Container Networking: The Play of BPF & Network NS with different Virtual Devices

Takshak Chahande & Martin KaFai Lau
Agenda

01 L3-Level networking (cgroup-bpf)

02 Building solution with Network Namespaces

03 How BPF helped to work around challenges

04 Performance data on different net devices solution
01 Building solution without Network Namespaces
Linux Containers?

![Diagram of Linux Containers]

- **Namespaces**
  - PIDs
  - MNTs

- **Cgroups**
  - PIDs
  - MNTs

- **Containers**

**Kernel**

**Hardware**
Linux Containers

- Container shapes
  - Square Shapes
  - L-Shape
- Host Accessibility
  - Single Tenant
  - Multi-Tenant Host
- Container Isolation
  - Resource Isolations
  - Shared Resources
Linux Containers: Shared-Network Resource

- Container1 & Container2 both share the host-network namespace
- No extra-network configuration setup
Shared-Network Resource: Port mgmt

```
lo

eth0

bind("::"/*wildcard*/, 443)
```

Container1

- PID namespace
- User namespace
- Mount namespace
- /sys/fs/cgroup/container1-cgroup/

Container2

- PID namespace
- User namespace
- Mount namespace
- /sys/fs/cgroup/container2-cgroup/
Coordinating between multiple containers for port-management
Shared-Network Resource : Port mgmt

- Coordinating between multiple containers for port-management
- Binding to localhost is exposed to other containers & host
Shared-Network Resource : Service traffic mgmt

- DC Network
- Rack Switch (RSW)
- Fixed Uplink Capacity
- Marking traffic at source (QoS)

- eth0
- lo

Physical-Host

- Container1
  - PID namespace
  - User namespace
  - Mount namespace
  - /sys/fs/cgroup/container1-cgroup/

- Container2
  - PID namespace
  - User namespace
  - Mount namespace
  - /sys/fs/cgroup/container2-cgroup/
Shared-Network Resource: Service traffic mgmt

DC Network

Rack Switch (RSW)

Fixed Uplink Capacity

Marking traffic at source (QoS)

Identify Service and apply network Policies?

Physical-Host

eth0

lo

Container1

PID namespace

User namespace

Mount namespace

/sys/fs/cgroup/container1-cgroup/

Container2

PID namespace

User namespace

Mount namespace

/sys/fs/cgroup/container2-cgroup/

/eBPF

Marking traffic at source (QoS)
Shared-Network Resource: Service traffic mgmt

- **Secure Network**: Which service is allowed to pass/deny?

- **Fixed Uplink Capacity**: Rack Switch (RSW)

- **Troubleshoot the traffic**: Which container originated the traffic from (Host IPv6)?

---

**Physical-Host**

- eth0
- lo

- **Container1**
  - PID namespace
  - User namespace
  - Mount namespace
  - `/sys/fs/cgroup/container1-cgroup/`

- **Container2**
  - PID namespace
  - User namespace
  - Mount namespace
  - `/sys/fs/cgroup/container2-cgroup/`
Need network identifier to each container and some level of isolation?
We decided to give unique IPv6 identity to each container
We decided to give unique IPv6 identity to each container

How to tie this IPv6 identity to the container?
We decided to give unique IPv6 identity to each container

How to tie this IPv6 identity to the container?

cgroup-bpf : bind/connect/sendmsg
/* User bpf_sock_addr struct to access socket fields and sockaddr struct passed
* by user and intended to be used by socket (e.g. to bind to, depends on
* attach type).
*/

struct bpf_sock_addr {
    __u32 user_family; /* Allows 4-byte read, but no write. */
    __u32 user_ip4;   /* Allows 1,2,4-byte read and 4-byte write.
                     * Stored in network byte order.
                     */
    __u32 user_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte write.
                         * Stored in network byte order.
                         */
    __u32 user_port;  /* Allows 1,2,4-byte read and 4-byte write.
                        * Stored in network byte order.
                        */
    __u32 family;     /* Allows 4-byte read, but no write */
    __u32 type;       /* Allows 4-byte read, but no write */
    __u32 protocol;   /* Allows 4-byte read, but no write */
    __u32 msg_src_ip4; /* Allows 1,2,4-byte read and 4-byte write.
                        * Stored in network byte order.
                        */
    __u32 msg_src_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte write.
                           * Stored in network byte order.
                           */
    __bpf_md_ptr(struct bpf_sock *, sk);
};
Play : Same Host-Network-namespace

/* User bpf_sock_addr struct to access socket fields and sockaddr
struct passed
* by user and intended to be used by socket (e.g. to bind to,
depends on
* attach type).
*/

struct bpf_sock_addr {
    __u32 user_family; /* Allows 4-byte read, but no write. */
    __u32 user_ip4; /* Allows 1,2,4-byte read and 4-byte write.
* Stored in network byte order.
*/
    __u32 user_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte
write.
* Stored in network byte order.
*/
    __u32 user_port; /* Allows 1,2,4-byte read and 4-byte write.
* Stored in network byte order
*/
    __u32 family; /* Allows 4-byte read, but no write */
    __u32 type; /* Allows 4-byte read, but no write */
    __u32 protocol; /* Allows 4-byte read, but no write */
    __u32 msg_src_ip4; /* Allows 1,2,4-byte read and 4-byte write.
* Stored in network byte order.
*/
    __u32 msg_src_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte
write.
* Stored in network byte order.
*/
    __bpf_md_ptr(struct bpf_sock *, sk);
};
Play: Same Host-Network-namespace

- Re-write the destination to container IP
- Set the source identity for the outgoing traffic

```c
sa.sin6_family = AF_INET6;
sa.sin6_port = bpf_htons(0);
in6cpy(&sa.sin6_addr, task_ip);

/* Rewrite source IP. */
if (bpf_bind(ctx, (struct sockaddr*)&sa, sizeof(sa)) != 0)
    return FAIL_OPEN;
```

Acknowledgement: Andrey Ignatov and Alexei S
Shared host-network space: Re-look challenges

→ Two containers cannot start service on the same fixed port
   - Unique IP per container helps at certain extent
   - Fails if the same port binds on the wildcard by other host-based services

→ Container localhost service gets exposed to host & whole meta
   - Services bind on container-IP which is routable in Meta fleet
   - eBPF helps but still adds an additional overhead to handle it

→ Does not allow wildcard binding inside the container (hacks for additional VIPs)
   - Hard to share the same port among container IP and BGP VIPs
Traffic Redirection over TLS

- Using eBPF hooks with socket cookies, it is easy to track TCP connections.
- For UDP sockets, where the same source IP:port can be used for multiple destinations; proxy can’t track the connections.
  - Packet encapsulation helps to solve this but that requires tc-bpf based solution.
  - Moving every container’s UDP traffic tracking at host-eth0 is again a challenge in multi-tenant host.
Process VM inside the container

Tw container

VM

eth0

tap0

HostNS

ttls-forward-proxy

Task Cgroup

FW egress tc-bpf
TTLS NAT 646 tc-bpf
TTLS egress redirection tc-bpf

FW ingress tc-bpf
TTLS NAT 464 tc-bpf
TTLS ingress redirection tc-bpf
Process VM inside the container

Stacking large number of containers, makes life hard to manage shared host resources.
Multi-NIC Host: Routing Mgmt (Ingress/Egress)

- Rack Switch (RSW)
- Physical-Host
  - lo
  - eth0
  - eth1
  - eth2
  - eth3
- Container1
  - /sys/fs/cgroup/container1-cgroup/
- Container2
  - /sys/fs/cgroup/container2-cgroup/
Multi-NIC Host : Routing Mgmt

Rack Switch (RSW)

BGP multi nexthop route1

BGP multi nexthop route2

Physical-Host

lo

eth0

eth1

eth2

eth3

ECMP based Multipath Routing eth0/eth1

ECMP based Multipath Routing eth2/eth3

Container1

/sys/fs/cgroup/container1-cgroup/

BGP multi nexthop route1

Container2

/sys/fs/cgroup/container2-cgroup/
Apart from the gaps, other use-cases?

- L2 level secure isolation to avoid all enforcement at host level
- per-container tc/XDP eBPF support again to avoid physical-eth0 a choking point
- Some of the emulated services need IPv4 support
- Running third-party services/applications with jailed environment
- Debug the container level traffic without having access to host
02 Building solution with Network Namespaces
Finally Network NS: Network Connection model

Virtual Ethernet (VETH)

Host network space

Container-1

Routing process

Container-2
Network Namespaces: Build network connectivity

Host network space

Upper Stack (IP, Netfilter, Routing..)

eth0

task-netns

task-veth-0

host-veth-0
Network Namespaces: Build network connectivity

- Global IP Forwarding enablement
- `veth(4)` is slower due to additional traversal of network stack
03 How bpf helped to work around the issue
Network Namespaces: Use of eBPF Kernel extensions

bpf: Add redirect_neigh helper as redirect drop-in

Add a redirect_neigh() helper as redirect() drop-in replacement for the xmit side. Main idea for the helper is to be very similar in semantics to the latter just that the skb gets injected into the neighboring subsystem in order to let the stack do the work it knows best anyway to populate the L2 addresses of the packet and then hand over to dev_queue_xmit() as redirect() does.

This solves two bigger items: i) skbs don't need to go up to the stack on the host facing veth ingress side for traffic egressing the container to achieve the same for populating L2 which also has the huge advantage that ii) the skb->sk won't get orphaned in ip_rcv_core() when entering the IP routing layer on the host stack.
Network Namespaces: Use of eBPF Kernel extensions

- Host network space
- eth0
- Upper Stack (IP, Netfilter, Routing..)
- task-netns
- task-veth-0
- host-veth-0
- Network Namespaces: Use of eBPF Kernel extensions

[Diagram showing network namespaces and eBPF kernel extensions]
Network Namespaces: Build network connectivity

- Global IP Forwarding enablement
- veth(4) is slower due to additional traversal of network stack
UDP Traffic Redirection over TLS

- Challenges
  - Packet makes a round trip from veth0-egress to host-end and back to task’s netns
  - Ingress program at veth0 is in-effective
UDP Traffic Redirection over TLS

- Challenges
  - Packet makes a round trip from veth0-egress to host-end and back to task’s netns
  - Ingress program at veth0 is in-effective for ttl-fwd-proxy → user client.

- Current Solution:
  - Ingress program needed to attach at “lo” due to kernel optimizing the route.
  - `bpf_redirect("eth0" → ifindex, BPF_F_INGRESS) & update MAC`
  - Change direction from EGRESS to INGRESS
04 Other Virtual Devices & Performance Improvements
ipvlan/veth/netkit/bare-metal
veth -> netkit (cpu-util)
veth -> netkit (cpu-sys + cpu-softirq)
veth -> netkit (cpu-softirq)
veth -> netkit (cpu-sys)
egress veth/netkit => phy-eth0
bpf prog at netkit
netkit at L2 mode (cpu-sys + softirq)
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 x task</td>
<td>no netns (bare metal)</td>
<td>ipvlan L2</td>
<td>meta L2 (no fib lookup)</td>
<td>veth L2 (no fib lookup)</td>
</tr>
<tr>
<td>2</td>
<td>cpu-util</td>
<td>84.78</td>
<td>85.19</td>
<td>85.85</td>
<td>87.09</td>
</tr>
<tr>
<td>3</td>
<td>cpu-softirq</td>
<td>2.48</td>
<td>2.85</td>
<td>2.17</td>
<td>14.78</td>
</tr>
<tr>
<td>4</td>
<td>cpu-sys</td>
<td>63.24</td>
<td>63.73</td>
<td>64.79</td>
<td>53.89</td>
</tr>
<tr>
<td>5</td>
<td>cpu-user</td>
<td>18.46</td>
<td>18.09</td>
<td>18.38</td>
<td>17.89</td>
</tr>
<tr>
<td>6</td>
<td>Transactions (M) / s</td>
<td>1.63</td>
<td>1.61</td>
<td>1.62</td>
<td>1.62</td>
</tr>
<tr>
<td>7</td>
<td>trans_per_s (K)/ cpu%</td>
<td>19.17</td>
<td>18.90</td>
<td>18.88</td>
<td>18.61</td>
</tr>
</tbody>
</table>
Ipvlan vs meta (L2)
ipvlan vs ipvlan (background difference)