**Exploring Kubernetes 1.28 Sidecar Container Support**

**Introduction**

This article documents how to explore the latest sidecar container functionality in Kubernetes 1.28.

Sidecar Container is a well-known and common design pattern in Kubernetes, where multiple containers are deployed inside a Pod. The Sidecar Container assists the main container in various functions, such as:

1. Network Proxy: For example, in Service Mesh architectures, it helps forward and handle different types of network traffic.
2. Log Collection: It processes logs generated by the main container.

However, for Kubernetes, these two types of containers fundamentally have no difference in terms of lifecycle and management. This similarity causes some issues in the use of sidecar containers:

1. In the case of a Job, if the main container finishes, but the sidecar container continues running, it prevents the Job from correctly determining whether the Pod has terminated successfully.
2. The sidecar container starts too late, causing the main container to be unable to use it when it starts, leading to errors that require waiting for the container to restart.

Regarding the second issue, common examples include:

1. In cases like Istio, where the Istio sidecar container starts later than the main container, causing momentary network unavailability when the main container starts.
2. In Google Kubernetes Engine (GKE), when accessing Cloud SQL through the Cloud SQL proxy, if the Cloud SQL proxy starts later than the main container, the main container cannot connect to the database, resulting in errors.

Therefore, workarounds were necessary in the past to address these issues.

Now, Kubernetes officially supports the sidecar container architecture internally. Its independent lifecycle management fundamentally resolves the common problems mentioned above, making the entire solution much more elegant.

**Environment**

This article is based on the following software:

1. Ubuntu 22.04
2. Kubeadm: 1.28.2–1.1
3. Kubelet: 1.28.2–1.1
4. Kubectl: 1.28.2–1.1
5. Containerd: 1.6.24–1

**System**

As the environment utilizes **kubeadm** for installation and **containerd** as the container runtime, the following script is prepared to install all related software and configure the necessary environment variables.

# Add Docker's official GPG key:  
sudo apt-get update  
sudo apt-get install ca-certificates curl gnupg  
sudo install -m 0755 -d /etc/apt/keyrings  
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg  
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg  
  
sudo chmod a+r /etc/apt/keyrings/docker.gpg  
  
# Add the repository to Apt sources:  
echo \  
 "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \  
 "$(. /etc/os-release && echo "$VERSION\_CODENAME")" stable" | \  
 sudo tee /etc/apt/sources.list.d/docker.list > /dev/null  
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list  
sudo apt-get update  
sudo apt-get install containerd.io=1.6.24-1  
https://github.com/containernetworking/plugins/releases/download/v1.3.0/cni-plugins-linux-amd64-v1.3.0.tgz  
sudo mkdir -p /opt/cni/bin  
sudo tar Cxzvf /opt/cni/bin cni-plugins-linux-amd64-v1.3.0.tgz  
sudo apt-get install -y kubelet=1.28.2-1.1 kubeadm=1.28.2-1.1 kubectl=1.28.2-1.1  
  
sudo modprobe br\_netfilter  
sudo modprobe overlay  
sudo sysctl -w net.ipv4.ip\_forward=1  
  
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf  
net.bridge.bridge-nf-call-iptables = 1  
net.bridge.bridge-nf-call-ip6tables = 1  
net.ipv4.ip\_forward = 1  
EOF  
  
  
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf  
overlay  
br\_netfilter  
EOF  
  
sudo sysctl --system  
  
containerd config default | sudo tee /etc/containerd/config.toml  
sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml  
sudo systemctl restart containerd

**Kubeadm**

Prepare the following kubeadm.config file to configure the Sidecar Container:

apiVersion: kubeadm.k8s.io/v1beta3  
kind: ClusterConfiguration  
networking:  
 podSubnet: 192.168.0.0/16  
apiServer:  
 extraArgs:  
 feature-gates: "SidecarContainers=true"  
controllerManager:  
 extraArgs:  
 feature-gates: "SidecarContainers=true"  
scheduler:  
 extraArgs:  
 feature-gates: "SidecarContainers=true"  
---  
apiVersion: kubelet.config.k8s.io/v1beta1  
featureGates:  
 SidecarContainers: true  
kind: KubeletConfiguration

Next, install Calico CNI and perform taint removal operations using kubeadm to establish a functional Kubernetes 1.28 environment.

sudo kubeadm init - config=kubeadm.config  
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-operator.yaml  
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/custom-resources.yaml  
$ kubectl taint nodes master node-role.kubernetes.io/control-plane:NoSchedule-

**Experiment**

Here, I will demonstrate two possible scenarios of using sidecar containers and discuss how to address the issues using the new features in Kubernetes 1.28.

**Case 1**

The first scenario explores the problem caused by using a sidecar container in a Job.

For example, deploying a Job service with a sidecar container using the following YAML. In this example, the functionality of the sidecar is not important; it’s just for demonstration purposes.

apiVersion: batch/v1  
kind: Job  
metadata:  
 name: pi  
spec:  
 template:  
 spec:  
 containers:  
 - name: pi  
 image: perl:5.34.0  
 command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]  
 - name: sidecar  
 image: hwchiu/netutils  
 restartPolicy: Never  
 backoffLimit: 4

When deploying this YAML, the following situation can be observed:

$ kubectl get pods,job  
NAME READY STATUS RESTARTS AGE  
pod/pi-6q4lh 1/2 NotReady 0 17m  
  
NAME COMPLETIONS DURATION AGE  
job.batch/pi 0/1 17m 17m

The main container finishes its execution, but the sidecar container continues running. As a result, the current Pod cannot reach the “Completed” state, and the Job cannot determine its COMPLETIONS.

Now, let’s try the new feature in version 1.28.

This feature is built upon a “never-ending initContainer,” so the setup starts from an initContainer and uses the restartPolicy to enable the functionality of the sidecar container. Once the sidecar container is set with "RestartPolicy: Always", its behavior changes slightly:

1. It can continue without needing to terminate to run other init containers.
2. If it exits due to any issues, it will automatically restart.
3. Its running state doesn’t affect the Pod’s state determination.

Next, try the following YAML file. We move the sidecar container to the init container stage to see if the Pod can successfully complete in this scenario.

apiVersion: batch/v1  
kind: Job  
metadata:  
 name: pi-sidecar  
spec:  
 template:  
 spec:  
 initContainers:  
 - name: network-proxy  
 image: hwchiu/python-example  
 restartPolicy: Always  
 containers:  
 - name: pi  
 image: perl:5.34.0  
 command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]  
 restartPolicy: Never  
 backoffLimit: 4

From the results, the Pod still recognizes 2 containers, but now it can successfully finish and reach the “Completed” state.

$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
pi-sidecar-bszf2 0/2 Completed 0 42s

Additionally, by observing kubectl describe pods, you can see the last line "Stopping container network-proxy." This means that after the main container finishes, the sidecar container (network-proxy) will be terminated by the Kubernetes and does not impact the main container's lifecycle.

Events:  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal Scheduled 79s default-scheduler Successfully assigned default/pi-sidecar-bszf2 to master  
 Normal Pulling 78s kubelet Pulling image "hwchiu/python-example"  
 Normal Pulled 77s kubelet Successfully pulled image "hwchiu/python-example" in 1.511s (1.511s including waiting)  
 Normal Created 77s kubelet Created container sidecar  
 Normal Started 77s kubelet Started container sidecar  
 Normal Pulled 76s kubelet Container image "perl:5.34.0" already present on machine  
 Normal Created 76s kubelet Created container pi  
 Normal Started 76s kubelet Started container pi  
 Normal Killing 67s kubelet Stopping container network-proxy

**Case 2**

In the second example, the simulated scenario involves establishing a connection similar to a proxy through a sidecar container. Therefore, the sidecar container must start before the main container.

Here’s an example YAML file:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: proxy  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 run: proxy  
 template:  
 metadata:  
 labels:  
 run: proxy  
 spec:  
 containers:  
 - name: app  
 image: hwchiu/netutils  
 command: ["/bin/sh"]  
 args: ["-c", "nc -zv localhost 5000 && sleep 1d"]  
 - name: proxy  
 image: hwchiu/python-example  
 ports:  
 - containerPort: 5000  
 startupProbe:  
 httpGet:  
 path: /  
 port: 5000

In this example, two containers are deployed, where the sidecar container is a server listening on port 5000. If the sidecar is not ready when the main container starts, the main container will exit and wait for the next restart.

$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
proxy-74dc7b8d88-77cft 2/2 Running 1 (49s ago) 52s  
proxy-74dc7b8d88-rlz8m 2/2 Running 1 (47s ago) 52s  
proxy-74dc7b8d88-zjkdh 2/2 Running 1 (46s ago) 52s  
$ kubectl logs -p proxy-74dbbdccd5-cf9pg  
Defaulted container "app" out of: app, proxy  
localhost [127.0.0.1] 5000 (?) : Connection refused

From the deployment results mentioned above, it can be observed that due to the order problem, all Pods will restart once. The failure logs of the previous container also indicate the failure to run due to the unpreparedness of the sidecar container.

Normal Scheduled 6m20s default-scheduler Successfully assigned default/proxy-74dbbdccd5-dzdmz to master  
 Normal Pulled 6m18s kubelet Successfully pulled image "hwchiu/netutils" in 1.447s (1.447s including waiting)  
 Normal Pulling 6m18s kubelet Pulling image "hwchiu/python-example"  
 Normal Pulled 6m15s kubelet Successfully pulled image "hwchiu/python-example" in 2.459s (2.459s including waiting)  
 Normal Created 6m15s kubelet Created container proxy  
 Normal Started 6m15s kubelet Started container proxy  
 Normal Pulling 6m14s (x2 over 6m19s) kubelet Pulling image "hwchiu/netutils"  
 Normal Created 6m13s (x2 over 6m18s) kubelet Created container app  
 Normal Pulled 6m13s kubelet Successfully pulled image "hwchiu/netutils" in 1.47s (1.47s including waiting)  
 Normal Started 6m12s (x2 over 6m18s) kubelet Started container app

By observing the events with kubectl describe pod, it can be seen that the Proxy container immediately grabs the App container with almost no gap.

Next, introduce the mechanism of the sidecar container and try again:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: proxy-sidecar  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 run: proxy-sidecar  
 template:  
 metadata:  
 labels:  
 run: proxy-sidecar  
 spec:  
 initContainers:  
 - name: proxy  
 image: hwchiu/python-example  
 ports:  
 - containerPort: 5000  
 restartPolicy: Always  
 startupProbe:  
 httpGet:  
 path: /  
 port: 5000  
 containers:  
 - name: app  
 image: hwchiu/netutils  
 command: ["/bin/sh"]  
 args: ["-c", "nc -zv localhost 5000 && sleep 1d"]

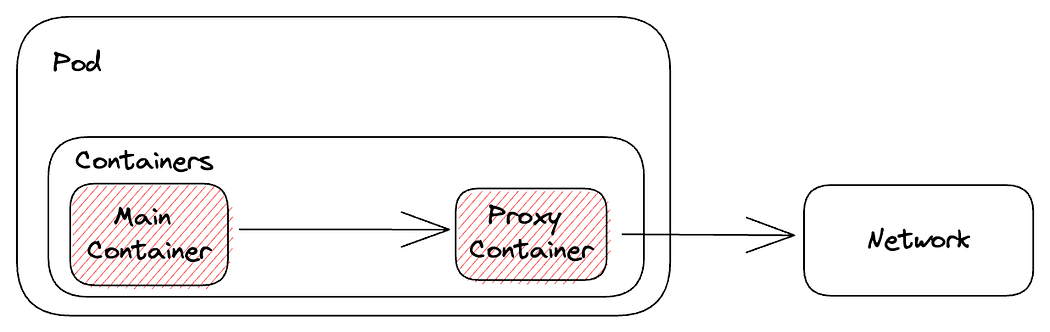
$ kubectl get pods  
proxy-sidecar-5dd9ff76f8-47mll 2/2 Running 0 2m34s  
proxy-sidecar-5dd9ff76f8-cmjjs 2/2 Running 0 2m34s  
proxy-sidecar-5dd9ff76f8-qctk8 2/2 Running 0 2m34s

Upon observation and execution, it can be seen that there are no Restart Counts, indicating smooth operation.

$ kubectl describe pods proxy-sidecar-5dd9ff76f8-qctk8  
...  
 Normal Scheduled 2m16s default-scheduler Successfully assigned default/proxy-sidecar-5dd9ff76f8-qctk8 to master  
 Normal Pulling 2m14s kubelet Pulling image "hwchiu/python-example"  
 Normal Pulled 2m11s kubelet Successfully pulled image "hwchiu/python-example" in 1.507s (2.923s including waiting)  
 Normal Created 2m11s kubelet Created container proxy  
 Normal Started 2m11s kubelet Started container proxy  
 Normal Pulling 2m5s kubelet Pulling image "hwchiu/netutils"  
 Normal Pulled 2m1s kubelet Successfully pulled image "hwchiu/netutils" in 1.475s (4.405s including waiting)  
 Normal Created 2m kubelet Created container app  
 Normal Started 2m kubelet Started container app

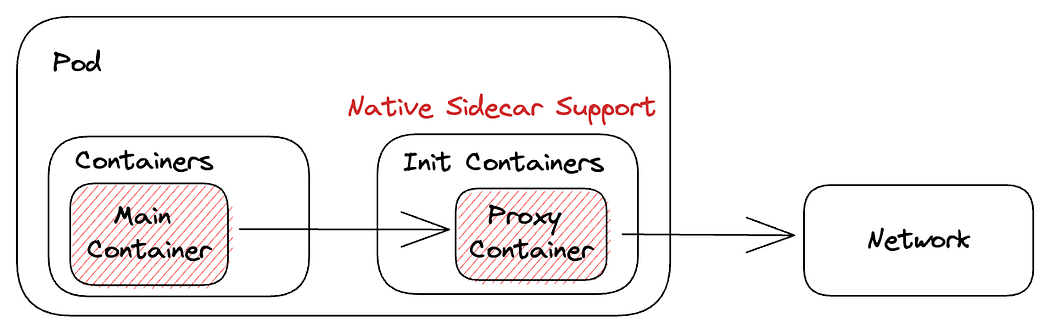
After introducing the sidecar container configuration, it can be observed that the Proxy container waits for a while before fetching the new image. This is because the sidecar container waits until its **StartupProbe** is complete before proceeding to run the main container. Through this mechanism, it ensures that the sidecar container starts before the main container.

Finally, these two diagrams illustrate the process of example two. In the past, all containers were placed within the containers section to handle the logic of the sidecar container:



Previous Sidecar Container Implementations

In the new architecture, the configuration is moved to initContainer and handled internally by Kubernetes, ensuring a dedicated lifecycle:



Native Sidecar Contaienr Support