1 BFD

From the following still available site:
https://yurmagccie.wordpress.com/2015/08/05/bidirectional-forwarding-detection/

Most of the content has been left unchanged, some stuff (yellow background) has been added to better clear some concepts, add some details, better specify something. This is useful most of all to understand bfd timers. Chapter 3 has been completely written by me.

BFD [**RFC5880 – Bidirectional Forwarding Detection**] is designed to detect failures in communication with a forwarding plane next hop. It is intended to be implemented in some component of the **forwarding engine** of a system, in cases where the forwarding and control engines are separated. The BFD state machine implements a **three-way handshake**, both when establishing a BFD session and when tearing it down for any reason, to ensure that both systems are aware of the state change.

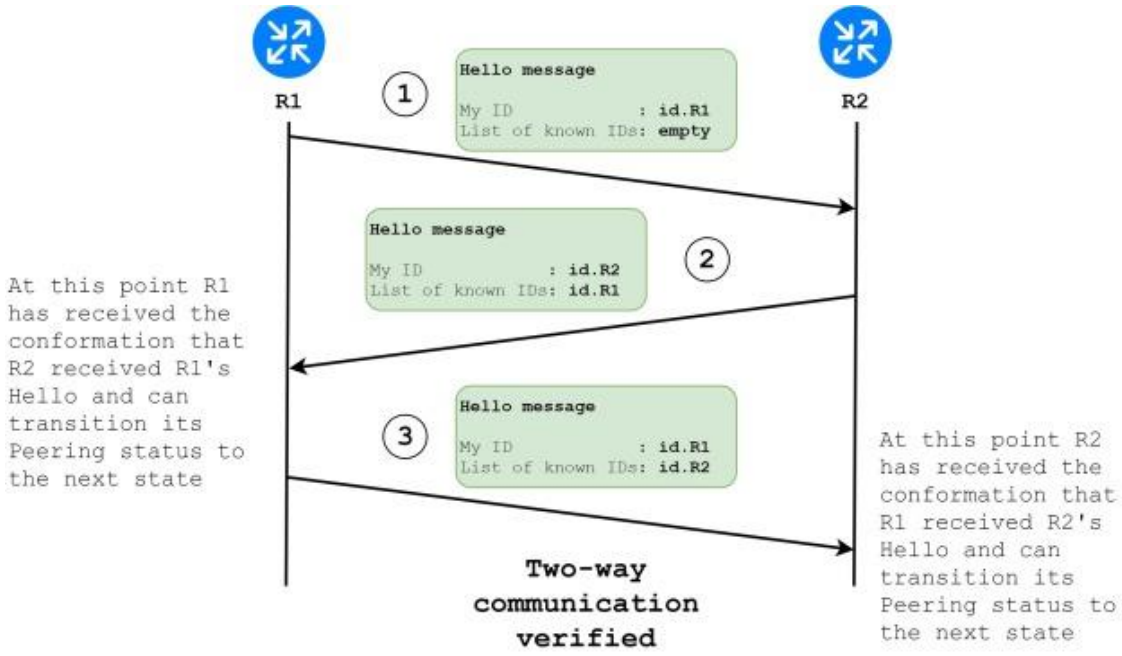
BFD is a simple Hello protocol. A pair of systems transmit BFD packets periodically over each path between the two systems, and if a system stops receiving BFD packets for long enough, some component in that particular bidirectional path to the neighboring system is assumed to have failed. Under some conditions, systems may negotiate not to send periodic BFD packets (**also called 'echo packets'**) in order to reduce overhead.
Each system estimates how quickly it can send and receive BFD packets in order to come to an agreement with its neighbor about how rapidly detection of failure will take place. These estimates can be modified in real time.

Why BFD?
Existing Keepalive mechanisms are not efficient:

– default timers are way too slow (OSPF Dead timer is 40 seconds, EIGRP Hold timer is 15 seconds)
– most protocol keepalives are CPU intensive
– every protocol use its own hello mechanism – more protocols – more CPU load

It is **lightweight**:
– – 24 bytes (BFD Control payload)
– – 12 bytes (BFD Echo payload)

A single BFD session can be used for many protocols that coexist in a single link (OSPF, PIM, HSRP).
BFD uses **UDP** port 3784 for **Control** packets.
BFD uses **UDP** port 3785 for **Echo** packets.
BFD three-way handshake
BFD uses "**I see you**" concept in its Hello messages, like OSPF and IS-IS.



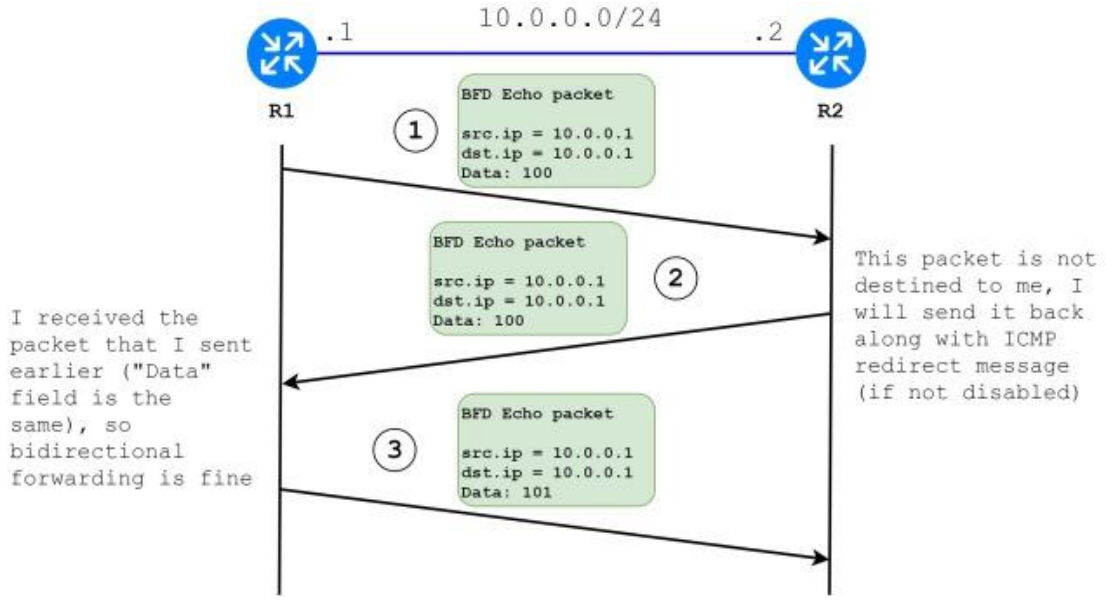
Another approach is to use sequence numbers and acknowledgements, like EIGRP and TCP.

BFD session

BFD has two operating modes that may be selected (Asynchronous and Demand), as well as an additional function that can be used in combination with the two modes. Cisco supports only asynchronous mode. In Asynchronous mode mode, the systems periodically send BFD Control packets to one another, and if a number of those packets in a row are not received by the other system, the session is declared to be down.

Echo function

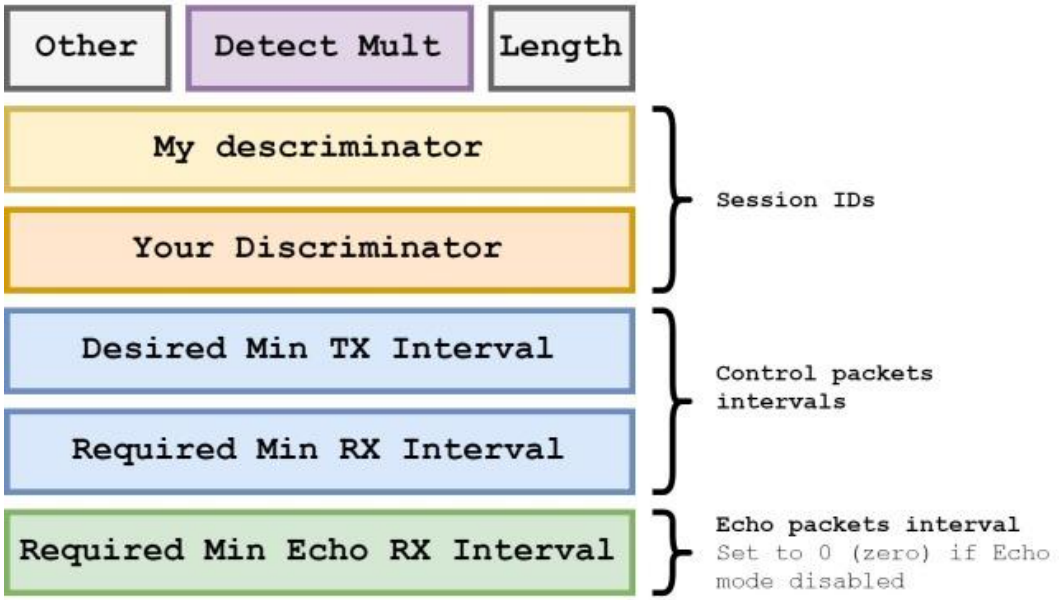
When the Echo function is active, a stream of BFD Echo packets is transmitted in such a way as to have the other system loop them back through its forwarding path. If a number of packets of the echoed data stream are not received, the session is declared to be down. Since the Echo function is handling the task of detection, the rate of periodic transmission of Control packets may be reduced.



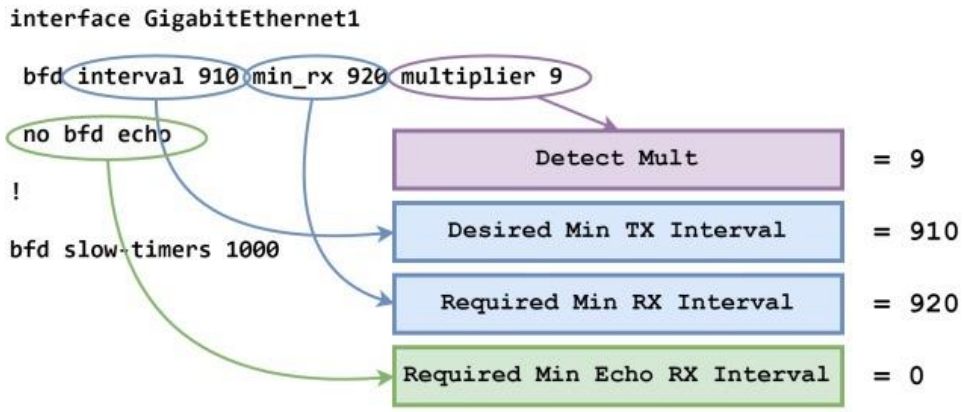
Note: beware that as depicted in the above picture, echo packets are sent with the source address equal to the destination address ... in normal conditions, this would trigger the receiving router R2 to send an 'ip redirect' packet on Ethernet interfaces, because the destination ip is on the same LAN but has been sent to another ip on the same LAN. On certain Cisco devices and software releases the command 'no ip redirect' MUST be manually configured, and is not disabled in case bfd is configured. This can lead to high cpu usage on the devices, when bfd is activated, and until redirects are de-activated again.

Asynchronous mode without Echo function

In Asynchronous mode with Echo function disabled routers exchange only BFD Control packets.
BFD control packet format:



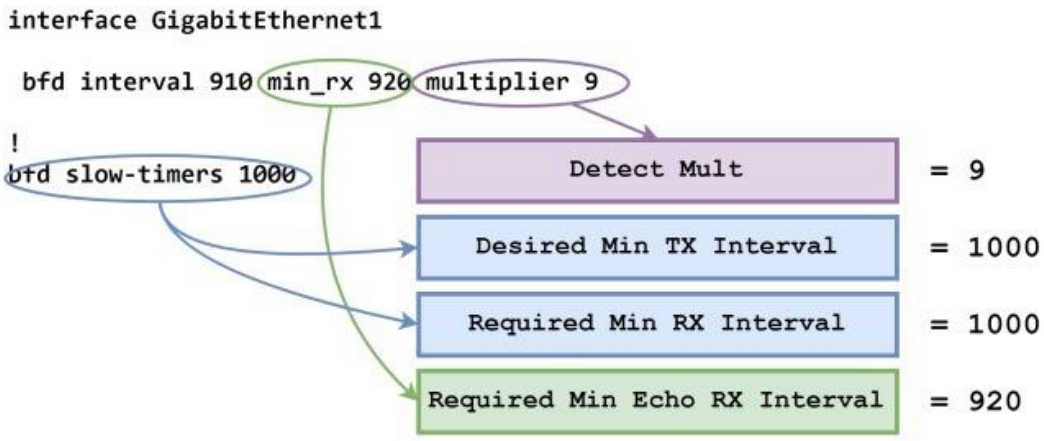
The discriminator fields are used to distinguish the different bfd sessions, but was also used on Cisco proprietary implementation on Bundles or Port-Channels. This was a great problem because bfd couldn't be used between different vendors on PoCh interfaces, until **rfc 7130** that should have solved the problem.
Here is how BFD configuration on Cisco devices without Echo function is reflected in BFD Control packet:



So echo packets avoidance is signaled by the required min echo rx interval set to 0.

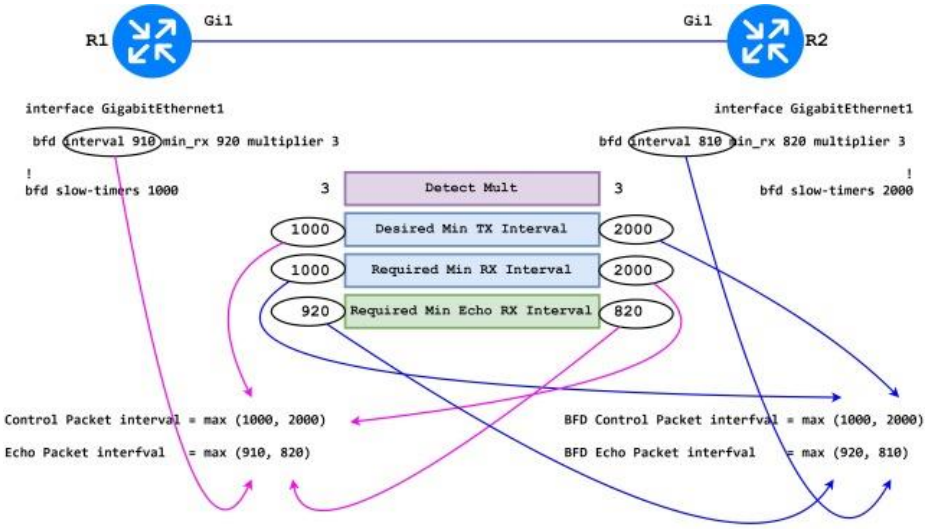
Asynchronous mode with Echo function

In Asynchronous mode with **Echo function enabled** routers exchange both BFD Control packets and BFD Echo packets. Here is how BFD configuration with Echo function is reflected in BFD Control packet:



During the exchange of the first two packets when the session is established, the timers are agreed selecting the less aggressive ones to avoid overloading the weaker device.

Here is a router makes a decision for BFD intervals based on Control packet exchange:



So we need to decide the rate and timers to use for control packets:

- the rx/tx interval for control packets is set with the 'bfd slow-timers' command, thus the used value will be the maximum of the two (the default value on Cisco devices should be 2 seconds)
- regarding echo packet timers, again you have a minimum rx interval and a desired tx interval specified in the command '`bfd interval <tx_int> min_rx <rx_int> multiplier <mul>`', again the maximum value for the two parties is chosen

R1:

```
CSR1#sh bfd neighbors details

IPv4 Sessions
```

```
NeighAddr                              LD/RD         RH/RS     State     Int
10.0.0.2                              4097/4097      Up        Up        Gi1
Session state is UP and using echo function with 910 ms interval.
Session Host: Software
OurAddr: 10.0.0.1
Handle: 1
Local Diag: 0, Demand mode: 0, Poll bit: 0
MinTxInt: 1000000, MinRxInt: 1000000, Multiplier: 3
Received MinRxInt: 2000000, Received Multiplier: 3
Holddown (hits): 0(0), Hello (hits): 2000(2311)
Rx Count: 335, Rx Interval (ms) min/max/avg: 1/29972/14275 last: 292 ms ago
Tx Count: 336, Tx Interval (ms) min/max/avg: 1/29994/14233 last: 53 ms ago
Elapsed time watermarks: 0 0 (last: 0)
Registered protocols: OSPF CEF
Uptime: 01:57:29
Last packet: Version: 1                  - Diagnostic: 0
             State bit: Up               - Demand bit: 0
             Poll bit: 0                 - Final bit: 0
             C bit: 0
             Multiplier: 3               - Length: 24
             My Discr.: 4097             - Your Discr.: 4097
             Min tx interval: 2000000    - Min rx interval: 2000000
             Min Echo interval: 820000
```

R2:

```
CSR2#sh bfd neighbors details

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
10.0.0.1                              4097/4097      Up        Up        Gi1
Session state is UP and using echo function with 920 ms interval.
Session Host: Software
OurAddr: 10.0.0.2
Handle: 1
Local Diag: 0, Demand mode: 0, Poll bit: 0
MinTxInt: 2000000, MinRxInt: 2000000, Multiplier: 3
Received MinRxInt: 1000000, Received Multiplier: 3
Holddown (hits): 0(0), Hello (hits): 2000(2302)
Rx Count: 335, Rx Interval (ms) min/max/avg: 1/29994/14276 last: 53 ms ago
Tx Count: 335, Tx Interval (ms) min/max/avg: 1/29998/14275 last: 293 ms ago
Elapsed time watermarks: 0 0 (last: 0)
Registered protocols: OSPF CEF
Uptime: 01:57:29
Last packet: Version: 1                  - Diagnostic: 0
             State bit: Up               - Demand bit: 0
```

```
            Poll bit: 0                 - Final bit: 0
            C bit: 0
            Multiplier: 3               - Length: 24
            My Discr.: 4097             - Your Discr.: 4097
            Min tx interval: 1000000    - Min rx interval: 1000000
            Min Echo interval: 920000
```

Summary
Echo mode enabled by default.
Default interval for BFD Control packets in IOS – 1s. It can be changed using **`bfd slow-timers`** command.
Default interval for BFD Control packets in IOS-XR – 2s. **Hard-coded**.

2 BFD and protocol configuration

From the following blog:
https://brbccie.blogspot.com/2014/06/everything-bfd.html

So what is BFD (Bidirectional Forwarding Detection)?

BFD is a high-speed "are you up" protocol that other routing protocols subscribe to. It can detect link failures in milliseconds, with the potential for microseconds on the right platform. All routing protocols have some way of detecting failure, usually timer-related. Tuning the timers can theoretically get you sub-second failure detection in some protocols, but this produces unnecessary high overhead as the average IGP wasn't designed with that in mind. BFD was specifically built for fast/low CPU detection, and in the case of single-hop, can offload a great deal of the checks to CEF (by using echo mode - more later), even on a typical router. Some high-end platforms can even offload the entire BFD process to the linecard. The CEF or hardware offload makes BFD a major improvement over the other obvious choice, IP SLA.

Some key items to know:

- BFD has **no neighbor detection**. When the routing protocol needs to monitor a neighbor, it informs BFD, and BFD establishes the neighbor relationship at that point.

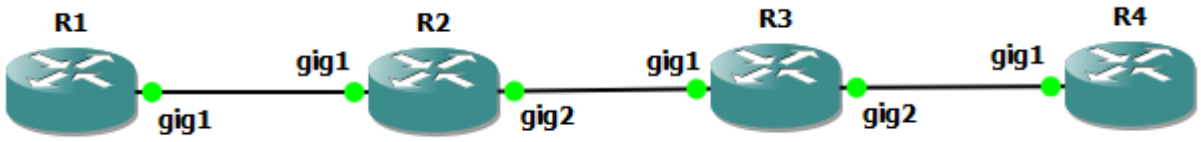
- various routing protocols can piggyback **a single BFD session**. If you have BGP and EIGRP running between the same two subnets on the same two routers, there's no need to have two BFD sessions for checking the same exact topology.

- there are two modes, asynchronous and demand. Asynchronous is the "normal" BFD mode that you're used to when you think Cisco BFD: continuous, high-speed detection of neighbor failure. Demand mode is more of a steady-state operation, where it's assumed the neighbor and link are generally stable, but you'd periodically want to check to see if it's up. Cisco has output for the demand bit in show commands, and they also talk about it (vaguely) in the documentation in places, but best I can tell there's just no command to enable it - at least not on my device (CSR1000v v15.4.1). I read elsewhere that enabling echo mode (more on echo mode later) put the control packets in demand mode, but the demand bit is not set in these cases, so that appears incorrect. Perhaps it would work if the other neighbor initiated? Regardless, assume out of scope for the v5 lab.

- BFD is always unicast.

- BFD control packets are always UDP, sourced from 49152 and sent to 3784. BFD echo packets are also UDP, and are sourced from 3785 and sent to 3785 (why this is will become obvious later).

Let's start with basic, single-hop configuration. This will be our topology:



As with many of my other blogs, I use GNS3 as an easy diagramming tool. In this case, I am not using GNS for the actual lab, because I have never been able to get BFD to work in GNS3. The second the BFD relationship tries to establish, dynamips hard locks. Instead, I am using CSR1000v running on VMWare.

Each router's gigabit interfaces are assigned 192.168.AB.X/24, where AB is the router numbers on the link, and X is the router number: i.e. R2's Gig1 is 192.168.12.2 and Gig2 is 192.168.23.2. Each router has a Loopback0 address of X.X.X.X, where X is the router number: i.e. R1's Loopback0 is 1.1.1.1.

In addition, each router's gigabit interfaces are assigned AB::X/64, where AB is the router numbers on the link, and X is the router number: i.e. R2's Gig1 is 12::2/64. Each router has a Loopback0 address of X::X/128, where X is the router number: i.e. R1's Loopback0 is 1::1/128. IPv6 unicast routing is enabled on all devices.

For this section, we'll mostly be working with R1 and R2.

```
R1#conf t
R1(config)#interface GigabitEthernet1
R1(config-if)#bfd interval 300 min_rx 300 multiplier 3

R2#conf t
R2(config)#interface GigabitEthernet1
R2(config-if)#bfd interval 300 min_rx 300 multiplier 3
```

Now with a routing protocol, you'd probably expect some sort of output now.

```
R1#show bfd neighbor
```

As I mentioned above, no auto-discovery process. The routing protocol has to tell BFD it's needed first, and where to establish its neighbor relationship. It's also very noteworthy from a debugging perspective that if you get the blank output I showed above, you're missing something locally. Having half a BFD session configured (or having your authentication messed up) will produce output with a status of "AdminDown".

Let's get this working:

```
R1(config-if)#ip ospf 1 area 0
R1(config-if)#ip ospf bfd

R2(config-if)#ip ospf 1 area 0
R2(config-if)#ip ospf bfd
```

Now we should see some output.

```
R1#show bfd neighbor

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                         4097/4097       Up        Up        Gi1
```

R2 has a similar output referencing R1.
Now that we have a base config up, let's test the detection.

```
R2(config-if)#shut

R1#
*Jun 21 02:17:21.983: %OSPF-5-ADJCHG: Process 1, Nbr 2.2.2.2 on GigabitEthernet1 from
FULL to DOWN, Neighbor Down: BFD node down
R1#
```

It's hard to demonstrate speed in a blog, but it happens very fast. We see from the output that BFD told the routing protocol that it's neighbor had been lost - "BFD node down".
Let's bring the link back up.

```
R2(config-if)#no shut
```

The command usage is not as simple as it seems - the variable names are terrible, in my opinion. Let's build a more confusing example:

```
R1(config-if)#bfd interval 200 min_rx 500 multiplier 5
R2(config-if)#bfd interval 250 min_rx 400 multiplier 5
```

The first value is the "min_tx" and the second value is the "min_rx". I don't care for the names at all. min_rx from R1 will be compared to min_tx from R2, and a per-direction transmission value will be calculated.

In our scenario above, R1's min_tx - 200 - will be compared to R2's min_rx - 400. The **slower (larger)** value wins. Clearly, 400ms is longer than 200ms, so 400ms will be the negotiated **transmission** value for R1 towards R2. Vice-versa, R2's min_tx is 250ms, and R1's min_rx is 500, so 500ms will be the transmission speed from R2 to R1.

For output clarity I am going to disable echo mode for the moment (not shown here) and show the simplified output of **show bfd neighbor detail**:

```
R1(config-if)#do show bfd neigh det

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                         4097/4097       Up        Up        Gi1
Session state is UP and not using echo function.
Session Host: Software
OurAddr: 192.168.12.1
Handle: 1
Local Diag: 0, Demand mode: 0, Poll bit: 0
MinTxInt: 200000, MinRxInt: 500000, Multiplier: 5
Received MinRxInt: 400000, Received Multiplier: 3
Holddown (hits): 1122(0), Hello (hits): 400(952)
Rx Count: 47, Rx Interval (ms) min/max/avg: 1/500/403 last: 379 ms ago
Tx Count: 58, Tx Interval (ms) min/max/avg: 1/398/331 last: 59 ms ago
Elapsed time watermarks: 0 0 (last: 0)
Registered protocols: OSPF CEF
Uptime: 00:13:25
Last packet: Version: 1                - Diagnostic: 0
             State bit: Up             - Demand bit: 0
             Poll bit: 0               - Final bit: 0
             C bit: 0
             Multiplier: 3             - Length: 24
             My Discr.: 4097           - Your Discr.: 4097
             Min tx interval: 250000   - Min rx interval: 400000
             Min Echo interval: 0
```

Here's the relevant output:

```
Rx Count: 47, Rx Interval (ms) min/max/avg: 1/500/403 last: 379 ms ago
Tx Count: 58, Tx Interval (ms) min/max/avg: 1/398/331 last: 59 ms ago
```

We see that our maximum Rx is 500 (speed from R2 to R1), and maximum Tx is 398 (speed from R1 to R2). It can send/receive a touch faster than this, the idea being that if the BFD control packet doesn't arrive in *under* the maximum time, it'll be considered lost.

The multiplier is reasonably obvious, if you miss that many BFD control packets, consider the link failed.

Since I've got this output up, also noteworthy are:
* **Registered Protocols**: What protocols are "subscribed" to this BFD session? We see OSPF and CEF here.
* Session state is UP and not using echo function. - as I mentioned I disabled echo.
* C bit: 0 - This is only relevant on platforms that can completely hardware offload BFD, and we'll talk about it later.
* Demand bit: 0 - I talked about this earlier, interesting that there's output for it, but I couldn't find any way to enable it.

Since we started with an OSPF example, let me recap what I did above and then we'll look at the alternative way to enable BFD. Presently:

```
R1:
interface GigabitEthernet1
 ip address 192.168.12.1 255.255.255.0
 ip ospf bfd
 ip ospf 1 area 0
 negotiation auto
 bfd interval 200 min_rx 500 multiplier 5
 no bfd echo

R2:
interface GigabitEthernet1
 ip address 192.168.12.2 255.255.255.0
 ip ospf bfd
 ip ospf 1 area 0
 negotiation auto
 bfd interval 250 min_rx 400 multiplier 3
 no bfd echo
```

The other option:

```
R1(config)#int gig1
R1(config-if)#no ip ospf bfd

R2(config-if)#int gig1
R2(config-if)#no ip ospf bfd

R1(config)#router ospf 1
R1(config-router)#bfd all-interfaces

R2(config)#router ospf 1
R2(config-router)#bfd all-interfaces
```

and if you wanted to selectively turn it off on an interface:

```
R2(config-router)#int gig2
R2(config-if)#ip ospf bfd disable

R1(config-router)#do sh bfd neigh

IPv4 Sessions
```

```
NeighAddr                              LD/RD         RH/RS      State     Int
192.168.12.2                            1/1          Up         Up        Gi1
```

Next I'll quickly burn through the rest of the IGPs, and single-hop BGP. For clarity, I've removed all the OSPF config beforehand.

```
R1(config)#router eigrp 100
R1(config-router)#network 0.0.0.0
R1(config-router)#bfd all-interfaces

R2(config)#router eigrp 100
R2(config-router)#network 0.0.0.0
R2(config-router)#bfd interface gig1
```

Note the syntax difference between R1 and R2. Showing both configuration methods at once, single interface vs all interfaces. I of course still have the interface-level BFD config in place.

```
R1#sh ip eigrp neigh
EIGRP-IPv4 Neighbors for AS(100)
H   Address                 Interface              Hold Uptime   SRTT   RTO  Q  Seq
                                                   (sec)         (ms)       Cnt Num
0   192.168.12.2            Gi1                      13 00:01:42 1596  5000  0  3

R1#sh bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS      State     Int
192.168.12.2                            1/1          Up         Up        Gi1
```

Removing EIGRP config for clarity.

RIP is supported, but it's a bit of an oddity. If you know RIP at all, your first question should be "how can BFD work with a neighborless routing protocol?".
It's a bit of a hack.
First item of note, Cisco advertises the feature as "BFD for RIPv2". Just to prove that it's not RIPv2 specific, I'm going to do this lab on RIPv1.

```
R1(config)#router rip
R1(config-router)# version 1
R1(config-router)# network 192.168.12.0
R1(config-router)# neighbor 192.168.12.2 bfd
R1(config-router)# bfd all-interfaces  ! note, this is the only option

R2(config)#router rip
R2(config-router)#version 1
R2(config-router)#network 192.168.12.0
R2(config-router)#neighbor 192.168.12.1 bfd
R2(config-router)#bfd all-interfaces

R1(config-if)#do show bfd neigh
R1(config-if)#
```

Hmm, no luck.

Turns out RIP requires you to be advertising a route *other than the transit link* for the BFD relationship to establish.

```
R1(config-router)#network 1.1.1.1
R2(config-router)#network 2.2.2.2
R1(config-router)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS      State     Int
```

```
192.168.12.2                    4097/1          Up        Up        Gi1
```

So, clearly we can't tear down a non-existent neighbor relationship if the link fails. So what's this good for?

```
R2(config-router)#int gig1
R2(config-if)#shut

R1(config-router)#do sh ip rip data | i 2.0.0.0
2.0.0.0/8 is possibly down
2.0.0.0/8 is possibly down

R1(config-router)#do sh ip route 2.0.0.0
Routing entry for 2.0.0.0/8
  Known via "rip", distance 120, metric 4294967295 (inaccessible)
  Redistributing via rip
  Last update from 192.168.12.2 on GigabitEthernet1, 00:00:56 ago
  Hold down timer expires in 142 secs
```

It marks the route as invalid immediately, rather than waiting on painfully slow RIP timers. Removing RIP config for cleanliness...

Single-hop BGP BFD is very simple. It's also probably the most-deployed implementation of BFD.

```
R1(config)#router bgp 100
R1(config-router)#neighbor 192.168.12.2 remote-as 200
R1(config-router)#neighbor 192.168.12.2 fall-over bfd

R2(config)#router bgp 200
R2(config-router)#neighbor 192.168.12.1 remote-as 100
R2(config-router)#neighbor 192.168.12.1 fall-over bfd

R1(config-router)#do show bfd neigh det | i protocols
```

Registered protocols: **BGP CEF**

There's an extra flag you can use with BGP that takes some explaining. It's called the **C-Bit**, and if you don't understand the usage, it's a confusing thing.

There are some service provider platforms that can run BFD completely in hardware. Meaning, the line card itself knows the BFD logic, and the control plane can actually crash and BFD will keep working. On these platforms, graceful restart (GR) or non-stop forwarding (NSF) can keep the FIB populated on the line card while the control plane reboots itself. GR is actually a negotiated BGP parameter -- when BGP needs to reboot, the neighbor keeps the routes from the rebooting device. In this fashion, the **neighbor** keeps forwarding traffic to the device that's rebooting even though BGP keepalives have failed.

So what's this got to do with BFD?

There could be circumstances where both the control plane needs to reboot and the forwarding plane dies at the same time. BFD can help detect this. Consider this topology.

R1 --> R2. R2 is a provider platform that has NSF enabled.

R1 learns that R2 is an NSF device, and assumes that it's OK for R2's control plane to die and still forward it traffic.

If...: BFD control packets are still coming, and C-BIT = 0 or 1, then R1 should keep forwarding to R2
If...: BFD control packets stop coming, and the C-BIT = 0, then R1 should assume that BFD was run **in software** on the neighbor, and should keep forwarding packets during graceful restart.
If...: BFD control packets stop coming, and the C-BIT = 1, then R1 should assume that BFD was run **in hardware on the linecard** on the neighbor, and that the neighbor is genuinely broken, and to yank the routes rather than wait on graceful restart.

As best I can tell, without a platform to lab this on, the C-Bit is set by the BFD process itself, and isn't

something you can toggle. However, you can tell your BFD process whether to ignore the setting or not. The default is to ignore. If you want to use it:

```
R1(config-router)#neighbor 192.168.12.2 fall-over bfd check-control-plane-failure
```

Of note, this feature is also available in multi-hop BGP, which we'll cover further below.

And now for PIM! I'm not going to build a full multicast lab up here, but we can see the basics.

```
R1(config)#int gig1
R1(config-if)#ip pim sparse-mode
R1(config-if)#ip pim bfd

R2(config)#int gig1
R2(config-if)#ip pim sparse-mode
R2(config-if)#ip pim bfd

R1(config-if)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                          4097/1         Up        Up        Gi1

R1(config-if)#do show bfd neigh det | i protocols
```

Registered protocols: PIM CEF
That's all there is to it. Removing PIM config...
I'll cover HSRP now as well.

```
R1(config)#int gig1
R1(config-if)#standby 1 ip 192.168.12.100
R1(config-if)#standby bfd

R2(config)#int gig1
R2(config-if)#standby 1 ip 192.168.12.100
R2(config-if)#standby bfd

R1(config-if)#do show bfd neigh det | i protocol
Registered protocols: HSRP CEF

R1(config-if)#do show standby | i BFD
    BFD enabled
```

Alternatively, HSRP BFD support can be enabled globally with:

```
R1(config)#standby bfd all-interfaces
```

IOS-based VRRP doesn't appear to have BFD support at the time of this writing. I've seen some documents indicating it is supported in IOS-XR and Nexus. And now for IPv6 IGPs and BGP. OSPFv3's BFD usage is very similar to OSPFv2's.

```
R1(config-if)#int gig1
R1(config-if)#ipv6 ospf 1 area 0
R1(config-if)#ipv6 ospf bfd

R2(config-if)#int gig1
R2(config-if)#ipv6 ospf 1 area 0
R2(config-if)#ipv6 ospf bfd

R1(config-rtr)#do show bfd neigh

IPv6 Sessions
```

```
NeighAddr                               LD/RD         RH/RS      State      Int
FE80::20C:29FF:FECF:21FF                 1/1          Up         Up         Gi1
```

You can also use the "all interfaces" style like from OSPFv2.

EIGRPv6 supports BFD in named EIGRP configuration, which is pretty darn different from the "old" way of doing EIGRPv6:

```
R1(config)# router eigrp FOO
R1(config-router)# address-family ipv6 unicast autonomous-system 1
R1(config-router-af)# af-interface default
R1(config-router-af-interface)# bfd
R1(config-router-af-interface)# exit-af-interface
R1(config-router-af)#  topology base
R1(config-router-af-topology)#  exit-af-topology
R1(config-router-af)#  exit-address-family
```

R2's config is 100% identical so I am omitting it.

```
R1(config-router)#do show bfd neigh

IPv6 Sessions
NeighAddr                               LD/RD         RH/RS      State      Int
FE80::20C:29FF:FECF:21FF                1/1           Up         Up         Gi1

R1(config-router)#do show bfd neigh det | i protocol
Registered protocols: EIGRP CEF
```

and RIPng? No such luck. It's not supported.

Multiprotocol (IPv6) BGP is basically the same as v4:

```
R1(config-router)#router bgp 100
R1(config-router)#neighbor 12::2 remote-as 200
R1(config-router)#neighbor 12::2 fall-over bfd
R1(config-router)#address-family ipv6
R1(config-router-af)#neighbor 12::2 activate

R2(config-router)#router bgp 200
R2(config-router)#bgp log-neighbor-changes
R2(config-router)#neighbor 12::1 remote-as 100
R2(config-router)#neighbor 12::1 fall-over bfd
R2(config-router)#address-family ipv6
R2(config-router-af)#  neighbor 12::1 activate

R1(config-router)#do show bfd neigh

IPv6 Sessions
NeighAddr                               LD/RD         RH/RS      State      Int
12::2                                    1/1          Up         Up         Gi1
```

IPv6 PIM is just as easy as v4:

```
R1(config)#int gig1
R1(config-if)#ipv6 pim bfd
```

HSRP for IPv6:

```
interface GigabitEthernet1
 standby version 2
 standby 1 ipv6 autoconfig
 standby bfd
```

Similar to v4, VRRP support for v6 doesn't seem to be supported in traditional IOS at this time.

Back to v4 for static routing.

Static routing takes a little more work, because there's no IGP to notify BFD of who to peer with, nor is there any neighbor relationship.

Let's start with the most basic usage.

```
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.2
R1(config)#ip route 2.2.2.2 255.255.255.255 GigabitEthernet1 192.168.12.2

R2(config)#ip route static bfd GigabitEthernet1 192.168.12.1
R2(config)#ip route 1.1.1.1 255.255.255.255 GigabitEthernet1 192.168.12.1

R1(config)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                          4097/2         Up        Up        Gi1

R1(config)#do show bfd neigh det | i protocols
Registered protocols: CEF IPv4 Static
```

With the absence of a routing protocol, we use **ip route static bfd <interface> <next-hop to monitor>**

Where <next-hop to monitor> is the IP we'll be pointing out static routes to.
We still need to fulfill two more prerequisites:

- we must have a static route pointing that the specified next-hop. BFD doesn't setup the neighbor otherwise. Alternatively, you can set it up unassociated mode, covered below.
- the static route that points at the next hop must specify the egress interface *if we're doing single-hop routes.* (multi-hop covered below)

But what if the neighbor doesn't need a static route back to us?

Imagine R2 knew R1's routes via another protocol, or even a default, and had no need to setup static routes back towards it:

```
R2(config)#no ip route 1.1.1.1 255.255.255.255 GigabitEthernet1 192.168.12.1

R1(config)#int gig1
R1(config-if)#ip ospf 1 area 0
R1(config-if)#int lo0
R1(config-if)#ip ospf 1 area 0

R2(config)#int gig1
R2(config-if)#ip ospf 1 area 0
```

But, R1 still doesn't have a route to R2's Loopback0. So it needs that static.

Now, the BFD session has failed, because there's no route dependent on R2's statement:
ip route static bfd GigabitEthernet1 192.168.12.1

This is because the dependent method is known as an "associated" route. An unassociated route brings the BFD up anyway:

```
R2(config)#ip route static bfd GigabitEthernet1 192.168.12.1 unassociate

R1(config-if)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
```

```
192.168.12.2                            4097/1          Up        Up        Gi1
```

We can also hierarchically group static routes.

```
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.2 group DOWNSTREAM
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.50 group DOWNSTREAM passive
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.75 group DOWNSTREAM passive
R1(config)#ip route 2.2.2.2 255.255.255.255 GigabitEthernet1 192.168.12.2
R1(config)#ip route 10.10.10.10 255.255.255.255 GigabitEthernet1 192.168.12.50
R1(config)#ip route 100.100.100.100 255.255.255.255 GigabitEthernet1 192.168.12.75
```

Let's walk this line by line:

```
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.2 group DOWNSTREAM
```

This is our non-passive route - basically an anchor route. Let's say for example's sake that from our topology, if the BFD session to 192.168.12.2 (my neighbor) goes down, all the passive routes in my group, DOWNSTREAM, will also be offline. Perhaps they're all attached to some sort of shared Ethernet segment and 192.168.12.2 is the management IP of the first switch - if we can't reach it, we can't reach other devices on it's link. There may be no reason to run BFD with the other devices, as perhaps they're on super-stable/redundant links. However, we need to pull them from our table if they're not reachable.

```
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.50 group DOWNSTREAM passive
R1(config)#ip route static bfd GigabitEthernet1 192.168.12.75 group DOWNSTREAM passive
```

192.168.12.50 and 192.168.12.75 are imaginary next-hops on the shared Ethernet segment of 192.16812.0. They don't exist in our topology anywhere, but they don't need to for our example. They're passive, meaning they're reliant on the status from the anchor BFD session (the non-passive entry). If it goes down, they need to fail their BFD "status" too.

```
R1(config)#ip route 2.2.2.2 255.255.255.255 GigabitEthernet1 192.168.12.2
```

This references our "anchor" next-hop, and is necessary for BFD to establish.

```
R1(config)#ip route 10.10.10.10 255.255.255.255 GigabitEthernet1 192.168.12.50
R1(config)#ip route 100.100.100.100 255.255.255.255 GigabitEthernet1 192.168.12.75
```

These are our routes that reference the passive BFD next-hops above.
BFD is already up, and we can see the imaginary downstream hosts 10.10.10.10 and 100.100.100.100 are installed in our routing table.

```
R1#sh ip route | b subnets
      1.0.0.0/32 is subnetted, 1 subnets
C        1.1.1.1 is directly connected, Loopback0
      2.0.0.0/32 is subnetted, 1 subnets
S        2.2.2.2 [1/0] via 192.168.12.2, GigabitEthernet1
      10.0.0.0/32 is subnetted, 1 subnets
S        10.10.10.10 [1/0] via 192.168.12.50, GigabitEthernet1
      100.0.0.0/32 is subnetted, 1 subnets
S        100.100.100.100 [1/0] via 192.168.12.75, GigabitEthernet1
      192.168.12.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.12.0/24 is directly connected, GigabitEthernet1
L        192.168.12.1/32 is directly connected, GigabitEthernet1
```

I'm going to fail the interface on R2, which, of important note, does not bring down the line protocol on R1 in my virtual lab.

```
R2#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#int gig1
R2(config-if)#shut

R1#sh ip route | b subnets
```

```
      1.0.0.0/32 is subnetted, 1 subnets
C        1.1.1.1 is directly connected, Loopback0
      192.168.12.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.12.0/24 is directly connected, GigabitEthernet1
L        192.168.12.1/32 is directly connected, GigabitEthernet1
```

and all three routes gone!

Now, just to prove my line protocol is still up on that subnet, and this is actually BFD removing the routes and it isn't just a generic next-hop failure:

```
R1#sh ip cef 192.168.12.50
192.168.12.0/24
  attached to GigabitEthernet1

R1#sh ip cef 192.168.12.75
192.168.12.0/24
  attached to GigabitEthernet1
```

The IPv6 implementation isn't quite as feature-filled:

```
R1(config)#ipv6 route static bfd GigabitEthernet1 12::2
R1(config)#ipv6 route 2::/64 GigabitEthernet1 12::2

R2(config)#ipv6 route static bfd GigabitEthernet1 12::1 unassociated

R2(config)#do show bfd neigh | b IPv6
IPv6 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
12::1                                   2/2          Up        Up        Gi1
```

This works much the same way IPv4 does - R1 specifies an associated BFD neighbor and corresponding static route, R2 has an unassociated route (we'll assume it knows how to get back to R1 through other means). And... that's it for v6. No groups!

I've left all the multihop options for one section, as they all share some of the same configuration.

We'll start with multihop IPv4 BGP. Now we'll be peering R1 to R4. I've setup interim IGPs throughout; assume full reachability.

```
R1(config)#bfd-template multi-hop MHOP-TEMPLATE
R1(config-bfd)#interval min-tx 300 min-rx 300 multiplier 3
R1(config)#bfd map ipv4 4.4.4.0/24 1.1.1.0/24 MHOP-TEMPLATE
R1(config)#router bgp 14
R1(config-router)#neighbor 4.4.4.4 remote-as 14
R1(config-router)#neighbor 4.4.4.4 update-source lo0
R1(config-router)#neighbor 4.4.4.4 fall-over bfd multi-hop

R4(config)#bfd-template multi-hop MHOP-TEMPLATE
R4(config-bfd)#interval min-tx 300 min-rx 300 multiplier 3
R4(config)#bfd map ipv4 1.1.1.1/32 4.4.4.4/32 MHOP-TEMPLATE
R4(config)#router bgp 14
R4(config-router)#neighbor 1.1.1.1 remote-as 14
R4(config-router)#neighbor 1.1.1.1 update-source lo0
R4(config-router)#neighbor 1.1.1.1 fall-over bfd multi-hop
```

We'll walk through this config as well:

```
R1(config)#bfd-template multi-hop MHOP-TEMPLATE
R1(config-bfd)#interval min-tx 300 min-rx 300 multiplier 3
```

This is just a series of settings to apply to the multi-hop session. Clearly we can't glean it from the interface BFD configuration because there might be different settings for different neighbors. Of note, you can also set

authentication here. There are also single-hop templates, which we'll talk about later.

```
R1(config)#bfd map ipv4 4.4.4.0/24 1.1.1.0/24 MHOP-TEMPLATE
```

The BFD map is the slightly confusing part. This statement could be interpreted as:

"**If** I establish a multi-hop BFD session to a destination inside 4.4.4.0/24, sourced from any of my interfaces inside of 1.1.1.0/24, then use the settings from MHOP-TEMPLATE"

Note it doesn't matter what mask size you use on this. In fact, if you look at R2, I specifically used /32s instead, just to prove a point. As long as the mask encompasses the IPs in question, you're good.

It's also important to note that the BFD map isn't neighbor discovery or a static neighbor. It just assigns settings to a neighbor session that another protocol informs BFD of.

Also important to note as, at least for me, the configuration is backwards from the way I think. It's destination/source: 4.4.4.0/24 is my TARGET, 1.1.1.0/24 is my SOURCE. I mis-type it almost every time, because I think source/dest.

```
R1(config)#router bgp 14
R1(config-router)#neighbor 4.4.4.4 remote-as 14
R1(config-router)#neighbor 4.4.4.4 update-source lo0
R1(config-router)#neighbor 4.4.4.4 fall-over bfd multi-hop

The BGP config is pretty obvious.

And the outcome...

R1(config-router)#do show bfd neigh

IPv4 Multihop Sessions
NeighAddr[vrf]                                LD/RD         RH/RS     State
4.4.4.4                                     4097/4097       Up        Up
```

Let's validate by shutting down the link between R2 and R3, which is not participating in BFD other than forwarding packets for R1 and R4.

```
R2(config)#int gig2
R2(config-if)#shut

R1(config-router)#
*Jun 24 04:26:34.371: %BGP-5-NBR_RESET: Neighbor 4.4.4.4 reset (BFD adjacency down)
*Jun 24 04:26:34.371: %BGP-5-ADJCHANGE: neighbor 4.4.4.4 Down BFD adjacency down
*Jun 24 04:26:34.372: %BGP_SESSION-5-ADJCHANGE: neighbor 4.4.4.4 IPv4 Unicast topology
base removed from session  BFD adjacency down
```

BGP IPv6 multi-hop is identical, so I'm not going to demonstrate it here.

You may want to consider QoS on the interim routers when it comes to BFD. Not very helpful if your RTP packets continuously push your BFD out of the way, just to have BFD completely remove the link:
- BFD packets are marked with precedence 6 by default
- Be sure the value isn't reset by your interim routers, and that they prioritize/LLQ the Prec 6 traffic.

This leads us into static route multihop. I've removed the previous BGP config.

Much the same as BGP multihop BFD, static route multihop BFD uses multihop templates and BFD maps. Let's create a static route multihop session between R1 and R3. I've added a new loopback to R3, Lo1, with IP address 33.33.33.33/32 for validation purposes. It is not in the IGP.

```
R1(config)#bfd-template multi-hop MHOP-TEMPLATE
R1(config-bfd)#interval min-tx 300 min-rx 300 multiplier 3
R1(config)#bfd map ipv4 192.168.23.0/24 192.168.12.0/24 MHOP-TEMPLATE
R1(config)#ip route static bfd 192.168.23.3 192.168.12.1
```

```
R1(config)#ip route 33.33.33.33 255.255.255.255 192.168.23.3

R3(config)#bfd-template multi-hop MHOP-TEMPLATE
R3(config-bfd)#interval min-tx 300 min-rx 300 multiplier 3
R3(config-bfd)#bfd map ipv4 192.168.12.0/24 192.168.23.0/24 MHOP-TEMPLATE
R3(config)#ip route static bfd 192.168.12.1 192.168.23.3 unassociate

R1(config)#do show bfd neigh

IPv4 Multihop Sessions
NeighAddr[vrf]                                LD/RD         RH/RS     State
192.168.23.3                                 4097/4097      Up        Up
```

Most of this config should be familiar if you read the entire article up until now, but there are some peculiar ways to do this incorrectly that will bust it.

```
R1(config)#ip route static bfd 192.168.23.3 192.168.12.1
```

This may seem very similar to single-hop, but here's a sample from single-hop above:

```
R2(config)#ip route static bfd GigabitEthernet1 192.168.12.1
```

Note the lack of an interface on multi-hop, and the presence of one in single-hop. These are a mutually exclusive setting: You *must not* specify an interface on multi-hop, and you *must* specify an interface on single-hop. This is very poorly documented, unfortunately - the samples on the DocCD do show the right thing, but it never calls it out like this.

```
R1(config)#ip route 33.33.33.33 255.255.255.255 192.168.23.3
```

A normal static route from our multihop config - but let's look at our earlier single-hop sample:

```
R2(config)#ip route 1.1.1.1 255.255.255.255 GigabitEthernet1 192.168.12.1
```

Now this genuinely surprised me. If you specify the interface on a static route with multi-hop - even though *all the other information needed is present* - destination prefix and next hop - it will break multi-hop BFD. On the other hand you **must** have it for single-hop. Check out a quick before & after on multihop:

```
R1(config)#ip route 33.33.33.33 255.255.255.255 192.168.23.3
R1(config)#do show bfd neigh

IPv4 Multihop Sessions
NeighAddr[vrf]                                LD/RD         RH/RS     State
192.168.23.3                                 4097/4097      Up        Up

R1(config)#no ip route 33.33.33.33 255.255.255.255 192.168.23.3
R1(config)#ip route 33.33.33.33 255.255.255.255 Gigabit1 192.168.23.3
R1(config)#do show bfd neigh
R1(config)#
```

Multi-hop static BFD can also use groups like single-hop can, but the config is identical (aside from not specifying the egress interfaces!), so I'm going to skip them here for brevity.

I've referred to echo mode in various places in the article up until now. **Echo mode** is a very clever way of decreasing BFD's hit on the CPU. It took me a while to figure out how it worked, however, mostly because the RFC wins the "too vague" award of the year: "When the Echo function is active, a stream of BFD Echo packets is transmitted in such a way as to have the other system loop them back through its forwarding path." http://tools.ietf.org/html/rfc5880

I already knew echo mode was a way to save on CPU, so I theorized that the idea was to get the BFD "are you up?" packets to be processed in fast switching instead of the control plane, but that description doesn't exactly explain it programatically. After more googling and some Wireshark, I figured out the implementation.

Echo is single-hop only, so let's use R1 and R2 as my examples.

R1 sends an echo packet (instead of a control packet) to R2, formatted as:

```
  L3 Source: R1 (192.168.12.1)
  L3 Destination: R1 (192.168.12.1)
  MAC Source: Itself (000c.298f.aca3)
  MAC Destination: (000c.29cf.21ff)
```

R2's receives this packet, sees this packet, and CEF-switches it straight back to R1! In this fashion, R1 knows that R2 is reachable.

R2 would perform similar behavior towards R1, for it's own echo process.

There's more to know, however:
- the echo packets are sent at the rate negotiated in the BFD interval (on interface or single-hop template)
- echo mode is only supported single-hop, obviously.
- control-plane packets are still sent, but they are sent at the "slow timers" speed, specified as: **bfd slow-timers** <speed>. Since the control packets are no longer vital to knowing that the neighbor is up at high-speed, you can crank down these heavier-CPU-intensive packets to slower rates.
- the Cisco documentation says you need to disable ICMP redirects first - as technically speaking, the traffic above should generate a redirect - but in modern 15.1x+ IOS I have yet to see this requirement; it appears IOS is smart enough to know not to send redirects to echo packets.
- echo mode is on by default. It needs to be on on both sides of the link in order to work.

On a side note, I've periodically had problems getting echo mode to come up when labbing on the CSR1000v; it usually seems to have to do with other BFD config on the device. I would call it a bug. With some cleanup and tinkering you can usually get it to come up.

```
R1(config)#int gig1
R1(config-if)#bfd echo
! this is on by default, but I'd disabled it earlier in the article.
R1(config-if)#ip ospf bfd
R1(config-if)#exit
R1(config)#bfd slow-timers 30000 ! send control packets every 30 seconds

R2(config)#int gig1
R2(config-if)#bfd echo
R2(config-if)#ip ospf bfd
R2(config-if)#exit
R2(config)#bfd slow-timers 30000

R1#show bfd neigh det | i echo
Session state is UP and using echo function with 400 ms interval.

R1#show bfd neigh det | i Min
MinTxInt: 30000000, MinRxInt: 30000000, Multiplier: 5
Received MinRxInt: 30000000, Received Multiplier: 3
             Min tx interval: 30000000   - Min rx interval: 30000000
             Min Echo interval: 400000
```

We now see in "Min Echo interval" that the echo packets are going at the pace we expected control packets at before (400ms - negotiated by the interface values), and control packets are now sending every 30 seconds.

I mentioned single-hop templates briefly above. They're not of much use outside of authentication and dampening:

```
R1(config)#bfd-template single-hop TEST
R1(config-bfd)#?
BFD template configuration commands:
  authentication  Authentication type
  dampening       Enable session dampening
  echo            Use echo adjunct as bfd detection mechanism
  interval        Transmit interval between BFD packets
```

Dampening works much the same way as any other protocol's dampening works. If the BFD session flaps a bunch, mark it as "down" (pull it out of the routing table) for a certain amount of time to wait on stabilization. I did lab this and it does work, but it's too hard to demonstrate it in a blog, so here's the basic usage:

```
R1(config)#bfd-template single-hop TEST-SH
R1(config-bfd)#interval both 300 multiplier 3
R1(config-bfd)#dampening 5 4000 4000 10
R1(config-bfd)#int gig1
R1(config-if)#bfd ?
R1(config-if)#no bfd interval 200 min_rx 500 multiplier 5 ! mutually exclusive from a
single-hop template
R1(config-if)#bfd template TEST-SH
```

BFD Authentication is also reasonably straightforward.

```
R1(config-if)#key chain BFD
R1(config-keychain)#key 1
R1(config-keychain-key)#key-string cisco
R1(config-keychain-key)#exit
R1(config-keychain)#exit
R1(config)#bfd-template single-hop TEST-SH
R1(config-bfd)#authentication sha-1 keychain BFD
```

Since we configured this on only one side....

```
R1(config-bfd)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                          4097/0         Down      Down      Gi1

R2(config-if)#key chain BFD
R2(config-keychain)#key 1
R2(config-keychain-key)#key-string cisco
R2(config-keychain-key)#exit
R2(config-keychain)#exit
R2(config)#bfd-template single-hop TEST-SH
R2(config-bfd)#interval both 300 multiplier 3
R2(config-bfd)#authentication sha-1 keychain BFD
R2(config-bfd)#int gig1
R2(config-if)#no bfd interval 250 min_rx 400 multiplier 3
R2(config-if)#bfd-template single-hop TEST-SH

R1(config-bfd)#do show bfd neigh

IPv4 Sessions
NeighAddr                              LD/RD         RH/RS     State     Int
192.168.12.2                          4097/4097      Up        Up        Gi1
```

BFD authentication can use MD5, SHA1, or meticulous MD5 or SHA1. So what's meticulous? Out of scope of this document, but here's the RFC: http://tools.ietf.org/html/draft-ietf-bfd-generic-crypto-auth-06

And last but certainly not least, how do you debug BFD? Honestly, most of the times I break BFD, it's because I missed a requirement - for example, forgetting to put an egress interface on single-hop static routes. In these circumstances, you get nearly zero debug output, because IOS doesn't detect that anything needs to happen.

If you can get BFD to realize you're trying to get it to work, you can see some inner-workings with:
debug bfd event

I hope you enjoyed,

3 BFD and port-channels

The discriminator fields are used to distinguish the different bfd sessions, but was also used on Cisco proprietary implementation on Bundles or Port-Channels. This was a great problem because bfd couldn't be used between different vendors on PoCh interfaces, until **rfc 7130** that should have solved the problem.