Troubleshooting Kubernetes Networking on Windows: Part 1

By David Schott

Published May 15 2019 06:00 AM   26.2K Views

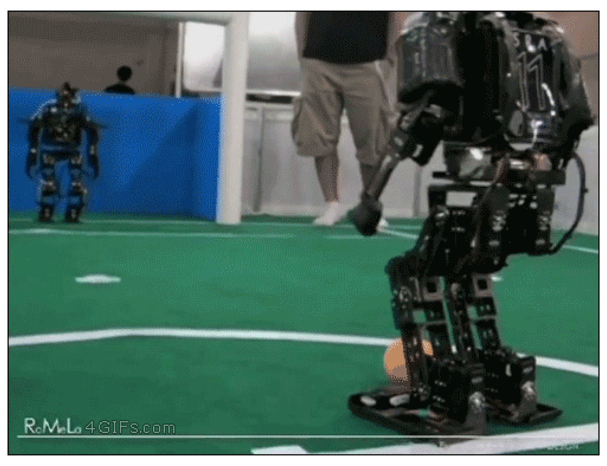
Co-Authors

David Schott

Version history

Last update: Mar 31 2021 09:54 AM
Updated by: David Schott

We've all been there: Sometimes things just don't work the way they should even though we followed everything down to a T.



Kubernetes in particular, is not easy to troubleshoot – even if you're an expert. There are multiple components involved in the creation/deletion of containers that must all harmoniously interoperate end-to-end. For example:

- Inbox platform services (e.g. WinNAT, HNS/HCS, VFP)
- Container runtimes & Go-wrappers (e.g. Dockershim, ContainerD, hcsshim)
- Container orchestrator processes (e.g. kube-proxy, kubelet)
- CNI network plugins (e.g. win-bridge, win-overlay, azure-cni)
- IPAM plugins (e.g. host-local)
- Any other host-agent processes/daemons (e.g. FlannelD, Calico-Felix, etc.)
- … (more to come!)

This, in turn, also means that the potential problem space to investigate can grow overwhelmingly large when things **do** end up breaking. We often hear the phrase: *"I don't even know where to begin."*

The intent of this blog post is to educate the reader on the available tools and resources that can help unpeel the first few layers of the onion; it is not intended to be a fully exhaustive guide to root-cause every possible bug for every possible configuration. However, by the end one should at least be able to narrow down on an observable symptom through a pipeline of analytical troubleshooting steps and come out with a better understanding of what the underlying issue could be.

> NOTE: Most of the content in this blog is directly taken from the amazing Kubecon Shanghai '18 video "Understanding Windows Container Networking in Kubernetes Using a Real Story" by Cindy Xing (Huawei) and Dinesh Kumar Govindasamy (Microsoft).

Table of Contents

Step 1: Ensure Kubernetes is installed and running correctly

As mentioned in the introduction, there are *a lot* of different platform and open-source actors that are needed to operate a Kubernetes cluster. It can be hard to keep track of all of them - especially given that they release at a different cadence.

One quick sanity-check that can be done without any external help is to employ a validation script that verifies supported bits are installed:

```
PS C:\k\yaml> .\Debug-WindowsNode.ps1
Checking for common problems with Windows Kubernetes nodes
Container Host OS Product Name: Windows Server 2019 Datacenter
Container Host OS Build Label: 17763.1.amd64fre.rs5_release.180914-1434
Describing Windows Version and Prerequisites
 [+] Is Windows Server 2019 108ms
 [+] Has 'Containers' feature installed 3.31s
 [+] Has HNS running 77ms
Describing Docker is installed
 [+] A Docker service is installed - 'Docker' or 'com.Docker.Service'  74ms
 [+] Service is running 47ms
 [+] Docker.exe is in path 251ms
 [+] Should be a supported version 61ms
Docker version: 18.09.5
Describing Kubernetes processes are running
 [+] There is 1 running kubelet.exe process 70ms
 [+] There is 1 running kube-proxy.exe process 54ms
PS C:\k\yaml> _
```

Verifying Kubernetes is installed

While trivial, another step that can be equally easily overlooked is ensuring that all the components are indeed running. Any piece of software can crash or enter a deadlock-like state, including host-agent processes such as kubelet.exe or kube-proxy.exe. This can result in unexpected cluster behavior and detached node/container states, but thankfully it's easy to check. Running a simple ps command usually suffices:

```
Administrator: Windows PowerShell
PS C:\Users\Administrator> ps |findstr "kube flanneld"
    276      16    24148      27392       0.39   1568   2 flanneld
    347      21    42164      56220       1.86   4708   2 kubelet
    239      18    23320      28436       0.41   8136   2 kube-proxy
PS C:\Users\Administrator> _
```

Typical Kubernetes processes running

Unfortunately, the above command won't capture that the processes themselves could be stuck waiting in a deadlock-like state; we will cover this case in step 4.

Step 2: Use a script to validate basic cluster connectivity

Before diving head-first into analyzing HNS resources and verbose logs, there is a handy Pester test suite which allows you to validate basic connectivity scenarios and report on success/failure here. The only pre-requisite in order to run it is that you are using Windows Server 2019 (requires minor fix-up otherwise) and that you have more than one node for the remote pod test:

```
PS C:\k> .\ValidateKubernetes.Pester.tests.ps1
Describing Kubernetes Prerequisites
   Context Checking Docker images
    [+] should have windowsservercore image 1.47s
   Context Checking Kubernetes Binaries are running
    [+] kubelet.exe is running 339ms
    [+] kube-proxy.exe is running 60ms
    [+] flanneld.exe is running 54ms
Describing Basic Connectivity Tests
   Context Windows Connectivity
Waiting for the deployment (win-webserver) to be complete. @{availableReplicas=2;
adyReplicas=2; replicas=4; unavailableReplicas=2; updatedReplicas=4}
Waiting for the deployment (win-webserver) to be complete. @{availableReplicas=2;
adyReplicas=2; replicas=4; unavailableReplicas=2; updatedReplicas=4}
Waiting for the deployment (win-webserver) to be complete. @{availableReplicas=2;
adyReplicas=2; replicas=4; unavailableReplicas=2; updatedReplicas=4}
Waiting for the deployment (win-webserver) to be complete. @{availableReplicas=2;
adyReplicas=2; replicas=4; unavailableReplicas=2; updatedReplicas=4}
Waiting for the deployment (win-webserver) to be complete. @{availableReplicas=2;
adyReplicas=2; replicas=4; unavailableReplicas=2; updatedReplicas=4}
    [+] should have more than 1 local container 32.28s
    [+] should have at least 1 remote container 56ms
Checking win-webserver-7f64f4ff9f-2dg2g has IP address 10.244.17.49
Checking win-webserver-7f64f4ff9f-5s9qx has IP address 10.244.17.50
Checking win-webserver-7f64f4ff9f-j9648 has IP address 10.244.19.32
Checking win-webserver-7f64f4ff9f-p2dhw has IP address 10.244.19.31
    [+] Pods should have correct IP 2.19s
Testing from win-webserver-7f64f4ff9f-2dg2g 10.244.17.49
```

Snippet from Kubernetes Connectivity Test Suite

The intent of running this script is to have a quick glance of overall networking health, as well as hopefully accelerate subsequent steps by knowing what to look for.

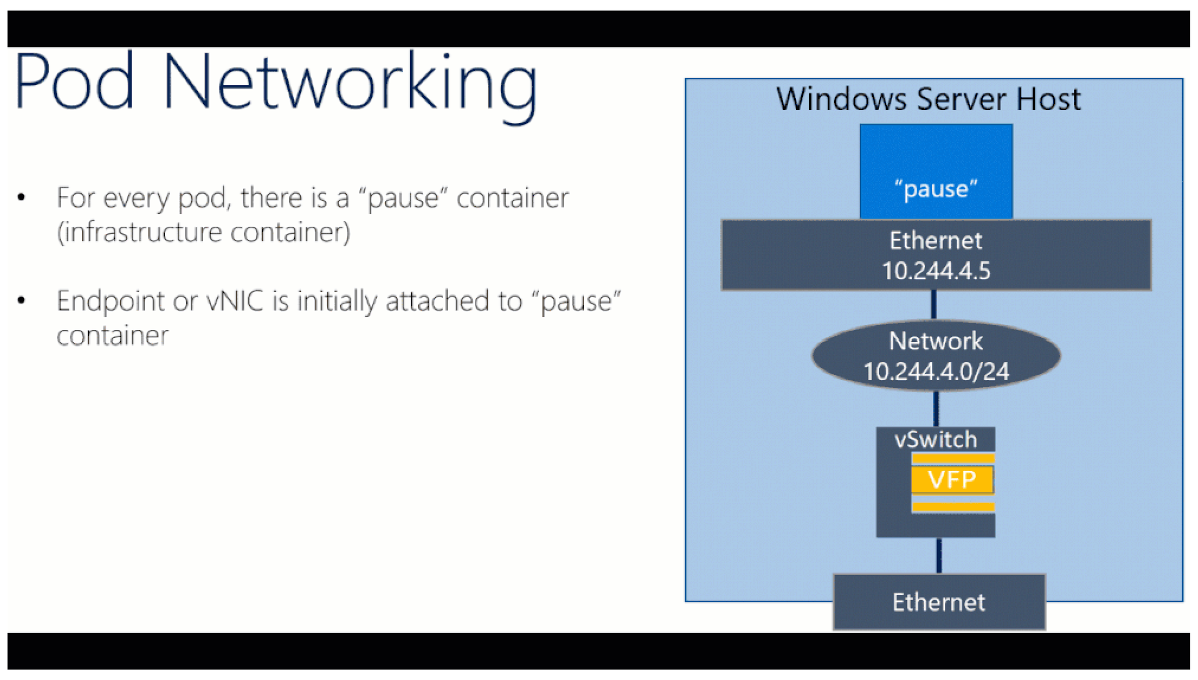
Step 3: Query the built-in Kubernetes event logs

After verifying that all the processes are running as expected, the next step is to query the built-in Kubernetes event logs and see what the basic built-in health-checks that ship with K8s have to say:

```
PS C:\k\yaml> kubectl get pods -o wide
NAME                             READY   STATUS              RESTARTS   AGE   IP       NODE
win-webserver-69495c7694-76nxd   0/1     ContainerCreating   0          5s    <none>   win-24elfepi8d9
win-webserver-69495c7694-tjf7f   0/1     ContainerCreating   0          6s    <none>   win-24elfepi8d9
PS C:\k\yaml> kubectl describe po/win-webserver-69495c7694-76nxd_
```

Kubernetes pods that are stuck in "ContainerCreating" state

More Information about misbehaving Kubernetes resources such as event logs can be viewed using the "kubectl describe" command. For example, one frequent misconfiguration on Windows is having a misconfigured "pause" container with a kernel version that doesn't match the host OS:



Shared pause container vNIC in a pod

Here are the corresponding event logs from kubectl describe output, where we accidentally built our "kubeletwin/pause" image on top of a Windows Server, version 1803 container image and ran it on a Windows Server 2019 host:

```
Events:
  Type     Reason     Age   From                    Message
  ----     ------     ----  ----                    -------
  Normal   Scheduled  17m   default-scheduler       Successfully assigned default/win-webserver-69495c7694-76nxd to win-24elfepi8d9
  Normal   Pulling    16m   kubelet, win-24elfepi8d9  pulling image "mcr.microsoft.com/windows/nanoserver:1803"
  Normal   Pulled     14m   kubelet, win-24elfepi8d9  Successfully pulled image "mcr.microsoft.com/windows/nanoserver:1803"
  Warning  Failed     14m   kubelet, win-24elfepi8d9  Error: failed to start container "windowswebserver": Error response from daemon: CreateComputeSystem windowswebserver
The container operating system does not match the host operating system.
(extra info: {"SystemType":"Container","Name":"windowswebserver","Owner":"docker","VolumePath":"\\\\?\\Volume{0b24c67a-6083-11e9-8b4c-f2f8b2bdf7ae}","IgnoreFlushesDuringBoot":true,"LayerFolderPath":"C:\\ProgramData\\docker\\windowsfilter\\windowswebserver","Layers":[{"ID":"2b148eee-41bc-59b3-ae7c-1d6fd9130f5b","Path":"C:\\ProgramData\\docker\\windowsfilter\\81efe47748146e3dd688a578a4c42db5419b1eefe6f5280a8f1d352549a5660c"},{"ID":"1b0cefa3-4cdf-5611-9ed2-12bb79c79edf","Path":"C:\\ProgramData\\docker\\windowsfilter\\7e3879d6a9db9d165b822465f2ff72733cba0b8b7aaa37dcb0ca4a773ba1f4bb"}],"ProcessorWeight":5000,"HostName":"win-webserver-69495c7694-76nxd","MappedDirectories":[{"HostPath":"c:\\var\\lib\\kubelet\\pods\\885e65df-6094-11e9-b170-00155d05c30a\\volumes\\kubernetes.io~secret\\default-token-d68xj","ContainerPath":"c:\\var\\run\\secrets\\kubernetes.io\\serviceaccount","ReadOnly":true,"BandwidthMaximum":0,"IOPSMaximum":0,"CreateInUtilityVM":false}],"HvPartition":false,"NetworkSharedContainerName":"0365e8097eaee4b440e5af5174a2c5c361420342e9103804e9fd9ae512639742"})
  Warning  Failed  13m  kubelet, win-24elfepi8d9  Error: failed to start container "windowswebserver": Error response from daemon: CreateComputeSystem windowswebserver: The container operating system does not match the host operating system.
```

Erroneous Kubernetes event logs

(On a side note, this specific example issue can be avoided altogether if one references the multi-arch pause container image mcr.microsoft.com/k8s/core/pause:1.0.0 which will run on both Windows Server, version 1803 and Windows Server 2019).

Step 4: Analyze kubelet, kube-proxy logs

Another useful source of information that can be leveraged to perform root-cause analysis for failing container creations is the kubelet, FlannelD, and kube-proxy logs.

These components all have different responsibilities. Here is a very brief summary of what they do which should give you a rough idea on what to watch out for:

Component	Responsibility	When to inspect?
Kubelet	Interacts with container runtime (e.g. Dockershim) to bring up containers and pods.	Erroneous pod creations/configurations
Kube-proxy	Manages network connectivity for containers (programming policies used for NAT'ing or load balancing).	Mysterious network glitches, in particular for service discovery and communication
FlannelD	Responsible for keeping all the nodes in sync with the rest of the cluster for events such as node removal/addition. This consists of assigning IP blocks (pod subnets) to nodes as well as plumbing routes for inter-node connectivity.	Failing inter-node connectivity

Log files for all of these components can be found in different locations; by default a log dump for kubelet and kube-proxy is generated in the C:\k directory, though some users opt to log to a different directory.

If the logs appear to not have updated in a longer time, then perhaps the process is stuck, and a simple restart or sending the right signal can kick things back into place.

Step 5: Inspect CNI network plugin configuration

Another common source of problems that can cause containers to fail to start with errors such as "FailedCreatePodSandbox" is having a misconfigured CNI plugin and/or config. This usually occurs whenever there are bugs or typos in the deployment scripts that are used to configure nodes:

```
Events:
  Type     Reason                  Age   From                    Message
  ----     ------                  ----  ----                    -------
  Normal   Scheduled               60s   default-scheduler       Successfully assigned default/win-webserver-7f64f4ff9f-g2j5d to win-qleogglvscd
  Warning  FailedCreatePodSandBox  7s    kubelet, win-qleogglvscd  Failed create pod sandbox: rpc error: code = Unknown desc = failed to set up sandbox container "d537d3edb6fa984204133767f619ecf75df2fe10a894512cc2f5bda0a25a7f4a" network for pod "win-webserver-7f64f4ff9f-g2j5d": NetworkPlugin cni failed to set up pod "win-webserver-7f64f4ff9f-g2j5d_default" network: error while executing ADD command: error while ProvisionEndpoint(d537d3edb6fa984204133767f619ecf75df2fe10a894512cc2f5bda0a25a7f4a_cbr0,AC27E0EC-B5AA-4950-B5A8-619C8883DD4E,d537d3edb6fa984204133767f619ecf75df2fe10a894512cc2f5bda0a25a7f4a): failed to create the new HNSEndpoint: hnsCall failed in Win32: There was an operation attempted on a degraded object. (0x803b0017)
```

Example error due to misconfigured CNI config

Thankfully, the network configuration that is passed to CNI plugins in order to plumb networking into containers is a very simple static file that is easy to access. On Windows, this configuration file is stored under the "C:\k\cni\config\" directory. On Linux, a similar file exists in "/etc/cni/net.d/".

Here is the corresponding typo that caused pods to fail to start due to degraded networking state:

```
"Name":  "EndpointPolicy",
"Value":  {
        "Type":  "OutBoundNAT",
        "ExceptionList":  [
                        "10.244.0.0/",
                        "10.96.0.0/12",
                        "10.127.130.0/24"
                    ]
        }
```

Highlighted Typo in CNI Config

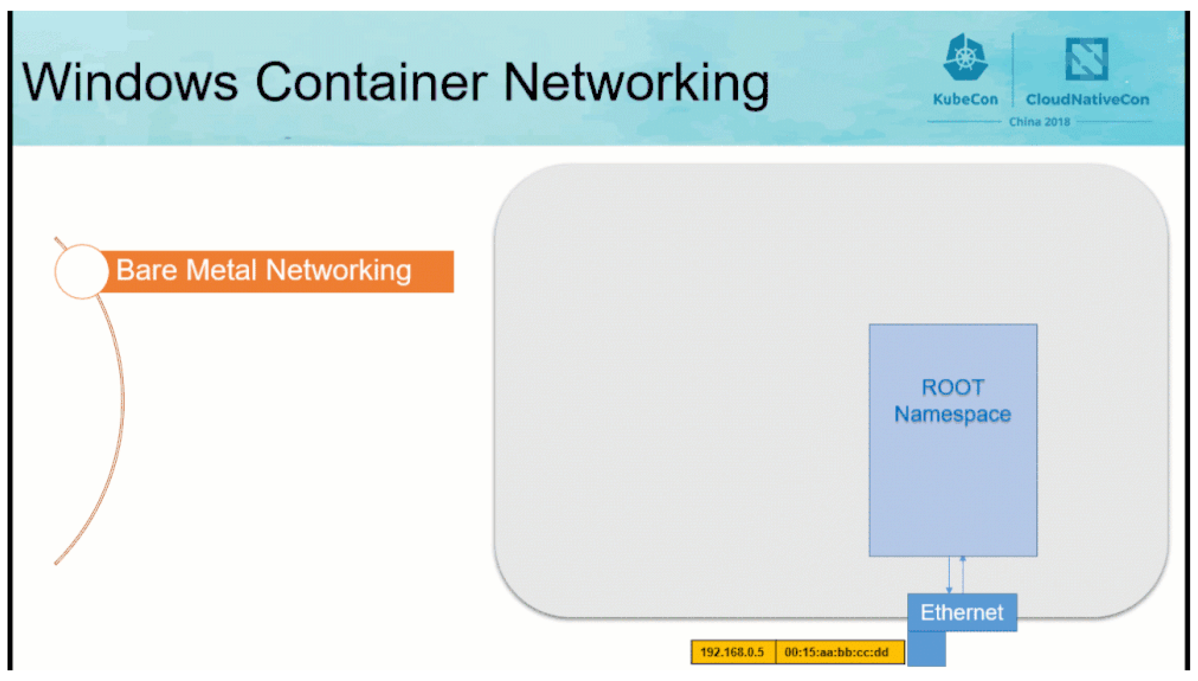
Whenever there are failing pod creations or unexpected network plumbing, we should always inspect the CNI config file for typos and consult the CNI plugins documentation for more details on what is expected. Here are the docs for the Windows plugins:

- win-bridge
- win-overlay
- flannel

Step 6: Verify HNS networking state

Having exhaustively examined Kubernetes-specific event logs and configuration files previously, the next step usually consists of collecting network information programmed on the networking stack (control plane and data plane) used by containers. All of the information can be collected conveniently by the "CollectLogs.ps1" script, which will be done in step 7.

Before reviewing the contents of the "CollectLogs.ps1" tool, the Windows container networking architecture needs to be understood at a high-level.

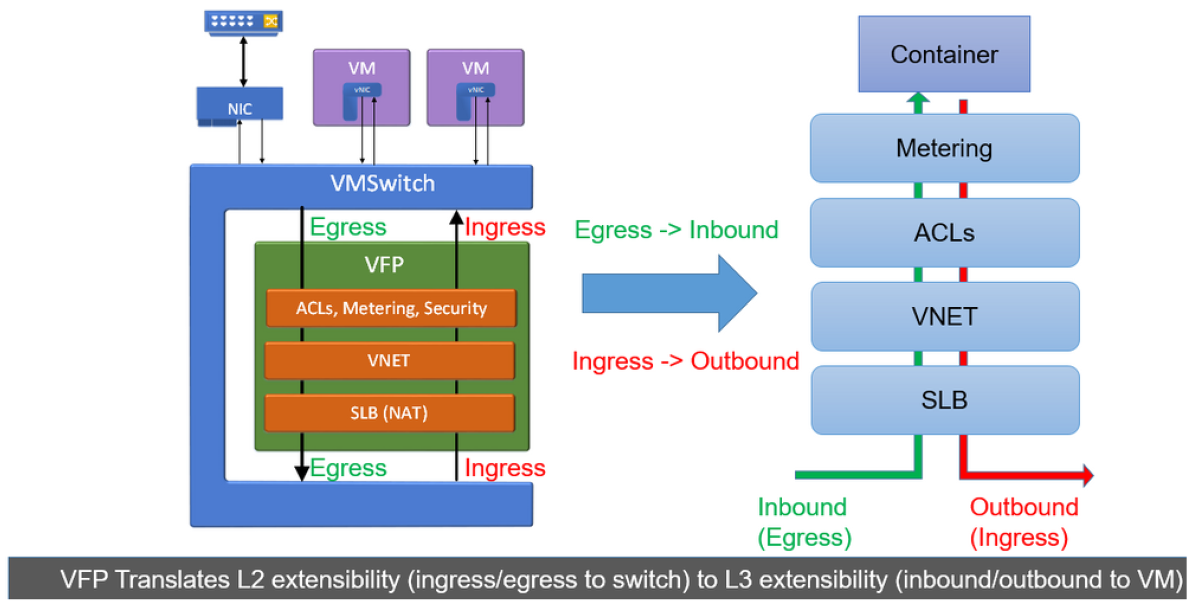


Windows container networking Overview

Windows Component	Responsibilities	Linux Counterpart
Network Compartment	• Logical separation in the TCP/IP stack. • Packet forwarding between compartments is prevented (by default). • All IP objects (addresses, routes, etc.) stay unique to the compartment.	Network namespace
vSwitch and HNS networks	• Provides L2 switching and L3 functionality. • Each vSwitch has its own forwarding table and forwards packets based on MAC address or VLAN tag. • Dynamically add/remove switch ports. • 1 (external) vSwitch / NIC. • 1 vSwitch / HNS network.	Bridge and IP routing
vNICs, HNS endpoints, and vSwitch ports	• Container NICs (vNICs) are bound to a corresponding port in the vSwitch. • Endpoints are a HNS abstraction for a container vNIC.	IP Links and virtual network interfaces
HNS policies, VFP rules, Firewall	• VFP is the programmable, match-action based filtering engine. • Applies rules to incoming outgoing packets from vPort. • VFP rules are different for each vPort.	iptables

One particular component to highlight is VFP (Virtual Filtering Platform), which is a vSwitch extension containing most of the decision logic used to route packets correctly from source to destination by defining operations to be performed on packets such as:

- Encapsulating/Decapsulating packets
- Load balancing packets
- Network Address Translation
- Network ACLs



Overview of Virtual Filtering Platform (VFP)

To read up more on this topic, many more details on VFP can be found here.

Our first starting point should be to check that all the HNS resources indeed exist. Here is an example screenshot that shows the HNS networking state for a cluster with kube-DNS service (10.96.0.10) and a sample Windows web service (10.104.193.123) backed by 2 endpoint DIPs ("768b4bd1-774c-47e8-904f-91c007a4b183", "048cd973-b5db-45a6-9c65-16dec22e871d"):

```
PS C:\k> hnsdiag list all
Networks:
Name            ID
nat             4B064BE3-7724-4E14-BF1C-14EDEEEAAC66
External        5CBFEB28-859E-499B-A267-6DDF2F064BAE
cbr0            E83BE224-D184-4F51-9516-7A4BC4D28E37

Endpoints:
Name            ID                                   Virtual Network Name
cbr0_ep         8c956a28-97db-40a5-abf0-f101af879bbd cbr0
Ethernet        08ce4a68-ec11-461a-b673-59e6bf3aa440 cbr0
Ethernet        90fbd2ff-f238-464e-a2a4-11569c4e2935 cbr0
99d5d6b7893df79974c5f6edf92ecdb4d3bfb347ba29912fcb3adaed95a7f71b_cbr0 048cd973-b5db-45a6-9c65-16dec22e871d cbr0
Ethernet        611fb097-51a4-45cc-9d61-99e0e7cd4de5 cbr0
79eabe19b039b289532a0f4823905b61cfc9a23ab4039a91a773584c5c1e063c_cbr0 768b4bd1-774c-47e8-904f-91c007a4b183 cbr0

Namespaces:
ID                                   | Endpoint IDs

LoadBalancers:
ID                                   | Virtual IPs     | Direct IP IDs
6a4ea0c5-3ca0-4b30-806f-dd76822fea60 |  10.104.193.123 | 768b4bd1-774c-47e8-904f-91c007a4b183 048cd973-b5db-45a6-9c65-16dec22e871d
ca1a6993-2c7b-4b0c-aee9-6569606ed143 |  10.96.0.1      | 08ce4a68-ec11-461a-b673-59e6bf3aa440
643f8c94-af7a-4883-9c8f-b7675a0a0a84 |  10.96.0.10     | 90fbd2ff-f238-464e-a2a4-11569c4e2935 611fb097-51a4-45cc-9d61-99e0e7cd4de5
93f5e7ce-141c-485f-a95c-6840d3dad64b |                 | 768b4bd1-774c-47e8-904f-91c007a4b183 048cd973-b5db-45a6-9c65-16dec22e871d
PS C:\k> _
```

Summary of typical Host Networking Service (HNS) objects

We can take a closer look at the network object representing a given endpoint DIP using Get-HNS* cmdlets (this even works for remote endpoints!)

```
PS C:\k> get-hnsendpoint | ? ID -Like "768b4bd1-774c-47e8-904f-91c007a4b183"


ActivityId                : 04FD5FAD-FA5F-4059-9947-3A9560092503
AdditionalParams          :
CreateProcessingStartTime : 132000913955753843
DNSServerList             : 10.96.0.10
DNSSuffix                 : svc.cluster.local
EncapOverhead             : 50
GatewayAddress            : 10.244.19.2
Health                    : @{LastErrorCode=0; LastUpdateTime=132000913954743735}
ID                        : 768B4BD1-774C-47E8-904F-91C007A4B183
IPAddress                 : 10.244.19.29
MacAddress                : 00-15-5D-C9-3C-A4
Name                      : 79eabe19b039b289532a0f4823905b61cfc9a23ab4039a91a773584c5c1e063c_cbr0
Policies                  : {@{ExceptionList=System.Object[]; Type=OutBoundNAT}, @{DestinationPrefix=10.96.0.0/12; NeedEncap=True; Type=ROUTE},
                            @{DestinationPrefix=10.127.130.38/32; NeedEncap=True; Type=ROUTE}, @{Type=L2Driver}}
PrefixLength              : 24
Resources                 : @{AdditionalParams=; AllocationOrder=6; Allocators=System.Object[]; Health=;
                            ID=04FD5FAD-FA5F-4059-9947-3A9560092503; PortOperationTime=0; State=1; SwitchOperationTime=0; VfpOperationTime=0;
                            parentId=4EB57087-A91E-4B17-A244-0421EFE90910}
SharedContainers          : {79eabe19b039b289532a0f4823905b61cfc9a23ab4039a91a773584c5c1e063c,
                            e215cbf7836833b2825ff37d94b46050259e46aab82667bf7e6faac9f79e37c3}
StartTime                 : 132000913960967243
State                     : 3
Type                      : L2Bridge
Version                   : 38654705666
VirtualNetwork            : E83BE224-D184-4F51-9516-7A4BC4D28E37
VirtualNetworkName        : cbr0
```

Typical HNS endpoint object

The information listed here (DNSSuffix, IPAddress, Type, VirtualNetworkName, and Policies should match what was passed in through the CNI config file.

Digging deeper, to view VFP rules we can use the inbox "vfpctrl" cmdlet. For example, to view the layers of the endpoint:

```
PS C:\k> vfpctrl /port 768b4bd1-774c-47e8-904f-91c007a4b183 /list-layer

 ITEM LIST
===========

LAYER : ACL_ENDPOINT_LAYER
    Friendly name : ACL_ENDPOINT_LAYER
    Priority : 15

LAYER : SLB_NAT_LAYER
    Friendly name : SLB_NAT_LAYER
    Flags : 0x9 Default Allow , Update flows on address change
    Priority : 50

LAYER : SLB_DECAP_LAYER_STATEFUL
    Friendly name : SLB_DECAP_LAYER_STATEFUL
    Flags : 0x9, Default Allow , Update flows on address change
    Priority : 100

LAYER : CUSTOMER_ROUTE
    Friendly name : CUSTOMER_ROUTE
    Flags : 0x1, Default Allow
    Priority : 1500

LAYER : VNET_PA_ROUTE_LAYER
    Friendly name : VNET_PA_ROUTE_LAYER
    Flags : 0x1, Default Allow
    Priority : 2000

Command list-layer succeeded!
PS C:\k> _
```

Listing VFP layers for a given container vPort

Similarly, to print the rules belonging to a specific layer (e.g. SLB_NAT_LAYER) that each packet goes through:

```
PS C:\k> vfpctrl /port 768b4bd1-774c-47e8-904f-91c007a4b183 /layer SLB_NAT_LAYER /list-rule

 ITEM LIST
==========

  GROUP : SLB_GROUP_NAT_IPv4_IN
      Friendly name : SLB_GROUP_NAT_IPv4_IN
      Priority : 100
      Direction : IN
      Type : IPv4
        Conditions:
            <none>
      Match type : Longest prefix match on Destination IP

    RULE :
        Friendly name : NAT_Allow
        Priority : 65535
        Flags : 1 terminating
        Type : allow
        Conditions:
            <none>
        Flow TTL: 0
        FlagsEx : 0

  GROUP : SLB_GROUP_NAT_IPv4_OUT
      Friendly name : SLB_GROUP_NAT_IPv4_OUT
      Priority : 300
      Direction : OUT
      Type : IPv4
        Conditions:
            <none>
      Match type : Longest prefix match on Destination IP

    RULE :
        Friendly name : EXCEPTIONS_OUTBOUNDNAT_3FEC9_10.127.130.3810.127.130.0
        Priority : 0
        Flags : 1 terminating
        Type : allow
        Conditions:
            Destination IP : 10.127.130.0-10.127.130.255
        Flow TTL: 240
        FlagsEx : 0
```

Snippet of VFP rules for the SLB_NAT_LAYER of a container vPort

The information programmed into VFP should match with what was specified in the CNI config file and HNS Policies.

Step 7: Analyze snapshot of network using CollectLogs.ps1

Now that we familiarized ourselves with the state of the network and its basic constituents let's take a look at some common symptoms and correlate it against the likely locations where the culprit may be.

Our tool of choice to take a snapshot of our network is CollectLogs.ps1. It collects the following information (amongst a few other things):

File	Contains
endpoint.txt	Endpoint information and HNS Policies applied to endpoints.
ip.txt	All NICs in all network compartments (and which)
network.txt	Information about HNS networks
policy.txt	Information about HNS policies (e.g. VIP -> DIP Load Balancers)
ports.txt	Information about vSwitch (ports)
routes.txt	Route tables
hnsdiag.txt	Summary of all HNS resources
vfpOutput.txt	Verbose dump of all the VFP ports used by containers listing all layers and associated rules

Example #1: Inter-node communication Issues

L2bridge / Flannel (host-gw)

When dealing with inter-node communication issues such as pod-to-pod connectivity across hosts, it is important to check static routes are programmed. This can be achieved by inspecting the routes.txt or Skip to primary navigation

```
PS C:\Users\Administrator> Get-NetRoute

ifIndex DestinationPrefix                              NextHop                                  RouteMetric if
                                                                                                            Me
                                                                                                            tr
                                                                                                            ic
------- -----------------                              -------                                  ----------- --
45      255.255.255.255/32                             0.0.0.0                                          256 15
41      255.255.255.255/32                             0.0.0.0                                          256 50
24      255.255.255.255/32                             0.0.0.0                                          256 15
1       255.255.255.255/32                             0.0.0.0                                          256 75
45      224.0.0.0/4                                    0.0.0.0                                          256 15
41      224.0.0.0/4                                    0.0.0.0                                          256 50
24      224.0.0.0/4                                    0.0.0.0                                          256 15
1       224.0.0.0/4                                    0.0.0.0                                          256 75
41      172.17.63.255/32                               0.0.0.0                                          256 50
41      172.17.48.1/32                                 0.0.0.0                                          256 50
41      172.17.48.0/20                                 0.0.0.0                                          256 50
1       127.255.255.255/32                             0.0.0.0                                          256 75
1       127.0.0.1/32                                   0.0.0.0                                          256 75
1       127.0.0.0/8                                    0.0.0.0                                          256 75
45      10.244.19.255/32                               0.0.0.0                                          256 15
45      10.244.19.2/32                                 0.0.0.0                                          256 15
45      10.244.19.0/24                                 0.0.0.0                                          256 15
24      10.244.18.0/24                                 10.127.130.35                                    256 15
24      10.244.17.0/24                                 10.127.130.36                                    256 15
24      10.244.0.0/24                                  10.127.130.37                                    256 15
24      10.127.130.255/32                              0.0.0.0                                          256 15
24      10.127.130.38/32                               0.0.0.0                                          256 15
24      10.127.130.0/24                                0.0.0.0                                          256 15
45      0.0.0.0/0                                      10.244.19.1                                      256 15
24      0.0.0.0/0                                      10.127.130.1                                     256 15
```

Get-NetRoute output highlighting static routes for pod networks

There should be routes programmed for each pod subnet (e.g. 10.244.18.0/24) => container host IP (e.g. 10.127.130.35).

When using Flannel, users can also consult the FlannelD output to watch for the appropriate events for adding the pod subnets after launch:

```
I0430 13:11:12.523893    4580 route_network_windows.go:51] Watching for new subnet leases
I0430 13:11:12.603540    4580 route_network_windows.go:94] Subnet added: 10.244.0.0/24 via 10.127.130.37
I0430 13:11:14.593103    4580 route_network_windows.go:94] Subnet added: 10.244.18.0/24 via 10.127.130.35
I0430 13:11:16.276277    4580 route_network_windows.go:94] Subnet added: 10.244.19.0/24 via 10.127.130.38
```

Flannel "subnet lease" events

Overlay (Flannel vxlan)

In overlay, inter-node connectivity is implemented using "REMOTESUBNETROUTE" rules in VFP. Instead of checking static routes, we can reference "REMOTESUBNETROUTE" rules directly in vfpoutput.txt, where each pod subnet (e.g. 10.244.2.0/24) assigned to a node should have its corresponding destination IP (e.g. 10.127.130.38) specified as the destination in the outer packet:

```
RULE : REMOTESUBNETROUTE_ENCAP_4A240_10.244.2.0_24_4096_10.127.130.38_00-15-5D-F9-D2-63
    Friendly name : REMOTESUBNETROUTE_ENCAP_4A240_10.244.2.0_24_4096_10.127.130.38_00-15-5D-F9-D2-63
    Priority : 10
    Flags : 1 terminating
    Type : routeencap
    Conditions:
        Destination IP : 10.244.2.0-10.244.2.255
    Flow TTL: 0
    Rule Data:
    Encap Type: VXLAN
    Fixing inner MAC
    Using source PA MAC
    Decrementing TTL
    Encap Source IP : 10.127.130.36
    Encap Destination(s) :
        { IP address=10.127.130.38, MAC address=00-15-5D-F9-D2-63, GRE key=4096 }

    FlagsEx : 0

RULE : REMOTESUBNETROUTE_ENCAP_4B9DD_10.244.0.0_24_4096_10.127.130.37_F6-3C-2E-AB-57-6A
    Friendly name : REMOTESUBNETROUTE_ENCAP_4B9DD_10.244.0.0_24_4096_10.127.130.37_F6-3C-2E-AB-57-6A
    Priority : 10
    Flags : 1 terminating
    Type : routeencap
    Conditions:
        Destination IP : 10.244.0.0-10.244.0.255
    Flow TTL: 0
    Rule Data:
    Encap Type: VXLAN
    Fixing inner MAC
    Using source PA MAC
    Decrementing TTL
    Encap Source IP : 10.127.130.36
    Encap Destination(s) :
        { IP address=10.127.130.37, MAC address=F6-3C-2E-AB-57-6A, GRE key=4096 }

    FlagsEx : 0
```

VFP RemoteSubnetRoute rules used for VXLAN packet encapsulation

For additional details on inter-node container to container connectivity in overlay, please take a look at this video.

When can I encounter this issue?

One common configuration problem that manifests in this symptom is having mismatched networking configuration on Linux/Windows.

To double-check the network configuration on Linux, users can consult the CNI config file stored in /etc/cni/net.d/. In the case of Flannel on Linux, this file can also be embedded into the container, so users may need to exec into the Flannel pod itself to access it:

```
kubectl exec -n kube-system kube-flannel-ds-amd64-<someid> cat /etc/kube-flannel/net-conf.json
kubectl exec -n kube-system kube-flannel-ds-amd64-<someid> cat /etc/kube-flannel/cni-conf.json
```

Example #2: Containers cannot reach the outside world

Whenever outbound connectivity does not work, one of the first starting points is to ensure that there exists a NIC in the container. For this, we can consult the "ip.txt" output and compare it with the output of an "docker exec <id> ipconfig /all" in the problematic (running) container itself:

```
===============================================================================
Network Information for Compartment 2
===============================================================================
   Host Name . . . . . . . . . . . . : WIN-24ELFEPI8D9
   Primary Dns Suffix  . . . . . . . :
   Node Type . . . . . . . . . . . . : Hybrid
   IP Routing Enabled. . . . . . . . : No
   WINS Proxy Enabled. . . . . . . . : No
   DNS Suffix Search List. . . . . . : default.svc.cluster.local

Ethernet adapter vEthernet (506dc7f4402fa0edbad341f638e51f30620ef195173b268961deaa9176f1803d_cbr0):

   Connection-specific DNS Suffix  . : default.svc.cluster.local
   Description . . . . . . . . . . . : Hyper-V Virtual Ethernet Adapter #5
   Physical Address. . . . . . . . . : 00-15-5D-E1-56-F4
   DHCP Enabled. . . . . . . . . . . : Yes
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::89e0:e44:fb9:8a41%46(Preferred)
   IPv4 Address. . . . . . . . . . . : 10.244.4.7(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.244.4.2
   DNS Servers . . . . . . . . . . . : 10.96.0.10
   NetBIOS over Tcpip. . . . . . . . : Disabled
   Connection-specific DNS Suffix Search List :
                                       default.svc.cluster.local
```

Reference Container NIC in network compartment 2

In l2bridge networking (used by Flannel host-gw backend), the container gateway should be set to the 2. address exclusively reserved for the bridge endpoint (cbr0_ep) in the same pod subnet.

In overlay networking (used by Flannel vxlan backend), the container gateway should be set to the .1 address exclusively reserved for the DR (distributed router) vNIC in the same pod subnet.

L2bridge

Going outside of the container, on l2bridge one should also verify that the route tables on the node itself are setup correctly for the bridge endpoint. Here is a sample with the relevant entries containing quad-zero routes for a node with pod subnet 10.244.19.0/24:

```
PS C:\Users\Administrator> Get-NetRoute

ifIndex DestinationPrefix                              NextHop                                  RouteMetric if
                                                                                                            Me
                                                                                                            tr
                                                                                                            ic
------- -----------------                              -------                                  ----------- --
45      255.255.255.255/32                             0.0.0.0                                          256 15
41      255.255.255.255/32                             0.0.0.0                                          256 50
24      255.255.255.255/32                             0.0.0.0                                          256 15
1       255.255.255.255/32                             0.0.0.0                                          256 75
45      224.0.0.0/4                                    0.0.0.0                                          256 15
41      224.0.0.0/4                                    0.0.0.0                                          256 50
24      224.0.0.0/4                                    0.0.0.0                                          256 15
1       224.0.0.0/4                                    0.0.0.0                                          256 75
41      172.17.63.255/32                               0.0.0.0                                          256 50
41      172.17.48.1/32                                 0.0.0.0                                          256 50
41      172.17.48.0/20                                 0.0.0.0                                          256 50
1       127.255.255.255/32                             0.0.0.0                                          256 75
1       127.0.0.1/32                                   0.0.0.0                                          256 75
1       127.0.0.0/8                                    0.0.0.0                                          256 75
45      10.244.19.255/32                               0.0.0.0                                          256 15
45      10.244.19.2/32                                 0.0.0.0                                          256 15
45      10.244.19.0/24                                 0.0.0.0                                          256 15
24      10.244.18.0/24                                 10.127.130.35                                    256 15
24      10.244.17.0/24                                 10.127.130.36                                    256 15
24      10.244.0.0/24                                  10.127.130.37                                    256 15
24      10.127.130.255/32                              0.0.0.0                                          256 15
24      10.127.130.38/32                               0.0.0.0                                          256 15
24      10.127.130.0/24                                0.0.0.0                                          256 15
45      0.0.0.0/0                                      10.244.19.1                                      256 15
24      0.0.0.0/0                                      10.127.130.1                                     256 15
```

Quad-zero static routes for a node with pod subnet 10.244.19.0/24

The next thing to check on l2bridge is verify that the OutboundNAT policy and the ExceptionList is programmed correctly. For a given endpoint (e.g. 10.244.4.7) we should verify in the endpoint.txt that there exists an OutboundNAT HNS Policy and that the ExceptionList matches with what we entered into the deployment scripts originally:

```
"ID":  "52B0992D-65A7-4E39-BBA8-B19856BF240A",
"IPAddress":  "10.244.4.7",
"MacAddress":  "00-15-5D-E1-56-F4",
"Name":  "506dc7f4402fa0edbad341f638e51f30620ef195173b268961deaa9176f1803d_cbr0",
"Policies":  [
                 {
                     "ExceptionList":  [
                                           "10.96.0.0/12",
                                           "10.127.130.0/24",
                                           "10.244.0.0/16"
                                       ],
                     "Type":  "OutBoundNAT"
                 },
                 {
                     "DestinationPrefix":  "10.96.0.0/12",
                     "NeedEncap":  true,
                     "Type":  "ROUTE"
                 },
                 {
                     "DestinationPrefix":  "10.127.130.36/32",
                     "NeedEncap":  true,
                     "Type":  "ROUTE"
                 },
                 {
                     "Type":  "L2Driver"
                 }
             ],
```

HNS Policies for a typical l2bridge endpoint

Finally, we can also consult the vfpOutput.txt to verify that the L2Rewrite rule exists so that the container MAC is rewritten to the host's MAC as specified in the l2bridge container networking docs.

In the EXTERNAL_L2_REWRITE layer, there should be a rule which matches the container's source MAC (e.g. "00-15-5D-AA-87-B8") and rewrites it to match the host's MAC address (e.g. "00-15-5D-05-C3-0C"):

```
LAYER : EXTERNAL_L2_REWRITE_LAYER
    Friendly name : EXTERNAL_L2_REWRITE_LAYER
    Priority : 500

  GROUP : EXTERNAL_L2_REWRITE_GROUP_IPV4_IN
      Friendly name : EXTERNAL_L2_REWRITE_GROUP_IPV4_IN
      Priority : 0
      Direction : IN
      Type : IPv4
        Conditions:
            <none>
      Match type : Longest prefix match on Destination IP

    RULE : FC7CEE89-4FF1-4A13-B3FD-A7622C49039C_to_EN
        Priority : 200
        Flags : 1 terminating
        Type : transposition
        Conditions:
            Source MAC : 00-15-5D-AA-87-B8
        Flow TTL: 0
        Transposition:
        Modify:
            Source MAC: 00-15-5D-05-C3-0C
        FlagsEx : 0
```

Reference EXTERNAL_L2_REWRITE_LAYER with rules transposing a container MAC to a host MAC

Overlay

For overlay, we can check whether there exists an ENCAP rule that encapsulates outgoing packets correctly with the hosts IP. For example, for a given pod subnet (10.244.3.0/24) with host IP 10.127.130.36:

```
    RULE : ENCAP_10.244.3.1
        Friendly name : ENCAP_10.244.3.1
        Priority : 100
        Flags : 1 terminating
        Type : mapencap
        Conditions:
            Destination IP : 10.244.3.0-10.244.3.255
        Flow TTL: 0
        Rule Data:
        Map space : 21A6C8A3-813D-4D85-980D-A1C827AA6AEC
        GRE key : 4096
        Encap Type: VXLAN
        CA routing enabled
        Encap Source IP : 10.127.130.36
        FlagsEx : 0
```

Reference encapsulation rule used by overlay container networks

When can I encounter this issue?

One example configuration error for Flannel (vxlan) overlay that may results in failing east/west connectivity is failing to delete the old SourceVIP.json file whenever the same node is deleted and re-joined to a cluster.

> NOTE: When deploying L2bridge networks on Azure, user's also need to configure user-defined routes for each pod subnet allocated to a node for networking to work. Some users opt to use overlay in public cloud environments for that reason instead, where this step isn't needed.

Example #3: Services / Load-balancing does not work

Let's say we have created a Kubernetes service called "win-webserver" with VIP 10.102.220.146:

```
Administrator: Windows PowerShell
PS C:\k> kubectl get svc/win-webserver
NAME            TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
win-webserver   NodePort   10.102.220.146   <none>        80:31486/TCP   91m
```

Example Kubernetes service on Windows

Load Balancing is usually performed directly on the node itself by replacing the destination VIP (Service IP) with a specified DIP (pod IP). HNS Loadbalancers can be viewed using the "hnsdiag" cmdlet:

```
PS C:\k> hnsdiag list loadbalancers
ID                                   | Virtual IPs      | Direct IP IDs
6bac3ce0-ae79-4b4a-a04a-b4e3c761710a |  10.96.0.1       | 4dbbeca1-ccfa-405c-86f3-ddd9eb331094
c6308ecf-4ae7-49d1-965e-3ec3e15c7f58 |  10.96.0.10      | c885f112-6fa3-45de-85e9-16084c91c353 92128f23-27a5-4a4a-8b16-7bdbb0889757
44ce1995-dd12-4198-bf94-1faa06966d6a |  10.96.0.10      | c885f112-6fa3-45de-85e9-16084c91c353 92128f23-27a5-4a4a-8b16-7bdbb0889757
8792b31e-4aff-4691-b04b-fea174029583 |  10.102.220.146  | 4d1d1b8c-c12d-461a-a608-11825b6a9189 fde55279-a198-4bd8-a389-2b58d970c69b
PS C:\k> _
```

Typical HNS Loadbalancer objects on Windows

For a more verbose output, users can also inspect policy.txt to check for "ELB" policies (LoadBalancers) for additional information:

```
        "ID":  "8792B31E-4AFF-4691-B04B-FEA174029583",
        "IsApplied":  false,
        "Policies":  [
                         {
                             "ExternalPort":  80,
                             "InternalPort":  80,
                             "Protocol":  6,
                             "Type":  "ELB",
                             "VIPs":  [
                                          "10.102.220.146"
                                      ]
                         }
                     ],
        "References":  [
                           "/endpoints/4d1d1b8c-c12d-461a-a608-11825b6a9189",
                           "/endpoints/fde55279-a198-4bd8-a389-2b58d970c69b"
                       ],
```

Example HNS LoadBalancer configuration in policy.txt

The next step usually consists of verifying that the endpoints (e.g. "4d1d1b8c-c12d-461a-a608-11825b6a9189") still exist in endpoint.txt and are reachable by IP from the same source:

```
        "ID":  "4D1D1B8C-C12D-461A-A608-11825B6A9189",
        "IPAddress":  "10.244.19.31",
        "MacAddress":  "00-15-5D-AA-8F-3D",
        "Name":  "d9cbb5e6f8e9643c6484968fb6b5bd2559394461eab3d82f409d56636dff87cc_cbr0",
        "Policies":  [
                         {
                             "ExceptionList":  [
                                                   "10.96.0.0/12",
                                                   "10.127.130.0/24",
                                                   "10.244.0.0/16"
                                               ],
                             "Type":  "OutBoundNAT"
                         },
                         {
                             "DestinationPrefix":  "10.96.0.0/12",
                             "NeedEncap":  true,
                             "Type":  "ROUTE"
                         },
                         {
                             "DestinationPrefix":  "10.127.130.38/32",
                             "NeedEncap":  true,
                             "Type":  "ROUTE"
                         },
                         {
                             "Type":  "L2Driver"
                         }
                     ],
```

Example pod DIP endpoint referenced by a HNS Loadbalancer

Finally, we can also check whether the VFP "lbnat" rules exist in the "LB" layer for our service IP 10.102.220.146 (with NodePort 31486):

```
    RULE :
        Friendly name : LB_72C28_10.127.130.38_10.102.220.146_80_80_6
        Priority : 100
        Flags : 1 terminating
        Type : lbnat
        Conditions:
            Protocols : 6
            Destination IP : 10.102.220.146
            Destination ports : 80
        Flow TTL: 240
        Rule Data:
        Decrementing TTL
        Fixing MAC
        Modifying destination IP
        Modifying destination port
        Creating a flow pair
        Map space : 53F19CE3-474C-4059-98F5-870001DD5A4F
        Count of DIP Ranges: 2
        DIP Range(s) :
                { 10.244.19.31 : 80 }
                { 10.244.19.32 : 80 }
        FlagsEx : 0

    RULE :
        Friendly name : LB_B6C16_10.127.130.38_10.127.130.38_31486_80_6
        Priority : 100
        Flags : 1 terminating
        Type : lbnat
        Conditions:
            Protocols : 6
            Destination IP : 10.127.130.38
            Destination ports : 31486
        Flow TTL: 240
        Rule Data:
        Decrementing TTL
        Fixing MAC
        Modifying destination IP
        Modifying destination port
        Creating a flow pair
        Map space : 53F19CE3-474C-4059-98F5-870001DD5A4F
        Count of DIP Ranges: 2
        DIP Range(s) :
                { 10.244.19.31 : 80 }
                { 10.244.19.32 : 80 }
        FlagsEx : 0
```

Reference VFP rules used for load-balancing containers

When can I encounter this issue?

One possible issue that can cause erroneous load balancing is a misconfigured kube-proxy which is responsible for programming these policies. For example, one may fail to pass in the --hostname-override parameter, causing endpoints from the local host to be deleted.

> NOTE that service VIP resolution from the Windows node itself is not supported on Windows Server 2019, but planned for Windows Server, version 1903.

Example #4: DNS resolution is not working from within the container

For this example, let's assume that the kube-DNS cluster addon is configured with service IP 10.96.0.10.

Failing DNS resolution is often a symptom of one of the previous examples. For example, (external) DNS resolution would fail if outbound connectivity isn't present or resolution could also fail if we cannot reach the kube-DNS service.

Thus, the first troubleshooting step should be to analyze whether the kube-DNS service (e.g. 10.96.0.10) is programmed as a HNS LoadBalancer correctly on the problematic node:

```
    {
        "ActivityId":  "799495A6-986B-491C-BC4F-C12F71B746B4",
        "AdditionalParams":  {

                             },
        "Health":  {
                       "LastErrorCode":  0,
                       "LastUpdateTime":  131895666619838345
                   },
        "ID":  "9DD7BB8A-B10F-4C65-835E-98545989A840",
        "IsApplied":  false,
        "Policies":  [
                         {
                             "ExternalPort":  53,
                             "InternalPort":  53,
                             "Protocol":  17,
                             "Type":  "ELB",
                             "VIPs":  [
                                          "10.96.0.10"
                                      ]
                         }
                     ],
        "References":  [
                           "/endpoints/7CD7CAB6-59E2-42B5-AC84-AA6532F78C6D",
                           "/endpoints/CB85953C-2FF3-4B47-AB0A-79B1137B549E"
                       ],
```

HNS LoadBalancer object representing kube-DNS service

Next, we should also check whether the DNS information is set correctly in the ip.txt entry for the container NIC itself:

```
==========================================================================
Network Information for Compartment 3
==========================================================================
   Host Name . . . . . . . . . . . . : WIN-24ELFEPI8D9
   Primary Dns Suffix  . . . . . . . :
   Node Type . . . . . . . . . . . . : Hybrid
   IP Routing Enabled. . . . . . . . : No
   WINS Proxy Enabled. . . . . . . . : No
   DNS Suffix Search List. . . . . . : default.svc.cluster.local

Ethernet adapter vEthernet (745adc2248ed418f3ce014f92715b0cde87720a677aa1bb44a71f4906266966c_cbr0):

   Connection-specific DNS Suffix  . : default.svc.cluster.local
   Description . . . . . . . . . . . : Hyper-V Virtual Ethernet Adapter #6
   Physical Address. . . . . . . . . : 00-15-5D-E1-58-20
   DHCP Enabled. . . . . . . . . . . : No
   Autoconfiguration Enabled . . . . : Yes
   Link-local IPv6 Address . . . . . : fe80::8493:b5bd:ede6:d73e%50(Preferred)
   IPv4 Address. . . . . . . . . . . : 10.244.4.8(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.244.4.2
   DNS Servers . . . . . . . . . . . : 10.96.0.10
   NetBIOS over Tcpip. . . . . . . . : Disabled
   Connection-specific DNS Suffix Search List :
                                       default.svc.cluster.local
```

Reference DNS configuration in a Windows pod

We should also check whether it's possible to reach the kube-DNS pods directly and whether that works. This may indicate that there is some problem in resolving the DNS service VIP itself. For example, assuming that one of the DNS pods has IP 10.244.0.3:

```
PS C:\k> docker exec 8bc powershell.exe -Command resolve-dnsname kubernetes.default.svc.cluster.local -Server 10.244.0.3

Name                                   Type   TTL   Section    IPAddres
                                                               s
----                                   ----   ---   -------    --------
kubernetes.default.svc.cluster.local   A      5     Answer     10.96.0.
                                                               1


PS C:\k> _
```

Sending a DNS request directly to the DNS pod endpoint

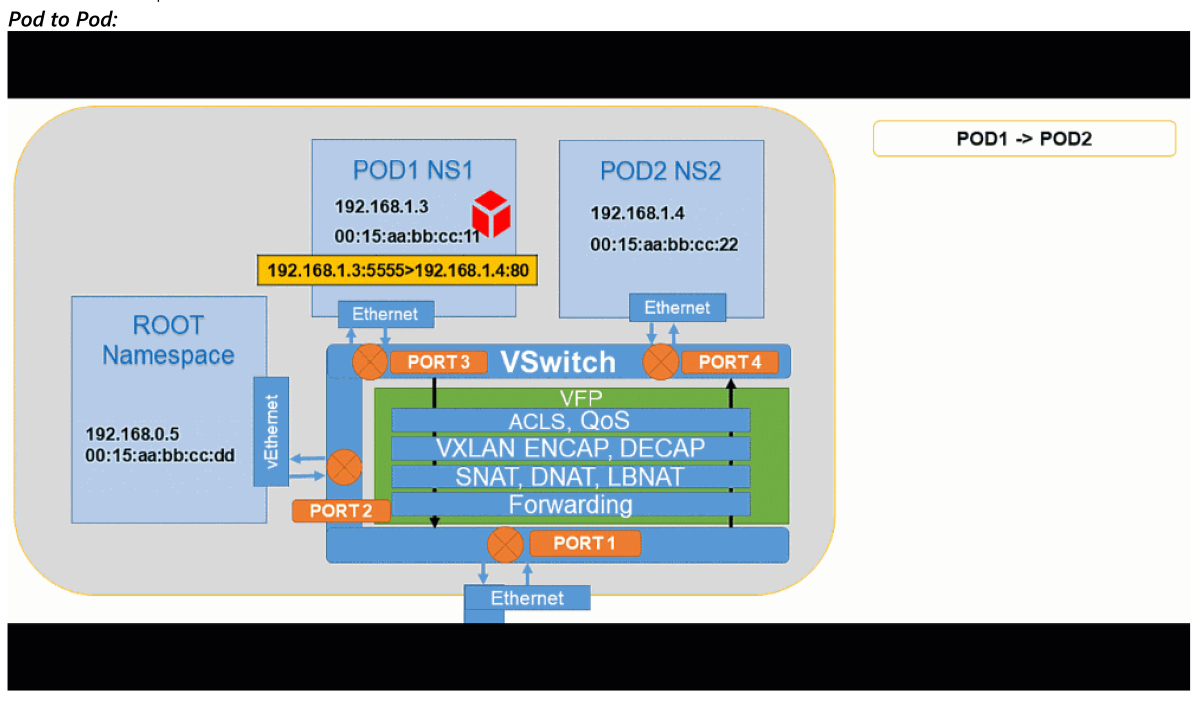
When can I encounter this issue?
One possible misconfiguration that results in DNS resolution problems is an incorrect DNS suffix or DNS service IP which was specified in the CNI config here and here.

Step 8: Capture packets and analyze flows
The last step requires in-depth knowledge of the operations that packets from containers undergo and the network flow. As such, it is also the most time-consuming to perform and will vary depending on the observed issue. At a high level, it consists of:

1. Running startpacketcapture to start the trace
2. Reproducing the issue – e.g. sending packets from source to destination
3. Running stoppacketcapture to stop the trace
4. Analyzing correct processing by the data path at each step

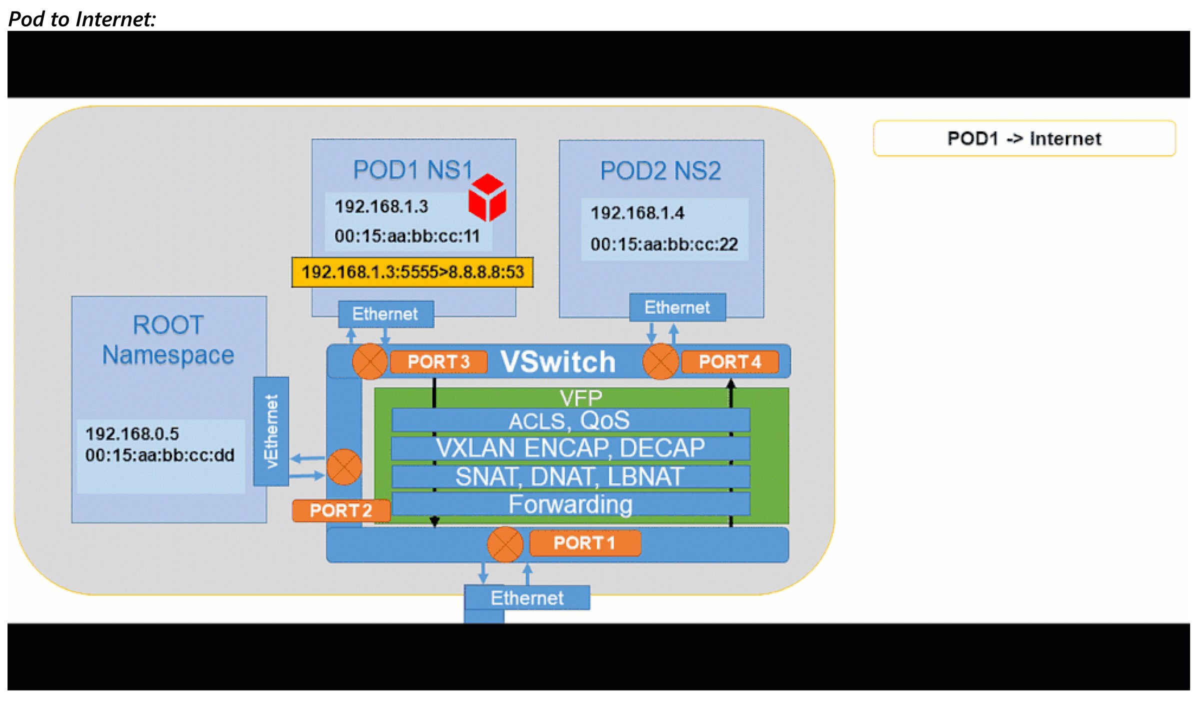
Here are a few example animations that showcase some common container network traffic flows:

Pod to Pod:



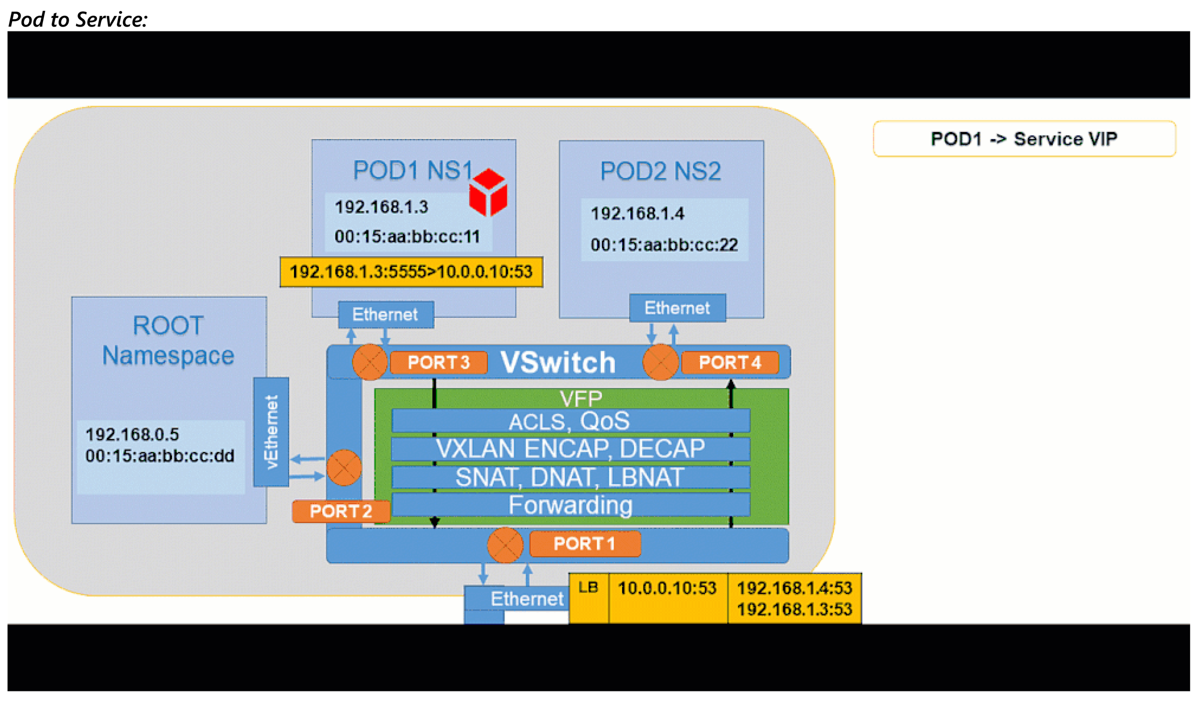
Animated visualization showing pod to pod connectivity

Pod to Internet:



Animated visualization showing pod to outbound connectivity

Pod to Service:



Animated visualization showing pod to service connectivity

Showing how to analyze and debug these packet captures will be done in future part(s) of this blog post series through scenario-driven videos showing packet captures for supported networking flows on Windows.

For a quick teaser, here is a video recording taken at KubeCon that shows debugging an issue live using startpacketcapture.cmd: https://www.youtube.com/watch?v=tTZFoiLObX4&t=1733

Summary
We looked at:

1. Automated scripts that can be used to verify basic connectivity and correct installation of Kubernetes
2. HNS networking objects and VFP rules used to network containers
3. How to query event logs from different Kubernetes components
4. How to analyze the control path at a high level for common configuration errors using CollectLogs.ps1
5. Typical network packet flows for common connectivity scenarios

Performing the above steps can go a great length towards understanding the underlying issue for an observed symptom, improve efficacy when it comes to implementing workarounds, and accelerate the speed at which fixes are implemented by others, having already performed the initial investigation work.

What's next?
In the future, we will go over supported connectivity scenarios and specific steps on how to troubleshoot each one of them in-depth. These will build on top of the materials presented here but also contain videos analyzing packet captures, data-path analysis as well as other traces (e.g. HNS tracing).

We are looking for your feedback!
Last but not least, the Windows container networking team needs your feedback! What would you like to see next for container networking on Windows? Which bugs are preventing you from realizing your goals? Share your voice in the comments below, or fill out the following survey and influence our future investments!

Thank you for reading,
David Schott

*Special thanks to Dinesh Kumar Govindasamy (Microsoft) for his fantastic work creating & presenting many of the materials used as a basis for this blog at KubeCon Shanghai '18!

1 Comment

HVL71 Occasional Visitor
Dec 21 2019 03:01 PM

Thanks for a great article!

Unfortunately I already get stuck in step 1 when checking if kube-proxy is running. It would be very helpful with some troubleshooting tips in that scenario. I'm running 1.17 and have tried to follow a combination of these 2 documents multiple times:
https://docs.microsoft.com/en-us/virtualization/windowscontainers/kubernetes/joining-windows-workers...
https://kubernetes.io/docs/setup/production-environment/windows/user-guide-windows-nodes/