Developing a dnstap to C-DNS converter at the IETF hackathon

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

First publication of this article on 17 July 2017

https://www.bortzmeyer.org/c-dns-tests.html

The weekend of 15-16 july 2017, I participated to the IETF 99 hackathon in Prague. The project was to develop a dnstap <http://dnstap.info/> to C-DNS converter. This is a small documentation of the result and of the lessons learned.

First, a bit of background. Most DNS operators these days gather a lot of data from the DNS traffic and then analyze it. Events like DITL <https://www.caida.org/projects/ditl/> can fill many hard disks with pcap files. pcap not being very efficient (both in producing it, in storing it, and in analyzing it), a work is under way at the IETF to create a better format : C-DNS (for "capture DNS"). C-DNS was specified at this time in an Internet-Draft, draft-ietf-dnsop-dns-capture-format (it is now RFC 8618[1]). It was only a draft, not yet a RFC. One of the goals of IETF hackathons is precisely to test drafts, to see if they are reasonable, implementable, etc, before they are approved.

Note that C-DNS is based on CBOR (RFC 7049). There is already an implementation of C-DNS, available under a free software licence <https://github.com/dns-stats/compactor>. Here, the idea was to write a second implementation, to test interoperability. The target was a C-DNS producer. Where to find the data to put in the file ? I choosed dnstap <http://dnstap.info/> since it is currently the best way to get data from a DNS server. dnstap relies on protocol buffers so it may be difficult to handle but there is already a good dnstap client <https://github.com/dns-stats/compactor>, written in Go.

So, the decisions were :

— Get DNS data via dnstap from a Unbound resolver (the only resolver with dnstap support at this time),
— Use the dnstap client <https://github.com/dns-stats/compactor> as a starting point (which means the hackathon project was to use Go),

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8618.txt

— Test the C-DNS produced with the inspector from the first implementation <https://github.com/dns-stats/compactor>.

First, compiling the first C-DNS implementation, which is in C++ (I used a not-up-to-date Ubuntu machine) : it requires several Boost libraries, and I'm too lazy to check exactly which, so I installed everything `aptitude install libboost1.58-all-dev liblzma-dev lzma-dev lzma`. The C++ capture library libtins <https://libtins.github.io/> does not appear to be in a Ubuntu "xenial" package (but there is a Debian one and it is in some Ubuntu versions). The "trusty" package is too old, we install the "yakkety" one manually :

```
./configure
make
```

Then, we can produce C-DNS files from pcaps :

```
./compactor/compactor -o short.cdns -c /dev/null short.pcap
```

And read C-DNS files :

```
./compactor/inspector short.cdns
```

So, now, we can test the produced files.

Next, compiling Unbound with dnstap support (it is not done by default, probably because it add a lot of dependencies, coming from protocol buffers) :

```
aptitude install libprotobuf-c-dev  protobuf-c-compiler libfstrm-dev fstrm-bin
./configure --enable-dnstap
make
```

We configure Unbound with :

```
dnstap:
    dnstap-enable: yes
    dnstap-socket-path: "/var/run/unbound/dnstap.sock" # "the dnstap socket won't appear in the filesystem u
    dnstap-send-identity: yes
    dnstap-send-version: yes
    dnstap-log-resolver-response-messages: yes
    dnstap-log-client-query-messages: yes
    dnstap-log-resolver-query-messages: yes
    dnstap-log-client-response-messages: yes
```

And we start it :

```
./unbound/unbound -d -c dnstap.conf
```

Then, when processing DNS queries and responses, Unbound sends a report to the dnstap socket. We just need a dnstap client to display it.

We forked the dnstap client, created a `c-dns` branch, and started our copy <https://github.com/bortzmeyer/golang-dnstap>. To compile it :

```
aptitude install golang-dns-dev golang-goprotobuf-dev
go get github.com/farsightsec/golang-framestream
go install github.com/bortzmeyer/golang-dnstap
go build dnstap/main.go
```

Then we can run it :

```
./main -y -u /var/run/unbound/dnstap.sock
```

And we can see the YAML output when Unbound receives or sends DNS messages. To store this output, I was not able to put the data in a file <https://github.com/dnstap/golang-dnstap/issues/11>, so I just redirected the standard output.

Most of the hackathon work took place in the new file `CdnsFormat.go`. First, producing CBOR. I had a look at the various Go implementations <http://cbor.io/impls.html> of CBOR. Most seem old and unmaintained and CBOR seemed to me simple enough that it was faster to reimplement from scratch. This is the code at the beginning of `CdnsFormat.go`, the `cborInteger()` and similar functions. I also developed a small Go program (en ligne sur `https://www.bortzmeyer.org/files/read-cbor.go`) to read CBOR files and display them as a tree, a tool which was very useful to debug my C-DNS producer. The grammar of C-DNS is specified in the CDDL language (currently specified in another Internet-Draft <https://datatracker.ietf.org/doc/draft-greevenbosch-appsawg-cbor-cddl/>). A validating tool <https://rubygems.org/gems/cddl> exists, taking CDDL and CBOR and telling if the CBOR file is correct, but this tool's error messages are awful. It is just good to say if the file is OK or not, afterwards, you have to examine the file manually.

Then, actually producing the C-DNS file. The C-DNS format is not simple : it is intended for performance, and harsh compression, not for ease of implementation. The basic idea is to capture in tables most of what is repeated, actual DNS messages being made mostly of references to these tables. For instance, a DNS query for `www.sinodun.com` won't include the string `www.sinodun.com`. This FQDN will be stored once, in the `name-rdata` table, and a numeric index to an entry of this table will be used in the message. For instance :

```
// Server address
s.Write(cborInteger(0))
s.Write(cborInteger(2))
```

Here, we don't write the server IP address directly, we write an index to the table where the IP address is (2 is the second value of the table, C-DNS indexes start at 1, read later about the value zero). Note that the inspector crashed when meeting indexes going outside of an array, something that Jim Hague fixed during the hackathon.

Speaking of IP addresses, "client" and "server" in the C-DNS specification don't mean "source" and "destination". For a DNS query, "client" is the source and "server" the destination but, for a DNS response, it is the opposite (dnstap works the same way, easing the conversion).

Among the things that are not clear in the current version of the draft (version -03) is the fact that CBOR maps are described with keys that are strings, while in the CDNS format, they are actually integers. So when you read in the draft :

```
FilePreamble = {
      major-format-version => uint,
      minor-format-version => uint,
      ...
```

Read on : `FilePreamble` is indeed a CBOR map but the actual keys are further away :

```
major-format-version = 0
minor-format-version = 1
```

So for a version number of 0.5 (the current one), you must write in Go the map as :

```
//    Major version
s.Write(cborInteger(0))
s.Write(cborInteger(0))
//    Minor version
s.Write(cborInteger(1))
s.Write(cborInteger(5))
```

Another choice to make : CBOR allow arrays and maps of indefinite of finite length. C-DNS does not specify which one to use, it is up to you, programmer, to choose. Of course, it is often the case that you don't know the length in advance, so sometimes you have no choice. An example of a finite length array :

```
s.Write(cborArray(3))
// Then write the three items
```

And an example of an indefinite length array :

```
s.Write(cborArrayIndef())
// Write the items
s.Write(cborBreak())
```

Note that you need to checkpoint from time to time (for instance by rotating the C-DNS file and creating a new one) otherwise a crash will leave you with a corrupted file (no break at the end). The fact that C-DNS (well, CBOR, actually), requires this break code at the end forced us to modify the API of dnstap modules, to have a new function `finish` (of type `TextFinishFunc`, this function is a no-op for the other formats).

Note there is a semantic mismatch between C-DNS and dnstap : C-DNS assumes all the data is mandatory while in dnstap, almost everything is optional. That means that we have to create dummy data in some places for instance :

```
s.Write(cborArray(2))
if m.QueryTimeSec != nil {
        s.Write(cborInteger(int(*m.QueryTimeSec)))
} else {
        s.Write(cborInteger(0))
}
```

Here, if we don't have the time of the DNS message, we write the dummy value 0 (because `earliest-time` is not optional in the C-DNS format). One possible evolution of the C-DNS draft could be to have profiles of C-DNS, for instance a "Full" profile where everything is mandatory and a "Partial" one where missing values are allowed.

The C-DNS specification is currently ambiguous about empty arrays. Most arrays are declared, in the grammar, as allowing to be empty. It raises some issues with the current tools (the inspector crashed when meeting these empty arrays, something that Jim Hague fixed during the hackathon). But it is also unclear in its semantics. For instance, the draft says that a value zero for an index mean "not found" but it does not appear to be supported by the current tools, forcing us to invent dummy data.

dnstap formats a part of the DNS messages but not everything. To get the rest, you need to parse the binary blob sent by dnstap. We use the excellent Go DNS package <https://miek.nl/2014/August/16/go-dns-package/>):

```
import (
  ...
  "github.com/miekg/dns"
...
msg = new(dns.Msg)
err := msg.Unpack(m.QueryMessage)
```

This allows us to get information like the QNAME (name used in the question) :

```
qname := make([]byte, 256)
...
n, err = dns.PackDomainName(msg.Question[0].Name, qname, 0, nil, false)
...
s.Write(cborByteString(qname[0:n]))
```

(`Unpack` leaves us with a domain name in the presentation - text - format, but C-DNS requires the label - binary - format, which we do with `PackDomainName`.) Domain names are defined as CBOR byte string and not strings 1) because you cannot be sure of their encoding (strings have to be UTF-8) 2) and for consistency reasons because the same tables can store other things than domain names.

OK, now, we have everything, compile again and run (there is a Unbound running in another window) :

```
go build dnstap/main.go
./main -c -u /var/run/unbound/dnstap.sock > test.cdns
../compactor/inspector   test.cdns
```

And the inspector produced a pcap file, as well as a report. Let's try the pcap :

```
% tcpdump -n -r test.cdns.pcap
reading from file test.cdns.pcap, link-type EN10MB (Ethernet)
13:02:02.000000 IP6 ::1.51834 > ::.0: UDP, length 25
13:02:02.000000 IP 0.0.0.0.0 > 193.0.14.129.53: 666 SOA? 142.com. (25)
13:02:02.000000 IP 193.0.14.129.53 > 0.0.0.0.0: 666- [0q] 0/0/0 (12)
13:02:02.000000 IP6 ::.0 > 2001:503:83eb::30.53: 666 AAAA? ns2.bodis.com. (31)
13:02:02.000000 IP 0.0.0.0.0 > 192.41.162.30.53: 666 AAAA? ns1.bodis.com. (31)
13:02:02.000000 IP6 2001:503:83eb::30.53 > ::.0: 666- [0q] 0/0/0 (12)
```

You immediately note the missing data. Most of the times, it was the lack of this data in dnstap (the IP addresses, for instance), sometimes it was my own lazyness (the microseconds in the time).

Sometimes, there are things that are available in dnstap but C-DNS offers no way to store them. This is the case with the bailiwick (the zone from which the DNS responses came) or with the fact thata the resolver cache experienced a hit or a miss.

This was the end of the IETF 99 hackathon. The code is available <https://github.com/bortzmeyer/golang-dnstap>. An example C-DNS file produced by my converter is attached here (en ligne sur https://www.bortzmeyer.org/files/test.cdns). Other things would of course be possible :

— The biggest limit of the current dnstap-to-C-DNS tool is that it uses one C-DNS block per DNS message. This is simpler for the lazy programmer, but it defeats the most important requirments of C-DNS : compression. Compression is done py putting in tables information which is common to several DNS messages. This is the obvious next step.
— Test with an authoritative name server like Knot (which has dnstap support).

Thanks to Jim Hague for working with me throughout the hackathon (and fixing many things, and answering many questions) and to Sara Dickinson.