Seeing Double?
Implementing Multicast with eBPF and Cilium

By: Louis DeLosSantos
Email: louis.delos@isovalent.com
Mastodon: https://fosstodon.org/@ldelossa

Who am I?

Datapath Engineer at Cisco (Isovalent Team)

Focus on Linux kernel networking and eBPF

Open source software enthusiast

Neovim plugin and Linux desktop developer in free time.

What we'll cover

- A gentle introduction to multicast
- How multicast is implemented with eBPF and Cilium

Follow Along

https://github.com/cilium/cilium/blob/main/bpf/lib/mcast.h

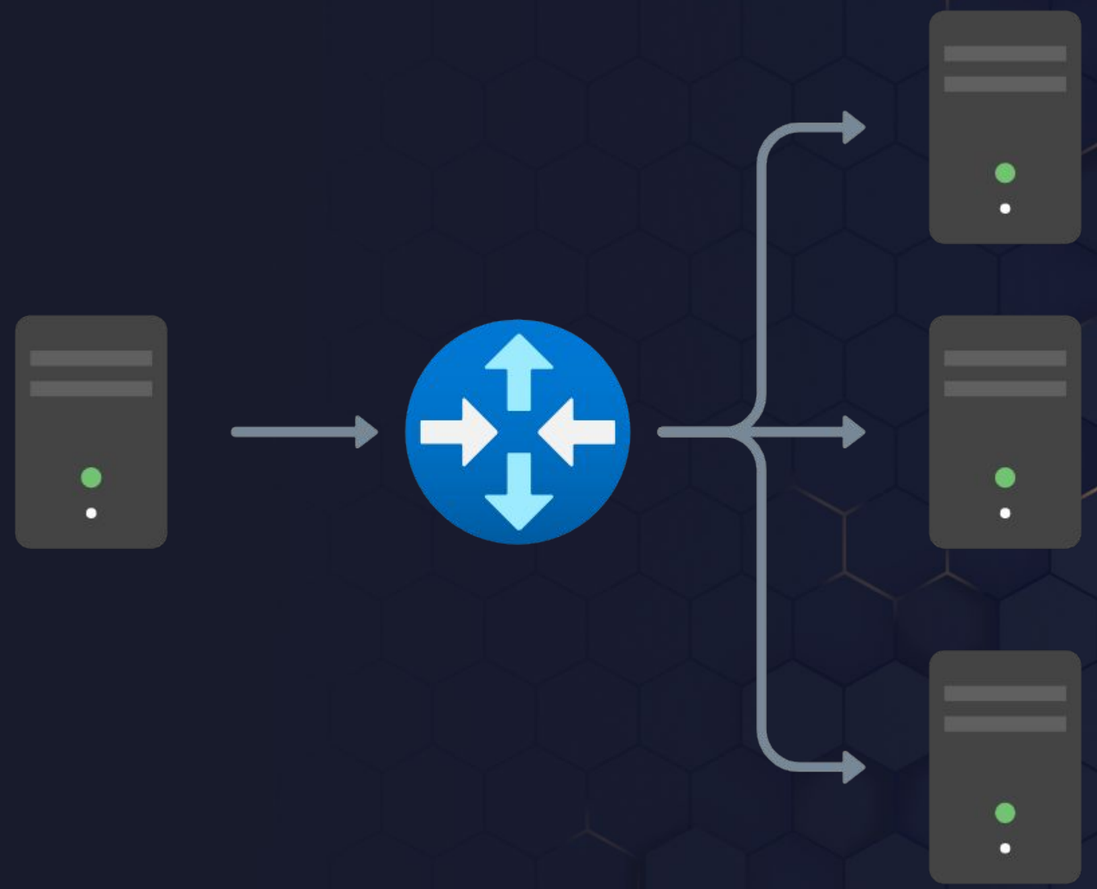
But first, some disclaimers!!

1. I am not a multicast expert.
2. Multicast within the context of Kubernetes and Cilium
3. A focus on the eBPF datapath

ISOVALENT

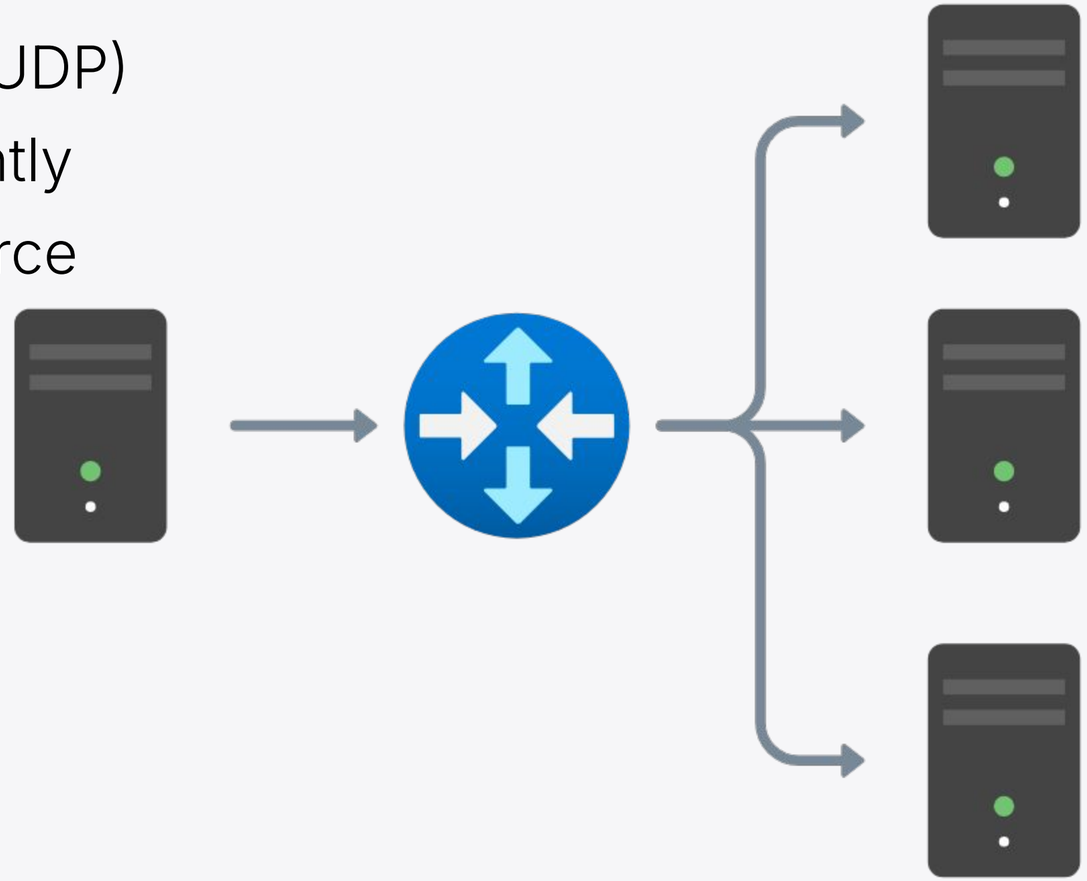
What is multicast?

A gentle introduction...



What is multicast?

- Properly introduced in **RFC 1112 - Host extension for IP Multicasting**
- Operates at the IP layer, layer 3
- Typically connectionless layer 4 protocols (UDP)
- Unicast delivery to a group of hosts, efficiently
- Removes the resource burden from the source

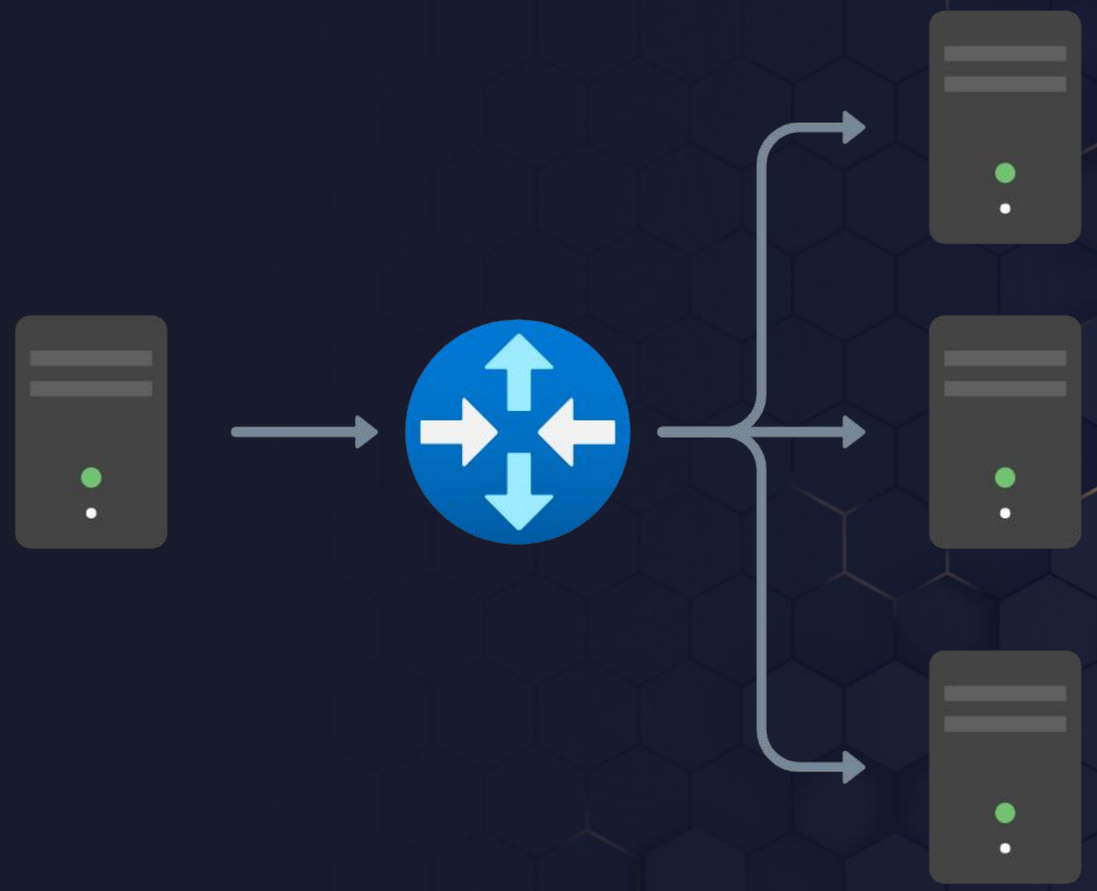


Where is multicast used?

- Broadcasting and Media Streaming
  - A/V streams efficiently delivered to, at times, millions of clients
- Financial Services
  - Market data distribution
- Online Gaming
  - Broadcasting of game state and updates to all players

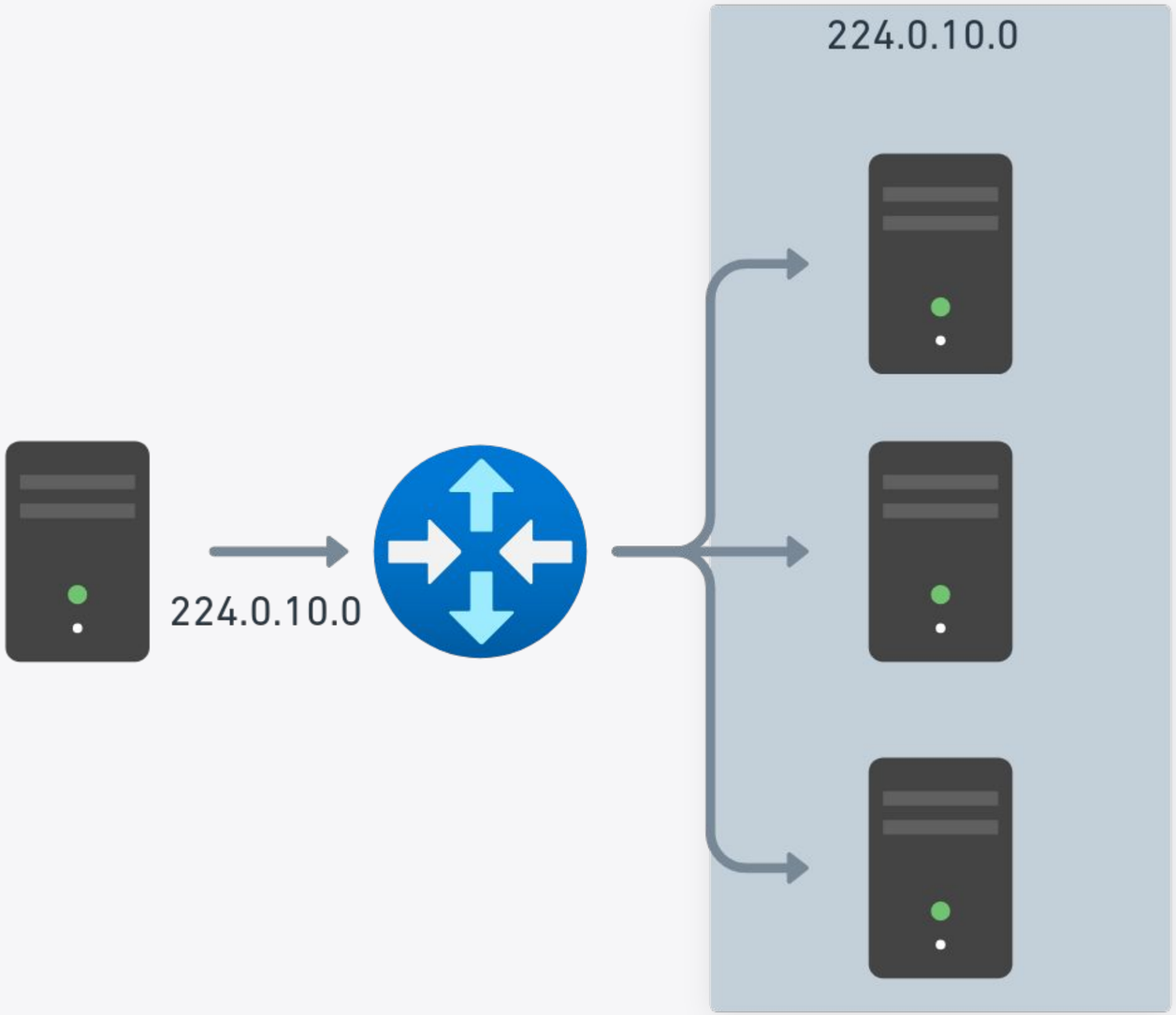
ISOVALENT

How multicast works



Multicast Group Addresses

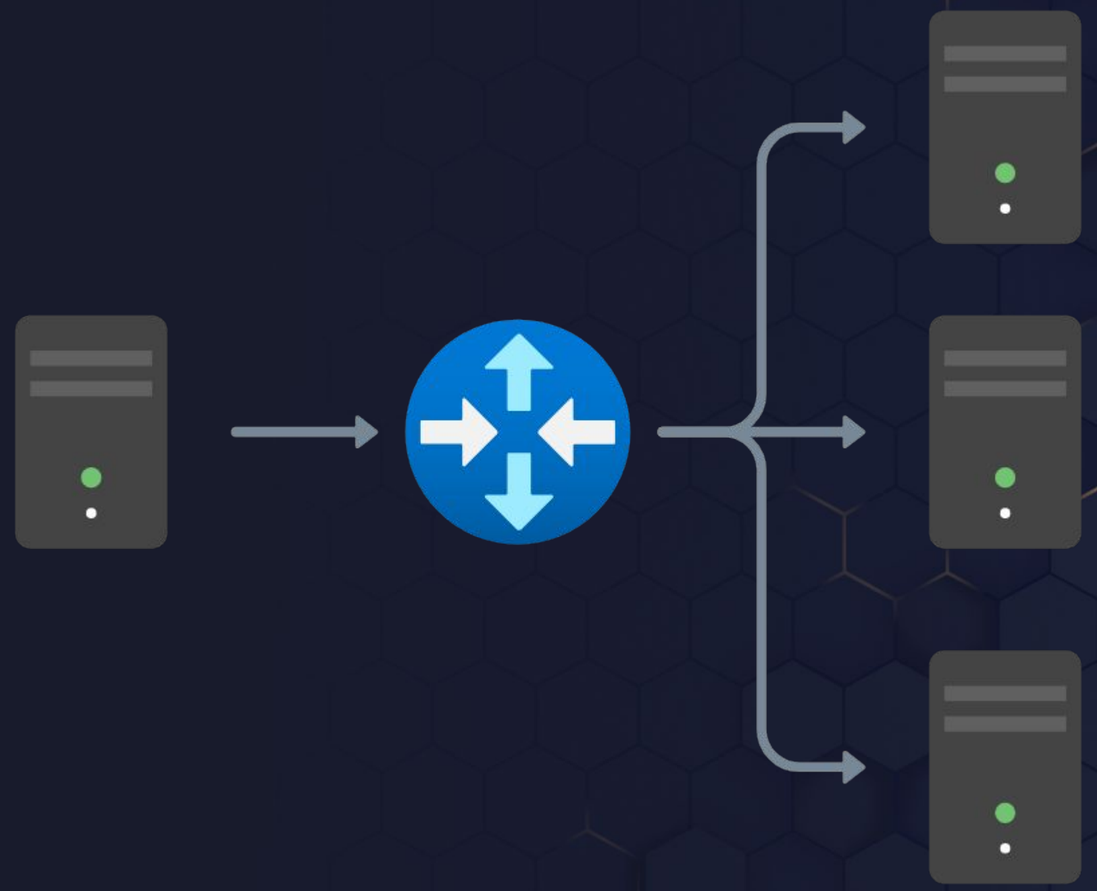
- Internet addresses which represent a group of hosts.
- Class D address
  - 224.0.0.0 - 239.255.255.255
  - 224.0.0.0- guaranteed not in use
  - 224.0.0.1 - all-hosts address for LAN
    - useful for service discovery! (mDNS)
  - 224.0.0.2 - all-routers address
- Traffic send to a group host address is delivered to all members of the group.



ISOVALENT

Group Management

The IGMP protocol



Internet Group Management Protocol (IGMP)

- Workhorse protocol for Multicast
- Implements group management
  - Hosts joining groups
  - Hosts leaving groups
  - Routers/Hosts querying groups
- A Layer 3 (IP) Protocol
  - IP Protocol 2
- Two major objects
  - Membership Reports
    - Sent by hosts for group management
  - Membership Queries
    - Sent by Multicast routers for group querying

Internet Group Management Protocol (IGMP)

- Multiple Version
- IGMPv1
  - Introduced in RFC 1112
  - Host membership query
  - Host membership report
- IGMPv2
  - Introduced in RFC 2236
  - Group specific membership query
  - LEAVE message type
- IGMPv3
  - Introduced in RFC 3376
  - Source Filtering*

IGMPv1 on the Wire

- Very simple format
- Kernel uses `struct igmphdr` for all IGMP versions
- 8 bytes
- `type` is a type code
  - `1` - Host Membership Query
  - `2` - Host Membership Report
- `code` is unused
- `group` is the Group Address

```
struct igmphdr {
        __u8 type;
        __u8 code;              /* For newer IGMP */
        __sum16 csum;
        __be32 group;
};
```

IGMPv2 on the Wire

- New type codes
- `type` is a type code
  - `0x11` - Membership Query
  - `0x16` - Membership Report V2
  - `0x17` - Leave Group
  - `0x12` - Membership Report V1
- `code` is used for maximum response time (Membership Queries only)
- `group` is the Host Group Address

```
struct igmphdr {
        __u8 type;
        __u8 code;              /* For newer IGMP */
        __sum16 csum;
        __be32 group;
};
```

IGMPv3 on the Wire

- IGMPv3 makes membership reports variable sized
- `type` is a type code
  - `0x22` - IGMPv3 Membership Report
- When `0x22` type is encountered the packet is parsed with `struct igmpv3_report`
- `igmpv3_report` holds a variable array of Group Records (`struct igmpv3_grec`)
- Each Group Record is an intent to joint a particular group with optional source filtering.

```
struct igmphdr {
        __u8 type;
        __u8 code;              /* For newer IGMP */
        __sum16 csum;
        __be32 group;
};
```

```
struct igmpv3_grec {
        __u8    grec_type;
        __u8    grec_auxwords;
        __be16  grec_nsrcs;
        __be32  grec_mca;
        __be32  grec_src[];
};

struct igmpv3_report {
        __u8 type;
        __u8 resv1;
        __sum16 csum;
        __be16 resv2;
        __be16 ngrec;
        struct igmpv3_grec grec[];
};
```

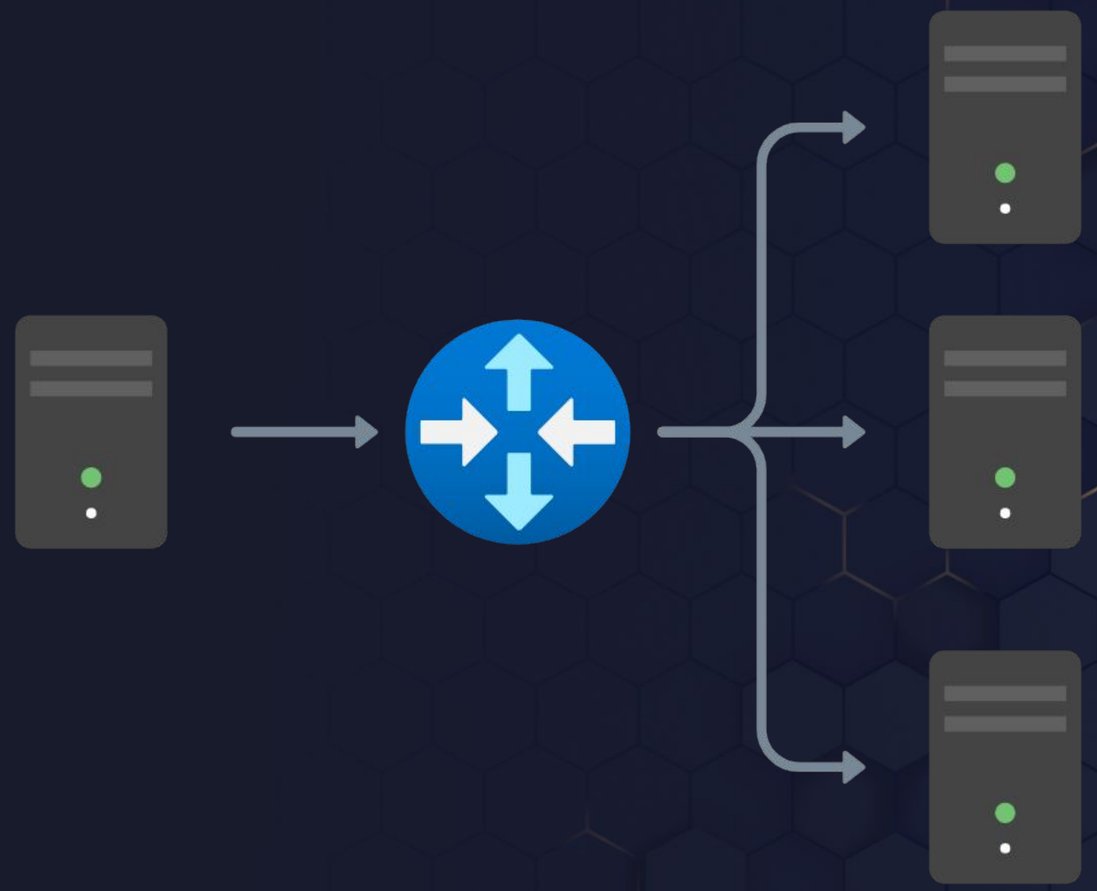
IGMPv3 Group Record

- Supports source filtering*
- `grec_type` is a type code which helps interpret list of sources
  - `1` - MODE_IS_INCLUDE
  - `2` - MODE_IS_EXCLUDE
  - ...
- `grec_auxwords` not used
- `grec_nsrcs` number of source addresses in `grec_src`
- `grec_mca` the multicast group address the Group Record refers to.
- `grec_src` a variable list of IP unicast source addresses.

```
struct igmpv3_grec {
        __u8    grec_type;
        __u8    grec_auxwords;
        __be16  grec_nsrcs;
        __be32  grec_mca;
        __be32  grec_src[];
};

struct igmpv3_report {
        __u8 type;
        __u8 resv1;
        __sum16 csum;
        __be16 resv2;
        __be16 ngrec;
        struct igmpv3_grec grec[];
};
```

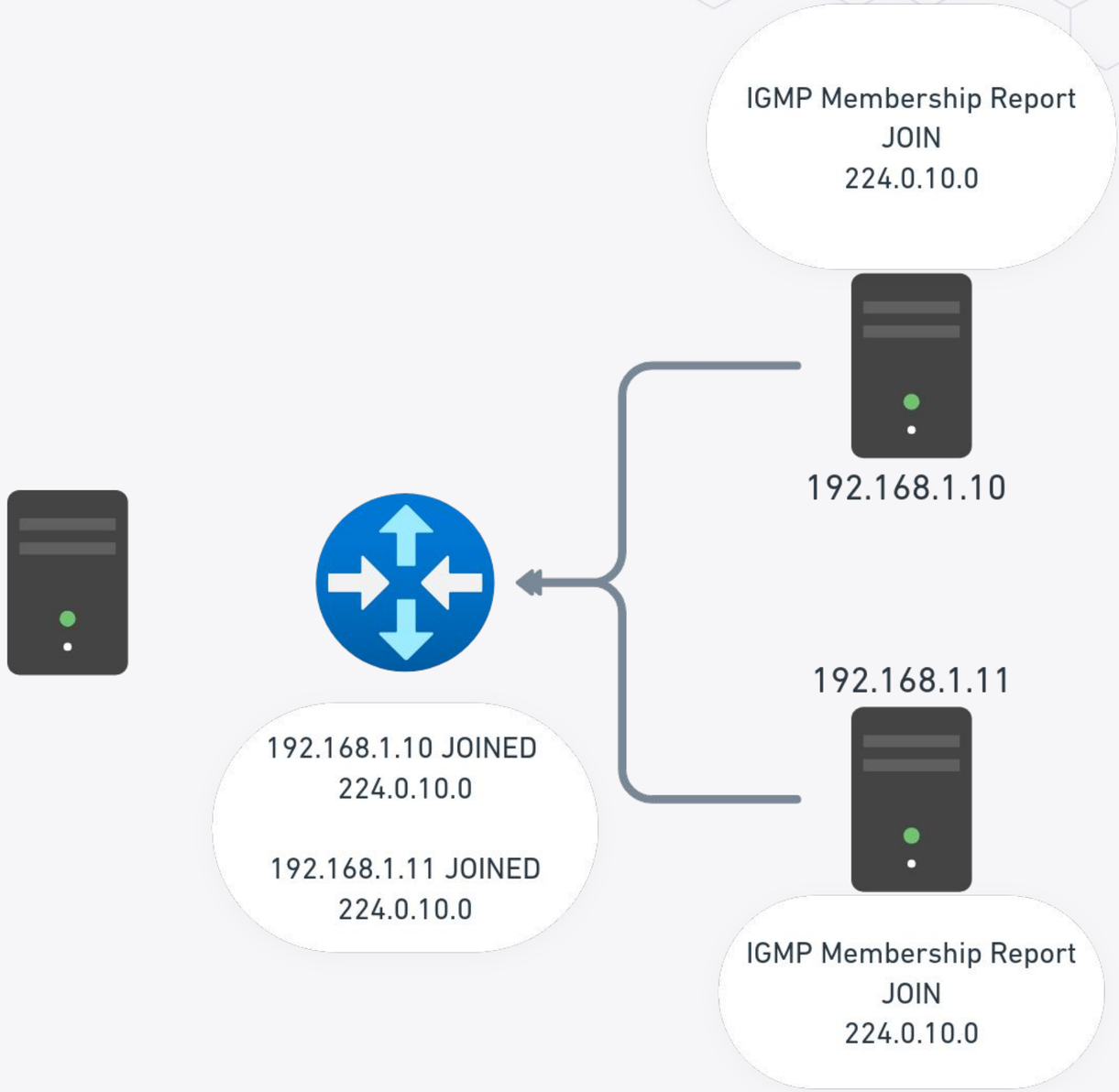
ISOVALENT

IGMP In Action



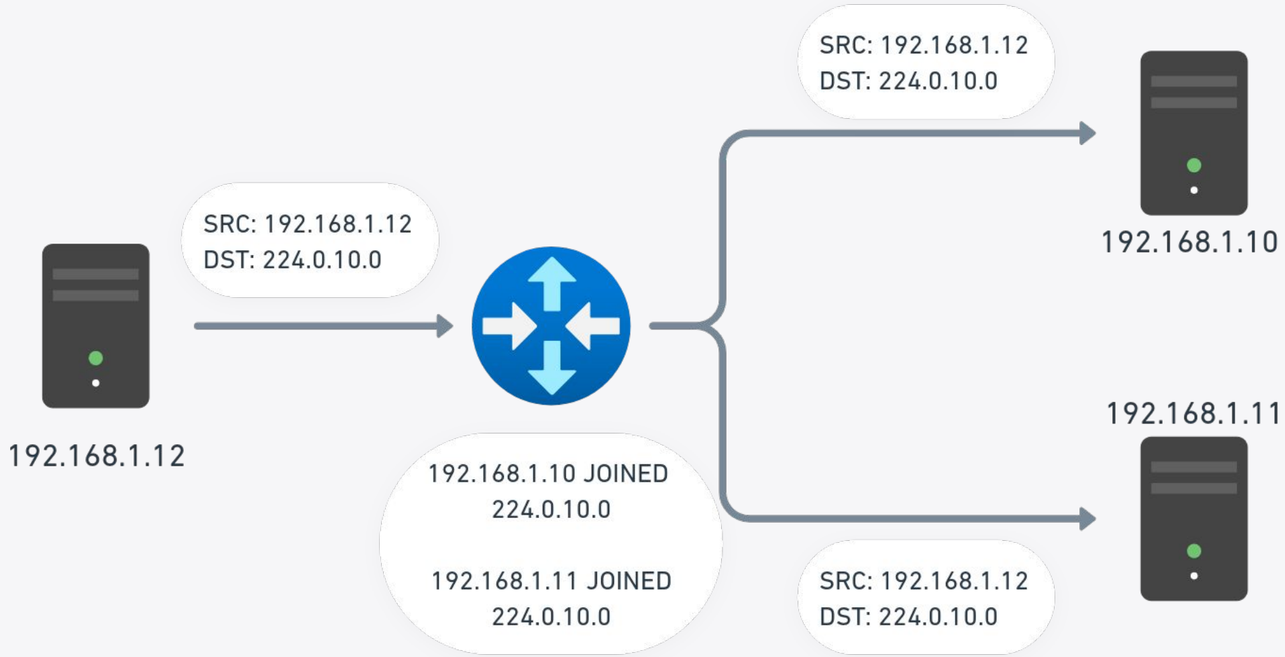
IGMP Snooping

- A multicast router listens for JOIN messages
- Keeps track of which sources joined which groups
- Multicast applications **MUST** send a JOIN message on initialization (per RFC 1112)



Packet Replication

- A media server sends data to a multicast group.
- The multicast router receives this data, replicates it, and delivers it.

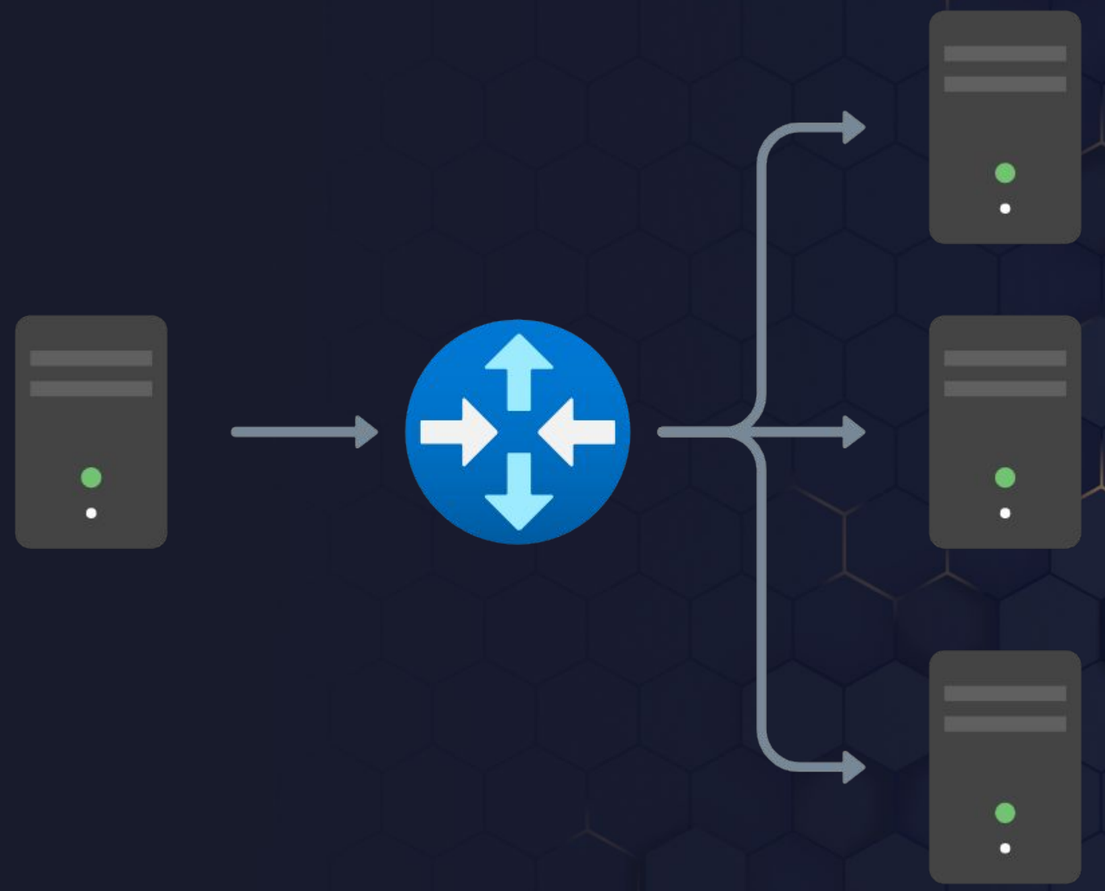


ISOVALENT

Implementing Multicast with eBPF and Cilium

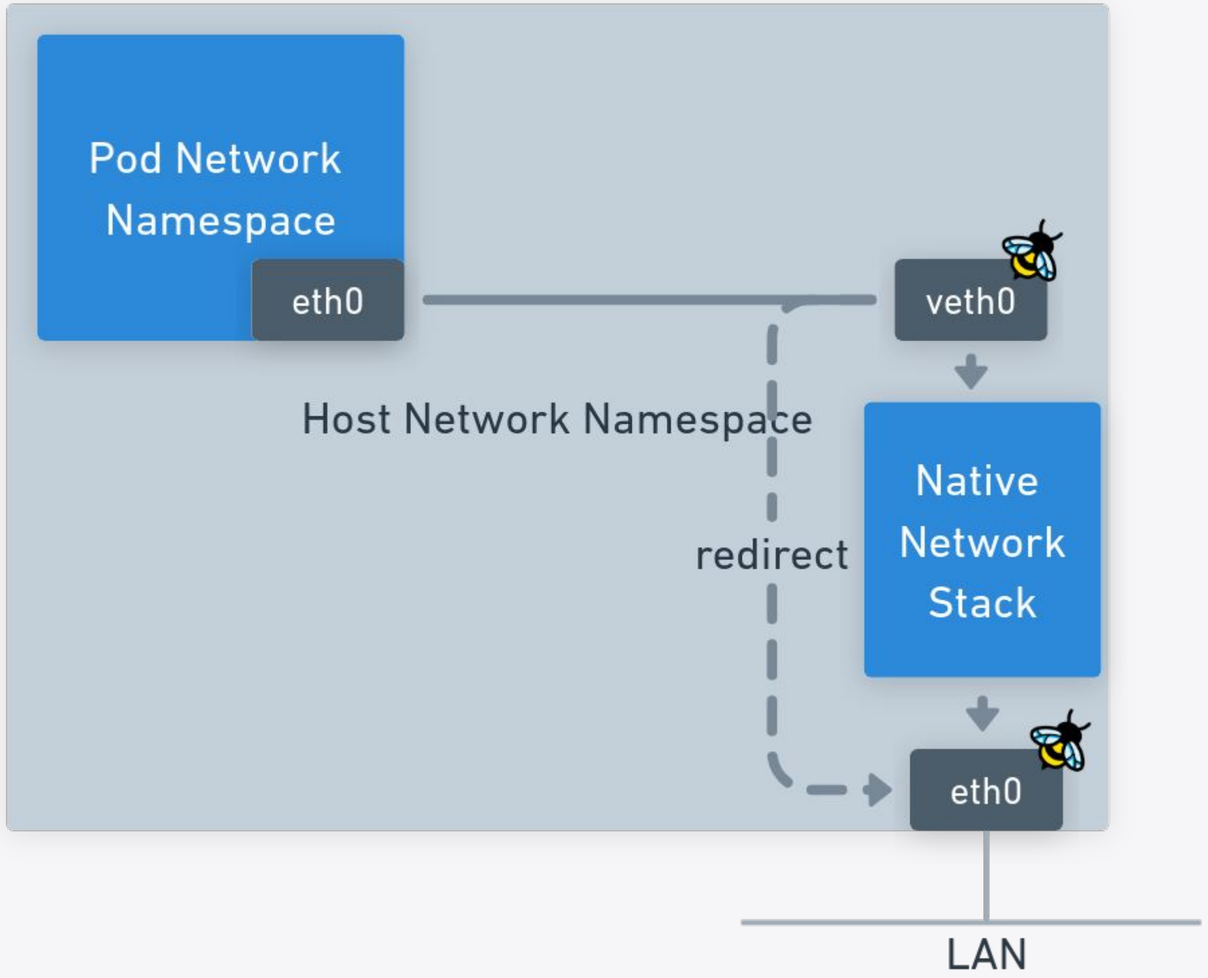
ISOVALENT

Kubernetes And Cilium Network Architecture



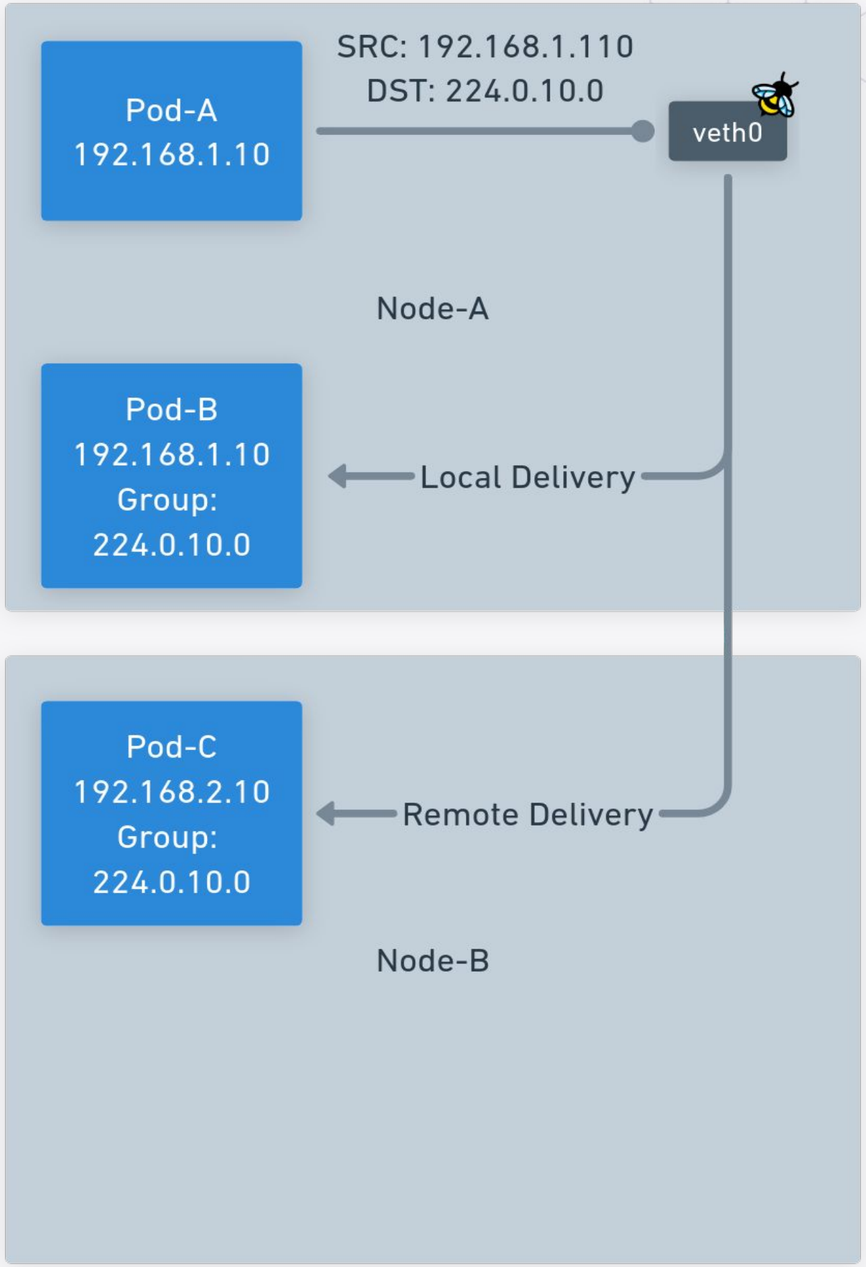
Simplified Topology

- Pods deployed in their own network namespace
- Veth pair connects Pod netns with host's
- Cilium attaches eBPF on the host-side veth pair.
- Cilium attaches eBPF to the native device attached to the LAN
- eBPF can push a packet to the native stack, or redirect to another interface



Multicast Topology

- Pod-A sends to a multicast group
- eBPF is used to replicate the packet and
  - Deliver to local pods in group
  - Deliver to remote pods in group



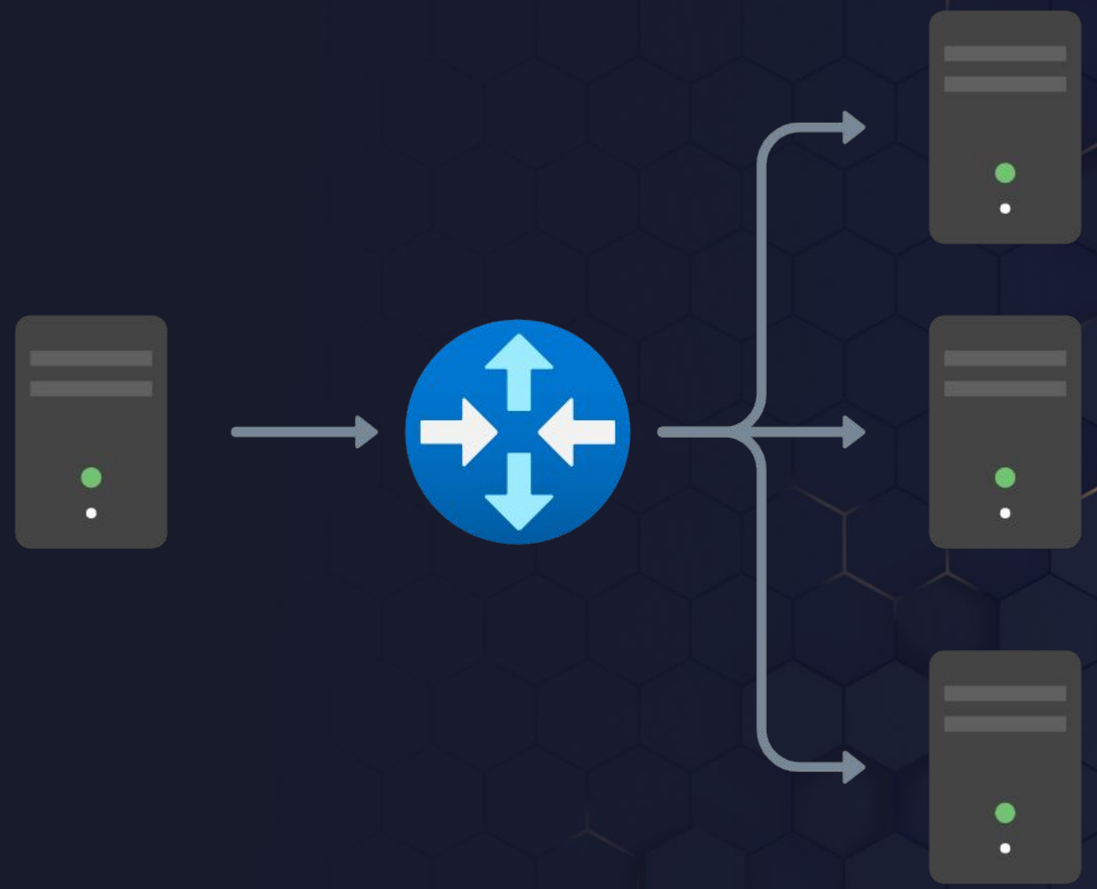
The puzzle pieces

- Group Membership Management
- Packet Replication
- Local Multicast Delivery
- Remote Multicast Delivery

ISOVALENT

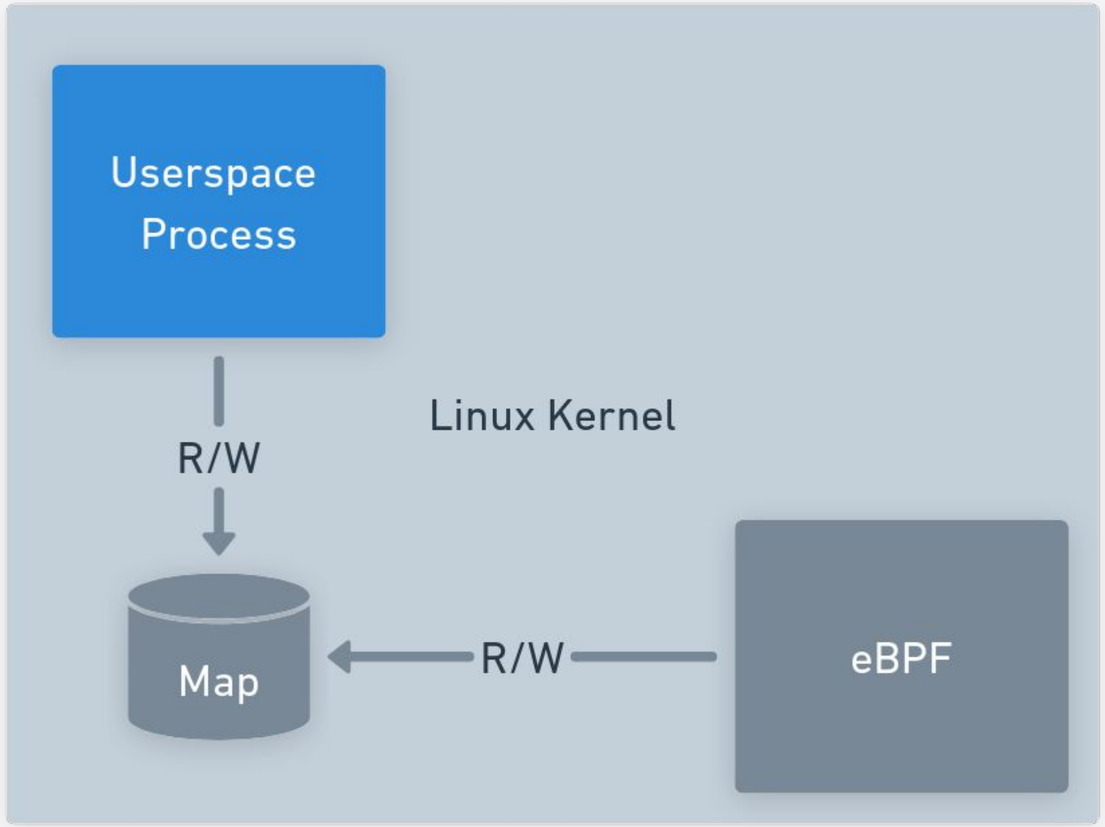
Group Management

Storing state



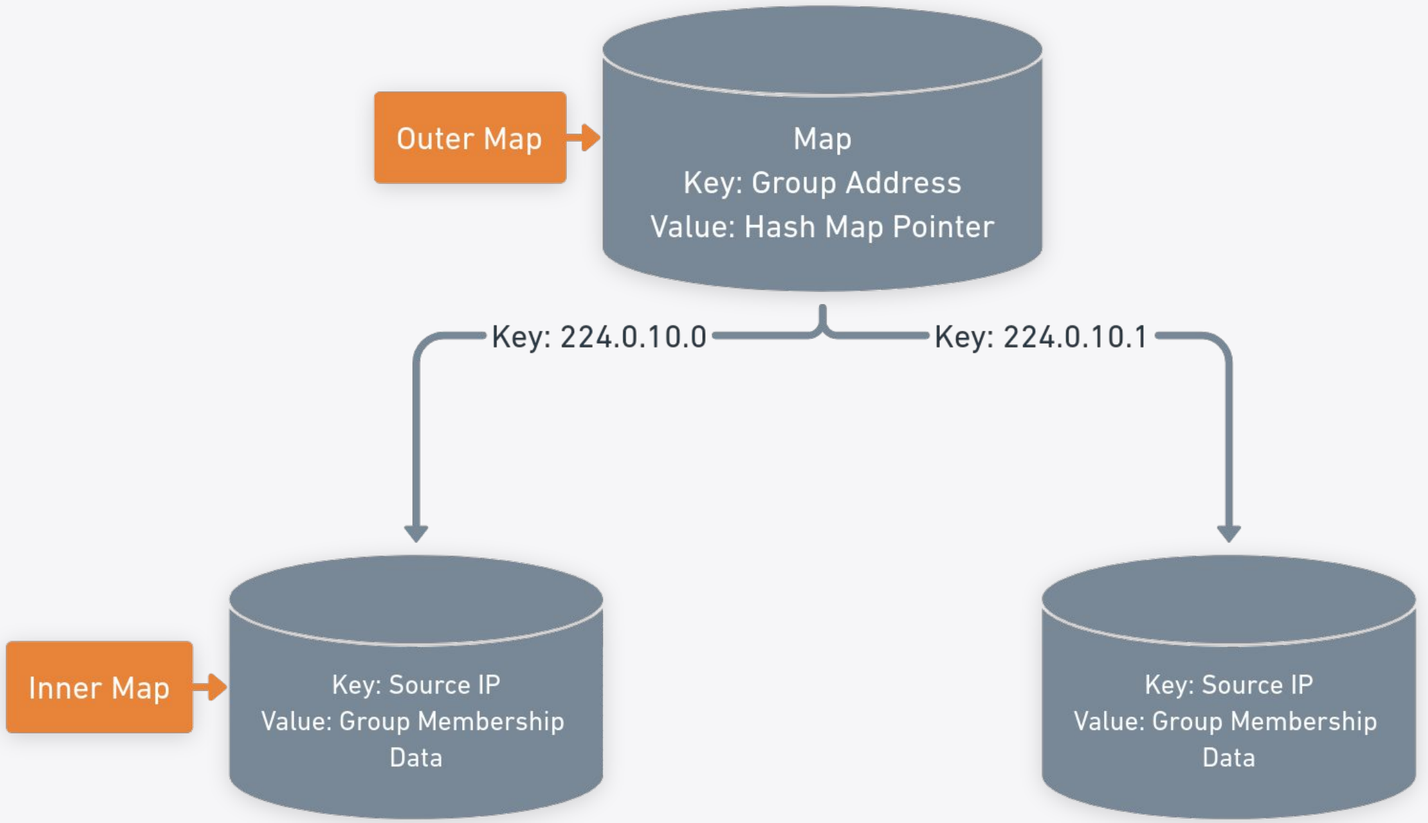
Storing State With Maps

- In-Kernel dynamic datastructures
- Created in userspace
- General Maps
  - Hash
  - Array
- Specific Maps
  - Nested Maps*
  - LPM Maps
  - Prog Maps
  - ...
- Provides:
  - State between eBPF invocations
  - Communication between eBPF and Userspace



Storing Group Membership with Maps

- Conceptually we want:
  - Map Key: Multicast Group Address
  - Map Value: Multicast Group Members
- Best fit?
  - A nested map type: `BPF_MAP_TYPE_HASH_OF_MAPS`
  - Hash map who's values point to another hash map.
- Lookup multicast address on outer map
- Lookup subscribers on returned inner map.



Nested Map Definition

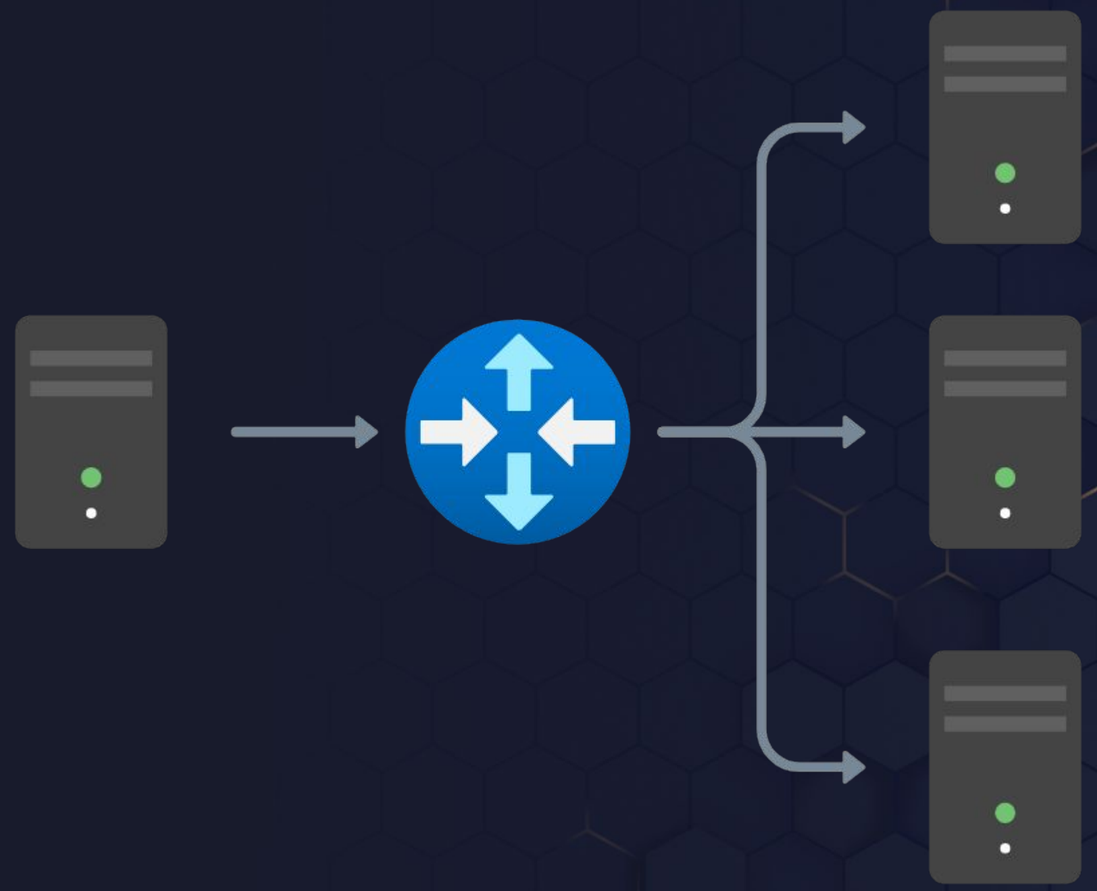
- `cilium_mcast_group_outer_v4_map` defines outer map
  - key: `mcast_group_v4`
  - value: inner map
    - key: `__be32` - subscriber's source address
    - value: `struct mcast_subscriber_v4`
- `struct mcast_subscriber_v4`
  - `saddr` - source address of subscriber
  - `ifindex` - interface used for multicast delivery
  - `flags` - flags used to specify properties of subscriber.

```
/* 32bit big endian multicast group address for use with ipv4 protocol */
typedef __be32 mcast_group_v4;

/* structure to describe a local or remote subscriber of a multicast group
 * for the ipv4 protocol.
 */
struct mcast_subscriber_v4 {
>       /* source address of the subscriber, big endian */
>       __be32 saddr;
>       /* local ifindex of subscriber of exit interface is remote subscriber */
>       __u32 ifindex;
>       /* reserved */
>       __u16 pad1;
>       /* reserved */
>       __u8  pad2;
>       /* flags for further subscriber description */
>       __u8  flags;
};

#ifdef ENABLE_MULTICAST

#define MCAST_MAX_GROUP 1024
#define MCAST_MAX_SUBSCRIBERS 1024
/* used to bound iteration of group records within an igmpv3 membership report */
#define MCAST_MAX_GREC 24

/* Multicast group map is a nested hash of maps.
 * The outer map is keyed by a 'mcast_group_v4' multicast group address.
 * The inner value is an hash map of 'mcast_subscriber_v4' structures keyed
 * by a their IPv4 source address in big endian format.
 */
struct {
>       __uint(type, BPF_MAP_TYPE_HASH_OF_MAPS);
>       __type(key, mcast_group_v4);
>       __type(value, __u32);
>       __uint(pinning, LIBBPF_PIN_BY_NAME);
>       __uint(max_entries, MCAST_MAX_GROUP);
>       __uint(map_flags, CONDITIONAL_PREALLOC);
>       /* Multicast group subscribers inner map definition */
>       __array(values, struct {
>       >       __uint(type, BPF_MAP_TYPE_HASH);
>       >       __uint(key_size, sizeof(__be32));
>       >       __uint(value_size, sizeof(struct mcast_subscriber_v4));
>       >       __uint(max_entries, MCAST_MAX_SUBSCRIBERS);
>       >       __uint(map_flags, CONDITIONAL_PREALLOC);
>       });
} cilium_mcast_group_outer_v4_map __section_maps_btf;
```

ISOVALENT

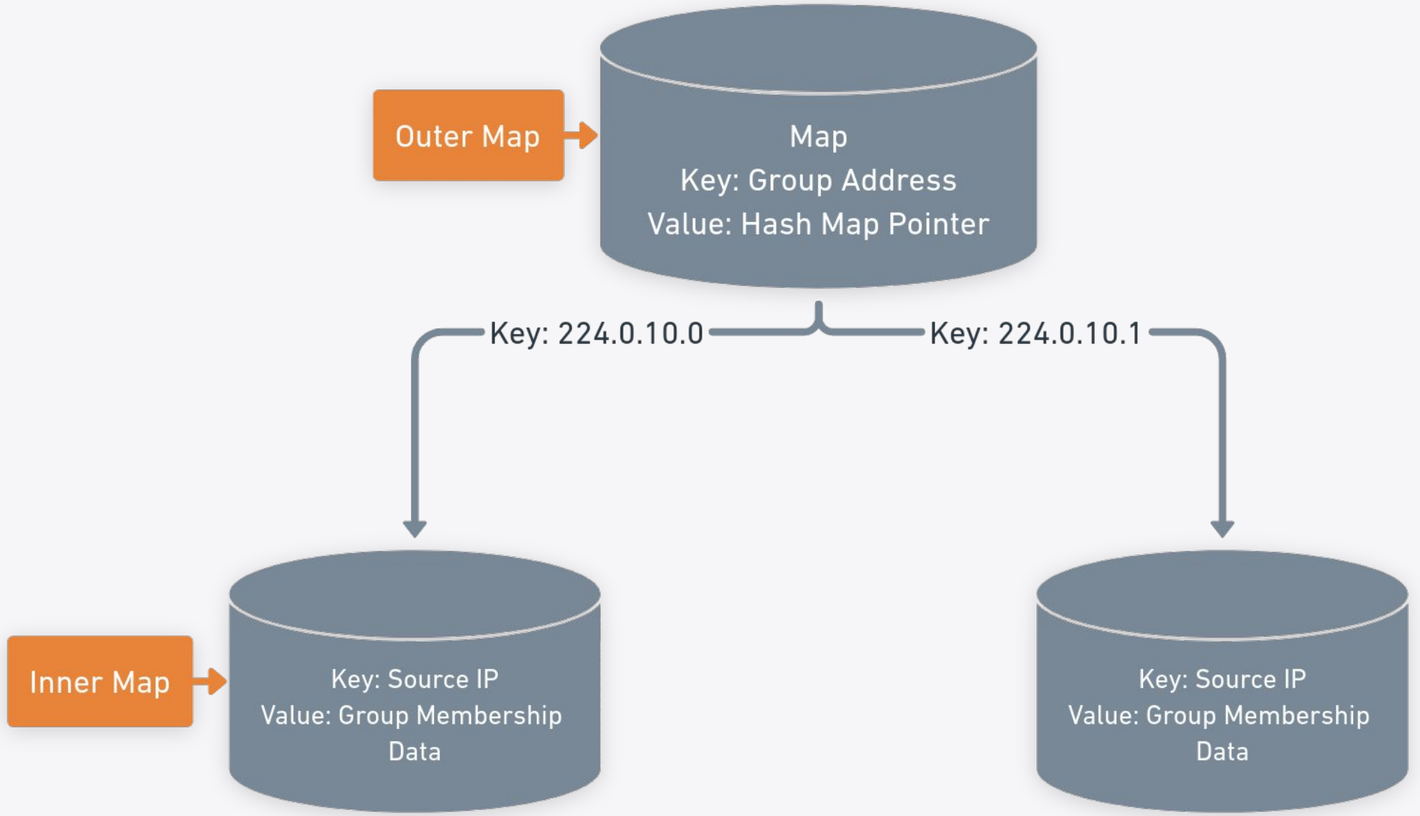
Group Management

IGMP Snooping



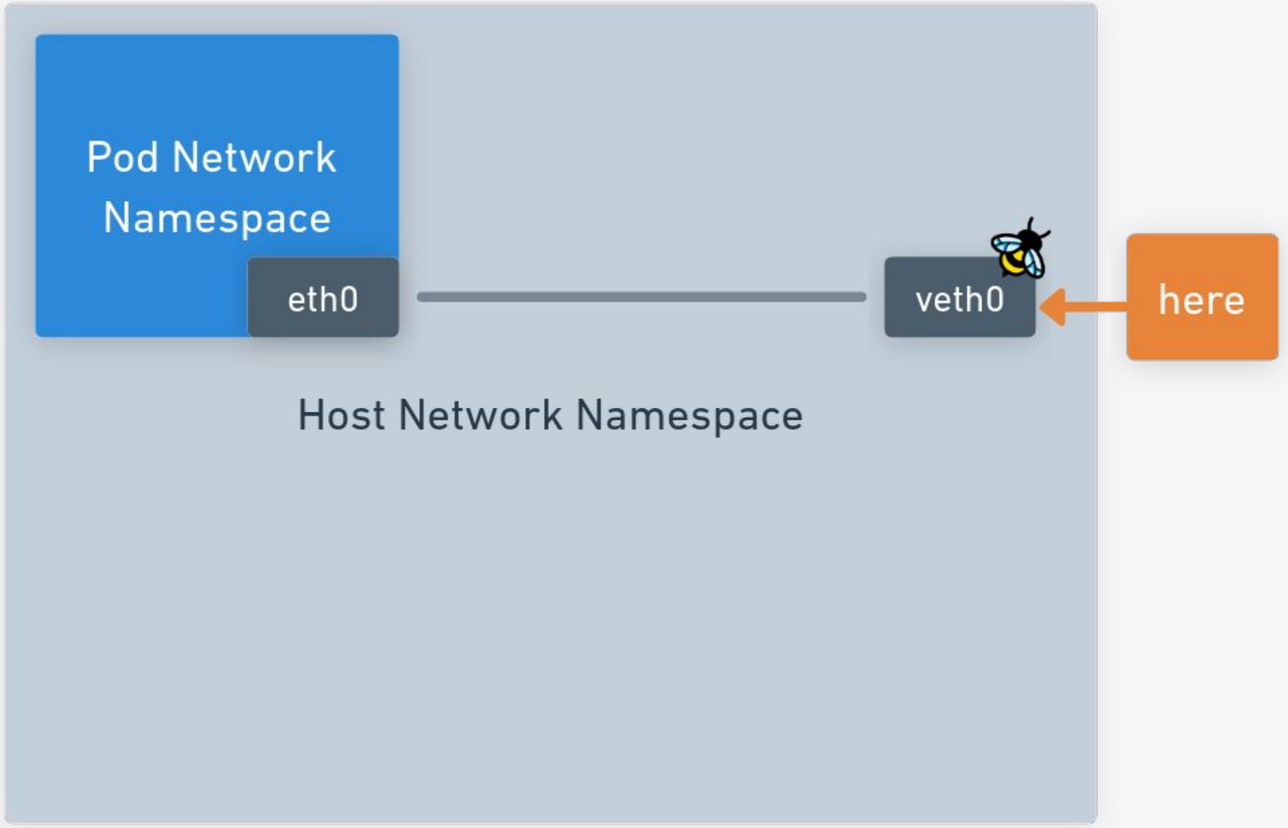
IGMP Snooping Operation

- Listen for IGMP JOIN and LEAVE messages
- On JOIN
  - Lookup associated Group Address in outer map
  - Add subscriber to returned inner map.
- On LEAVE
  - Lookup associated Group Address in outer map
  - Remove subscriber from returned inner map



Where to snoop?

- Cilium installs it's eBPF datapath on the pod's host-side veth.
- This veth is within the host's network namespace.
- Any multicast traffic sent from the pod will arrive here.
- eBPF TC/TCX ingress.



How to snoop?
Step 1: Identify IGMP traffic

- Identify IGMP traffic
- `mcast_ipv4_is_igmp` determines if IPv4 header is IGMP
- `mcast_ipv4_igmp_type` parses out IGMP type.
    - Perform eBPF bounds check, data pointers must be large enough
    - Seek data pointer past IPv4 header and cast it to `struct igmphdr *`
    - Return `hdr->type`

```c
/* returns 1 if ip4 header is followed by an IGMP payload, 0 if not */
static __always_inline bool mcast_ipv4_is_igmp(const struct iphdr *ip4)
{
        if (ip4->protocol == IPPROTO_IGMP)
                return 1;
        return 0;
}
```

```c
/* returns the IGMP type for a given IGMP message
 * a call to 'mcast_ipv4_is_igmp' must be used prior to this call to ensure an
 * igmp message follows the ipv4 header
 */
static __always_inline __s32 mcast_ipv4_igmp_type(const struct iphdr *ip4,
                                                  const void *data,
                                                  const void *data_end)
{
        const struct igmphdr *hdr;
        int ip_len = ip4->ihl * 4;

        if (data + ETH_HLEN + ip_len + sizeof(struct igmphdr) > data_end)
                return DROP_INVALID;

        hdr = data + ETH_HLEN + ip_len;
        return hdr->type;
}
```

How to snoop?
Step 2: Identify IGMP Version

- Cilium only supports IGMPv2 and IGMPv3
- Use `mcast_ipv4_igmp_type` to determine type.

```
#define IGMPV2_HOST_MEMBERSHIP_REPORT   0x16    /* V2 version of 0x12 */
#define IGMP_HOST_LEAVE_MESSAGE         0x17
#define IGMPV3_HOST_MEMBERSHIP_REPORT   0x22    /* V3 version of 0x12 */
```

```
/* ipv4 igmp handler which dispatches to specific igmp message handlers */
static __always_inline __s32 mcast_ipv4_handle_igmp(void *ctx
                                                    struct iphdr *ip4,
                                                    void *data,
                                                    void *data_end)
{
        __s32 igmp_type = mcast_ipv4_igmp_type(ip4, data, data_end);

        if (igmp_type < 0)
                return igmp_type;

        switch (igmp_type) {
        case IGMPV3_HOST_MEMBERSHIP_REPORT:
                return mcast_ipv4_handle_v3_membership_report(ctx,
                                                              &cilium_mcast_group_outer_v4_map,
                                                              ip4,
                                                              data,
                                                              data_end);
        case IGMPV2_HOST_MEMBERSHIP_REPORT:
                return mcast_ipv4_handle_v2_membership_report(ctx,
                                                              &cilium_mcast_group_outer_v4_map,
                                                              ip4,
                                                              data,
                                                              data_end);
        case IGMP_HOST_LEAVE_MESSAGE:
                return mcast_ipv4_handle_igmp_leave(&cilium_mcast_group_outer_v4_map,
                                                    ip4,
                                                    data,
                                                    data_end);
        }

        return DROP_IGMP_HANDLED;
}
```

How to snoop?
Step 3: Parse Membership Report (IGMPv2)

- IGMPv2 is simple
- Preemptively create a subscriber.
- Bounds check on data pointer, must be large enough for `igmphdr`
- Lookup the subscriber map by the Group Address
- Add new subscriber to the returned subscriber map

```
static __always_inline __s32 mcast_ipv4_handle_v2_membership_report(void *ctx,
                                                                    void *group_map,
                                                                    const struct iphdr *ip4,
                                                                    const void *data,
                                                                    const void *data_end)
{
        struct mcast_subscriber_v4 subscriber = {
                .saddr = ip4->saddr,
                .ifindex = ctx_get_ingress_ifindex(ctx)
        };
        int ip_len = ip4->ihl * 4;
        const struct igmphdr *hdr;
        void *sub_map = 0;

        if (data + ETH_HLEN + ip_len + sizeof(struct igmphdr) > data_end)
                return DROP_INVALID;

        hdr = data + ETH_HLEN + ip_len;

        if (hdr->type != IGMPV2_HOST_MEMBERSHIP_REPORT)
                return DROP_INVALID;

        /* lookup user configured multicast group */
        sub_map = map_lookup_elem(group_map, &hdr->group);
        if (!sub_map)
                return DROP_IGMP_HANDLED;

        if (mcast_ipv4_add_subscriber(sub_map, &subscriber))
                return DROP_IGMP_SUBSCRIBED;

        return DROP_IGMP_HANDLED;
}
```

```
/* add a subscriber to a subscriber map */
/* returns 1 on success or DROP_INVALID for error */
static __always_inline __s32 mcast_ipv4_add_subscriber(void *map,
                                                       struct mcast_subscriber_v4 *sub)
{
        if ((map_update_elem(map, &sub->saddr, sub, BPF_ANY) != 0))
                return DROP_INVALID;
        return 1;
}
```

How to snoop?
Step 3: Parse Membership Report (IGMPv3)

- IGMPv3 is more complex
- When a IGMPv3 Membership Report is found format is now `igmpv3_report`
- An `igmpv3_report` has a variable list of `igmpv3_grec` structures
- Each `igmpv3_grec` structure provides a Group Address and a list of sources.
- `igmpv3_grec.type` determines how `src` is interpreted

```
struct igmpv3_grec {
        __u8    grec_type;
        __u8    grec_auxwords;
        __be16  grec_nsrcs;
        __be32  grec_mca;
        __be32  grec_src[];
};

struct igmpv3_report {
        __u8 type;
        __u8 resv1;
        __sum16 csum;
        __be16 resv2;
        __be16 ngrec;
        struct igmpv3_grec grec[];
};
```

How to snoop?
Step 3: Parse Membership Report (IGMPv3)

- `mcast_ipv4_handle_v3_membership_report` preamble start the same at IGMPv2 (not shown).
- Obtain the number of Group Records in the IGMPv3 Membership Report.
- Loop over records, eBPF must have bounded loops.
- Similarly, lookup subscriber map for Group Record's Multicast address
- Handle addition or remove based on `type`
    - When `type` is `CHANGE_TO_EXCLUDE` with 0 source address, this means add the subscriber
    - When `type` is `CHANGE_TO_INCLUDE` with 0 sources, this means remove the subscriber

```
        ngrec = bpf_ntohs(rep->ngrec);

        if (ngrec > MCAST_MAX_GREC)
                return DROP_INVALID;

        /* start a bounded loop which exits when we hit the total number of
         * group records in the membership report.
         *
         * add our subscriber into each group advertised in the report.
         */
#pragma unroll
        for (i = 0; i < MCAST_MAX_GREC; i++) {
                /* Wrap this in an if, instead of breaking out of the loop,
                 * so unroll has a constant number of iterations.
                 *
                 * Compiler was not happy with a continue; statement and the
                 * wrap is necessary.
                 *
                 * remove this when Cilium's min supported kernel version is
                 * >= 5.3 with support for bounded loops.
                 */
                if (i < ngrec) {
                        rec = &rep->grec[i];

                        /* verifier seems to only be happy with a packet bounds check
                         * per iteration
                         */
                        if ((void *)rec + sizeof(struct igmpv3_grec) > data_end)
                                return DROP_INVALID;

                        /* lookup user configured multicast group */
                        sub_map = map_lookup_elem(group_map, &rec->grec_mca);
                        if (!sub_map)
                                continue;

                        /* note:
                         * the datapath currently assumes that no source addresses are
                         * present in the exclude message, indicating a join from all
                         * sources message
                         */
                        if (rec->grec_type == IGMPV3_CHANGE_TO_EXCLUDE) {
                                subscribed = mcast_ipv4_add_subscriber(sub_map, &subscriber);
                                if (subscribed != 1)
                                        return DROP_INVALID;
                                continue;
                        }

                        /* note:
                         * the datapath currently assumes that no source addresses are
                         * present in the include message, indicating a leave from all
                         * sources message
                         */
                        if (rec->grec_type == IGMPV3_CHANGE_TO_INCLUDE)
                                mcast_ipv4_remove_subscriber(sub_map, &subscriber);
                }
        }
        if (subscribed)
                return DROP_IGMP_SUBSCRIBED;

        return DROP_IGMP_HANDLED;
}
```

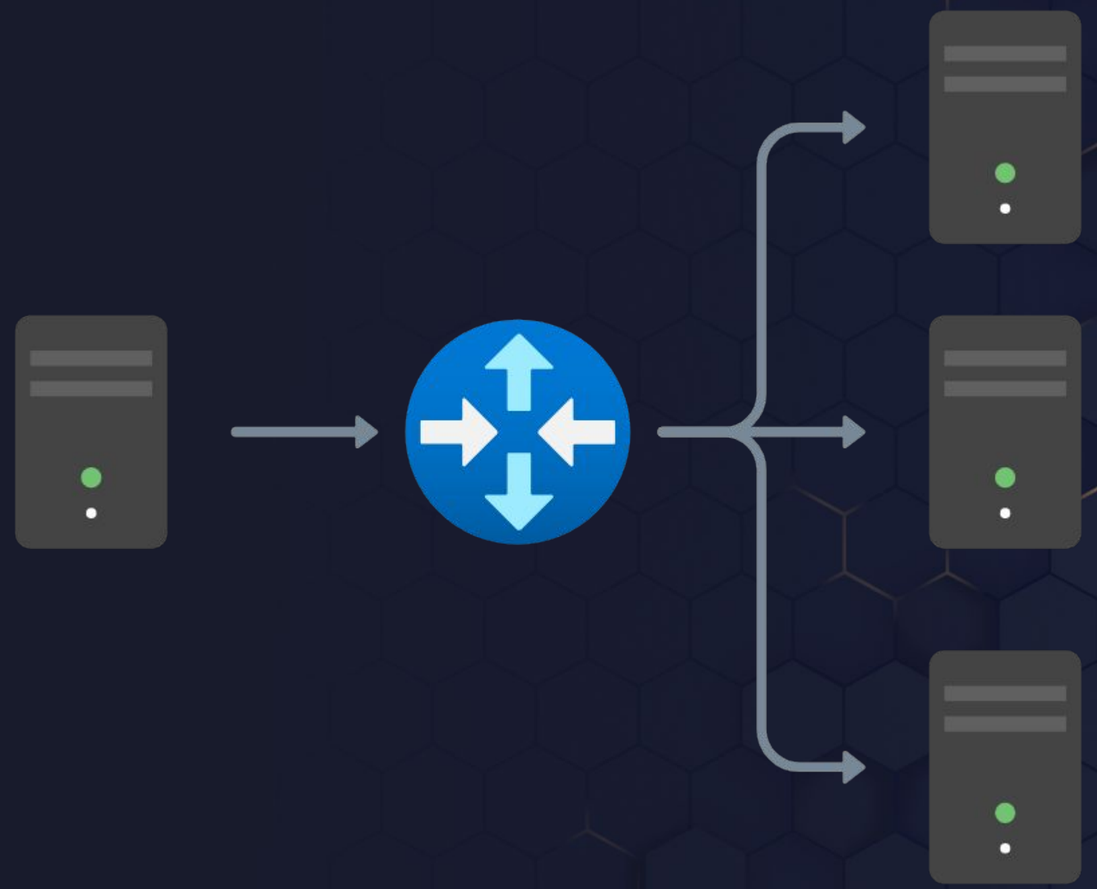
How to snoop?
Step 4: Parse LEAVE

- Simple.
- Look for the LEAVE type (same for all version) and remove our subscriber.

```
static __always_inline __s32 mcast_ipv4_handle_igmp_leave(void *group_map,
                                                          const struct iphdr *ip4,
                                                          const void *data,
                                                          const void *data_end)
{
        struct mcast_subscriber_v4 subscriber = {
                .saddr = ip4->saddr,
        };
        int ip_len = ip4->ihl * 4;
        const struct igmphdr *hdr;
        void *sub_map = 0;

        if (data + ETH_HLEN + ip_len + sizeof(struct igmphdr) > data_end)
                return DROP_INVALID;

        hdr = data + ETH_HLEN + ip_len;

        if (hdr->type != IGMP_HOST_LEAVE_MESSAGE)
                return DROP_INVALID;

        /* lookup user configured multicast group */
        sub_map = map_lookup_elem(group_map, &hdr->group);
        if (!sub_map)
                return DROP_IGMP_HANDLED;

        mcast_ipv4_remove_subscriber(sub_map, &subscriber);

        return DROP_IGMP_HANDLED;
}
```

ISOVALENT

Packet Replication



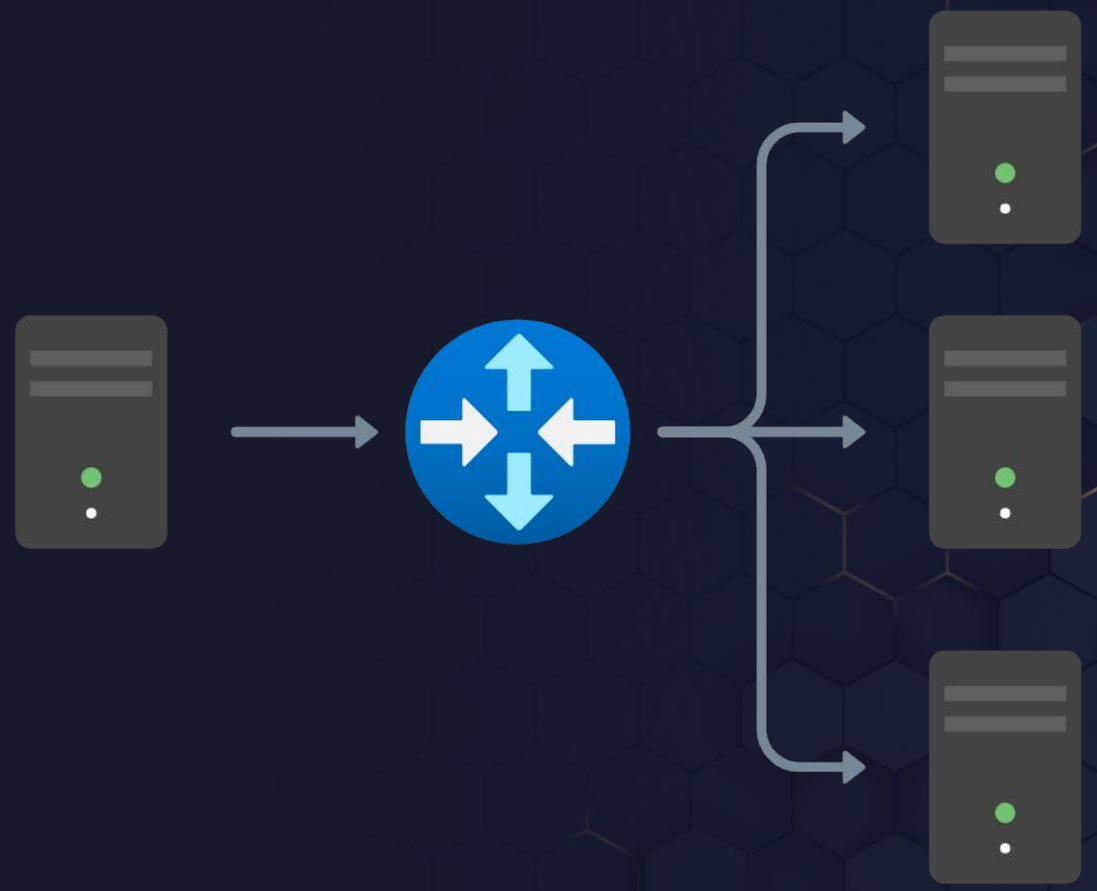
eBPF clone and redirect

- Redirect:
  - Inject an `struct sk_buff` back into the next stack, as if it came from a different interface.
  - Can inject on the ingress side, or egress side.
- Clone:
  - Create a copy of a struct sk_buff which refers to the original data buffer.
- We want to do both and we can!
  - `bpf_clone_redirect` helper
  - Makes a clone of an skb and redirects it another interface.
  - Any modifications desired on the clone must be done first.

```
* long bpf_clone_redirect(struct sk_buff *skb, u32 ifindex, u64 flags)
* >     Description
* >     >       Clone and redirect the packet associated to *skb* to another
* >     >       net device of index *ifindex*. Both ingress and egress
* >     >       interfaces can be used for redirection. The **BPF_F_INGRESS**
* >     >       value in *flags* is used to make the distinction (ingress path
* >     >       is selected if the flag is present, egress path otherwise).
* >     >       This is the only flag supported for now.
*                                                          s/out/inside
* >     >       In comparison with **bpf_redirect**\ () helper,
* >     >       **bpf_clone_redirect**\ () has the associated cost of
* >     >       duplicating the packet buffer, but this can be executed out of
* >     >       the eBPF program. Conversely, **bpf_redirect**\ () is more
* >     >       efficient, but it is handled through an action code where the
* >     >       redirection happens only after the eBPF program has returned.
*
* >     >       A call to this helper is susceptible to change the underlying
* >     >       packet buffer. Therefore, at load time, all checks on pointers
* >     >       previously done by the verifier are invalidated and must be
* >     >       performed again, if the helper is used in combination with
* >     >       direct packet access.
* >     Return
* >     >       0 on success, or a negative error in case of failure. Positive
* >     >       error indicates a potential drop or congestion in the target
* >     >       device. The particular positive error codes are not defined.
*
```

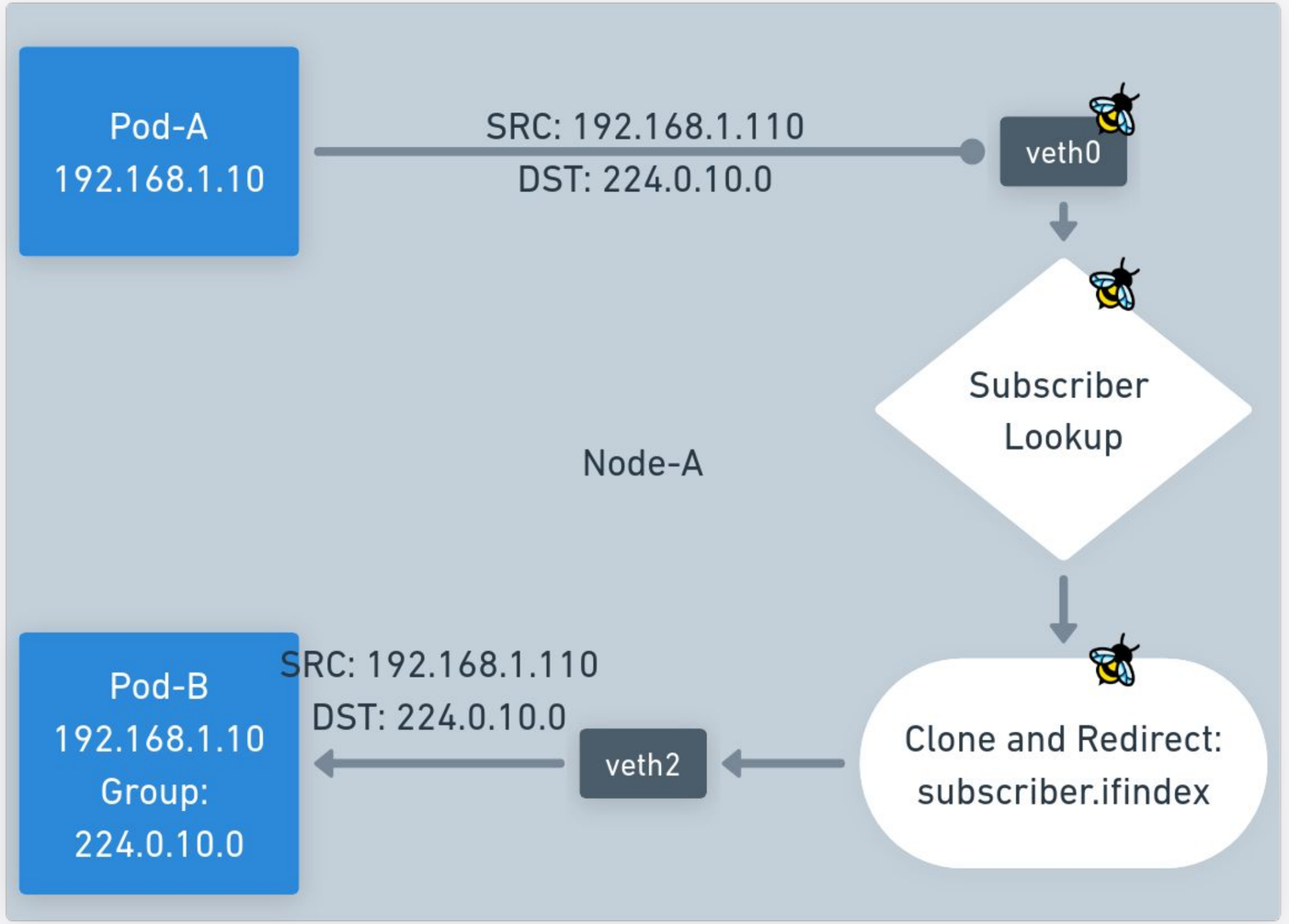
ISOVALENT

Multicast Delivery



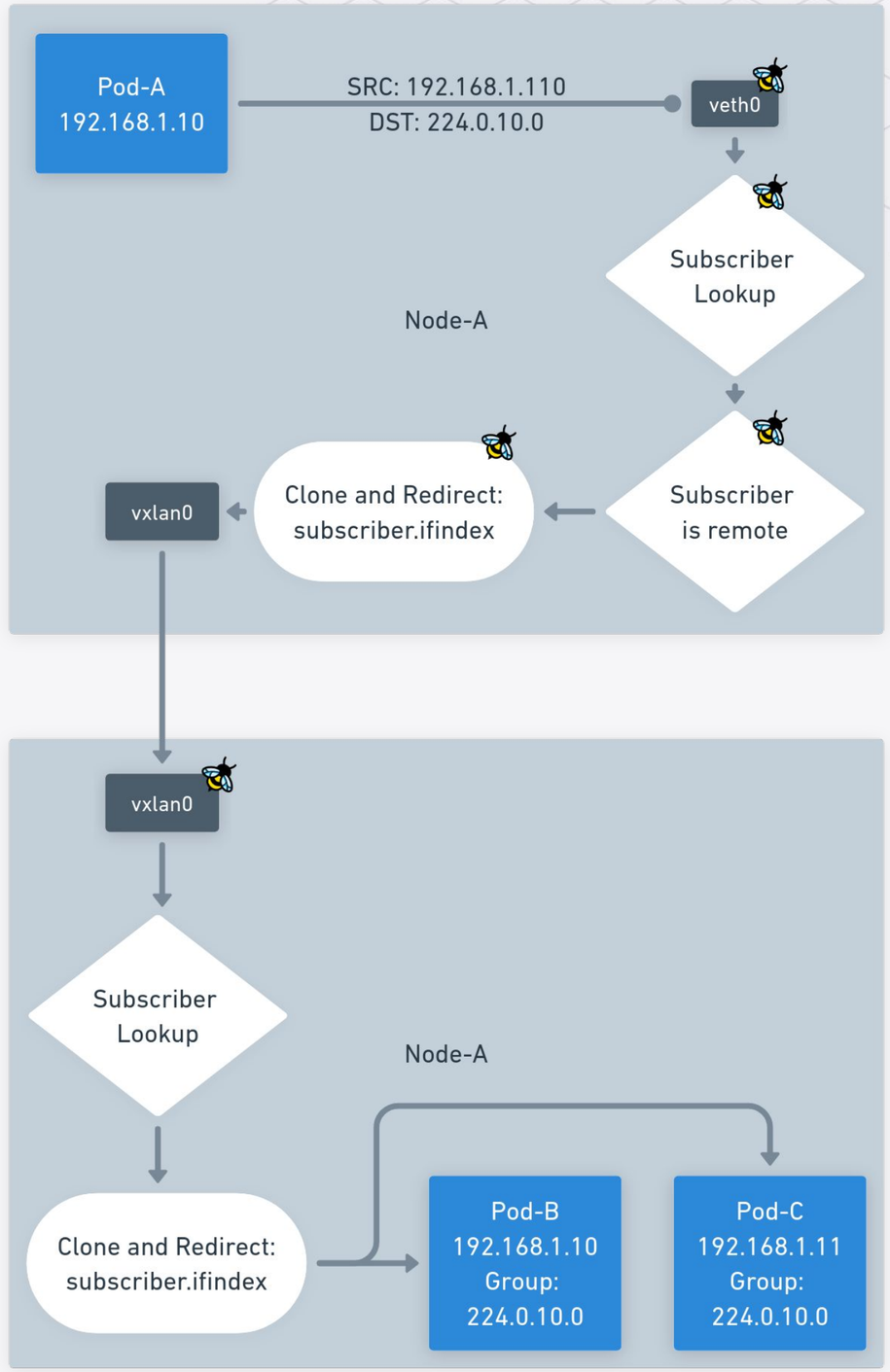
Local Multicast Delivery

- Pod-A send multicast packet to 224.0.10.0 group address.
- Packet arrives at eBPF datapath (ingress@veth0 in host netns)
- eBPF determines packet's DST is a multicast group address
- eBPF performs subscriber lookup for group
    - One (or more) local subscribers are found
    - Clone and redirect to the subscriber's ifindex.



Remote Multicast Delivery

- Use VXLAN to deliver multicast traffic between nodes
    - Required, Linux does not route Multicast without a multicast routing daemon.
- Similar to local delivery but we redirect to the remote node.
- On reception, perform local multicast delivery.



Multicast Delivery with eBPF: Identify Multicast Traffic

- Host-Side Pod's Veth
- `IN_MULTICAST` macro determines of destination address is multicast
    - Checks if address is CLASS D.
- Perform a tailcall to `tail_mcast_ep_delivery`
- `tail_mcast_ep_delivery` implements local and remote delivery

```
#ifdef ENABLE_MULTICAST
        if (mcast_ipv4_is_igmp(ip4)) {
                /* note:
                 * we will always drop IGMP from this point on as we have no
                 * need to forward to the stack
                 */
                return mcast_ipv4_handle_igmp(ctx, ip4, data, data_end);
        }

        if (IN_MULTICAST(bpf_ntohl(ip4->daddr))) {
                if (mcast_lookup_subscriber_map(&ip4->daddr))
                        return tail_call_internal(ctx,
                                                  CILIUM_CALL_MULTICAST_EP_DELIVERY,
                                                  ext_err);
        }
#endif /* ENABLE_MULTICAST */
```

Multicast Delivery with eBPF: Multicast Delivery

- Lookup subscribers.
- Rewrite MAC
- Multicast traffic has specific MAC format.
- `for_each_map_elem` with callback `__mcast_ep_delivery`

```
/* tailcall to perform multicast packet replication and delivery.
 * when this call is entered we should already know that the packet is destined
 * for a multicast group and the multicast group exists in
 * cilium_mcast_group_outer_v4_map
 */
__section_tail(CILIUM_MAP_CALLS, CILIUM_CALL_MULTICAST_EP_DELIVERY)
int tail_mcast_ep_delivery(struct __ctx_buff *ctx)
{
        struct _mcast_ep_delivery_ctx cb_ctx = {
                .ctx = ctx,
                .ret = 0
        };
        union macaddr mac = {0};
        void *data, *data_end;
        struct iphdr *ip4 = 0;
        void *sub_map = 0;

        if (!revalidate_data(ctx, &data, &data_end, &ip4))
                return DROP_INVALID;

        sub_map = map_lookup_elem(&cilium_mcast_group_outer_v4_map, &ip4->daddr);
        if (!sub_map)
                return DROP_INVALID;

        mcast_encode_ipv4_mac(&mac, (__u8 *)&ip4->daddr);

        eth_store_daddr(ctx, &mac.addr[0], 0);

        for_each_map_elem(sub_map, __mcast_ep_delivery, &cb_ctx, 0);

        return send_drop_notify(ctx,
                                UNKNOWN_ID,
                                UNKNOWN_ID,
                                TRACE_EP_ID_UNKNOWN,
                                DROP_MULTICAST_HANDLED,
                                CTX_ACT_DROP,
                                METRIC_INGRESS);
}
```

Multicast Delivery with eBPF: Multicast Delivery Cont...

- Check if we are coming from overlay
  - Avoids a delivery loop
- If subscriber is remote
  - Set tunnel key info
  - `sub->ifindex` will be the vxlan device
- Clone and redirect to `sub->ifindex`
- Same procedure on receiving node but at `vxlan` device

```c
/* performs packet replication and delivery for multicast traffic egressing
 * an endpoint.
 *
 * to be used as a callback function for bpf_for_each_map_elem
 *
 * callback functions must return 1 or 0 to pass eBPF verifier.
 */
static long __mcast_ep_delivery(__maybe_unused void *sub_map,
                                __maybe_unused const __u32 *key,
                                const struct mcast_subscriber_v4 *sub,
                                struct _mcast_ep_delivery_ctx *cb_ctx)
{
        int ret = 0;
        __u32 tunnel_id = WORLD_ID;
        __u8 from_overlay = 0;
        struct bpf_tunnel_key tun_key = {0};

        if (!cb_ctx || !sub)
                return 1;

        if (!cb_ctx->ctx)
                return 1;

        if (!sub->ifindex)
                return 1;

        from_overlay = (ctx_get_ingress_ifindex(cb_ctx->ctx) == ENCAP_IFINDEX);

        /* set tunnel key for remote delivery
         * this helper sets the tunnel metadata on the skb_buff but only
         * tunnel drivers will read it, therefore any local delivery will
         * simply ignore if its present and deliver without an issue.
         *
         * if the ingress interface is set to our tunnel interface, do not
         * perform delivery, this would cause a loop, since the sender's node
         * already delivered to all remote nodes.
         *
         * checking ctx->ingress_ifindex is reliable since
         * __netif_receive_skb_core sets the skb's input interface before
         * calling ingress TC programs.
         */
        if (sub->flags & MCAST_SUB_F_REMOTE) {
                if (from_overlay)
                        return 0;

#ifdef ENABLE_ENCRYPTED_OVERLAY
                /* if encrypted overlay is enabled we'll mark the packet for
                 * encryption via the tunnel ID.
                 */
                tunnel_id = ENCRYPTED_OVERLAY_ID;
#endif /* ENABLE_ENCRYPTED_OVERLAY */
                tun_key.tunnel_id = tunnel_id;
                tun_key.remote_ipv4 = bpf_ntohl(sub->saddr);
                tun_key.tunnel_ttl = IPDEFTTL;

                ret = ctx_set_tunnel_key(cb_ctx->ctx,
                                         &tun_key,
                                         TUNNEL_KEY_WITHOUT_SRC_IP,
                                         BPF_F_ZERO_CSUM_TX);

                if (ret < 0) {
                        cb_ctx->ret = ret;
                        return 1;
                }
        }

        ret = clone_redirect(cb_ctx->ctx, sub->ifindex, 0);
        if (ret != 0) {
                cb_ctx->ret = ret;
                return 1;
        }
        return 0;
};
```

Documentation:

https://docs.cilium.io/en/stable/network/multicast/#enable-multicast

Code:

https://github.com/cilium/cilium/blob/main/bpf/lib/mcast.h