A reason for unexplained connection timeouts on Kubernetes/Docker

Maxime Lagresle Feb 22, 2018

Abstract

The Linux Kernel has a known race condition when doing source network address translation (SNAT) that can lead to SYN packets being dropped. SNAT is performed by default on outgoing connections with Docker and Flannel using iptables masquerading rules. The race can happen when multiple containers try to establish new connections to the same external address concurrently. In some cases, two connections can be allocated the same port for the translation which ultimately results in one or more packets being dropped and at least one second connection delay.

This race condition is mentioned in the source code but there is not much documentation around it. While the Kernel already supports a flag that mitigates this issue, it was not supported on iptables masquerading rules until recently.

In this post we will try to explain how we investigated that issue, what this race condition consists of with some explanations about container networking, and how we mitigated it.

Edit 15/06/2018: the same race condition exists on DNAT. On Kubernetes, this means you can lose packets when reaching ClusterIPs. For those who don't know about DNAT, it's probably best to read this article first but basically, when you do a request from a Pod to a ClusterIP, by default kube-proxy (through iptables) changes the ClusterIP with one of the PodIP of the service you are trying to reach. One of the most used cluster Service is the DNS and this race condition would generate intermitent delays when doing name resolution, see issue 56903 or this interesting article from Quentin Machu.

Introduction

I'm part of the Backend Architecture Team at XING. The past year, we have worked together with Site Operations to build a Platform as a Service. In September 2017, after a few months of evaluation we started migrating from our Capistrano/Marathon/Bash based deployments to Kubernetes.

Our setup relies on Kubernetes 1.8 running on Ubuntu Xenial virtual machines with Docker 17.06, and Flannel 1.9.0 in host-gateway mode.

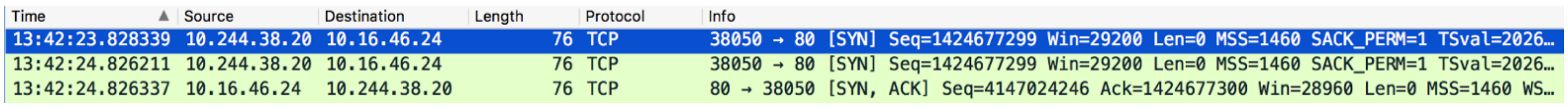
While migrating we noticed an increase of connection timeouts in applications once they were running on Kubernetes. This became more visible after we moved our first Scala-based application. Almost every second there would be one request being really slow to respond instead of the usual few hundred of milliseconds. The application was exposing REST endpoints and querying other services on the platform, collecting, processing and returning the data to the client. Nothing unusual there.

The response time of those slow requests was strange. Almost all of them were delayed for exactly 1 or 3 seconds! We decided it was time to investigate the issue.

Narrowing down the problem

We had a ticket in our backlog to monitor the KubeDNS performances. As depending on the HTTP client, the name resolution time could be part of the connection time, we decided to tackle that ticket first and make sure this component was working well. We wrote a small DaemonSet that would query KubeDNS and our datacenter name servers directly, and send the response time to InfluxDB. Soon the graphs showed fast response times which immediately ruled out the name resolution as possible culprit.

The next step was first to understand what those timeouts really meant. The team responsible for this Scala application had modified it to let the slow requests continue in the background and log the duration after having thrown a timeout error to the client. We took some network traces on a Kubernetes node where the application was running and tried to match the slow requests with the content of the network dump.



```
Time             Source        Destination   Length  Protocol  Info
13:42:23.828339  10.244.38.20  10.16.46.24       76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826211  10.244.38.20  10.16.46.24       76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826337  10.16.46.24   10.244.38.20      76  TCP       80 → 38050 [SYN, ACK] Seq=4147024246 Ack=1424677300 Win=28960 Len=0 MSS=1460 WS…
```

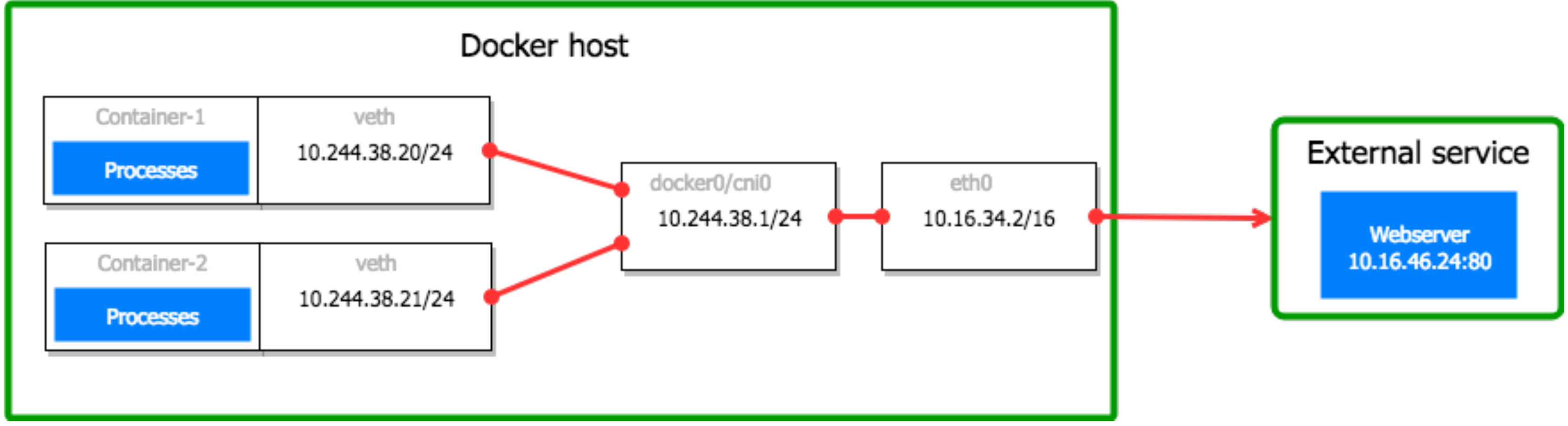
container's perspective, `10.244.38.20` trying to connect to `10.16.46.24` on port `80`

The results quickly showed that the timeouts were caused by a retransmission of the first network packet that is sent to initiate a connection (packet with a SYN flag). This was explaining very well the duration of the slow requests since the retransmission delays for this kind of packets are 1 second for the second try, 3 seconds for the third, then 6, 12, 24, etc.

This was an interesting finding because losing only SYN packets rules out some random network failures and speaks more for a network device or SYN flood protection algorithm actively dropping new connections.

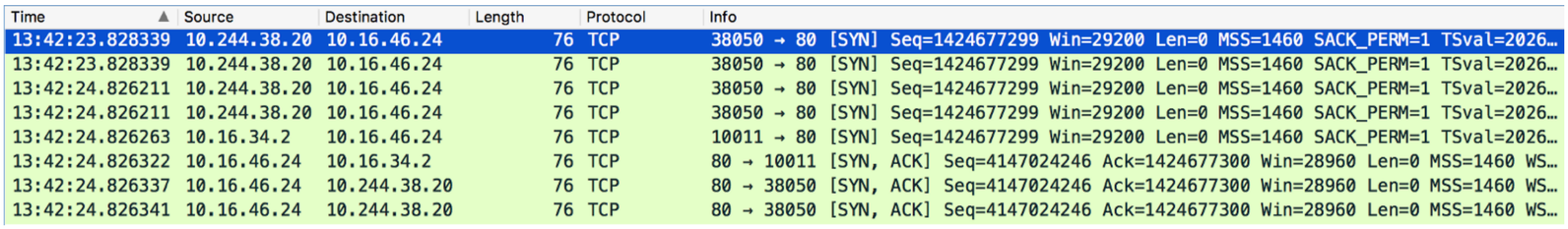
On default Docker installations, each container has an IP on a virtual network interface (`veth`) connected to a Linux bridge on the Docker host (e.g `cni0`, `docker0`) where the main interface (e.g `eth0`) is also connected to ([6](#)). Containers talk to each other through the bridge. If a container tries to reach an address external to the Docker host, the packet goes on the bridge and is routed outside the server through `eth0`.

The following example has been adapted from a default Docker setup to match the network configuration seen in the network captures:



in reality `veth` interfaces come in pairs but this doesn't matter in our context

We had randomly chosen to look for packets on the bridge so we continued by having a look at the virtual machine's main interface `eth0`. We would then concentrate on the network infrastructure or the virtual machine depending on the result.



```
Time            ▲ Source        Destination     Length  Protocol  Info
13:42:23.828339 10.244.38.20 10.16.46.24        76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:23.828339 10.244.38.20 10.16.46.24        76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826211 10.244.38.20 10.16.46.24        76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826211 10.244.38.20 10.16.46.24        76  TCP       38050 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826263 10.16.34.2   10.16.46.24        76  TCP       10011 → 80 [SYN] Seq=1424677299 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2026…
13:42:24.826322 10.16.46.24  10.16.34.2         76  TCP       80 → 10011 [SYN, ACK] Seq=4147024246 Ack=1424677300 Win=28960 Len=0 MSS=1460 WS…
13:42:24.826337 10.16.46.24  10.244.38.20       76  TCP       80 → 38050 [SYN, ACK] Seq=4147024246 Ack=1424677300 Win=28960 Len=0 MSS=1460 WS…
13:42:24.826341 10.16.46.24  10.244.38.20       76  TCP       80 → 38050 [SYN, ACK] Seq=4147024246 Ack=1424677300 Win=28960 Len=0 MSS=1460 WS…
```

capture from `veth0`, `cni0` and `eth0`, `10.244.38.20` trying to connect to `10.16.46.24` on port `80`

The network capture showed the first SYN packet leaving the container interface (`veth`) at `13:42:23.828339` and going through the bridge (`cni0`) (duplicate line at `13:42:23.828339`). After one second at `13:42:24.826211`, the container getting no response from the remote endpoint `10.16.46.24` was retransmitting the packet. Again, the packet would be seen on the container's interface, then on the bridge. On the next line, we see the packet leaving `eth0` at `13:42:24.826263` after having been translated from `10.244.38.20:38050` to `10.16.34.2:10011`. The next lines show how the remote service responded.

What this translation means will be explained in more details later in this post. Because we can't see the translated packet leaving `eth0` after the first attempt at `13:42:23`, at this point it is considered to have been lost somewhere between `cni0` and `eth0`. It was really surprising to see that those packets were just disappearing as the virtual machines had a low load and request rate. We repeated the tests a dozen of time but the result remained the same.

At that point it was clear that our problem was on our virtual machines and had probably nothing to do with the rest of the infrastructure.

Now that we had isolated the issue, it was time to reproduce it on a more flexible setup. We wrote a really simple Go program that would make requests against an endpoint with a few configurable settings:

- the delay between two requests
- the number of concurrent requests
- the timeout
- the endpoint to call

The remote endpoint to connect to was a virtual machine with Nginx. Our test program would make requests against this endpoint and log any response time higher than a second. After a few adjustment runs we were able to reproduce the issue on a non-production cluster.

Netfilter and source network address translation

For the comprehension of the rest of the post, it is better to have some knowledge about source network address translation. The following section is a simplified explanation on this topic but if you already know about SNAT and conntrack, feel free to skip it.

Not only is this explanation simplified, but some details are sometimes completely ignored or worse, the reality slightly altered. You've been warned!

Container IP and datacenter network infrastructure

On a default Docker installation, containers have their own IPs and can talk to each other using those IPs if they are on the same Docker host. However, from outside the host you cannot reach a container using its IP. To communicate with a container from an external machine, you often expose the container port on the host interface and then use the host IP. This is because the IPs of the containers are not routable (but the host IP is). The network infrastructure is not aware of the IPs inside each Docker host and therefore no communication is possible between containers located on different hosts (Swarm or other network backends are a different story).

With Flannel in host-gateway mode and probably a few other Kubernetes network plugins, pods can talk to pods on other hosts at the condition that they run inside the same Kubernetes cluster. You can reach a pod from another pod no matter where it runs, but you cannot reach it from a virtual machine outside the Kubernetes cluster. You can achieve this with Calico for example, but not with Flannel at least in host-gw mode.

Source network address translation

If you cannot connect directly to containers from external hosts, containers shouldn't be able to communicate with external services either. If a container sends a packet to an external service, since the container IPs are not routable, the remote service wouldn't know where to send the reply. In reality they can, but only because each host performs **source network address translation** on connections from containers to the outside world.

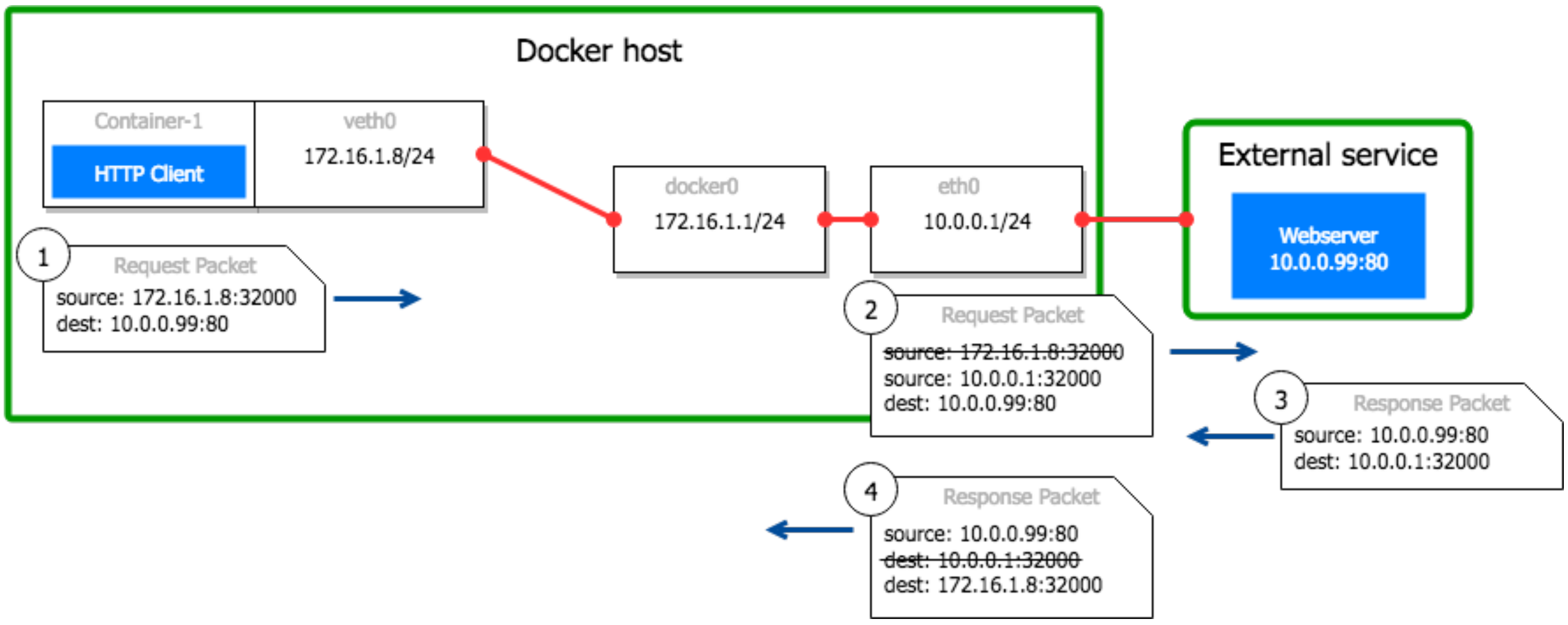
Our Docker hosts can talk to other machines in the datacenter. They have routable IPs. When a container tries to reach an external service, the host on which the container runs replaces the container IP in the network packet with its own IP. For the external service, it looks like the host established the connection itself. When the response comes back to the host, it reverts the translation. For the container, the operation was completely transparent and it has no idea such a transformation happened.

Example: A Docker host `10.0.0.1` runs a container named container-1 which IP is `172.16.1.8`. The process inside the container initiates a connection to reach `10.0.0.99:80`. It binds on its local container port `32000`.

1. The packet leaves the container and reaches the Docker host with

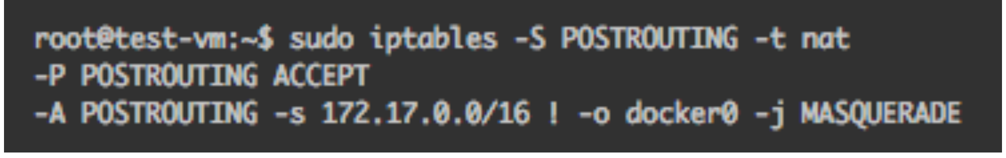
the source set to `172.16.1.8:32000`

2. The Docker host replaces the *source* header from `172.16.1.8:32000` to `10.0.0.1:32000` and forwards the packet to `10.0.0.99:80`. Linux tracks this translation in a table to be able to revert it in the packet reply.
3. The remote service `10.0.0.99:80` processes the request and answers to the host
4. The response packet reaches the host on port `32000` . Linux sees the packet is a reply to a connection that was translated. It modifies the *destination* from `10.0.0.1:32000` to `172.16.1.8:32000` and forwards the packet to the container



Iptables and netfilter

Linux comes with a framework named netfilter that can perform various network operations at different places in the kernel networking stack. It includes packet filtering for example, but more interestingly for us, network address translation and port address translation. Iptables is a tool that allows us to configure netfilter from the command line. The default installations of Docker add a few iptables rules to do SNAT on outgoing connections. In our Kubernetes cluster, Flannel does the same (in reality, they both configure iptables to do masquerading, which is a kind of SNAT).

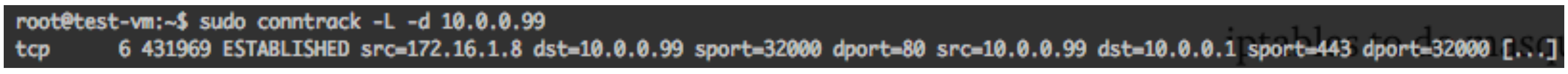


```
root@test-vm:~$ sudo iptables -S POSTROUTING -t nat
-P POSTROUTING ACCEPT
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
```

packets from containers (172.17.0.0/16 in this case) to anything but the bridge (docker0) will be masqueraded

When a connection is issued from a container to an external service, it is processed by netfilter because of the iptables rules added by Docker/Flannel. The NAT module of netfilter performs the SNAT operation by replacing the source IP in the outgoing packet with the host IP and adding an entry in a table to keep track of the translation. The entry ensures that the next packets for the same connection will be modified in the same way to be consistent. It also makes sure that when the external service answers to the host, it will know how to modify the packet accordingly.

Those entries are stored in the conntrack table (conntrack is another module of netfilter). You can look at the content of this table with `sudo conntrack -L`.



```
root@test-vm:~$ sudo conntrack -L -d 10.0.0.99
tcp      6 431969 ESTABLISHED src=172.16.1.8 dst=10.0.0.99 sport=32000 dport=80 src=10.0.0.99 dst=10.0.0.1 sport=443 dport=32000 [...]
```

connection from a container 172.16.1.8:32000 to 10.0.0.99:80 on host 10.0.0.1

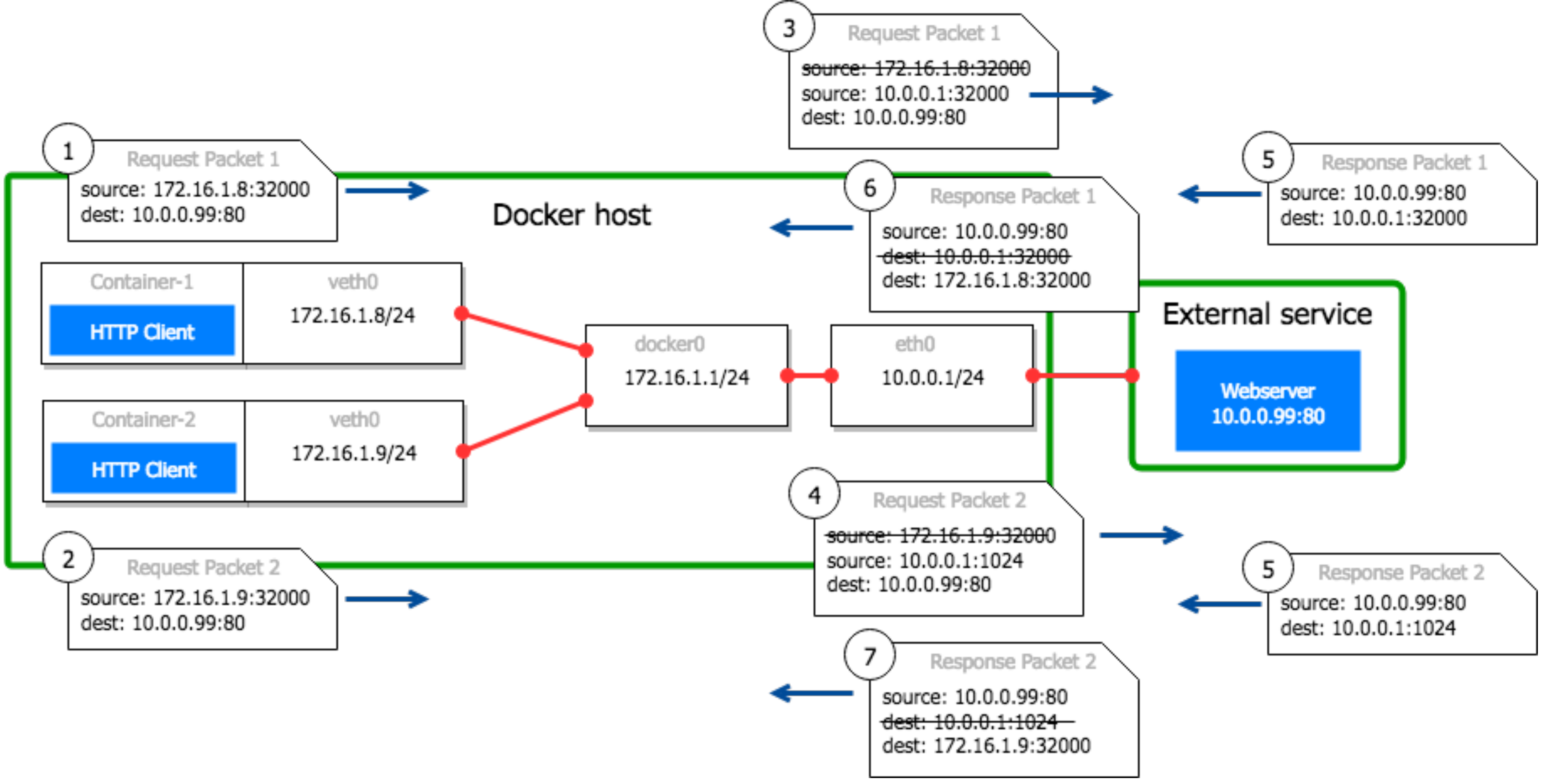
Port translation

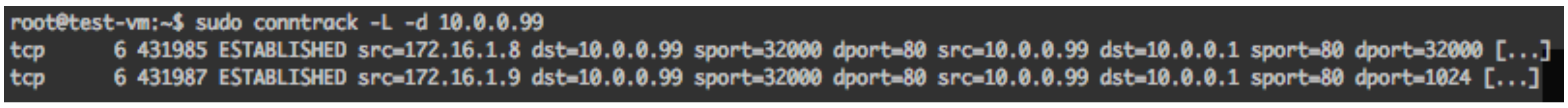
A server can use a 3-tuple ip/port/protocol only once at a time to communicate with another host. If your SNAT pool has only one IP, and you connect to the same remote service using HTTP, it means the only thing that can vary between two outgoing connections is the source port.

If a port is already taken by an established connection and another container tries to initiate a connection to the same service with the same container local port, netfilter therefore has to change not only the source IP, but also the source port.

Example with two concurrent connections: Our Docker host `10.0.0.1` runs an additional container named container-2 which IP is `172.16.1.9`.

1. container-1 tries to establish a connection to `10.0.0.99:80` with its IP `172.16.1.8` using the local port `32000`
2. container-2 tries to establish a connection to `10.0.0.99:80` with its IP `172.16.1.9` using the local port `32000`
3. The packet from container-1 arrives on the host with the source set to `172.16.1.8:32000`. There is no entry with `10.0.0.1:32000` in the table so the port 32000 can be kept. The Docker host replaces the source header from `172.16.1.8:32000` to `10.0.0.1:32000`. It adds a conntrack entry to keep track of the tcp connection from `172.16.1.8:32000` to `10.0.0.99:80`, which was SNATed to `10.0.0.1:32000`
4. The packet from container-2 arrives the host with the source set to `172.16.1.9:32000`.`10.0.0.1:32000` is already taken to communicate with `10.0.0.99:80` using tcp. The Docker host take the first available port (`1024`)and replaces the source header from `172.16.1.8:32000` to `10.0.0.1:1024`. It adds a conntrack entry to keep track of the tcp connection from `172.16.1.9:32000` to `10.0.0.99:80`, which was SNATed to `10.0.0.1:1024`
5. The remote service answers to both connections coming from `10.0.0.1:32000` and `10.0.0.1:1024`
6. The Docker host receives a response on port `32000` and changes the destination to `172.16.1.8:32000`
7. The Docker host receives a response on port `1024` and changes the destination to `172.16.1.9:32000`





```
root@test-vm:~$ sudo conntrack -L -d 10.0.0.99
tcp      6 431985 ESTABLISHED src=172.16.1.8 dst=10.0.0.99 sport=32000 dport=80 src=10.0.0.99 dst=10.0.0.1 sport=80 dport=32000 [...]
tcp      6 431987 ESTABLISHED src=172.16.1.9 dst=10.0.0.99 sport=32000 dport=80 src=10.0.0.99 dst=10.0.0.1 sport=80 dport=1024 [...]
```

this is how the conntrack table would look like

Note: when a host has multiple IPs that it can use for SNAT operations, those IPs are said to be part of a SNAT pool. This is not our case here.

Back to the original story

Our packets were dropped between the bridge and `eth0` which is precisely where the SNAT operations are performed. If for some reason Linux was not able to find a free source port for the translation, we would never see this connection going out of `eth0`. We decided to follow that theory.

There was a simple test to verify it. To try pod-to-pod communication and count the slow requests. We ran that test and had very good result. Not a single packet had been lost.

We had to look deeper into conntrack!

Conntrack in user-space

We had the strong assumption that having most of our connections always going to the same `host:port` could be the reason why we had those issues. However, at this point we thought the problem could be caused by some misconfigured SYN flood protection. We read the description of network Kernel parameters hoping to discover some mechanism we were not aware of. We could not find anything related to our issue. We had already increased the size of the conntrack table and the Kernel logs were not showing any errors.

The second thing that came into our minds was port reuse. If we reached port exhaustion and there were no ports available for a SNAT operation, the packet would probably be dropped or rejected. We decided to look at the conntrack table. This also didn't help very much as the table was underused but we discovered that the conntrack package had a command to display some statistics (`conntrack -S`). There was one field that immediately got our attention when running that command: "*insert_failed*" with a non-zero value.

We ran our test program once again while keeping an eye on that counter. The value increased by the same amount of dropped packets, if you count one packet lost for a 1-second slow requests, 2 packets dropped for a 3 seconds slow requests.

The man page was clear about that counter but not very helpful: *"Number of entries for which list insertion was attempted but failed (happens if the same entry is already present)."*

In which context would such an insertion fail? Dropping packets on a low loaded server sounds rather like an exception than a normal behavior.

We decided to figure this out ourselves after a vain attempt to get some help from the netfilter user mailing-list.

Netfilter NAT & Conntrack kernel modules

After reading the kernel netfilter code, we decided to recompile it and add some traces to get a better understanding of what was really happening. Here is what we learned.

The NAT code is hooked twice on the `POSTROUTING` chain (1). First to modify the packet structure by changing the source IP and/or PORT (2) and then to record the transformation in the conntrack table if the packet was not dropped in-between (4). This means there is a delay between the SNAT port allocation and the insertion in the table that might end up with an insertion failure if there is a conflict, and a packet drop. This is precisely what we see.

When doing SNAT on a tcp connection, the NAT module tries following (5):

1. if the source IP of the packet is in the targeted NAT pool and the tuple is available then return (packet is kept unchanged).
2. find the least used IPs of the pool and replace the source IP in the packet with it
3. check if the port is in the allowed port range (default `1024-64512`) and if the tuple with that port is available. If that's the case, return (source IP was changed, port was kept). *(note: the SNAT port range is not influenced by the value of the `net.ipv4.ip_local_port_range`kernel parameters)*
4. the port is not available so ask the tcp layer to find a unique port for SNAT by calling `nf_nat_l4proto_unique_tuple()` (3).

When a host runs only one container, the NAT module will most probably return after the third step. The local port used by the process inside the container will be preserved and used for the outgoing connection. When running multiple containers on a Docker host, it is more likely that the source port of a connection is already used by the connection of another container. . In that case, `nf_nat_l4proto_unique_tuple()` is called to find an available port for the NAT operation.

The default port allocation does following:

1. copy the last allocated port from a shared value. This value is used a starting offset for the search
2. increment it by one

3. check if the port is used by calling `nf_nat_used_tuple()` and start over from 2. if that's the case
4. update the shared value of the last allocated port and return

Since there is a delay between the port allocation and the insertion of the connection in the conntrack table, `nf_nat_used_tuple()` can return true for a same port multiple times. And because `nf_nat_l4proto_unique_tuple()` can be called in parallel, the allocation sometimes starts with the same initial port value. On our test setup, most of the port allocation conflicts happened if the connections were initialized in the same 0 to 2us. Those values depend on a lot a different factors but give an idea of the timing order of magnitude.

netfilter also supports two other algorithms to find free ports for SNAT:

- using some randomness when settings the port allocation search offset. This mode is used when the SNAT rule has a flag `NF_NAT_RANGE_PROTO_RANDOM` active.
- using full randomness with the flag `NF_NAT_RANGE_PROTO_RANDOM_FULLY`. This takes a random number for the search offset.

`NF_NAT_RANGE_PROTO_RANDOM` lowered the number of times two threads were starting with the same initial port offset but there were still a lot of errors. It's only with `NF_NAT_RANGE_PROTO_RANDOM_FULLY` that we managed to reduce the number of insertion errors significantly. On a Docker test virtual machine with default masquerading rules and 10 to 80 threads making connection to the same host, we had from 2% to 4% of insertion failure in the conntrack table.

With full randomness forced in the Kernel, the errors dropped to 0 (and later near to 0 on live clusters).

Activating full random port allocation on Kubernetes

The `NF_NAT_RANGE_PROTO_RANDOM_FULLY` flag needs to be set on masquerading rules. On our Kubernetes setup, Flannel is responsible for adding those rules. It uses iptables which it builds from the source code during the Docker image build. The iptables tool doesn't support setting this flag but we've committed a small patch that was merged (not released) and adds this feature.

We now use a modified version of Flannel that applies this patch and adds the `--random-fully` flag on the masquerading rules (4 lines change). The conntrack statistics are fetched on each node by a small DaemonSet, and the metrics sent to InfluxDB to keep an eye on insertion errors. We have been using this patch for a month now and the number of errors dropped from one every few seconds for a node, to one error every few hours on the whole clusters.

Conclusion

With the fast growing adoption of Kubernetes, it is a bit surprising that this race condition has existed without much discussion around it. The fact that most of our application connect to the same endpoints certainly made this issue much more visible for us.

Some additional mitigations could be put in place, as DNS round robin for this central services everyone is using, or adding IPs to the NAT pool of each host.

In the coming months, we will investigate how a service mesh could prevent sending so much traffic to those central endpoints. We will probably also have a look at Kubernetes networks with routable pod IPs to get rid of SNAT at all, as this would also also help us to spawn Akka and Elixir clusters over multiple Kubernetes clusters.

I want to thank Christian for the initial debugging session, Julian, Dennis, Sebastian and Alexander for the review