Containers Do Not Need Network Stacks

Ryo Nakamura
iijlab seminar 2018/10/16

Containers

- A package of an application execution environment
  - version-controllable
  - portable
  - lightweight

- Microservice architecture
  - An application (service) runs on a container
  - Multiple containers comprise a system
The beginning of container networking

• A container is a separated namespace in a host OS
  • Containers need to connect to other containers, host, and external networks
• The conventional approach: Adapters and Links
  • *Virtual NICs* (veth interface in Linux)
Overhead of container networking

- Container involves
  - virtual NIC (veth)
  - virtual bridge and NAT (docker0) in the host network stack

- Network performance degradation
  - degrade throughput by 50%
  - increase latency by 25%
The long data path

• from an application in container to NIC
  • Time to transmit a packet increases
  ➢ Throughput and latency are degraded
State-of-the-art container networking

1. Interface Virtualization
   • Directly attaching interfaces to containers (bypassing host network stack)
   • macvlan, SR-IOV

2. Optimized Network Stacks
   • Reinventing the entire or a part of network stacks
   • FreeFlow[1], Cilium[2]

State-of-the-art: Interface Virtualization

- Bypassing the host network stack
  - macvlan achieves comparable network performance with native host[3]

- Complicating management
  - Outer networks must manage container networks
    - addressing, tenant separation, access control, etc
  - NAT conceals container networks from outer networks and infrastructures

State-of-the-art: Optimized network stacks

• Using high-speed packet I/O techniques
  • FreeFlow uses DPDK and RDMA
  • Cilium uses XDP (eBPF)
State-of-the-art: Optimized network stacks

• Using high-speed packet I/O techniques
  • FreeFlow uses DPDK and RDMA
  • Cilium uses XDP (eBPF)

• The long data path will be the next bottleneck
  • Protocol processing cost do not disappear
    • In Arrakis OS[4], network protocol processing occupies 100% of processing cost on a simple UDP echo server
  • It will be more significant bottleneck in comparison with native hosts

The third approach: Bypassing container network stacks

• A container is
  • just an application execution environment
  • not interested in how packets are delivered

Then, we can bypass container network stacks to mitigate the overhead?
A question: Do containers *really* need network stacks?

Port forward:
```
docker -p 80:80
```

**Container has network stack**

**Container does not have network stack**
A question: Do containers really need network stacks?

Port forward docker -p 80:80

Container has network stack

Container does not have network stack

No differences
The third approach: Bypassing container network stacks, cont’d

• A container is
  • just an application execution environment
  • not interested in how packets are delivered
  • Then, *we can bypass container network stacks*

• Network stack separation should be retained
  • `docker run --net=host` can cause unintended or malicious resource uses
    • address, port, protocol, etc

• A new mechanism is needed
  • connecting App on a container to the host
  • with proper access control
The third approach: Bypassing container network stacks, cont’d

- A container is
  - just an application execution environment
  - not interested in how packets are delivered
  - Then, *we can bypass container network stacks*

- Network stack separation should be retained
  - `docker run --net=host` can cause unintended or malicious resource uses
    - address, port, protocol, etc

- **A new mechanism is needed**
  - connecting App on a container to the Host
  - with proper access control
Approach: Socket-Grafting

• Grafting sockets in containers onto sockets in hosts
  • A socket-layer communication channel design
  • \( \text{graft} = \text{接ぎ木する、移植する} \)
✓ One Network stack on the data path
✓ Independent from network stack implementations
Mechanism: AF_GRAFT

• A new address family for grafting sockets
  • Applications in containers create AF_GRAFT sockets
  • AF_GRAFT sockets are grafted onto other AF sockets across the network namespace boundary
Graft endpoint

- *Names* for AF_GRAFT sockets in the bind() semantics
  - Arbitrary strings
- GRAFT <-> Host endpoint mapping
  - AF_GRAFT manages the mapping table per container
  - preventing misuse of the host namespace

<table>
<thead>
<tr>
<th>Graft endpoint</th>
<th>Host endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep-http</td>
<td>10.0.0.1:80</td>
</tr>
<tr>
<td>ep4</td>
<td>10.0.0.1:8080</td>
</tr>
<tr>
<td>ep6</td>
<td>[2001:db8::beef]:8080</td>
</tr>
<tr>
<td>ep-un</td>
<td>/tmp/un-sk</td>
</tr>
</tbody>
</table>
AF_GRAFT Socket API

```c
/* Structure describing a graft socket address (endpoint) */
struct sockaddr_gr {
    __kernel_sa_family_t sgr_family;
    char sgr_epname[AF_GRAFT_EPNAME_MAX];
};

int sock;
struct sockaddr_gr saddr_gr;

sock = socket(AF_GRAFT, SOCK_STREAM, IPPROTO_TCP);

saddr_gr.sgr_family = AF_GRAFT;
strncpy(saddr_gr.sgr_epname, "ep-http", 7);

bind(sock, (struct sockaddr *)&saddr_gr, sizeof(saddr_gr));
/* Then, you can use sock as usual TCP sockets */
```
Outbound connections

- Dynamic-port graft endpoint
  - It uses randomly selected port numbers == typical client sockets
  - For example, mapping ep-out on X.X.X.X:random

```c
sock = socket(AF_GRAFT, SOCK_STREAM, IPPROTO_TCP);
saddr_gr.sgr_family = AF_GRAFT;
strncpy(saddr_gr.sgr_epname, "ep-out", 7);
bind(sock, (struct sockaddr *)&saddr_gr, sizeof(saddr_gr));

/* Then sock is grafted onto source IP:RandomPort socket*/
connect(sock, (struct sockaddr *)&dst, sizeof(dst));
```
Implementation

- https://github.com/upa/af-graft, AF_GRAFT kernel module
  - no kernel patches (but overwriting an existing AF number, AF_IPX)
  - Grafting is implemented as function call
    - no buffering, queueing, messaging => minimal overhead!
  - A few socket options for practical uses
  - A modified iproute2 for configuring the mapping table

$ ip graft add ep-http type ipv4 addr 10.0.0.1 port 80
$ ip graft add ep-out type ipv4 addr 10.0.0.2 port dynamic
$ ip graft del ep-un
$ ip graft show
Existing application with AF_GRAFT

- AF_GRAFT is a new address family
  - Applications need source code modifications
  - It is easy because of the familiar socket API, but difficult to deploy

Override system calls by the LD_PRELOAD trick

- `$ LD_PRELOAD libgraft-hijack.so app`
  - hijacking functions in shared library
- Hijacking:
  1. `getaddrinfo()`
  2. `socket()`, `bind()`, and `connect()`
- to convert address family-dependent socket operations into AF_GRAFT-capable ones
getaddrinfo()

- It was carefully designed to achieve AF-independent codes
  - Our modified getaddrinfo() can return AF_GRAFT and sockaddr_gr
- However, unfortunately, this is not the case in practical applications...

```c
/* IPv4 */
if (server_res->ai_family == AF_INET) {
    ... make ipv4 socket ...
}
/* IPv6 */
else if (server_res->ai_family == AF_INET6) {
    ... make ipv6 socket ...
}
/* Unknown protocol */
else {
    errno = EAFNOSUPPORT;
    return -1;
}
```

from iperf3

Hijacking socket() and bind()

• Hijacked socket()
  • returns AF_GRAFT sockets instead of AF_INET/INET6

• Hijacked bind()
  • uses sockaddr_gr instead of sockaddr_in/in6

• An env variable specifies which sockaddr convert to which sockaddr_gr
  • GRAFT_CONV_PAIRS="0.0.0.80=ep-http"
bind() before connect() for outbound connections

1. connect() does not need to call bind()
2. But, AF_GRAFT requires bind() to determine host sockets
   ✓ The hijacked connect() calls bind before connect()
   • sendto() and sendmsg() are also hijacked in the same manner
Evaluation

• Throughput and latency
  • iperf3 and sockperf
• HTTP server
  • NGINX and siege
• Message Queue
  • Zero MQ
• Networking
  • native host
  • docker0 (NAT)
  • AF_GRAFT
    • with libgraft-hijack.so

Baseline performance

Microservice Architecture

Host: Linux 4.4.0, Intel Core i7-3770K 3.5GHz CPU, 32GB memory, Mellanox ConnectX-4 LX 40Gbps NIC

Docker Container

docker0 or AF_GRAFT
• AF_GRAFT successfully mitigates the degradation
• Container to container communication via AF_GRAFT is the same as the communication via the loopback interface
Latency

- As well as the throughput test, AF_GRAFT also mitigates degradation from the latency perspective.
HTTP server benchmark

Increased by 40%

50 concurrent sessions

Transaction rate (Ktps)

File size (byte)

GRAFT
NAT

Increased by 40%
Message Queue benchmark

Approximately doubled

Throughput (Gbps)
Message size (Byte)
GRAFT
NAT

Throughput and measured

Docker Container

TX

RX

Host

Host

zmq app

zmq app
Limitations

• The LD_PRELOAD trick is not applicable to
  • Statically linked libraries
  • Golang that implements syscall without libc
• AF_GRAFT does not improve network stack performance
  • It never outperforms the performance of native hosts
• Network-*sensitive* applications
  • e.g., Container-based NFV
Conclusion

• Socket-Grafting
  • Containers with network-*insensitive* applications do not need network stacks
  • Bypassing container’s network stack by exploiting the socket layer
  • A new address family, called AF_GRAFT, as a practical mechanism for grafting

• The evaluation results demonstrated
  • Mitigating the network performance degradation due to the long data path
  • HTTP: 10-40% throughput improvement
  • ZeroMQ: up to doubled the throughput and 30% shorter latency
ToDo

• Integrating AF_GRAFT into Docker
  • Docker network driver plugin?
  • Option like -p?
  • We need comments or partners implementing such plugins ;)

• Integrating AF_GRAFT into Kubernetes
  • More complicated due to the service IP abstraction and load balancing
  • The Container Network Interface (CNI) focuses on the traditional abstraction (?)

• Go Go Netdev 0x13!