Kubernetes | Networking

Racy conntrack and DNS lookup timeouts

Recently there were many Kubernetes user bug reports about DNS lookups from Pods sometimes taking 5 or even more seconds: weave#3287, kubernetes#56903.

In this post, I will explain the root causes for such delays, discuss some mitigations and present the kernel fixes.

Background

In Kubernetes, the most common way for a Pod to access a DNS server (`kube-dns`) is via the Service abstraction. Therefore, before trying to explain the problem, it is important to understand how Service works, and consequently, how the Destination Network Address Translation (DNAT) is implemented in the Linux kernel.

NOTE: all examples in this post are based on Kubernetes v1.11.0 and Linux kernel v4.17.

How Service works

In `iptables` mode, which is a default, `kube-proxy` for each Service creates a few iptables rules in the `nat` table of the host network namespace.

Let's consider the `kube-dns` Service with two DNS server instances in a cluster. The relevant rules are the following:

```
(1) -A PREROUTING -m comment --comment "kubernetes service portals" -j KUBE-
<...>
(2) -A KUBE-SERVICES -d 10.96.0.10/32 -p udp -m comment --comment "kube-syst
<...>
(3) -A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment --comment "kube-system/kube-dns:
(4) -A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment --comment "kube-system/kube-dns:
<...>
(5) -A KUBE-SEP-LLLB6FGXBLX6PZF7 -p udp -m comment --comment "kube-system/ku
<...>
(6) -A KUBE-SEP-LRVEW52VMYCOUSMZ -p udp -m comment --comment "kube-system/ku
```

In our example, each Pod has the `nameserver 10.96.0.10` entry populated in its /etc/resolv.conf. Therefore, a DNS lookup request from a Pod is going to be sent to 10.96.0.10 which is a ClusterIP (a virtual IP) of the `kube-dns` Service.

The request enters the `KUBE-SERVICE` chain due to (1), then matches the rule (2), and finally, depending on a random value of (3) either jumps to the (5) or (6) rule (a poor-man's load balancing) which modifies the destination IPv4 address of the request UDP packet to the "real" IPv4 address of the DNS server. This modification is done by DNAT.

10.32.0.6 and 10.32.0.7 are IPv4 addresses of the Kubernetes DNS server containers in Weave Net network.

DNAT in Linux Kernel

As seen above, the foundation of Service (in the `iptables` mode) is DNAT which is performed by the kernel.

The main responsibilities of DNAT are to change a destination of an outgoing packet, a source of a reply packet and at the same time, to ensure that the same modifications are applied to all subsequent packets.

The latter heavily relies on the connection tracking mechanism also known as `conntrack` which is implemented as a kernel module. As the name suggests, `conntrack` keeps track of ongoing network connections in the system.

In a simplified way, each connection in `conntrack` is represented with two tuples - one for the original request (`IP CT DIR ORIGINAL`) and one for the

reply (`IP_CT_DIR_REPLY`). In the case of UDP, each of the tuples consists of the source IP address, the source port, as well as the destination IP address and the destination port. The reply tuple contains the real address of a target stored in the `src` field.

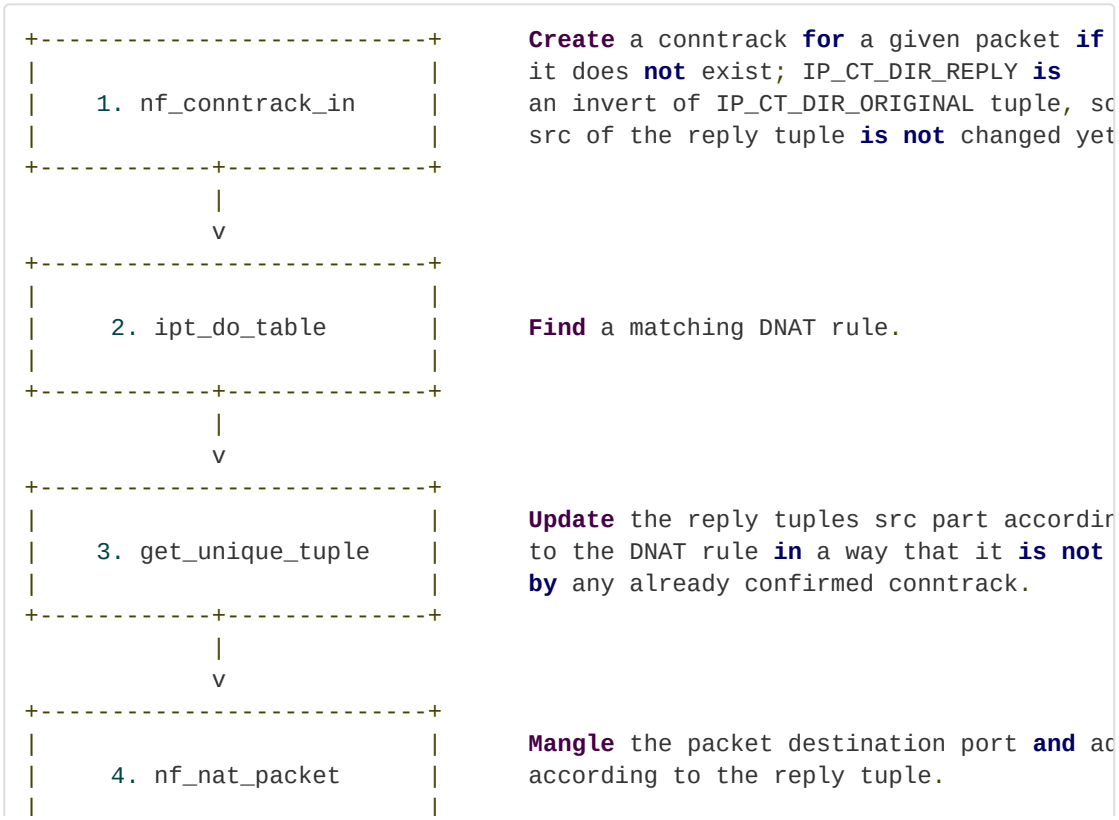
For example, if a Pod with the IP address 10.40.0.17 sends a request to the ClusterIP of `kube-dns` which gets translated to 10.32.0.6, the following tuples will be created:

- Original: `src=10.40.0.17 dst=10.96.0.10 sport=53378 dport=53`
- Reply: `src=10.32.0.6 dst=10.40.0.17 sport=53 dport=53378`

By having these entries the kernel can modify the destination and source addresses of any related packets accordingly without the need to traverse the DNAT rules again. Also, it will know how to modify a reply and to whom it should be sent.

When a `conntrack` entry is created, it is first unconfirmed. Later on, the kernel will try to confirm the entry if there is no confirmed `conntrack` entry with either the same original tuple or a reply tuple.

A simplified flow of the `conntrack` creation and DNAT is shown below:



```
+------------+-------------+
             |
             v
+----------------------------+
|                            |     Confirm the conntrack if there is no conf
|  5. __nf_conntrack_confirm |     conntrack with either the same original c
|                            |     a reply tuple; increment insert_failed co
+----------------------------+     and drop the packet if it exists.
```

Problem

A problem occurs when two UDP packets are sent via the same socket at the same time from different threads.

UDP is a connection-less protocol, so no packet is sent as a result of the connect(2) syscall (opposite to TCP) and thus, no `conntrack` entry has been created after the call.

The entry is created only when a packet is sent. This leads to the following possible races:

1. Neither of the packets finds a confirmed `conntrack` in the `1. nf_conntrack_in` step. For both packets two `conntrack` entries with the same tuples are created.
2. Same as in the above case, but a `conntrack` entry of one of the packets is confirmed before the other has called `3. get_unique_tuple`. The other packet gets a different reply tuple usually with the source port changed.
3. Same as in the 1st case, but two different rules with different endpoints are selected in the step `2. ipt_do_table`.

The outcome of the races is the same - one of the packets gets dropped in the step `5. __nf_conntrack_confirm`.

This is exactly what happens in the DNS case. The GNU C Library and musl libc both perform A and AAAA DNS lookups in parallel. One of the UDP packets might get dropped by the kernel due to the races, so the client will try to re-send it after a timeout which is usually 5 seconds.

It is worth mentioning that the problem is not only specific for Kubernetes - any Linux multi-threaded process sending UDP packets in parallel is prone to this race condition.

prone to this race condition.

Also, the 2nd race can happen even if you don't have any DNAT rules - it's enough to load the `nf_nat` kernel module to enable calls to `get_unique_tuple`.

The `insert_failed` counter which can be obtained with `conntrack -S` is a good indicator whether you are experiencing the problem.

Mitigations

Suggestions

There were many workarounds suggested: disable parallel lookups, disable IPv6 to avoid AAAA lookups, use TCP for lookups, set a real IP address of a DNS server in Pod's resolver configuration file instead, etc. See linked issues in the beginning of the post for more details. Unfortunately, many of them do not work due to limitations in musl libc used by a commonly used container base image Alpine Linux.

The one which seems to reliably work for Weave Net users is to delay DNS packets with tc. See Quentin Machu's write-up about it.

Also, you might be wondering whether kube-proxy in the `ipvs` mode can bypass the problem. The answer is no, as `conntrack` is enabled in this mode as well. Also, when using the `rr` scheduler, the 3rd race can be easily reproduced in a cluster with low DNS traffic.

Kernel Fix

Regardless of the workarounds, I decided to fix the root causes in the kernel.

The outcome is the following kernel patches:

1. "netfilter: nf_conntrack: resolve clash for matching conntracks" fixes the 1st race (accepted).
2. "netfilter: nf_nat: return the same reply tuple for matching CTs" fixes the 2nd race (waiting for a review).

These two patches fix the problem for a cluster that runs only one instance of a DNS server, while reducing the timeout hit rate for the others.

To completely eliminate the problem in all cases, the 3rd race needs to be addressed. One possible fix is to merge clashing `conntrack` entries with different destinations from the same socket in the step `5. __nf_conntrack_confirm`. However, this would invalidate a result of a previous iptables rules traversal for a packet of which the destination is changed in that step.

Another possible solution is to run a DNS server instance on each node and make a Pod to query a DNS server running on a local node as suggested by my colleague here.

Conclusions

First, I showed the underlying details of the "DNS lookup takes 5 seconds" problem and revealed the culprit - the Linux `conntrack` kernel module which is inherently racy. See this article for other possible races in the module.

Next, I presented the kernel fixes which eliminate two out of three relevant races in the module.

Finally, I emphasized that, at the time of writing, the root cause is not completely fixed, and in some cases requires workarounds from users.

Martynas is a recent graduate of ETH Zurich, who spends the majority of his time programming systems. When he is not hacking, most likely you can find him climbing the rock.

< PREVIOUS

NEXT >

You may also like:

JULY 20, 2021

Kubernetes | Gitops

Why developers need a self-service platform – and how to provide one

JULY 15, 2021

Gitops | Continuous delivery | Operations | Kubernetes

Application Portability for the Cloud Era

JUNE 30, 2021

Gitops | Kubernetes

Declarative delivery at scale: Weave GitOps