WHITEPAPER

Get Me to the Cluster

Providing External Access to Services in Kubernetes with NGINX and BGP

By Chris Akker
Solutions Architect – F5 NGINX

Table of Contents

You are a Network Architect, Engineer, or Operator – and those Kubernetes Apps people are driving you crazy! They keep asking you for solutions, environments, virtual IP addresses, DNS names, and customer access to applications running in Kubernetes clusters. They whine and complain about the ticketing process, status updates, and delays – and then they escalate! They build apps overnight, and expect them online in the morning, before you've even had your first cup of coffee . . .

You are ready to pull your hair out!

You are a Modern Apps Developer, working with the cloud, containers, and Kubernetes to build applications that support your company's business demands. You use an arsenal of open source code and tools as well as some commercial products. You are being asked to build and deliver high-quality digital experiences that set your company apart and keep it ahead of the competition. The ecosystems of hardware, software, and even some cloud providers move at relatively glacial speeds, impeding progress and slowing your application delivery times. Every time you need something, especially from the Network team, you are greeted with a daunting, infuriating gauntlet of requirements, processes, procedures, and those hideous tickets . . .

You are ready to pull your hair out!

Sound all too familiar? Network and Applications teams often have different goals, priorities, jobs, mandates, restrictions – the list goes on and continues to evolve. But at the end of the day, *both teams must be successful to deliver today's modern applications*.

How can you address the needs of both teams and make them successful, enabling the business to move forward at the required business and development velocity?

There **must** be a way to solve this problem . . .

Don't go bald, keep reading!

WE EXPLORE AND PROPOSE A SOLUTION TO RESOLVE THE CONFLICT BETWEEN NETWORK AND APPLICATIONS TEAMS

In this whitepaper, we explore and propose a solution to resolve the conflict between Network and Applications teams by providing both with recipes for success. It's a solution that uses modern networking tools, protocols, and existing architectures, and is designed to be inexpensive and easy to implement, manage, and support. It's based on standards and easily deployed and tested in most Kubernetes environments in a few hours.

The solution is designed for Kubernetes clusters **hosted on premises** in your data center, with Kubernetes nodes running on bare metal or traditional Linux virtual machines (VMs). Standard Layer 2 switches and Layer 3 routers provide the networking for communications in the data center.

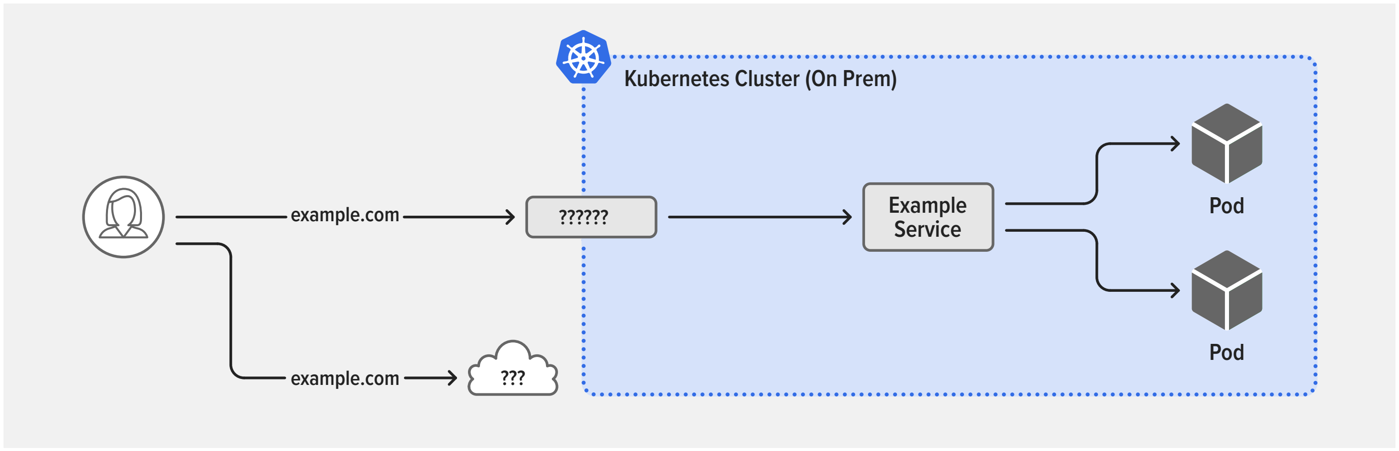
We intentionally are not trying to provide this solution for cloud-hosted Kubernetes clusters, because cloud providers don't allow us to control the core networking in their data centers nor the networking in their managed Kubernetes environment.

THE CHALLENGE: ENABLING EXTERNAL ACCESS TO KUBERNETES SERVICES

IN AN ON-PREMISES KUBERNETES CLUSTER, YOU GENERALLY DEPLOY APPLICATION PODS USING A DEPLOYMENT OR DAEMONSET MANIFEST

In an on-premises Kubernetes cluster, you generally deploy application pods using a Deployment or DaemonSet manifest. Using any of the standard Container Network Interface (CNI) plug-ins, each pod is assigned a unique ClusterIP address, allocated from a pool of IP addresses chosen by the scheduler that starts and runs pods on the Kubernetes node. No matter how you manage the pods or which CNI you are using, the end result is the same: each pod has a unique IP address assigned by Kubernetes.

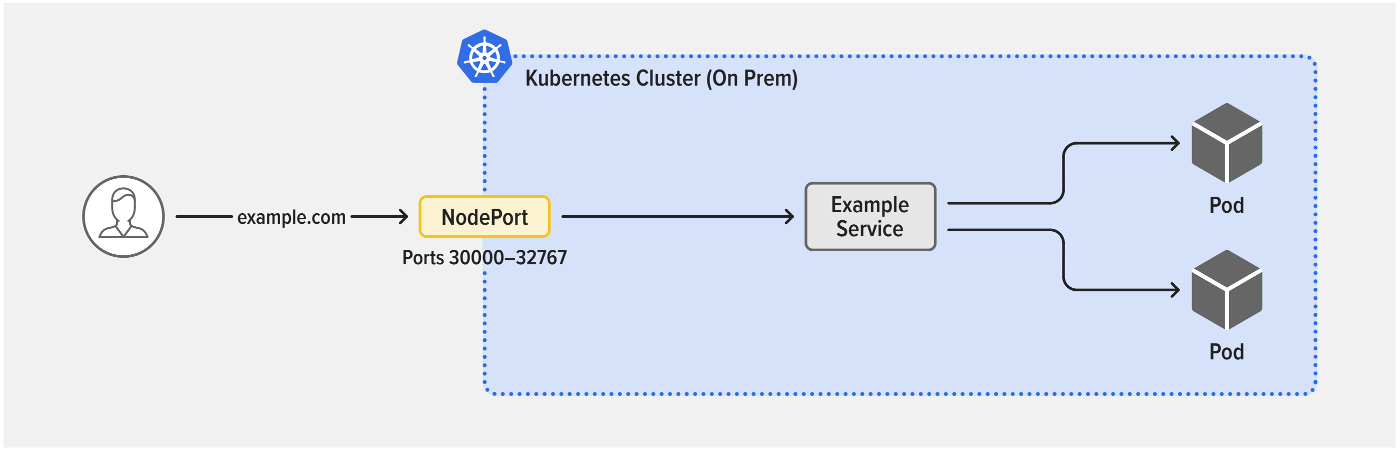
ClusterIP addresses are drawn from a private, non-routable IP address space (subnet), often called the *overlay network*. Non-routable means that the ClusterIP address cannot be reached from outside the cluster. So a problem arises when you want clients or users external to the cluster to have access to one or more pods or services.



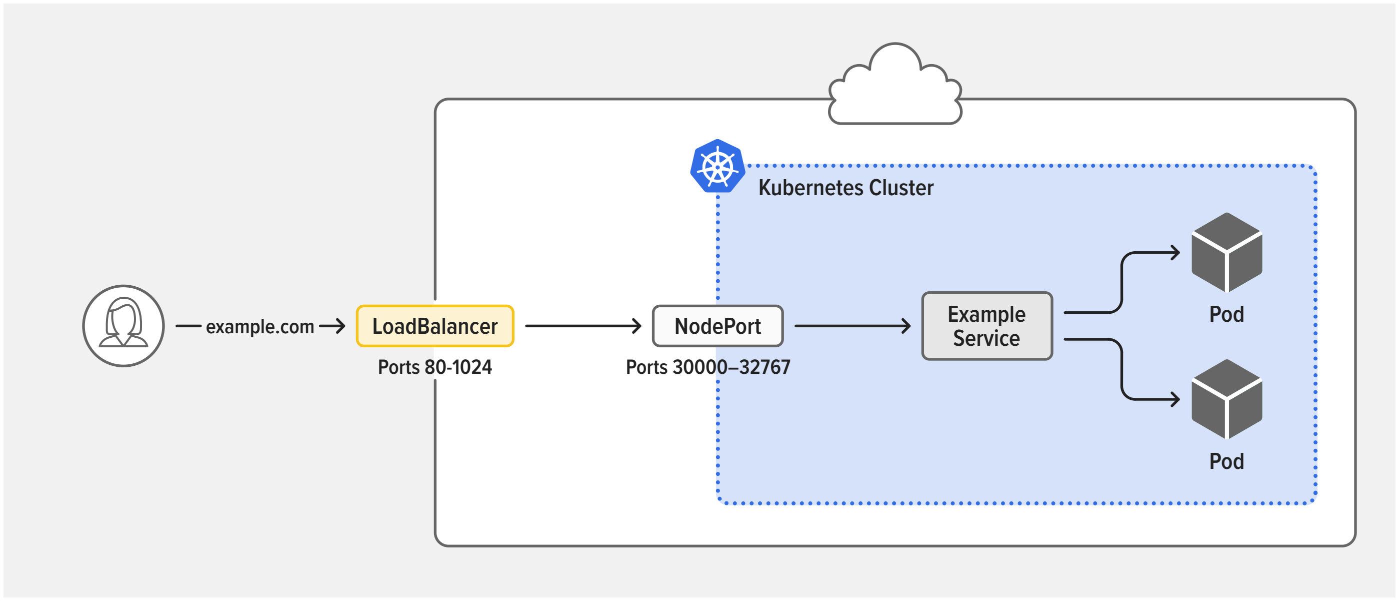
OUR SOLUTION USES THE INGRESS AS THE PREFERRED METHOD

The Kubernetes network standard for exposing application pods offers four different options: Service, NodePort, LoadBalancer, and Ingress. Our solution uses the last one, the Ingress, as the preferred method for exposing pods to the outside world. But it's worth reviewing why the other options are not suitable:

- Service – Represents a common group of pods running the same apps. This is great for internal, pod-to-pod communication, but it's defined and visible only inside the cluster and so doesn't really help us expose apps and services externally.
- NodePort – Opens a specific port on every node in the cluster, and forwards any traffic sent to the node on that port to the corresponding app. But NodePort is not an ideal solution for several reasons:
  - You have to use high-numbered TCP ports; the well-known lower port numbers are not allowed
  - You have to coordinate these port numbers with other apps
  - You can't share common TCP ports among different apps (each app must have its own unique port)
  - Not all pods run on every node, so there might be an additional cluster network hop required, which introduces latency
  - The configuration is static, which compromises the dynamic and ephemeral nature of Kubernetes



- LoadBalancer – Creates a network path from the outside world to your Kubernetes nodes using the NodePort definitions on each node. It often uses “well-known” low-numbered application TCP ports and translates them into the high-numbered TCP ports of NodePort.



The LoadBalancer object is the “easy button” for cloud-hosted Kubernetes clusters, because AWS, Google Cloud Platform, Microsoft Azure, and most other cloud providers support it as an easily configured feature that works well and provides a public IP address and matching DNS A record for a service – exactly what we need for external access! But a LoadBalancer doesn’t actually run in the cluster. Instead, it’s usually part of the cloud provider’s software-defined networking (SDN) infrastructure, which you cannot see, control, or directly access.

Perhaps more saliently to our solution, for on-premises clusters there is no equivalent to the LoadBalancer object!

You probably already have a load-balancer appliance on your network, so can you use it to provide external access? There are lots of docs on how to configure a TCP load-balancer appliance at the edge of the Kubernetes cluster, with instructions for mapping the pod IP address to the IP address and port of Kubernetes nodes, configuring NodePort, assigning pool membership, setting up virtual IP addresses (VIPs), and so on.

But with that approach you have to use high-numbered TCP ports (~30,000 and higher), and then the ports are opened on every host, whether the pod is running on that host or not . . . and when traffic arrives on a Kubernetes node where the pod is not running, the outcome is not ideal – the traffic gets bounced to another node, adding internode crosstalk and latency.

Another problem is that if you have multiple teams sharing the same cluster, you need a TCP port allocation and reservation system, yet another manual process that diminishes the value of Kubernetes ephemerality in the first place.

Further, even when using NodePort, it is nearly impossible to reconfigure routers and load balancers quickly enough, no matter how much automation is in place and how many API calls you make to those devices. This results in a serious configuration mismatch: new application pods are ready but the network routing parameters have not been updated. Even worse, when pods restart and get a new IP address, the network routing system might still be pointed to the pod's previous IP address, potentially sending customers to the dreaded *HTTP 500 Black Hole* just created in the center of your cluster . . . not a good customer experience!

The difficulty of making Kubernetes applications accessible by external clients is often the nexus of conflict between the Network and Application teams. ***As pods change, so must networking change to keep external access reliable and consistent***.

It's a pain, in fact . . . oh wait, pull out more hair!

A SOLUTION COMBINING KUBERNETES INGRESS, NGINX, AND BGP

But good news – there's one more option to explore: the Kubernetes Ingress object, which addresses some of these ugly NodePort allocation and tracking issues. It's designed specifically for traffic flowing from outside the cluster to pods in the cluster (north-south traffic). The design of Kubernetes dictates that the Ingress object is implemented by an Ingress controller. Not all Ingress controllers are the same, however, and our solution leverages the enterprise-class NGINX Ingress Controller from F5 NGINX, which is based on the commercially supported version of NGINX, NGINX Plus.

No more hair pulling, we have a solution for you!

WE'RE PROVIDING COMPLETE, STEP-BY-STEP INSTRUCTIONS

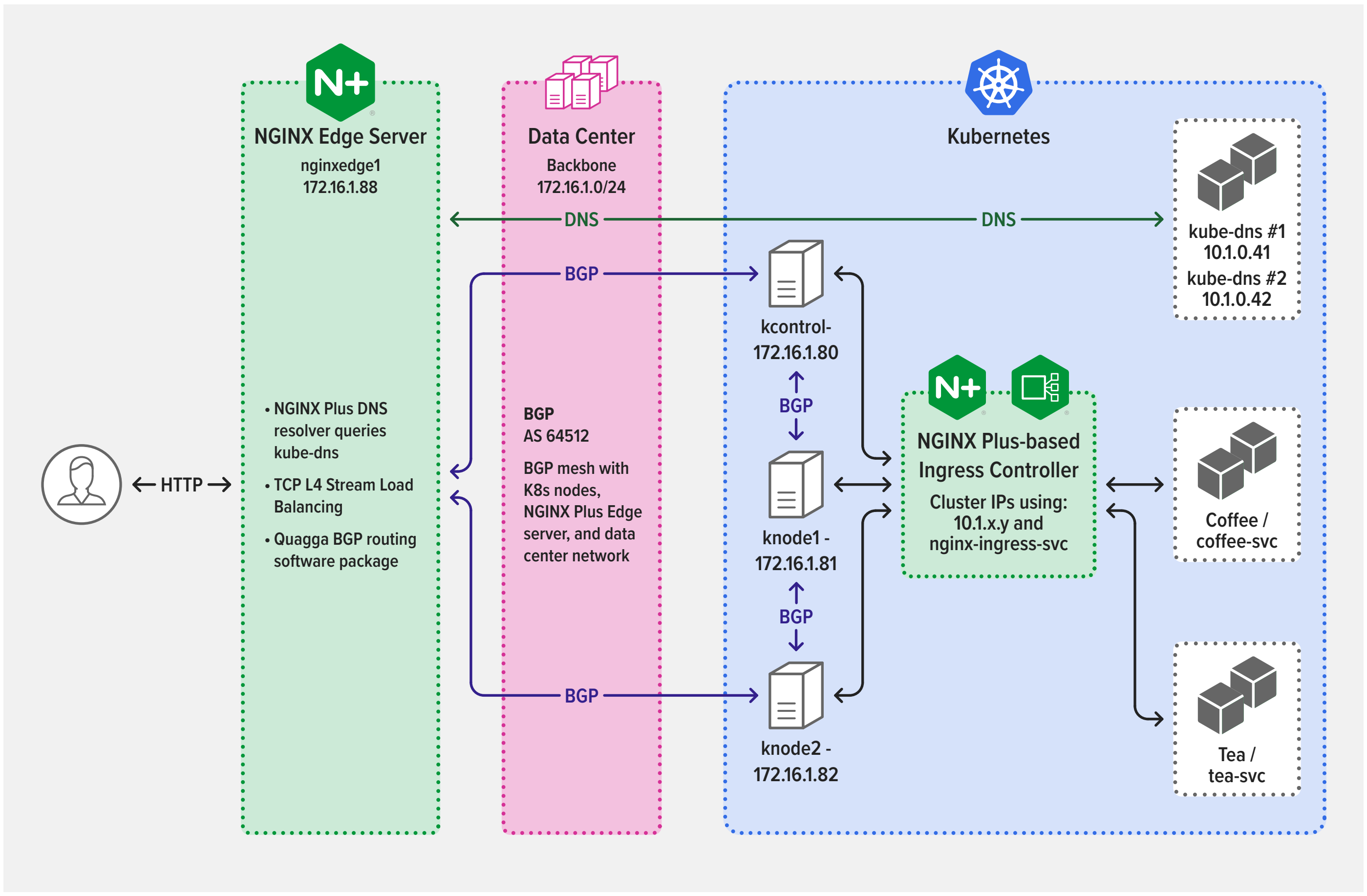
Even better, we're not just having a high-level theoretical discussion about how a solution might work. We're providing complete, step-by-step instructions – see Deploying the Solution. First, though, a bit more information about the solution's architecture.

Our solution to the challenges of providing external access to Kubernetes applications combines NGINX and the Border Gateway Protocol (BGP). Yes, that's right, the world famous NGINX software and a Layer 3 routing protocol. "No way!" you say, "it can't be that easy!"

Actually, the solution uses the NGINX Plus-based NGINX Ingress Controller and a second NGINX Plus server as a reverse proxy. More exactly, the solution has these components:

1. BGP routing, implemented as an internal BGP (iBGP) autonomous system (AS) in the data center
2. The CNI plug-in from Project Calico for networking in the Kubernetes cluster, with BGP enabled
3. NGINX Ingress Controller based on NGINX Plus, in the Kubernetes cluster
4. NGINX Plus as a reverse proxy at the edge of the Kubernetes cluster, with Quagga for BGP

The diagram depicts the solution architecture. Note that it indicates which protocols the solution components use to communicate, and (unlike many such diagrams) not the order in which data is exchanged during request processing.



HTTP
NGINX Edge Server
nginxedge1
172.16.1.88
• NGINX Plus DNS resolver queries kube-dns
• TCP L4 Stream Load Balancing
• Quagga BGP routing software package
Data Center
Backbone
172.16.1.0/24
BGP
AS 64512
BGP mesh with K8s nodes, NGINX Plus Edge server, and data center network
DNS
BGP
Kubernetes
kcontrol-
172.16.1.80
knode1 -
172.16.1.81
knode2 -
172.16.1.82
NGINX Plus-based Ingress Controller
Cluster IPs using: 10.1.x.y and nginx-ingress-svc
kube-dns #1
10.1.0.41
kube-dns #2
10.1.0.42
Coffee /
coffee-svc
Tea /
tea-svc

Let's look at the four components in detail.

The iBGP Network

The solution leverages the existing iBGP network that most modern enterprise data centers already have in place. (If you don't have one, you'll need to set it up to use the solution; see Configuring the iBGP Network for more details.) We won't take up 20 minutes of your time here explaining why data centers use iBGP – just suffice it to say that iBGP works well, has many benefits, and is well-known and supported by most Network teams. Our solution leverages the many options for routing and controlling IP traffic that come with using a private autonomous system (AS) number for BGP.

Project Calico CNI Networking

We use Project Calico as the CNI overlay for the Kubernetes, because it supports BGP. An advantage of Calico is that it works as a routing method overlay network, rather than an encapsulation method overlay. This is important to the Networking team, because *Ethernet packets remain in their native format*, making them easy to see, capture, and troubleshoot with common tools like **`ping`**, **`curl`**, **`dig`**, **`traceroute`**, **`tcpdump`**, and **`ssldump`**. Encapsulation overlays that modify packets can make it more difficult to use these tried-and-tested tools – you have to unwrap the packets to see what's really going on.

Calico also enables you to control the IP address pools allocated for the pods, which helps you quickly identify any networking issues – simply knowing the pod's IP address tells you immediately on which Kubernetes node the pod is running.

NGINX Ingress Controller Based on NGINX Plus

THE ADVANCED FEATURES IN OUR NGINX INGRESS CONTROLLER BASED ON NGINX PLUS PROVIDE A SUPERIOR SOLUTION AND USER EXPERIENCE

While there are many Ingress controllers to choose from, the advanced features in our NGINX Ingress Controller based on NGINX Plus provide a superior solution and user experience. The most compelling feature for our solution is the ability of NGINX Plus to watch the service endpoint IP addresses of the pods and automatically reconfigure the list of "upstreams" with no loss of application traffic. The Application team can take advantage of NGINX Plus's many other enterprise-grade Layer 7 HTTP features, including active health checks, host and URI request routing, mutual TLS (mTLS), authentication, and header controls.

NGINX Plus as a Reverse Proxy at the Edge

This is an NGINX Plus server that sits at the edge of the Kubernetes cluster and provides the path between the switches and routers running in the data center and the internal network in the Kubernetes cluster. The remainder of this whitepaper refers to this as the *NGINX edge server*. It functions in the on-premises cluster as a replacement for the missing Kubernetes LoadBalancer object and uses BGP to communicate with other components in the cluster about routing.

We use NGINX Plus on the edge server to leverage the same set of enterprise features for automatically tracking the dynamic changes to ephemeral pods in the Kubernetes cluster. Specifically, we need to track the NGINX Ingress Controller instances to make sure we can route traffic to them automatically.

DEPLOYING THE SOLUTION

We recommend that you start with a fresh Kubernetes cluster to reduce the chances of causing a serious network outage on an existing cluster. The CNI network layer is responsible for managing and controlling almost all the traffic in the cluster, so to err on the side of caution do not deploy the solution in an existing cluster unless you are already using Calico CNI.

In particular, if you use an existing cluster you must remove any other CNI overlay (Cillium, Flannel, Weaver, etc.) that is already deployed, and doing so stops all traffic!

Configuring the iBGP Network

THIS SOLUTION MAKES USE OF THE EXISTING iBGP NETWORK IN YOUR DATA CENTER

The solution makes use of the existing iBGP network in your data center to interconnect the Kubernetes nodes, the cluster IP network, and the NGINX edge server.

Make note of the following existing iBGP parameters, because you need to configure iBGP on the peers in the Kubernetes cluster and the NGINX edge server to match, in Configuring iBGP Peering on the Kubernetes Nodes and Installing Quagga and Configuring BGP respectively.

- iBGP autonomous system (AS) number
- iBGP IP networking details (subnet info)
- Any other settings in the existing iBGP network that need to be configured on all peers for successful operation

You also need the IP addresses of all iBGP hosts, including the Kubernetes nodes and the NGINX edge server. For the suggested addressing scheme for the Kubernetes nodes, see Configuring Calico IP Pools.

If you're not on the Network team, ask a team member for help gathering these parameters if necessary. Similarly, you can partner with the Network team to create an iBGP network if the data center doesn't already have one.

Installing and Configuring a Kubernetes Cluster with Calico

1. Follow the instructions in the Quickstart for Calico on Kubernetes to set up a new Kubernetes control node and install Calico. Then add more (worker) nodes as needed for your environment. The solution uses one control node and two worker nodes.
2. Follow the instructions in the Calico documentation to install the **`calicoctl`** CLI on the Kubernetes control node.

Configuring Calico IP Pools

Now configure Calico *IP pools*, which are ranges of IP addresses that Calico uses for *workload endpoints* (the virtual network interfaces a workload uses to connect to the Calico network).

The solution uses a Class C IP subnet for each Kubernetes node's pod IP address allocation, with the third octet in the subnet matching the index number at the end of the Kubernetes node name, as shown in the table. The reason for this scheme will become apparent when we get to the BGP routing tables in Step 9 of *Installing Quagga and Configuring BGP*.

KUBERNETES NODE	IP SUBNET
kcontrol	10.1.**0**.0/24
knode1	10.1.**1**.0/24
knode2	10.1.**2**.0/24
...	**...**
knodeX	10.1.***X***.0/24

As another example, the subnet for **knode5** is 10.1.**5**.0/24.

Note: This addressing scheme is not mandatory, and the Calico default of a /26 subnet for each node works just fine. However, allocating a /24 subnet with the third octet matching the index in each worker node hostname makes it easier to read the BGP routing tables, find each pod's running host, and troubleshoot. Plus, in some environments, the smallest allowable subnet for BGP route table entries is a /24 network.

1. Create a file called **ippools.yaml** with a section for each node, specifying the appropriate values in these fields:
   - `metadata.name`
   - `spec.cidr`
   - `spec.nodeSelector`

This example configures the three nodes in the solution.

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: pool0
spec:
  cidr: 10.1.0.0/24
  blockSize: 24
  ipipMode: CrossSubnet
  natOutgoing: true
  disabled: false
  nodeSelector: kubernetes.io/hostname == 'kcontrol.demo.local'
---
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: pool1
spec:
  cidr: 10.1.1.0/24
  blockSize: 24
  ipipMode: CrossSubnet
  natOutgoing: true
  disabled: false
  nodeSelector: kubernetes.io/hostname == 'knode1.demo.local'
---
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: pool2
spec:
  cidr: 10.1.2.0/24
  blockSize: 24
  ipipMode: CrossSubnet
  natOutgoing: true
  disabled: false
  nodeSelector: kubernetes.io/hostname == 'knode2.demo.local'
```

2. Apply the configuration to create the IP pools:

```
kcontrol$ calicoctl create -f ippools.yaml
```

3. Verify that a pool was created for each node:

```
kcontrol$ calicoctl get ippools
NAME   CIDR        SELECTOR
pool0  10.1.0.0/24  kubernetes.io/hostname == 'kcontrol.demo.local'
pool1  10.1.1.0/24  kubernetes.io/hostname == 'knode1.demo.local'
pool2  10.1.2.0/24  kubernetes.io/hostname == 'knode2.demo.local'
```

If you want to be extra sure that networking is working correctly, follow the Test networking instructions provided by Calico.

For more information about configuring IP pools, see the Calico documentation.

Configuring iBGP Peering on the Kubernetes Nodes

IN THIS SOLUTION, WE CONFIGURE iBGP PEERING AS A FULL MESH

In this solution, we configure iBGP peering as a full mesh. (For very large clusters, Calico also supports the BGP Route Reflector option, which we won't discuss further here.)

1. Create a file called **bgppeers.yaml** with a section for each node, specifying the appropriate values in these fields:
   - `metadata.name`
   - `spec.peerIP`
   - `spec.asNumber`

```
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-kcontrol
spec:
  peerIP: 172.16.1.80
  asNumber: 64512
---
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-knode1
spec:
  peerIP: 172.16.1.81
  asNumber: 64512
---
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-knode2
spec:
  peerIP: 172.16.1.82
  asNumber: 64512
---
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: bgppeer-global-nginxedge
spec:
  peerIP: 172.16.1.88
  asNumber: 64512
```

2. Apply the configuration:

```
kcontrol$ calicoctl create -f bgppeer.yaml
```

3. Check the Calico BGP configuration, verifying that the value in the `MESHENABLED` field is `true`:

```
kcontrol$ calicoctl get bgpConfiguration
NAME       LOGSEVERITY    MESHENABLED    ASNUMBER
default    Info           true           64512
```

4. Check the BGP peering configuration:

```
kcontrol# calicoctl get bgpPeer
NAME                         PEERIP         NODE        ASN
bgppeer-global-kcontrol      172.16.1.80    (global)    64512
bgppeer-global-knode1        172.16.1.81    (global)    64512
bgppeer-global-knode2        172.16.1.82    (global)    64512
bgppeer-global-nginxedge     172.16.1.88    (global)    64512
```

5. Check that the BGP peering and mesh is working as expected:

```
kcontrol# calicoctl node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-----+---------+--------------+
| PEER ADDRESS |     PEER TYPE     |STATE|  SINCE  |     INFO     |
+--------------+-------------------+-----+---------+--------------+
| 172.16.1.81  |node-to-node mesh |up -- |22:56:17 |Established  |
| 172.16.1.82  |node-to-node mesh |up -- |22:56:19 |Established  |
| 172.16.1.88  |global             |start|22:54:55 |Connect       |
|              |                   |     |         |Socket:       |
|              |                   |     |         |Host is       |
|              |                   |     |         |unreachable   |
|              |                   |     |         |#nginxedge    |
+--------------+-------------------+-----+---------+--------------+

IPv6 BGP status
No IPv6 peers found.
```

Note: The `Host is unreachable` message for the NGINX edge server is appropriate, because it's not yet configured for BGP. You'll do that in Configuring the NGINX Plus Edge Server and check the status of the NGINX edge server again there.

For more information about Calico BGP peering, see the Calico documentation.

Deploying the NGINX Ingress Controller Based on NGINX Plus

THIS SOLUTION USES THE NGINX INGRESS CONTROLLER BASED ON NGINX PLUS

The solution uses the NGINX Ingress Controller based on NGINX Plus, running on any supported Kubernetes platform that also supports the Calico CNI. If you have an NGINX Plus subscription, you can use your existing SSL certificate and key to access the NGINX Plus repo and build your own NGINX Ingress Controller as instructed in the documentation.

If you don't have an NGINX Plus subscription, start a free 30-day trial.

Note: The solution works only with the NGINX Plus-based NGINX Ingress Controller from NGINX. No open source Ingress controller has the necessary features, including the one maintained by the Kubernetes community (kubernetes/ingress-nginx) and the one from NGINX based on NGINX Open Source (nginxinc/kubernetes-ingress). If you are not sure which Ingress Controller you are running, see our blog.

As previously mentioned, the solution is intended for on-premises Kubernetes deployments and doesn't work in public clouds, where the provider only supports its own choice of CNI and BGP implementations. Because an on-premises Kubernetes cluster doesn't have a LoadBalancer service for exposing the IP address and port of the NGINX Ingress Controller, we instead reach its ClusterIP address directly using BGP.

To implement this:

1. If you don't already have an NGINX Ingress Controller running, install and deploy one using the instructions in the documentation.
2. Create a file called **nginx-ingress-svc.yaml** with the following contents. Change the values in these fields as appropriate for your cluster:
   - `metadata.name`
   - `metadata.namespace`

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: nginx-ingress-svc
     namespace: nginx-ingress
   spec:
     type: ClusterIP
     clusterIP: None
     ports:
     - port:80
       targetPort: 80
       protocol: TCP
       name: http
     - port: 443
       targetPort: 443
       protocol: TCP
       name: https
     selector:
     app: nginx-ingress
   ```

3. Apply the configuration:

   ```
   kcontrol# kubectl apply -f nginx-ingress-svc.yaml
   ```

Configuring the NGINX Plus Edge Server

Set up and configure an NGINX Plus edge server, running as a reverse proxy for TCP traffic at Layer 4, to discover and load balance the NGINX Ingress Controllers running in the Kubernetes cluster. You also install a BGP routing package of your choice (the solution uses Quagga). You can set up two NGINX edge servers for high availability if you wish, but these instructions cover just one.

The solution leverages the dynamic DNS resolution feature in NGINX Plus. To learn more, see our blog and the reference documentation.

Installing NGINX Plus

Follow the installation instructions in the NGINX documentation.

Installing Quagga and Configuring BGP

THIS SOLUTION USES THE BGP ROUTING SOFTWARE FROM QUAGGA

The solution uses the BGP routing software from Quagga, which uses a configuration syntax very similar to Cisco IOS. There are several other BGP routing software packages that also run on most Linux distributions, and for most of them it's straightforward to adapt these instructions.

1. On the NGINX edge server, sign on to an account with **`root`** privilege or equivalent **`sudo`** access. The commands in this whitepaper do not use **`sudo`**.
2. Use the OS package manager to install Quagga on the NGINX edge server:
   - For CentOS and RHEL systems:

     ```
     nginxedge# yum install quagga
     ```

   - For Debian and Ubuntu systems:

     ```
     nginxedge# apt-get install quagga
     ```

3. Configure a BGP mesh that includes the NGINX edge server and all other nodes in the Kubernetes cluster. Run the following commands to open the Quagga **`vtysh`** shell and configure the mesh.

   For each node, add a two-line **`neighbor`** definition like those shown for the **`kcontrol`**, **`knode1`**, and **`knode2`** nodes in the following sample command. Also make the appropriate substitutions:

   - Your AS number for **`64512`**
   - The IP address of your NGINX edge server for **`172.16.1.88`**
   - Your network subnet for **`172.16.1.0/24`**
   - The IP address of each node for **`172.16.1.8`*`x`***

   Note: There might also be parameters for your existing iBGP network that need to be replicated on the NGINX edge server, so remember to add them too. Consult with your Network team.

```
nginxedge# vtysh
nginxedge$
configure terminal
router bgp 64512
bgp router-id 172.16.1.88    #IP addr of your NGINX edge server
network 172.16.1.0/24
neighbor calico peer-group
neighbor calico remote-as 64512
neighbor calico capability dynamic
neighbor 172.16.1.80 peer-group calico
neighbor 172.16.1.80 description kcontrol
neighbor 172.16.1.81 peer-group calico
neighbor 172.16.1.81 description knode1
neighbor 172.16.1.82 peer-group calico
neighbor 172.16.1.82 description knode2
exit
exit
write
```

4. Verify the mesh configuration is correct:

```
nginxedge$ show running-config
Building configuration...

Current configuration:
!
hostname nginxedge.demo.local
log file /var/log/quagga/quagga.log
hostname nginxedge
!
interface ens33
  description VMnet-172-16
  ip address 172.16.1.88/24
  ipv6 nd suppress-ra
!
interface lo
!
router bgp 64512
  bgp router-id 172.16.1.88
  network 172.16.1.0/24
  neighbor calico peer-group
  neighbor calico remote-as 64512
  neighbor calico capability dynamic
  neighbor 172.16.1.80 peer-group calico
  neighbor 172.16.1.80 description kcontrol
  neighbor 172.16.1.81 peer-group calico
  neighbor 172.16.1.81 description knode1
  neighbor 172.16.1.82 peer-group calico
  neighbor 172.16.1.82 description knode2
!
ip forwarding
!
line vty
!
end
```

5. Verify BGP is working correctly:

```
nginxedge$ show ip bgp summary

BGP router identifier 172.16.1.88, local AS number 64512
RIB entries 1, using 112 bytes of memory
Peers 3, using 9120 bytes of memory
Peer groups 1, using 32 bytes of memory

Neighbor    V  AS     MsgRcvd  MsgSent  TblVer  InQ  OutQ  Up/Down State/PfxRcd
172.16.1.80 4  64512  0        0        0       0    0     never    Active
172.16.1.81 4  64512  0        0        0       0    0     never    Active
172.16.1.82 4  64512  0        0        0       0    0     never    Active

Total number of neighbors 3
```

6. Display the detailed record for each neighbor. The sample output shows the record for the **kcontrol** node.

```
nginxedge$ show bgp neighbors
BGP neighbor is 172.16.1.80, remote AS 64512, local AS 64512,
internal link
Description: kcontrol
  Member of peer-group calico for session parameters
  BGP version 4, remote router ID 0.0.0.0
  BGP state = Active
  Last read 00:14:05, hold time is 180, keepalive interval is
  60 seconds

  Message statistics:
     Inq depth is 0
    Outq depth is 0
                       Sent  Rcvd
    Opens:             0     0
    Notifications:     0     0
    Updates:           0     0
    Keepalives:        0     0
    Route Refresh:     0     0
    Capability:        0     0
    Total:             0     0
  Minimum time between advertisement runs is 5 seconds

  For address family: IPv4 Unicast
    calico peer-group member
    Community attribute sent to this neighbor(both)
    0 accepted prefixes

    Connections established 0; dropped 0
    Last reset never
Next connect timer due in 19 seconds
Read thread: off Write thread: off
```

7. Exit the **vtysh** shell.

```
nginxedge$ exit
```

8. On the **kcontrol** node, verify that the NGINX edge server is now connected to the BGP mesh (the value in the **INFO** field is **Established**), meaning the full mesh is complete. If not, fix the BGP mesh configuration before continuing (these configuration examples might be helpful).

```
kcontrol# calicoctl node status
Calico process is running.

IPv4 BGP status
+--------------+-------------------+-----+---------+--------------+
| PEER ADDRESS |     PEER TYPE     |STATE|  SINCE  |     INFO     |
+--------------+-------------------+-----+---------+--------------+
| 172.16.1.81  | node-to-node mesh |up --|22:56:17 | Established  |
| 172.16.1.82  | node-to-node mesh |up --|22:56:19 | Established  |
| 172.16.1.88  | node-to-node mesh |up --|22:54:55 | Established  |
+--------------+-------------------+-----+---------+--------------+

IPv6 BGP status
No IPv6 peers found.
```

9. On the NGINX edge server, check the IP route table:

```
nginxedge# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, A - Babel,
       > - selected route, * - FIB route

K>* 0.0.0.0/0 via 172.16.1.2, ens33

B>* 10.1.0.0/24 [200/0] via 172.16.1.80, ens33, 00:07:03
B>* 10.1.1.0/24 [200/0] via 172.16.1.81, ens33, 00:00:58
B>* 10.1.2.0/24 [200/0] via 172.16.1.82, ens33, 00:01:58

C>* 127.0.0.0/8 is directly connected, lo
K>* 169.254.0.0/16 is directly connected, ens33
C>* 172.16.1.0/24 is directly connected, ens33
```

The three entries highlighted in pink are the ones you created with Quagga. Each route entry points to the Calico pod subnet (`10.1.x.0`) with the IP address of a Kubernetes node as the next hop, which means that the pods running on each node can be reached on their respective subnets. You now have network reachability from the NGINX edge server through the BGP mesh and the Kubernetes ClusterIP network to the pod subnets!

10. If you want a second NGINX edge server for high availability, repeat Steps 1 through 9 on it.

Testing the BGP Configuration

DID WE GET YOU TO THE CLUSTER AS PROMISED?

You're getting close now! Here comes the fun part, where you find out if all this effort was worth it. What do you think, can the NGINX edge server talk directly to the NGINX Ingress Controllers over BGP? Did we get you to the cluster as promised? Let's check it out.

Next, *for testing only*, you update the NGINX edge server's Linux DNS client (the host resolver, usually found in **/etc/resolv.conf**) to add endpoints for the Kubernetes DNS service (implemented as **kube-dns** pods), which is used to resolve cluster service names to cluster IP addresses.

Specifically, you add **`svc.cluster.local`** and **`cluster.local`** as search domains. (These entries are only for testing and are not needed when actually running the solution.)

Note that the Linux host's DNS resolver is not the resolver used by NGINX Plus, which is configured separately. The Linux DNS resolver enables testing with tools like **`ping`**, **`curl`**, and **`dig`**.

1. On the **`kcontrol`** node, look up the endpoint IP addresses for the **kube-dns** pods (in the following sample output, they are **`10.1.0.41`** and **`10.1.0.42`**):

```
kcontrol# kubectl describe svc kube-dns -n kube-system

Name:                kube-dns
Namespace:           kube-system
Labels:              K8s-app=kube-dns
                     kubernetes.io/cluster-service=true
                     kubernetes.io/name=KubeDNS
Annotations:         prometheus.io/port: 9153
                     prometheus.io/scrape: true
Selector:            K8s-app=kube-dns
Type:                ClusterIP
IP:                  10.96.0.10
Port:                dns  53/UDP
TargetPort:          53/UDP
Endpoints:           10.1.0.41:53,10.1.0.42:53
Port:                dns-tcp  53/TCP
TargetPort:          53/TCP
Endpoints:           10.1.0.41:53,10.1.0.42:53
Port:                metrics  9153/TCP
TargetPort:          9153/TCP
Endpoints:           10.1.0.41:9153,10.1.0.42:9153
Session Affinity:    None
Events:              <none>
```

2. On the NGINX edge server, update the Linux DNS resolver configuration, adding the two search domains and a **`nameserver`** entry for each **kube-dns** pod. Here the changes as they appear in **/etc/resolv.conf**, highlighted in pink.

```
nginxedge# cat /etc/resolv.conf
search localdomain demo.local cluster.local svc.cluster.local
nameserver 10.1.0.41      #kube-dns endpoint #1
nameserver 10.1.0.42      #kube-dns endpoint #2
```

3. On the **`kcontrol`** node, verify that at least one NGINX Ingress Controller is running and that the **`nginx-ingress-svc`** service you added in Deploying the NGINX Ingress Controller Based on NGINX Plus is properly configured:

```
kcontrol# kubectl get pods -n nginx-ingress
  NAME                           READY   STATUS    RESTARTS  AGE
  nginx-ingress-fd4b9f484-t5pb6  1/1     Running   1         12h

kcontrol# kubectl describe svc nginx-ingress-svc -n nginx-ingress
Name:           nginx-ingress-svc
Namespace:      nginx-ingress
Labels:         <none>
Annotations:    Selector: app=nginx-ingress
Type:           ClusterIP
IP:             None
Port:           http  80/TCP
TargetPort:     80/TCP
Endpoints:      10.1.0.89:80
Port:           https  443/TCP
TargetPort:     443/TCP
Endpoints:      10.1.0.89:443
```

4. On the NGINX edge server, make a DNS query to verify the IP address for **`nginx-ingress-svc`** can be resolved and pinged (here, the resolved address, **`10.1.0.89`**, matches the NGINX Ingress Controller's ClusterIP address.

```
nginxedge# ping nginx-ingress-svc.nginx-ingress.svc.cluster.local
PING nginx-ingress-svc.nginx-ingress.svc.cluster.local (10.1.0.89)
56(84) bytes of data.
64 bytes from 10-1-0-89.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.0.89): icmp_seq=1 ttl=63 time=0.293 ms
64 bytes from 10-1-0-89.nginx-ingress-svc.nginx-ingress.svc
.cluster.local (10.1.0.89): icmp_seq=2 ttl=63 time=0.952 ms
```

5. On the **kcontrol** node, scale up NGINX Ingress Controller deployment to two replicas:

```
kcontrol# kubectl scale deployment nginx-ingress -n nginx-
ingress --replicas=2
```

6. Verify that there are now two endpoint IP addresses for **nginx-ingress-svc**:

```
kcontrol# kubectl describe svc nginx-ingress-svc -n nginx-ingress
Name:               nginx-ingress-svc
Namespace:          nginx-ingress
Labels:             <none>
Annotations:        Selector:  app=nginx-ingress
Type:               ClusterIP
IP:                 None
Port:               http  80/TCP
TargetPort:         80/TCP
Endpoints:          10.1.0.89:80,10.1.1.122:80
Port:               https  443/TCP
TargetPort:         443/TCP
Endpoints:          10.1.0.89:443,10.1.1.122:443
Session Affinity:   None
Events:             <none>
```

7. On the NGINX edge server, ping **`nginx-ingress-svc`** and verify that DNS returns its two endpoints (**`10.1.1.122`** and **`10.1.0.89`**) in round-robin fashion.

```
nginxedge# ping nginx-ingress-svc.nginx-ingress.svc.cluster.local
PING nginx-ingress-svc.nginx-ingress.svc.cluster.local (10.1.1.122)
56(84) bytes of data.
64 bytes from 10-1-1-122.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.1.122): icmp_seq=1 ttl=63 time=0.404 ms
64 bytes from 10-1-1-122.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.1.122): icmp_seq=2 ttl=63 time=0.828 ms
64 bytes from 10-1-1-122.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.1.122): icmp_seq=3 ttl=63 time=0.865 ms
^C
--- nginx-ingress-svc.nginx-ingress.svc.cluster.local ping
statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.404/0.699/0.865/0.209 ms

nginxedge# ping nginx-ingress-svc.nginx-ingress.svc.cluster.local
PING nginx-ingress-svc.nginx-ingress.svc.cluster.local (10.1.0.89)
56(84) bytes of data.
64 bytes from 10-1-0-89.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.0.89): icmp_seq=1 ttl=63 time=0.220 ms
64 bytes from 10-1-0-89.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.0.89): icmp_seq=2 ttl=63 time=0.909 ms
64 bytes from 10-1-0-89.nginx-ingress-svc.nginx-ingress.svc.
cluster.local (10.1.0.89): icmp_seq=3 ttl=63 time=0.435 ms
^C
--- nginx-ingress-svc.nginx-ingress.svc.cluster.local ping
statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.404/0.699/0.865/0.209 ms
```

Alternatively, you can run **dig** and verify there are two DNS **A** records, for **10.1.0.89** and **10.1.1.122**.

```
nginxedge# dig nginx-ingress-svc.nginx-ingress.svc.cluster.local

; <<>> DiG 9.11.4-P2-RedHat-9.11.4-26.P2.el7_9.4 <<>> nginx-ingress-
svc.nginx-ingress.svc.cluster.local
...
;; QUESTION SECTION:
;nginx-ingress-svc.nginx-ingress.svc.cluster.local.  IN A

;; ANSWER SECTION:
nginx-ingress-svc.nginx-ingress.svc.cluster.local. 30 IN A 10.1.0.89
nginx-ingress-svc.nginx-ingress.svc.cluster.local. 30 IN A 10.1.1.122

;; Query time: 0 msec
;; SERVER: 10.1.0.41#53(10.1.0.41)
;; WHEN: Day Mon DD HH:MM:SS TZ YYYY
;; MSG SIZE  rcvd: 208
```

As you see, scaling the number of NGINX Ingress Controllers up and down changes the number of endpoint IP addresses to match. Running multiple NGINX Ingress Controllers provides production-grade high availability, and we recommend running at least three for production workloads. (You might want to run Steps 5 through 7 again with three replicas. Do you get three IP addresses in Steps 6 and 7?)

Configuring Layer 4 Load Balancing on the NGINX Edge Server

YOU CAN CONFIGURE THE NGINX EDGE SERVER AS A LAYER 4 LOAD BALANCER FOR TCP TRAFFIC

Now that you have Layer 3 connectivity between the NGINX edge server and NGINX Ingress Controllers courtesy of BGP, you can configure the NGINX edge server as a Layer 4 load balancer for TCP traffic, to replace the LoadBalancer service that's available only in cloud deployments.

Create an NGINX configuration file like the following on the NGINX edge server, perhaps called **nginxedge.conf**. (We recommend placing configuration files for the NGINX **stream** context in the **/etc/nginx/stream.d** folder and using an **include** directive in the main NGINX configuration file to reference them; for details, see Configuration Best Practices on our blog.)

The following sample configuration for Layer 4 load balancing on the NGINX edge server handles traffic on both port 8080 and port 8443. These settings in the **`upstream`** configuration blocks are key to the solution:

- The **`resolver`** directive tells NGINX which DNS servers to query, and the **`valid=10s`** parameter specifies the frequency (here, every 10 seconds). The **`status_zone`** parameter collects metrics about the success or failure of these DNS queries; if you've configured the NGINX Plus live activity monitoring dashboard, the metrics are displayed on the **Resolvers** tab as shown below.
- The **`zone`** directive collects statistics for the TCP traffic handled by the virtual server in this **`upstream`** block (also displayed on the **TCP/UDP Upstreams** tab on the NGINX Plus dashboard as shown below, if configured).
- The **`resolve`** parameter on the **`server`** directive tells NGINX to query DNS for the list of IP addresses for this server's FQDN (its Kubernetes service name). For NGINX Ingress Controllers, the FQDN is:

  ```
  nginx-ingress-svc.nginx-ingress.svc.cluster.local
  ```

  This name complies with the standard **kube-dns** naming format for all Kubernetes services:

  ```
  service_name.namespace.svc.cluster.local
  ```

THE NGINX EDGE SERVER USES THE NGINX PLUS DNS RESOLUTION FEATURE

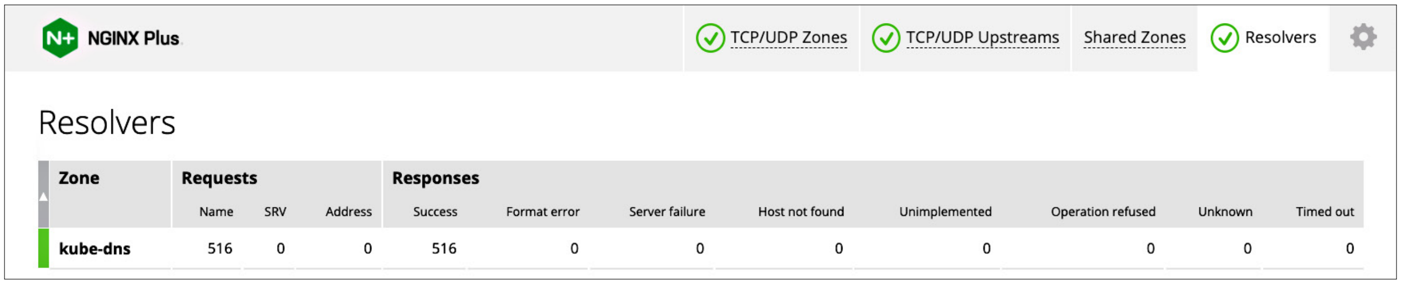
With this configuration in place, the NGINX edge server uses the NGINX Plus DNS resolution feature. It queries **kube-dns** for the DNS `A` records of the NGINX Ingress Controllers running in the cluster. The **kube-dns** service returns a list of the NGINX Ingress Controllers' ClusterIP addresses as allocated by Calico. The NGINX edge server can reach the **kube-dns** and NGINX Ingress Controller pods directly at their IP addresses over the BGP mesh network.

Notes:

- To use port 80 instead of 8080, you need to rename the existing **default.conf** file so that NGINX Plus doesn't read the configuration for port 80 there
- Substitute the IP addresses of your **kube-dns** servers for **`10.1.0.41`** and **`10.1.0.42`**

```
# NGINX edge server Layer 4 configuration file
# Use kube-dns ClusterIP addresses for the NGINX Plus resolver
# DNS query interval is 10 seconds

stream {
    log_format stream '$time_local $remote_addr - $server_addr -
$upstream_addr';
    access_log /var/log/nginx/stream.log stream;

    # Sample configuration for TCP load balancing
    upstream nginx-ingress-80 {
      # use the kube-dns endpoint IP addresses for the
      # NGINX Plus resolver
      resolver 10.1.0.41 10.1.0.42 valid=10s status_zone=kube-dns;
      zone nginx_kic_80 256k;

      server nginx-ingress-svc.nginx-ingress.svc.cluster.local:
80 resolve;
    }

    upstream nginx-ingress-443 {
      # use the kube-dns endpoint IP addresses for the
      # NGINX Plus resolver
      resolver 10.1.0.41 10.1.0.42 valid=10s status_zone=kube-dns;
      zone nginx_kic_443 256k;
      server nginx-ingress-svc.nginx-ingress.svc.cluster.local:
443 resolve;
    }

    server {
      listen 8080;
      status_zone tcp_server_8080;
      proxy_pass nginx-ingress-80;
    }

    server {
      listen 8443;
      status_zone tcp_server_8443;
      proxy_pass nginx-ingress-443;
    }
}
```

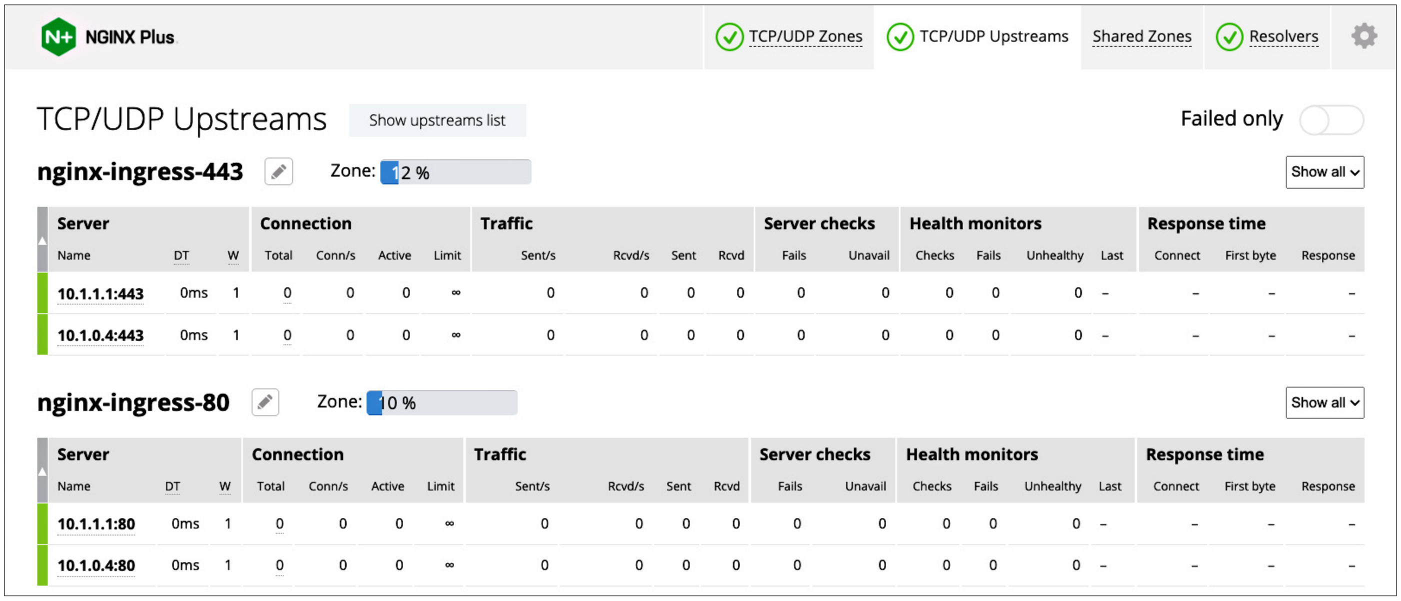
The **Resolvers** tab on the live activity monitoring dashboard shows that all DNS queries have been successful:



Resolvers

Zone	Requests			Responses							
	Name	SRV	Address	Success	Format error	Server failure	Host not found	Unimplemented	Operation refused	Unknown	Timed out
kube-dns	516	0	0	516	0	0	0	0	0	0	0

The **TCP/UDP Upstreams** tab shows upstream servers for HTTP and HTTPS like those configured on page 29:



TCP/UDP Upstreams

nginx-ingress-443 Zone: 12 %

Server			Connection				Traffic				Server checks		Health monitors				Response time		
Name	DT	W	Total	Conn/s	Active	Limit	Sent/s	Rcvd/s	Sent	Rcvd	Fails	Unavail	Checks	Fails	Unhealthy	Last	Connect	First byte	Response
10.1.1.1:443	0ms	1	0	0	0	∞	0	0	0	0	0	0	0	0	0	–	–	–	–
10.1.0.4:443	0ms	1	0	0	0	∞	0	0	0	0	0	0	0	0	0	–	–	–	–

nginx-ingress-80 Zone: 10 %

Server			Connection				Traffic				Server checks		Health monitors				Response time		
Name	DT	W	Total	Conn/s	Active	Limit	Sent/s	Rcvd/s	Sent	Rcvd	Fails	Unavail	Checks	Fails	Unhealthy	Last	Connect	First byte	Response
10.1.1.1:80	0ms	1	0	0	0	∞	0	0	0	0	0	0	0	0	0	–	–	–	–
10.1.0.4:80	0ms	1	0	0	0	∞	0	0	0	0	0	0	0	0	0	–	–	–	–

SUMMARY AND NEXT STEPS

You've deployed a solution for dynamically resolving the IP addresses of NGINX Ingress Controllers in an on-premises Kubernetes cluster, using the Kubernetes DNS service and BGP.

You deployed, configured, and tested the four components of this solution:

- An iBGP network in the data center
- Kubernetes cluster with Calico CNI and BGP
- NGINX Ingress Controller based on NGINX Plus
- NGINX Plus edge server, configured for BGP with Quagga and for Layer 4 load balancing

As a next step, deploy some application pods and configure NGINX Ingress Controller to route traffic to those pods and see what happens. If you don't have a test app, we suggest the "complete example" at our GitHub repo (which also has many other examples for specific use cases).

We hope this whitepaper helps you tackle the challenge of providing user access to your Kubernetes apps running on premises. We trust the Networking and App Dev teams at your organization can collaborate on a solution like this so you can sprint through milestones along your modern application journey.

NGINX®
Part of F5