Five nines, or 99.999% uptime, is considered the “Everest” of high availability. In today's IT world, a business continuity strategy must ensure that an organization maintains essential functions during minor failures, or in the event of a disaster. DNS is in the heart of any network, and having a performant DNS is an integral part of achieving five nines. This blog post will discuss how you can incorporate Dnsmasq in your setup to achieve five nines.

Glibc resolver

Glibc resolver interface provides some failover mechanisms by default. If we check resolv.conf manual for failover-related options, three options immediately stands out.

timeout:*n*
Sets the amount of time the resolver will wait for a response from a remote name server before retrying the query via a different name server. This may not be the total time taken by any resolver API call and there is no guarantee that a single resolver API call maps to a single timeout. Measured in seconds, the default is RES_TIMEOUT (currently 5, see <resolv.h>). The value for this option is silently capped to 30.

attempts:*n*
Sets the number of times the resolver will send a query to its name servers before giving up and returning an error to the calling application. The default is RES_DFLRETRY (currently 2, see <resolv.h>). The value for this option is silently capped to 5.

rotate:
Sets RES_ROTATE in _res.options, which causes round-robin selection of name servers from among those listed. This has the effect of spreading the query load among all listed servers, rather than having all clients try the first listed server first every time.

So, if we set up resolv.conf options to something like `options rotate timeout:1 retries:1` , we should have a failover mechanism. Let's put this to the test.

We modify the file to look like this:

```
options rotate timeout:1 retries:1
nameserver 10.10.10.144
nameserver 8.8.8.8
nameserver 9.9.9.9
```

Firstly, we put the rotate option to the test with this script, which tries to resolve redhat.com five times.

```
[root@rhel7 ~]# cat resolv.py

import socket
for x in range(5):
  print socket.getaddrinfo('redhat.com', 80);
```

If we trace that script, we can confirm that DNS queries are rotated among nameservers.

```
[root@rhel7 ~]# strace -e trace=connect python resolv.py 2>&1 | grep 53

connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("10.10.10.144")}, 16) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("8.8.8.8")}, 16) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("9.9.9.9")}, 16) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("9.9.9.9")}, 16) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("10.10.10.144")}, 16) = 0
```

```
[root@rhel7 ~]# for i in {1..5};do (time dig redhat.com) 2>&1|grep real;done
real 0m0.036s
real 0m0.037s
real 0m0.037s
real 0m0.037s
real 0m0.037s
```

Our baseline is an average response of 0.0368s.

And the baseline of our little Python script with multiple queries.

```
[root@rhel7 ~]# for i in {1..5};do (time python resolv.py) 2>&1|grep real;done
real 0m1.056s
real 0m1.055s
real 0m1.045s
real 0m2.049s
real 0m2.056s
```

The baseline of that script is 1.4522s.

Now, if we shut down DNS service on 10.10.10.144 while the IP is still reachable, we can see an impact.

```
[root@rhel7 ~]# for i in {1..5};do (time python resolv.py) 2>&1|grep real;done
real 0m2.151s
real 0m4.123s
real 0m2.516s
real 0m2.109s
real 0m4.209s
```

Obviously, the longer queries are the ones who had 10.10.10.144 involved. Now the average is 3.0216s, that is a 208% increase.

Let's try another scenario, where the nameserver IP itself is unreachable.

```
[root@rhel7 ~]# for i in {1..5};do (time python resolv.py) 2>&1|grep real;done
real 0m5.109s
real 0m5.122s
real 0m5.208s
real 0m4.208s
real 0m6.098s
```

The average is now 5.149s, that is a 354% increase.

There is an obvious problem here; the timeout config option can't take a value lower than one second. Also, this aggressive setting can be problematic in some scenarios. For example, if the first nameserver is slower than the 1-second timeout. We need a more robust solution that provides some high-availability features.

Of course, a network load-balancer can be used to achieve high-availability, but what if we can use a software-based solution that provides high-availability and more?

Using Dnsmasq

Dnsmasq provides a local DNS server, with forwarding and caching capabilities, and we can employ it to handle the scenarios above.

Once we install the Dnsmasq package, we define our upstream nameservers, where DNS queries will be forwarded by Dnsmasq.

```
[root@rhel6 ~]# cat /etc/resolv.dnsmasq
nameserver 10.10.10.144
nameserver 8.8.8.8
nameserver 9.9.9.9
```

And then we configure Dnsmasq:

```
[root@rhel7 ~]# cat /etc/dnsmasq.conf
resolv-file=/etc/resolv.dnsmasq
conf-dir=/etc/dnsmasq.d
interface=lo
bind-interfaces
all-servers
cache-size=0
no-negcache
```

Let us explore the key options added.

- `interface=lo` : Dnsmasq will listen on loopback interface.
- `bind-interfaces` : Turn off listening on other interfaces.
- `all-servers` : That's our key option, it causes Dnsmasq to forward DNS queries to all upstream nameservers simultaneously.
- `cache-size=0` : Turn caching off. We just need Dnsmasq to forward queries.
- `no-negcache` : We don't want to cache negative responses.

Finally, start the Dnsmasq service, and add Dnsmasq on localhost as the first nameserver

```
[root@rhel7 ~]# systemctl restart dnsmasq
[root@rhel7 ~]# cat /etc/resolv.conf

nameserver 127.0.0.1
nameserver 8.8.8.8
nameserver 9.9.9.9
```

Now if we run the same test with one nameserver not reachable.

```
[root@rhel7 ~]# for i in {1..5};do (time python resolv.py) 2>&1|grep real;done

real 0m0.103s
real 0m0.102s
real 0m0.124s
real 0m0.102s
real 0m0.101s
```

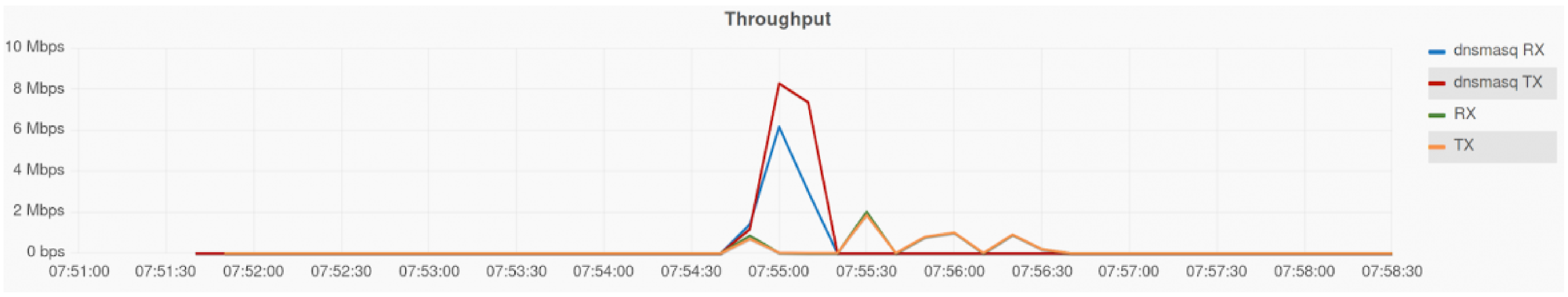
Amazing, We get an average response of 0.1064s, which is lower than our baseline.

Taking a look into Dnsmasq logs with query logging enabled:

```
Jul 21 08:54:49 dnsmasq[5373]: query[A] redhat.com from 127.0.0.1
Jul 21 08:54:49 dnsmasq[5373]: forwarded redhat.com to 10.10.10.144
Jul 21 08:54:49 dnsmasq[5373]: forwarded redhat.com to 8.8.8.8
Jul 21 08:54:49 dnsmasq[5373]: forwarded redhat.com to 9.9.9.9
Jul 21 08:54:49 dnsmasq[5373]: reply redhat.com is 209.132.183.105
```

Indeed, Dnsmasq forwarded our query to all configured nameservers at the same time and provided back the fastest answer it got. That is good for our high availability needs, but what about bandwidth utilization?

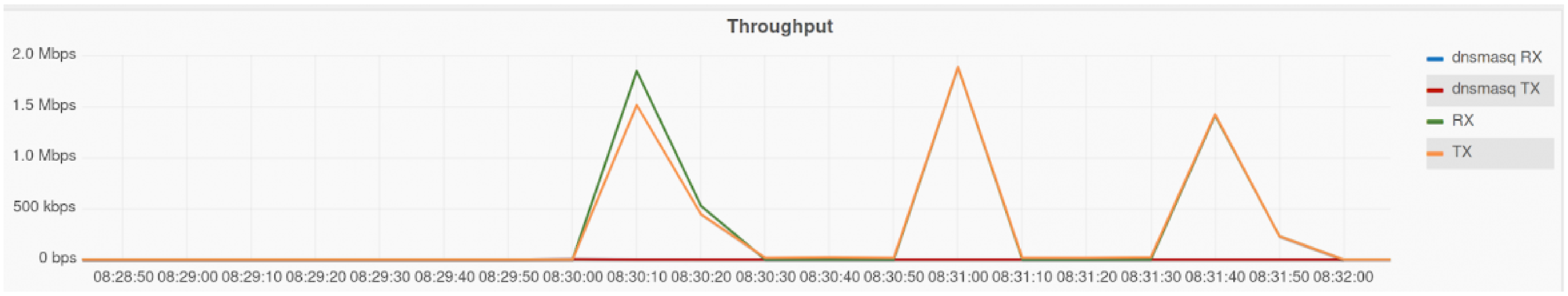
We can put that to the test using dnsperf (https://github.com/cobblau/dnsperf). We will send 100000 DNS queries to a Dnsmasq setup, add an often used setting with one nameserver and compare the results.



Obviously, there is a huge difference; here is the data in a table for a better comparison.

Attribute	Dnsmasq	Traditional DNS Setting
Number of DNS queries	100011	100000
Peak TX	8.27 Mb/s	1.86 Mb/s
Peak RX	6.16 Mb/s	2.02 Mb/s
Bandwidth consumed (TX)	20 MiB	6.6 MiB
Bandwidth consumed (RX)	12.6 MiB	6.9 MiB
Queries Per Second	4875	1074
Elapsed time	20 seconds	93 seconds

Bandwidth consumption has more than doubled. If bandwidth utilization is of a concern to you, you might consider using dnsmasq caching, by setting cache-size to a non-zero value. Caching will also improve response speed. If we rerun the same test above with caching on, the graph will be completely different.



To put that into numbers:

Attribute	Dnsmasq	Traditional DNS Setting
Number of DNS queries	100037	100000
Peak TX	208 b/s	1.88 Mb/s
Peak RX	543 b/s	1.84 Mb/s
Bandwidth consumed (TX)	5.7 KiB	6.6 MiB
Bandwidth consumed (RX)	25.8 KiB	7.0 MiB
Queries Per Second	12486	1016
Elapsed time	8 seconds	98 seconds

And, if we check Dnsmasq with caching ON against our earlier baseline test.

```
[root@rhel7 ~]# for i in {1..5};do (time dig redhat.com) 2>&1|grep real;done real 0m0.008s
real 0m0.008s
real 0m0.008s
real 0m0.008s
real 0m0.008s
```

It doesn't get much faster than that.

Summary and Closing

In this blog-post, we explored how dnsmasq can be integrated into your setup to provide a highly-available and fault tolerant DNS service. We also went through specific performance gains and bandwidth utilization.

Dnsmasq is a versatile tool with a multitude of configuration options, that can be leveraged to provide modern cloud needs. Give this setup a try, and let your TAM know how it performed.