IPv6 link-local addresses on underlying interfaces will be preserved. In particular, IPv6 link-local addresses will still never fail over, and will stillonly be used as test addresses by `in.mpathd` if explicitly marked `IFF_NOFAILOVER` by the administrator[37].

[37] We considered having the IP module automatically mark all link-locals on underlying IP interfaces as `IFF_NOFAILOVER`, but this introduces additional complexity – e.g., the IP module would need to track whether the administrator or IP module set `IFF_NOFAILOVER` so that the IP module could restore the old value if the IP interface later leaves the group.

4.6.2 IPv6 Link-Local Data Addresses

Since link-local test addresses cannot fail over on interface failure, they are unsuitable for application use. To fill this need, when an IPv6 IPMP interface is created, the IP module will automatically configure an IPv6 "link-local" data address on it. As described in RFC 2373, IPv6 link-local addresses are constructed from a 64-bit *Interface ID*, which is in turn typically generated using the associated link's hardware address. Since the IPMP interface does not have a dedicated hardware address, a different approach must be used to generate the Interface ID.

Specifically, we propose a variant of the algorithm specified in section 3.2.1 of RFC 3041 (Privacy Extensions for Stateless Address Autoconfiguration) which will compute the MD5 hash using the IPMP interface name, the zone name, and the host identifier (as reported by `hostid(1)`), and will map that MD5 hash to an Interface ID as per RFC 3041. This will ensure that a given IPMP interface will consistently generate the same Interface ID (and thus the same link-local address) while minimizing risk of duplicates. Note that as with other IPv6 interfaces, the administrator can also override the generated Interface ID by using the `ifconfig token` subcommand.

Duplicate address detection for the link-local address will be performed on one of the active underlying interfaces (provided one exists) to ensure that no other on-link hosts are currently using it. If there are no active underlying interfaces in the IPMP group, then the link-local address will be unreachable by other hosts. Otherwise, it will be associated with the hardware address of one of the active interfaces in the IPMP group. If the link-local address must be reassociated with a different underlying interface's hardware address (e.g., as part of interface failure), and an unsolicited Neighbor Advertisement message will be sent to update other hosts on the link.

The Neighbor Solicitation and Neighbor Advertisement messages associated with the IPMP interface's IPv6 link-local address will use the currently-associated hardware address for the link-layer address fields. However, note that the messages themselves may be sent over any active interface in the group, which may have a different hardware address.

In total, the IPv6 link-local test and data address configuration might look like:

```
ce0: flags=a000841<UP,RUNNING,MULTICAST,IPv6,NOFAILOVER> mtu 1500 index 2
        inet6  fe80::203:baff:fe16:51bd/10
        groupname a
ce1: flags=a000841<UP,RUNNING,MULTICAST,IPv6,NOFAILOVER> mtu 1500 index 3
        inet6  fe80::a00:20ff:feec:f42f/10
        groupname a
ipmp0: flags=b000843<UP,RUNNING,MULTICAST,IPv6,IPMP> mtu 1500 index 4
        inet6  fe80::45c0:9dc2:2f82:89c3/10
        groupname a
```

4.6.3 IPv6 Stateless Address Autoconfiguration

Currently, when an interface is plumbed for IPv6, stateless address autoconfiguration is attempted. Thus, if autoconfiguration is successful, each physical interface in an IPMP group has its own global address, and the IPMP group will collectively have as many global addresses as interfaces.

However, this approach will not work with IPMP interface, because no data addresses will be hosted on the physical interfaces. While we could automatically migrate these data addresses to the IPMP interface, or update `in.ndpd` to automatically populate these addresses on the IPMP interface, there is a more fundamental problem. Specifically, the global addresses are tied to the hardware addresses of the underlying interfaces and are really owned by them. Thus, if one unplumbed IPv6 on an interface, one would expect the addresses to disappear from the IPMP interface. This conflicts with our proposed model, which states that once an address has become part of an IPMP interface, it will remain there until explicitly removed by the administrator. Moreover, it would mean that the set of global IPv6 addresses would change across a DR operation, causing any applications using the changed address(es) to fail.

Thus, we will instead perform stateless address autoconfiguration on the IPMP interface itself, as part of configuring IPv6 on it. Specifically, as with any other IPv6 interface, `in.ndpd` will combine the IPMP interface's Interface ID (whose generation was discussed in section 4.6.1) with the global prefixes learned via router advertisements to generate and configure IPv6 global addresses. Note that only one address per prefix will be configured – and thus in the common case of a single prefix, inbound global unicast IPv6 traffic will be received on a single underlying interface. This limitation will be documented, and impacted sites will be advised to configure additional global IPv6 addresses, either statically or through DHCPv6.

4.7 `dladm`

Because IPMP groups do not exist at the data-link layer (section 3.15), they cannot be administered with `dladm`. However, the links used by the underlying interfaces in a group will be individually displayed with `dladm show-link`, and the devices which are used by those links will be individually displayed with `dladm show-dev`.

However, `ipmpstat` (section 4.2) will allow the administrator to check the status of the IPMP interfaces in the system. Because this tool will be IPMP-specific, its output will quickly allow the administrator to answer specific questions about the health of their IPMP interfaces, mitigating the lack of visibility from `dladm`.

Sadly, aside from `ifconfig`, there is no tool available to get a broad view of the health and configuration of the IP layer of the system. While adding such a tool is beyond the scope of this work, the addition of the IPMP interface will make IPMP significantly easier to represent in such a tool. Further, the absence of this tool is not sufficient justification to violate layering and provide support for IPMP in `dladm`.

Since IPMP interfaces will not be visible to `dladm`, it will not be used to provide administratively-chosen (vanity) names. However, as previously discussed, IPMP interfaces will be able to be given administratively-chosen names when explicitly created through `ifconfig`. See section 3.16 for a summary of naming issues associated with IPMP interfaces.

4.8 `kstat`

The IPMP interface will also have kstats which will be displayed with `kstat(1M)`, and consumed by utilities such as `netstat`. The following hardware-independent kstats (described in `gld(7D)`) will be supported:

```
obytes      opackets     ipackets     multircv     link_up
obytes64    opackets64   ipackets64   multixmt
rbytes      oerrors      ierrors      brdcstrcv
rbytes64                              brdcstxmt
```

Note that the counters listed above will represent the number of packets or bytes that have flowed through the given IPMP interface since it was created, which may not be the same as the number of packets that have flowed through the underlying interfaces. For instance, if an interface is removed from an IPMP group, the packets that were sent through that interface will still be counted in the IPMP interface's `opackets` kstat. The `link_up` kstat will be non-zero if at least one of the underlying interface links is up (i.e., the IPMP interface has `IFF_RUNNING` set).

4.9 netstat

Like any IP interface, basic statistics about each IPMP interface will be available through `netstat`:

```
$ netstat -i
Name  Mtu  Net/Dest     Address        Ipkts  Ierrs Opkts  Oerrs Collis Queue
lo0   8232 loopback     localhost      165180  0    16280  0     0      0
ce0   1500 monkeys      129.146.17.56  263828  0    329580 0     0      0
ce1   1500 monkeys      129.146.17.58  481035  0    104774 0     0      0
ipmp0 1500 monkeys      129.146.17.55  744863  0    434354 0     0      0
```

The interface statistics for the underlying interfaces in the group will count the number of packets actually sent over that interface, whereas the interface statistics for the group represent all the packets sent to and received by the group as a whole.

In addition, `netstat -p` can be used to display the IPv4 ARP table, and `netstat -p -f inet6` can be used to display the IPv6 Neighbor Discovery table. For instance, invoking `netstat -p` might currently produce output such as:

```
ce0    129.146.17.55         255.255.255.255 SP    08:00:20:ed:79:05
ce1    129.146.17.56         255.255.255.255 SP    08:00:20:ed:79:01
ce0    129.146.17.58         255.255.255.255 SP    08:00:20:ed:79:05
ce1    129.146.17.57         255.255.255.255 SP    08:00:20:ed:79:01
ce0    224.0.0.0             240.0.0.0       SM    01:00:5e:00:00:00
ce1    224.0.0.0             240.0.0.0       SM    01:00:5e:00:00:00
ce1    huon.SFBay.Sun.COM    255.255.255.255       08:00:20:89:d4:17
ce0    zion-sp.SFBay.Sun.COM 255.255.255.255       00:09:3d:00:16:17
```

In the future, this same ARP table will look like:

```
ipmp0  129.146.17.55         255.255.255.255 SP    08:00:20:ed:79:05
ce0    129.146.17.56         255.255.255.255 SP    08:00:20:ed:79:01
ce1    129.146.17.58         255.255.255.255 SP    08:00:20:ed:79:05
ipmp0  129.146.17.57         255.255.255.255 SP    08:00:20:ed:79:01
ce0    224.0.0.0             240.0.0.0       SM    01:00:5e:00:00:00
ce1    224.0.0.0             240.0.0.0       SM    01:00:5e:00:00:00
ipmp0  huon.SFBay.Sun.COM    255.255.255.255       08:00:20:89:d4:17
ipmp0  zion-sp.SFBay.Sun.COM 255.255.255.255       00:09:3d:00:16:17
```

That is, published ARP entries associated with the IPMP test addresses and multicast mappings will remain tied to a specific underlying interface (as they must for correctness). However, ARP cache entries associated with other hosts (such as `huon` and `zion-sp`) will be associated with the group's IPMP interface that they can be used by *any* interface in the group. Similarly, the IPMP group's data addresses will be associated with the IPMP group interface.

Semantically identical changes will be made to the output of the Neighbor Discovery table entries displayed through `netstat -p -f inet6`.

4.10 arp

The arp command does not currently operate on interface names, and is thus unaffected by this proposal. However, arp with no arguments invokes netstat -p, which is discussed in section 4.9.

4.11 Packet Monitoring

We propose to to use Clearview's IP-level observability component[38] in order to provide packet monitoring of an IPMP group as a whole at the IP layer. Specifically, the aforementioned configuration would provide the following nodes in /dev/ipnet:

```
/dev/ipnet/lo0
           ce0
           ce1
           ipmp0
```

Packets sent to and received from ce0 and ce1 will be visible by monitoring the corresponding ipnet node; monitoring ipmp0 will allow one to see all IP packets sent to and received from *all* underlying interfaces in the IPMP group. If promiscuous-mode is not enabled, only IP packets destined to IP addresses hosted on a particular interface (along with broadcast and multicast IP packets) will be visible. Since only test addresses will be hosted on ce0 and ce1, IP promiscuous mode must be enabled to see the non-test traffic flowing over those interfaces. However, promiscuous mode is the default behavior for most packet monitoring programs, including snoop, so this should not matter in practice.

Because IPMP is an IP abstraction, it will not be possible to monitor an IPMP group at the data-link layer. However, as is currently the case, each underlying interface will be able to be monitored at the data-link layer. Additionally, ipmpstat -p (section 4.2.5) may be used by itself to track down simple IPMP probe-related problems, or in concert with packet monitoring tools to root-cause more subtle issues (using the probe number for data correlation).

4.12 Packet Filtering

Since the integration of Packet Filtering Hooks[39] into Solaris, packet filtering is no longer performed by inserting STREAMS modules below IP, but by registering packet filtering routines to be called back by IP itself on a per-IP-interface basis. Thus, the introduction of the IPMP group interface will enable packet filtering rules to be applied to the group as a whole. As such, administrators will be able to specify rules that apply to IPMP group interfaces. To ensure that the filtering behavior of the group is predictable, once an interface joins a group, packets flowing over it will only be subject to the group interface's rules. If an interface is removed from a group, the underlying interface's rules (if any) will again be applied.

As part of this work, the ipmp_hook_emulation ndd tunable – which provided an expedient way for IPMP and IP Filter to work together – will be rendered unnecessary and thus removed. Subsequent work is planned to make Packet Filtering Hooks available for public use, which will enable third-party packet filtering software to work with IPMP as well.

[38]See PSARC/2006/475 and http://opensolaris.org/os/project/clearview/ipnet/

[39]http://opensolaris.org/os/community/networking/files/pfhooks-design-2006-03-09.pdf

4.13 DHCP Interaction

As shown in section 4.1.2, DHCP can be used on an IPMP interface just as it can on a physical interface. This makes administrative sense because data addresses no longer move, and thus the lease's handle (e.g., `ipmp0:1`) will remain constant, allowing core attributes to be specified in `/etc/default/dhcpagent` and enabled at boot using `/etc/dhcp.`*if* files. The DHCP client can already automatically generate client identifiers for logical interfaces, and we will extend this to also support physical interfaces that do not have a hardware address. Thus, explicit configuration of the `CLIENT_ID` parameter in `/etc/default/dhcpagent` will *not* be necessary. Because the semantics of `IFF_RUNNING` on an IPMP interface mirror the semantics on a traditional IP interface, the DHCP client will automatically refresh any data address leases across a group failure. As with traditional IP interfaces, the MTU of an IPMP interface must be sufficient to support DHCP. As a degenerate case, an IPMP interface with no interfaces (see section 3.10) does not have a large enough MTU to support DHCP, and thus DHCP lease acquisition will fail.

Test addresses will also be able to be configured using DHCP. Although a test address can be configured on any logical interface, given that the test address will typically be the only address on an underlying interface, we will recommend that it be configured on the physical interface. Thus:

```
# ifconfig ce0 plumb
# ifconfig ce0 group a
# ifconfig ce0 dhcp start
#
```

Note in particular that `-failover` need not be specified, as the DHCP client will automatically set `IFF_NOFAILOVER` prior to starting lease acquisition[40]. Although the DHCP client will use any active interface in the group to maintain the address lease, infinite leases will be strongly recommended for test addresses so that a group failure will not trigger spurious repairs. For example, if the DHCP client is unable to renew a test address lease (e.g., because of a group failure), then it must unconfigure the test address, which will cause `in.mpathd` to disable probe-based failure detection on the interface, possibly triggering a spurious repair.

Although it is not documented, the boot scripts currently process the contents of `/etc/hostname.`*if* files even for interfaces using DHCP. So, to make the above configuration persistent:

```
# > /etc/dhcp.ce0
# echo group a > /etc/hostname.ce0
```

This will be explicitly documented in the future.

One oddity is that if the `REQUEST_HOSTNAME` tunable is enabled, `dhcpagent` will read each line of the `/etc/hostname.`*if* file, and if the a line is of the form `inet` *hostname*, then it will pass *hostname* as a hint to the DHCP server (see `dhcpagent(1M)`). While awkward, this feature can be used on either an underlying interface or an IPMP interface. For example, to have `dhcpagent` request hostname `frobnitz` for IPMP interface `ipmp0`:

```
# echo REQUEST_HOSTNAME=yes >> /etc/default/dhcpagent
# > /etc/dhcp.ipmp0
# echo ipmp group b > /etc/hostname.ipmp0
# echo inet frobnitz >> /etc/hostname.ipmp0
```

[40] As per section 5.3, attempting to clear `IFF_NOFAILOVER` on an address managed by DHCP will fail.

Ostensibly to avoid interfering with static address configuration, the DHCP client has traditionally monitored the `IFF_UP` flag on DHCP-managed addresses and abandoned the lease if a change is detected. However, the process of offlining an interface brings down its test addresses, which would cause the current DHCP client to erroneously sabandon test address leases. As such, this monitoring will be removed and the DHCP client will instead rely on normal lease expiration. As before, the DHCP client *will* abandon a lease if its address is changed, thus ensuring that conflicts with static address configuration will still be avoided. Although `IFF_UP` monitoring is (briefly) documented, this is to ease problem diagnosis rather than to provide a programmatic interface (and further, programmatic interfaces already exist to drop or release a lease), and thus the change is low-risk.

To allow DHCP-managed test addresses to work with DR, RCM will be enhanced to also restore DHCP configuration across a DR operation; see section 3.6 for details.

4.14 Zones Interaction

4.14.1 Shared IP Stack Zones

An IPMP interface's addresses can be assigned to a shared-stack zone in the same manner as a traditional physical IP interface. Specifically, since the IPMP interface *is* an IP interface, it (and any addresses) can be specified using the `zonecfg(1M) physical` property – e.g.,:

```
# zonecfg -z a
a: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:a> create
zonecfg:a> set zonepath=/export/home/a
zonecfg:a> set autoboot=true
zonecfg:a> add net
zonecfg:a:net> set address=192.168.0.1/24
zonecfg:a:net> set physical=ipmp0
zonecfg:a:net> end
zonecfg:a> exit
```

Legacy configurations will also continue to work. For instance, if `ce0` is in the IPMP group associated with `ipmp0`, and `set physical=ce0` is instead specified above, then when `zoneadmd` brings up the address during zone boot, the address will be automatically migrated to `ipmp0`. Since the address (rather than the interface) is assigned to a zone, a subsequent `ifconfig -a` in the booted zone will show the address on a logical interface of `ipmp0`. Regardless, since the migration at zone boot is potentially confusing and the configuration suggests the wrong administrative model for IPMP, this will not be a recommended configuration. As such, `zoneadm verify` will be updated to warn if a `physical` property is set to an underlying interface.

Since `in.mpathd` requires privileges that are not granted to a shared-stack zone by default, and moreover, manages the health of IP interfaces that are often in use by multiple zones, `in.mpathd` will continue to run from the global zone. Because `ipmpstat` must communicate with `in.mpathd` to retrieve its information, and inter-zone communication facilities would noticeably complicate the implementation, `ipmpstat` will not operate in a shared-stack non-global zone. If this becomes a significant limitation, `ipmpstat` support could be implemented, but only to show the IPMP state associated with that zone. Specifically, `ipmpstat` output would be restricted to showing addresses assigned to the zone, along with the IP interfaces associated with those addresses and the IPMP groups associated with those IP interfaces.

4.14.2 Exclusive IP Stack Zones

Since IPMP is an IP-level construct, and since the resources exported to exclusive IP-stack zones are data-links, the introduction of the IPMP interface will not affect the configuration of exclusive-stack zones. For instance, the following will still configure a zone with two data-links, `ce1000` and `ce1001`, across which IPMP could be configured:

```
# zonecfg -z a
a: No such zone configured
Use 'create' to begin configuring a new zone.
zonecfg:a> create
zonecfg:a> set zonepath=/export/home/a
zonecfg:a> set autoboot=true
zonecfg:a> set ip-type=exclusive
zonecfg:a> add net
zonecfg:a:net> set physical=ce1000
zonecfg:a:net> end
zonecfg:a> add net
zonecfg:a:net> set physical=ce1001
zonecfg:a:net> end
zonecfg:a> exit
```

The local zone could then configure its `/etc/hostname.`*if* files to use IPMP using one of the approaches discussed in section 4.3.1. As with all exclusive IP-stack zones, the resulting IP configuration would be local to the zone and invisible to the global zone. Therefore, as is currently the case, any IP interfaces using IPMP in that zone will be exclusively monitored by a zone-local `in.mpathd` daemon. Similarly, running `ipmpstat` in that zone will show only the IPMP state associated with the IP interfaces in the zone, and those interfaces will be invisible to `ipmpstat` in the global zone. One caveat is that sysevents do not work yet in non-global zones, so `ipmpstat -p` will not work in non-global zones until that restriction is lifted.

5 IPMP Rearchitecture: Programming Impact

The majority of networking programming is done using abstractions that are above the IP interface layer, and thus most software will be unaffected by this rearchitecture. Indeed, it was this thought that motivated the original IPMP model – though hindsight has shown that the exceptions constitute a set of core programming interfaces that are used by a number of critical applications, such as routing daemons.

To address this problem, a variety of "solutions" have been explored, including avoiding certain IPMP configurations, "porting" applications to be IPMP-aware, or simply prohibiting use of the application on systems running IPMP. All of these approaches limit widespread adoption of IPMP on Solaris and are therefore inadequate.

As will become clear, the introduction of the IPMP interface will resolve a wide variety of existing programming issues concerning IPMP. In particular, the new model will *finally* allow IP-based networking applications to operate without modification on systems using IPMP. It will provide facilities whereby applications that would benefit from being IPMP-aware (e.g., network management software) can opt-in and thus provide first-class IPMP support. The API changes required to support the new model are outlined in this section.

5.1 Socket Interface Discovery `ioctl` Operations

Networking applications discover the set of configured network interfaces and IP addresses on the system through four socket `ioctl` operations: `SIOCGIFNUM`, `SIOCGLIFNUM`, `SIOCGIFCONF` and `SIOCGLIFCONF`. Currently, these operations return all interfaces and IP addresses, regardless of whether the interface or IP address is appropriate for use by the application. As a result, applications often end up using IPMP test addresses or attempting to send packets out a network interface that is `INACTIVE`, `FAILED`, or `OFFLINE`. Thus, currently any applications which perform these operations must be "ported" to be IPMP-aware and recompiled in order to work properly on IPMP systems[41].

With the new model, this will no longer be necessary. Specifically, by default, **any physical network interface in an IPMP group will be invisible to these `ioctl` operations, preventing them from being accidentally used by IPMP-unaware applications.** However, any IPMP interfaces and any addresses hosted on them *will* be visible to applications.

For administrative applications such as `ifconfig`, **a new `LIFC_UNDER_IPMP` flag will be provided**. Specifically, if this flag is set in the `lifc_flags` member of the `struct lifreq` when using `SIOCGLIFNUM` or `SIOCGLIFCONF`, then physical network interfaces in IPMP groups will be returned as well. While this flag will be documented, we expect that only a handful of unbundled administrative applications will be revised to set it. It is important to note that administrative applications that are not updated will still operate properly, but will simply be unaware of the underlying network interfaces in an IPMP group. Further, these applications will likely require other changes, e.g., in order to indicate the relationship between an IPMP interface and the physical interfaces that comprise the group.

Table 1 summarizes these changes by showing the results of the various `ioctls` on the configuration we have been using throughout. (Since `lo0` is always configured on any functioning machine, it is included in the results.)

[41] One prominent example is SNMP.

ioctl	current	proposed
SIOCGIFNUM	5	3
SIOCGLIFNUM	5	3
SIOCGLIFNUM LIFC_UNDER_IPMP	*N/A*	5
SIOCGIFCONF	ce0 ce0:1 ce1 ce1:1 lo0	ipmp0 ipmp0:1 lo0
SIOCGLIFCONF	ce0 ce0:1 ce1 ce1:1 lo0	ipmp0 ipmp0:1 lo0
SIOCGLIFCONF LIFC_UNDER_IPMP	*N/A*	ce0 ce1 ipmp0 ipmp0:1 lo0

Table 1: Proposed Socket Discovery ioctl Changes

5.2 Examination and Manipulation of Socket Interface Flags

Table 2 summarizes the flags that will be able to be set through `SIOCSLIFFLAGS`, and may be seen as set with `SIOCGLIFFLAGS`, on both an IPMP interface and the underlying interfaces in an IPMP group. Aside from the asterisked entries, the semantics of issuing `SIOCSLIFFLAGS` on the underlying interface will be unchanged from the current behavior. Interesting cells of the table are shown in bold and discussed below:

1. Bringing an address on an underlying interface `IFF_UP` is still permitted, but unless the address is marked `IFF_NOFAILOVER`, the kernel will immediately move it to the corresponding IPMP interface.
2. Per section 3.9, `IFF_STANDBY`, `IFF_OFFLINE`, `IFF_INACTIVE`, and `IFF_NOFAILOVER` will be prohibited on the IPMP interface. However, so that `in.mpathd` can implement `STANDBY` (section 3.7) and `FAILBACK=no` (section 3.8) support, `IFF_INACTIVE` will now be modifiable on underlying interfaces. Section 5.3 details how `IFF_INACTIVE` interacts with `IFF_FAILED` and `IFF_STANDBY`.
3. Per section 3.1, the new `IFF_IPMP` flag will be automatically set on all IPMP interfaces.
4. Although the IPMP interface is not associated with any specific physical hardware, it is not marked `IFF_VIRTUAL` since it can still be used to send and receive IP traffic. (`IFF_VIRTUAL` means that an interface has **no** associated physical hardware at all, thus e.g., routing daemons will never forward "through" it.)
5. `IFF_COS_ENABLED` will only be set if *all* of the interfaces in the IPMP group have it set.
6. Per section 4.1.4, interfaces with `IFF_POINTOPOINT` addresses will not be allowed into a group. Since all `IFF_UNNUMBERED` addresses are `IFF_POINTOPOINT`, `IFF_UNNUMBERED` will never be set.
7. Per section 3.9, setting `IFF_NOFAILOVER` will now also set `IFF_DEPRECATED`. Per section 5.3, clearing `IFF_NOFAILOVER` on an `IFF_UP` or application-managed address will be prohibited.
8. Since setting `IFF_ROUTER` on only part of an IPMP group makes no sense, attempting to set or clear it on an IPMP interface or any interface in the group will set or clear it on *all* interfaces in the IPMP group. Additionally, any interfaces added to a group will have their `IFF_ROUTER` flags updated to match the current group value.
9. Setting `IFF_NOARP`, `IFF_NONUD`, or `IFF_XRESOLV` on an underlying interface will affect only the packets associated with applications directly bound to those underlying interfaces (e.g., `in.mpathd`'s probes). As IPMP is tightly integrated with ARP and IPv6 Neighbor Discovery, attempting to set `IFF_NOARP`, `IFF_NONUD`, or `IFF_XRESOLV` will fail on an IPMP interface.
10. Though the behavior of `IFF_FIXEDMTU` will be unchanged, it remains special since it can be set by an application, but not directly through `SIOCSLIFFLAGS`. Instead, `SIOCS[L]IFMTU` must be used.

ioctl	Modifiable?		Possibly Set?	
	IPMP Interface	**Underlying**	**IPMP Interface**	**Underlying**
IFF_UP	Yes	**Yes***	Yes	Yes
IFF_RUNNING	No	No	Yes	Yes
IFF_FAILED	No	Yes	Yes	Yes
IFF_STANDBY	**No**	Yes	**No**	Yes
IFF_INACTIVE	**No**	**Yes***	**No**	Yes
IFF_OFFLINE	**No**	Yes	**No**	Yes
IFF_VIRTUAL	No	No	**No**	No
IFF_IPMP*	**No**	**No**	**Yes**	**No**
IFF_LOOPBACK	No	No	No	No
IFF_POINTOPOINT	No	No	**No**	**No***
IFF_UNNUMBERED	No	No	**No**	**No***
IFF_NOFAILOVER	**No**	**Yes***	**No**	Yes
IFF_DUPLICATE	No	No	Yes	Yes
IFF_DHCPRUNNING	Yes	Yes	Yes	Yes
IFF_PRIVATE	Yes	Yes	Yes	Yes
IFF_PREFERRED	Yes	Yes	Yes	Yes
IFF_TEMPORARY	Yes	Yes	Yes	Yes
IFF_DEPRECATED	Yes	Yes	Yes	Yes
IFF_ADDRCONF	Yes	Yes	Yes	Yes
IFF_ANYCAST	Yes	Yes	Yes	Yes
IFF_NOXMIT	Yes	Yes	Yes	Yes
IFF_NOLOCAL	Yes	Yes	Yes	Yes
IFF_NOARP	**No**	**Yes***	No	Yes
IFF_NONUD	**No**	**Yes***	No	Yes
IFF_XRESOLV	**No**	**Yes***	No	Yes
IFF_COS_ENABLED	No	No	**Yes**	Yes
IFF_BROADCAST	No	No	Yes	Yes
IFF_MULTICAST	No	No	Yes	Yes
IFF_MULTI_BCAST	No	No	No	Yes
IFF_ROUTER	Yes	**Yes***	Yes	Yes
IFF_IPV4	No	No	Yes	Yes
IFF_IPV6	No	No	Yes	Yes
IFF_FIXEDMTU	**Yes**	**Yes***	Yes	Yes

Table 2: Proposed Interface Flag Behavior ("*" denotes changed semantics)

5.3 IPMP-Specific Interface Flag Considerations

As just discussed, unlike the underlying physical interfaces, the following IPMP-specific flags will *never* be set on an IPMP group interface:

flag	meaning
IFF_STANDBY	Interface is administratively assigned to only be used if another interface fails
IFF_INACTIVE	Interface is functioning, but currently inactive and must not be used
IFF_OFFLINE	Interface is an unknown state and unavailable for use
IFF_NOFAILOVER	Address is administratively assigned as a test address and thus must not be used

Thus, applications will no longer encounter interfaces with these flags set, obviating the need to understand and respect the flags.

Although `IFF_NOFAILOVER` prevents an address from failing over, it does not currently prevent it from being used as a source address. As such, one must also set `IFF_DEPRECATED` to ensure that the source address selection algorithm will not prefer the address. While preserving the separation between `IFF_NOFAILOVER` and `IFF_DEPRECATED` is necessary (e.g., so that IPv6 data addresses can be deprecated as per the RFC's), an `IFF_NOFAILOVER` address should always be `IFF_DEPRECATED`. As such, setting `IFF_NOFAILOVER` will now automatically set `IFF_DEPRECATED`.

Because clearing `IFF_NOFAILOVER` on an `IFF_UP` address would require migration of an `IFF_UP` address, it will not be permitted. Since the only application that manipulates the `IFF_NOFAILOVER` flag is **`ifconfig`**, it will instead be changed to clear `IFF_UP` prior to setting `IFF_NOFAILOVER` and to restore `IFF_UP` afterwards, preserving administrative compatibility. To avoid confusing applications that may be managing an `IFF_NOFAILOVER` address, clearing `IFF_NOFAILOVER` on an address with `IFF_DHCPRUNNING` or `IFF_ADDRCONF` set will not be permitted.

The current IPMP design has an additional IPMP-specific flag, `IFF_FAILED`, which indicates that the interface has failed and thus can no longer be used. This flag is similar to the well-known and well-established `IFF_RUNNING` flag, but differs in that an external entity (in this case, **`in.mpathd`**) has determined that the interface is no longer fit for use. However, since `IFF_FAILED` is peculiar to Solaris, most applications do not consult it, defeating its purpose.

While both flags will remain, the IP module **will be updated to clear `IFF_RUNNING` whenever `IFF_FAILED` is set on the IPMP interface.** Since IPMP-unware applications will interact exclusively with IPMP interfaces, this change will allow them to work correctly without modification. As before, `IFF_RUNNING` and `IFF_FAILED` will remain independent on underlying interfaces, which will enable IPMP-aware applications (and administrators with established habits) to determine an underlying interface's link state without having to query DLPI.

The management of the IPMP interface's `IFF_RUNNING` and `IFF_FAILED` flags will be handled by the IP module, and will not be able to be manipulated by applications. In particular, the IP module will set `IFF_FAILED` and clear `IFF_RUNNING` when all of the underlying interfaces have `IFF_FAILED` set. As a degenerate case, since an IPMP interface with no underlying interfaces will not be able to send or receive data, it too will have `IFF_RUNNING` cleared and `IFF_FAILED` set.

As discussed in section 3.7, `IFF_INACTIVE` will now be managed by **`in.mpathd`**. To preserve existing semantics that an `INACTIVE` interface is *always* functioning, setting `IFF_INACTIVE` on an `IFF_FAILED` interface or vice-versa will fail. Further, to preserve the existing relationship between `IFF_INACTIVE` and `IFF_STANDBY`, setting `IFF_STANDBY` on a functioning interface will set `IFF_INACTIVE`, and clearing `IFF_STANDBY` will clear `IFF_INACTIVE` if set. In addition, to ensure that the `STANDBY` and `FAILBACK=no INACTIVE` semantics remain distinct, setting `IFF_STANDBY` will fail if `IFF_INACTIVE` is already set.

As previously discussed, one additional IPMP-specific flag, `IFF_IPMP`, will be set on all IPMP group interfaces to indicate that the interface has the special properties and semantics described throughout this document.

5.4 Routing Socket Interaction

5.4.1 Using Routing Sockets on the IPMP Group Interface

As mentioned previously, applications using routing sockets currently get an incomplete and confusing picture of the system's networking configuration. The IPMP group interface abstraction will fix this, since applications will now be able to monitor a routing socket for changes to the IPMP interface, rather than changes to the collection of underlying interfaces in the group. Specifically:

- The addition and removal of `UP` data addresses from the group will trigger `RTM_NEWADDR` and `RTM_DELADDR` events associated with the IPMP group.
- State changes associated with the group will trigger `RTM_IFINFO` events associated with the IPMP group.

However, applications will be insulated from irrelevant interface failures, repairs, and DR operations since:

- Addresses never migrate on the IPMP interface.
- State changes on the underlying interfaces will not cause `RTM_IFINFO` messages to be generated unless they affect the IPMP group interface as a whole (e.g., the last interface in the group has failed).

5.4.2 Using Routing Sockets on the Underlying Physical Interfaces

For consistency with our proposed changes to the socket discovery `ioctl` operations (section 5.1), **by default, routing sockets will not report events on interfaces in IPMP groups**, and messages sent to the kernel that refer to the underlying interfaces will fail with `EINVAL`. Further, when an interface is placed into an IPMP group, `RTM_DELADDR` messages will be generated for each of the `IFF_NOFAILOVER` addresses that are left behind, and an `RTM_IFINFO` message will be sent indicating that the interface is now down. Similarly, when an underlying interface is removed from an IPMP group, an `RTM_IFINFO` message will be sent indicating that the interface is now up, and `RTM_NEWADDR` messages will be generated for each `IFF_UP` address found on the interface. Note that since only `IFF_NOFAILOVER` addresses are permitted on interfaces in an IPMP group, all `RTM_NEWADDR` messages will be for `IFF_NOFAILOVER` addresses.

As with `SIOCGLIFCONF`, it is expected that most applications will wish to remain unaware of the underlying interfaces in an IPMP group. However, administrative applications which need to be aware of changes to the underlying interfaces will be able to set the new `RT_AWARE` socket option to the value `RTAW_UNDER_IPMP`. This will cause routing socket messages to:

- Be received for each change to an underlying interface in an IPMP group.
- Be able to be sent that refer to underlying interfaces in an IPMP group. However, since routes must be associated with the IPMP group interface, operations other than `RTM_GET` will fail with `EINVAL`.
- *Not* be generated when an interface is placed into or removed from an IPMP group (as described earlier).

As with `LIFC_UNDER_IPMP`, the `RT_AWARE` socket option will be documented, along with the values `RTAW_UNDER_IPMP` and `RTAW_DEFAULT` (aka "unaware"). Note that additional `RT_AWARE` values can be added in the future to further to align routing sockets with existing `SIOCGLIFCONF` behavior – e.g., `RTAW_ALLZONES` could mirror the existing `LIFC_ALLZONES` flag. Finally, since the `RT_AWARE` socket option will not be a generic `SOL_SOCKET` option, nor will it be tied to a specific IP protocol, a new `SOL_ROUTE` socket option level will be introduced and documented.

5.5 Routing Table Manipulation Operations

Solaris provides two interfaces for manipulating the routing table: the legacy BSD `SIOCADDRT` / `SIOCDELRT` `ioctl` operations, and routing sockets.

The legacy operations do not operate on interface names and thus are unaffected. However, as described in the last subsection, the routing socket interfaces *are* affected. For backward compatibility, an RTM_ADD request issued on an underlying interface will be automatically mapped by the IP module to an RTM_ADD request on the corresponding IPMP interface. Nonetheless, performing RTM_ADD requests directly on the IPMP interface will be strongly preferred. Other routing socket operations that manipulate the routing table (RTM_DELETE, and RTM_CHANGE) *must* be performed on the IPMP interface associated with the group. Attempting to issue these operations on an underlying interface will fail with EINVAL.

While this is a new restriction, it should have little impact in practice since the other changes described in this section effectively hide the underlying interfaces in a group from applications. Thus, the only way such an interface should become known to an application is if it is explicitly provided due to misconfiguration by an administrator. If future experience shows this restriction to be too strict, we can implement mapping functionality for all operations.

5.6 Interface Index Considerations

Currently, since an IPMP group is not modeled as a single interface, APIs that require an interface index are overloaded: they usually refer to *any* interface in the group, but may sometimes refer to a specific interface in the group. To help the kernel make the choice, several undocumented socket options (IP_DONTFAILOVER_IF, IPV6_DONTFAILOVER_IF, and IPV6_BOUND_PIF) have been introduced over the years.

With the introduction of the IPMP interface, each IPMP group will have its own interface index, retrieved with SIOCG[L]IFINDEX and set with SIOCS[L]IFINDEX. Thus, if an application enables the IP_RECVIF, IP_RECVPKTINFO, or IPV6_RECVPKTINFO socket options, then the ancillary data associated with packets sent to the data addresses in the group will have the index of the appropriate IPMP interface. Similarly, applications such as routing protocols can bypass the routing table and send a packet over an active interface in the group by specifying the group's IPMP interface index to the IP_PKTINFO, IPV6_PKTINFO, IP_BOUND_IF, or IPV6_BOUND_IF socket options.

Because the IPMP group interface and all IP interfaces under it will have their own interface indices, the standard socket options will enable applications to send or receive on any or all IP interfaces in a group[42]. As such, the aforementioned undocumented socket options will be rendered unnecessary and thus will be removed[43].

5.7 Interface Metric Considerations

Each IPMP interface will also have its own interface metric, retrieved with SIOCGLIFMETRIC and set with SIOCSLIFMETRIC. Since the IPMP interface metric is an administratively-assigned property that is not directly associated with any underlying interface, it will be set to zero when the IPMP interface is created. Because the routing metrics of underlying interfaces in an IPMP group will never be adjusted by routing daemons, issuing SIOCSLIFMETRIC on an underlying interface will fail with EINVAL.

[42] Since IPMP-unaware applications will not discover underlying interfaces, socket options will generally apply to the IPMP group interfaces.

[43] **in.mpathd** will use IP{,V6}_BOUND_IF in place of IP{,V6}_DONTFAILOVER_IF to send and receive probes.

5.8 Logical Interface Configuration

Logical interfaces will be able to be added or removed from an IPMP group interface using the `SIOCLIFADDIF` and `SIOCLIFREMOVEIF ioctl` operations. Once a new logical interface has been added to an IPMP group interface, it will be configurable using the other socket `ioctl` operations discussed throughout this section. Removing a logical interface from an IPMP group interface will cause the system to stop using any IP addresses that had been configured on it.

Logical interfaces will still be able to be created on or removed from the underlying interfaces in an IPMP group as well. This is primarily useful for adding or removing IPMP test addresses. However, if the `IFF_UP` flag is later set, any logical interfaces that are not marked `IFF_NOFAILOVER` will be brought up on the appropriate IPMP group interface instead (see section 5.2).

5.9 Address Manipulation Operations

As with any IP interface, IPv4 and IPv6 addresses will be able to be configured on an IPMP group interface through the following `ioctl` operations:

`ioctl`	**meaning**
`SIOC[GS][L]IFADDR`	Get or set IPv4 or IPv6 address
`SIOC[GS][L]IFNETMASK`	Get or set IPv4 netmask
`SIOC[GS][L]IFBRDADDR`	Get or set IPv4 broadcast address
`SIOC[GS]LIFTOKEN`	Get or set IPv6 address token
`SIOC[GS]LIFSUBNET`	Get or set IPv6 subnet prefix

These operations will still be able to be issued on the underlying interfaces in a IPMP group. However, if the `IFF_UP` flag is later set on the address and `IFF_NOFAILOVER` is not set, the address will be brought up on the appropriate IPMP group interface instead (see section 5.2).

5.10 Interface Parameter Operations

All of the documented `ioctl` operations to get and set interface parameters will be usable on an IPMP group interface:

`ioctl`	**meaning**
`SIOC[GS]LIFLNKINFO`	Get or set interface link information
`SIOC[GS]LIFMTU`	Get or set interface MTU
`SIOC[GS]LIFMUXID`	Get or set interface mux ID

These parameters will be used by IP when using the addresses configured on a given IPMP group. These parameters will also be able to be set or retrieved on the underlying interfaces, but their values will only apply to the test addresses configured on those interfaces.

5.11 ARP Table Configuration

Solaris provides two sets of interfaces to delete, get, and set (add) entries in the ARP table: `SIOC[DGS]ARP` and `SIOC[DGS]XARP`. The first set only allows an IP address and a hardware address to be specified; the second set also allows an IP interface to be specified.

5.11.1 ARP for On-Link Hosts

The current handling of `SIOC[DGS]ARP` and `SIOC[DGS]XARP` for on-link hosts is incorrect because each operation only applies to a specific interface in an IPMP group, rather than to *all* interfaces in the group. That is, if an ARP entry is established on one interface in an IPMP group, it cannot be used by another interface in the same group.

To fix this, in the future, on-link hosts reachable through an interface in an IPMP group will always be associated with the interface's corresponding IPMP interface. That is, `SIOC[DS]ARP` (or `SIOC[DS]XARP` with no specified interface) will delete or add an ARP entry for use by all interfaces in an IPMP group. Further, attempts to manipulate ARP entries on underlying interfaces via `SIOC[DS]XARP` will fail with `EPERM`. Finally, `SIOCG[X]ARP` will retrieve a given ARP entry; for `SIOCGXARP`, the IPMP group interface associated with the ARP entry will also be returned.

5.11.2 ARP for IPMP Data Addresses

As previously discussed, although IPMP data addresses will no longer be hosted on physical interfaces, the IP module will ensure that each usable IP data address in the IPMP group is associated with an underlying physical interface in the group (section 3.3). Thus, `SIOCG[X]ARP` will return an arbitrary hardware address of a physical interface in the group, and `SIOCGXARP` will return the IPMP group interface associated with the ARP entry. This behavior will not be a regression because the current behavior is not actually deterministic: nothing prevents the IPMP data address from migrating to another physical interface after the lookup operation.

Because the ARP entries associated with IPMP data addresses are system-managed, attempting to add or delete ARP entries for IPMP data addresses via `SIOC[DS][X]ARP` will fail with `EPERM`.

5.11.3 ARP for IPMP Test Addresses

Since IPMP test addresses remain associated with specific underlying interfaces, an `SIOCG[X]ARP` for an IPMP test address will always return the corresponding interface's hardware address; for `SIOCGXARP`, the associated underlying interface will also be returned. Because the ARP entries associated with IPMP test addresses are system-managed, attempting to add or delete ARP entries for IPMP test addresses via `SIOC[DS][X]ARP` will fail with `EPERM`.

5.11.4 Proxy ARP

Some sites make use of proxy ARP in order to allow a host to "pretend to be" another host – making it appear to be directly reachable from the perspective of other on-link hosts. Currently, the fact that ARP entries must be associated with underlying interfaces precludes proxy ARP from working with IPMP.

In the future, proxy ARP will work transparently with IPMP. Specifically, when `SIOCS[X]ARP` is performed, the system will check if the entry being added has a hardware address corresponding to an interface that is in an IPMP group, but has an IP address that does not correspond to any of the system's usable IP addresses. If so, the system knows that a proxy ARP entry is being established for an interface using IPMP, and checks to see if the interface associated with the hardware address is active. If so, or if there are no active interfaces in the IPMP group, the ARP entry will be added verbatim. Otherwise, the hardware address will be changed to that of an active interface in the IPMP group. From this point on, the proxy ARP entry will be treated like an ARP entry for an IPMP data address, and the system will adjust the entry's hardware address in order to keep the

IP address reachable (provided there is an active interface in the group).

As with ARP entries for on-link hosts, proxy ARP entries can be deleted with SIOCD[X]ARP and retrieved with SIOCG[X]ARP; for SIOCGXARP the IPMP group interface associated with the ARP entry will also be returned. Attempts to manipulate ARP entries on underlying interfaces via SIOC[DS]XARP will fail with EPERM.

5.12 Neighbor Discovery Configuration

Not surprisingly, configuration of the IPv6 neighbor discovery cache mirrors configuration of the ARP table. Specifically, SIOCLIFGETND, SIOCLIFSETND, and SIOCLIFDELND are used to get, set, and delete entries from the neighbor discovery cache. Like SIOC[GS]XARP, all three of these ioctls include a network interface name to associate the entry with, and also like ARP, the current architecture incorrectly forces operations to be applied to a specific interface in an IPMP group.

The new handling of these operations will be analogous to SIOC[SG]XARP: SIOCLIFGETND will always return the name of the IPMP group interface hosting the hardware address (e.g., ipmp0), and SIOCLIF{DEL,SET}ND requests associated with underlying interfaces will fail with EPERM. Similarly, because IPMP data addresses are system-managed, SIOCLIF{DEL,SET}ND requests for IPMP data addresses will fail with EPERM.

5.13 Zone Configuration

For a shared-stack zone, the SIOCGLIFZONE and SIOCSLIFZONE operations will still be used to get or set the zone associated with an address on an IPMP interface or an underlying interface in the group. Note that the zone only controls whether the given address can be used by the zone; as is true today, other interfaces in the group may be used to send or receive packets.

For an exclusive-stack zone, the datalinks themselves will still be assigned to the zone (through the private DLDIOC_SETZID ioctl), thus ensuring that any IP interfaces configured atop of them (and any resulting IPMP groups) are also in the same zone. Additionally, each exclusive stack will continue to have its own instance of in.mpathd, which will perform probe-based failure detection atop any addresses marked IFF_NOFAILOVER.

5.14 USESRC Manipulation

As discussed in section 3.19.4, OSPF-MP and IPMP will remain mutually exclusive for now. As such, one will not be able to use either an IPMP group interface or an underlying IPMP interface as the donor or recipient interface in SIOCSLIFUSESRC operations. Similarly, if an interface has been configured with SIOCSLIFUSESRC, attempting to place it into an IPMP group will fail. Since OSPF-MP and IPMP are mutually exclusive, the SIOCGLIFUSESRC and SIOCGLIFSRCOF operations will trivially succeed when used with IPMP group interfaces or underlying IPMP interfaces. Note that if support for OSPF-MP on IPMP group interfaces is added in the future, the restrictions on the IPMP group interface will be removed.

5.15 Multicast Interaction

From an application's perspective, as with unicast, all multicast traffic will be associated with the IPMP interface rather than its underlying physical interfaces. Specifically, since all IP data addresses will be hosted on IPMP group interfaces, the address provided in an IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, or IP_MULTICAST_IF socket option will be mapped to the IPMP group interface. Similarly, the application will specify the interface index associated with the appropriate IPMP interface when issuing the IPV6_JOIN_GROUP, IPV6_LEAVE_GROUP, and IPV6_MULTICAST_IF socket options. Additionally, the application will specify the interface index associated with the appropriate IPMP interface in the group_req and group_source_req structures associated with the MCAST_JOIN_GROUP, MCAST_LEAVE_GROUP, MCAST_JOIN_SOURCE_GROUP, MCAST_LEAVE_SOURCE_GROUP, MCAST_BLOCK_SOURCE, and MCAST_UNBLOCK_SOURCE socket options.

When an IP interface is placed into an IPMP group, any existing multicast memberships on that interface are torn down. This may seem surprising, but it is necessary to ensure consistency with the application's view that the IP interface which they had joined on "no longer exists". Moreover, long-running applications will track the functioning set of IP interfaces (either via routing sockets or by polling SIOCGLIFCONF), and thus will simply re-join their multicast group on the new IPMP interface (if need be). Since multicast memberships must be consistent across the group, attempts to establish new multicast memberships on specific underlying interfaces will fail with EINVAL.

As per section 3.13, the IP module will nominate one active interface in the IPMP group to send and receive multicast traffic, and to apply any needed multicast filtering. If that interface subsequently fails, the multicast membership and filtering will be moved to another active interface in the group, if one exists. When sending multicast packets, the nominated interface's multicast mapping (visible in section 4.9) is used to determine the destination link-layer multicast address.

5.16 Routing Table Bypass Socket Options

The IP_NEXTHOP, IPV6_NEXTHOP, and SO_DONTROUTE socket options all enable the routing table to be bypassed when sending a packet. In a non-IPMP configuration, the on-link destination specified when enabling either IP_NEXTHOP or IPV6_NEXTHOP – or when sending a packet after enabling SO_DONTROUTE – also indirectly specifies an outgoing interface. In the new model, this will also indirectly specify an outgoing interface, though that interface may be an IPMP group interface, which may use any active underlying interface to actually transmit on. Since no applications other than in.mpathd should be generating IPMP test traffic, IP_NEXTHOP, IPV6_NEXTHOP, and SO_DONTROUTE will not necessarily be honored if the specified address is on a network that is only accessible through the system's IPMP test addresses.

5.17 IPMP ioctl Operations

To enable applications such as in.mpathd and ifconfig to set or get the kernel's IPMP state, there are a collection of undocumented IPMP-related ioctl operations. Specifically, the following private IPMP-related ioctl operations are currently supported:

ioctl	meaning
SIOCLIFFAILOVER	Perform an interface failover
SIOCLIFFAILBACK	Perform an interface failback
SIOCSIPMPFAILBACK	Enable or disable FAILBACK=no
SIOC[GS]LIFOINDEX	Get or set interface's original index
SIOC[GS]LIFGROUPNAME	Get or set interface's IPMP group

In the new model, most of these ioctls are now unnecessary. Specifically, because the kernel now manages address distribution, the SIOCLIFFAILOVER and SIOCLIFFAILBACK operations no longer make sense and will be removed. Further, as discussed in section 3.8, SIOCSIPMPFAILBACK will no longer be necessary and will be removed (along with the lifr_movetoindex member of the `struct lifreq`). Similarly, as discussed in section 3.6, the `SIOCGLIFOINDEX` and `SIOCSLIFOINDEX` operations will also be rendered unnecessary and will be removed.

`SIOCGLIFGROUPNAME` and `SIOCSLIFGROUPNAME` will remain. However, a number of new requirements will need to be met for `SIOCSLIFGROUPNAME` to successfully place an interface in a new group (e.g., it must not be in a group yet, and the IPMP interface for the group must exist), which will simplify a number of IPMP codepaths in the kernel. Since `SIOCSLIFGROUPNAME` is private to the IPMP subsystem, these changes pose no compatibility problems[44].

The new model will also necessitate two new private IPMP `ioctl` operations:

`ioctl`	**meaning**
`SIOCGLIFBINDING`	Get inbound interface binding for address
`SIOCGLIFGROUPINFO`	Get IPMP group information

The `SIOCGLIFBINDING` operation will be needed for `ipmpstat -a` to report the inbound interface binding for a given address (since those bindings will be under kernel control). It will expect a `struct lifreq` with the `lifr_name` set to the logical interface of the address to look up, and return the result in a new `lifr_binding` member, which will be part of the existing `union lifr_lifru`.

The `SIOCGLIFGROUPINFO` operation will be needed by `in.mpathd` and `ifconfig` for a variety of tasks, including quickly mapping an IPMP group to its IPMP interface, obtaining the nominated multicast and broadcast interfaces for a group, and determining whether a given IPMP group is currently in-use. It will expect a `struct lifgroupinfo`, which will be a new structure with contents private to the IPMP subsystem.

5.18 MIB II Interaction

Network interface names and indices are also exposed through several tables in the Networking MIB. While access to the Networking MIB is not currently documented, numerous popular third-party applications have discovered how to access it in order to obtain information about the networking stack. In addition, internal utilities such as `netstat` make extensive use of it to display information such as the routing and ARP tables.

The following MIB II structures have fields which expose network interface names or indices:

MIB structure name	**structure purpose**
`mib2_ipAddrEntry`	IPv4 address information
`mib2_ipv6AddrEntry`	IPv6 address information
`mib2_ipRouteEntry`	IPv4 routing table entry
`mib2_ipv6RouteEntry`	IPv6 routing table entry
`mib2_ipNetToMediaEntry`	ARP table entry
`mib2_ipv6NetToMediaEntry`	ND table entry
`mib2_ipv6IFStatsEntry`	IPv6 interface statistics
`mib2_ipv6IfIcmpEntry`	IPv6 ICMPv6 interface statistics
`mib2_tcp6ConnEntry`	TCP over IPv6 connection information
`mib2_udp6Entry`	UDP over IPv6 listener information
`ip_member`	IPv4 multicast membership
`ipv6_member`	IPv6 multicast membership
`ip_grpsrc`	IPv4 multicast source filtering
`ipv6_grpsrc`	IPv6 multicast source filtering

[44] With the exception of the Sun Cluster use covered in section 3.17.

In almost all of the above cases, the network interface names and indices returned through these structures will refer to the IPMP group interface, rather than the underlying interface. The only exceptions will be for addresses, routing table entries and hardware address mappings that are explicitly tied to an `IFF_NOFAILOVER` address, which will use the interface name or index of the underlying interface the address is hosted on. To prevent IPMP-unaware applications from tripping over routing table entries only used for test traffic, such entries will *not* be returned unless an application specifies a new synthetic MIB II level (`EXPER_IP_AND_TESTHIDDEN`) when issuing the report request. Since such information is chiefly useful only to `in.mpathd`, we do not anticipate unbundled applications will need to make use of this synthetic level.

5.19 Kstats

As previously discussed, the IPMP interface will have kstats, which can be retrieved using the documented `kstat(3KSTAT)` interfaces. However, many networking kstats are hardware-specific and thus do not apply to the IPMP group interface. See section 4.8 for a list of the kstats that will be supported.

5.20 DLPI Interaction

As discussed in section 3.15, because each data-link in an IPMP group maintains its own identity, it is not semantically feasible to provide data-link (DLPI) access to the IPMP interface as a whole. In particular, one would need to make significant changes to the core DLPI model to allow each data-link to be individually accessed (e.g., by allowing a DLPI consumer to attach to a particular data-link, or to send a packet out a particular data-link using a particular hardware address), or make incompatible changes to the IPMP model to make all interfaces in a group appear homogeneous (e.g., a single hardware address a la 802.3ad).

As such, **DLPI access to an IPMP group is not planned**.

However, the Solaris IP stack expects all IP interfaces (other than loopback) to terminate in a DLPI device, on top of which the IP kernel module is pushed. Thus, in order to satisfy this architecture, a stub driver called `dlpistub` will be implemented and be visible in the filesystem as `/dev/ipmpstub`. This driver will be an enhanced version of the current `vni(7D)` stub driver that is used for OSPF-MP support, implementing a few core STREAMS and DLPI messages that IP expects. As such, `dlpistub` will also be able to export itself as `/dev/vni` for `vni(7D)` support[45], and the existing `vni` DLPI stub driver will be able to be removed. Note that `/dev/ipmpstub` will be provided strictly for implementation convenience. It **will not be able to send or receive packets, nor will it be documented for public use**. The name `/dev/ipmp` was intentionally avoided to prevent administrators from being misled into believing that `snoop -d ipmp0` will work.

5.21 `libipmp`

The `libipmp` library currently provides an undocumented *query* API used to snapshot and query the IPMP subsystem, as originally described in PSARC/2002/615. It was initially created for use by Sun Cluster and the IPMP test suite, but was never used by Sun Cluster. In the new architecture, the query API provides an elegant foundation for implementing `ipmpstat`, though the API will need to be revised to accommodate the proposed changes to the IPMP subsystem. Because it is currently only used by the IPMP test suite (which will itself undergo extensive changes), when necessary, elegance of the resulting API will be prioritized over backward compatibility.

[45] `/dev/vni` and `/dev/ipmpstub` will be identical except for the DLPI type they export.

In addition, `libipmp` provides a logical place to house an additional *administrative* API that can be used to control the IPMP subsystem. This API will be used by `ifconfig`, `if_mpadm`, and `ip_rcm`, which will allow large chunks of duplicated code to be removed from those applications and modules, and will ensure that all administrative IPMP operations are handled consistently. With the introduction of the administrative API, all communication with `in.mpathd` will be performed through `libipmp`, which enables the previously-exposed IPC mechanism used between `in.mpathd` and `libipmp` to become private to the implementation.

A separate specification details the complete `libipmp` API; this document only covers the high-level changes and enhancements.

5.21.1 `libipmp` Query API

The existing set of functions (declared in <ipmp_query.h>) will remain unchanged:

routine	purpose
`ipmp_setqcontext()`	Set the IPMP query context (live or snapshot)
`ipmp_getgrouplist()`	Get list of interfaces monitored by `in.mpathd`
`ipmp_freegrouplist()`	Free list returned from `ipmp_getgrouplist()`
`ipmp_getgroupinfo()`	Get IPMP group information
`ipmp_freegroupinfo()`	Free information returned from `ipmp_getgroupinfo()`
`ipmp_getifinfo()`	Get IPMP-related interface information
`ipmp_freeifinfo()`	Free information returned from `ipmp_getifinfo()`

However, the `ipmp_groupinfo_t` returned by `ipmp_getgroupinfo()` will be extended to also include the group's failure detection time, IPMP group interface name, data addresses, and nominated broadcast and multicast interface names. Similarly, the `ipmp_ifinfo_t` structure returned by `ipmp_getifinfo()` will be extended to also include the interface's test address (if any), probe target mode, probe targets (if any), and probe- and link-based failure detection status.

In addition, a new `ipmp_getaddrinfo()` function will be added to provide IPMP data address information via a new `ipmp_addrinfo_t` structure. This structure will contain the address's IPMP group, state (either up or down), and inbound underlying interface binding (if any). As with the existing query functions, `ipmp_getaddrinfo()` will return either `IPMP_SUCCESS`, or an IPMP error code on failure. A new `IPMP_EUNKADDR` error code will be returned if the requested address is not an IPMP data address. Also mirroring the existing functions, a new `ipmp_freeaddrinfo()` function will free an `ipmp_addrinfo_t` structure.

routine	purpose
`ipmp_getaddrinfo()`	Get IPMP data address information
`ipmp_freeaddrinfo()`	Free information returned from `ipmp_getaddrinfo()`

5.21.2 `libipmp` Administrative API

The proposed administrative API will initially consist of just three undocumented functions, which will be declared in a new <ipmp_admin.h> header file:

routine	purpose
`ipmp_ping_daemon()`	Check if `in.mpathd` is operational
`ipmp_offline()`	Offline an underlying interface
`ipmp_undo_offline()`	Undo a previous offline of an underlying interface

The `ipmp_ping_daemon()` routine will enable an application to easily check if `in.mpathd` is running. The `ipmp_offline()` routine will attempt to offline the specified IP interface; the caller must also specify the minimum number of usable interfaces in the group that may remain after the offline operation. If the interface is not under IPMP control, or if the offline would lower the number usable interfaces below the minimum, the operation will fail. Finally, the `ipmp_undo_offline()` routine allows a previously-offlined IP interface to be brought back online. If the interface was not previously offlined, or if the interface cannot be brought back online (e.g., because its hardware address is not unique across the group), the operation will fail.

5.22 IPMP Asynchronous Events Enhancements

In addition to the query API described above, there's also an undocumented set of sysevents that enable IPMP subsystem state changes to be observed, as originally described in PSARC/2002/137. As with the query API, it was created for use by Sun Cluster and the IPMP test suite, but was never used by Sun Cluster.

With the rearchitecture, the existing set of IPMP sysevents will be enhanced slightly to account for new states (such as the "degraded" group state) and new configurations (such as an empty group), but will be otherwise unchanged. In addition, one new IPMP sysevent, `ESC_IPMP_PROBE_STATE`, will be introduced to provide state change notifications for each IPMP probe packet, which will provide the underlying mechanism for `ipmpstat -p`. To ensure that `syseventd` and other sysevent consumers do not become saturated with `ESC_IPMP_PROBE_STATE` sysevents on large IPMP configurations, `in.mpathd` will switch from using the legacy sysevent mechanism to the "General-Purpose Event Channels" (GPEC) sysevent mechanism[46].

In particular, a new `com.sun:ipmp:events` event channel will be introduced to carry IPMP sysevents. Because of these changes, the existing `IPMP_EVENT_VERSION` field of each IPMP sysevent will be increased to the value `2`.

5.23 Miscellaneous Library API Enhancements

5.23.1 `if_nametoindex(3SOCKET)` and `if_indextoname(3SOCKET)`

Since the IPMP group interface is an IP interface, the existing `if_nametoindex(3SOCKET)` and `if_indextoname(3SOCKET)` routines will be able to map an IPMP group interface name to its interface index, and vice-versa. More subtly, the APIs will also continue to work for underlying interfaces in an IPMP group (rather than fail with `ENXIO`), on the grounds that the caller has already discovered the interface name or index (e.g., through an `SIOCGLIFCONF` with `LIFC_UNDER_IPMP`).

5.23.2 `if_nameindex(3SOCKET)`

The rarely-used and oddly-named `if_nameindex(3SOCKET)` routine (which actually retrieves the system's IP interface names and indices) will retrieve IPMP group interfaces, but to avoid confusing IPMP-unaware applications, it will not retrieve underlying interfaces in an IPMP group.

[46] In particular, the GPEC API provides fine-grained event channels which do not flow through `syseventd`; see PSARC/2002/321 for details.

5.23.3 ifaddrlist()

The undocumented `libinetutil` library currently has an "enhanced" (and incompatible) version of BSD's `ifaddrlist()` routine, which retrieves a list of IPv4 or IPv6 addresses currently configured on the system, along with their associated IP interface name, flags, and index. Currently, `ifaddrlist()` builds the list of addresses by calling `SIOCGLIFCONF` with `lifc_flags` set to zero, which means that `IFF_NOXMIT`, `IFF_TEMPORARY`, and all-zones addresses are never retrieved. Similarly, without modification, interfaces under IPMP would never be retrieved in the future.

To address this limitation, `ifaddrlist()` will be further enhanced to take an additional `lifc_flags` argument. Passing zero for this argument will match current behavior; passing other values (e.g., `LIFC_UNDER_IPMP`) will allow addresses tied to various "hidden" IP interfaces to be retrieved.

5.23.4 ifaddrlistx() and ifaddrlistx_free()

An additional undocumented `ifaddrlistx()` convenience function will be added to `libinetutil`. This routine will be similar to `ifaddrlist()`, but will only retrieve addresses for a specific interface name that have a specific set of address flags set and clear. In addition, it will return a linked-list of `ifaddrlistx_t` structures (rather than an array) so that the `ifaddrlistx_t` can grow in the future without affecting binary compatibility. An accompanying `ifaddrlistx_free()` function will be provided to free the linked-list. As with `if_nametoindex()`, `ifaddrlistx()` will succeed for underlying interfaces in an IPMP group on the grounds that the caller has already discovered the interface name. A separate specification covers the details of these interfaces; they are mentioned here only for completeness.

5.23.5 sockaddrcmp()

An additional undocumented `sockaddrcmp()` convenience function will be added to `libinetutil`. This function takes two pointers to `struct sockaddr_storage`, and returns `B_TRUE` if they contain the same IPv4 or IPv6 address.

6 Summary

In closing, we briefly revisit our original set of requirements, and verify that the proposed model satisfies them. Taking the list from the top:

1. IP addresses in an IPMP group must never migrate as part of failure, repair, offline, and undo-offline operations.
2. IP addresses in an IPMP group must be administered using the same tools and in the same manner as IP addresses not placed in an IPMP group. As a corollary, address autoconfiguration through DHCP must work.

Since all data address will be hosted on IPMP interfaces, address migration never occurs as a result of failure, repair, offline, and undo-offline operations. Further, because the data addresses exist on stable logical interfaces, it is possible to administer them just like any other IP address. Enhancements made to `dhcpagent` and `in.ndpd` will allow address autoconfiguration to work even though the addresses are not associated with any specific hardware.

3. The IPMP group's core configuration and current state must be able to be easily and concisely obtained.

The new `ipmpstat` utility will allow the administrator to examine the state and configuration of the IPMP subsystem on a variety of levels. Specifically, IPMP groups, IPMP data addresses, IPMP interfaces, IPMP probe targets, and IPMP-related probes can all be observed in real-time, providing insight into a previously-opaque subsystem.

4. The IPMP group as a whole must be observable through packet monitoring tools such as `snoop(1M)`. It must also still be possible to observe individual interfaces as well.
5. The IPMP group as a whole must be able to administered as a whole – e.g., specified directly to `route(1M)`, and visible directly through `netstat(1M)`. The Clearview network interface requirements must also be met.

By modeling IPMP groups as IP interfaces, Clearview's IP Observability component can be leveraged to provide packet monitoring of the IPMP group as a whole. Further, representing IPMP groups as interfaces will enable IPMP groups to be specified directly to `route(1M)`, and to be visible through `netstat(1M)`, along with a variety of other tools. In order to meet Clearview's network interface requirements, a new `ipmp ifconfig` subcommand will allow the administrator to name an IPMP group interface at creation time.

6. IPMP test addresses must not be visible to applications unless explicitly requested.

By making a collection of changes to `SIOCGLIFCONF` and routing sockets semantics, networking applications will be shielded from test addresses, unless they explicitly request to see them. Since the revised semantics will become the default, all applications will automatically benefit with no coding changes.

7. If possible, address migration must be obviated (to prevent complex migration code in the kernel).

Where necessary, migration of an address will be modeled as the removal of the address on one interface followed by the insertion of the address on another interface, allowing the current complex migration code to be removed.

8. The documented IPMP administrative commands (`if_mpadm`, `ifconfig`, and `cfgadm`) must continue work as before.
9. The documented IPMP semantics, e.g., for failure detection, repair detection, outbound interface selection and source address selection must continue to operate as before.
10. The documented IPMP configuration files (`/etc/hostname.`*if* files) must continue to operate as before.

As outlined throughout this proposal, the documented behavior of all administrative commands, semantics, and configuration files will be preserved. However, a new administrative model centered around the IPMP group interface is also proposed, and administrators will be encouraged to move to the new administrative model so that they can take full advantage of the new IPMP architecture.

Glossary of IPMP Terminology

Active Interface: An underlying interface that is currently being used by the system to send or receive data traffic. An interface is active if it has at least one `UP` address, and the `FAILED`, `INACTIVE`, and `OFFLINE` flags are *not* set. Compare: usable interface, functioning interface, `INACTIVE` interface.

Address Migration: The act of moving an address from one network interface to another network interface. Currently, address migration frequently occurs, most often as part of failover or failback. In the new model, migration of an `UP` address will never occur, drastically simplifying the implementation.

Data Address: An IP address which can be used as the source or destination address for data. Data addresses are part of an IPMP group and can be used to send and receive traffic on *any* interface in the group. Moreover, the set of data addresses in an IPMP group can continue to be used as long as one interface in the group is active. Currently, data addresses are hosted on the underlying interfaces of an IPMP group. In the new model, data addresses will be hosted on the IPMP interface. Compare: Test Address.

`DEPRECATED` **Address:** An IP address which should not be used as the source address for data. Typically, IPMP Test Addresses are `DEPRECATED`, but any addres may be marked `DEPRECATED` to discourage it from being used as a source address.

Explicit IPMP Interface Creation: In the new model, the act of explicitly defining an IPMP interface by using the `ipmp` subcommand of `ifconfig`. Explicit IPMP interface creation is preferred since it allows the IPMP interface name and IPMP group name to be set by the administrator. Compare: Implicit IPMP Interface Creation.

Failback: Currently, the act of undoing a previous failover operation, once the previously-failed interface has been repaired. The concept of failback will not exist in the new model.

`FAILBACK=no` **Mode:** A mode which minimizes rebinding of incoming addresses to interfaces by avoiding redistribution during interface repair. This mode is currently broken, but will be fixed in the new model. Specifically, when an interface repair is detected, `FAILED` will be cleared, but if another interface in the group is functioning, `INACTIVE` will be set, preventing its use. However, if another interface in the IPMP group fails, the `INACTIVE` interface will be eligible to take over for the failed interface. While the concept of failback will not exist in the new model, the name of this mode will be preserved for administrative compatibility.

`FAILED` **Interface:** An interface which `in.mpathd` has determined to be malfunctioning The `FAILED` flag will be set on any `FAILED` interface. Opposite: functioning interface.

Failover: Currently, the act of migrating the data addresses on a failed interface to another functioning interface in an IPMP group. In the new model, all data addresses will be hosted on the IPMP interface, and thus be unaffected by interface failure. Thus, the concept of failover will not exist in the new model.

Failure Detection: The act of detecting that a physical interface in the group has failed. Two forms of failure detection are implemented: link-based failure detection, and probe-based failure detection.

Functioning Interface: An interface which `in.mpathd` has determined to be healthy. This includes both active interfaces and `INACTIVE` interfaces. All functioning interfaces will have the `FAILED` flag clear. Opposite: `FAILED` interface.

`INACTIVE` **Interface:** A functioning interface which must not be used according to administrative policy. All `INACTIVE` interfaces will be have the `INACTIVE` flag set. Compare: active interface, usable interface.

Implicit IPMP Interface Creation: In the new model, the act of implicitly defining an IPMP interface by placing an underlying interface in an IPMP group that does not yet exist using `ifconfig`. Implicit IPMP interface creation is provided for backward compatibility, and as such does not provide the ability to set the IPMP interface name or IPMP group name. Compare: Explicit IPMP Interface Creation.

Inbound Load-Spreading: The act of spreading inbound traffic across the set of interfaces in an IPMP group. Inbound load-spreading cannot be controlled directly with IPMP, but is indirectly manipulated by the source address selection algorithm.

IPMP Anonymous Group Support: An IPMP feature enabled by setting `TRACK_INTERFACES_ONLY_WITH_GROUPS` to `no` in `/etc/default/mpathd`. When this feature is enabled, `in.mpathd` tracks the health of all network interfaces in the system, regardless of whether they belong to an IPMP group. However, since the interfaces are not actually in an IPMP group, the addresses on these interfaces will not be available upon interface failure.

IPMP Group: A set of network interfaces that are treated as interchangeable by the system in order to improve network availability and utilization. Each IPMP Group has a set of data addresses that the system may associate with any set of active interfaces in the group in order to maintain network availability and improve network utilization. The administrator can select which interfaces to place into an IPMP group, but all interfaces in the same group must share a common set of properties, such as being attached to the same link and configured with the same set of protocols (e.g., IPv4 and IPv6).

IPMP Group Interface: See: IPMP Interface.

IPMP Group Name: The name of an IPMP group, as assigned with the `ifconfig group` subcommand. All underlying interfaces with the same IPMP group name are defined to be part of the same IPMP group. In the new model, IPMP group names will be deemphasized in favor of IPMP interface names, and administrators will be encouraged to use the same name for both.

IPMP Interface: In the new model, the IP interface that represents a given IPMP group, any or all of its underlying interfaces, and all of its data addresses. The IPMP interface will become the core administrative abstraction for an IPMP group, and will be used in routing tables, ARP tables, firewall rules, and so forth.

IPMP Interface Name: In the new model, the name of an IPMP interface. By convention, this document uses the naming convention of `ipmp`*N*, which will also be used by the system via implicit IPMP interface creation. However, the administrator will be able to choose any name by using explicit IPMP interface creation.

IPMP Singleton: An IPMP group containing only a single IP interface. IPMP Singleton is typically used by Sun Cluster, which in turn provides higher level failover. Traditionally, IPMP Singleton has also allowed a data address to also act as a test address, but this incompatible with the new model and will no longer be supported.

Link-Based Failure Detection: A passive form of failure detection, in which the link status of the network card (as reported by the network card's driver) is monitored in order to determine an interface's health. Link-based failure detection only tests whether the link is up, and is not supported by all network card drivers. However, it requires no explicit configuration and provides instantaneous detection of link failures. Compare: Probe-Based Failure Detection.

Load-Spreading: The act of spreading inbound or outbound traffic over a set of interfaces. Unlike load balancing, there is no guarantee that the load will be *evenly* distributed.

`NOFAILOVER` **Address:** An address that is associated with an underlying interface and thus will not remain available if the underlying interface fails. All `NOFAILOVER` addresses will have the

NOFAILOVER flag set on them. IPMP test addresses *must* be NOFAILOVER, and IPMP data addresses *must never* be NOFAILOVER. Although the concept of failover will not exist in the new model, the name NOFAILOVER will remain for administrative compatibility.

OFFLINE **Interface:** An interface which is unavailable for system use, usually in preparation for being removed from the running system. All OFFLINE interfaces will have the OFFLINE flag set.

Offline Operation: The act of making an interface unavailable for system use, usually in preparation for removing it from the running system. The if_mpadm(1M) command can be used to offline an interface.

Outbound Load-Spreading: The act of spreading outbound traffic across the set of interfaces in an IPMP group. Outbound load-spreading is performed on a per-destination basis by the IP module, and adjusted as necessary based on the health and members of the interfaces in the IPMP group.

Physical Interface: See: Underlying Interface.

Probe: An ICMP packet, similar to the ones used by ping(1M), used to test the send and receive paths of a given interface. Probe packets will be sent by in.mpathd if probe-based failure detection is enabled. A probe packet will use an IPMP test address as its source address.

Probe-Based Failure Detection: An active form of failure detection, in which probes are exchanged with probe targets in order to determine an interface's health. Probe-based failure detection tests the entire send and receive path of each interface it is enabled on, but requires the administrator to explicitly configure each interface with a test address. Compare: Link-Based Failure Detection.

Probe Target: A system on the same link as an interface in an IPMP group, selected by in.mpathd to help test the health of a given interface using probe-based failure detection. The probe target may be *any* host on the link capable of sending and receiving ICMP probes. Probe targets are usually routers, and several probe targets are usually used in order to insulate the failure detection logic from failures of the probe targets themselves.

Source Address Selection: The act of selecting a data address in the IPMP group as the source address for a particular packet. Source address selection is performed by the system whenever the application has not specifically selected a source address to use. Since each data address is associated with only one hardware address, source address selection indirectly controls inbound load-spreading.

STANDBY **Interface:** An interface which has been administratively configured to be used only when another interface in the group has failed. All STANDBY interfaces will have the STANDBY flag set.

Test Address: An IP address which *must* be used as the source or destination address for probes, and *must not* be used as a source or destination address for data traffic. Test addresses are associated with an underlying interface, and are marked NOFAILOVER so that they remain on the underlying interface even if it fails, in order to facilitate repair detection. Since test addresses will not be available upon interface failure, all test addresses *must* be marked DEPRECATED to keep the system from using them as a source addresses for data packets.

Underlying Interface: An IP interface that is part of an IPMP group and directly associated with an actual network device. For instance, if ce0 and ce1 are placed into IPMP group ipmp0, then they comprise the underlying interfaces of ipmp0. Currently, IPMP groups consist solely of underlying interfaces. However, in the new model, these interfaces will *underly* the IPMP interface (e.g., ipmp0) that represents the group, hence the name.

Undo-Offline Operation: The act of enabling a previously-offlined interface, so that it is again available for system use. The `if_mpadm(1M)` command can be used to perform an undo-offline operation.

Usable Interface: In the new model, an underlying interface which could be used to send or receive data traffic. While all active interfaces are usable, not all usable interfaces are active (e.g., `INACTIVE` interfaces). An interface is usable if it has at least one `UP` address, and has does not have `FAILED` or `OFFLINE` set. Compare: active interface, `INACTIVE` interface. Opposite: unusable interface.

Unusable Interface: An interface is unusable if it has no `UP` addresses, or has `FAILED` or `OFFLINE` set. Opposite: usable interface.

`UP` **Address:** An address which has been made administratively available to the system by setting its `UP` flag. An address which is not `UP` will be treated as not belonging to the system, and thus will never be considered during source address selection.