Filter Based Forwarding on MX

Several examples of FBF on MX

David Roy

2025-04-10

TTLBOOKS
BY JUNIPER & HPE NETWORKING

TIME TO LEARN...

Contents

List of Figures

List of Tables

Goal of this Article

This article provides a detailed overview of Filter-Based Forwarding (**FBF**), also known as Policy-Based Routing (**PBR**), on MX Series routers (AFT), using common deployment scenarios to illustrate configuration methods.

The Filter-Based Forwarding (FBF) concept is relatively simple. On ingress, filtering (via the Firewall Filter toolkit) is applied *before* the the source or destination route lookup. The diagram below illustrates this process. In a standard routing scenario without any constraints, the destination IP from the IP datagram is used for a rou te lookup (using Longest Prefix Match[1]), which returns a next-hop and an associated egress interface. (Encapsulation may occur prior to egress.)

With FBF, we alter the ingress lookup behavior in one of the following ways:

- Forcing traffic to exit through a specific egress port;
- Using a "proxy" or alias IP address as the lookup key (as shown in our example below);
- Leveraging a specific, constrained forwarding instance to influence the lookup outcome.

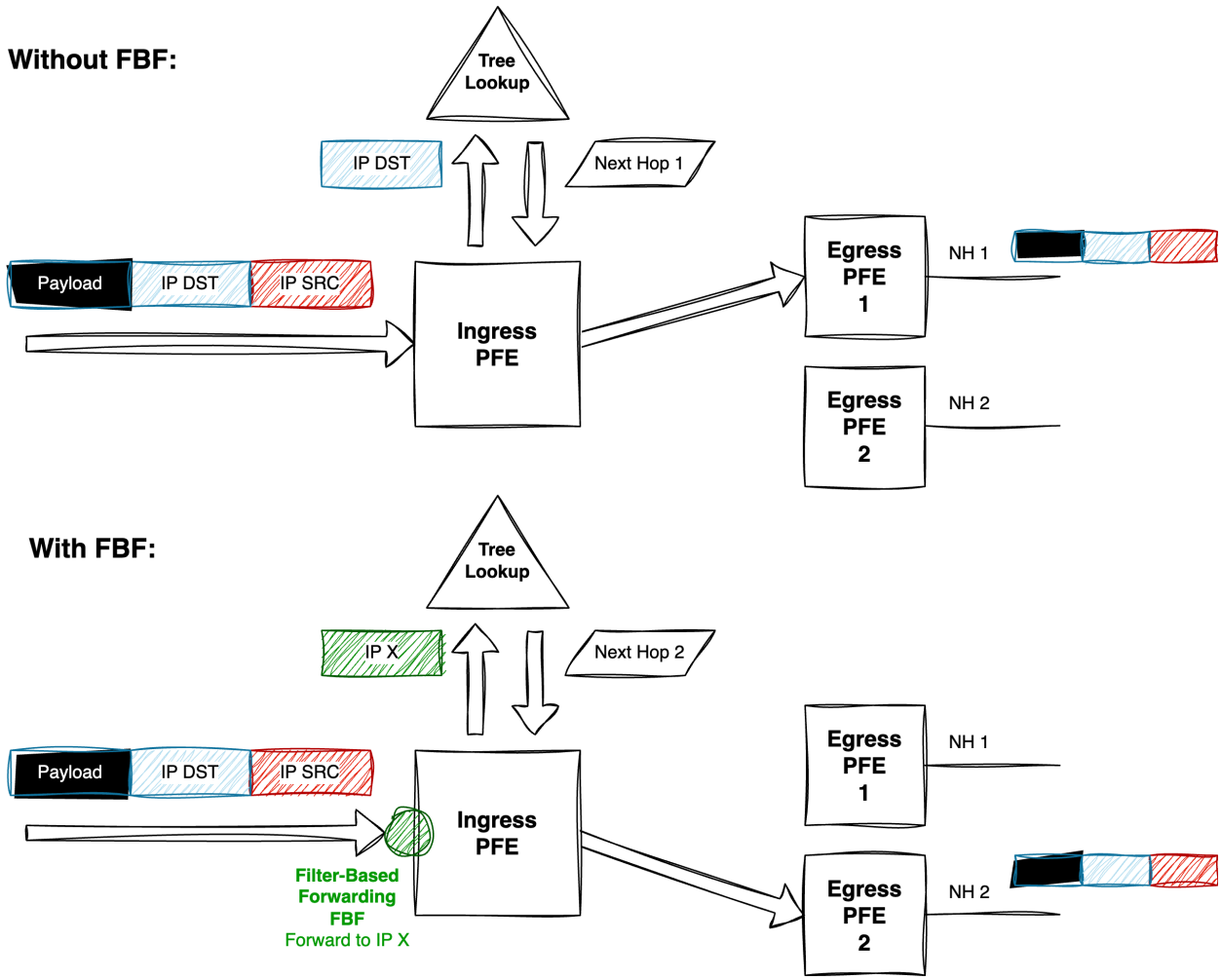

Figure 1: FBF concepts

To summarize, on Juniper platforms, FBF can generally be implemented using two main approaches. The

[1] aka. LPM

first is more straightforward and involves minimal configuration, but offers limited flexibility. It uses a single firewall filter to directly redirect traffic. The second approach requires slightly more configuration but offers more granular traffic handling.

All examples are based on Junos OS release **24.2**. We'll begin with the simpler method.

Case 1 - FBF Using next-ip / next-interface Actions

RLI 14784 introduces two new terminating actions in firewall filters:

- next-interface routing-instance
- next-ip routing-instance

> **Note:** IPv6 is also supported via the `next-ip6` variant.

These actions are **terminating**, meaning there's no need to include an explicit `accept` statement. Both are supported under the `inet` and `inet6` families.

The typical usage for these actions is illustrated below:

```
firewall {
    family inet|inet6 {
      filter foo {
        from {
          Any from the set of ip match conditions
        } then {
          next-interface <intf-name> routing-instance <RI-name>
      }
    }
  }

  family inet {
      filter foo4 {
        from {
          Any from the set of ipv4 match conditions
        } then {
          next-ip <prefix> routing-instance <RI-name>
      }
    }
  }

  family inet6 {
      filter foo6 {
        from {
          Any from the set of ipv6 match conditions
        } then {
          next-ip6 <prefix> routing-instance <RI-name>
      }
    }
  }
}
```

Overall Behavior

When a packet matches a term using one of the below actions:

- **`next-interface`**: The system verifies the operational state of the specified interface, along with the availability of an ARP (or ND for IPv6) entry for the next-hop. If a `routing-instance` is specified, the interface lookup is performed within that context. If all conditions are met, the packet is forwarded via the corresponding egress IFL. If the interface is down or unresolved, the packet is dropped.

- **`next-ip(6)`**: The IP address specified in `next-ip` or `next-ip6` is not automatically resolved. On Ethernet interfaces, reachability must be ensured through routing—either via dynamic protocols or static routes. If a matching route (exact or more specific) is found, the packet follows the next-hop associated with that route. If no matching route exists, the packet is **rejected**.

Read carefully: if next-ip address becomes unreachable the default approch is to point the traffic to the default reject next-hop. Traffic rejected are thus punted to the RE for sending back an ICMP unreachable. However, no worries about "overloading" the internal host-path. Indeed, there is a default DDOS protection policer that will rate-limit those rejected punted packets to 2Kpps. We will see later, how to handle this behavior in case you want silencly discard the packet when next-ip address becomes unreachable.

Caveats

Known limitations include:

- Supported only for **ingress** filtering
- No fallback mechanism (The EVO **exact** match option is not supported)
- Not supported on **LT** interfaces

Example 1 - topology

The diagram below illustrates the topology used to demonstrate simple FBF on an MX platform.
The Device Under Test (**DUT**) is an **MX480** equipped with an **MPC10E** line card.

This simplified setup represents a typical **DCI** router connected to an **IP Fabric**, providing access to remote resources via two distinct paths:

- A **quality** path through an **MPLS/SR** core network, and
- A **best-effort** path via a **direct peering or transit (PNI)** connection.

By default, remote resources are reached via the direct PNI link. The DUT hosts an **Internet VRF**, which is also used by a remote **ASBR** in the same AS. This ASBR advertises "public/remote" prefixes to all PE routers—including the DUT—via **L3VPN (inet-vpn)**. The direct PNI interface is also part of the Internet VRF.

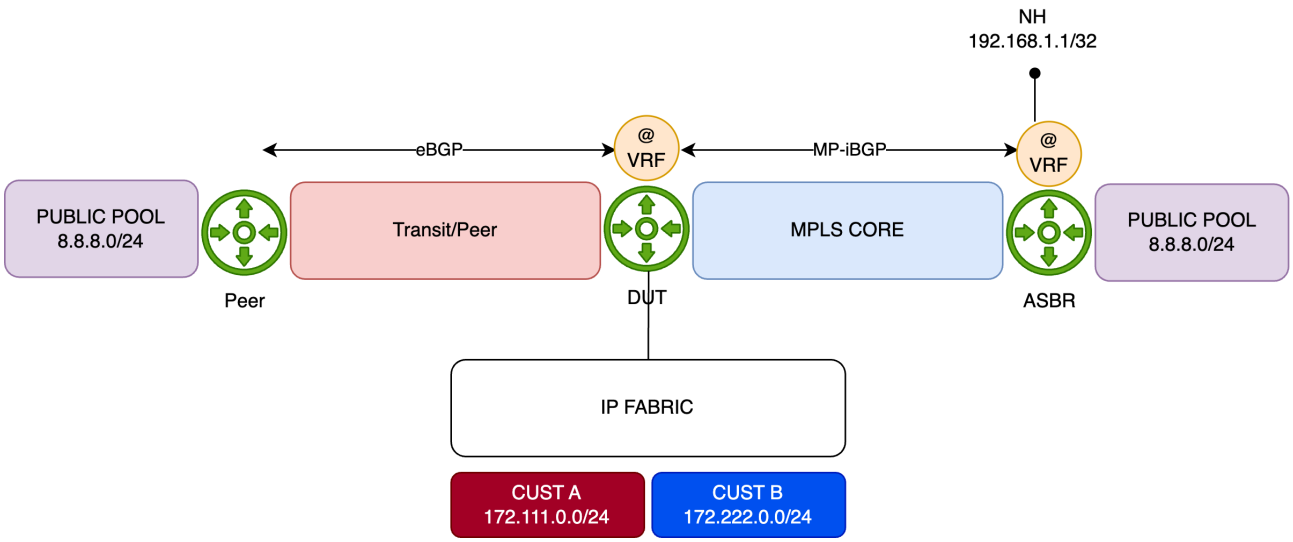

Figure 2: Network topology

For demonstration purposes, the remote resource is simulated using the public prefix **8.8.8.0/24**. This prefix is preferred by default via the direct PNI, with a backup path available through the MPLS/SR core:

```
regress@rtme-mx-62> show route 8.8.8.0/24

VRF1.inet.0: 9 destinations, 13 routes (9 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

8.8.8.0/24          *[BGP/170] 00:00:04, localpref 10000
                       AS path: 1234 6000 I, validation-state: unverified
                     >  to 172.16.8.1 via et-2/0/0.0                          <<< Via PNI
                     [BGP/170] 21:23:18, localpref 100, from 193.252.102.201
                       AS path: I, validation-state: unverified
                     >  to 172.16.1.1 via et-5/0/0.0, Push 16, Push 20103(top)   <<< Via MPLS/SR
                         Core
```

The IP Fabric serves two customer types—**A** and **B**—represented by IP prefixes **172.111.0.0/24** and **172.222.0.0/24**, respectively. The DUT exchanges IP traffic with these customers directly.

This is just a table for test:

Table 1: Interface Naming of the 6xQSFPP PIC

PFE COMPLEX	Port Number	10GE mode	40GE or 100GE modes
EA ASIC 0	0	xe-x/1/0:[0..3]	et-x/1/0
	1	xe-x/1/1:[0..3]	et-x/1/1
	2	xe-x/1/2:[0..3]	et-x/1/2
	3	xe-x/1/3:[0..3]	et-x/1/3
EA ASIC 1	4	xe-x/1/4:[0..3]	et-x/1/4
	5	xe-x/1/5:[0..3]	et-x/1/5
	6	xe-x/1/6:[0..3]	et-x/1/6
	7	xe-x/1/7:[0..3]	et-x/1/7
EA ASIC 2	8	xe-x/1/8:[0..3]	et-x/1/8
	9	xe-x/1/9:[0..3]	et-x/1/9
	10	xe-x/1/10:[0..3]	et-x/1/10
	11	xe-x/1/11:[0..3]	et-x/1/11

Initial Configuration

Below is the initial configuration for the DUT's **Internet VRF**, kept simple for clarity:

- The DUT receives customer prefixes via **eBGP** from the peer group **FABRIC**.
- It receives public prefixes from the peer ASBR via **eBGP**, with an import policy (**PREF**) setting a high **local preference** to make this path the best.
- Two interfaces belong to the Internet VRF—one facing the IP Fabric, and the other towards the PNI peer.
- The DUT also connects to the MPLS core, running **IS-IS with Segment Routing** for label distribution.

```
regress@rtme-mx-62> show configuration routing-instances VRF1
instance-type vrf;
```

```
protocols {
    bgp {
        group FABRIC {
            type external;
            local-address 172.16.0.4;
            peer-as 5000;
            neighbor 172.16.0.5;
        }
        group PEER {
            type external;
            local-address 172.16.8.0;
            import PREF;
            family inet {
                unicast;
            }
            peer-as 1234;
            neighbor 172.16.8.1;
        }
    }
}
interface et-4/0/0.0;
interface et-5/2/0.100;
route-distinguisher 193.252.102.101:1;
vrf-target target:65000:1234;
vrf-table-label;
```

Configuration of FBF

Using the previous topology, we demonstrate a typical FBF use case leveraging the **`next-ip`** action (the same behavior applies to **`next-ip6`** and **`next-interface`**).

The objective is to override the default forwarding behavior—where traffic exits via the direct PNI interface—for traffic sourced from **Customer B** (**172.222.0.0/24**). Instead, traffic from this prefix should be redirected through the **MPLS backbone**, targeting the remote ASBR to reach the public resource.

Traffic from other sources will continue to follow the default "best path," which remains the direct PNI link.

The diagram below illustrates this behavior:

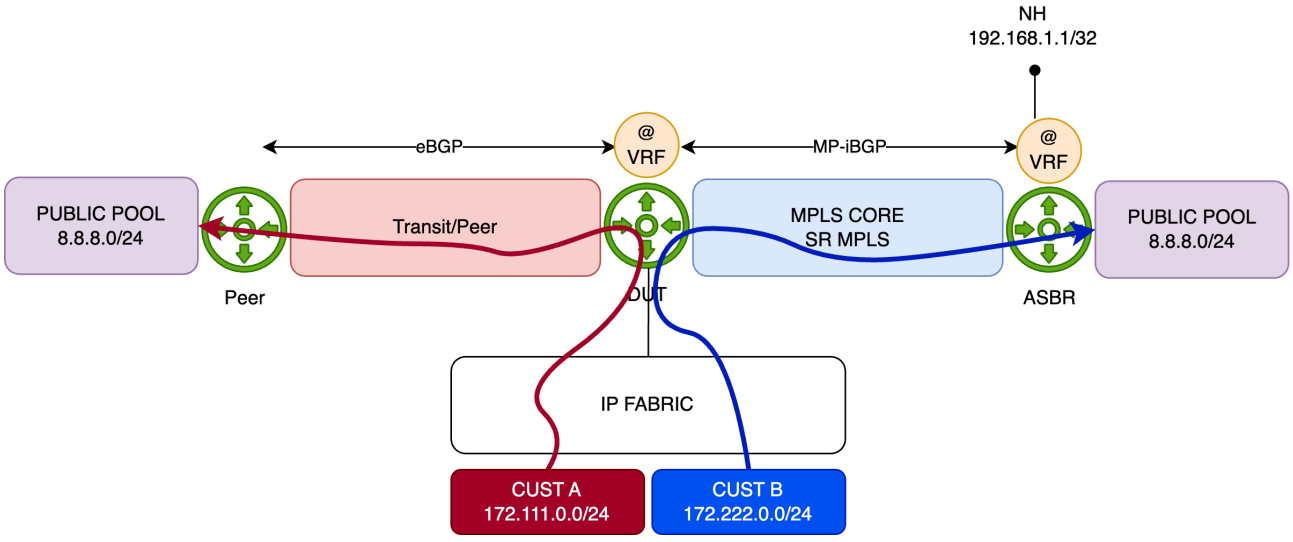

Figure 3: Source based forwarding with next-ip

How will we achieve this?

The configuration is straightforward. First, we define a firewall filter that matches the source prefix **172.222.0.0/24**, and apply the `next-ip` action to redirect traffic.

Which `next-ip` address should be used?

That depends on the network design. In this example, we target the **loopback address of the remote ASBR**, which is advertised via **BGP (L3VPN)**. As shown below, the route to this loopback is reachable through the MPLS/SR core via an established tunnel:

```
regress@rtme-mx-62> show route 192.168.1.1 table VRF1.inet

VRF1.inet.0: 9 destinations, 13 routes (9 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.1.1/32     *[BGP/170] 20:42:26, localpref 100, from 193.252.102.201
                      AS path: I, validation-state: unverified
                    >  to 172.16.1.1 via et-5/0/0.0, Push 16, Push 20103(top)
```

Now let's configure the FBF filter. Since we're operating within a **VRF context**, the `routing-instance` parameter is specified along with the `next-ip` action—this ensures that the next-hop lookup is performed in the correct **FIB instance**.

An additional term is included to match all remaining traffic, allowing it to follow the default forwarding behavior.

```
family inet {
    filter FBF {
        term PBR_CUSTOMER_B {
            from {
                source-address {
                    172.222.0.0/24;
                }
            }
            then {
                count CUSTOMERB;
                next-ip 192.168.1.1/32 routing-instance VRF1;
            }
        }
        term OTHER {
            then {
                count OTHER;
                accept;
            }
        }
    }
}
```

Before applying the filter, we'll generate traffic from **Customer A** and **Customer B**. To distinguish between the two flows, we configure the traffic rates as follows:

- **Customer A**: 1000 packets per second (pps)
- **Customer B**: 5000 packets per second (pps)

Both customers will send traffic toward the **8.8.8.0/24** prefix.

We now start the traffic and verify the statistics on the **PNI interface**:

```
regress@rtme-mx-62> monitor interface et-2/0/0.0

<- truncated output ->

Remote statistics:
  Input bytes:                132708516 (0 bps)                          [0]
  Output bytes:             48292808336 (23518344 bps)              [11195520]
  Input packets:                270844 (0 pps)                             [0]
  Output packets:             98556757 (6000 pps)                     [22848] <<< Customer A +
      Customer B traffics
```

At this point, all traffic is following the best active path—via the **PNI interface**—to reach the **8.8.8.0/24** prefix.

Another table just for test:

Table 2: Interface Naming of the 4xQSFPP PIC

PFE COMPLEX	Port Number	10GE mode	40GE or 100GE mode
EA ASIC 0	0	xe-0/0/0:[0..3]	et-0/0/0
	1	xe-0/0/1:[0..3]	et-0/0/1
	2	xe-0/0/2:[0..3]	et-0/0/2
	3	xe-0/0/3:[0..3]	et-0/0/3

We now apply the **FBF filter** in the **ingress** direction on the interface connected to the **IP Fabric**:

```
edit private

set interfaces et-5/2/0 unit 100 family inet filter input FBF

commit comment "ADD_FBF" and-quit
```

And then, recheck the PNI interface statistics:

```
regress@rtme-mx-62> monitor interface et-2/0/0.0
Interface: et-2/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:                 132708516 (0 bps)                          [0]
  Output bytes:              49128737556 (3921056 bps)                    [0]
  Input packets:                   270844 (0 pps)                         [0]
  Output packets:              100262735 (1000 pps)                       [0] <<< Only Customer
       A traffic
```

The **FBF filter** is functioning as expected. Only **Customer A** traffic continues to follow the default best path toward **8.8.8.0/24** via the PNI interface.

Next, we check the statistics on the **core-facing interfaces** to confirm that **Customer B** traffic is being properly redirected through the **SR/MPLS tunnel** as intended by the FBF configuration:

```
regress@rtme-mx-62> monitor interface et-5/0/0.0
Interface: et-5/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:                 253750639 (584 bps)                        [0]
  Output bytes:              57241664165 (19600072 bps)                   [0]
  Input packets:                   533880 (1 pps)                         [0]
  Output packets:              116839924 (5000 pps)                       [0] <<< the tunneled
      Customer B traffic
```

Everything looks good! To demonstrate that there is no fallback mechanism with **next-ip**-based FBF, we'll remove the **loopback (192.168.1.1/32)** announcement from the ASBR. As a result, the **DUT** will no longer have a route to the loopback, and the traffic will be dropped:

```
regress@rtme-mx-62> show route 192.168.1.1 table VRF1.inet
```

This means that traffic from **Customer B** should be dropped, which is exactly what we observe. As shown below, there is no longer any traffic on the **Core interface**, and **Customer A** traffic continues to flow through the **PNI port**.

```
regress@rtme-mx-62> monitor interface et-5/0/0.0
```

```
Interface: et-5/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:                 253763429 (248 bps)                          [0]
  Output bytes:              58533097602 (0 bps)                           [57]
  Input packets:                   534091 (0 pps)                           [0]
  Output packets:              119475689 (0 pps)                            [1] <<< Customer B
      traffic dropped

monitor interface et-2/0/0.0
Interface: et-2/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:                 132708516 (296 bps)                          [0]
  Output bytes:              49685006066 (3921096 bps)                      [0]
  Input packets:                   270844 (0 pps)                           [0]
  Output packets:              101397976 (1000 pps)                         [0] <<< Customer A
      still forwarded
```

As discussed earlier, the default action when next-ip address becomes unreachable is to redirect traffic to the reject next-hop. Above, we issue a show route of the next-ip address and nothing was return as expected. Just now issue the show route forwarding-table:

```
regress@rtme-mx-62> show route forwarding-table destination 192.168.1.1 table VRF1
Routing table: VRF1.inet
Internet:
Destination        Type RtRef Next hop           Type Index    NhRef Netif
default            perm     0                    rjct      695     1       <<< rjct = reject
    next-hop
```

The route points to reject next-hop in the FIB. As said, the next-hop will punted the packets to the RE for further processing (ICMP unreachable). As also mentioned those punted packet are rate-limited by the ASIC to 2kpps. We can verify this behavior, by checking the DDOS protection statistics for the “reject” protocol:

```
regress@rtme-mx-62> show ddos-protection protocols reject statistics terse
Packet types: 1, Received traffic: 1, Currently violated: 1

Protocol    Packet      Received        Dropped        Rate     Violation State
group       type        (packets)       (packets)      (pps)    counts
reject      aggregate   6396663340756   6395549305435  5000     9         viol
```

We saw our 5K of Customer B traffic before being rate-limited. Issue the “violation” check command to see the rate-limit value of 2K pps:

```
regress@rtme-mx-62> show ddos-protection protocols violations
Packet types: 255, Currently violated: 1

Protocol    Packet      Bandwidth  Arrival   Peak      Policer bandwidth
group       type        (pps)      rate(pps) rate(pps) violation detected at
reject      aggregate   2000       5000      11682678  2025-04-10 08:22:39 PDT  <<< Bandwidth =
    rate-limit = 2k pps
           Detected on: FPC-5
```

So it means our RE will received a maximum of 2K rejected packets and will generate 2K ICMP unreachable packets in reply. Just check our port connected to the IP Fabric and oh ! Suprise 2Kpps in output. These are our ICMP unreachable sent out back to the Customer B.

```
regress@rtme-mx-62> monitor interface et-2/0/0.0
Interface: et-2/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:              117022326356 (23519664 bps)             [11197970]
  Output bytes:                178738122 (895984 bps)                 [426496]
  Input packets:               238821075 (6000 pps)                    [22853]
```

```
10    Output packets:               3191752 (2000 pps)                    [7616] <<< 2K ICMP
          Unreachable targeting Cust. B
```

How we can avoid that?

The easiest solution is to have in your FIB always a last resort route entry, that could be discard but why not a fallback path to route the next-ip address. In our case, if 192.168.1.1 deaseappers, we may want:

- to not reject/discard the traffic but move back to the PNI interface. For that we need to configure a static route pointing to PNI peer, with a higher preference as a backup path for 192.168.1.1. Let's do simply add this static route in our VRF and check just after the commit the statistics of our PNI interface to see if all our 6K pps (A+B traffic) are forwarded back:

```
1  edit private
2
3  set routing-instances VRF1 routing-options static route 192.168.1.1/32 next-hop 172.16.8.1
       preference 254
4
5  commit comment "ADD_BACKUP_NEXT_IP" and-quit
6
7  regress@rtme-mx-62> monitor interface et-2/0/0.0
8  Interface: et-2/0/0.0, Enabled, Link is Up
9
10 <- truncated output ->
11
12 Remote statistics:
13   Input bytes:                132708516 (0 bps)                        [0]
14   Output bytes:             50812435866 (23522568 bps)              [5629120]
15   Input packets:                270844 (0 pps)                         [0]
16   Output packets:             103698854 (6000 pps)                   [11488] <<< We backup B
         traffic to PNI
```

- or to silently discard the traffic. In this scenario we can create a static route with higher preference pointing to **discard** next-hop. In our case, I've just added a default discard route in the VRF. so, if 192.168.1.1/32 is not announce anymore, the lookup of the next-ip address will fall back to this default discard instead of matching the default reject. Let's remove the previous static route and add the new default one:

```
1  edit private
2
3  delete routing-instances VRF1 routing-options static route 192.168.1.1/32
4  set routing-instances VRF1 routing-options static route 0.0.0.0/0 discard
5
6  commit comment "ADD_DEFAULT" and-quit
```

So, with this last configuration, our Customer B traffic should be now silently discared and we shouldn't observe DDOS protocol violation and ICMP unreachable traffic:

```
1  regress@rtme-mx-62> monitor interface et-2/0/0.0
2  Interface: et-2/0/0.0, Enabled, Link is Up
3  Flags: SNMP-Traps 0x4000
4  Encapsulation: ENET2
5  Local statistics:                                              Current delta
6    Input bytes:                   216186                              [0]
7    Output bytes:                  505399                              [0]
8    Input packets:                   3530                              [0]
9    Output packets:                  6274                              [0]
10 Remote statistics:
11   Input bytes:              119682374166 (23516136 bps)             [5628630]
12   Output bytes:                230359258 (0 bps)                       [0]
13   Input packets:               244249744 (5999 pps)                  [11487]
14   Output packets:                4113558 (0 pps)                       [0] <<< No more ICMP
         Unreach.
15
16
17 regress@rtme-mx-62> show ddos-protection protocols violations
18 Packet types: 255, Currently violated: 0   <<< No more Violation
```

PFE analysis

Now, let's re-announce the **192.168.1.1/32** prefix and examine how the FBF filter is applied on the **PFE**. Begin by running the following command to access the **PFE CLI**:

```
regress@rtme-mx-62> start shell pfe network fpc5
```

Next, list all the filters available on the linecard:

```
root@rtme-mx-62-fpc5:pfe> show firewall
Name                                Index          Token            Status
FBF                                 1               2875            DMEM
<- truncated output ->
```

Now resolve the token index `2875` to display the filter's program:

```
root@rtme-mx-62-fpc5:pfe> show sandbox token 22571514

<- truncated output ->

Filter properties: None
Filter state = CONSISTENT
term PBR_CUSTOMER_B
term priority 0
    source-address
        172.222.0/24
        false branch to match action in rule OTHER

    then
        accept
        count CUSTOMERB
         Policy Route:
         Destination Prefix: 192.168.1.1
         Routing instance: VRF1
         Policy route action is valid: TRUE
term OTHER
term priority 0

    then
        accept
        count OTHER
Previous nodes count: 1
Next nodes count    : 0
Entries count       : 0
```

The above output shows the filter program optimized by the Firewall Filter compiler. To display the actual program pushed into hardware, use the following PFE commands:

```
root@rtme-mx-62-fpc5:pfe> show firewall instance
Name,Index              Instance Key                    InstanceToken           LinkCount
FBF,1                   no-next-filter-0                5424                    1
<- truncated output ->
```

Then, run this second command using the Token ID `5424`, retrieved from the previous step:

```
root@rtme-mx-62-fpc5:pfe> show sandbox token 5424

<- truncated output ->

JNH_FW_START:
        opcode = 0x0000000c
        anonymous_union_0 = 0x00008605
        filter_id = 0x00000000
        cntr_base = 0x00000001
        flt_base = 0x00000000
        base_ptr = 0x000255af

Filter is not interface specific 1

Current context : 0.

Pfe Inst:0 Hw Instance 1, type:1 op:2 ref 0
  Counter Base:- 0x5400f8
```

```
19    Policer Base:- 0
20    Number of counters : 2 (including PSAs)
21
22  term PBR_CUSTOMER_B (pfe-inst 0)
23       Start Addr    :- 0x8605
24       Stop Addr     :- 0x8607
25       Stop NH       :- 0x6808255ab012ad58
26           Decoding :- FW_STOP: pdesc:0x104ab56 desc:0x4ab56
27
28       Action Addr   :- 0x4ab56
29       Action NH     :- 0x812aed800010000
30
31  match type: prefix
32    loc: 0x8605 nh: 0x7e30004002800000
33    FW_4BMATCH: fwop:0 desc:0x8005 koffset:396 boffset:0 mask:0x149c00e8 data:0x7fb7
34
35  Inst: 0 Action-Type: 134
36          JNH       :- 0x2bfffffd00000300:   <<< count CUSTOMERB;
37                    CounterNH: Relative Base = 1, Offset = 0x0, nextNH = 0xffffff
38
39  Inst: 0 Action-Type: 0
40          JNH       :- 0x201282240000000c:   <<< next-ip 192.168.1.1/32 routing-instance VRF1;
41                    UcodeNH: Indirect Decode: Indirect, Next = 0x4a089, pnh_id = 0, ,
42
43  <- truncated output ->
```

Pick the JNH dword corresponding to the next-ip action `0x201282240000000c` (line 40), and decode it:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x201282240000000c inst 2
2  UcodeNH: Indirect Decode: Indirect, Next = 0x4a089, pnh_id = 0,
```

UcodeNH will run a micro-code sequence. Here, it is an indirection to a virtual address found in the **Next** field. To read the data at `0x4a089`, run:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh vread vaddr 0x4a089 NH inst 2
2  Addr:Nexthop 0x4a089 Paddr:0x104a089, Data = 0x0812659400040000
```

This returns two key values:

- **Paddr** (physical address): `0x104a089`
- **Data** read: `0x0812659400040000`

> Note: To read a physical address, use `show pread paddr xxx`.

Since the value is a JNH word, decode it again. This time it reveals a **CallNH**, a list of ordered next-hops (when **mode=0**):

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x0812659400040000 inst 2
2  CallNH:desc_ptr:0x49965, mode=0, count=0x5
3    0x049960  0 : 0x168c000000000000
4    0x049961  1 : 0x40101ffffff81510
5    0x049962  2 : 0x40181fffffe02030
6    0x049963  3 : 0x1810318200200008
7    0x049964  4 : 0x23fffffc00000001 <<< this one will be skipped - related to fabric encap.
```

Now decode each action (excluding the fifth one, which is outside the scope of this article):

- The first is a `ModifyNH`, which modifies local memory — in this case, resetting the encapsulation length:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x168c000000000000 inst 2
2  ModifyNH: Subcode=Misc(26)
3  (Reset encap len)
```

- The second and third entries are **BitOpNH** actions, performing operations on specific data:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x40101ffffff81510 inst 2
2  BitOpNH: opcode=0x00000008, data32=0, desc_ptr=0xffffff, key=0x4/0 PPPoE Session ID, op=0, data
       =49320, nbits=16
```

```
3
4  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x40181fffffe02030 inst 2
5  BitOpNH: opcode=0x00000008, data32=0, desc_ptr=0xffffff, key=0x6/0 Unknown, op=0, data=257,
       nbits=16
```

These actions extract the next-ip address from local memory in two steps:

- First, we fetch `49320` (0xC0A8 = 192.168)
- Then, we fetch `257` (0x0101 = 1.1)

Combined, we recover the next-ip address from our FBF filter: **192.168.1.1**. This is a "key" that will be used for further processing (route lookup).

- The fourth NH in the list `0x1810318200200008` is a **KTREE** structure used for route lookup:

> A **KTREE** is Juniper's implementation of a binary structure known as a **Patricia Tree**.

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x1810318200200008 inst 2
2  KtreeNH: skip=8, sw_token=0, arOffset=0, mode=1, descPtr=0xc60800, key=0x4/0 LookupKey
```

To dump the full KTREE, use:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x1810318200200008 inst 2 ktree yes dump yes
2  Route                                Depth   JNH
3  -----------------------------------  ------  ------------------
4  Default                                   0  0x0812ad6000000000
5  00000000/32                               1  0x0812a08400000000
6  080808/24                                 1  0x08129d0400000000
7  acde00/24                                 2  0x08129df400000000
8  ac6f00/24                                 2  0x08129df400000000
9  ac100800/31                               3  0x000000000004ab3c
10 ac100801/32                               4  0x0812ade000000000
11 ac100800/32                               4  0x0812ac9800010000
12 ac100004/31                               3  0x000000000004abc0
13 ac100005/32                               4  0x08129d4800000000
14 ac100004/32                               4  0x0812aec800010000
15 c0a80101/32                               1  0x08129a1800000000 <<< 192.168.1.1/32
16 e0000001/32                               1  0x0812a0c400000000
17 ffffffff/32                               1  0x0812a0e400000000
18
19 Routes found: 14, Bytes used: 6664
```

This KTREE acts as the FIB for our **VRF1** instance. For instance, traffic hitting the **192.168.1.1/32** prefix is redirected to the action identified by the JNH word `0x08129a1800000000`, pointing to a **CallNH**:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x08129a1800000000 inst 2
2  CallNH:desc_ptr:0x4a686, mode=0, count=0x1
3    0x04a685  0 : 0x20129af00000000c
```

This value (`0x20129af00000000c`) is another **UcodeNH** indirection:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x20129af00000000c inst 2
2  UcodeNH: Indirect Decode: Indirect, Next = 0x4a6bc, pnh_id = 0, ,
```

To follow the indirection, read the next-hop at `0x4a6bc`:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh vread vaddr 0x4a6bc NH inst 2
2  Addr:Nexthop 0x4a6bc Paddr:0x104a6bc, Data = 0x08129ad400000000
```

Once again, decode the JNH word retrieved from the virtual memory address — it points to another **CallNH** list:

```
1  root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x08129ad400000000 inst 2
2  CallNH:desc_ptr:0x4a6b5, mode=0, count=0x1
3    0x04a6b4  0 : 0x11c0000000026c14
```

Decode the final NH word `0x11c0000000026c14`:

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x11c0000000026c14 inst 2
ModifyNH: Subcode=SetNH-Token(7),Desc=0x0,Data=0x26c14,NextNH=0
 (pfeDest:20, TokenIdMode:0/  , VC memberId:0, isReass:0, token:0x26c/620)
```

We've reached the final forwarding action — setting the forwarding NH index via the **NH Token**. Here, the token `0x26c` corresponds to the NH ID from our route lookup.

To get more info on this next-hop:

```
root@rtme-mx-62-fpc5:pfe> show nh db index 0x26c
Index          Type        Func-Type    Proto        Nh-Flags        Ifl-Name        Ifl-Index
           Nh-Token        Nh-Prefix
620            Unicast     -            ipv4_tag     0x41            et-5/0/0.0      356
                5519           ac100101/32
```

Perfect — this confirms that the traffic is forwarded via the correct path: our core-facing interface `et-5/0/0.0` and doesn't follow anymore the "best path".

To go further (e.g., check Layer 2 headers or MPLS encapsulation), use:

```
root@rtme-mx-62-fpc5:pfe> show nh detail index 0x26c
Nexthop Info:

NH Index                      : 620
NH Type                       : Unicast
NH Proto                      : ipv4_tag
NH Flags                      : 0x41
IF Name                       : et-5/0/0.0
Prefix                        : ac100101/32
NH Token Id                   : 5519
NH Route Table Id             : 0
Sgid                          : 0

OIF Index                     : 356
Underlying IFL                : .local..0 (0)
Session Id                    : 788
Num Tags                      : 2
Label                         : 0x4e87eff0 (20103)lbFlags: 0
Label                         : 0x10fff000 (16)lbFlags: 0
MTU                           : 0
L2 Length                     : 12
L2 Data                       : 00:00:01:64:80:00:a8:d0:e5:ef:6a:87
Filter Index                  : 0

Platform Info
-------------
  FabricToken: 5527
  EgressToken: 5526
  IngressFeatures:

Container token: 5519
#5 SetNhToken tokens:
 Mask : 0x0
 [  SetNhToken:5518  ]

  EgressFeatures:

Container token: 5526
#2 PushLabels tokens:
 Mask : 0x0
 [  PushLabels:5522  Token-1:5520  Token-2:5521  ]
#4 StatsCounter tokens:
 Mask : 0x1
 [  StatsCounter:5523  ]
#11 UcastEncap tokens:
 Mask : 0x0
 [  UcastEncap:5524  ]
#12 SetOIF tokens:
 Mask : 0x0
 [  SetOIF:5525  ]
```

The figure below summarizes our packet walkthrough and highlights the main FBF steps:

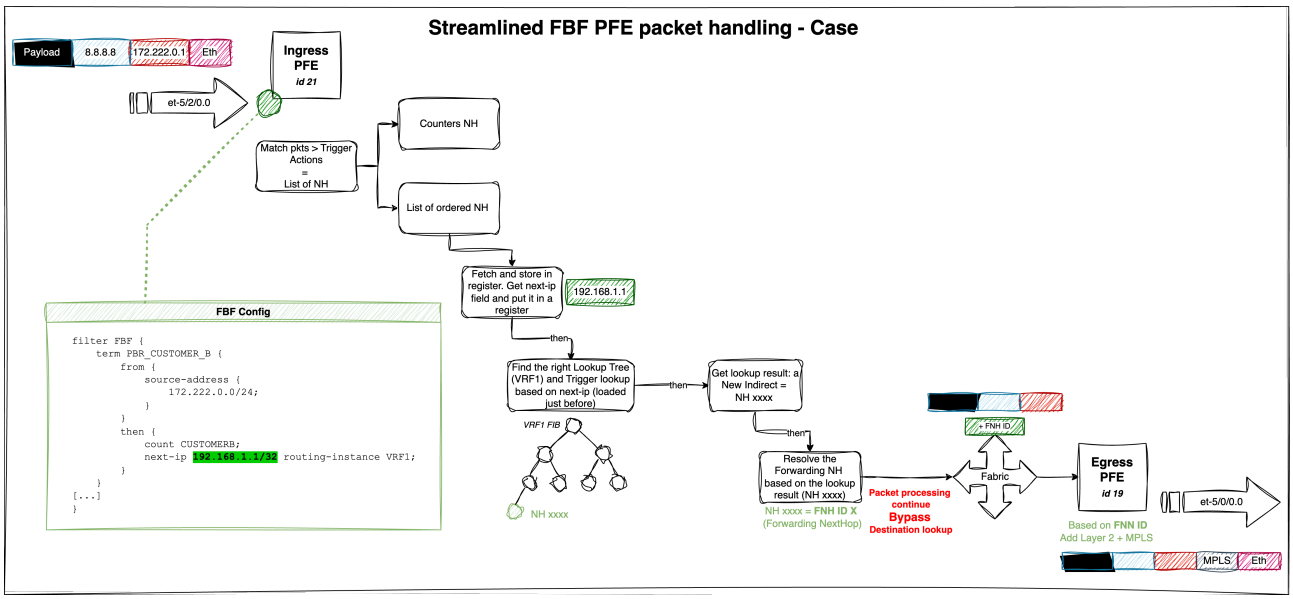

Figure 4: FBF next-ip at PFE level

And that wraps up part one. Take a break — in part two, we'll dive into configuring FBF thanks to the **rib-group** feature.

Case 2 - FBF Using forwarding instance

The second approach to achieving Filter-Based Forwarding (FBF) leverages the **forwarding-type routing instance**. In this method, the FBF filter term action redirects traffic to a specific routing instance of type `forwarding`.

This solution is considered the legacy approach and is widely supported on MX platforms. Compared to Case 1, it requires a deeper understanding of the `rib-group` concept. The following section will elucidate the necessary concepts to effectively implement FBF using this method.

rib-group Concept

A **RIB group** on Juniper devices enables routes learned in one routing table (the *source* or *initial* RIB) to be simultaneously installed into multiple routing tables (the *destination* RIBs). This feature is commonly utilized to share routes between routing instances (VRFs) or between the global routing table and a VRF. Policies can control which routes are imported, making RIB groups a flexible method for internal route redistribution without relying on BGP or other protocols.

Without a RIB group, each protocol — depending on the address family — feeds routes into a default routing table. In the context of RIB groups, this default table is referred to as the *source* or *initial* RIB. Similarly, protocols fetch routes (for a given family) from this default table when exporting routes.

The figure below illustrates the default routing behavior:

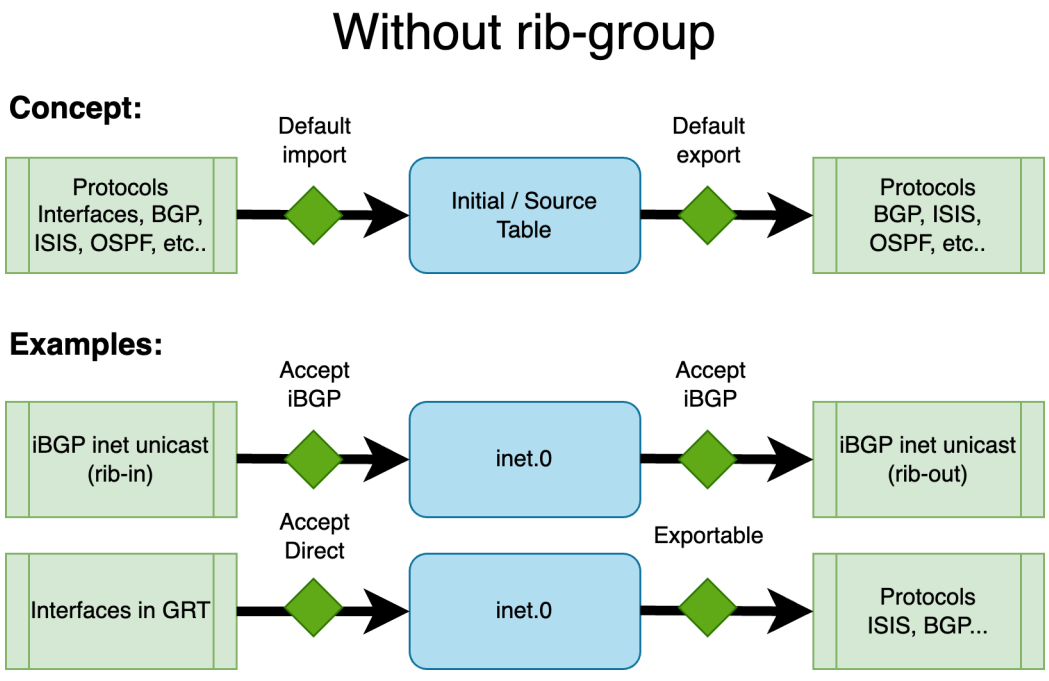

Figure 5: Default routing table management

As shown above, for BGP with the `inet` family (IPv4 unicast), the default routing table in the global context is `inet.0` — both for importing and exporting routes. Similarly, interface IPv4 addresses (direct routes) in the global context also reside in `inet.0`.

To leak routes from one table to another, the **RIB group** feature is employed, configured under **routing-options**. A RIB group is defined with the following parameters:

```
[edit routing-options]
rib-groups {
    <rib-group-name> {
        import-rib [ <source_table> <destination_table_1> <destination_table_2> ... ];
        import-policy <rib-group-import-policy>;
        export-policy <rib-group-export-policy>;
    }
}
```

> **Note:** import-policy and export-policy are optional, but at least one destination table must be specified in import-rib.

The **import-rib** statement is crucial. The order of tables matters: the first table is the *source* (or *initial*, *standard*, *contributing*) table. This is the default RIB associated with a given protocol/family combination. With a RIB group, routes are taken from this *source* table and replicated into one or more *destination* tables.

The RIB group establishes a link between the source RIB and the destination RIBs. The **import-policy** provides fine-grained control, allowing only specific routes or protocols to be leaked to the destination tables. Similarly, **export-policy** controls which routes from the destination tables are eligible for export by routing protocols.

The next figure illustrates the concept:

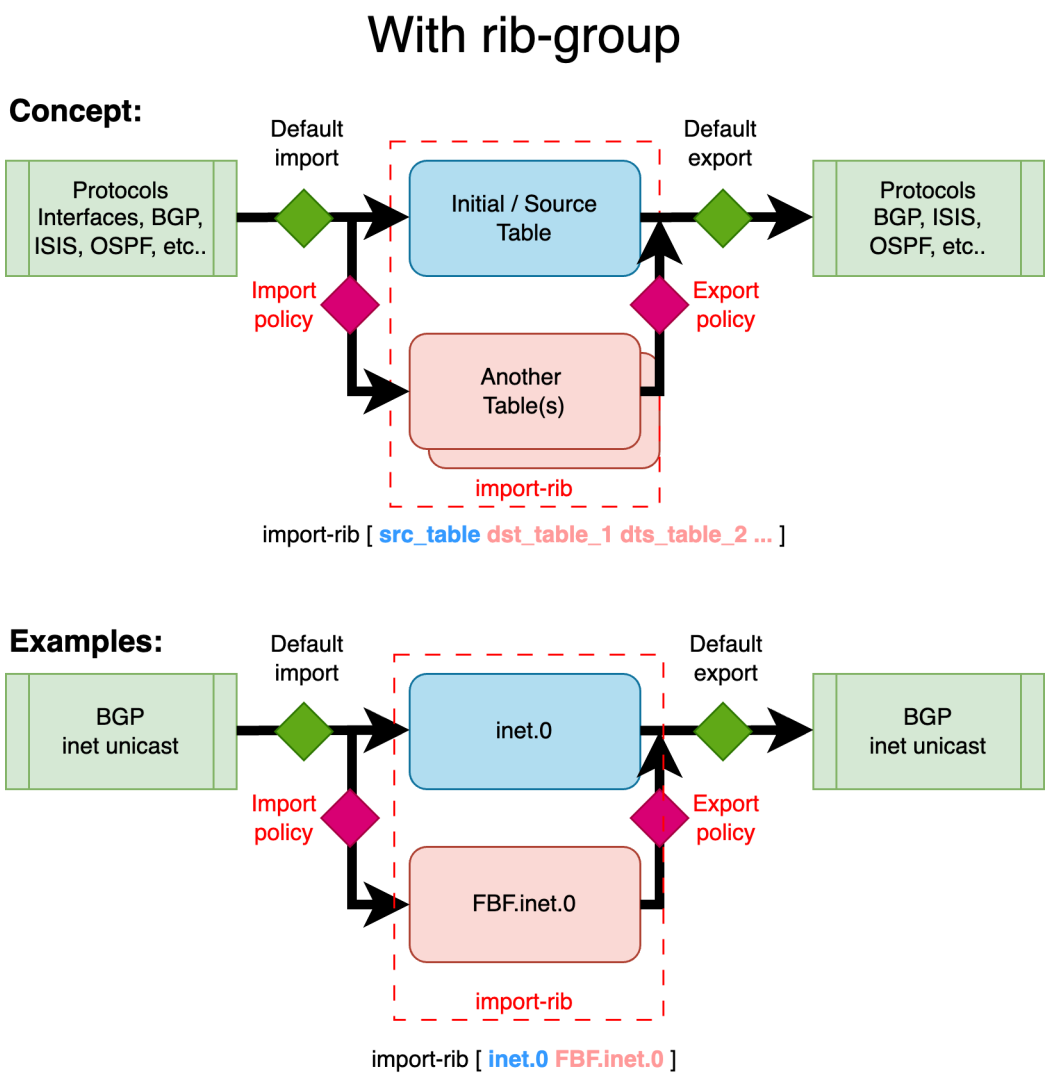

Figure 6: Routes leaking thanks to rib-group feature

This example demonstrates how BGP unicast routes from the global routing context are leaked into a new routing table (or instance) called **FBF**. While these routes remain in their default (source) table — `inet.0` — they are also copied into an additional table, in this case, the destination `FBF.inet.0`, using a RIB group.

In the context of Filter-Based Forwarding (FBF), this allows you to constrain routing decisions to a specific set of routes — not by using the standard FIB, but by relying on a custom FIB built from the custom destination RIB. This destination RIB can be selectively populated with only the routes you want to use for FBF-based traffic steering.

Let's illustrate this second FBF method with an example.

Example 2 - Topology

The Device Under Test (**DUT**) is an **MX480** equipped with an **MPC10E** line card.

This simplified setup represents a typical **DCI** router connected to an **IP Fabric**, providing access to remote resources via two distinct paths:

- A **quality** path through an **MPLS/SR** core network, and

- A **best-effort** path via a **direct peering or transit (PNI)** connection.

In this scenario, remote resources are reached via the direct PNI link connected in the Global Routing Table (GRT) and sent from the peer to our DUT via an eBGP session. The DUT also receives the same remote resources from a remote **ASBR** through an iBGP session. The remote ASBR sets the next-hop address of these routes with its Segment Routing node-SID (advertised in the ISIS SR domain). This allows for a BGP Free-core by tunneling traffic in a transport SR tunnel.

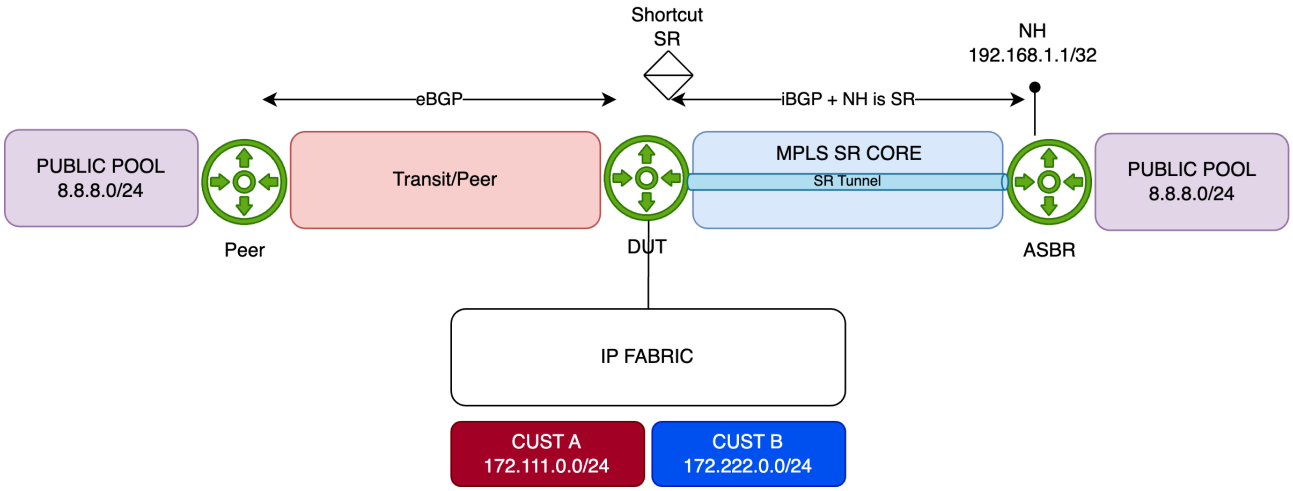

Figure 7: Network topology

For demonstration purposes, the remote resource is simulated using the public prefix **8.8.8.0/24**. This prefix is preferred by default via the direct PNI, with a backup path available through the MPLS/SR core (shortcut SR):

```
regress@rtme-mx-62> show route 8.8.8.0/24

inet.0: 43 destinations, 44 routes (43 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

8.8.8.0/24          *[BGP/170] 6d 17:31:10, localpref 10000
                       AS path: 1234 6000 I, validation-state: unverified
                     >  to 172.16.8.1 via et-2/0/0.0                      <<< PNI
                     [BGP/170] 00:00:11, localpref 50, from 193.252.102.2
                       AS path: I, validation-state: unverified
                     >  to 172.16.1.1 via et-5/0/0.0, Push 20002          <<< SR Tunnel
```

Initial Configuration

Below is the initial configuration for the DUT, kept simple for clarity:

- The DUT receives customer prefixes via **eBGP** from the peer group **FABRIC**.
- The DUT receives public prefixes from the PNI peer—this is the primary/best path. A higher local-preference is set using the `PREF` import policy.
- It also receives public prefixes from the peer ASBR via **iBGP**—this is a backup path—remotely reachable through an MPLS SR Tunnel.

```
regress@rtme-mx-62> show configuration
protocols {
    bgp {
        group FABRIC {
            type external;
            local-address 172.16.0.4;
            peer-as 5000;
            neighbor 172.16.0.5;
```

```
        }
        group PEER {
            type external;
            local-address 172.16.8.0;
            import PREF;
            family inet {
                unicast;
            }
            peer-as 1234;
            neighbor 172.16.8.1;
        }
        group ASBR {
            type internal;
            local-address 193.252.102.101;
            family inet {
                unicast;
            }
            neighbor 193.252.102.2;
        }
    }
}
```

Configuration of FBF

The next steps involve creating a new routing instance (type `forwarding`) and leaking both the interface routes (i.e., `direct` routes for resolving the Layer 2 header) and the remote resources (in our case, 8.8.8.0/24), but only those announced by the core network - not those learned via the PNI.

First, create the FBF routing instance:

```
edit
load merge terminal relative

routing-instances {
    FBF {
        instance-type forwarding;
    }
}

commit comment "add_fbf_ri"
```

Next, create a new `rib-group` called **RG_FBF**, establishing a relationship between the `inet.0` table and `FBF.inet.0`. In other words, a relation between the default IPv4 table and the IPv4 table of our newly created **FBF** routing instance.

```
edit
load merge terminal relative

routing-options {
    rib-groups {
        RG_FBF {
            import-rib [ inet.0 FBF.inet.0 ];
        }
    }
}

commit comment "add_rib-group"
```

Remember, the order inside the `import-rib` option is important. The first table is considered the source table, and the second one is the destination table. At this point, nothing is leaked between these two tables; this configuration merely establishes the relationship.

The first routes to leak are the direct interfaces attached to `inet.0`. This can be achieved with the following configuration under `routing-options`:

```
edit
load merge terminal relative

routing-options {
    interface-routes {
```

```
        rib-group inet RG_FBF;
    }
}

commit comment "import-direct"
```

Once committed, you should see `direct` and `local` routes in the `FBF.inet.0` table. Verify with:

```
regress@rtme-mx-62> show route table FBF.inet.0

FBF.inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.0.4/31      *[Direct/0] 6d 20:12:58
                    >  via et-5/2/0.0          <<< Fabric
172.16.0.4/32      *[Local/0] 00:00:59
                       Local via et-5/2/0.0
172.16.1.0/31      *[Direct/0] 6d 20:12:58
                    >  via et-5/0/0.0          <<< Core
172.16.1.0/32      *[Local/0] 00:00:59
                       Local via et-5/0/0.0
172.16.8.0/31      *[Direct/0] 6d 20:12:58
                    >  via et-2/0/0.0          <<< PNI
172.16.8.0/32      *[Local/0] 00:00:59
                       Local via et-2/0/0.0
193.252.102.101/32 *[Direct/0] 6d 20:12:58
                    >  via lo0.0
```

Everything looks good so far. The next step is to leak some BGP routes into the `FBF` routing table. But which ones?

Our goal is to redirect traffic entering this forwarding instance along a specific path — the one advertised by our remote ASBR. This route is available in the default `inet.0` table as a backup path. To achieve this, we'll configure BGP to use the `RG_FBF` rib-group. This rib-group allows routes normally imported into `inet.0` to be simultaneously leaked into `FBF.inet.0`.

Let's apply this on the `ASBR` peer-group:

```
edit
edit protocols
load merge terminal relative

bgp {
    group ASBR {
        type internal;
        local-address 193.252.102.101;
        family inet {
            unicast {
                rib-group RG_FBF;    <<< This tells BGP to use the rib-group and specifies where
                     to leak learned routes
            }
        }
        neighbor 193.252.102.2;
    }
}

commit comment "import_bgp"
```

With this configuration, all routes received from this peer-group will be leaked into `FBF.inet.0`. However, for demonstration purposes, we'll restrict the leaking to a specific BGP prefix: **8.8.8.0/24**.

To do that, we'll use the `import-policy` feature of the rib-group. First, we define a policy to authorize leaking from `inet.0` to `FBF.inet.0`, but only for:

- direct routes, and
- the BGP prefix 8.8.8.0/24

Here's the policy definition, followed by its application to the `RG_FBF` rib-group:

```
edit
load merge terminal relative

policy-options {
    policy-statement FBF_POLICY {
        term LEAK_DIRECT {
            from protocol direct;
            then accept;
        }
        term LEAK_BGP {
            from {
                protocol bgp;
                route-filter 8.8.8.0/24 orlonger;
            }
            then accept;
        }
        term REJECT_OTHER {
            then reject;
        }
    }
}

routing-options {
    rib-groups {
        RG_FBF {
            import-rib [ inet.0 FBF.inet.0 ];
            import-policy FBF_POLICY;         <<< controls which routes get leaked
        }
    }
}

commit comment "add_leaking_policy"
```

After committing, you can verify the result with:

```
regress@rtme-mx-62> show route table FBF.inet.0 protocol bgp

FBF.inet.0: 7 destinations, 7 routes (7 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

8.8.8.0/24          *[BGP/170] 01:16:31, localpref 50, from 193.252.102.2
                       AS path: I, validation-state: unverified
                     >  to 172.16.1.1 via et-5/0/0.0, Push 20002
```

Perfect — only the 8.8.8.0/24 prefix received from the internal ASBR is installed. The primary route via the PNI isn't present in this instance and remains solely in the `inet.0` table.

Now, to actually redirect the traffic, we'll use a simple firewall filter. It will match traffic sourced from Customer B and redirect it to the `FBF` routing instance. This filter is applied to the physical interface connected to the IP Fabric:

```
edit
load merge terminal relative

firewall {
    family inet {
        filter FBF_FWD {
            term CUSTOMER_B {
                from {
                    source-address {
                        172.222.0.0/24;
                    }
                }
                then {
                    count FBF;
                    routing-instance FBF;
                }
            }
            term other {
                then {
                    count OTHER;
                    accept;
                }
            }
        }
```

```
    }
}

interfaces {
    et-5/2/0 {
        mtu 9200;
        unit 0 {
            family inet {
                filter {
                    input FBF_FWD;
                }
                address 172.16.0.4/31;
            }
        }
    }
}

commit comment "apply_fbf"
```

At this point, traffic should be redirected as expected:

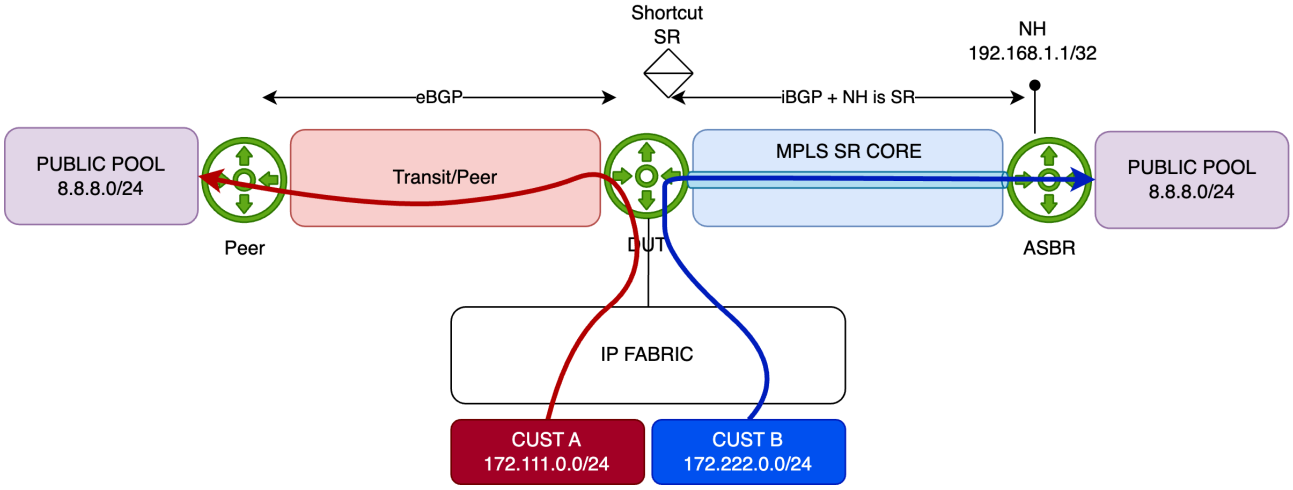

Figure 8: FBF in action

To help distinguish the flows, we've configured traffic rates as follows:

- **Customer A**: 1000 packets per second (pps)
- **Customer B**: 5000 packets per second (pps)

Let's monitor the PNI interface. As expected, only Customer A traffic goes through it (1000pps):

```
regress@rtme-mx-62> monitor interface et-2/0/0.0
Interface: et-2/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
  Input bytes:                132708516 (0 bps)                          [0]
  Output bytes:             49128737556 (3921056 bps)                    [0]
  Input packets:                270844 (0 pps)                           [0]
  Output packets:             100262735 (1000 pps)                       [0] <<< Only Customer
       A traffic
```

Now let's check the core-facing interface — we can see Customer B's traffic being redirected via FBF to the ASBR:

```
regress@rtme-mx-62> monitor interface et-5/0/0.0
Interface: et-5/0/0.0, Enabled, Link is Up

<- truncated output ->

Remote statistics:
```

```
  Input bytes:                 253750639 (584 bps)                      [0]
  Output bytes:              57241664165 (19600072 bps)                 [0]
  Input packets:                 533880 (1 pps)                         [0]
  Output packets:              116839924 (5000 pps)                     [0] <<< the tunneled
      Customer B traffic
```

Now, what if the iBGP session to the ASBR drops, or the 8.8.8.0/24 prefix is no longer announced?

In that case, the route will vanish from `FBF.inet.0`, and the default reject route will take over:

```
regress@rtme-mx-62> show route forwarding-table destination 0.0.0.0/0 table FBF
Routing table: FBF.inet
Internet:
Destination        Type RtRef Next hop           Type Index    NhRef Netif
default            perm     0                    rjct      520     1       <<< default reject
    route
```

Just like in **Case 1**, traffic to 8.8.8.0/24 will be rejected and punted to the routing engine (RE), which will respond with ICMP Unreachable messages (after being HW-policed to 2k pps).

To silently drop such packets instead of punting them, configure a static discard route as the default inside the FBF instance. But what if you want a fallback to the default routing table instead?

That's simple. If fallback forwarding is needed, just configure a default route inside the FBF instance pointing to the `inet.0` table. If 8.8.8.0/24 disappears, traffic is then forwarded via the primary PNI path through `inet.0`.

Here's the configuration:

```
edit
load merge terminal relative

routing-instances {
    FBF {
        instance-type forwarding;
        routing-options {
            static {
                route 0.0.0.0/0 next-table inet.0; <<< default fallback route
            }
        }
    }
}

commit comment "add_fallback_route"
```

PFE analysis

Assuming the FBF configuration is active, let's break down how the firewall filter behaves at the PFE level. Start by listing the filter instances:

```
regress@rtme-mx-62> start shell pfe network fpc5

root@rtme-mx-62-fpc5:pfe> show firewall instance
Name,Index              Instance Key                  InstanceToken           LinkCount
FBF_FWD,2                       no-next-filter-0              4777            1
<- truncated output ->
```

Next, pick up the **InstanceToken** and resolve it using the following command:

```
root@rtme-mx-62-fpc5:pfe> show sandbox token 4777

<- truncated output ->

Pfe Inst:0 Hw Instance 1, type:1 op:2 ref 0
  Counter Base:- 0x5400e8
  Policer Base:- 0
  Number of counters : 2 (including PSAs)
```

```
9
10 term CUSTOMER_B (pfe-inst 0)
11      Start Addr    :- 0x85fc
12      Stop Addr     :- 0x85fe
13      Stop NH       :- 0x680825491812a48c
14          Decoding :- FW_STOP: pdesc:0x104a923 desc:0x4a923
15
16      Action Addr   :- 0x4a923
17      Action NH     :- 0x812a81c00020000
18
19 match type: prefix
20   loc: 0x85fc nh: 0x7e3000401e000000
21   FW_4BMATCH: fwop:0 desc:0x803c koffset:396 boffset:0 mask:0x149c00e8 data:0x7fb7
22
23 Inst: 0 Action-Type: 134
24         JNH       :- 0x2bfffffd00000300:
25                   CounterNH: Relative Base = 1, Offset = 0x0, nextNH = 0xffffff
26
27 Inst: 0 Action-Type: 0
28         JNH       :- 0x23fffffc0000100c:
29                   UcodeNH: Indirect Decode: Indirect, Next = 0xffffff, pnh_id = 0, ,
30
31 Inst: 0 Action-Type: 0
32         JNH       :- 0x20129a4c0000000c:
33                   UcodeNH: Indirect Decode: Indirect, Next = 0x4a693, pnh_id = 0, ,
34
35 <- truncated output ->
```

Focus on the **Action NH** (line 17) and decode the **JNH** word. As shown, this represents a chain of Next-hops:

> The parameter `inst` below refers to the PFE id.

```
1 root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x812a81c00020000 inst 2
2 CallNH:desc_ptr:0x4aa07, mode=0, count=0x3
3   0x04aa04  0 : 0x2bfffffd00000300  <<< Action Counter
4   0x04aa05  1 : 0x23fffffc0000100c  <<< dummy action for clearing some internal states
5   0x04aa06  2 : 0x20129a4c0000000c  <<< Action routing-instance FBF
```

Now let's analyze the third action by decoding the JNH word `0x20129a4c0000000c` - **UcodeNH** will execute a micro-code sequence:

```
1 root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x20129a4c0000000c inst 2
2 UcodeNH: Indirect Decode: Indirect, Next = 0x4a693, pnh_id = 0, ,
```

This is an indirection to a virtual address found in the **Next** field. To read data at `0x4a693`, use:

```
1 root@rtme-mx-62-fpc5:pfe> show jnh vread vaddr 0x4a693 NH inst 2
2 Addr:Nexthop 0x4a693 Paddr:0x104a693, Data = 0x0812944c00020000
```

This yields two important values:

- **Paddr** (physical address): `0x104a693`
- **Data** read: `0x0812944c00020000`

> Note: to read a physical address, use `show pread paddr xxx`.

Since this is a JNH word, decode it again:

```
1 root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x0812944c00020000 inst 2
2 CallNH:desc_ptr:0x4a513, mode=0, count=0x3
3   0x04a510  0 : 0x168c000000000000
4   0x04a511  1 : 0x08129d1c00000000
5   0x04a512  2 : 0x23fffffc00000001  <<< this one will be skipped - related to fabric encap.
```

Now decode each action (except the third one - out of the scope of this article):

- First is a `ModifyNH` (we modify some bytes of the Local Memory - here we reset en encap len)

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x168c000000000000 inst 2
ModifyNH: Subcode=Misc(26)
(Reset encap len)
```

- Second points to another JNH word `0x1810318100200008` - This one is a **CallNH** which means execute a list of NH in order (when `mode=0`):

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x08129d1c00000000 inst 2
CallNH:desc_ptr:0x4a747, mode=0, count=0x1
  0x04a746  0 : 0x1810318100200008
```

Decode the first and single NH in the list, `0x1810318100200008` to reveal a **KTREE** structure used for route lookup:

> A **KTREE** is the Juniper implementation of a standard binary structure known as **Patricia Tree**

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x1810318100200008 inst 2
KtreeNH: skip=8, sw_token=0, arOffset=0, mode=1, descPtr=0xc60400, key=0x4/0 LookupKey
```

Use additional options to dump the full KTREE:

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x1810318100200008 inst 2 ktree yes dump yes
Route                                Depth   JNH
-----------------------------------  ------  ------------------
Default                                   0  0x08129df400000000
00000000/32                               1  0x08129d4000000000
080808/24                                 1  0x0812a04000000000  <<< 8.8.8.0/24 entry
<- truncated output ->
```

This KTREE acts as the FIB for our FBF routing-instance. For example, if traffic hits the 8.8.8.0/24 prefix, it's "redirected" to the action identifed by the JNH word `0x0812a04000000000` which refers to a list of NH (**CallNH**):

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x0812a04000000000 inst 2
CallNH:desc_ptr:0x4a810, mode=0, count=0x1
  0x04a80f  0 : 0x2012a0a40000000c
```

Whose the value `0x2012a0a40000000c` refers to indirection - **UcodeNH**:

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x2012a0a40000000c inst 2
UcodeNH: Indirect Decode: Indirect, Next = 0x4a829, pnh_id = 0, ,
```

To read this NH indirection at the virtual address `0x4a829`:

```
root@rtme-mx-62-fpc5:pfe> show jnh vread vaddr 0x4a829 NH inst 2
Addr:Nexthop 0x4a829 Paddr:0x104a829, Data = 0x0812a5fc00000000
```

One mmore time, we decode the JNH word retrieved from the previous virtual memory address - and we find out a new list of NH (**CallNH**):

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x0812a5fc00000000 inst 2
CallNH:desc_ptr:0x4a97f, mode=0, count=0x1
  0x04a97e  0 : 0x11c0000000038d14
```

And decode `0x11c0000000038d14`:

```
root@rtme-mx-62-fpc5:pfe> show jnh decode word 0x11c0000000038d14 inst 2
ModifyNH: Subcode=SetNH-Token(7),Desc=0x0,Data=0x38d14,NextNH=0
 (pfeDest:20, TokenIdMode:0/  , VC memberId:0, isReass:0, token:0x38d/909)
```

We've reached the final forwarding result — which will set the forwarding NH index into the **NH Token**. Here, the token `0x38d` corresponds to the NH ID of our lookup result.

You can get more info on this NH:

```
root@rtme-mx-62-fpc5:pfe> show nh db index 0x38d
Index          Type        Func-Type    Proto        Nh-Flags        Ifl-Name        Ifl-Index
          Nh-Token        Nh-Prefix
909            Unicast     -            ipv4_tag     0x1             et-5/0/0.0      356
                5187           ac100101/32
```

Perfect — this matches expectations. Traffic targeting 8.8.8.0/24 via the FBF routing instance will be forwarded to the core interface `et-5/0/0.0`.

You can dig deeper using this detailed view (to retrieve Layer 2 header information, MPLS encap…)

```
root@rtme-mx-62-fpc5:pfe> show nh detail index 0x38d
Nexthop Info:

NH Index                        : 909
NH Type                         : Unicast
NH Proto                        : ipv4_tag
NH Flags                        : 0x1
IF Name                         : et-5/0/0.0
Prefix                          : ac100101/32
NH Token Id                     : 5187
NH Route Table Id               : 0
Sgid                            : 0

OIF Index                       : 356
Underlying IFL                  : .local..0 (0)
Session Id                      : 788
Num Tags                        : 1
Label                           : 0x4e22fff0 (20002)lbFlags: 0
MTU                             : 0
L2 Length                       : 12
L2 Data                         : 00:00:01:64:80:00:a8:d0:e5:ef:6a:87
Filter Index                    : 0

Platform Info
-------------
  FabricToken: 5193
  EgressToken: 5192
  IngressFeatures:

Container token: 5187
#5 SetNhToken tokens:
 Mask : 0x0
 [  SetNhToken:5186  ]

  EgressFeatures:

Container token: 5192
#2 PushLabels tokens:
 Mask : 0x0
 [  PushLabels:5188  ]
#4 StatsCounter tokens:
 Mask : 0x1
 [  StatsCounter:5189  ]
#11 UcastEncap tokens:
 Mask : 0x0
 [  UcastEncap:5190  ]
#12 SetOIF tokens:
 Mask : 0x0
 [  SetOIF:5191  ]
```

If we repeat the exercise with the 8.8.8.0/24 prefix no longer present in the `FBF.inet.0` table, any traffic destined for 8.8.8.0/24 will match the default 0/0 route, which redirects to the `inet.0` table via a `next-table` action. In this scenario, we would notice slightly longer processing — just a few extra instructions — due to the need for two lookups:

- The first lookup occurs in the FBF FIB instance, which triggers the **next-table** action and redirects the processing to the main IPv4 `inet.0` table,
- The second lookup happens in the `inet.0` FIB, where the best route to 8.8.8.0/24 is selected—in our case, through interface et-2/0/0.0, which connects to our PNI.

> In this situation, two KTREE structures are involved: first the `FBF.inet.0` KTREE, followed by the `inet.0` KTREE.

and another table - just for illustration

Table 3: Interface Naming of the 8xSFP+ PIC

PFE COMPLEX	Port Number	10GE mode
EA ASIC 0	0	xe-0/1/0
	1	xe-0/1/1
	2	xe-0/1/2
	3	xe-0/1/3
	4	xe-0/1/4
	5	xe-0/1/5
	6	xe-0/1/6
	7	xe-0/1/7

The following figure provides a consolidated view of the packet traversal process and outlines the key steps of the FBF mechanism:

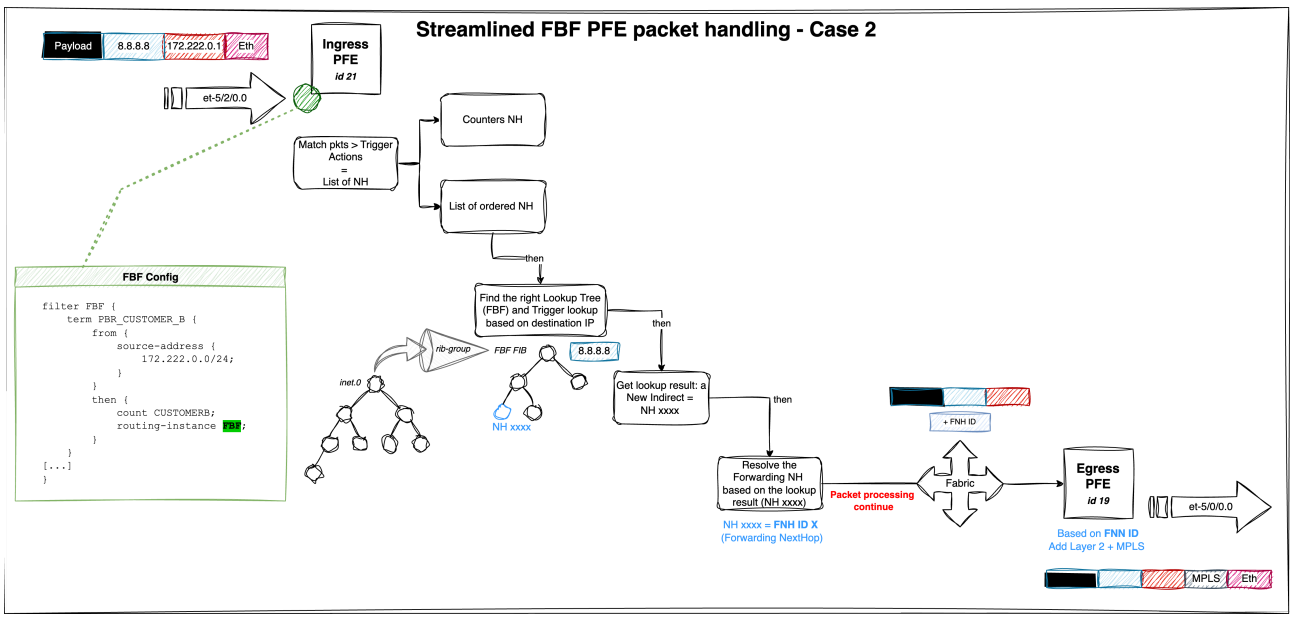

Figure 9: FBF with rib-group at PFE level

This is a test to refer to this section

Conclusion

In this article, we explored the mechanics of the **Filter-Based Forwarding (FBF)** feature, focusing on how traffic steering is implemented at the PFE level. We demonstrated **two distinct approaches** to configuring and validating FBF behavior. While both are effective, the second method provides **greater flexibility**, making it especially useful for advanced use cases.

All tests and examples provided were conducted within the **IPv4 address family**, but it's important to highlight that the same FBF logic and infrastructure are **fully supported for IPv6** as well, offering consistent behavior across protocol versions.

By understanding the inner workings of FBF down to the hardware abstraction layer, network engineers can confidently design and validate sophisticated traffic steering policies.