1 IPMP-Tutorial Sneak Preview

Failure is not an option...

(Gene Kranz, Flight Director Apollo 13)

January 21, 2010-19:36

1.1 The bridges at SuperUser Castle

root was sitting in SuperUser castle and everything was fine in the kingdom. But then a loud squeaking and creaking noise found root's attention. The demons of hypertext wrote into the Scrolls of Log, that they couldn't fulfill their work any longer, as the sole bridge into the vast kingdom of root lowered at this moment was broken.

Root spoke: "Use the other bridge ... there are two for a reason. Do I have to think for you all?". But the demons replied: "We can't do that ... only the infinite power of root can lower the bridge". Thus root lowered the bridge but thought "I have to do more important things than lowering bridges".

Thus root spoke a chant of infinite power and a daemon was spawned from the ether. root told the daemon "You are the guardian of the link! Protect it. Guard it. And when everything else fails, you are allowed to lower the second bridge to SuperUser castle."

1.2 Introduction

Before people start to think about clusters and load balancers to ensure the availability, they should start with the low hanging fruits. Such a low hanging fruit is the protection of the availability of the network connection. Solaris has an integrated mechanism to ensure this availability. It's called IP Multipathing.

IP Multipathing is an important part of the solution for an ever reoccurring problem , as almost all applications interact with the outside world on one way or the other. Thus ensuring the mechanisms of communication is a part of almost all architectures.

Even when you have other availability mechanisms like balancers, you want to use a protection of the IP connection out of a simple reason: Many applications have a session context and not all software architectures can replicate those session contexts to another system to enable a failover without loosing the session. So do you really want to loose this context just because of a failing network card or because of a admin unplugging a cable? Or do you really want to provoke a cluster failover because of a failing network card? IPMP can keep such failures on a low level without needing high-availability mechanisms with a much larger impact.

Out of this reason IP Multipathing is an important part for most HA infrastructures. This tutorial wants to give you an introduction in this topic. It's not really an "less known feature" because for many people working with Solaris, IPMP is a daily part of their work. But many people new to Solaris or OpenSolaris aren't aware of the fact that Solaris has an integrated mechanism for IP Multipathing[1]. Furthermore this tutorial wants to give some insights into new developments in the field of IP Multipathing.

1.2.1 Where should I start?

This tutorial will explain two mechanisms, because the realm of "IP Multipathing" is a topic in flux at the moment. The implementation in Solaris 10 and older releases of Opensolaris (before Build 107) is vastly different to the implementation in current releases of Opensolaris(Build 107 and up).

I thought a while about the problem, what method should make the start in this tutorial. At the end I decided to explain the new IPMP mechanism first as the concepts of multipathing are a little bit more obvious in the new implementation.

1.2.2 Basic Concept of IP Multipathing

The fundamental concepts of both implementations are pretty much the same, thus I will start with a short introduction into the nomenclature of IPMP:

- **IP Link:** An IP link is the logical connection to an IP network. Think of a router, that has to legs ... one to the Internet and one to the inside network. Such a router has two IP links (even when the router has multiple connections to both networks).
- **Physical Interface:** The physical interface is the foundation of all networking, but it isn't really the basic entity in IPMP. The basic entity is the IP interface that is bound to a physical interface. Or to simply it: It's the IP address, not the cable

[1] As well as most newbies to Solaris aren't aware of MPxIO ... the counterpart of IPMP for storage

that is managed by IPMP. Of course, you need physical interfaces. At best two or more of them, because with one path you can't do multipathing.[2]

- **IPMP Group:** Now you have physical interfaces on some network interface cards into several IP Links. How do you tell the system, that certain interfaces are redundant connections into the same IP link?

  The concept of the IPMP group solves this problem. You put all interfaces into a IPMP group that connect into an IP link into a IPMP group. All interfaces in this group are considered as redundancy to each other, so the IPMP can use them to receive and transmit the traffic out of this network.

- **Failure:** Okay, you may think, this one is so obvious you don't have to talk about it. Well, not really. This is one of the most frequent errors in HA. Buying or using a HA product without thinking about the failure modes that are addressed by the mechanism.

  You have to think about what failures are in-scope of IPMP and which one are out-of-scope. IPMP is called IP Multipathing for a reason: It's a tool for IP, it isn't meant for other protocols. So it protects the availability of IP services against failures. But for this task it uses information of other layers of the stack, for example the information if there is a link on the physical connection. Primarily it uses this information to speed up failover. There is no need to check upper layers if you already know that lower layers went away.

- **Failure Detection:** You do IPMP for a reason. You want protect your system from loosing its network connection in the case a networking component fails. One of the most important component of an automatic availability protection mechanism is it's capability to detect the need of doing something like switching the IP configuration to another physical interface. Without such a mechanism it's just an easier interface to switch such configuration manually.

  That said, IPMP provides two mechanisms to detect failures:

  - **Link based:** As the name suggests, the link based failure detection checks if the physical interface has an active and operational link to the physical network. When a physical interface looses its link - for example by problems with the cabling or a switch powered down - IPMP considers the interface as failed and starts to failover to a operational link.

    The monitoring mechanism for the link state is quite simple. It's done by monitoring the `RUNNING` flag of an IP interface. When you look at a functional interface with `ifconfig` you will recognize this flag:

[2] Albeit I can think of remote cases were IPMP with one path can be useful

```
e1000g0: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,
    NOFAILOVER,CoS> mtu 1500 index 11
        inet 192.168.56.201 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
```

When you unplug the cable, the `RUNNING` flag is missing:

```
e1000g0: flags=219040803<UP,BROADCAST,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    FAILED,CoS> mtu 1500 index 11
        inet 192.168.56.201 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
```

This method of monitoring the interfaces mandates an capability of the networking card driver to use link-based IPMP. They have to set and unset the `RUNNING` flag based on the link state.[3]

– **Probe based:** The probe base mechanism itself is independent from the hardware. It checks the IP layer on the IP layer. The basic idea is: When you are able to communicate via IP to other systems, it's safe to assume that the IP layer is there.

The probing itself is a simple mechanism. The probe based failure detection sends ICMP messages to a number of systems. As long the other systems react on those ICMP packets, a link is considered as ok. When those other systems don't react in a certain time, the link is considered as failed and the failover takes place

I will talk about the advantages and disadvantages of both in a later section.

- **Data Address:** In IPMP terminology the data addresses are the addresses that are really used for communication. An IPMP group be used for multiple data addresses. However, all data addresses have to be in the same IP link.
- **Test Address:** When you send ICMP messages to detect a failure, you need a sourcing IP address for those messages. So each physical interface needs an address that is just used for testing purposes. This address is called test address.
- **Repair and Repair detection:** When you talk about failures, you have to talk about repairs as well. When an interface is functional again - for example by using another cable or a different switch - you have to detect this situation and reactivate the interface. Without repairs and the detection of repairs you would run out of interfaces pretty soon. The repair detection is just the other side of the failure dection, just that you check for probes getting through or a link that's getting up again.

[3] hme, eri, ce, ge, bge, qfe, dmfe, e1000g, ixgb, nge, nxge, rge, xge definitely work, ask the provider of the driver for other cards

- **Target systems:** A target system is the matching opposite part of the test address. When you want to check the availability of a network connection via sending probe messages via ICMP, you need a source as well as a target for this ICMP communication.

  In IPMP speak a target system is a system that is used to test the availability of an IP interface. The IPMP mechanism tries to ping the target system in order to evaluate if the network interface is still fully functional. This is done for each interface by choosing the test address as the source address of the IPMP request.

  Target systems are chosen by the IPMP mechanism. The mechanism to do so is quite simple:

  – Routers in an ip link are chosen as target systems automatically.

  – When there are no routers connected to the IP-link, the IPMP mechanism tries to find hosts in the neighborhood. A ping is sent to the "all hosts"-multicast address `224.0.0.1`.[4]

    ```
    jmoekamp@hivemind:~$ ping -s 224.0.0.1
    PING 224.0.0.1: 56 data bytes
    64 bytes from hivemind-prod (192.168.178.200): icmp_seq=0. time=0.052 ms
    64 bytes from 192.168.178.22: icmp_seq=0. time=0.284 ms
    64 bytes from 192.168.178.114: icmp_seq=0. time=20.198 ms
    ```

    The first few systems replying to this ping are chosen as target systems.

  – The automatic mechanism doesn't always choose the most optimal system for this check, thus you can specify them in the case you think a manual configuration ensures that the target system really represent a set of system, whose availability represents a check the availability of the network. Manually defined hosts have always precedence over routers, so manually defining such systems can reduce the ICMP load on your router. However, in most cases the automatic mechanism yields reasonable and sufficient results.

1.2.3 Link based vs. probe based failure/repair detection

As I wrote before, there are two methods of failure detection. Link based failure detection and probe based failure detection. Both have advantages and disadvantages:

[4] This address is specified by RFC 1112 `http://tools.ietf.org/html/rfc1112`

Link based

The link based method is the fasted method of both. Whenever the link goes down, the IPMP gets a notification of the state change of the interface almost immediately. So it can react instantaneously on such failures.

Furthermore it doesn't need any test addresses. It doesn't check the availability on the IP layer and so there is no need for the interface to communicate independently from the data address.

But there is a big disadvantage. The challenge lies in the point that it doesn't check the health of your IP connection, it just checks if there is a link. It's like a a small signal light, that indicates that there's power on the plug, but doesn't tell you if it's 220v or 110v.

There are situations when a purely link-based mechanism is misguiding, especially when the networks are getting more complex. Just think about the following network: Let's assume that link 1 fails. Obviously the link at the physical interface goes down. The link based mechanism can detect this failure and the system can react to this problem and switch over to the other networking card. But now let's assume that link 2 fails. The link on the connection 1 is still up and the system considers the connection to the network as functional. There is no change in the flags of the IP interface. However your networking connection is still broken as your defaultrouter is gone. A link means nothing when you can't communicate over it.

At first such scenarios doesn't sound so common and an intelligent network design can prevent such situations. Yes, that's correct, but just think about el-cheapo media converters from fibre to copper, that doesn't take down the link on the copper side when the link is down on the fibre side[5]. Or small switches that are misused as media converters[6]

Probe based

So how you can circumvent this problem? The solution is somewhat obvious. Don't check only the link on the physical layer. Check it on the layer that really matters. In the case of networking: Don't check if there's a physical link ... check if you can reach other systems with the IP protocol. And the probe base failure detection does exactly this.

[5] Albeit any decent media converter has a feature that mirrors the link down state from one side to the other to ease management and to notify the connected system about problems

[6] Dont' laugh about it ... I found dusty old 10BASET hubs in raised floors working perfectly as media converters for years and years

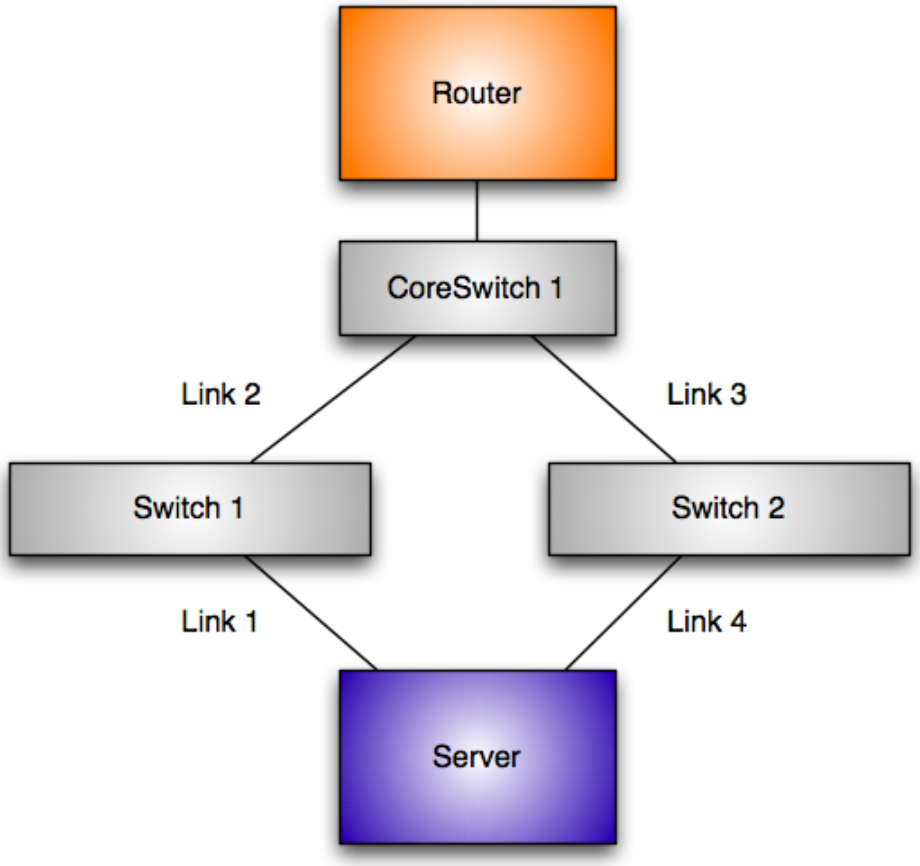

Figure 1.1: Simple network with redundant server connection

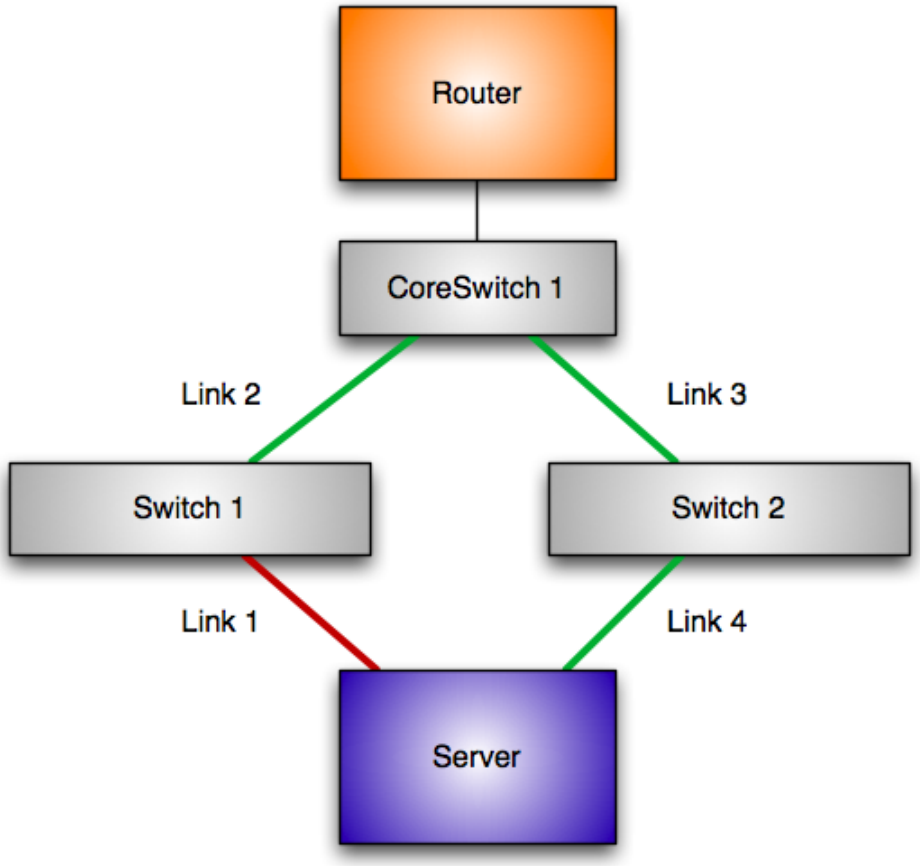

Figure 1.2: Simple network with redundant server connection

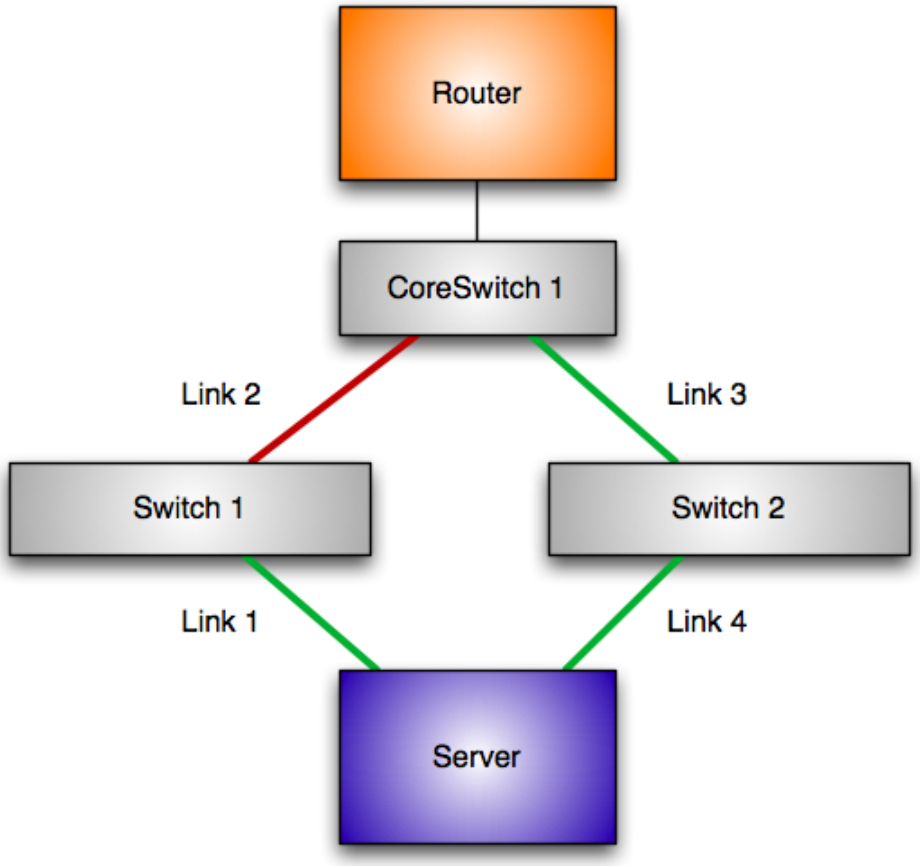

Figure 1.3: Simple network with redundant server connection

As i wrote before, the probe based failure detection uses ICMP messages to check a functional IP network. So it can check if you really have an IP connection to your default router and not just a link to a switch somewhere between the server and the router.

But this method has a disadvantage as well: You need vastly more IP-addresses. Every interface in the IPMP address needs a test address. The test address is used to test the connection and stays on the interface even in the case of a failure[7].

The IP address consumption is huge. Given you have n interfaces you need n test addresses.An IPMP group with four connections needs 4 test addresses. However you can ease the consumption of IP-Address by using a private network for the test addresses different to the network containing the data addresses. But I will get to this at the end of this chapter.

1.2.4 Failure/Repair detection time

Another interesting question is the speed of the failure and repair detection. It's different for both mechanisms.

For link based failure detection it's easy, as soon as the IPMP subsystem gets aware of the situation, that an interface lost the `RUNNING` flag, it's considered down. It's nearly instantaneous. Even probe-based IPMP uses this mechanism to speed up the failover. Link-based failure detection is still in action, even when you use probe-based failure detection.

But what's with the reaction time of the probe based failure detection? As i've told you before, the mechanism is based on ICMP messages. There are two simple rules:

- When 5 consecutive probes fail, the interface is considered as failed
- When 10 consecutive probes get through on an interface considered as failed, it's considered as repaired

In the default configuration, probing takes place roughly every 2 seconds. You can observe this by `snoop`ing the interface when you have put it into a IPMP group.

```
jmoekamp@hivemind:~# snoop -d e1000g0 -t a -r  icmp
Using device e1000g0 (promiscuous mode)
18:56:10.82015 192.168.178.201 -> 192.168.178.1 ICMP Echo request (ID: 11017 Sequence
     number: 27065)
18:56:10.82045 192.168.178.1 -> 192.168.178.201 ICMP Echo reply (ID: 11017 Sequence
    number: 27065)
18:56:12.64018 192.168.178.201 -> 192.168.178.1 ICMP Echo request (ID: 11017 Sequence
     number: 27066)
18:56:12.64053 192.168.178.1 -> 192.168.178.201 ICMP Echo reply (ID: 11017 Sequence
    number: 27066)
```

[7] Obviously you need the test mechanism to check if the physical link was repaired by the admin

Given the 2 seconds between the probes, a failure is detected in 10 seconds by default, a repair is detected in 20 seconds. However you can change this number in the case you need a faster failure. I will explain that on page 37 in section 1.9.4

1.2.5 IPMP vs. Link aggregation

Link aggregation is available on many switches for quite a while now. With link aggregation it is possible to bundle a number of interfaces into a single logical interface. There is a failure protection mechanism in Link Aggregation as well. At start it was somewhat similar to the link based failure detection. When the link is down on a member of an aggregation, the switch takes the link out of the aggregation and put it's back as soon as the link get's up again. Later something similar to the probe-based mechanism found it's way into the Ethernet standards. It's called LACP. With LACP special frames are used on a link to determine if the other side of the connection is in the same aggregate[8] and if there is really an Ethernet connection between both switches.I won't go in the details now, as this will be the topic of another tutorial in the next days.

But the main purpose of link aggregation is to create a bigger pipe when a single Ethernet connection isn't enough.

So ... why should you use IPMP? The reason is a simple one. When you use link aggregation, all your connections have to terminate on the same switch, thus this mechanisms won't really help you in the case of a switch failure.

The mechanisms of IPMP doesn't work in the Layer 2 of the network, it works in the third layer and so it doesn't have this constraint. The connections of an IPMP group can end in different switches, they can have different speeds, they could be even of a different technology, as long they use IP (this was more of advantage in the past, today in the "Ethernet Everything" age this point lost its appeal).

I tend to say that link aggregation is a performance technology with some high availability capabilities, where as IPMP is a high-availability technology with some performance capabilities.

1.3 Loadspreading

A source of frequent questions is the load spreading feature in IPMP. Customers have asked me if this comparable to the aggregation. My answer is "Yes, but not really!"

[8] It was common configuration error in early times to have non-matching aggregation configuration

Perhaps this is the right moment to explain a thing about IPMP. When you look at the interfaces of a classic IPMP configuration, it looks like the IP addresses are assigned to physical interfaces. But that isn't the truth. When you send out data on such an interface, it's spread on all active[9] interfaces of such a group.

But you have to be cautious: IPMP can do this only for outbound traffic. As IPMP is a server-only technology, there is no counterpart for it on the switch. So there is no load spreading on the switch.

The switches doesn't know about this situation. When an inbound packet reaches the default gateway, the router uses the usual mechanisms to get the ethernet address of the IP address and sends the data to this ethernet address. As there can be just one ethernet address for every IP address, the inbound communication will always use just one interface.

This isn't a problem for many workloads as many server applications send more data than they receive[10]. But as soon your application receives at lot of data[11], you should opt for another load distribution mechanism.

However there is a trick to circumvent this constraint: A single IPMP group can provide several data addresses. By carefully distributing this data addresses over the physical interfaces you are able to distribute the inbound load as well. So when you are able to use multiple IP addresses you could do such a manual spreading of the inbound load.

However real load spreading mechanisms with the help of the switches[12] will yield a much better distribution for the inbound traffic in many cases.

But this disadvantage comes with an advantage: You are not bound to a single switch to use this load spreading. You could terminate every interface of you server in a separate switch and the IPMP group still spreads the traffic on all interfaces. That isn't possible with the standard link aggregation technologies of Ethernet.

I want to end this section a short warning: Both aggregation technologies will not increase your bandwidth when you have just a single IP data stream. Both technologies will use the same Ethernet interface for a communication relation between client and server. It's possible to separate them even based on Layer 4 properties, but at the end the single

[9] Active doesn't mean functional. An interface can be functional but it isn't used by IP traffic. An interface can be declared as a standby interface, thus it may be functional but the IPMP subsystem wouldn't use it. That's useful when you have a 10 GBe Interface and a 1 GBe Interface. You don't want the 1 GBE interface for normal use, but it's better than nothing in the case the 10 GBe interface fails

[10] For example a webserver

[11] For example a webserver

[12] Like bundling of Ethernet links via LACP

ftp download will use just one of your lines. This is necessary to prevent out-of-order packets[13] due to different trip times of the data on separate links.

1.3.1 Classic IPMP vs. new IPMP

There are many similar concepts in Classic IPMP and New IPMP. But the both implementations have important differences as well.

The most important difference is the binding of the data address to the interfaces.

- With classic IPMP the data address is bound to a certain interface. In the case of the failure of an interface, the interface isn't used anymore for outbound traffic and the data address gets switched to an operational and active interface.
- With new IPMP you have a virtual `ipmp` interface in front of the physical network interfaces representing the IPMP group. The `ipmp` interface holds the data address and it isn't switched at any time. A physical interface may have a test address, but they are never configured with a data address. The virtual IPMP interface is your point of administration when you want to snoop network traffic for all interfaces in this group for example.

1.4 in.mpathd

There is a component in both variants that controls all the mechanisms surrounding IP multipathing. It's the `in.mpathd` daemon.

```
jmoekamp@hivemind:~$ ps -ef | grep "mpathd" | grep -v "grep"
    root  4523     1   0   Jan 19 ?           8:22 /lib/inet/in.mpathd
```

This daemon is automatically started by `ifconfig`, as soon you are configuring something in conjunction with IPMP on your system. The `in.mpathd` process is responsible for network adapter failure detection, repair detection, recovery, automatic failover and failback.

[13] You want to prevent this out of performance reasons

1.5 Prerequisites

At first you need a testbed. In this tutorial I will use a system with three interfaces. Two of them are Intel networking cards. They are named `e1000g0` and `e1000g0`. The third interface is an onboard Realtek LAN adapter called `rge0`.

The configuration of the ip network is straight forward. The subnet in this test is `192.168.178.0/24`. I have a router at `192.168.178.1`.

The physical network is a little bit more complex to demonstrate the limits of link-based failure detection. `e1000g0` and `e1000g1` are connected to a first switch called `Switch A`. This switch connects to to a second switch called `Switch B`. The `rge0` interface connects directly to `Switch B`. The router of this network is connected to `Switch B` as well.

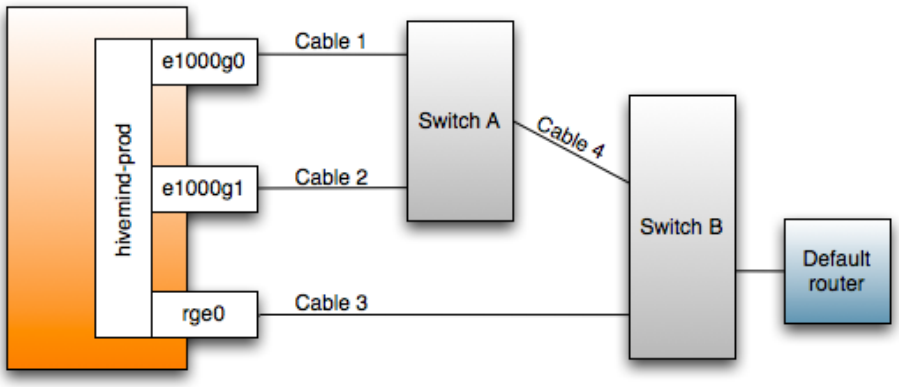

Figure 1.4: Configuration for the demo

To make the configuration a little bit more comfortable, we add a few hosts to our `/etc/hosts` file. We need four adresses while going through the tutorial. At first we need the name for the data address:

```
echo "192.168.178.200 hivemind-prod" >> /etc/hosts
```

Now we need names for our test addresses. It's a good practice to use the name of the data address appended with the name of the physical address:

```
echo "192.168.178.201 hivemind-prod-e1000g0" >> /etc/hosts
echo "192.168.178.202 hivemind-prod-e1000g1" >> /etc/hosts
echo "192.168.178.203 hivemind-prod-rge0" >> /etc/hosts
```

1.6 New IPMP

When you want to try new IPMP, you need a fairly recent build of OpenSolaris. New IPMP was integrated into Build 107 for the first time.

At first: If you have already a working IPMP configuration, you can simply reuse this config. However it yields a different looking, but functionally equivalent result compared to your system with Classic IPMP. This is made possible by some automagic functions in New IPMP. One example is the implicit creation of the IPMP interface with the name of the IPMP group when there isn't already an IPMP interface in the group. However explicit creation should be prefered as you can choose a better name for your IPMP interface and the dependencies are much more obvious.

As I wrote before, the new IPMP doesn't use a logical interface switching from one interface to the other. You have a special kind of interface for it. It's a virtual interface. It's looking like a real interface but there is no hardware behind this interface.

So ... at first we have to configure this interface:

```
jmoekamp@hivemind:/etc# ifconfig production0 ipmp hivemind-prod up
```

With this command you've configured the IPMP interface. You can use any name for it you want, it just has to begin with a letter and has to end on a number. I have chosen the name `production0` for this tutorial

Now let's look at the interface:

```
jmoekamp@hivemind:~# ifconfig production0
production0: flags=8011000803<UP,BROADCAST,MULTICAST,IPv4,FAILED,IPMP> mtu 68 index 6
        inet 192.168.178.200 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
```

As you see, it's pretty much looking like a normal network interface with some specialities: At first it's in the mode `FAILED` at the moment. There are no network interfaces configured to the group, thus you can't connect anywhere over this interface.

The interface is already configured with the data address.[14]. The data address will never move away from there. At the end you see the name of the IPMP group. The default behavior sets the name of the IPMP group and the name of the IPMP interface to the same value.

Okay, now we have to assign some physical interfaces to it. This is the moment where we have to make a decision. Do we want to use IPMP with probes or without probes? As I've explained before it's important to know at this point, what failure scenarios you want to cover with your configuration. You need to know it now, as the configuration is slightly different.

[14] Additional data addresses are configured as logical interfaces onto this virtual interface. You won't configure additional virtual IPMP interfaces

1.6.1 Link based failure detection

I want to explain the configuration of the link based failure detection first not only because it's easier, but to show you the problems of link based failure detection, too.

Configuration

As explained before, the link based failure detection just snoops on certain events of the networking card like a lost link. Thus we just have to configure the interface into the IPMP group that you want to protect against a link failure, but you don't have to configure any IP addresses on the member interfaces of the IPMP group.

Okay, at first we plumb the interfaces we want to use in our IPMP group:

```
jmoekamp@hivemind:/etc# ifconfig e1000g0 plumb
jmoekamp@hivemind:/etc# ifconfig e1000g1 plumb
jmoekamp@hivemind:/etc# ifconfig rge0 plumb
```

Okay, now we add the three member interfaces into the IPMP group:

```
jmoekamp@hivemind:/etc# ifconfig e1000g0 -failover group production0 up
jmoekamp@hivemind:/etc# ifconfig e1000g1 -failover group production0 up
jmoekamp@hivemind:/etc# ifconfig rge0 -failover group production0 up
```

As you may have noticed, we really didn't specify an IP address or a hostname. With link-based failure detection you don't need it. The IP address of the group is located on the IPMP interface we've defined a few moments ago.

But let's have a look at the `ifconfig` statements. There are two parameters you may not know:

- `-failover`: This parameter marks an interface as a non-failover one. In case of a failure, this interface configuration doesn't move. While a little bit strange in the context of a physical interface[15], but the rationale gets clearer with probe-based IPMP.
- `group production0`: the parameter `group` designates the IPMP group membership of an interface.

Let's look at one of the interfaces:

```
rge0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
    1500 index 5
        inet 0.0.0.0 netmask ff000000 broadcast 0.255.255.255
        groupname production0
```

[15] Moving hardware is a little bit problematic just by software....

We find the consequences of both `ifconfig` parameters: The `NOFAILOVER` is obviously the result of the `-failover` and `groupname production0` is the result of the `group production0` statement. But there is another flag that is important in the realm of IPMP. It's the `DEPRECATED` flag.

The `DEPRECATED` flag has a very simple meaning: Don't use this IP interface. When an interface has this flag, the IP address won't be used to send out data[16]. As those IP addresses are just for test purposes, you don't want them to appear in packets to the outside world.

Playing around

Now we need to interact with the hardware, as we will fail network connections manually. Or to say it differently: We will pull some cables.

But before we are doing this, we look at the initial status of our IPMP configuration. The new IPMP model improved the monitoring capabilities of it's state by introducing a command for this task. It's called `ipmpstat`.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
e1000g1     yes     production0 -------   up        disabled  ok
e1000g0     yes     production0 --mb---   up        disabled  ok
```

Just to give you a brief tour through the output of the command. The first column reports the name of the interface, the next one reports the state of the interface from the perspective of IPMP. The third column tells you which IPMP group was assigned to this interface.

The next columns gives us some more in-depth information about the interface. The fourth column is a multipurpose column to report a number of states. In the last output, the `--mb--` tells us, that the interface e1000g0 was chosen for sending and receiving multicast and broadcast data. Other interfaces doesn't have a special state, so there are just dashes in the respective `FLAGS` field of these interfaces. The fifth column reveals, that we've disabled probes[17]. The last column details on the state of the interface. In this example it is `OK` and so it's used in the IPMP group.

Okay, now pull the cable from the `e1000g0` interface. It's Cable 1 in the figure. The system automatically switches to e1000g1 as the active interface.

[16] Of course there are exceptions like an application specifically binding to the interface. Please look into the man page for further information

[17] Or to be more exact, that there is no probing as we didn't configured it so far

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
e1000g1     yes     production0 --mb---   up        disabled  ok
e1000g0     no      production0 -------   down      disabled  failed
```

As you can see, the failure has been detected on the `e1000g0` interface. The link is down, thus it is no longer active. Okay, let's repair it.

Put the cable back to the port of the `e1000g0` interface. After a moments, the link is up. The `in.mpathd` gets aware of the `RUNNING` flag on the interface. `in.mpathd` assumes that the network connection got repaired, so the state of the interface is set to `ok` and thus the interface is reactivated.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
e1000g1     yes     production0 --mb---   up        disabled  ok
e1000g0     yes     production0 -------   up        disabled  ok
```

The problem with link-based failure detection

Just in case you've played with the ethernet cables, ensure that IPMP chooses an interface connecting via Switch A as the active interface by zipping Cable 3 from the switch B for a moment. When you check with `ipmpstat -i` the `mb` has to be assigned to the interface `e1000g0` or `e1000g1`.

As i wrote before there are failure modes link-based failure detection can't detect. Now let's introduce such a fault. To do so, just remove Cable 4 between switch A and B.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
e1000g1     yes     production0 --mb---   up        disabled  ok
e1000g0     yes     production0 -------   up        disabled  ok
```

As there is still a link on the Cables 1 and 2 everything is fine from the perspective of IPMP. It doesn't switch to the connection via `rge0` which presents the only working connection to the outside world. IPMP is simply not aware of the fact that Switch A was seperated from the IP link `192.168.178.0/24` due to the removal of cable 4.

1.6.2 Probe based failure detection

The probe based detection has some additional capabilities. At first it has all the capabilities of the link-based detection. It switches over to the other network card as soon as the card loses the link. But additionally it checks the availability of the connection by pinging other IP addresses called target systems. When the system doesn't get a reply

on the ICMP messages, the interface is assumed to be in failure state and it isn't used anymore. `in.mpathd` switches the data addresses to other interfaces. So how do you configure probe based IPMP?

Configuration

Okay, at first we revert back to the original state of the system. This is easy, we just have to unplumb the interfaces. In my example I'm unplumbing all interfaces. You could reuse the `production0` interface, but I'm including it here just in case you've started reading this tutorial at the beginning of this paragraph[18]. It's important that you unplumb the member interfaces of the group before you unplumb the IPMP interface, otherwise you get an error message:

```
jmoekamp@hivemind:/etc# ifconfig e1000g0 unplumb
jmoekamp@hivemind:/etc# ifconfig e1000g1 unplumb
jmoekamp@hivemind:/etc# ifconfig rge0 unplumb
jmoekamp@hivemind:/etc# ifconfig production0 unplumb
```

Okay, now all the interfaces are away. Now we recreate the IPMP group.

```
jmoekamp@hivemind:/etc# ifconfig production0 ipmp hivemind-prod up
```

We can check the successful creation of the IPMP interface by using the `ipmpstat` command.

```
jmoekamp@hivemind:/etc# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 failed    --        --
```

At start there isn't an interface configured into the IPMP group. So let's start to fill the group with some life.

```
jmoekamp@hivemind:/etc# ifconfig e1000g0 plumb hivemind-prod-e1000g0 -failover group
    production0 up
```

There is an important difference. This `ifconfig` statement contains an IP address, that is assigned to the physical interface. This automatically configures IPMP to use the probe based failure detection.

The idea behind the `-failover` setting gets clearer now. Obviously the test addresses of an interface should be failovered by IPMP. They should stay on the logical interface. As the interface has the `FAILOVER` flag, the complete interface including it's IP address is exempted from any failover.

Let's check the ipmp group again:

[18] In this case, the first three commands will fail, but you have the explicitly defined IPMP interface

```
jmoekamp@hivemind:/etc# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        10.00s    e1000g0
```

There is now an interface in the group. Of course an IPMP group with just one interface doesn't really make sense. So configure we will configure a second interface into the group. You may have recognized the `FTD` column. `FTD` stands for "Failure Detection Time". Why is there an own column for this number? Due to the dynamic nature of the Failure Detection time, the `FDT` may be different for every group. With this column you can check the the current `FDT`.

```
jmoekamp@hivemind:/etc# ifconfig e1000g1 plumb hivemind-prod-e1000g1 -failover group
    production0 up
```

Let's check again.

```
jmoekamp@hivemind:/etc# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        10.00s    e1000g1 e1000g0
```

Now we add the third interface that is connected to the default gateway just via Switch B.

```
jmoekamp@hivemind:/etc# ifconfig rge0 plumb hivemind-prod-rge0 -failover group
    production0 up
```

Let's check again.

```
jmoekamp@hivemind:/etc# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        10.00s    rge0 e1000g1 e1000g0
```

All three interfaces are in the IPMP group now. And that's all ... we've just activated failure detection and failover by this four commands. Really simple, isn't it?

Playing around

I hope, you have still the hardware configuration in place, I used to show the problems of link based failure detection. In the case you haven't please create the configuration we've used there.

At first we do a simple test: We simply unplug a cable from the system. In my case I removed the cable 1:

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     no      production0 -------   down      failed    failed
```

The system reacts immediately, as the link-based failure detection is still active, even when you use the probe-based mechanism. You can observe this in the `ipmpstat` output by monitoring the state of the link column. It's `down` at the moment and obviously probes can't reach their targets. The state is assumed as `failed`. Now plug the cable back to the system:

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     no      production0 -------   up        failed    failed
```

The link is back, but the interface is still failed. IPMP works as designed here. The probing of the interface with ICMP messages still considers this interface as down. As we have now two mechanism to check the availability of the interface, both have to confirm the repair. IPMP doesn't consider an interface as repaired when just one ICMP probe gets through, it waits until 20 ICMP probes were correctly replied by the target system. Due to this probing at repair time instead of just relying on the link, you can prevent that an interface is considered as `OK` when an unconfigured switch brings the link back online, but the configuration of the switch doesn't allow to the server to connect anywhere (because of VLAN configuration for example).

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     yes     production0 -------   up        ok        ok
jmoekamp@hivemind:~#
```

As soon as the probing of the interface is successful, it brings the interface back to the `OK` state and everything is fine.

Now we get to a more interesting use case of probe-based failure detection. Let's assume we've repaired everything and all is fine. You should see a situation similar to this one in your `ipmpstat` output:

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     yes     production0 --mb---   up        ok        ok
```

Now unplug cable 4, the cable between the switch A and B. At first nothing happens, but a few seconds later IPMP switches the IP addresses to `rge0` and set the state of the other interfaces to `failed`.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     no      production0 -------   up        failed    failed
e1000g0     no      production0 -------   up        failed    failed
```

When you look at the output of `ipmpstat` you will notice that the link is still up, but the probe has failed, thus the interfaces were set into the state failed.

When you plug the cable 3 back to the switches nothing will happen at first. You have to wait until the probing mechanism reports that the IPMP messages were correctly returned by the target systems.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     no      production0 -------   up        failed    failed
e1000g0     no      production0 -------   up        failed    failed
```

After a few seconds it should deliver an `ipmpstat` output reporting everything is well again.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     yes     production0 -------   up        ok        ok
```

1.6.3 Making the configuration boot persistent

As you have recognized for sure, all this configuration took place with the `ifconfig` statement. This configuration is lost when you reboot the system. But there is already an entity that configures the interfaces at system start. It's using the `hostname.*` files. Thus we could use these files for IPMP as well.

Boot persistent link-based configuration

Okay, to recreate our link-based IPMP configuration in a boot persistent, we need to fill the `hostname.*` files with the following statements:

```
jmoekamp@hivemind:/etc# echo "ipmp group production0 hivemind-prod up" > /etc/
    hostname.production0
jmoekamp@hivemind:/etc# echo "group production0 -failover up" > /etc/hostname.e1000g0
jmoekamp@hivemind:/etc# echo "group production0 -failover up" > /etc/hostname.e1000g1
jmoekamp@hivemind:/etc# echo "group production0 -failover up" > /etc/hostname.rge0
```

We reboot the system now to ensure that we did everything correctly. When the system has booted up, we will check if we made an error.

```
jmoekamp@hivemind:~$ ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        --        rge0 e1000g1 e1000g0
jmoekamp@hivemind:~$ ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
e1000g1     yes     production0 -------   up        disabled  ok
e1000g0     yes     production0 --mb---   up        disabled  ok
```

Looks good. Now let's look into the list of interfaces.

```
jmoekamp@hivemind:~$ ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
production0: flags=8001000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500
    index 2
        inet 192.168.178.200 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
e1000g0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
     1500 index 3
        inet 0.0.0.0 netmask ff000000 broadcast 0.255.255.255
        groupname production0
e1000g1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
     1500 index 4
        inet 0.0.0.0 netmask ff000000 broadcast 0.255.255.255
        groupname production0
rge0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
    1500 index 5
        inet 0.0.0.0 netmask ff000000 broadcast 0.255.255.255
        groupname production0
lo0: flags=2002000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv6,VIRTUAL> mtu 8252 index 1
        inet6 ::1/128
jmoekamp@hivemind:~$
```

IPMP configured and boot-persistent? Check.

Boot persistent probe-based configuration

We can do the same for the probe-based IPMP:

```
jmoekamp@hivemind:/etc# echo "ipmp group production0 hivemind-prod up" > /etc/
    hostname.production0
jmoekamp@hivemind:/etc# echo "group production0 -failover hivemind-prod-e1000g0 up" >
     /etc/hostname.e1000g0
jmoekamp@hivemind:/etc# echo "group production0 -failover hivemind-prod-e1000g1 up" >
     /etc/hostname.e1000g1
jmoekamp@hivemind:/etc# echo "group production0 -failover hivemind-prod-rge0 up" > /
    etc/hostname.rge0
```

Reboot the system and login afterwards to check the list of interfaces.

```
jmoekamp@hivemind:~$ ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
production0: flags=8001000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,IPMP> mtu 1500
    index 2
        inet 192.168.178.200 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
e1000g0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
     1500 index 3
        inet 192.168.178.201 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
e1000g1: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
     1500 index 4
        inet 192.168.178.202 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
rge0: flags=9040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER> mtu
    1500 index 5
```

```
        inet 192.168.178.203 netmask ffffff00 broadcast 192.168.178.255
        groupname production0
lo0: flags=2002000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv6,VIRTUAL> mtu 8252 index 1
        inet6 ::1/128
```

Let's check the configuration via `ipmpstat`, too:

```
jmoekamp@hivemind:~$ ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        ok        ok
e1000g1     yes     production0 -------   up        ok        ok
e1000g0     yes     production0 -------   up        ok        ok
jmoekamp@hivemind:~$
```

Everything is find.

1.6.4 Using IPMP and Link Aggregation

As I wrote before there is another way to protect your system against the failure of a network connection - Link Aggregation. As i've explained before, there are failure modes that can't be addressed by link aggregation. But you can use both in conjunction. This makes sense, when your main connection is a 10GBe interface and you don't want to plug a second one into the system and use already existent 1GBe Interfaces as a backup for it instead.

It's pretty straightforward to do so. At first you have to configure the link aggregation.

```
jmoekamp@hivemind:~$ pfexec bash
jmoekamp@hivemind:~# ifconfig e1000g0 unplumb
jmoekamp@hivemind:~# ifconfig e1000g1 unplumb
jmoekamp@hivemind:~# dladm create-aggr -l e1000g0 -l e1000g1 aggregate0
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS             PORTSTATE
aggregate0  --              0Mb   unknown  unknown   0:1b:21:3d:91:f7    --
            e1000g0         0Mb   half     down      0:1b:21:3d:91:f7    standby
            e1000g1         0Mb   half     down      0:1b:21:16:8d:7f    standby
```

The `dladm create-aggr` creates an aggregation, that bundles the interfaces `e1000g0` and `e1000g1` into a single virtual interface. Now I plug both cables into the switch.

```
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS             PORTSTATE
aggregate0  --              100Mb full     up        0:1b:21:3d:91:f7    --
            e1000g0         100Mb full     up        0:1b:21:3d:91:f7    attached
            e1000g1         100Mb full     up        0:1b:21:16:8d:7f    attached
```

Interfaces are up, the aggregation is ready for use.

```
jmoekamp@hivemind:~# ifconfig rge0 unplumb
jmoekamp@hivemind:~# ifconfig production0 ipmp hivemind-prod up
jmoekamp@hivemind:~# ifconfig aggregate0 plumb
jmoekamp@hivemind:~# ifconfig aggregate0 -failover group production0 up
jmoekamp@hivemind:~# ifconfig rge0 plumb
jmoekamp@hivemind:~# ifconfig rge0 -failover group production0 up
```

Looks pretty much like a standard IPMP configuration. You can think of `aggregate0` as a plain-standard physical interface from the perspective the the admin. When we check the IPMP configuration we will see both interfaces.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
aggregate0  yes     production0 --mb---   up        disabled  ok
jmoekamp@hivemind:~# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        --        rge0 aggregate0
```

Now we unplug one of the aggregated cables.

```
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS            PORTSTATE
aggregate0  --              100Mb full     up        0:1b:21:3d:91:f7   --
            e1000g0         100Mb full     up        0:1b:21:3d:91:f7   attached
            e1000g1         0Mb   half     down      0:1b:21:16:8d:7f   standby
jmoekamp@hivemind:~# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 ok        --        rge0 aggregate0
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 -------   up        disabled  ok
aggregate0  yes     production0 --mb---   up        disabled  ok
```

Everything is still okay. The aggregate hides the fact of the one failed interface from the IPMP subsystem. Now we unplug the second interface.

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        disabled  ok
aggregate0  no      production0 -------   down      disabled  failed
jmoekamp@hivemind:~# ipmpstat -g
GROUP       GROUPNAME   STATE     FDT       INTERFACES
production0 production0 degraded  --        rge0 [aggregate0]
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        disabled  ok
aggregate0  no      production0 -------   down      disabled  failed
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS            PORTSTATE
aggregate0  --              0Mb   unknown  down      0:1b:21:3d:91:f7   --
            e1000g0         0Mb   half     down      0:1b:21:3d:91:f7   standby
            e1000g1         0Mb   half     down      0:1b:21:16:8d:7f   standby
```

The links are both down, and without a functional interface left, the "link" of the aggregate goes down as well[19]. Of course the IPMP subsystem switches to `rge0` now. When we plug one cable back to the switch, the aggregate is functional again and IPMP detects this and the interface is considered as functional in IPMP again, too.

```
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS            PORTSTATE
aggregate0  --              100Mb full     up        0:1b:21:3d:91:f7   --
            e1000g0         100Mb full     up        0:1b:21:3d:91:f7   attached
            e1000g1         0Mb   half     down      0:1b:21:16:8d:7f   standby
```

[19]It stays up, as long as there's a functional interface in the aggregate

```
jmoekamp@hivemind:~# ipmpstat -i
INTERFACE   ACTIVE  GROUP       FLAGS     LINK      PROBE     STATE
rge0        yes     production0 --mb---   up        disabled  ok
aggregate0  yes     production0 -------   up        disabled  ok
```

When you plug the second interface into the interface, the aggregate is complete. But it doesn't change a thing from the IPMP side, as the `aggregate0` interface was already functional from the perpective of IPMP with just one interface.

```
jmoekamp@hivemind:~# dladm show-aggr -x aggregate0
LINK        PORT            SPEED DUPLEX   STATE     ADDRESS             PORTSTATE
aggregate0  --              100Mb full     up        0:1b:21:3d:91:f7    --
            e1000g0         100Mb full     up        0:1b:21:3d:91:f7    attached
            e1000g1         100Mb full     up        0:1b:21:16:8d:7f    attached
jmoekamp@hivemind:~#
```

1.6.5 Monitoring the actions of IPMP in your logfiles

All actions of the IPMP subsystem are logged by syslog. In this section I will show you the log messages that you get when a failure occurs and a repair takes place. The `mpathd` is somewhat chatty about the stuff it does.

Failure and repair of a single interface

```
Jan  8 20:01:06 hivemind in.mpathd[15113]: [ID 215189 daemon.error] The link has gone
     down on rge0
Jan  8 20:01:06 hivemind in.mpathd[15113]: [ID 968981 daemon.error] IP interface
    failure detected on rge0 of group production0
Jan  8 20:01:28 hivemind in.mpathd[15113]: [ID 820239 daemon.error] The link has come
     up on rge0
Jan  8 20:01:43 hivemind in.mpathd[15113]: [ID 341557 daemon.error] IP interface
    repair detected on rge0 of group production0
```

Failure of all interfaces and repair of a single interface

```
Jan  8 20:00:35 hivemind in.mpathd[15113]: [ID 773107 daemon.error] All IP interfaces
     in group production0 are now unusable
Jan  8 20:00:51 hivemind in.mpathd[15113]: [ID 561795 daemon.error] At least 1 IP
    interface (rge0) in group production0 is now usable
```

1.7 Classic IPMP

IPMP itself is a really old feature. It's in Solaris for several versions now. Just the implementation I've described before is a new one. But in Solaris 10 you don't have this new IPMP implementation. Solaris 10 still uses the old implementation. I will call the old implementation classic IPMP. The basic mechanism of new and classic IPMP is pretty much the same: Providing failure detection mechanisms and switch something

to do a failover thus the data address stays available. But internally it's a completely different implementation. While the new mechanism is certainly the future of IPMP, I' m pretty sure you will old mechanism more often in the wild.

1.7.1 Prerequisites

This example works with the same configuration, but you need a system with Solaris 10 or an Opensolaris System with a build earlier than 107. I just use Opensolaris on my lab machines, thus i used an virtualized Solaris 10 to explain the configuration of classic IPMP.

I will use the following addresses:

```
192.168.178.200 vhivemind-prod
192.168.178.201 vhivemind-e1000g0
192.168.178.202 vhivemind-e1000g1
```

I will demonstrate this on a recent release of Solaris 10:

```
bash-3.00# cat /etc/release
                        Solaris 10 5/09 s10x_u7wos_08 X86
           Copyright 2009 Sun Microsystems, Inc.  All Rights Reserved.
                        Use is subject to license terms.
                             Assembled 30 March 2009
```

1.7.2 Link based classic IPMP

An important difference to the new IPMP implementation is the point that you doesn't create a distinct IPMP interface because the concept of such a thing doesn't exist in classic IPMP. With link based classic IPMP you just put the interfaces in a group

```
bash-3.00# ifconfig e1000g0 plumb
bash-3.00# ifconfig e1000g1 plumb
bash-3.00# ifconfig e1000g0 vhivemind-prod netmask + broadcast + group production0 up
bash-3.00# ifconfig e1000g1 group production0 up
bash-3.00# ifconfig -a
```

Let's have a short look onto the network configuration.

```
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 15
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
e1000g1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 16
        inet 0.0.0.0 netmask ff000000
        groupname production0
        ether 8:0:27:6d:9:be
```

The data address is directly bound to one of the interfaces. It's important to know,that even when the `ifconfig` output suggest something different, outbound data flows to the network on both interfaces, not just the one which holds the data address.

Now unplug the cable connecting to `e1000g0`

```
bash-3.00# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=219000802<BROADCAST,MULTICAST,IPv4,NOFAILOVER,FAILED,CoS> mtu 0 index
    15
        inet 0.0.0.0 netmask 0
        groupname production0
        ether 8:0:27:11:34:43
e1000g1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 16
        inet 0.0.0.0 netmask ff000000
        groupname production0
        ether 8:0:27:6d:9:be
e1000g1:1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 16
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
```

The data address was moved away from `e1000g1` and a logical interface was created to hold it instead.

1.7.3 Probe based classic IPMP

The failure detection by IPMP probes is available in classic IPMP as well. Again all the configuration is done via `ifconfig`

```
bash-3.00# ifconfig e1000g0 plumb
bash-3.00# ifconfig e1000g1 plumb
bash-3.00# ifconfig e1000g0 vhivemind-e1000g0 deprecated -failover netmask +
    broadcast + group production0 up
bash-3.00# ifconfig e1000g0 addif vhivemind-prod netmask +  broadcast + up
Created new logical interface e1000g0:1
bash-3.00# ifconfig e1000g1 vhivemind-e1000g1 deprecated -failover netmask +
    broadcast + group production0 up
bash-3.00# ifconfig -a
```

Please note that you have to use the `deprecated` option to set the `DEPRECATED` flag on your own. New IPMP do this automagically. Forgetting this option leads to interesting, but not always obvious malfunctions. Let's check the network configuration.

```
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    CoS> mtu 1500 index 11
        inet 192.168.56.201 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
e1000g0:1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 11
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
e1000g1: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    CoS> mtu 1500 index 12
        inet 192.168.56.202 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
```

```
        ether 8:0:27:6d:9:be
```

Both interfaces have their test addresses. The data address is configured to an additional logical interface. As it's the only interface without the `-failover` statement, this interface is automatically managed by IPMP. Now remove the cable from the `e1000g0` networking card.

```
bash-3.00# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=219040803<UP,BROADCAST,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,FAILED,CoS
    > mtu 1500 index 11
        inet 192.168.56.201 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
e1000g1: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    CoS> mtu 1500 index 12
        inet 192.168.56.202 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:6d:9:be
e1000g1:1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 12
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
```

The virtual interface with the data address has moved from `e1000g0` to `e1000g1`

1.7.4 Making the configuration boot persistent

Making the configuration boot-persistent works works pretty much the same in both implementation. As we used `ifconfig` commands again, we can use the `hostname.*` files. We just have to translate the command lines accordingly:

Link-based IPMP

At first we configure the `e1000g0` interface by creating the file `/etc/hostname.e1000g0` containing a single line.

```
vhivemind-prod netmask + broadcast + group production0 up
```

Afterwards we do the same for `e1000g1`. We create a file named `/etc/hostname.e1000g1` and put the following line (and just this line) in it:

```
group production0 up
```

Now reboot the system. After a few moments you can get a shell and check your configuration.

```
# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 2
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
e1000g1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 3
        inet 0.0.0.0 netmask ff000000 broadcast 0.255.255.255
        groupname production0
        ether 8:0:27:6d:9:be
```

Everything configured as we've planned it.

Probe-based IPMP

Okay, let's do the same for the probe-based IPMP. This is the `/etc/hostname.e1000g0` file configuring the test address on the physical interface and the data address:

```
vhivemind-e1000g0 deprecated -failover netmask + broadcast + group production0 up \
addif vhivemind-prod netmask + broadcast + up
```

The file `/etc/hostname.e1000g1` with the following line will configures the `e1000g1` interface of our system at boot:

```
vhivemind-e1000g1 deprecated -failover netmask + broadcast + group production0  up
```

Okay, reboot your system and you should yield an `ifconfig` output like this one afterwards.

```
# ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
e1000g0: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    CoS> mtu 1500 index 2
        inet 192.168.56.201 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:11:34:43
e1000g0:1: flags=201000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,CoS> mtu 1500 index 2
        inet 192.168.56.200 netmask ffffff00 broadcast 192.168.56.255
e1000g1: flags=209040843<UP,BROADCAST,RUNNING,MULTICAST,DEPRECATED,IPv4,NOFAILOVER,
    CoS> mtu 1500 index 3
        inet 192.168.56.202 netmask ffffff00 broadcast 192.168.56.255
        groupname production0
        ether 8:0:27:6d:9:be
```

Everything is fine.

1.8 Classic and new IPMP compared

We've configured both mechanisms now. Let's summarize what we have seen so far. When you use the new IPMP the data address is bound to its own interface. Whatever happens to the physical interface there are no changes to the binding of interfaces to ip addresses.

Classic IPMP is different. In the figure I've highlighted the data address by using a bold font. When an interface fails it moves the data address is moved to the functional interface.

There are other differences as well:

- New IPMP provides better observability by the `ipmpstat` tool
- You can assign test addresses with DHCP. This is especially useful when you using a distinct network for your test addresses. As the test addresses are just used for the failure probing you can use ephemeral addresses for them and don't have to manually track them.
- As new IPMP uses an distinct interface, it solves a lot of deficiencies of classic IPMP. To get an overview of this shortcomings you should look at the development portal of the new IPMP mentioned in the sources of this document stated at the end of this chapter.

1.9 Tips, Tricks and other comments

1.9.1 Reducing the address sprawl of probe based failure detection

The probe based failure detection isn't without a disadvantage. You need a test IP adresses for every interface in an IPMP group. But: The IP addresses doesn't have to be routable ones, they doesn't even have to be in the same subnet as the data address. It's perfectly possible to use a routable IP address as the data address and to use for example the 192.168.1.0/24 address just for the IPMP test addresses . You have just to take care of one additional configuration item: You have to provide test targets in the same network. This can be done by configuring an an additional IP address on your router interfaces for example.

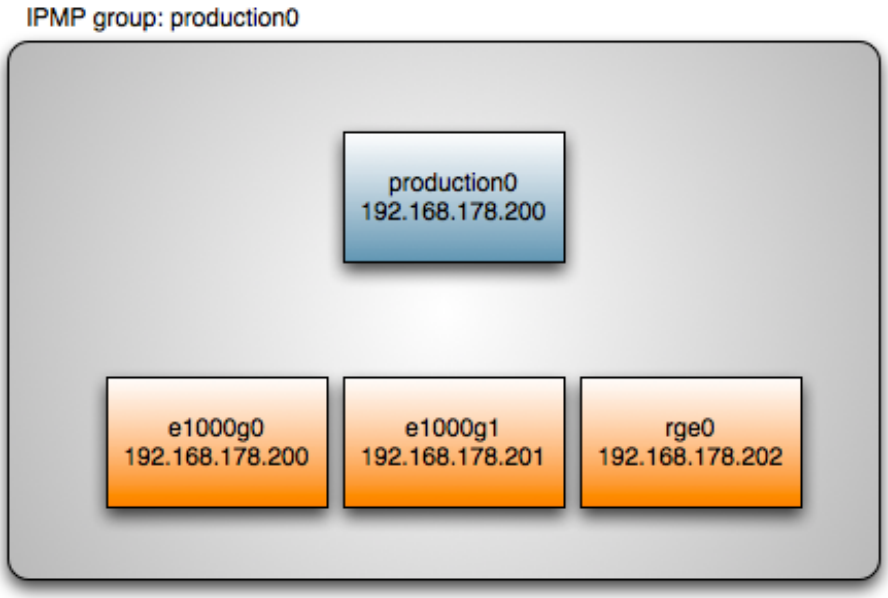

Figure 1.5: New IPMP - everything okay

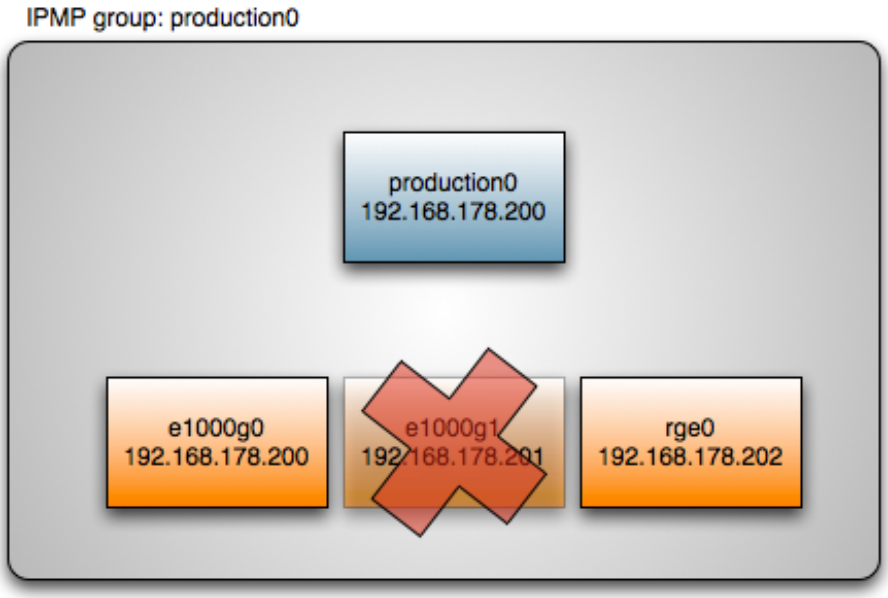

Figure 1.6: New IPMP - e1000g1 failed

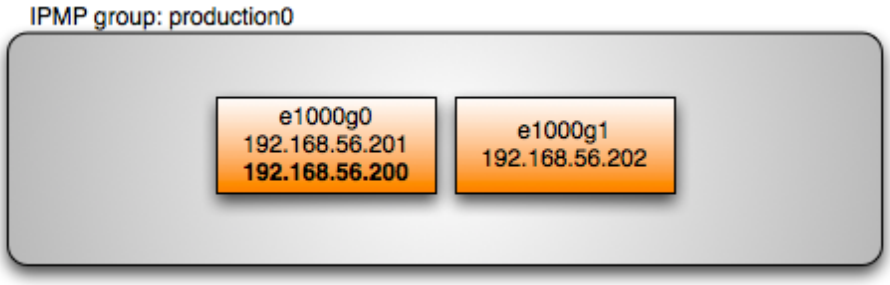

Figure 1.7: Classic IPMP - everything okay

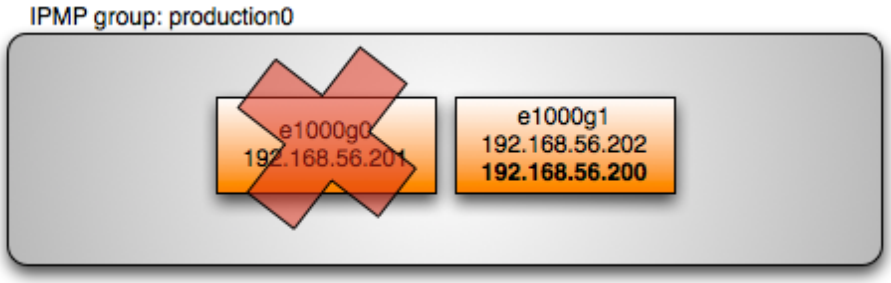

Figure 1.8: Classic IPMP - e1000g0 failed

1.9.2 Explicitly configuring target systems

Configuring explicit target systems is easy. You just have to configure host routes to them. Thus `route add -host 192.168.178.2 192.168.178.2 -static` would force `mpathd` to use this system as a target system.

A transient service to configure those target systems

Those routes aren't boot persistent. So you need a way to ensure that those routes are configured at boot up. You could use a simple legacy ¡code¿init.d¡/code¿ script, but I want to implement it as a configurable transient SMF service. A transient service is a service that is executed once at the start and isn't monitored by the restarter daemon. A transient service is a useful mechanism to do such initial configuration tasks at system start.

The service consists out of a simple manifest. Store it in a file `ipmptargets.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">
<service_bundle type='manifest' name='ipmptargets'>
<service
        name='network/ipmptargets'
        type='service'
        version='1'>

       <dependency
         name='network'
         grouping='require_all'
         restart_on='none'
         type='service'>
                <service_fmri value='svc:/milestone/network:default' />
       </dependency>

       <exec_method
         type='method'
         name='start'
         exec='/lib/svc/method/ipmptargets %m'
         timeout_seconds='60' />

        <exec_method
          type='method'
          name='refresh'
          exec='/lib/svc/method/ipmptargets %m'
          timeout_seconds='60' />

        <exec_method
          type='method'
          name='stop'
          exec='/lib/svc/method/ipmptargets %m'
          timeout_seconds='60' />

        <property_group name='startd' type='framework'>
                <propval
                  name='duration' type='astring' value='transient' />
         </property_group>
```

```
        <property_group name='general' type='framework'>
                <propval
                  name='action_authorization'
                  type='astring'
                  value='solaris.smf.manage.ipmptargets' />
                <propval name='value_authorization'
                  type='astring'
                  value='solaris.smf.manage.ipmptargets' />
        </property_group>

        <instance name='target1' enabled='false'>
                <property_group
                  name='config_params' type='application'>
                        <propval
                        name='ip' type='astring' value='192.168.178.1'/>
                </property_group>
        </instance>

        <instance name='target2' enabled='false'>
                <property_group
                  name='config_params' type='application'>
                        <propval
                          name='ip' type='astring' value='192.168.178.20'/>
                </property_group>
        </instance>

        <stability value='Unstable' />
        <template>
                <common_name>
                        <loctext xml:lang='C'>
                                system-wide configuration of IP routes for IPMP
                        </loctext>
                </common_name>
                <documentation>
                        <manpage
                                title='ifconfig'
                                section='1M'
                                manpath='/usr/share/man' />
                </documentation>
        </template>
</service>
</service_bundle>
```

The specific host routes are implemented as instances of this service. So it is possible to control the routes with a fine granularity.

Okay, obviously we need the script mentioned in the exec methods of the manifest. So put the following script into the the file `/lib/svc/method/ipmptargets`:

```
#!/bin/sh

. /lib/svc/share/smf_include.sh

getproparg() {
val=`svcprop -p $1 $SMF_FMRI`
[ -n "$val" ] && echo $val
}

if [ -z "$SMF_FMRI" ]; then
echo "SMF framework variables are not initialized."
exit $SMF_EXIT_ERR_CONFIG
```

```
fi

OPENVPNBIN='/usr/sbin/route'
IP=`getproparg config_params/ip`


if [ -z "$IP" ]; then
echo "config_params/ip property not set"
exit $SMF_EXIT_ERR_CONFIG
fi

case "$1" in
'start')
route add -host $IP $IP -static
;;

'stop')
echo "not implemented"
route delete -host $IP $IP -static
;;

'refresh')
 route delete -host $IP $IP -static
 route add -host $IP $IP -static
;;

*)
echo $"Usage: $0 {start|refresh}"
exit 1
;;

esac
exit $SMF_EXIT_OK
```

Okay. Now we have to import the the SMF manifest into the repository.

```
jmoekamp@hivemind:~# svccfg import  ipmp_hostroutes.xml
```

It's ready to use now. You can enable and disable your IPMP host routes as you need them:

```
jmoekamp@hivemind:~# svcadm enable ipmptargets:target1
jmoekamp@hivemind:~# netstat -nr

Routing Table: IPv4
  Destination           Gateway           Flags  Ref     Use     Interface
-------------------- -------------------- ----- ----- ---------- ---------
default              192.168.178.1        UG       23     496909
127.0.0.1            127.0.0.1            UH        2       2796 lo0
192.168.56.0         192.168.56.1         U         2          0 vboxnet0
192.168.178.0        192.168.178.9        U         3          0 production0
192.168.178.1        192.168.178.1        UGH       1          0

Routing Table: IPv6
  Destination/Mask            Gateway                   Flags Ref   Use    If
--------------------------- --------------------------- ----- --- ------- -----
::1                         ::1                         UH      2      20 lo0
jmoekamp@hivemind:~# svcadm disable ipmptargets:target1
jmoekamp@hivemind:~# netstat -nr

Routing Table: IPv4
  Destination           Gateway           Flags  Ref     Use     Interface
```

```
-------------------- -------------------- ----- ----- ---------- ---------
default              192.168.178.1        UG       24     496941
127.0.0.1            127.0.0.1            UH        2       2796 lo0
192.168.56.0         192.168.56.1         U         2          0 vboxnet0
192.168.178.0        192.168.178.9        U         3          0 production0

Routing Table: IPv6
  Destination/Mask            Gateway                   Flags Ref   Use    If
--------------------------- --------------------------- ----- --- ------- -----
::1                         ::1                         UH      2      20 lo0
```

1.9.3 Migration of the classic IPMP configuration

Albeit new IPMP should be configured like described above a correct configuration for the classic IPMP will setup the new IPMP correctly as well.

1.9.4 Setting a shorter or longer Failure detection time

When i talked about probe based failure detection, i've told you that you have to wait a certain time until a failure or a repair is detected by the `in.mpathd`. The default is 10 seconds for failure detection. The time for repair detection is always twice the time of the failure detection, so it's 20 seconds per default. But sometimes you want a faster reaction to failures and repairs. You can control the failure detection time by editing `/etc/default/mpathd`. You will find a configuration item controlling this time span in the file

```
#
# Time taken by mpathd to detect a NIC failure in ms. The minimum time
# that can be specified is 100 ms.
#
FAILURE_DETECTION_TIME=10000
```

By using a smaller number, you can speed up the failure detection but you have a much higher load of ICMP probes on you system. Keep in mind that i've told you that if 5 consecutive probes fail, the interfaces is considered as failed. When the failure detection time is 10000 ms, the probes have to be sent every 2000 ms. When you configure 100 ms, you will see a probe every 20ms. Furthermore this probing is done on every interface. Thus at 100ms failure detection time, the targets will see 3 ping requests every 20 milliseconds.

So keep in mind that lowering this number will increase load on other systems. So choose your the failure detection based on your business and application needs, not on the thought "I want the lowest possible time".

Just to demonstrate this effect and to learn how you set the failure detection time, you should modify the value in the line `FAILURE_DETECTION_TIME` to 100. Restart the

`in.mpathd` afterwards by sending a HUP signal to it with verb=.pkill -HUP in.mpathd=. When you start a snoop with `snoop -d e1000g0 -t a -r  icmp` you will have an output on your display scrolling at a very high speed.

1.10 Conclusion

The nice thing about IPMP is: It's simply there. You can use it. When you have more than one interface in your system and you care about the availability of your network, it takes you just a few seconds to activate at least the link-based variant of IPMP. This fruit is really hanging just a few centimeters above the ground.

1.11 Do you want to learn more?

Documentation

docs.sun.com - Solaris 10 - System Administration Guide: IP Services - Part VI: IPMP[20]

man pages

docs.sun.com - ifconfig(1M)[21]

docs.sun.com - in.mpathd(1M)[22]

docs.sun.com - ipmpstat(1M)[23]

Misc.

Project Clearview: IPMP Rearchitecture[24]

[20] `http://docs.sun.com/app/docs/doc/816-4554/ipmptm-1?l=en&a=view`
[21] `http://docs.sun.com/app/docs/doc/816-5166/ifconfig-1m?a=view`
[22] `http://docs.sun.com/app/docs/doc/816-5166/in.mpathd-1m?l=en&a=view`
[23] `http://docs.sun.com/app/docs/doc/819-2240/ipmpstat-1m?a=view`
[24] `http://hub.opensolaris.org/bin/view/Project+clearview/ipmp`