DNS issues in Kubernetes @ [redacted]

DNS issues in Kubernetes @ [redacted]
- Summary
- DNS timeout
  - Symptom
  - Scope
  - Investigation
  - Summary
  - Workarounds
    - Parallel
    - Socket
    - UDP
    - IPv4
  - Solution
    - iptables
- DNS resolution failure
  - Symptom
  - Scope
  - Investigation
    - Test script
    - Istio
    - What?
    - Clone wars
    - Have faith
    - CoreDNS
    - Search path
    - Bits and pieces
    - Curl
    - Resolut
    - CureDNS
    - Cash
    - AI Path
    - Fruit of mistake
    - Context
    - Reproduce
      - Step 1
      - Step 2
      - Step 3
      - Step 4
    - If
  - Summary
  - Workarounds
  - Solution
    - AAAA

Summary

We've been experiencing two separate type of DNS related issues in our Kubernetes clusters. This document aims to explain both in details. The two issues have different symptom. One results **increased DNS** query response **time** in 5 second increments, the other has no latency but occasionally **fails** to **resolve** A **records**.

DNS timeout

Symptom

Sometimes we see increased latency in New Relic of our services up to seconds, which supposed to respond under a second. Usually the latency is somewhere near multiple of 5 seconds, for example ~6.1 second, or ~10.5 seconds.

Scope

This type of DNS issue occurs around **5 times per day** in our production cluster, based on my metrics.

Investigation

This is the result of the netfilter conntrack module which is used by `kube-proxy` to maintain connection tracking. For TCP packets connection tracking is easy, as TCP is connection oriented and it has its own methods of connection create (3-way handshaking) and teardown (4-way terminating). UDP has no such concept as connection at the transport layer, so to emulate connections one very common method is to use the combination of

- source IP
- source port
- destination IP
- destination port

representing a connection, so whenever there is an outgoing UDP packet these four values together form a composite unique connection ID, a new connection. When a UDP packet arrives back using these same infos, the connection can be considered closed.

Since DNS by default is using UDP, and most modern OS' are sending queries for both ipv4 (A) and ipv6 (AAAA) records to the DNS server, this connection tracking algorithm can cause problems. As the resolver sends both the A and AAAA queries out on the same socket right after each other, it means that even though there are 2 packets leaving the host (one for A another for AAAA) there will only be a single connection recorded.

When the response to these two queries arrive, the first one will be considered as closing the connection so the second one will be dropped by iptables, so it'll never arrive to the original requestor, the resolver. As the resolver won't receive one of the packets (it's random which one was faster, the A or AAAA packet) it'll wait 5 seconds to timeout the lost packet before it resends the queries again. As a result, the DNS resolution will at least take 5+ seconds.

If there is a bad luck, this can even happen twice after each other, resulting a total of 10 seconds delay in the DNS resolution, true though the chance are fairly low for this to happen.

Summary

Modern DNS resolvers are sending out both IPv4 and IPv6 queries for a single request. These two queries are two separate UDP packets, which if send from the same socket will account as a single connection. As a result the first packet, whichever arrives first will be accounted for as closing the connection, which will result the second, outstanding packet to be dropped. This leads to DNS timeout of 5 seconds, for each packet drop.

This is a limitation of netfilter's connetion tracking (conntrack), which we use in our `kube-proxy` to set up network policies and routing in our cluster.

Workarounds

There are few workarounds to this problem, although we have to differentiate two library groups as their level of support for these varies. In one group, there are Linuxes which are glibc based, like Ubuntu, and the other group is musl based, like Alpine.

Parallel

Normally the resolver sends out the A and AAAA queries "at the same time", in parallel, which is part of the problem. There is a flag called `single-request` which will switch the parallel mode into sequential, so the second request will wait before the first one arrives. As this is a resolver option, it must be set in `/etc/resolv.conf`.

Socket

Part of the problem is that by default the resolver is using the same socket to send both A and AAAA queries. As a result the two UDP packets will have the same quad info which will account as the same connection. There is an option called `single-request-reopen` which will not reuse the same socket but will use a different one for each UDP packet. This will result a different source port, so the quad info will be different for the A and AAAA packets, so the two packets will account for two separate connections. As this is a resolver option, it must be set in `/etc/resolv.conf`.

UDP

We can tell the resolver by setting `use-vc` to use TCP instead of UDP for DNS communication. Since TCPs are easily trackable and not suffer from the same problem what UDP does, this also will resolve this issue. Unfortunately TCP is an expensive and slow protocol compared to UDP. As this is a resolver option, it must be set in `/etc/resolv.conf`.

IPv4

If we would avoid having IPv6 queries in the first place that would also solve this problem. We can achieve that by setting the `net.ipv6.conf.all.disable_ipv6` and `net.ipv6.conf.default.disable_ipv6` kernel parameters. Positive side is that even though the previous workarounds were all DNS only cases, this solution helps all UDP traffic.

Solution

iptables

Alternatevily we have to look for an iptables replacement which doesn't use netfilter conntrack. Two candidates are

- Cilium
- ipvs

These both have their own connention tracking implementation, and we have to test whether they are affected by this type of issue.

Unfortunately none of the `resolv.conf` solutions are supported by musl, only by glibc. Also worth taking into account that we can configure the `resolv.conf` solutions via the `deployment` , we don't need to change the image at all. Potentially the best solution would be moving away from using netfilter connection tracking/iptables, to a solution which handles UDP tracking properly. This is the most laborous of all, but the most rewarding as well as it tackles the problem at its root.

DNS resolution failure

Symptom

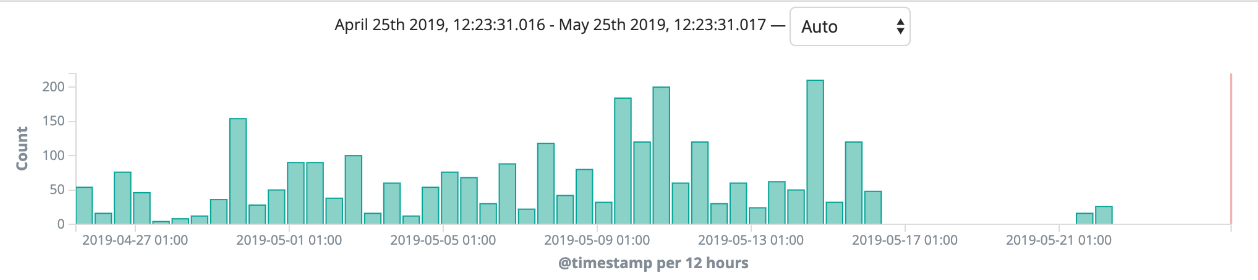
The most well-known symptom we experienced about this issue was related to curl and Twilio, with the following error message:

```
curl: (6) Could not resolve host: api.twilio.com
```

This originally hit us as an exception in `clientapi`, the Twilio SDK failed to connect to `api.twilio.com`.

```
message: Services_Twilio_TinyHttpException: Could not resolve host:
api.twilio.com in
/opt/project/source/vendor/twilio/sdk/Services/Twilio/TinyHttp.php:119
```

Once we tracked down where this comes from, we saw that the Twilio SDK is using `curl` to connect to `api.twilio.com`. These are the the failure events in the past month:



The majority of these cases are related to `api.twilio.com`, but this issue is **not limited** to `api.twilio.com`, or **even** to **curl**. This is a DNS resolution problem which affects **any** external **hostname**. We even see occasions where this issue impacts resolving our S3 buckets as well.

On May 16th we implemented a workaround by defining static IPs for `api.twilio.com`, which dropped the visible failure count significantly, but this is just a temporary solution and also won't prevent from other external names to fail within our cluster.

Scope

Just in **May, we had 2378** occasions, or **~80 per day**, when we failed to resolve a hostname using curl. This is just curl, so most likely we had more resolution failures than this.

Investigation

First and foremost if I want to fix a problem, I have to be able to reproduce the symptom. If I can't reproduce it, I can't succeed. I can't test it, I can't analyze it, and the worst is that I can't provide proof that my solution works.

So as the firt step, I always try to isolate the symptom, replicate it an a controlled environment. Since this issue hit us in production, my tools are limited of what I can use to analyze the case. I was able to track down very fast that the exception we saw from clientapi, through the twilio SDK is actually a curl call. I can call curl outside of clientapi, so step one complete.

Because at that time I knew nothing about the issue, I wanted to stay as close as possible to the original problem, sa I couldn't rule out anything. I started testing within the clientapi, using curl. I wrote a small shell script, which calls curl every 5 seconds. I could have had a more frequent call, but I chose 5 seconds to make sure I'm not putting load on the production infrastructure.

When I call `curl https://api.twilio.com`, I get this:

```
<?xml version='1.0' encoding='UTF-8'?>
<TwilioResponse><Versions><Versions><Version><Name>2010-04-01</Name>
<Uri>/2010-04-01</Uri><SubresourceUris><Accounts>/2010-04-
01/Accounts</Accounts></SubresourceUris></Version></Versions></Versions>
</TwilioResponse>
```

Test script

That's not bad but too long for effective analysis. I knew that I had to analyze hours, maybe days of data, so I had to cut out any noise I could to be get an already filtered result to work with. So, I came up with this tiny script:

```
#!/bin/sh

while true
do
  date | tr '\n' ' '
  curl https://api.twilio.com 2>&1 | egrep "TwilioResponse|Could not resolve" | sed 's/^.*curl/curl/g;s/<Versions>.*$//g'
  sleep 5
done
```

I called it `test1.sh` because I'm *creative*.

This helped me to get a single keywork when we can query Twilio, and the curl error message when we can't. For example:

```
root@testimg7:/var/tmp# ./test1.sh
Sat May 25 00:58:10 UTC 2019 <TwilioResponse>
Sat May 25 00:58:15 UTC 2019 <TwilioResponse>
Sat May 25 00:58:21 UTC 2019 <TwilioResponse>
Sat May 25 00:58:26 UTC 2019 <TwilioResponse>
...
```

I can work with that. Nice, tidy. So, I started to run my scipt in the clientapi Pod. One step forward so I can't move away from production while replicating the problem. Within minutes, this is what I got:

```
Sat May 25 00:58:37 UTC 2019 <TwilioResponse>
Sat May 25 00:58:42 UTC 2019 <TwilioResponse>
Sat May 25 00:58:47 UTC 2019 <TwilioResponse>
Sat May 25 00:58:53 UTC 2019 curl: (6) Could not resolve host: api.twilio.com
Sat May 25 00:58:58 UTC 2019 curl: (6) Could not resolve host: api.twilio.com
Sat May 25 00:59:03 UTC 2019 curl: (6) Could not resolve host: api.twilio.com
Sat May 25 00:59:08 UTC 2019 <TwilioResponse>
Sat May 25 00:59:13 UTC 2019 <TwilioResponse>
Sat May 25 00:59:18 UTC 2019 <TwilioResponse>
Sat May 25 00:59:24 UTC 2019 <TwilioResponse>
Sat May 25 00:59:29 UTC 2019 <TwilioResponse>
Sat May 25 00:59:34 UTC 2019 <TwilioResponse>
```

It was time for celebration, as I achieved my first goal: replicate the issue witout clientapi/twilio SDK.

Istio

Since clientapi is in the default namespace, and this namespace is Istio enabled, my first **guess** was that the issue could be related to **Istio**. Even though Istio is a TCP proxy and doesn't process UDP at all, Istio is usually among our top candidates when it comes to potential root cause of bizarre behaviours.

Since. I was at the very beginning of my journey, I knew that I can even shoot in the dark, even though that's not really my style but it helps to identify the scope of the issue, in other words who are affected? Once I can find out the components that are consistently affected, I can start looking for their common pattern, which supposed to get me closer to the root cause. Also, shooting in the dark is fun. We have to make the investigation fun so our brain enjoys solving it. Joy is a great motivator for anything brain related. Or at least I was told.

Spinned up two Pods for testing. Wanted to see if we experience this even outside of Istio, and outside of clientapi. One pod was in the `sre` namespace (Istio disabled) the other was next to clientapi, in the `default` namespace.

To my surprise, the Pod in the `sre` namespace didn't show any symptoms, but the the one in the `default` namespace did. Haha, Istio. I knew it. I always warn myself not to fell in love with my idea, so how could I confirm that it's Istio indeed? By spinning up a Pod as similar as the the istio enabled ones, but not istio enabled. Got it! So, I have to spin up a pod in the `default` namespace, but opt-out of the Istio injection.

Cool, let's do this quickly, and claim victory of the mighty **Logic**.

We can opt-out of Istio with the annotation `sidecar.istio.io/inject: "off"`, so I spin up my Pod, ran my sweet `test1.sh` script.. and it keps producing me the symptom.

What?

So I experience this even outside of Istio in the `default` namespace, but I don't experience this in the `sre` namespace? How is that possible? Also, god why you hate me? So it's not Istio. Fair enough.

Ok, so I know that it's **not Istio**. **How** the namespace comes into play? No idea. I know that the Pods are using their namespace as part of the DNS resolution, but this still doesn't add up. So, knew that I had to dig deeper. Install wireshark, pick up strace, trace, maybe even gdb. Luckily I already have a test image built for such purposes. I call it `testimg`, because I'm *creative*.

I also knew that I need a lab to play with, and since don't want to risk any production impact, I wanted to move this off of our production cluster. So, I tested this symptom in our lower environments. `dev`, impacted.. GOOD. `feat1`... not. `stag`, neither. What #2? How?

I've set up my test script in the Kubernetes **node** itself, which hosts the Pod. It never made a mistake, always received the correct answer.

I've set up a second test script which ran in parallel with my twilio test. It was querying Amazon S3, a **different** target. I was hoping to see to fail resolving at the same time when twilio fails to resolve. It **wasn't**. Even when twilio failed to resolve, Amazon S3 resolution was **working fine**.

Anyway, at least luckily I have the symptom in `dev`, where I can do whatever I want to hunt this down. So, I moved to dev. Let's see which components are suspect so far:

- I can rule out Istio. Istio has an alibi. It was busy preparing 503 packages for christmas.
- CoreDNS
- Upstream DNS (Amazon)
- Networking/Calico
- Node
- curl
- source Linux Kernel

First enabled logging in CoreDNS to find out if I see anything useful, but it was too noisy I couldn't make any sense of it. Also, I couldn't trust any of the components listed above, so I also wanted to have an extra pair of eyes on the DNS traffic. Wanted to run Wireshark both inside CoreDNS and the source Pod, to see if we're losing any packets. Wanted full control.

Clone wars

So, I can replicate the problem, now I have to isolate it as much as possible reducing noise so I can inspect each component one by one, and find out who's the murderer. So, I set up a second DNS service called `test-kube-dns` (the original is called `kube-dns`), which had a single Pod for the sake of simplicity. That pod is running my testimg with all the tools. I fetched CoreDNS and ran it manually. Server side is ready for testing.

That's nice to have test server, but we need a Pod which will be configured to use the test server. Normally `resolv.conf` is configured by Kubernetes, but we can override that, so I defined this for pod of my test Pods in `sre` and the `default`.

```yaml
dnsConfig:
   nameservers:
   - 100.69.111.130
   options:
   - name: ndots
     value: "5"
   searches:
   - default.svc.cluster.local
   - svc.cluster.local
   - cluster.local
   - eu-west-1.compute.internal
```

I was happy with the test environment, I had a test server with all the tools I need and 2 test pods, one in `sre` another one in `default`, none of them Istio enabled.

Let's start testing.

Have faith

At this stage I prayed to see the symptom again, because that would be a huge step ahead in my analysis. So, I started to run my test script. Waited.. 5 minutes. Nothing. 15 minutes. Nothing. I started to get nervous. 30 minutes, no symptom.

As a desperate move I started my test script in my other test Pod as well. I left my other test on. After a few minutes, I started to see:

```
Sat May 25 03:29:44 UTC 2019 <TwilioResponse>
Sat May 25 03:29:49 UTC 2019 <TwilioResponse>
Sat May 25 03:29:54 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
Sat May 25 03:29:59 UTC 2019 <TwilioResponse>
Sat May 25 03:30:05 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
Sat May 25 03:30:10 UTC 2019 <TwilioResponse>
Sat May 25 03:30:15 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
Sat May 25 03:30:20 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
Sat May 25 03:30:25 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
Sat May 25 03:30:30 UTC 2019 <TwilioResponse>
Sat May 25 03:30:35 UTC 2019 curl: (6) Could not resolve host:
api.twilio.com
```

The odd thing was that the failures were in sync with each other. If I ran only one of the test, either of them, I had no problem. But when I ran both of them, I started to see the symptom again.

So, I my attention moved towards CoreDNS. It should be CoreDNS.

CoreDNS

We use this configuration for CoreDNS:

```
.:53 {
    log
    errors
    health
    kubernetes cluster.local. in-addr.arpa ip6.arpa {
      pods verified
      upstream
      fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    loop
    cache 30
    loadbalance
    reload
    autopath @kubernetes
}
```

So, I enabled logging in my test CoreDNS and also started to record the network traffic. I can't rule out none of these components. The most obvious question was, are we losing UDP packets? This is what comes to your mind as an obvious option, but it's not realistic. Our client Pod *receives* the DNS answer, it's just not the correct one.

So, I started to record everything, and I saw a few thing interesting things that I wasn't suspicious about, even though it was weird. One was autopath, which aims to take over the resolution of external names from the client to the server side, so it reduces the overall DNS traffic between CoreDNS and the Pods. We know it works, we saw the DNS request count drops when we enabled it. It's a pretty cool feature. During my tests, autopath needed time to get triggered, at the beginning of the session it was the client side which took care of the complete resolution.

Search path

In DNS resolution there is a search path defined `/etc/resolv.conf`, which helps the resolver to find out which domain your entry belongs to. This is important to support short names, so if you're in the domain of `cnn.com`, and you want to query `www.cnn.com`, you can just type `ping www` and the resolver will try all your default domains the request.

Our search path looks like this, for example:

```
root@testimg11:/var/tmp/curl# more /etc/resolv.conf
nameserver 100.69.111.130
search default.svc.cluster.local svc.cluster.local cluster.local eu-west-1.compute.internal
options ndots:5
```

It's worth to note that the `default` keyword is the namespace of the pod, so our `resolv.conf` always reflect which namespace our Pod is in. This goes back to the Kubernetes DNS mechanism on how you resolve Pods and Services, and it's a normal behaviour.

So, just to put this theory into practice, if you call `curl https://api.twilio.com` this will trigger a name resolution of `api.twilio.com`, but the resolver will try to resolve this in the following order:

- `api.twilio.com.default.svc.cluster.local`
- `api.twilio.com.svc.cluster.local`
- `api.twilio.com.cluster.local`
- `api.twilio.com.eu-west-1.compute.internal`
- and finally, `api.twilio.com`

The reason is that the resolver believes you used a short name. There are two ways you can change this behavior:

- add a dot at the end of your entry, for example: `curl api.twilio.com.`. This will tell the resolver that this is **not** a short name, it's an absolute FQDN so we don't need the search

path. Just resolve as it is.
- the parameter called `ndots` you see in the `resolv.conf`, which is a number, in this case 5. This means that if my entry has more than 5 dots in it (it doesn't, `api.twilio.com` has only 2), the order of the request will change in a way that my original entry will be queried *first*, instead of *last*. 5 is a well known and fair compromise in Kubernetes clusters, and works fairly well.

Just to clarify, these options are all change on ***how*** effective the DNS resolution is, ***not*** whether it works or not.

To sum up, we have **autopath** which supports to query **external** addresses, and we have the **local** search path in our `resolv.conf` which will help our resolver library to resolve the address anyway, even if autopath doesn't work, for whatever reason. It needs some warmup time. Or it's Christmas. Or Hannukah.

Bits and pieces

So I had my first packet dump of the DNS conversation which fails after a few minutes. Great. Let's review that. One thing that occurred to me, is that whenever we start failing the DNS resolution with curl, the client side, the Linux side stops going through the search path.

It works for a few minutes, it goes through the search path for a few minutes, and suddenly it just gives up on the first item, which is `api.twilio.com.default.svc.cluster.local`.

I couldn't understand why, so I had to go deeper. I was sure that I found a bug on the client side. I was comparing two DNS packets, they were **identical**, the first still triggered the search path in Linux, the second did not. My suspicion from CoreDNS moved to Linux, more precisely to curl or the resolver.

What #n ? Why? How?

Two identical DNS packets, 5 seconds apart, the first triggers the search path (correct way), the second doesn't (incorrect way). I see why this results a DNS resolution problem. I knew it couldn't be the resolver, as this would be too big of a problem. This *must* be a curl issue. Maybe they use the resolver in a unique way?

Curl

I picked up gdb and downloaded curl to recompile it. I wanted to see where it goes wrong. I enabled debugging symbols and disabled all C optimisation. When we compile with optimisation, we can't inspect variables while debugging. I wanted to see it all. I'm Neo. Unfortuntely I'm closer to Butters in South Park than to Keanu Reeves, but that's good enough. Professor Chaos always has a few tricks in his sleeve. Never fails to disappoint.

So, I enabled all debugging, started to drop breakpoints all over the place, and I had to realize that the curl is not the issue here. I even tried with the latest curl, same issue. It goes deeper. The code path leads me to getaddrinfo, dns-host and res-query.

Resolut

My next step has to go deeper, to the resolver, and I was able to pinpoint the code path which acts differently when we give up on the search path. The code path was interesting: it stopped going through the search path, because we received an answer. I couldn't believe it. No, we **didn't**. So, I went deeper, when I realised, actually we might have. Just not what I expected.

The `libresolv` stopped going through the search path for our entry `api.twilio.com.default.svc.cluster.local` despite the fact that we received an NXDOMAIN for our A record, because we received a NOERROR for our AAAA record.

Wow. That is interesting. For many reasons. One, if you look up the the A record for `api.twilio.com`, it has a healthy A record:

```
ip-10-12-8-46:~ sendai$ host -t a api.twilio.com
api.twilio.com is an alias for virginia.us1.api-lb.twilio.com.
virginia.us1.api-lb.twilio.com is an alias for nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com.
nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com has address 18.208.54.140
nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com has address 18.211.224.155
nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com has address 18.212.47.248
```

It has a broken AAAA record. Why? Because it returns a CNAME:

```
ip-10-12-8-46:~ sendai$ host -t aaaa api.twilio.com
api.twilio.com is an alias for virginia.us1.api-lb.twilio.com.
virginia.us1.api-lb.twilio.com is an alias for nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com.
```

Which is **not** NXDomain and it leads to **no** AAAA addresses:

```
ip-10-12-8-46:~ sendai$ host -t aaaa nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com
nlb-api-public-c3207ffe0810c880.elb.us-east-1.amazonaws.com has no AAAA record
```

Let's think this through, a few observations/questions occurred to me:

- if instead of a broken AAAA record Twilio would not have an AAAA record at all, the local resolver would keep going through the search path, and finally would be able to resolve the address
- how the hell we receive a half correct record from CoreDNS? True that Twilio has a broken AAAA record, but its A record is correct, so we should either have both A and AAAA complete for `api.twilio.com.default.svc.cluster.local` if autopath works, or we should have NXDomain for both if autopath doesn't work. How we get to a state where it's half/half?

CureDNS

It was time for my puzzled suspicion to turn again, back to my old friend CoreDNS. The question was: why do I get an inconsistent A/AAAA response from CoreDNS?

Previously tested the symptom with all CoreDNS versions, including the latest one, and they were all affected. It was time to fetch CoreDNS and recompile it with debug symbols and no optimization. Unfortuntely CoreDNS logging is insanely weak, most modules don't even log errors, not that helpful tracing so I can understand the inner workings of this odd beast.

Cash

Going back to my raw packet dump, I started to review again and again, to make sure I'm not missing anything. After a while, I was able to point out the exact problem. The cache within CoreDNS can become inconsistent as occasionnaly for the same request autopath either works or not. Since the cache module is also caching autopath results, that's how we end up having half consistent records in cache of A and AAAA records for the same entry, one NXDomain one NOERROR. When that happens, any client that's trying to resolve that entry will fail to do so, until the cache expires in CoreDNS.

One more question to ask:

- The cache is set to 30 seconds, **how** these items **expire at a different pace** and not at the same time?

The reason is that the TTL we receive for A and AAAA differs, and TTL is stronger than the cache expire value. If both TTLs are larger than the cache expire time, they'll expire together, which is fine, as whatever method we use to resolve, autopath or without autopath, it'll be consistent. The problem is that when the TTLs are smaller than the cache timeout value and different from each other. In that case, they both will expire from the cache in different times, which will leave room to make a query. If that query is different from the one which still in cache, in a sense that it's autopath enabled or not, BANG. We hit inconsistency.

We could blame the cache here, but a better question is why autopath stops working occasionally? Under what circumstances it stops working? If it would work consistently, that would keep the cache consistent, but as it isn't it result a cache entry for either NXDomain for A or AAAA in cases when autopath wasn't triggered, and NOERROR when it was. This half state will break our client resolvers.

Next step: Identify why autopath decides to stop functioning occasionally. *Maybe* it has AI in it?

AI Path

At this stage I knew that the whole problem is around inconsistent cache state, which is caused by the fact that autopath is triggered in same cases while it's not in others, for the same entry. So I downloaded CoreDNS, and decided to run it through gdb, but this type of analysis is fairly challenging, as the DNS server still has to be fairly responsive, even during my analysis. Having breakpoints and manually controlling the process flow is too slow for that, also CoreDNS is fairly simple, so I knew I had to go with another approach.

As a matter of fact CoreDNS is *too* simple. Even too simple to have proper logging within the plugins, otherwise I just would have tuned up the log level and viola. Not here, not with CoreDNS. So, my approach was that I went through the code and placed analytical checkpoints where I print the significant parts of the internal state. I just needed visibility.

Once I placed my extra logging into CoreDNS I ran my first few tests, I started to see something interesting.

```
[INFO] plugin/autopath: FirstInSearchPath:
api.twilio.com.default.svc.cluster.local. - [argo-events.svc.cluster.local.
svc.cluster.local. cluster.local. eu-west-1.compute.internal. ]

```

I'm playing in an isolated test scenario with my own dedicated test Pods and test CoreDNS. My Pod is called `testimg7` and it's in the `default` namespace.

How the hell `argo-events` namespace comes into play here?

The reason I added a checkpoint to FirstInSearchPath, because if this returns False autopath won't get applied. Based on this output, I'm not surprised. We query from the `default` namespace, querying `api.twilio.com`, which ends up being `api.twilio.com.default.svc.cluster.local.` because of our local search path.. so far that's normal.

Where `argo-events.svc.cluster.local.` comes from and **why** it's here?

Autopath works in a way that whenever there is a request coming to CoreDNS, autopath has to figure out the client's search path in `/etc/resolv.conf`. It has to know what search path to try, for each request, and that's not constant, that's changing based on the namespace. That's not easy, as it has no access to the Pod at all. But, it can reconstruct it.

Once autopath reconstructed the search path of the client, it'll compare the request. If the request ends with the first domain in the search path, autopath is triggered and will do its thing. If not, it won't process the request. This feature is not that relevant in Kubernetes, as this always should be the case, but autopath supports other modules as well, not just Kubernetes where you have to check this.

Autopath expects the plugin to do this job on behalf of autopath. In this case, this is the Kubernetes plugin. Kubernetes plugin is doing this by getting the source IP of the Pod, and asking Kubernetes:

Which namespace this IP belongs to?

This way, autopath in Kubernetes plugin in CoreDNS can reconstruct the search path of the source Pod.

CoreDNS is clearly making a mistake here, the reconstructed search path is incorrect, which makes autopath to opt-out. So, I had to go and see this myself.

"This IP is **not** in the `argo-events` namespace." - I told myself.

To my surprise, I was only partially correct. I saw this:

```
andras.spitzer@blue.dev.[redacted]com:~/tmp/coredns/testimgpod$ kubectl
get pods --all-namespaces -o wide | grep 100.104.247.253
argo-events     webhook-j78g2    0/2      Error         0   45d   100.104.247.253
argo-events     webhook-mx99l    0/2      Completed     0   30d   100.104.247.253
default         testimg7         1/1      Running       0   10d   100.104.247.253
```

As Kubernetes is reusing virtual IP addresses, even historical Pods will show up in the list, with that IP. And they *can* be in another namespace. And this is what happens in our case.

My image `testimg7` is in the `default` namespace and has the IP of `100.104.247.253`.

It's clear that only my test image is running, but the autopath code in the Kubernetes module only cares about the Pod's IP, *not* its state. It'll get the Pod info for the first item, `webhook-j78g2`, which is in the `argo-events` namespace.

Since our test Pod is in the `default` namespace, our query of `curl https://api.twilio.com` will first try to resolve `api.twilio.com.default.svc.cluster.local`, which when arrives to autopath will try to rebuild the search path of the source Pod by quering the Pod info using only the IP address. It'll get the Pod info for `webhook-j78g2`, which is a historical Pod in a different namespace, so autopath will build an incorrect search path, and `api.twilio.com.default.svc.cluster.local` won't be equal to the first domain in the search path, which in this case incorrectly will be `api.twilio.com.argo-events.svc.cluster.local`.

Autopath won't get applied, and since we since the A and AAAA records expire at different times, one of these entries will become NXDomain in the cache, while the other remains NOERROR.

This response will stop our clients to follow their local search path, so the app who initiated the connection/resolution will fail, just as curl did.

Fruit of mistake

Now we know that the problem is coming from the fact that the search path in a the source Pod (which is correct) is different what CoreDNS's autopath module believes (which is incorrect). How was I able to test reproduce this at the very beginning?

Because I **made a mistake**.

If you go back to **Clone wars**, you'll see that I've set up my test pods with the following manual DNS configuration:

```yaml
dnsConfig:
   nameservers:
   - 100.69.111.130
   options:
   - name: ndots
     value: "5"
   searches:
   - default.svc.cluster.local
   - svc.cluster.local
   - cluster.local
   - eu-west-1.compute.internal
```

My goal was to point my test pods to my test CoreDNS. What I didn't know at that time, that I also made a mistake which helped me to reproduce the symptom. I've set the search path of my test Pods this:

- `default.svc.cluster.local`
- `svc.cluster.local`
- `cluster.local`
- `eu-west-1.compute.internal`

This is fine for my test Pod in the `default` namespace, but it's incorrect for my Pod in the `sre` namespace. The one in the `sre` namespace should have had `sre.svc.cluster.local` as its first item. The fact that I made this mistake, even though my source Pod in the `sre` namespace wasn't impacted by this bug, it still allowed me to experience the symptom, just the other way around.

This time CoreDNS was correct and my Pod was incorrect. Very similar to what we have in production, just the other way around.

How odd is that a small mistake can lead towards a solution. This was the fruit of my mistake.

Context

We need to have the following items in place to trigger this bug:

- CoreDNS running with
  - cache enabled (`cache 30`, for example)
  - autopath enabled using the kubernetes plugin (`autopath @kubernetes`)
- glibc Linux (tested with Ubuntu 18.04.1 LTS)
  - may also work with musl/Alpine, haven't tested
- the context has to support IPv4 and IPv6
- 2 test Pods
  - one with the default (correct) search path
  - one with an incorrect search path (the first entry should suggest a different namespace, to emulate the scenario autopath fails to identify our Pod properly)
- an external address (for example, `api.twilio.com`) we want to resolve, and which
  - has a proper resolvable A record

  - has also an AAAA record with a CNAME, which has no AAAA record (broken configuration)
- the A and AAAA records must have different TTLs
  - so A and AAAA will expire at different times

Reproduce

Step 1

Write a shell script, `test.sh`, which will try to resolve `api.twilio.com` at every 5 seconds, for example:

```
#!/bin/sh

while true
do
  date | tr '\n' ' '
  curl https://api.twilio.com 2>&1 | egrep "TwilioResponse|Could not resolve" | sed 's/^.*curl/curl/g;s/<Versions>.*$//g'
  sleep 5
done
```

This script will give us two type of output:

```
Sat May 25 03:29:44 UTC 2019 <TwilioResponse>
Sat May 25 03:29:49 UTC 2019 <TwilioResponse>
```

When we were able to resolve `api.twilio.com`

and

```
Sat May 25 03:29:54 UTC 2019 curl: (6) Could not resolve host: api.twilio.com
```

When we failed.

Also, it wouldn't matter much if we increase the resolution frequency, as the inconsistency potentially can hit only when the CoreDNS entry cache expired and has to be repopulated.

Step 2

Set up two test Pods in the default namespace, called `test1` and `test2`. Let test1 configuration be default, while have this configuration for `test2`:

```
dnsConfig:
  searches:
    - test2.svc.cluster.local
    - svc.cluster.local
    - cluster.local
    - eu-west-1.compute.internal
```

This will make Kubernetes autopath to ignore this Pod, as when we'll try to resolve `api.twilio.com` which will turn into `api.twilio.com.default.svc.cluster.local`, this won't match `test2.svc.cluster.local` .

With this we simulate the case when autopath gets a historical Pod which used to have our IP, and which was in a different namespace, like `test`.

With this, `test1` will be served by autpath while `test2` won't.

Step 3

Start running our test script in `test2`, this will work, never fail, autopath will never kick in, `api.twilio.com.default.svc.cluster.local` will always result NXDomain, and the client side iterates through the search path. At the end of the seach path, it'll resolve `api.twilio.com` which has a proper A and a broken AAAA address. Curl will be happy as we have a proper A address. Always.

Step 4

Leave `test2` on, and start running the test script on `test` as well. Within 5-20 minutes you'll see

```
Sat May 25 03:29:54 UTC 2019 curl: (6) Could not resolve host: api.twilio.com
```

messages in sync with each other, randomly come and go, depending on whether our current cache is consistent/inconsistent. It can stay broken for only a few seconds even up to minutes, depending on the request patterns, cache settings and TTLs.

When the inconsistency hits, noone can resolve `api.twilio.com` via CoreDNS from the cluster.

If

- If we would use CoreDNS without cache, we would never hit this bug
- If we would use CoreDNS without autopath with the Kubernetes plugin, we would never hit this bug
- If we would use only IPv4 in this example, we would never hit this bug. Cache would still be inconsistent, sometimes `api.twilio.com.default.svc.cluster.local` would resolve as NOERROR when autopath was applied and would resolve to NXDomain when it wasn't, but even when it's NXDomain our client side resolver will keep going through the search path and finally would resolve `api.twilio.com`. We need both IPv4 and IPv6 entries to have this half-way inconsistency.
- If IPv6 address would be properly configured with twilio, we might hit this bug, depending on

whether our network can route IPv6.
- If `api.twilio.com` would have no IPv6 DNS entry configured at all we would never hit this bug.
- If the TTLs for the A and AAAA records would be in sync, we would never hit this bug, as both entries would expire at the same time. This means it would not give a chance to have A and AAAA records for the same entry, one triggered by a correctly identified and the other by an incorrectly identified Pod. Only one of these, which in itself would mean consistency, even with or without autopath. It would result differing performance, but not inconsistency.

As you see the bug is highly contextual.

Note: it's interesting to see why `libresolv` / `getaddrinfo` gives up on search path when for a single request it receives an IPv4 NXDomain and IPv6 CNAME which has no AAAA record, at the end, this is not sufficient info to initiate a connection to my requested address. I haven't dig deep enough to see whether this is a bug or feature, I only went down to confirm the behavior. Still, makes me wonder if this is the correct behavior.

Summary

Cache inconsistencies in CoreDNS leads to occasional DNS resolution problems in any client. The inconsistencies are coming from the fact that Autopath is occasionally identifies the source Pod incorrectly in Kubernetes.

Workarounds

In order of preference:

Disable cache

Disabling cache will make sure that we'll always get fresh data, also the cost of having an internal DNS query is low so the cache is not that important in our case. Even if we disable the cache and leave in autopath not all requests will trigger autopath, but at least it won't respond with an inconsistent state to our clients, so it should never fail to resolve as a result.

Disable autopath

If we disable autopath, the resolution will work always but the number of requests will increase 3-5 times, which in itself might result higher number of latency related issues we just discussed at the beginning.

Overall either of these options will result no more resolution failures, but their performance implications differ, hence the preference of leaving autopath in instead of the cache.

Solution

The solution is to make sure autopath either works all the time or none at all, so cache can remain consistent. If it works half the time, it'll result inconsistent cache entries, which will result failing DNS resolutions.

The autopath code in the Kubernetes plugin withint CoreDNS has to add a check to make sure they not just look up Pods based on IPs, but also make sure to look for Pods *only* that are Running, ignoring any previous Pods from the past.

Just opened a bug report with CoreDNS about this and proposed a fix.

AAAA

One of the reason why this bug is hitting us, as `api.twilio.com` has a a broken AAAA record. It returns a CNAME, which CNAME has no AAAA record at the end. This normally is not a big deal, as the A record has a proper address, but it's still a misconfiguration and can lead to consequences like this.

Unfortunately, even our own API endpoint is having the same issue:

```
andras.spitzer@blue.dev.[redacted]com:~$ host -t aaaa api.[redacted]com
api.[redacted]com is an alias for 2e9080ca-ambassador-ambass-61b5-1874540413.eu-west-1.elb.amazonaws.com.
```

Our AAAA record returns with a CNAME.

In itself is not a problem, but the alias has no AAAA record:

```
andras.spitzer@blue.dev.[redacted]com:~$ host -t aaaa 2e9080ca-ambassador-ambass-61b5-1874540413.eu-west-1.elb.amazonaws.com
2e9080ca-ambassador-ambass-61b5-1874540413.eu-west-1.elb.amazonaws.com has no AAAA record
```

This means we respond to an AAAA query with an entry which is not suitable for connecting. This is a broken configuration, which most of the time won't hurt us.. until it does, when the stars align at a client we have no control over. In other words, even though we tested this with `api.twilio.com`, this would fail with our own `api.[redacted]com` as well.

To prove my point, I created a second test script for `api.[redacted]com`:

```
#!/bin/sh

while true
do
  date | tr '\n' ' '
  curl https://api.[redacted]com 2>&1 | egrep "DOMContentLoaded|Could not resolve" | sed 's/^.*curl/curl/g;s/^.*DOMContentLoaded.*$/api.[redacted]com > OK/g'
  sleep 5
done
```

And the result is, within 5 minutes:

```
Mon May 27 10:55:52 UTC 2019 api.[redacted]com > OK
Mon May 27 10:55:57 UTC 2019 api.[redacted]com > OK
Mon May 27 10:56:02 UTC 2019 api.[redacted]com > OK
Mon May 27 10:56:07 UTC 2019 curl: (6) Could not resolve host:
api.[redacted]com
Mon May 27 10:56:12 UTC 2019 api.[redacted]com > OK
Mon May 27 10:56:17 UTC 2019 api.[redacted]com > OK
Mon May 27 10:56:22 UTC 2019 api.[redacted]com > OK
```

One luck we have is that our A and AAAA records' TTL are almost in sync, which leaves a smaller window for the inconsistency state to kick in. It' still fairly often, but less often than `api.twilio.com`.

And our S3 bucket name is also broken:

```
andras.spitzer@blue.dev.[redacted]com:~$ host -t aaaa [redacted]user-data.s3.eu-west-1.amazonaws.com
[redacted]user-data.s3.eu-west-1.amazonaws.com is an alias for s3-r-w.eu-west-1.amazonaws.com.
```

A CNAME pointing to a record with no AAAA address:

```
andras.spitzer@blue.dev.[redacted]com:~$ host -t aaaa s3-r-w.eu-west-1.amazonaws.com
s3-r-w.eu-west-1.amazonaws.com has no AAAA record
```

The correct configuration would be either have an AAAA record at the end, or not have a CNAME at all.

sendai

2019 May 30

SRE @ [redacted]