Kubernetes V1.0的IP-per-pod网络模型

Kubernetes V1.0 Networking Model of IP-per-POD

摘要 Abstract

网络虚拟化技术, 通常可分为桥接模式 (Bridge), 网址翻译模式 (NAT, Network Address Translation), 主机模式 (Host only), 无网模式 (None). 如: 开源的桌面虚拟化软件VirtualBox所示的虚拟化网络. 原理上，组合数据链路层(L2)和网络层(L3)的虚拟实现，以及广域网(WAN)和局域网(LAN)的虚拟架构，简单描述如下：

- 只虚拟化了guest(无论虚拟机或容器)的虚拟网卡(vif)

每个guest都具有网口，但没有可供连接的虚拟交换机, 以实现guest的组网 (none), 而虚拟组网则留给DevOps定制

- 虚拟化到L2层

这时，要求主机实现虚拟交换机(virtual Switch), 以提供guest通过虚拟网卡进行连接. 具体组网上，则可以：

1. 虚拟交换机组成一个孤立的局域网 (internal)
   在Linux系统, 简单可以动过创建bridge设备作为虚拟交换机, 和一对TAP设备分别连接bridge和guest实现. 这时, 虚拟机之间可以自由设置IP地址, 组成所需的网络. 但host和guest之间, 并不存在tcp/ip上的通信规则; 同时, guest与host所在的外部网络也互不相通.
2. 虚拟交换机通过TUN/TAP设备连接主机 (host only)
   创建一个TUN/TAP设备连到虚拟交换机, 使用ifconfig或ip link/address赋予该设备IP地址, 而虚拟机同样配置到该IP地址所在的网段, 这时, host和guest之间实现了通信, 通常这样做的目的是在host上启用一个DHCP服务器, 以便能为guest自动分配IP地址.
3. 虚拟交换机与外部交换机在同一个网络中 (POD) (bridging) :
   首先，将主机的物理网络端口配置为混杂模式(promiscuous), 然后使用ifconfig或ip link将该端口加到bridge上, 再后, 将物理网络端口使用的IP地址配置给bridge. 这时，guest的vif将得到外部网络规划下的地址 (DHCP分配的或网络管理员手工分配), 该地址与主机在同一个网络中.

- 虚拟化到L3层

该场景针对将网络虚拟化作为主机外部物理网络的子网. 主机提供为虚拟机路由的能力, 包括:

1. 虚拟机可以单向访问外部网络如internet
   在Linux系统，可以通过iptables的SNAT或masquerade实现
2. 外部网络通过PAT (Port Address Translation)访问DMZ内的虚拟机
   使用iptables的DNAT实现, 将主机的IP地址和端口映射到DMZ中的一台虚拟机
3. 外部网络通过NAT访问DMZ内的虚拟机
   同样, 通过iptable的DNAT实现, 为主机配置多个IP地址, 将其中一个IP地址映射到DMZ中虚拟机.

以上是一个主机上的虚拟化技术，针对集群，就演变为IaaS的网络虚拟机架构：

- 基于bridging模式的flat网络

所有主机采用bridging模式, 由外部交换机互连，所有主机的网络配置均需要去DevOps. 另一方面, 网络交通是平的

- 基于VLAN模式的tenant网络

VLAN即针对flat模式的网络交通隔离, 从而提供了多租户模式, 缺点是VLAN资源的限制: 802.1q规范了最大只能4095个VLAN, 大量交换机需要做VLAN配置

- 基于数据包encapsulation的tenant网络

类似802.1规范VLAN tag, 将正常的L2/3数据包tag并封装为专用L3(IP)或L4(UDP)的数据包, 实际上形成的是tunnel. 成熟的技术有GRE和VxLAN. 但如果只使用传统的bridge技术, 在构建云环境时, 需要大量的网络DevOps

- 分布式vSwitch

分布式vSwitch解决cluster上同步每个bridge设备的复杂性, 要求有一个active-active模式的网络配置数据库, 其中任一host上的网络配置, 均可同步到其它host. 成熟的技术有OpenVSwitch

- 网络名字空间 (netns)

使用网络名字空间区别每一个租户, 则每个租户都拥有了全部internet私有地址空间 (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16), 同时, 解耦了租户与VLAN tag的绑定.

Docker V1.7的网络模式 Network Model

Docker定义的网络模式有概念的区别:

- Bridge

Docker Engine默认创建一个docker0的bridge, 创建容器时, 在该容器PID标识的netns下创建一对veth, 用于连接容器vif和bridge. 当然bridge可以定制.

在Bridge模式下，可以在Host上配置NAT使容器访问internet，或配置PAT使internet访问容器上的服务，即www.

- Host

这与硬件虚拟化中的host only模式不同，Docker Engine没有为容器分配独立的网络栈, 而是共享主机的网络栈.

在Host模式下, 容器有权使用本只有root权限才可以访问的1~1024端口. 典型应用场景是主机配置多个internet地址，而容器上的www分别listen一个主机地址，提供网站服务; 以及再由iptables实现DNAT, 组建容器的load balancer.

- Container:Name_or_ID

应用容器网络映射网络容器 (POD容器)模式. 首先创建POD容器, 并使用PAT或NAT将网络访问规则配置给POD容器 (类似设置iptables规则), 然后该模式下所创建的App容器共享POD容器的网络栈.

该模式的好处是减少重复配置网络到每一个容器的DevOps, 典型应用是一个分布式系统可以由一个POD容器和多个App容器组件.

- None

Docker Engine只为容器分配网络栈, 但并没有为容器创建veth设备, 而是留给用户或软件框架高度定制

Kubernetes V1.0的网络模式 Network Model

由Docker Engine目前所提供的网络模式, 对于在host cluster上实现分布式容器 (CaaS, Container as a Service)来说, 可选的技术包括:

- PAT和NAT, 容器映射host的地址和端口, 容器内的应用服务只能配置在该端口才能对外提供服务, 缺点是系统管理人员或管理软件需要维护全部容器的映射数据.
- POD容器映射, 将host的一个地址和多个端口映射给POD容器, 其它App容器则使用POD容器分配到的其中一个端口, 该模式下, 只需要维护POD容器的地址和端口

Kubernetes采用的是POD容器映射的模式, 并规定如下：

1. 所有容器之间的通信不使用NAT

   也就是分布在cluster上的容器是在同一个IP网段内, 或者使用静态路由表, 在实现上:

   - POD容器配置为Docker的Bridge模式时, POD容器由docker0获取IP地址, 并以docker0为网关. 显然, cluster上所有docker0之间的通信要以所在host为网关建静态路由;
   - 或者, POD容器映射主机的地址和端口, 如果主机之间能通信即确保了POD容器及App容器之间的通信, 而与docker0的配置无关
   - POD容器配置为Docker的Host模式时, 每个容器都共享了主机的网络栈, 与Bridge模式下的映射方式类似
2. 所有主机与容器之间的通信不使用NAT

   同上, 由于容器和cluster都在同一个IP网段内, 或者使用静态路由表, 则无论Bridge模式或Host模式, 主机和容器之间都是可以在
3. 容器本身的IP地址是其它容器和主机访问的实际地址

POD容器 Container

使用kubectl create ...命令(V0.21.2)创建Pod, 用 kubectl get pods ...查看POD下的App容器, 用docker ps查看到POD容器, image为gcr.io/google_containers/pause:0.8.0.

可在github上获取pause镜像的Dockerfile:

```
FROM scratch
ADD pause /
ENTRYPOINT["/pause"]
```

从Dockerfile的内容:

- 第一行: 镜像是个精简内核的Linux
- 第二行: 镜像的root目录下放置了一个pause程序, 源代码是pause.go
- 第三行: 使用Docker Remote API或docker run创建容器时, 将运行pause程序

从pause.go代码:

```
package main
```

```
import (
        "os"
        "os/signal"
        "syscall"
)
func main() {
        c := make(chan os.Signal, 1)
        signal.Notify(c, os.Interrupt, os.Kill, syscall.SIGTERM)
        // Block until a signal is received.
        <-c
}
```

这是一个服务程序的骨架原型：

- 为go标准库定义的操作系统信号量类型os.Signal创建chan类型变量
- 库的signal运行时负责将操作系统发出的终止程序运行信号(如由ctrl-c, kill, kill -9命令)发送到chan变量
- 程序阻塞(block)直到收到系统发出的程序结束信号

制作镜像和发布镜像到DockerHub的DevOps可以通过make工程进行, 其中用到缩减二进制运行代码长度的工具以最大化精简镜像的尺寸, 如果操作失败, 则可以定制而忽略缩减步骤

POD容器只运行一个服务程序骨架程序, 是因为其目的是仅仅为App容器提供一个共享的网络栈， Kubernetes称之为IP-per-pod模型, POD容器明确了IP地址, App容器之间使用POD容器的IP地址在不同port上相互通信. 如果POD容器可以被外部访问, 就实现了容器cluster.

在Docker Engine下实现POD的过程

通过Kubernetes API或Kubectl命令, 根据POD的模板, 如:

```
id: "webpod"
kind: "Pod"
apiVersion: "v1"
desiredState:
  manifest:
    version: "v1"
    id: "webpod"
    containers:
      - name: "webpod80"
        image: "jonlangemak/docker:web_container_80"
        cpu: 100
        ports:
          - containerPort: 80
            hostPort: 80
      - name: "webpod8080"
        image: "jonlangemak/docker:web_container_8080"
        cpu: 100
```

```
      ports:
        - containerPort: 8080
          hostPort: 8080
```

该POD描述文件中，包括2个App容器:

1. 第一个容器名为webpod80, 网络配置为映射主机端口80到容器的80端口
2. 第二个容器名为webpod8080, 映射的则是8080端口

Kubernetes API Server收到该请求后:

- 首先, 通过Docker Remote API请求Docker Engine以pause镜像创建POD容器, 并将两个App容器要求的端口映射实际让POD容器去创建
  用docker命令即如:
  `docker run -d -p 80:80 -p 8080:8080 gcr.io/google_containers/pause:0.8.0`
- 然后, 再依次请求Docker Engine创建两个App容器, 而网络模式则选择POD容器映射
  用docker命令即如:
  `docker run -d --name webpod80 --net=container:<POD_name_or_ID> jonlangemak/docker:web_container_8080`

从Kubernetes的代码了解创建流程

(TBD)

结论 Conclusion

(TBD)

参考 Reference

1. Docker Documentation: How Docker networks a container
2. Kubernetes Documentation: Kubernetes Networking Model (doc repo in github)
3. Das Blinken Lichten: Kubernetes 101 - Networking
4. Das Blinken Lichten: Docker Networking 101 - Mapped Container Mode