動いていると言えば動いている FRR の EVPN-Multihoming

さくらインターネット
合田 和也

2026/08/28 ENOG91 Meeting @ 柏崎

自己紹介

- 名前: 合田 和也 <Kazuya Goda>
- 経歴:
  - 2012/04 ~ 某 ISP にてソフトウェアルーターの開発
  - 2025/02 ~ さくらインターネットにてクラウドのネットワーク機能の開発
- ENOG歴:
  - ENOG90: FRR の Scripting は何ができるの?
    - 2026/06 以来 2.5ヶ月振りの登壇です
- 新潟県民歴:
  - 出生地は柏崎です🏖️
  - 小学5年生～高校卒業まで長岡に住んでました

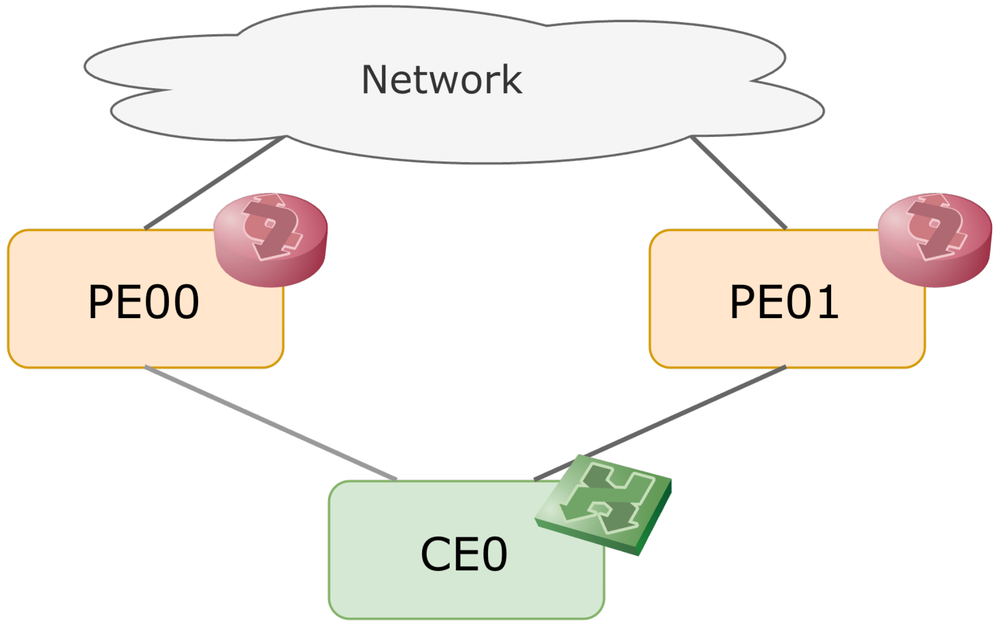
本日のお話

- EVPN には Multihoming を実現する機能がある
  - L2ブリッジングで Multihoming すると出てくる frame 重複や Loop 問題を解決している
- FRR に EVPN-MH が実装されているように見えるが:
  - コントロールプレーンはある程度必要な機能は実装されている
  - しかしデータプレーンの対応が足りてないように見える
- そこで不足機能を実装し「**動いていると言えば動いている**」状態へ
- EVPN-MH の概要や FRR のサポート状況も話すので共有できたら

EVPN Multihoming

まずは Multihoming とは

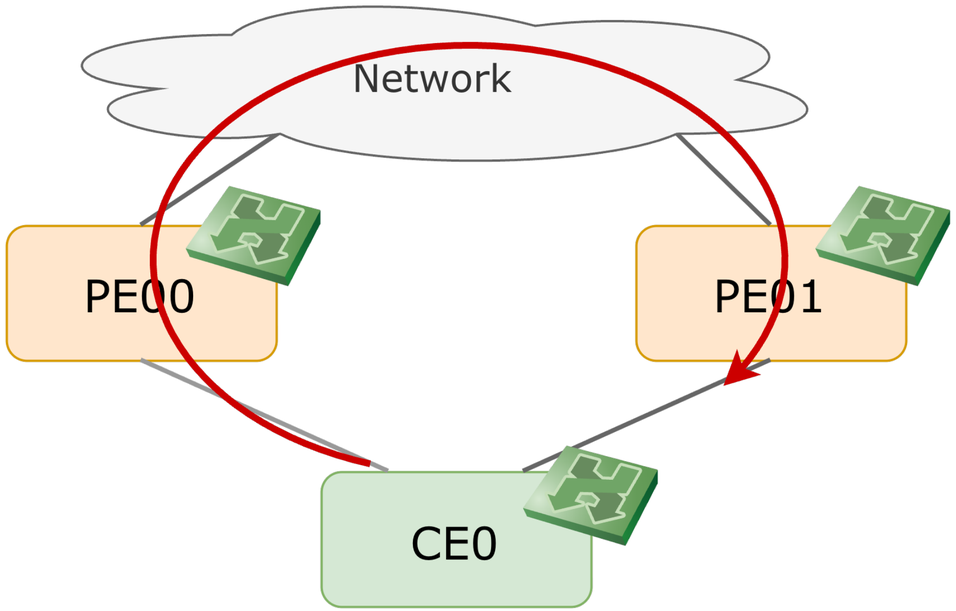
- 1 つのノードを複数の上位ノードに同時に接続する概念
  - 冗長性を確保するために使う
  - ノードはサーバーやネットワーク機器



L2ブリッジングでの Multihoming

- 課題
  - Ethernet フレームには TTL やホップという概念がないので重複、ループが発生する
    - 特に BUM フレームは flooding されるので指数関数的に増加する
  - 「同じ宛先への複数パス」という概念がない
    - L3 の ECMP のような概念がない

- 従来の解決策
  - ベンダー独自プロトコルの MLAG を利用する



EVPN Multihoming (EVPN-MH)

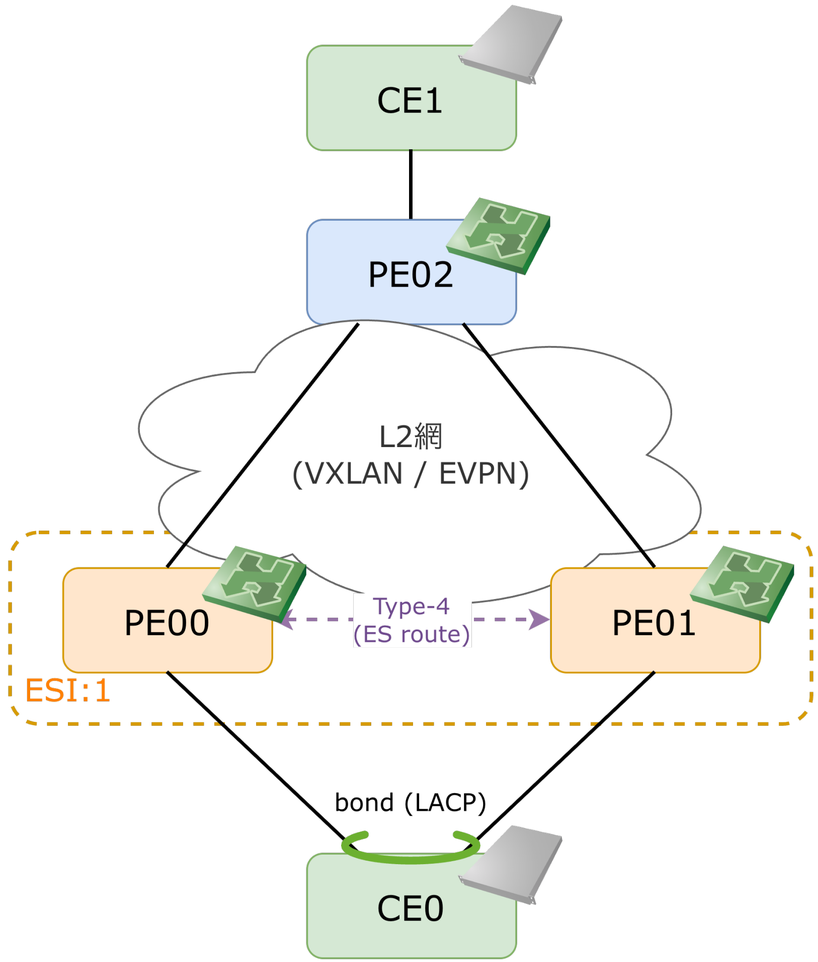
- EVPN は L2VPN のコントロールプレーンを BGP で実現する技術
  - 注) L3VPN もできるのがメリットだが、今回は L2VPN の話のみ
- EVPN-MHは「複数スイッチにまたがる冗長多重接続」を標準化
  - 従来 (今も) はベンダー独自(vPC/MLAG/MC-LAG)で実装されていた
  - EVPN-MH は上記を BGP を使った標準プロトコルとして仕様化

EVPN-MH を構成する 5 つの機能

- Ethernet Segment
- 指定フォワーダ (DF) と non-DF Filter
- Split Horizon Filter
- Aliasing
- Mass Withdraw

Ethernet Segment

- Ethernet Segment (ES) とは
  - 複数 PE にまたがる LAG 相当の概念
- ES は EVPN Type4 メッセージで広告
  - ES は 10 byte の ESI で識別する
  - 同じ ESI を広告した PE 同士が同一 ES となる

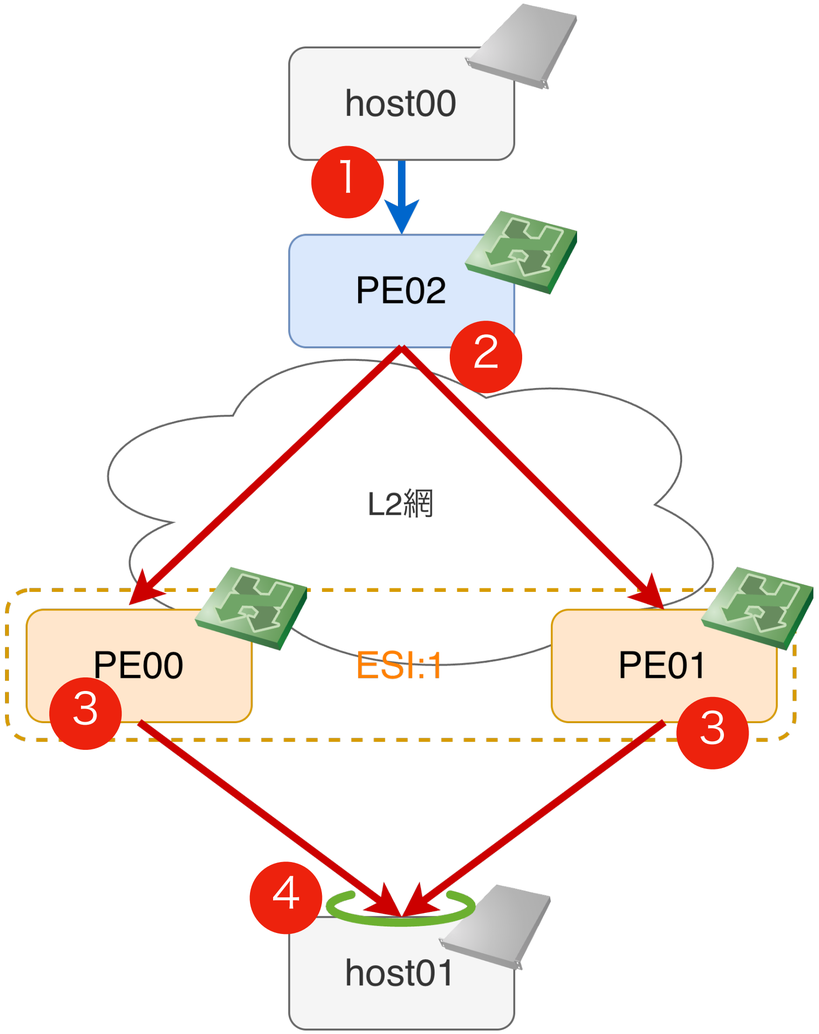


CE から見ると 1 本の LAG

DF と non-DF Filter

BUM フレームの重複を防ぐ仕組み

- BUM フレーム重複の流れ
  1. host00 が BUM を送信
  2. PE02 が PE00|01 に flooding
  3. PE00|01 は host01 に転送
  4. host01 は PE00|01 から重複したフレームを受信

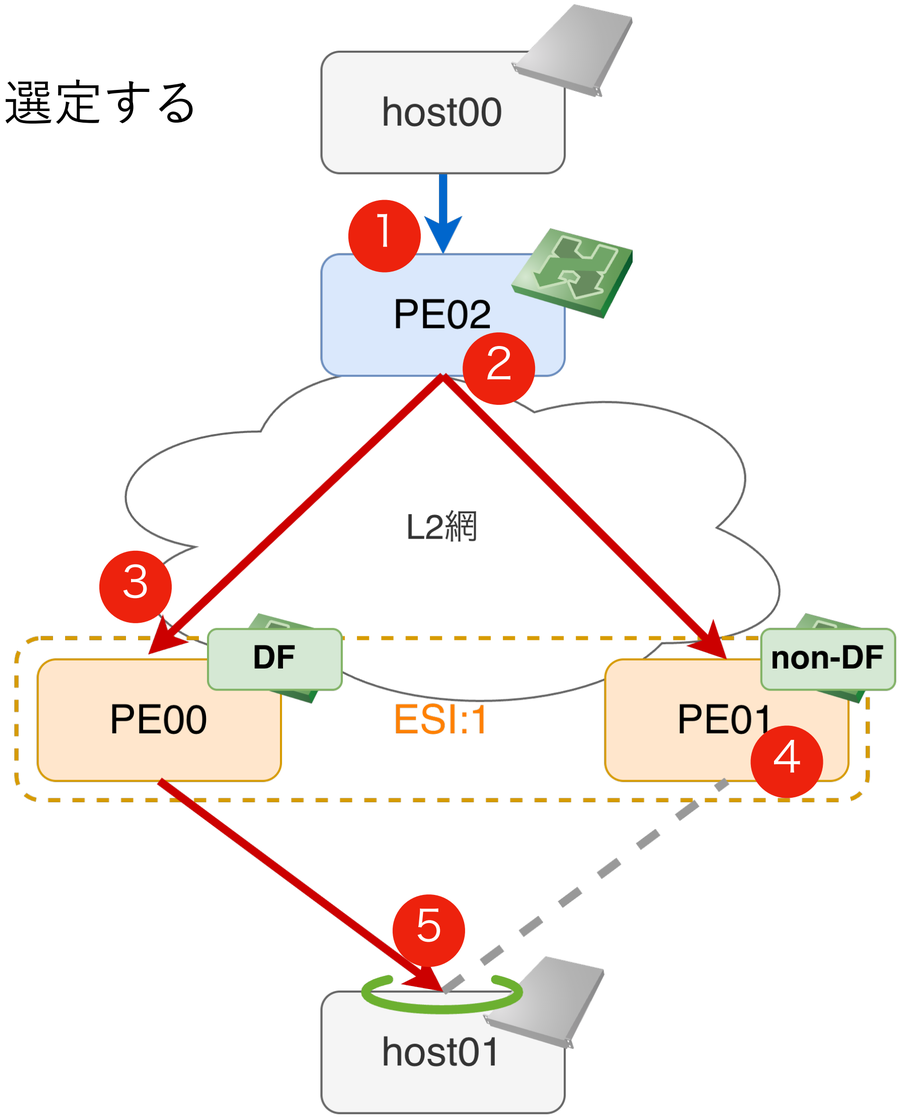


DF と non-DF Filter で解決

- ES ごとに BUM を送信するPE (DF:Designated Forwarder) を 1台選定する
  - EVPN Typ4 メッセージを利用
  - DF 選定のアルゴリズムはいくつかある

- non-DF Filter で重複を防ぐ
  - non-DF が BUM トラフィックを DROP する

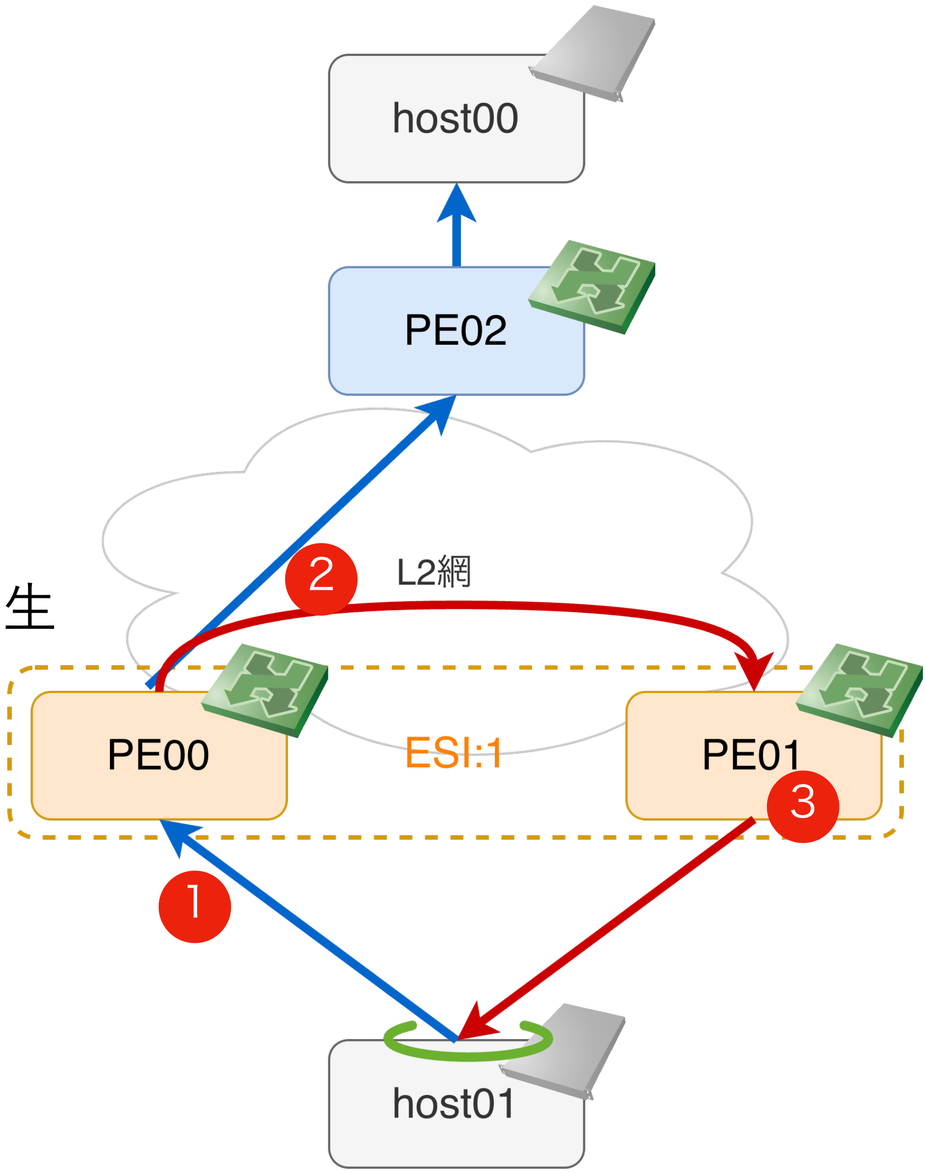
- non-DF Filter の挙動
  1. host00 が BUM を送信
  2. PE02 が PE00、PE01 に BUM を flooding
  3. DF の PE00 のみが BUM を host01 へ転送する
  4. non-DF の PE01 は BUM フレームは破棄する
     1. Unknown unicast は PE00|01 どちらも転送する (All-Act)
  5. host01 には PE00 からのみフレームが受信される



Split Horizon Filter (SPH Filter) で Loop を防ぐ

BUM の Loop の流れ

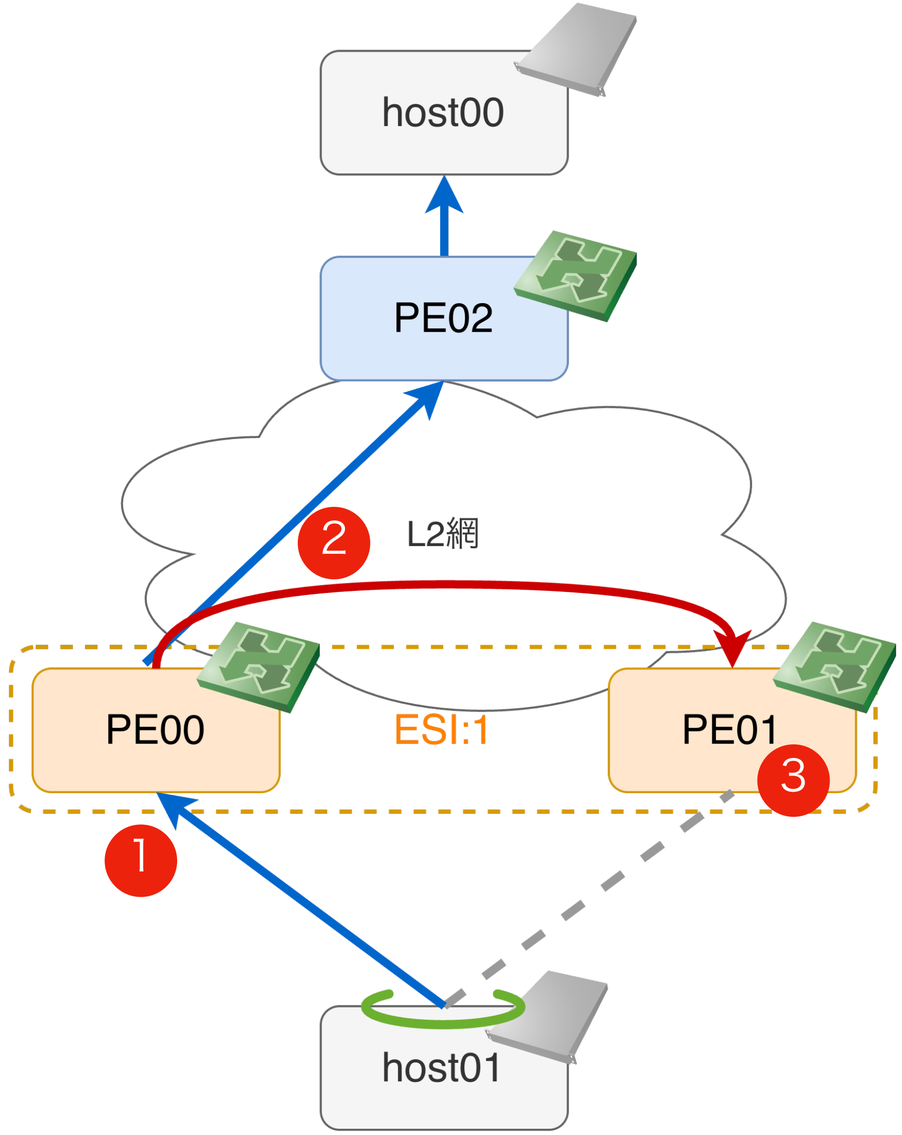
- BUM Loop の流れ
  1. host01 が BUM フレームを送信
  2. PE00 は Peer に BUM をflooding
     - 対向の PE02 だけでなくPE01 にも flooding
  3. PE01 は受信したフレームを host01 へ転送しLOOPが発生



Split Horizon Filter (SPH Filter) で解決

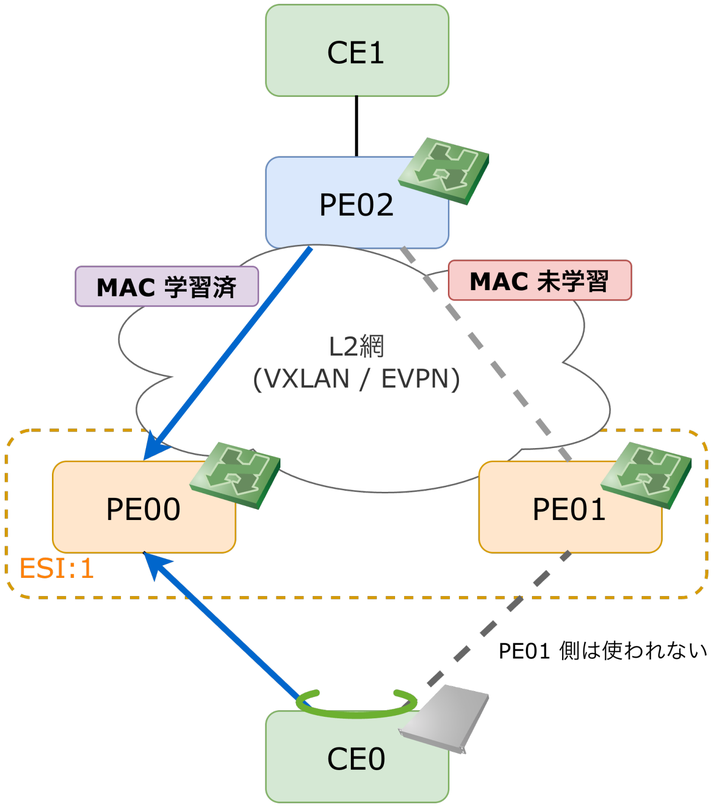
- 送信元が同一 ES VTEP の BUM は転送しない
  - PE は同一 ES の VTEP (IP) リストを持っている
    - 例) PE00 なら SPH Filter は PE01 の VETP IP
    - ES を見つける Type4 メッセージから作れる

- SPH-Filter での Loop 防止
  1. host01 が BUM フレームを送信
  2. PE00 は Peer に BUM をflooding
     - 対向の PE02 だけでなく同一 ES の PE01 にも flooding
  3. PE01 は ESI:1 の Peer から受信したフレームなので破棄し Loop を防ぐ

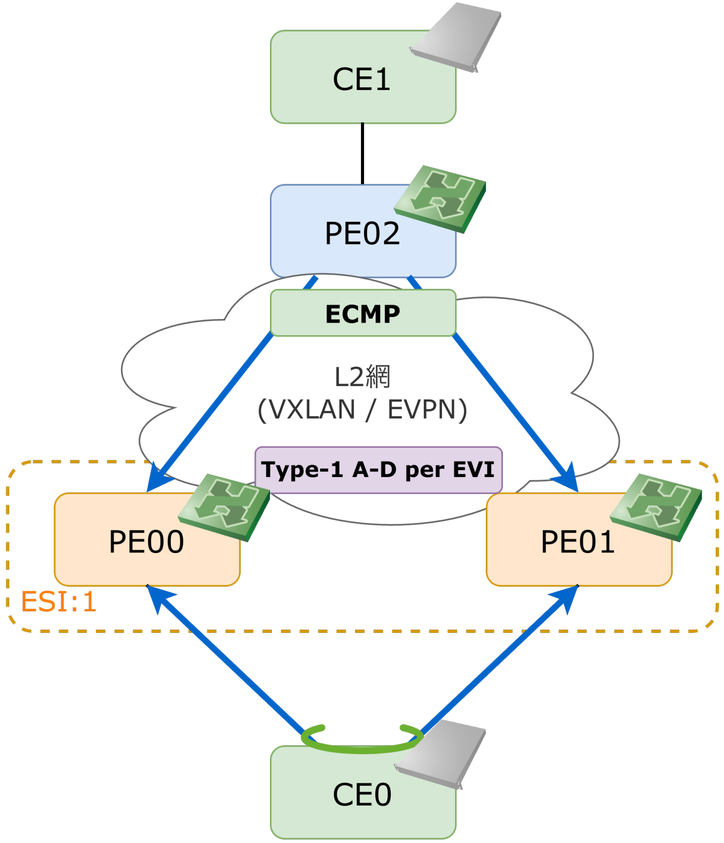


Aliasing - 経路を ES 単位のものとして扱う

- Aliasing がないと…
  - CE0 の MAC を PE00 が学習 & 広告
  - PE02 は CE0 宛を常に PE00 に送る
  - PE01 にも到達性があるが無駄となっている

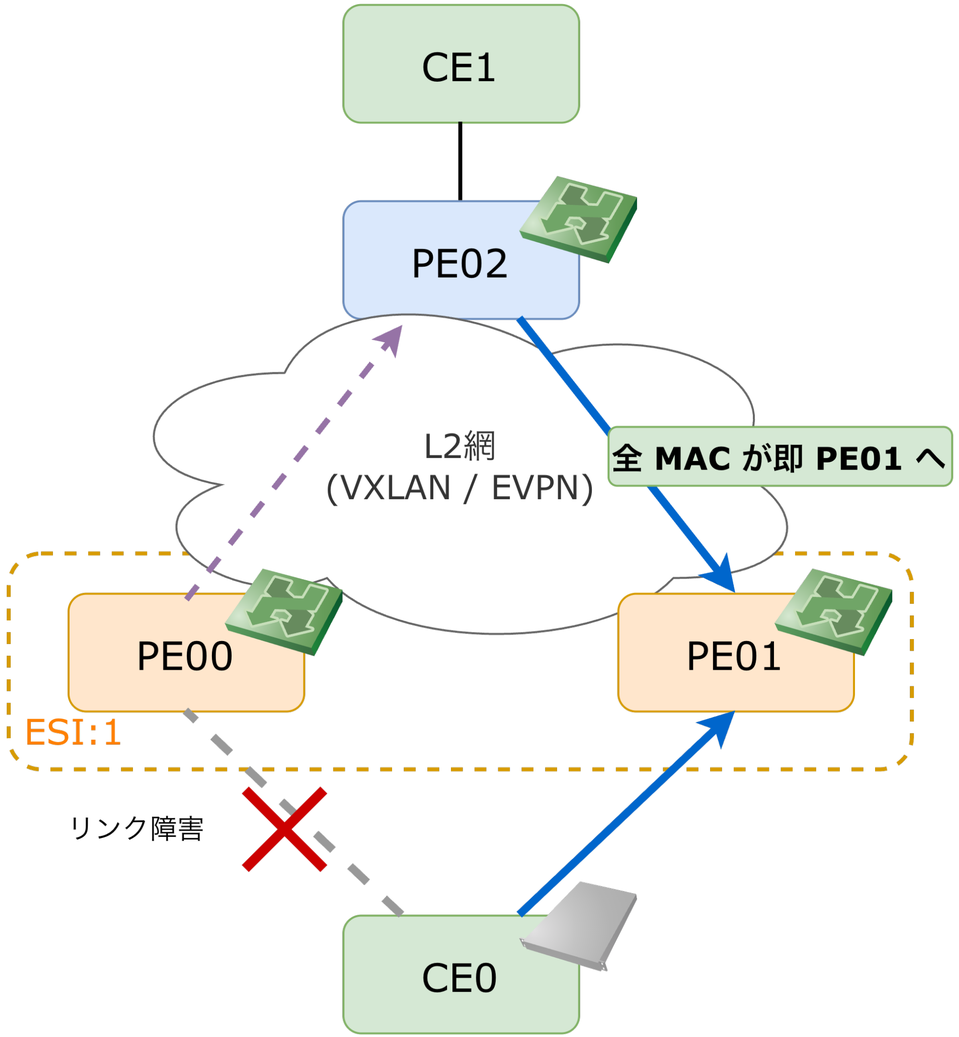


- Aliasing により
  - CE0 の MAC を PE00 が学習 & 広告
  - PE02 は ESI:1 (PE00 & PE01) の経路として受信
  - ESI:1 宛に ECMP ができるようになる



Mass Withdraw

- Mass Withdraw がないと…
  - PE00 は ES に紐づく経路を 1 つづつ Withdraw する
- Mass Withdraw により
  - PE00 は ES 単位で Withdraw できる
  - EVPN Type1 メッセージを利用



FRR での EVPN-MH 対応状況

EVPN-MH 主要機能サポート状態

経路関連	Aliasing	対応 (Linux kernel NHG 利用)
	Mass Withdraw	対応 (Linux kernel NHG 利用)
構成	All-Active	対応
	Single-Active	非対応 / Multihomed Network (MHD) 非対応
重複・ループ制御	DF Election	RFC 8584 preference-based のみ対応 RFC 7432 (EVPN) のデフォルト mould 方式は非対応
	non-DF Filter	コントロールプレーンのみ対応
	SPH Filter	コントロールプレーンのみ対応

EVPN-MH 主要機能サポート状態

経路関連	Aliasing	対応 (Linux kernel NHG 利用)
	Mass Withdraw	対応 (Linux kernel NHG 利用)
構成	All-Active	対応
	Single-Active	非対応 / Multihomed Network (MHD) 非対応
重複・ループ制御	DF Election	RFC 8584 preference-based のみ対応 RFC 7432 (EVPN) のデフォルト mould 方式は非対応
	non-DF Filter	**コントロールプレーンのみ対応**
	SPH Filter	**コントロールプレーンのみ対応**

non-DF Filter & SPH Filter が機能していない

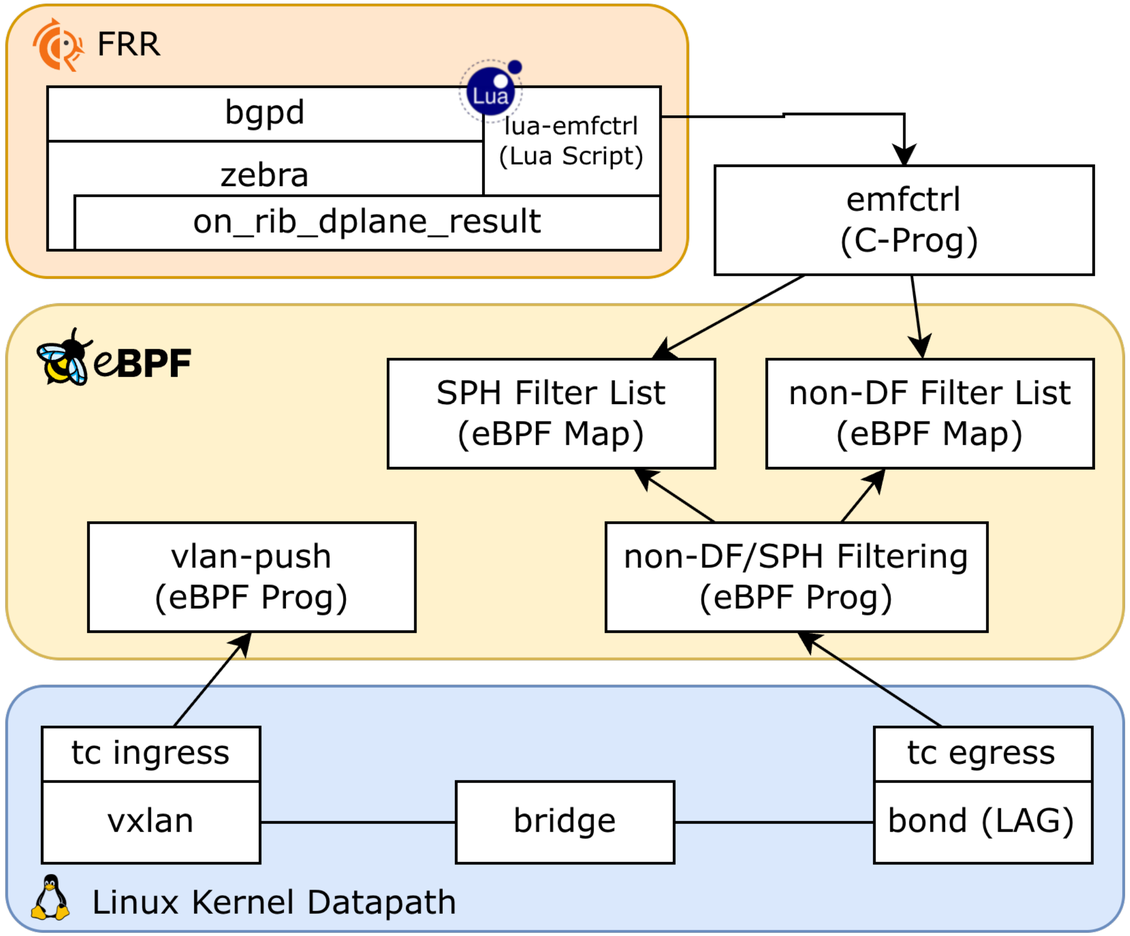
- コントロールプレーンは対応している
- データプレーンには何もしていない
  - BUM フレームの重複・Loop が発生する
  - non-DF/SPH Filter を実現する機能が Linux に存在しない
- 「動いている」とは言えないのでは!?
  - https://github.com/FRRouting/frr/issues/15400

non-DF/SPH Filter を実装する

https://github.com/gokzy/frr-evpn-mh-filter

実装方針

- FRR や Linux kernel へ変更なしで実装
  - 各種フックでコード変更なしに動作を変える
  - 必要十分なフックがある
- FRR
  - on_rib_process_dplane_results で D-Plane の変更をフック
  - フィルタルールの生成に利用
- Linux Kernel
  - tc + eBPF でパケット処理をフック
  - フィルタリングに利用



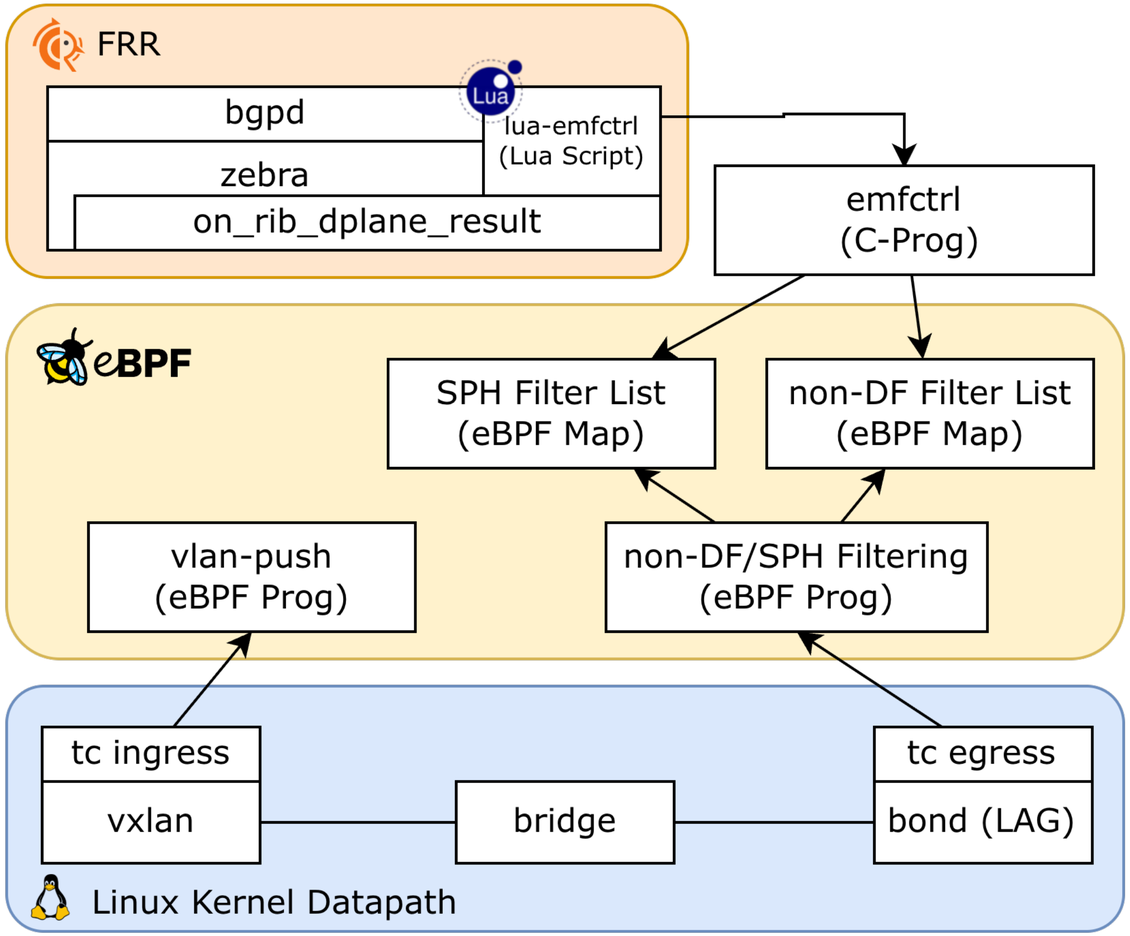
アーキテクチャとコンポーネント概要

- C/D-Plane 間のデータ共有に eBPF Map を利用
  - userland/kernel 両方から利用

- C-Plane は EVPN 経由の情報を emfctrl で登録
  - non-DF Filter : Egress Port (ES) が DF かを登録
  - SPH Filter: 同一 ES の Remote-VTEP を登録

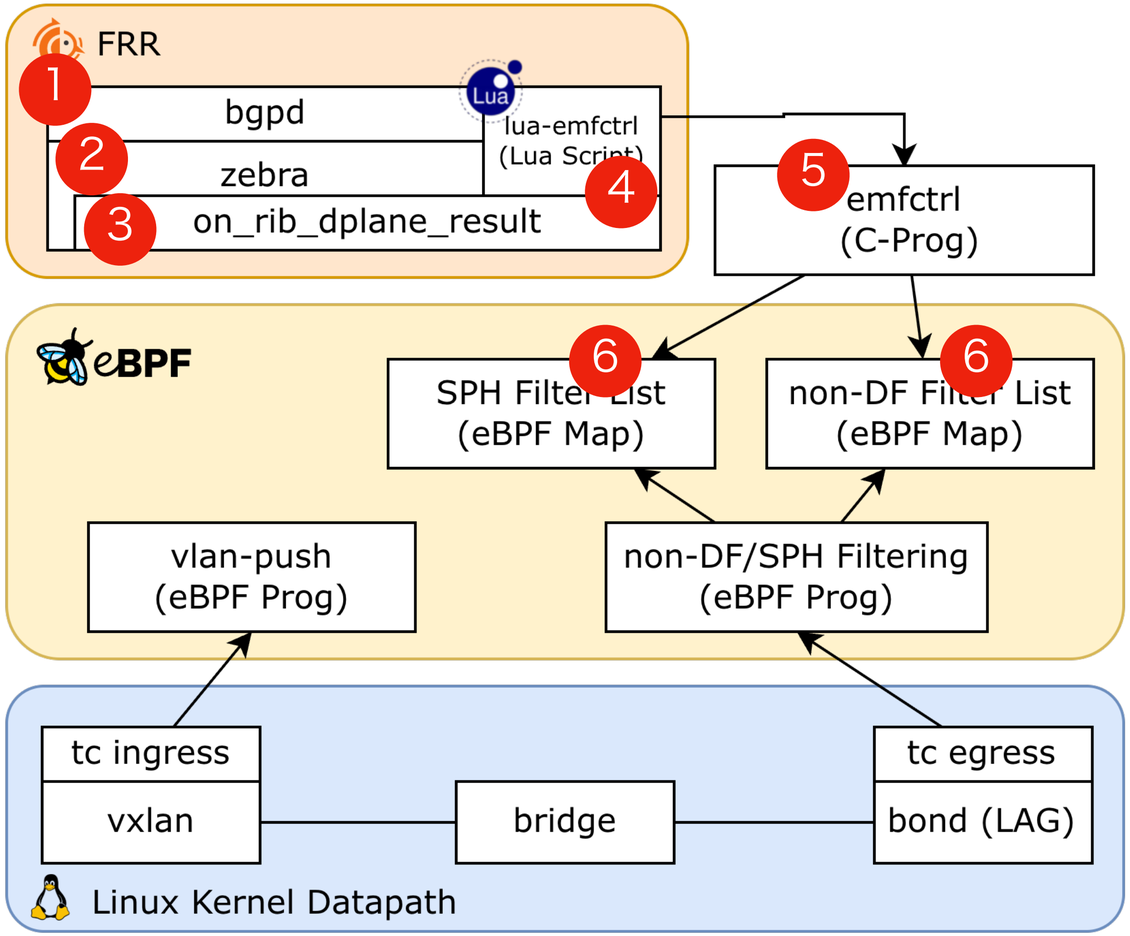
- D-Plane は eBPF Prog で BUM 転送を判断
  - non-DF Filter: Egress Port (ES) が DF かを引く
  - SPH Filter: 送信元 VTEP が同一 ES かを引く

- D-Plane は vlan tag を push する処理もある
  - 後述する SPH Filter 向の処理



コントロールプレーンの処理の流れ

1. EVPN Type-4 から DF、SPH List が決まる
2. 結果が bgpd から zebra へ送られる
3. zebra が on_rib_process_dplane_results hook を実行
   - zebra の dplane の変更を hook できる
4. Lua Script (lua-emfctrl) が実行される
5. lua-emfctrl が os.exec() で emfctrl を実行する
   - e.g) emfctrl df set bond22022 0
   - e.g) emfctrl sph replace bond22022 10.0.0.1
6. 対象の Filter の eBPF Map が更新される



zebra on_rib_process_dplane_results hook とは

- zebra の D-Plane 処理結果に Lua Script で Hook できる仕組み
  - zebra の D-Plane は Linux kernel 以外も扱うための抽象化レイヤー

- 今回は **DPLANE_OP_BR_PORT_UPDATE** を Hook する
  - 他にも MAC 経路を kernel に追加/削除した時に呼ばれる DPLANE_OP_MAC_* などがある
    - https://docs.frrouting.org/en/latest/zebra.html#const-struct-zebra-dplane-ctx

- **DPLANE_OP_BR_PORT_UPDATE** では Lua Script に次の値が渡される
  - `integer sph_filter_cnt`
  - `integer flags`
  - `integer backup_nhg_id`

zebra on_rib_process_dplane_results hook とは

- zebra の D-Plane 処理結果に Lua Script で Hook できる仕組み
  - zebra の D-Plane は Linux kernel 以外も扱うための抽象化レイヤー

- 今回は **DPLANE_OP_BR_PORT_UPDATE** を Hook する
  - 他にも MAC 経路を kernel に追加/削除した時に呼ばれる DPLANE_OP_MAC_* などがある
    - http

sph_filter_cnt でフィルタ数はわかるけど、実態の sph_filters がない!？

- **DPLANE_OP_BR_PORT_UPDATE** では Lua Script に次の値が渡される
  - `integer sph_filter_cnt`
  - `integer flags`
  - `integer backup_nhg_id`

FRR をちょっとだけ修正する

https://github.com/gokzy/frr/commit/dd149267dd829edf6eca40903ed38d06cfdadfde

- 実装方針で FRR のコード変更をせずに実現すると言ったが…
  - SPH Filters (同一 ES の VTEP-IP リスト) が取れないのは困る
    - Lua Script から os.exec で vtysh を実行して取得する手もあるが何か間違っている
- zebra は SPH Filters を持っいて Lua Script に渡していないだけ
  - コード的には何の障害もなさそう
- 実装漏れ or 実装当時に IP アドレスの配列を渡すのが手間だった可能性
  - これは本来 on_rib_process_results に実装されていると考えるべきものであり、であれば実質 FRR のコード変更をせずに実現してると言えるので、実装方針通りであると言っても過言ではない

non-DF/SPH Filter List の eBPF Map 構造

- non-DF Filter List
  - ES と対応する bond-if の ifindex をキーとして non_df か決まる

ifindex (key, u32)	non_df (value, u8)
12	1
15	0

- SPH Filter List
  - ES と対応する bond-if の (ifindex, VTEP-IP) をキーとして Filter するか決まる
  - 実際は atomic replace をするため map-in-map 構成となっている

key			value
ifindex	af	VTEP-IP	
12	AF_INET	10.0.0.2	1
12	AF_INET	10.0.0.5	1
15	AF_INET6	2001:db8::2	1

lua-emfctrl のコード

```
function on_rib_process_dplane_results(ctx)
    -- 29 means DPLANE_OP_BR_PORT_UPDATE
    if ctx.zd_op != 29 then
        return {}
    end
    -- non-DF filter
    if (ctx.br_port.flags & 1) == 1 then
        os.execute("/opt/ebpf/emfctrl df set " .. ctx.zd_ifname .. " 1")
    else
        os.execute("/opt/ebpf/emfctrl df set " .. ctx.zd_ifname .. " 0")
    end
    -- sph filter
    local vteps = {}
    for i, v in ipairs(ctx.br_port.sph_filters) do
        vteps[i] = tostring(v.string)
    end
    local vteps_str = table.concat(vteps, " ")

    os.execute("/opt/ebpf/emfctrl sph replace " .. ctx.zd_ifname .. " " .. vteps_str)

    return {}
end
```

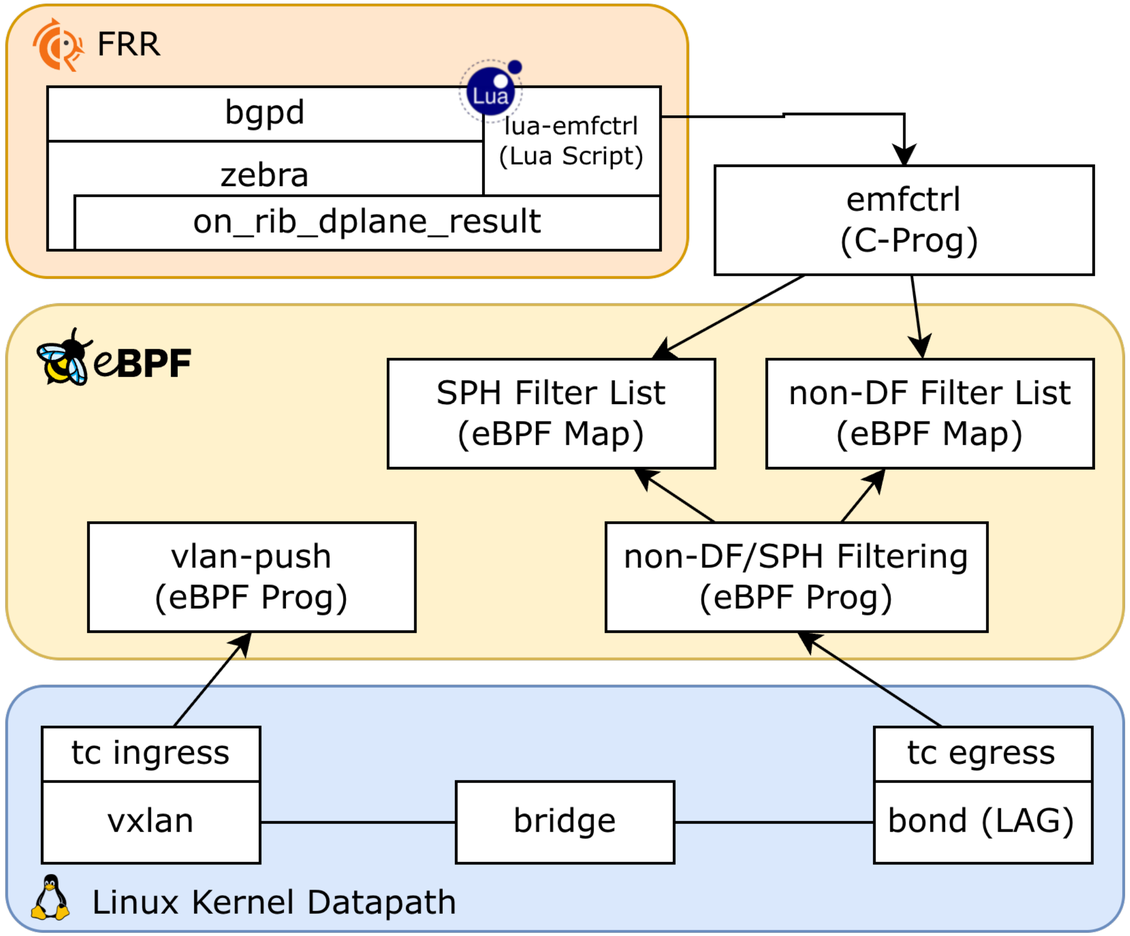
データプレーン

- C/D-Plane 間のデータ共有に eBPF Map を利用
  - userland/kernel 両方から利用

- C-Plane は EVPN 経由の情報を emfctrl で登録
  - non-DF Filter : Egress Port (ES) が DF かを登録
  - SPH Filter: 同一 ES の Remote-VTEP を登録

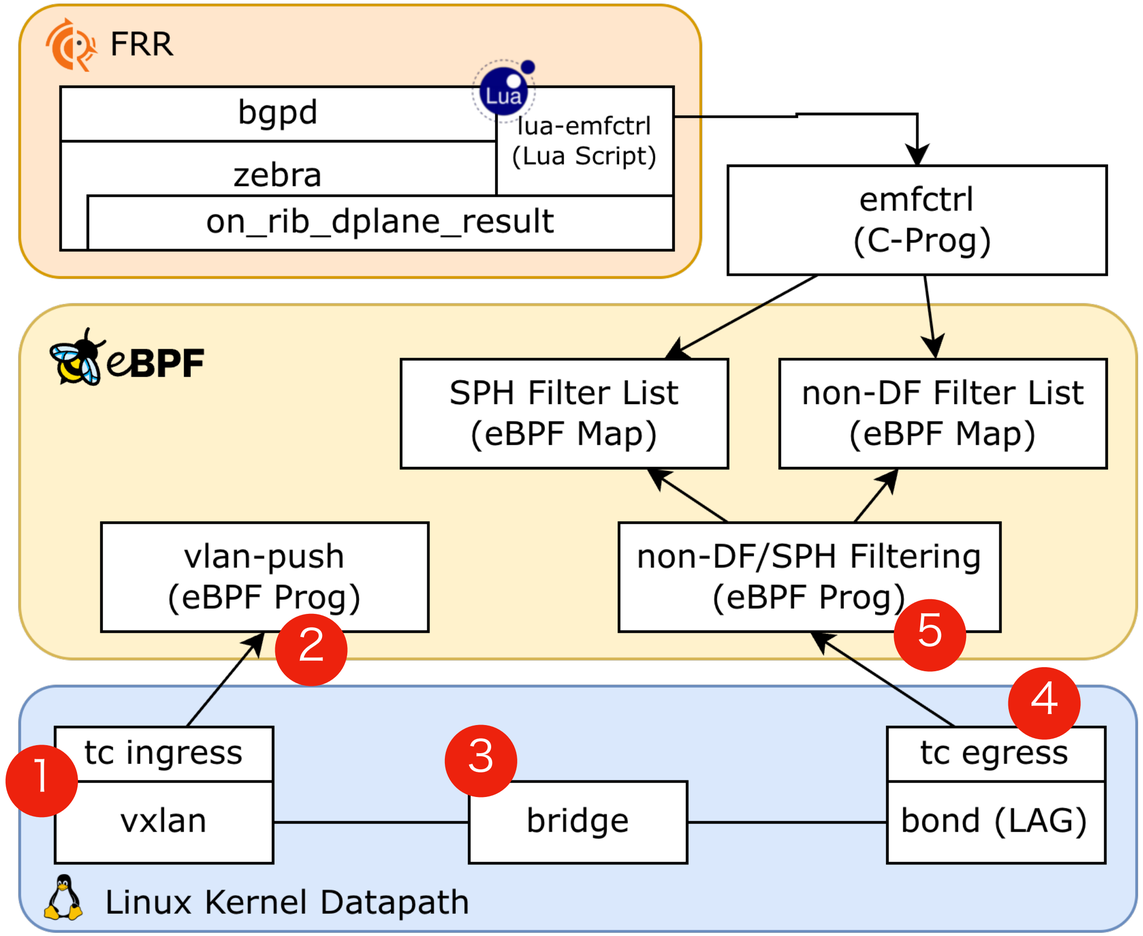
- D-Plane は eBPF Prog で BUM 転送を判断
  - non-DF Filter: Egress Port (ES) が DF かを引く
  - SPH Filter: 送信元 VTEP が同一 ES かを引く

- D-Plane は vlan tag を push する処理もある
  - 後述する SPH Filter 向の処理



データプレーンの処理概要

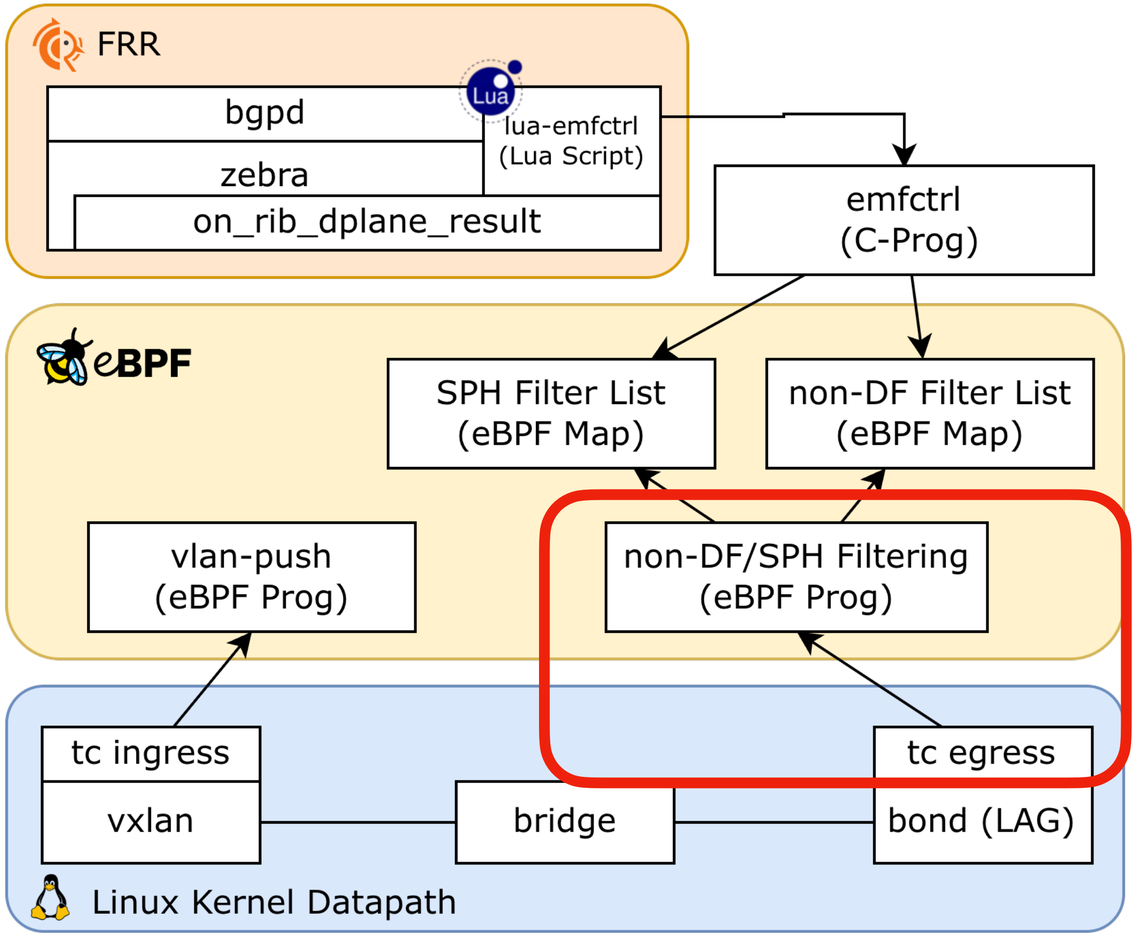
1. vxlan-if が VXLAN パケットを受信
2. vxlan-if の ingress で vlan tag を push
   - 事前に設定する vni vid mapping から決まる
   - tc で vlan-push eBPF Prog を実行
3. bridge で egress port が bond-if に決まる
4. bond-if の egress で BUM フレーム判定をする
   - tc flower により判定 Known unicast は pass
5. bond-if の egress で BUM に Filter を適用
   1. tc で non-DF/SPH Filtering eBPF Prog を実行
   2. eBPF Map から情報取得



大きく 2 つの課題がある

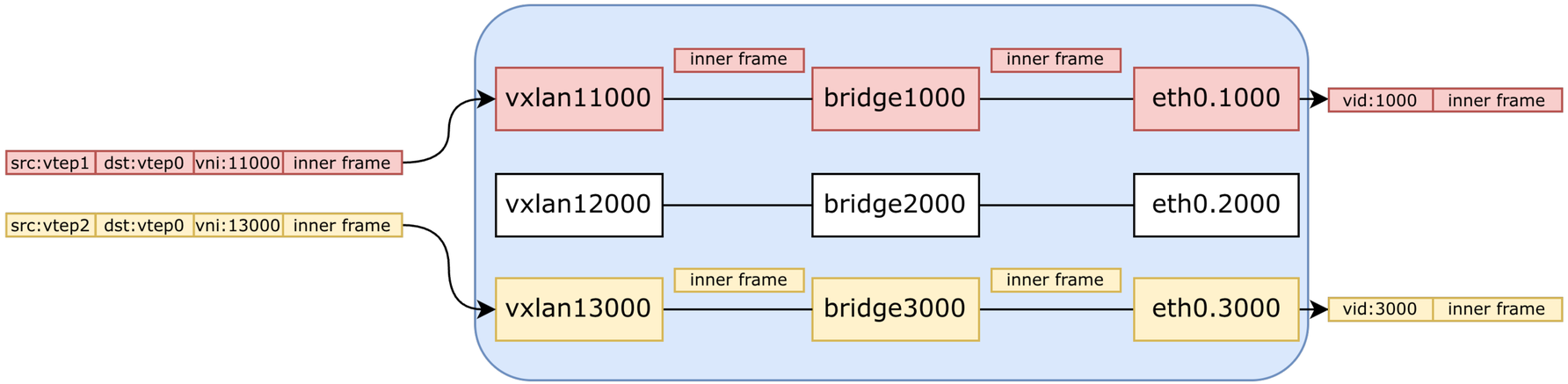
1. どうやって SPH Filter に 送信元 VTEP を伝えるか
   - SPH Filter がかかるのは bond-if egress
   - vxlan は decap 済みなのでパケットからは読めない

2. どうやって Unknown Unicast 判定をするか
   - non-DF Filter がかかるのは bond-if egress
   - Unknown かどうかは bridge が判断すること

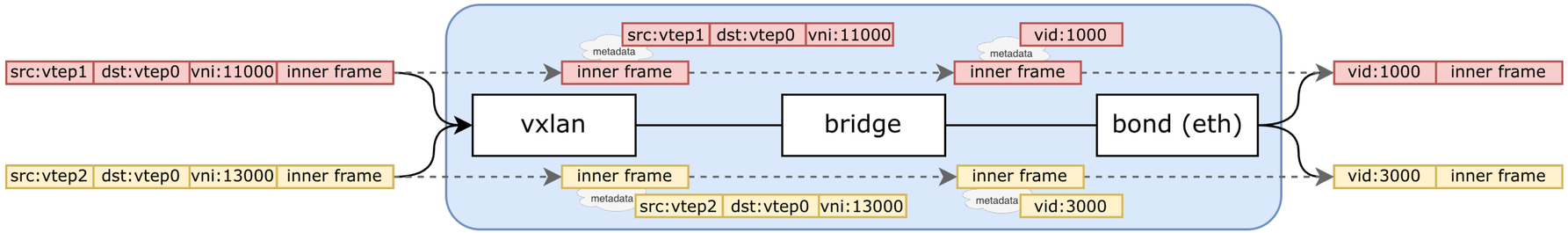


Single VXLAN Device (SVD) 構成をとる

- Traditional な 1 VNI を 1 VXLAN netdev で収容する構成
  - VNI が増えると vxlan、bridge 、vlan netdev が増える

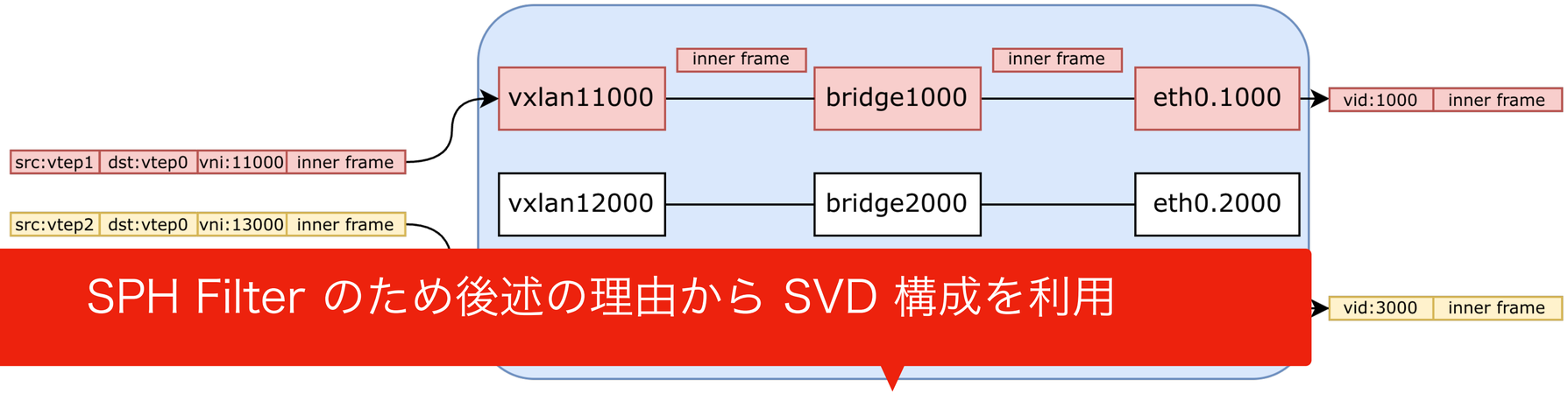


- 複数 VNI を 1 VXLAN netdev で収容する構成
  - bridge は vlan-aware 構成にする必要がある
  - ver 10.7.0 のドキュメントから SVD (Single VXLAN Device) 構成が recommended と記述

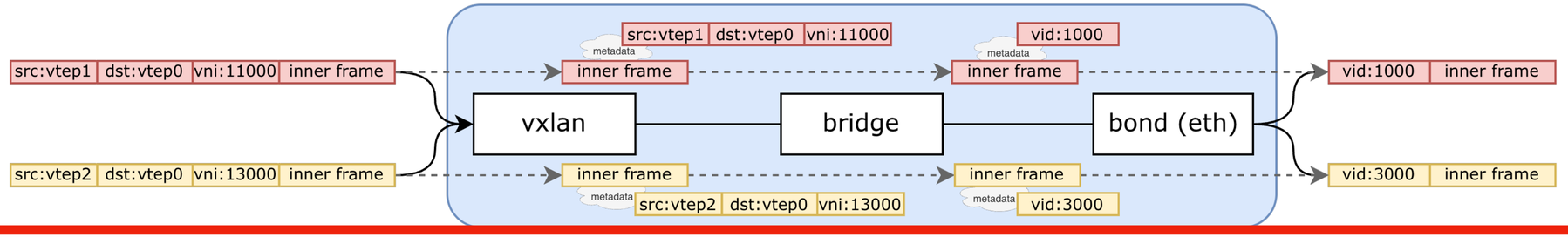


Single VXLAN Device (SVD) 構成をとる

- Traditional な 1 VNI を 1 VXLAN netdev で収容する構成
  - VNI が増えると vxlan、bridge 、vlan netdev が増える



- 複数 VNI を 1 VXLAN netdev で収容する構成
  - bridge は vlan-aware 構成にする必要がある
  - ver 10.7.0 のドキュメントから SVD (Single VXLAN Device) 構成が recommended と記述

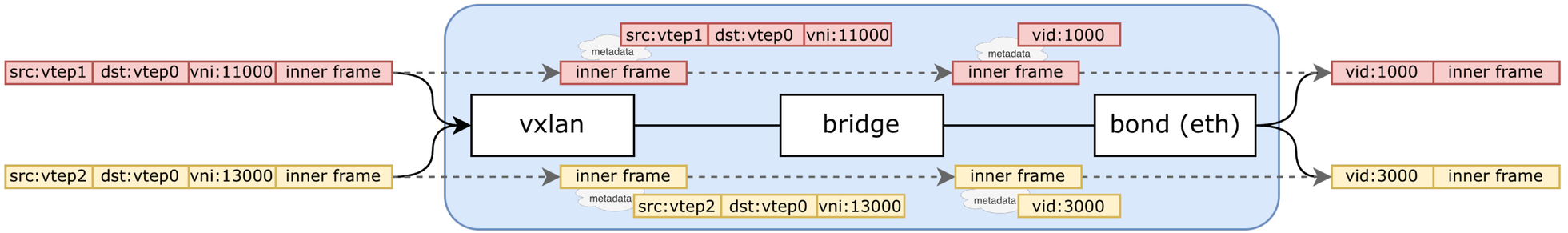


SVD 構成の設定と挙動

- 設定例 : vni:11000 を vid:1000 にmapping

```
ip link add br0 type bridge vlan_filtering 1 vlan_default_pvid 0
ip link add vxlan0 type vxlan dstport 4789 local 100.64.0.1 nolearning external vnifilter

bridge vlan add dev vxlan0 vid 1000
bridge vni add dev vxlan0 vni 11000
bridge vlan add dev vxlan0 vid 1000 tunnel_info id 11000
```

- 内部の挙動:
  1. 入力 vxlan パケットから src/dst ip と vni をメタデータとして skb に紐付け
  2. bridge から egress port に enqueu するタイミングで vni を vid に変換



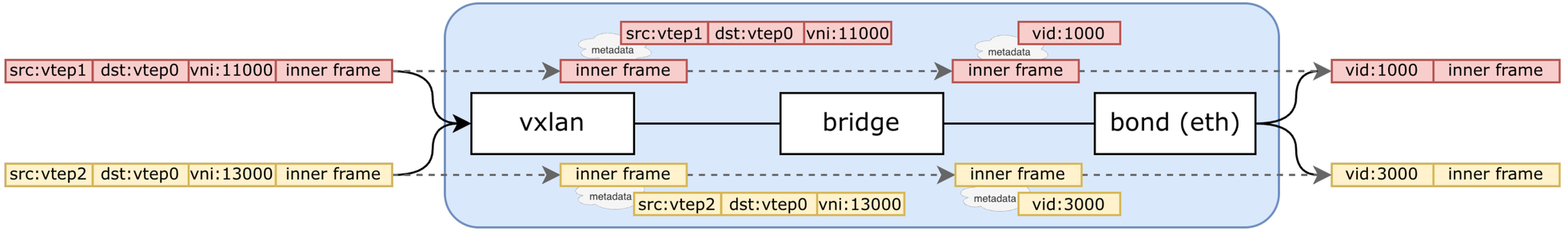
SVD (Single VXLAN Device) 設定と挙動

- 設定例 : vni:11000 を vid:1000 にmapping

```
ip link add br0 type bridge vlan_filtering 1 vlan_default_pvid 0
ip link add vxlan0 type vxlan dstport 4789 local 100.64.0.1 nolearning external vnifilter

bridge vlan add dev vxlan0 vid 1000
bridge vni add dev vxlan0 vni 11000
bridge vlan a
```

SPH Filter にこの送信元 VTEP-IP が必要。Traditional 構成では付かない。

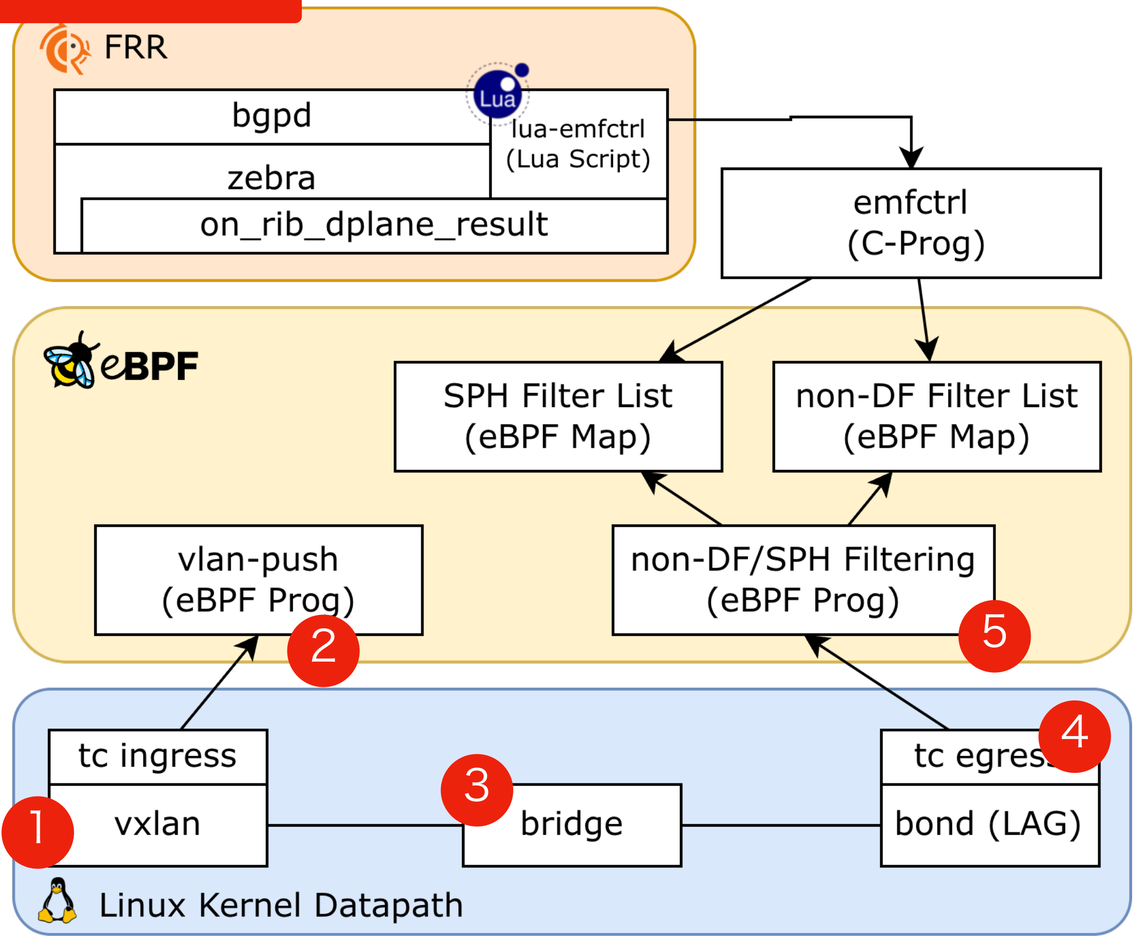
- 内部の挙動:
  1. 入力 vxlan パケットから **src/dst ip と vni をメタデータとして skb に紐付け**
  2. bridge から egress port に enqueu するタイミングで vni を vid に変換



(再掲) データプレーンの処理概要

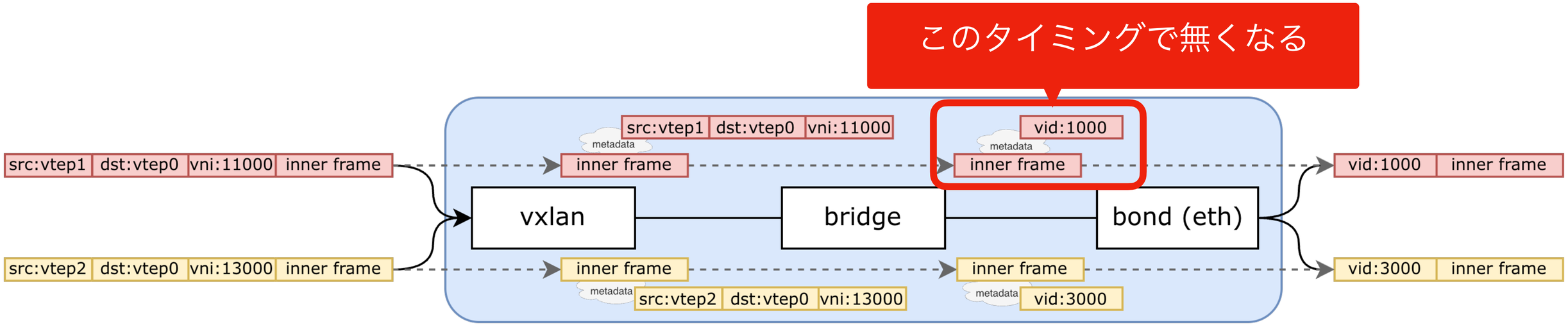
vlan-aware bridge が vni:vid 変換をするのでは?

1. vxlan-if が VXLAN パケットを受信
2. vxlan-if の ingress で vlan tag を push
   - tc で vlan-push eBPF Prog を実行
   - vni と vid の mapping は事前に設定
3. bridge で egress port が bond-if に決まる
4. bond-if の egress で BUM フレーム判定をする
   - tc flower により判定 Known unicast は pass
5. bond-if の egress で BUM に Filter を適用
   1. tc で non-DF/SPH Filtering eBPF Prog を実行
   2. eBPF Map から情報取得



ip_tunnel_metadata は VNI-VID 変換で消える

- bond-if での Filtering 処理で ip_tunnel_metadata が欲しい
- vlan-aware bridge の内部挙動:
  1. 入力 vxlan パケットから src/dst ip と vni をメタデータとして skb に紐付け
  2. bridge から egress port に enqueue するタイミングで vni を vid に変換
     - ここで ip_tunnel_info metadata は削除される

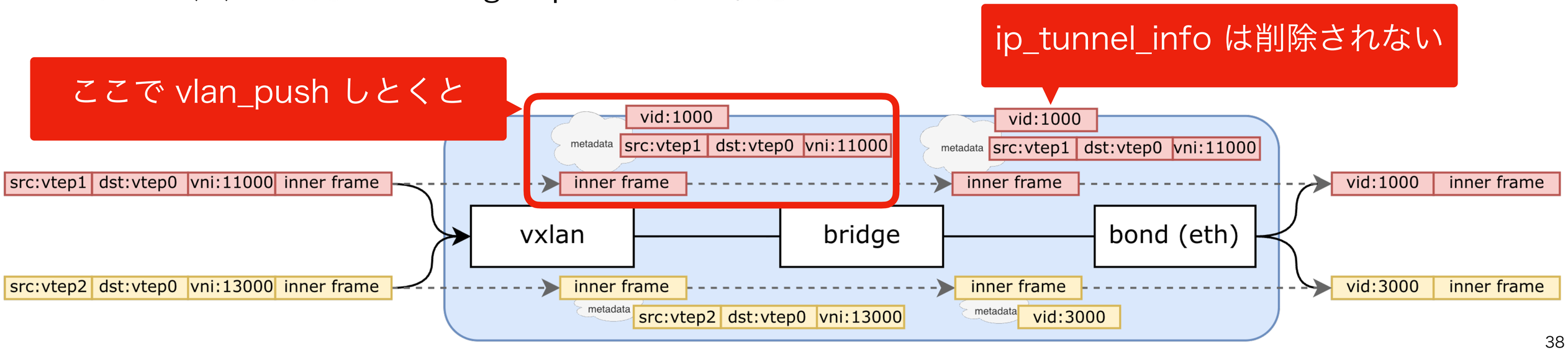


ワークアラウンド

- vlan-aware bridge の内部挙動:
  1. bridge から egress port に enqueu するタイミングで vni を vid に変換
     - ここで ip_tunnel_info metadata は削除される

- 上記 (1.) のタイミングで vlan tag が push されていると bridge での vni-vid 変換は行われない
  - vni-vid 変換が行われない場合 ip_tunnel_info metadata はそのまま残る

- そこで (1.) より先に vlan tag が push された状態としておく

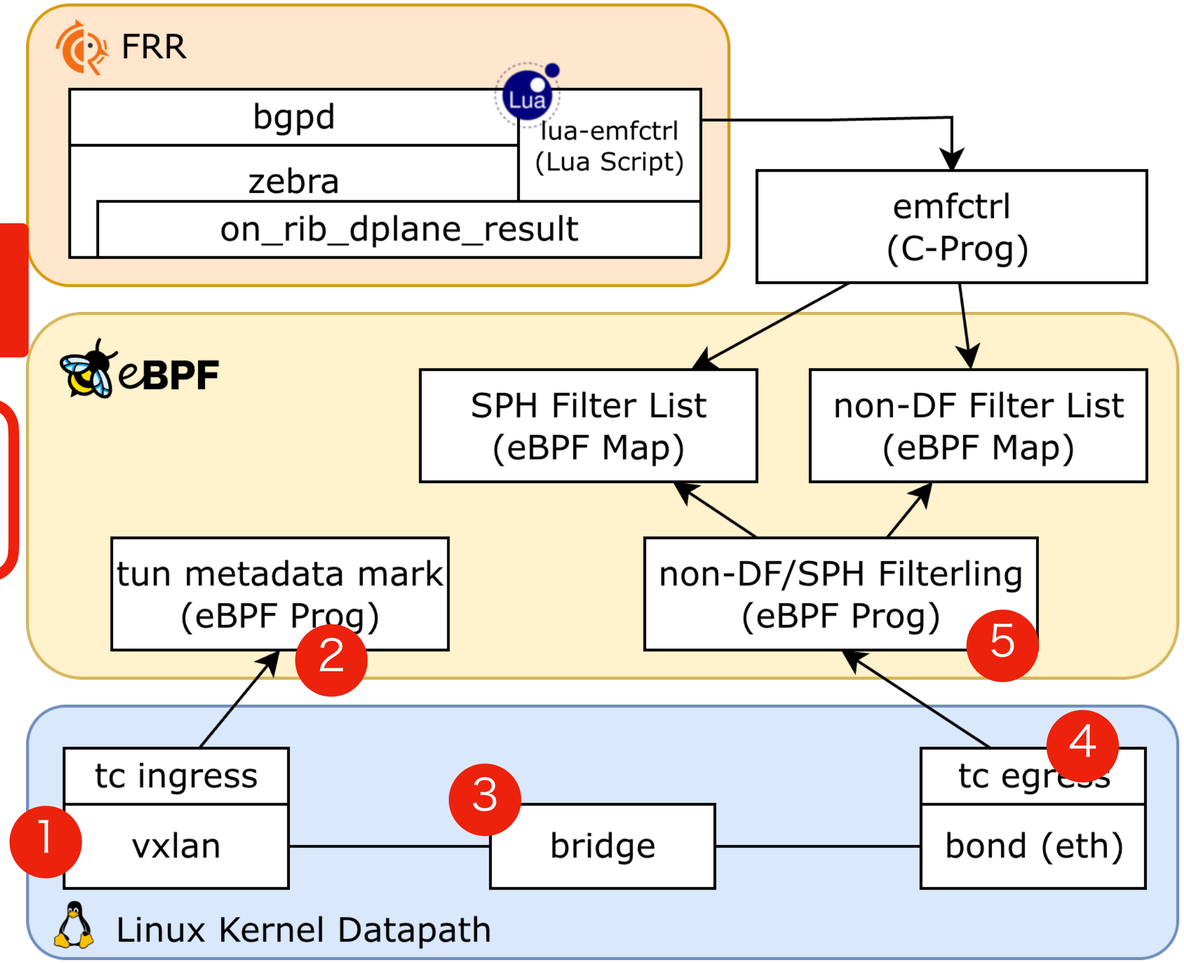


(再掲) データプレーンの処理概要

1. vxlan-if が VXLAN パケットを受信
2. vxlan-if の ingress で vlan tag を push
   - tc で vlan-push eBPF Prog を実行

eBPF Prog で判定すれば良いのでは？

3. bridge で egress port が bond-if に決まる
4. bond-if の egress で BUM フレーム判定をする
   - tc flower により判定 Known unicast は pass
5. bond-if の egress で BUM に Filter を適用
   1. tc で non-DF/SPH Filtering eBPF Prog を実行
   2. eBPF Map から情報取得



eBPF Prog から l2miss 属性にアクセスできない

- bridge の learning table にヒットしなかった場合 l2miss がセット
  - skb のメタデータ領域にセットされる
- eBPF Prog は skb のメタデータには直接アクセスできない
  - BPF helper 関数などを介してアクセスするのが基本
    - tunnel metadata も helper 関数経由でアクセスしている
  - l2miss を取得する helper 関数はない
- 今回は tc の flower と bpf の組み合わせで対応

```
tc filter add dev bond egress pref 1 flower dst_mac 00:00:00:00:00:00/01:00:00:00:00:00 l2_miss 0 action pass
tc filter add dev bond egress pref 2 bpf da obj egress_mh_filter.o sec tc
```

(参考) eBPF プログラムはシンプル

vlan_push

```c
int
evpn_mh_vlan_push(struct __sk_buff *skb)
{
        struct bpf_tunnel_key key = {};
        __u16 *vid;
        int ret;

        ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key), 0);
        if (ret == -EPROTO)
                ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key),
                    BPF_F_TUNINFO_IPV6);
        if (ret != 0)
                return TC_ACT_OK;

        vid = bpf_map_lookup_elem(&evpn_mh_vni_vlan, &key.tunnel_id);
        if (!vid)
                return TC_ACT_OK;

        bpf_skb_vlan_push(skb, bpf_htons(ETH_P_8021Q), *vid);

        return TC_ACT_OK;
}
```

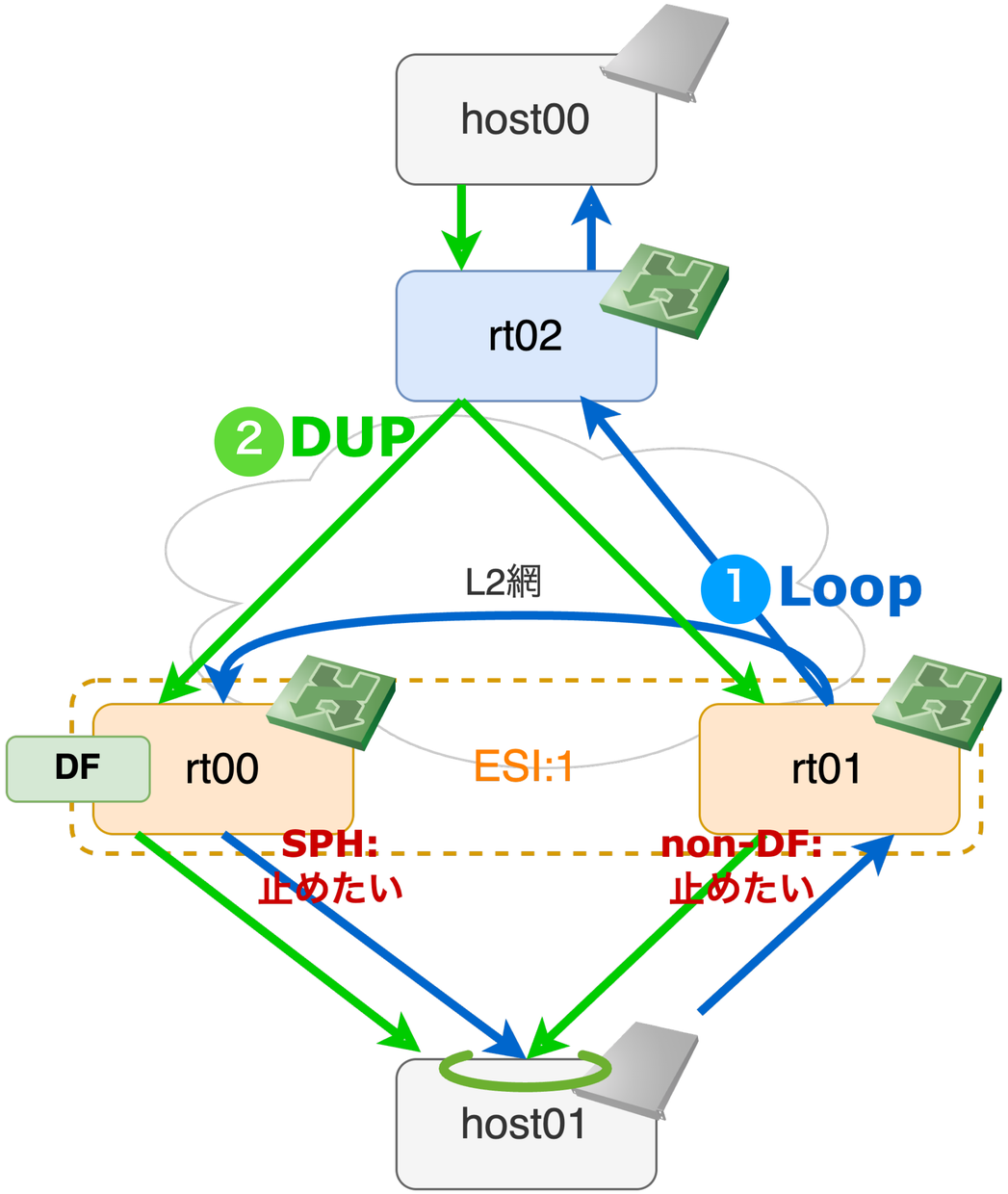
non-DF/SPH Filter

```c
int
evpn_mh_egress_filter(struct __sk_buff *skb)
{
                        :
        non_df = bpf_map_lookup_elem(&evpn_mh_df, &ifindex);
        if (non_df && *non_df) {
                bpf_printk("evpn-mh: DF-block ifindex=%d\n", ifindex);
                return TC_ACT_SHOT;
        }

        sph_inner = bpf_map_lookup_elem(&evpn_mh_sph_list, &ifindex);
        if (!sph_inner)
                return TC_ACT_OK;

        ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key), 0);
        if (ret == 0) {
                vk.af = VTEP_AF_INET;
                vk.addr.v4.s_addr = bpf_htonl(key.remote_ipv4);
        } else if (ret == -EPROTO) {
                ret = bpf_skb_get_tunnel_key(skb, &key, sizeof(key),
                    BPF_F_TUNINFO_IPV6);
                if (ret == 0) {
                        vk.af = VTEP_AF_INET6;
                        vk.addr.v6.s6_addr32[0] = key.remote_ipv6[0];
                        vk.addr.v6.s6_addr32[1] = key.remote_ipv6[1];
                        vk.addr.v6.s6_addr32[2] = key.remote_ipv6[2];
                        vk.addr.v6.s6_addr32[3] = key.remote_ipv6[3];
                }
        }

        if (ret != 0)
                return TC_ACT_OK;

        if (bpf_map_lookup_elem(sph_inner, &vk)) {
                bpf_printk("evpn-mh: split-horizon block ifindex=%d af=%d\n",
                           ifindex, vk.af);
                return TC_ACT_SHOT;
        }

        return TC_ACT_OK;
}
```

デモ

- PE は MAC を学習しない状態
  - Unknown Unicast とするため

- host01 は rt01 側から送信
  1. rt01 は rt00|02 に flooding
     - rt00 は SPH Filtering する
  2. rt02 は rt00|rt02 に flooding
     - rt00 が DF なので rt01 が non-DF Filtering



考察

- 基本的にコードの変更なしで実現できる
  - FRR は on_rib_process_dplane_results、Linux は eBPF hook を利用
- 複数 netdev にまたがるパケット metadata のやり取りは悩みどころ
  - skb の metadata はどの範囲まで有効か問題
    - ip_tunnel_metadata が消えるのは bridge と egress port の関係では自然
  - l2miss を取得できる eBPF helper 関数は欲しい
    - Unknown MAC Route (UMR) という概念があるのでそれのハンドル用にも使える
  - 実はこの仕組みは br_ntefilter を有効にすると動かない
    - ip_tunnel_metadata が削除されるため

まとめ

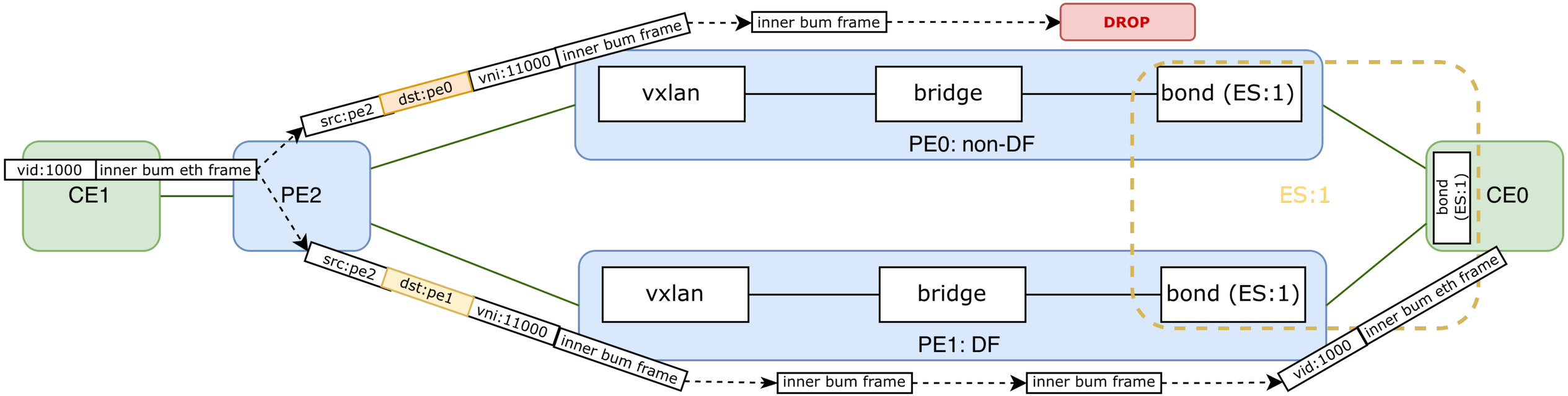
- FRR の EVPN-MH は D-lane の non-DF/SPH Filter が未実装
- D-Plane に non-DF/SPH Filter を実装
  - FRR は on_rib_process_dplane_results、Linux は eBPF hook を利用
  - 基本的にコードの変更なしで実現できる
- 「動いていると言えば動いている」FRR EVPN-MH の話でした

Backup

non-DF Filter

CE1 -> CE0 への BUM フレーム送信

- PE0 は Egress ES (Port) が non-DF & BUM フレームの場合 DROP する
  - ▸



SPH Filter

CE0 via PE1 -> CE1 への BUM フレーム送信

- PE0 は vxlan の src-ip が SPH List に含まれている & BUM の場合に DROP
  - ▸