My first Nagios plugin in Go

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

First publication of this article on 4 December 2012

https://www.bortzmeyer.org/go-dns-icinga.html

An important feature of the Nagios monitoring system (or of its compatibles like the one I use, Icinga), is that you can easily write your own plugins, to test things that cannot be tested with the "standard" Nagios plugins <http://nagiosplugins.org/>. I wrote several such plugins in my life, to do tests which were useful to me. This one is the first I wrote in Go, and it tests a DNS zone by querying every name server, thus checking that **all** name servers are OK.

This is an important test because DNS is too robust for monitoring : even with almost all the name servers down or broken, the zone will work, thus leaving people with a feeling of false security. Hence the idea of testing every name server of the zone (monitoring typically requires that you test differently from what the ordinary user does, to see potential problems). The original idea comes from the `check_soa` program, distributed in the famous book "DNS and BIND <http://shop.oreilly.com/product/9780596100575.do>" (the code is available on line <http://examples.oreilly.com/9780596100575/>).

And why using Go ? Well, Nagios plugins can be written in any language but I like Go : faster and lighter than Perl or Python but much easier to deal with than C. A good language for system and network programming.

Nagios plugins have to abide by a set of rules <http://nagiosplug.sourceforge.net/developer-guideline html> (in theory : in practice, many "quick & dirty" plugins skip many of the rules). Basically, the testing plugin receives standard arguments telling it what to test and with which parameters, and it spits out one line summarizing the results and, more important, exits with a status code indicating what happened (OK, Warning, Critical, Unknown).

That's one point where my plugin fails : the standard Go library for parsing arguments (flag <http://golang.org/pkg/flag/>) cannot do exactly what Nagios wants. For instance, Nagios wants repeating options (`-v` increases the verbosity of the output each time it appears) and they are not possible with the standard module. So, in practice, my plugin does not conform to the standard. Here is its use :

```
% /usr/local/share/nagios/libexec/check_dns_soa  -h
CHECK_DNS_SOA UNKNOWN: Help requested
  -4=false: Use IPv4 only
  -6=false: Use IPv6 only
  -H="": DNS zone name
  -V=false: Displays the version number of the plugin
  -c=1: Number of name servers broken to trigger a Critical situation
  -i=3: Maximum number of tests per nameserver
  -r=false: When using -4 or -6, requires that all servers have an address of this family
  -t=3: Timeout (in seconds)
  -v=0: Verbosity (from 0 to 3)
  -w=1: Number of name servers broken to trigger a Warning
```

The most important argument is the standard Nagios one, `-H`, indicating the zone you want to test. For instance, the TLD `.tf` :

```
% /usr/local/share/nagios/libexec/check_dns_soa  -H tf
CHECK_DNS_SOA OK: Zone tf. is fine

% echo $?
0
```

The return code show that everything was OK. You can test other, non-TLD, domains :

```
% /usr/local/share/nagios/libexec/check_dns_soa  -H github.com
CHECK_DNS_SOA OK: Zone github.com. is fine
```

The plugin works by sending a SOA request to every name server of the zone and checking there is an answer, and it is authoritative. With tcpdump, you can see its activity (`192.168.2.254` is the local DNS resolver) :

```
# Request the list of name servers
12:00:39.954330 IP 192.168.2.4.58524 > 192.168.2.254.53: 0+ NS? github.com. (28)
12:00:39.979266 IP 192.168.2.254.53 > 192.168.2.4.58524: 0 4/0/0 NS ns2.p16.dynect.net., NS ns3.p16.dynect.

# Requests the IP addresses
12:00:40.063714 IP 192.168.2.4.37993 > 192.168.2.254.53: 0+ A? ns3.p16.dynect.net. (36)
12:00:40.064237 IP 192.168.2.254.53 > 192.168.2.4.37993: 0 1/0/0 A 208.78.71.16 (52)
12:00:40.067517 IP 192.168.2.4.47375 > 192.168.2.254.53: 0+ AAAA? ns3.p16.dynect.net. (36)
12:00:40.068085 IP 192.168.2.254.53 > 192.168.2.4.47375: 0 1/0/0 AAAA 2001:500:94:1::16 (64)

# Queries the name servers
12:00:40.609507 IP 192.168.2.4.51344 > 208.78.71.16.53: 0 SOA? github.com. (28)
12:00:40.641900 IP 208.78.71.16.53 > 192.168.2.4.51344: 0*- 1/4/0 SOA (161)
12:00:40.648342 IP6 2a01:e35:8bd9:8bb0:ba27:ebff:feba:9094.33948 > 2001:500:94:1::16.53: 0 SOA? github.com.
12:00:40.689689 IP6 2001:500:94:1::16.53 > 2a01:e35:8bd9:8bb0:ba27:ebff:feba:9094.33948: 0*- 1/4/0 SOA (161)
```

Then, you can configure Nagios (or Icinga, in my case), to use it :

```
define command {
        command_name    check_dns_soa
        command_line    /usr/local/share/nagios/libexec/check_dns_soa -H $HOSTADDRESS$ $ARG1$ $ARG2$
        }

define hostgroup
```

```
       hostgroup_name my-zones
       members example.com,example.org,example.net
}

define service {
       use generic-service
       hostgroup_name my-zones
       service_description CHECK_DNS_SOA
       check_command check_dns_soa
}
```

Nagios (or Icinga) forces you to declare a host, even if it makes no sense in this case. So, my zones are declared as a special kind of hosts :

```
define host{
        name                            dns-zone
        use                       generic-host
check_command             check-always-up
        register 0
        # Other options, quite ordinary
}

define host{
        use dns-zone
        host_name example.com
}
```

Nagios wants a command to check that the "host" is up (otherwise, the "host" stays in the "pending" state for ever). That's why we provide `check-always-up` which is defined as always true :

```
define command{
        command_name    check-always-up
        command_line    /bin/true
}
```

With this configuration, it works fine and you can see nice trends of 100 % uptime if everything is properly configured and connected. And if it's not? Then, the plugin will report the zone as Warning or Critical (depending on the options you used, see `-c` and `-w`) :

```
% /usr/local/share/nagios/libexec/check_dns_soa  -H ml
CHECK_DNS_SOA CRITICAL: Cannot get SOA from ciwara.sotelma.ml./217.64.97.50: read udp 192.168.2.4:40836: i/o tim

% echo $?
2
```

If you are more lenient and accept two broken name servers before setting the status as Critical, use `-c 3` and you'll just get a Warning :

```
% /usr/local/share/nagios/libexec/check_dns_soa  -v 3 -c 3 -H ml
CHECK_DNS_SOA WARNING: Cannot get SOA from ciwara.sotelma.ml./217.64.97.50: read udp 192.168.2.4:35810: i/o time
217.64.100.112 (SERVFAIL)

% echo $?
1
```

This plugin is hosted at FramaGit <https://framagit.org/bortzmeyer/check_dns_soa>, where you can retrieve the code, report bugs, etc. To compile it, see the README. You'll need the excellent Go DNS <http://miek.nl/projects/godns/> library first (a good way to make DNS requests from Go).

Two implementation notes : the plugin does not use one of the Go's greatest strengths, its built-in parallelism, thanks to "goroutines". This is because making it parallel would certainly decrease the wall-clock time to run the tests but, for an unattended program like Nagios, it is not so important (it would be different for an interactive program). And, second, there is a library to write Nagios plugins in Go <https://github.com/laziac/go-nagios> but I did not use it because of several limitations (and I do not use performance data, where this library could be handy). Note also that other people are using Go with Nagios in a different way <http://shortbus.org/bloggin/tag/golang/>.

Thanks to Miek Gieben for his reading and patching.