Dmitri Lerko

Personal blog where I cover my experiences and best practices with GCP, GKE, GitOps, Certifications and DevOps.

When GKE ran out of IP addresses

Feb 3, 2020
5 min read
gke kubernetes gcp

Setup

In our GKE environments, we use Shared VPC functionality to have a separation of concerns between networking and compute projects. This also allows for clearer networking architecture when common networking services need to be shared amongst a number of compute projects, for example, VPN tunnels, Network Peering and Cloud Interconnects. Additionally, GKE offers VPC-native clusters which come with several of scalability and security benefits. This feature allocates pod IPs from a subnet's secondary range.

One of our clusters runs in a VPC-native mode with a dedicated subnet. The subnet has a /16 subnet mask for IP secondary range, yielding a maximum of 65'536 pods. That's a lot of pods!

Incident

We've received a slack message from a colleague saying that deployment of his application takes a long time. On a quick check, we could confirm that an application that normally deploys in minutes has been hanging around for about an hour with only the subset of new pods live. Usually, when deployments are stuck, you expect a missing ConfigMap, Secret or wrong nodeSelector. This time it was something else as about half of the newly deployed pods were running and serving traffic, while the second half was in a pending state.

We saw the following event on the describe of one of the pending pods:

```
Warning  FailedScheduling   2m18s (x4 over 3m48s)   default-scheduler   0/256 nodes are available:
1 node(s) were unschedulable, 173 Insufficient memory, 214 node(s) didn't match node selector, 87 Insufficient cpu.
```

Typically, GKE Cluster Autoscaler takes care of node pool scale-ups when there are not enough nodes to meet pod scheduling requirements. However, this time it wasn't getting anywhere. Also, "0/256 nodes are available" is an awfully "power of two" round number that raised our suspicion.

Next, we've looked into Cluster Autoscaler events:

```
kubectl describe -n kube-system configmap cluster-autoscaler-status
# Below event output is stripped from any identifiable information
Warning  ScaleUpFailed   11m (x14 over 36m)     cluster-autoscaler  (combined from similar events):
Failed adding 3 nodes to group
https://content.googleapis.com/compute/v1/projects/PROJECT/zones/OUR_REGION/instanceGroups/INSTANCE_GROUP due to
Other.OTHER; source errors: Instance 'INSTANCE' creation failed: IP space of
'projects/PROJECT/regions/OUR_REGION/subnetworks/SUBNETWORK' is exhausted. , Instance 'INSTANCE' creation
failed: IP space of 'projects/PROJECT/regions/OUR_REGION/subnetworks/SUBNETWORK' is exhausted. , Instance 'INSTANCE'
creation failed: IP space of 'projects/PROJECT/regions/OUR_REGION/subnetworks/SUBNETWORK' is exhausted.
```

How could we run out of 65'536 IPs? While we can be running thousands of pods, we are still far away from the maximum capacity. So, we went back to the GKE documentation. We were aware that by default, GKE assigns 110 pods maximum per node, however, we didn't know that it does pre-allocations. So, even if you are only planning on running a couple of pods per node, GKE will pre-allocate 110 IPs.

With this knowledge, we can expect to run 595 nodes (65'536 / 110), but this is not what we saw. The very same GCP documentation also mentions that Kubernetes assigns a subnet mask that can store at least x2 of maximum pods per node.

> "By having approximately twice as many available IP addresses as the number of pods that can be created on a node, Kubernetes is able to mitigate IP address reuse as Pods are added to and removed from a node."

This meant that each of our 110 maximum pod nodes was consuming 256 IP addresses, which aligns exactly with the observed behaviour: 65'536 / 256 = 256.

Solutions

Most effective

Reduce the maximum number of pods per node on the node pools where you don't expect large numbers of pods running on the node. This could be nodes for stateful applications, caches or increased security workloads. By selectively reducing the maximum number of pods we were able to reduce our overall IP consumption by about 30%.

Relatively effective

GCE supports subnet expansion, which is a pretty unique feature, which allows you to reduce the subnet mask of a subnet, as long as this will not result in the IP range collision with other subnets. For example, you can move from /24 to /23 subnet mask, effectively doubling available IP range in an instant. However, documentation is quiet about secondary ranges, therefore we choose not to go ahead with subnet expansion.

Has the potential to work, needs additional validation

Another option is to increase the size of the existing nodes, therefore run more pods per node. This is useful for scenarios where you run large, shared node-pools with many nodes. This won't work when there is not enough workload to fill up larger nodes. Interestingly, you also can improve utilisation with larger nodes. It is improved by having fewer resources spent on system DeamonSets per pod, but at the same time, larger nodes are more likely to have higher bin-packing as larger CPU and memory allocations allow for a greater combination of pods to fill up available capacity. Larger nodes, however, will take more pods down when they fail. We've increased some of our node pool instance sizes and were able to benefit from it.

Useful debug commands

```
# Check autoscaler status
kubectl describe -n kube-system configmap cluster-autoscaler-status

# Check maximum pods configuration for the node
kubectl get node NODE-NAME -ojson | jq .status.allocatable.pods
```

TL;DR

By default, GKE allocates 256 IPs per node, meaning that even large subnets like /16 can run out pretty quickly when you running 256 nodes.

Find posts by tags

certifications dataflow flux gcp gcr gitops gke jekyll kubernetes label_replace prometheus recording-rules redis sensu