Digital X

# Accelerate your service mesh With eBPF and Merbridge

## Content

## Introduction

### What is Merbridge

- A project uses eBPF technology to accelerate network packets processing in the service mesh scenario
- Blog: https://istio.io/latest/blog/2022/merbridge/

### How does it work?

- Shorten the data-path between data-plane’s sidecars and services, and allow packets to be transported directly from one socket to another
- Simulate and replace what iptables does to the mesh

## Why design Merbridge

- eBPF’s popularity and community acceptance
- A new perspective on reducing the network latency
- No open-sourced solution to be used directly yet
- Promote the development of the community with the technology accumulation from our company

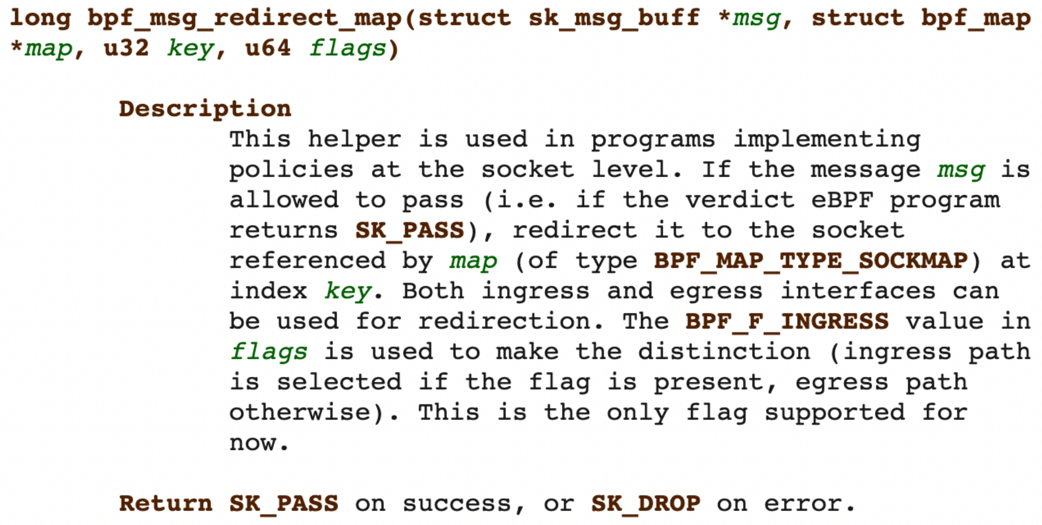
## Benefits

- Optimize the long-standing request latency issue
- One line Installation without modifying the mesh
- Compatibility to different mesh projects (Istio, Linkerd)

## Some Approaches

## How to shorten the data-path

- eBPF provides a function [bpf_msg_redirect_hash] to process policys at the socket level (https://man7.org/linux/man-pages/man7/bpf-helpers.7.html)
- After processing, the packet will be redirected to the destination socket referenced from the sock map
- But how to process…



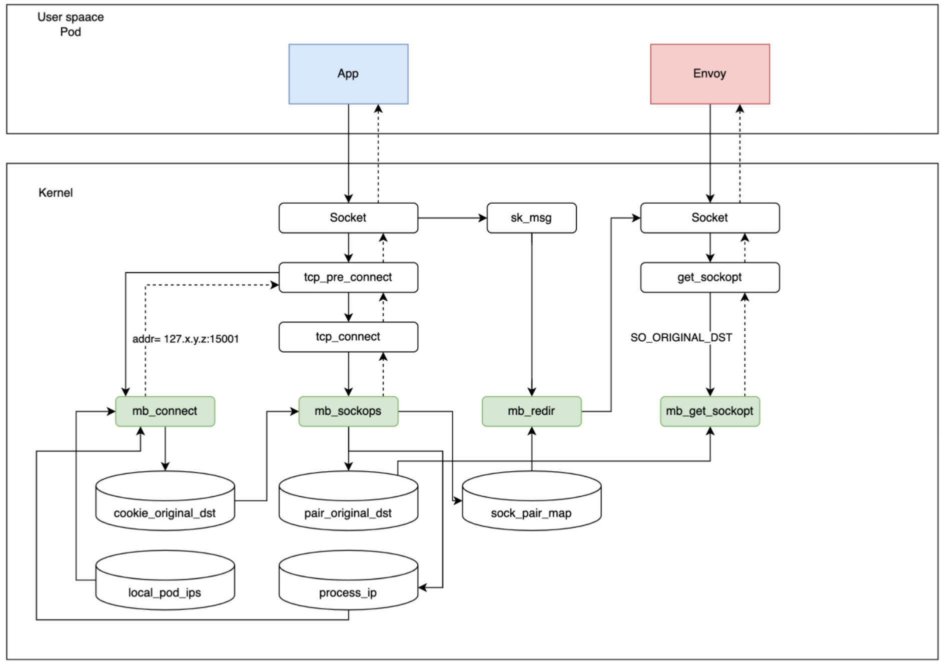
```
long bpf_msg_redirect_map(struct sk_msg_buff *msg, struct bpf_map
*map, u32 key, u64 flags)

       Description
              This helper is used in programs implementing
              policies at the socket level. If the message msg is
              allowed to pass (i.e. if the verdict eBPF program
              returns SK_PASS), redirect it to the socket
              referenced by map (of type BPF_MAP_TYPE_SOCKMAP) at
              index key. 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.

       Return SK_PASS on success, or SK_DROP on error.
```

## How to shorten the data-path

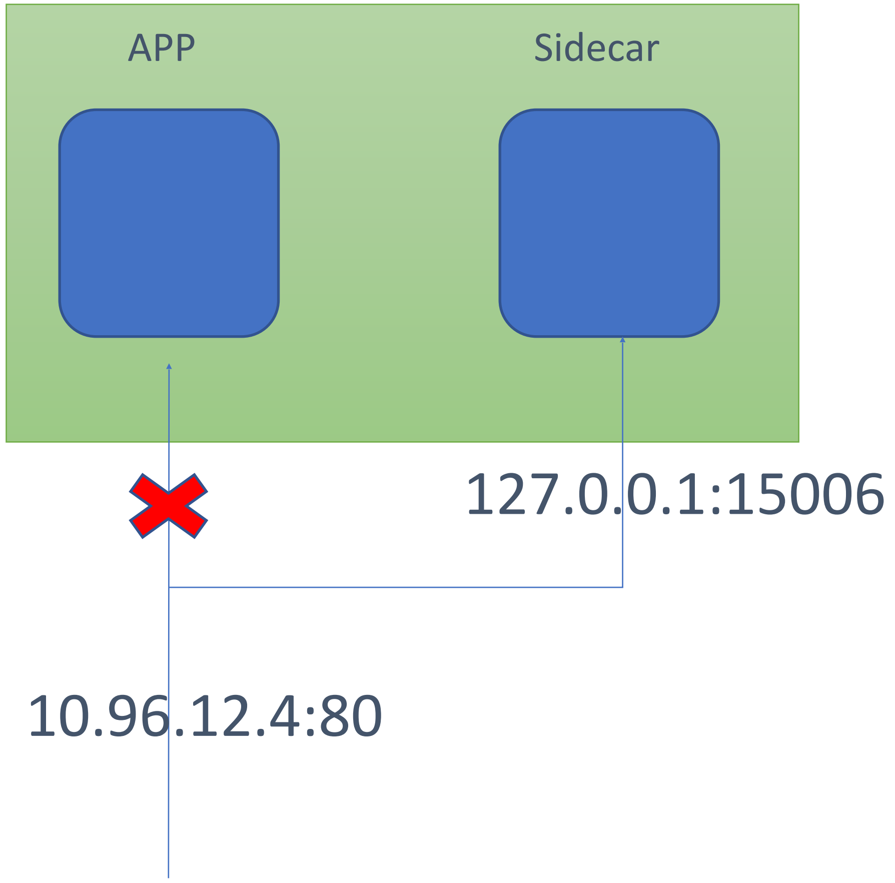
- eBPF provides a function [bpf_msg_redirect_hash] to process policys at the socket level (https://man7.org/linux/man-pages/man7/bpf-helpers.7.html)
- After processing, the packet will be redirected to the destination socket referenced from the sock map
- But how to process…

**Simulate what iptables does to Istio**

https://istio.io/latest/blog/2022/merbridge/

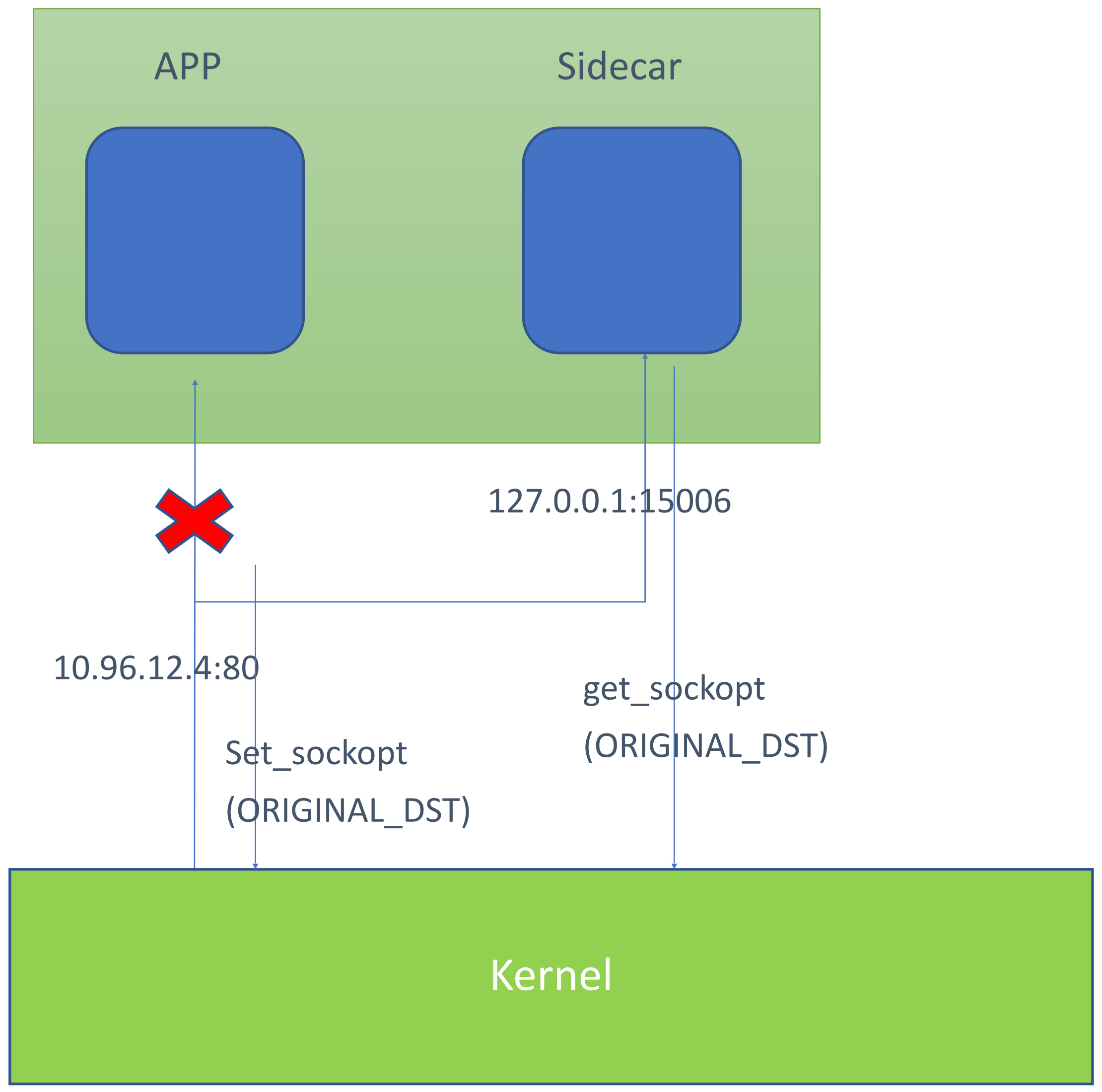


## How to find the original destination



- Istio manipulates Iptables to forward traffic to proxy inbound, and in this case the sidecar container will see the the quadruple (source IP, source port, 127.0.0.1, 15006).
- Istio requires the destination address to determine the service it will send requests to, but in this case the destination information is missed.

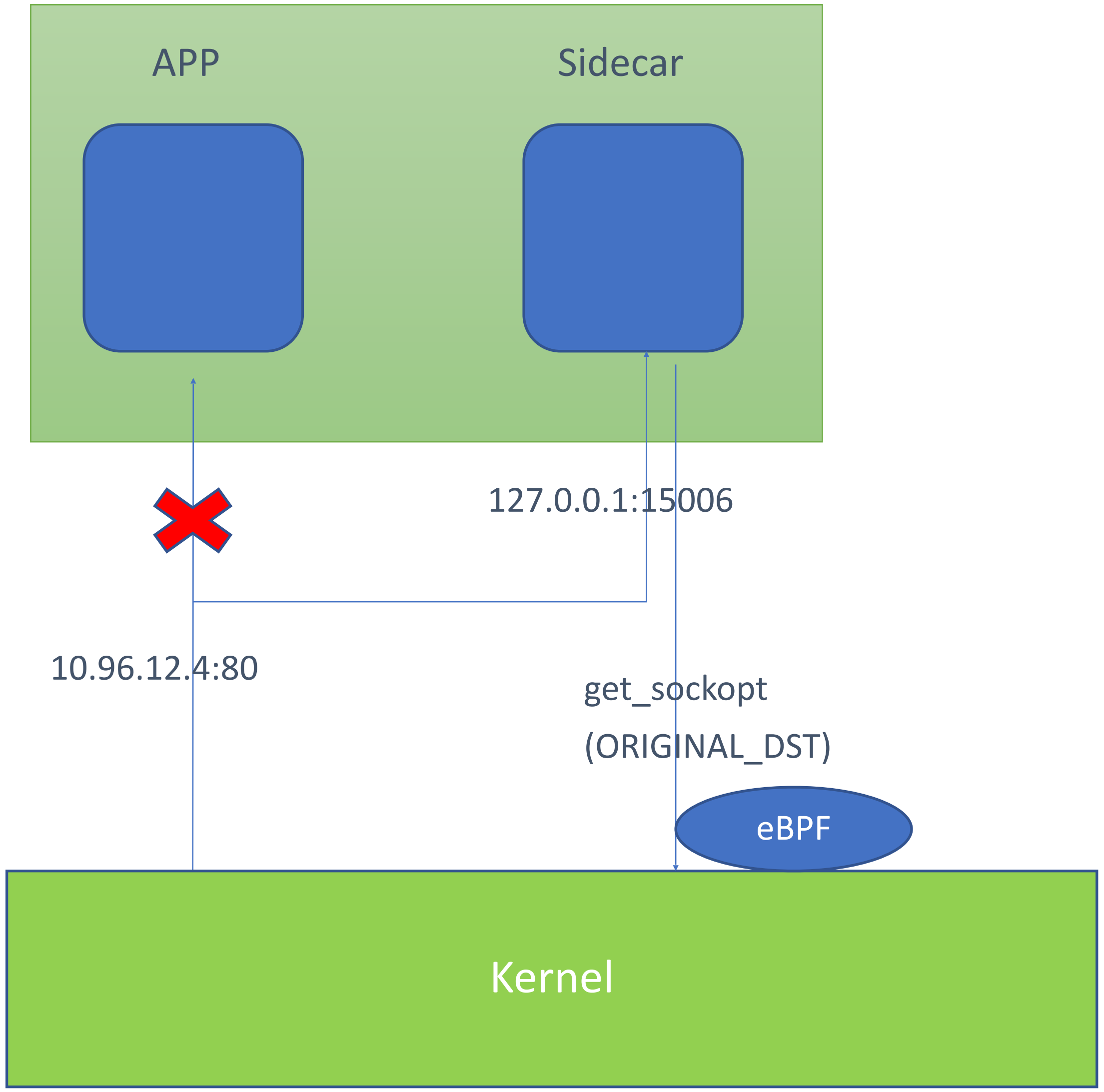
# How to find the original destination



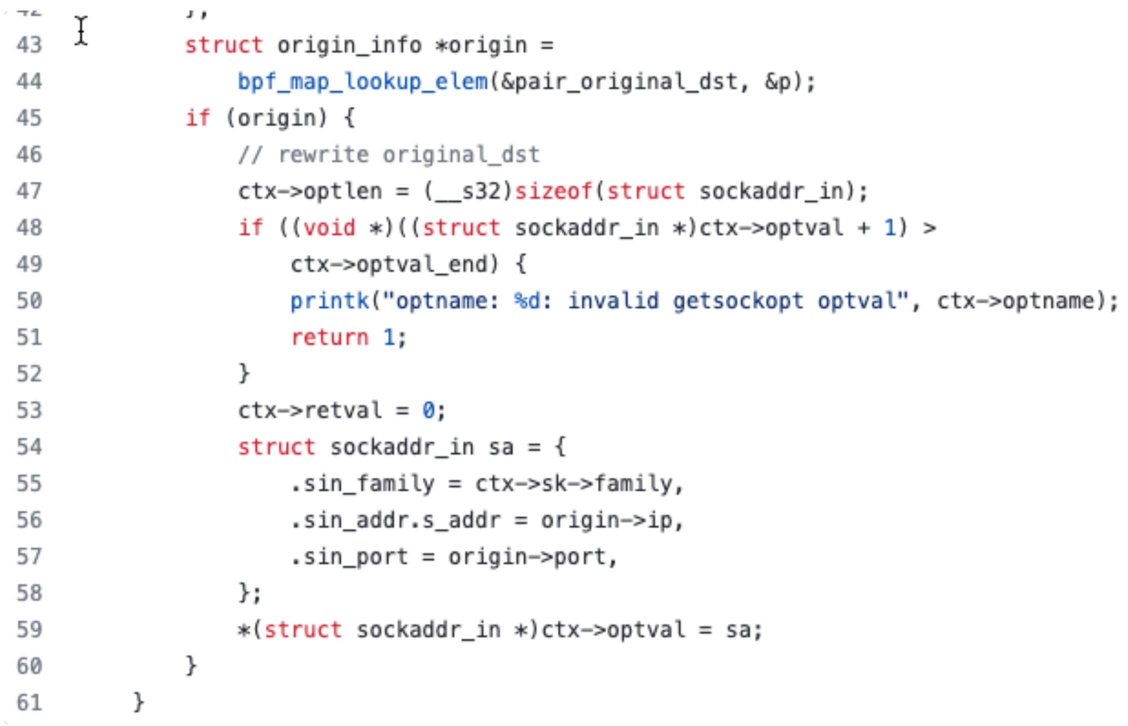
For the iptables REDIRECT process, it will call set_sockopt to save the ORIGINAL_DST to its socket options in the sock map, and later called by get_sockopt.

However, eBPF cannot do such process since it's dedicated to the netfilter module.

## How to find the original destination

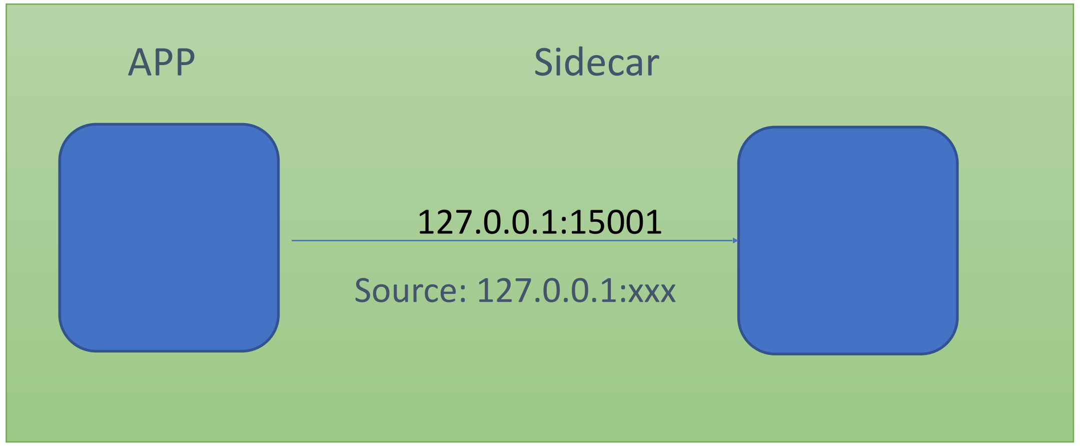


Intercept get_sockopt function call, and rewrite ORIGINAL_DST with the actual original destination



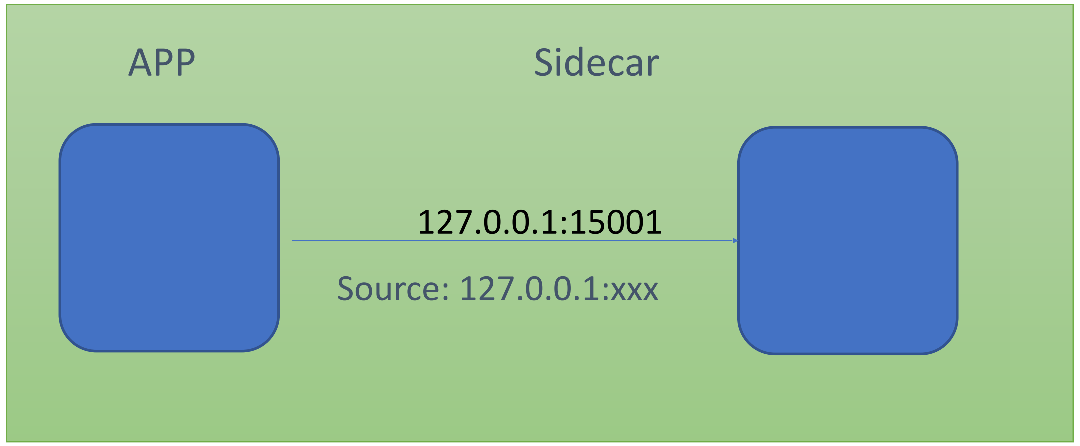
```
        struct origin_info *origin =
            bpf_map_lookup_elem(&pair_original_dst, &p);
        if (origin) {
            // rewrite original_dst
            ctx->optlen = (__s32)sizeof(struct sockaddr_in);
            if ((void *)((struct sockaddr_in *)ctx->optval + 1) >
                ctx->optval_end) {
                printk("optname: %d: invalid getsockopt optval", ctx->optname);
                return 1;
            }
            ctx->retval = 0;
            struct sockaddr_in sa = {
                .sin_family = ctx->sk->family,
                .sin_addr.s_addr = origin->ip,
                .sin_port = origin->port,
            };
            *(struct sockaddr_in *)ctx->optval = sa;
        }
    }
```

## Resolve quadruple conflicts



Why conflicts

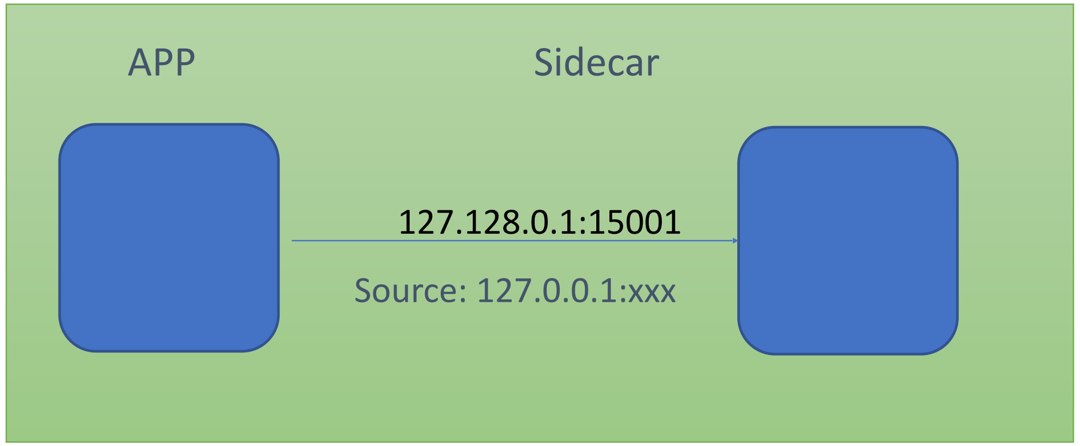
- When sending requests, the destination address will be redirected to 127.0.0.1:15001
- Pods share the kernel space, but eBPF works on the entire kernel space level
- Pods‘ local ports are independent



Potential solutions

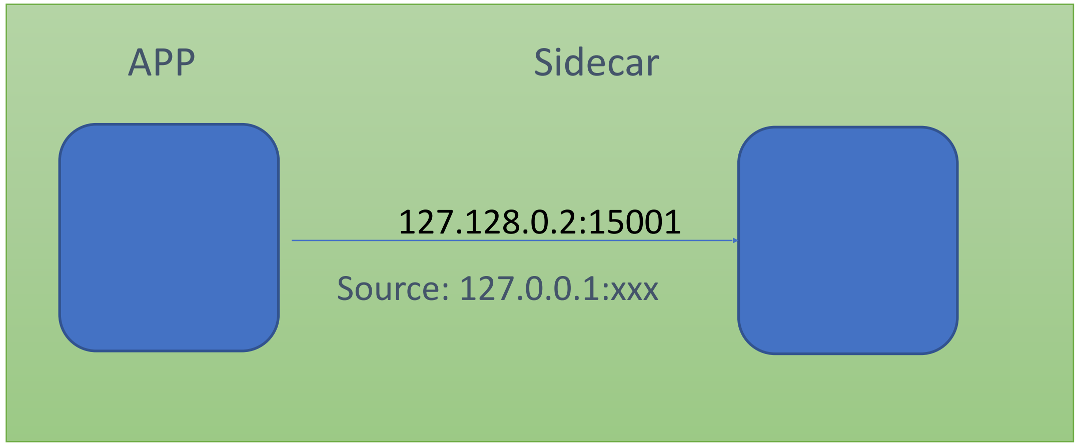
- Revise Istio code to provide different policys (not maintainable in the future)
- Patch kernel with related functions (requires high kernel version)

# Resolve quadruple confliction



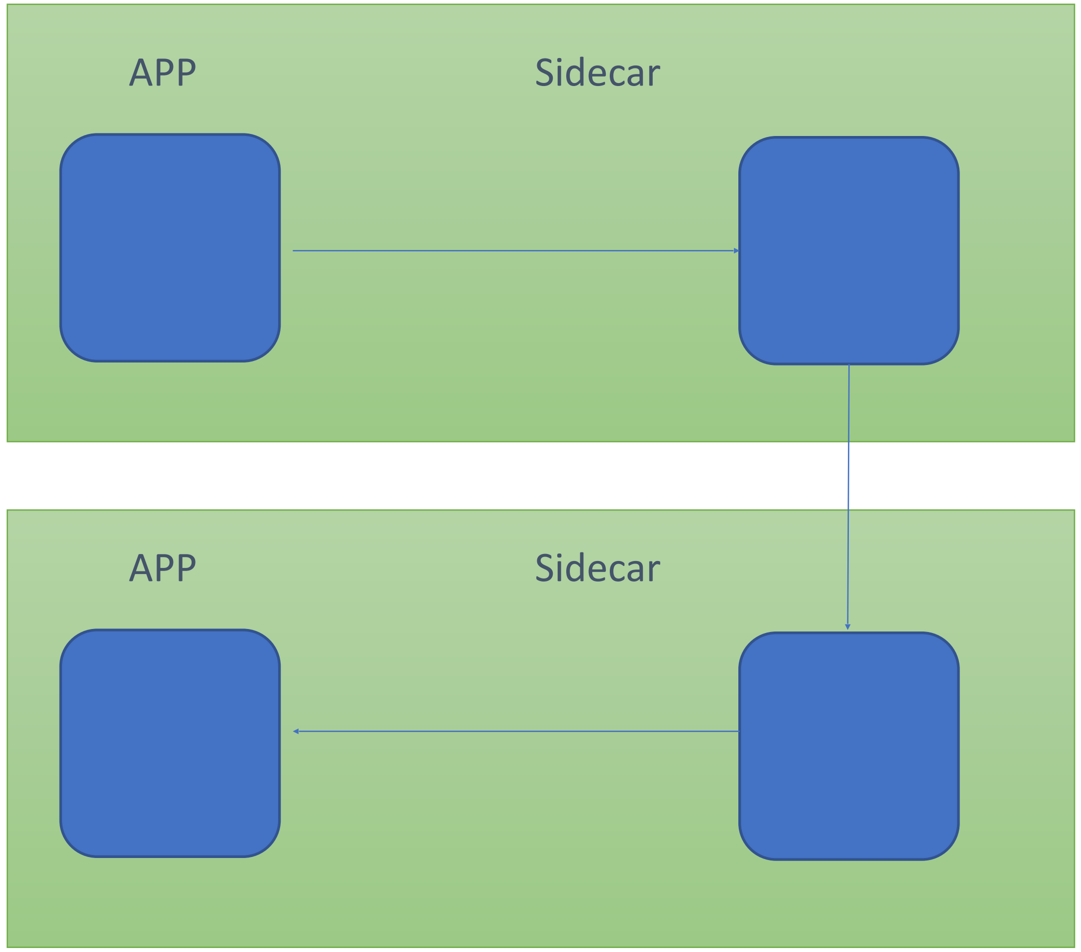
In the quadruple, the source information, and the destination port seems to be unable to change.

Destination IP is the best part to be revised.



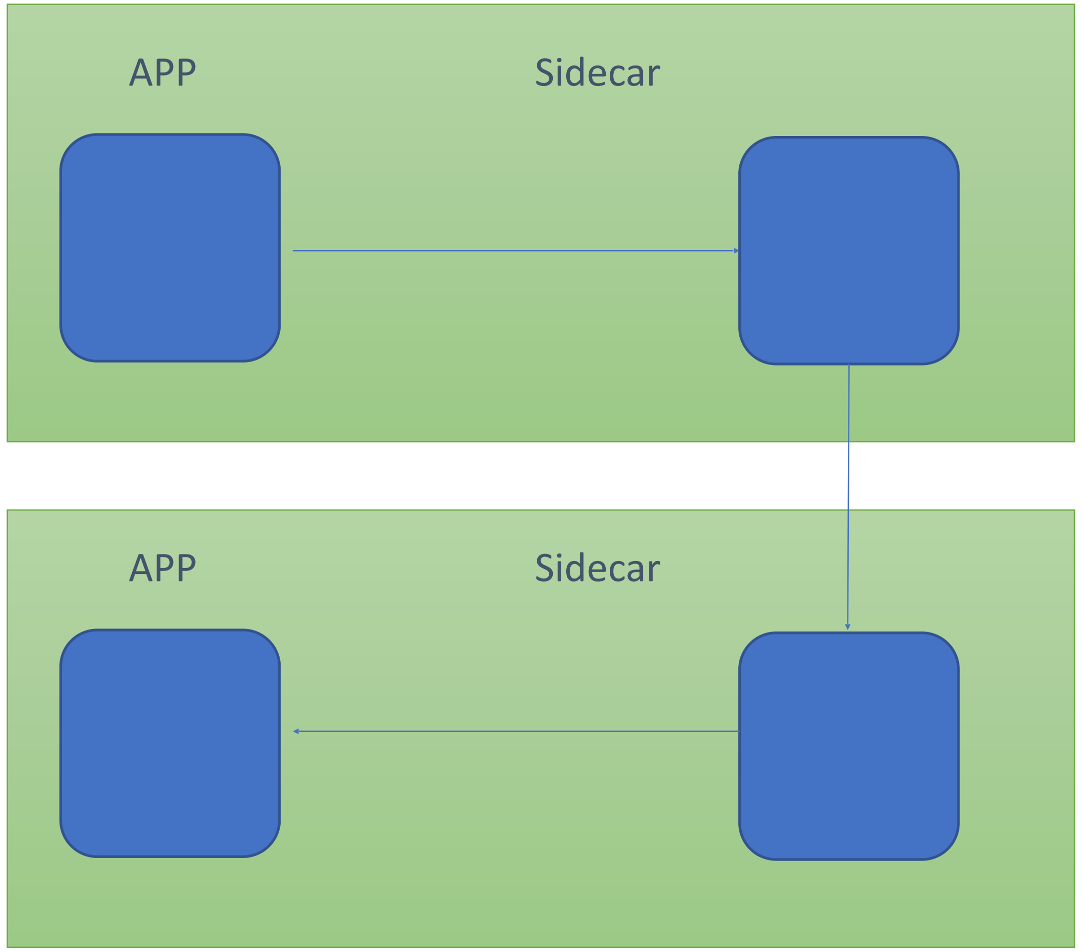
Solution:
Replace 127.0.0.1 with dynamic IP allocation 127.128.x.x.
When a new connection is established, the IP address will be changed as well

## Same node acceleration



Without iptables, when to perform the redirection is the problem. For example, the source request cannot be redirected if the destination is on another node, or the public network.

## Same node acceleration



Thoughts:
Whether or not to do the redirection, to know all the pods in the current node is enough. However, to avoid redirecting to pods internal IP, we also needs to know the current context IP.

Solution:
Add a control plane to synchronize the IP address table of the current node that has been managed by the service mesh.
Establish the corresponding table of process ID + IP address through the feedback mechanism.

## Future plan

- Metrics
- Cross-node mode
- IPv6 support
- Istio sidecar annotations

https://github.com/merbridge/merbridge/blob/main/ROADMAP.md

## Demo

# Thanks!