eighty-twenty

File distribution over DNS: (ab)using DNS as a CDN

Mon 31 Jul 2023 18:32 CEST

Tony Garnock-Jones
#distsys #tech #hack
permalink

This is the story of a one-afternoon hack that turned into a one-weekend hack.

I woke up on Saturday with a silly idea: what would it be like to use content-defined chunking (CDC) and serve the chunks over DNS? DNS caching could make scaleout and incremental updates quite efficient, at least in theory. Strong hashing gives robust download integrity. DNS replication gives you a kind of high availability, even!

After a coffee, I figured I may as well try it out.

TL;DR. It works, more or less, so long as your resolver properly upgrades to DNS-over-TCP when it gets a truncated UDP response. The immutable, strongly-named chunks are served in TXT records (!) and are cached by resolvers for a long time. This lowers load on the authoritative DNS server. Each stored "file" gets a user-friendly name for its root chunk in the form of a CNAME with a much shorter TTL.

You can try it out with:

```
    docker run -i --rm leastfixedpoint/nscdn \
        nscdn get SEKIENAKASHITA.DEMO.NSCDN.ORG. > SekienAkashita.jpg
```

which downloads this image using nothing but DNS.

A demo server is running on the domain `demo.nscdn.org` and the code is available at https://gitlab.com/tonyg/nscdn.

How it works

The Ronomon variant of FastCDC, due to Joran Greef, is a JavaScript-friendly, 32-bit content-defined chunking algorithm that splits big files into chunks with a distribution of sizes having an average, minimum and maximum length.[1]

For this project I chose an 8k minimum size, a 16k average, and a 48k upper limit, because of the limitations involved in serving large amounts of data via DNS.

The core idea is to use the CDC algorithm to slice up a data file, and then construct a broad, shallow Merkle tree from it, serving each leaf and inner node of the tree as a separate DNS TXT record associated with a domain label including the Base32 encoding of the BLAKE2b hash of the data.

The example file from above, `SekienAkashita.jpg`, is split up into five chunks and one inner node that lists the chunks:

```
SEKIENAKASHITA.DEMO.NSCDN.ORG (CNAME)
 └─□ 2-NPNQZ4OQGZCXTLGFT6SGXO5TFLVZWMCFOXAV6GEYNDPYL67QSMPQ.DEMO.NSCDN.ORG (TXT, 320 bytes)
      ├─□ 1-3NZRAZ4RTLWEFFSE27EBEVNKFOM6SFI4ISQEH7NGNPCG2CS4SIUA.··· (TXT, 22366 bytes)
      ├─□ 1-ZYNVPQYLBJYT7EOWRUDNY6XSURS3KWOMTQUODKUCCWXXQLKZFF3Q.··· (TXT, 8282 bytes)
      ├─□ 1-VPJYS4PUGX275YDFEKZF6BC3SXYZDMNTANV5LXMDWMB2PDSKWJPQ.··· (TXT, 16303 bytes)
      ├─□ 1-IBGDKGR2IXRIISKASJIP5CLPLHCUSGE5V6SVRWKHFHJSIAZVXOHQ.··· (TXT, 18696 bytes)
      └─□ 1-QT7EHDMMEVKJF77MOL4T4PXU3FGSCBXRVNFMYJK4NOQ4BJ6I7YCA.··· (TXT, 43819 bytes)
```

For a larger file—say, the Linux kernel—the index node itself would be longer than permitted for a chunk, so it would be split up itself and another level would be added to the tree to index the chunks of the lower-level node. This recursion can be repeated as required. I chose a 64-bit file size limit.

Storing a tree in DNS

A CNAME record from the human-usable name of each file (`SEKIENAKASHITA.DEMO.NSCDN.ORG`) points to the DNS record of the root node of the file's Merkle tree.

Each tree node is stored as raw binary (!) in a DNS TXT record. Each record's DNS label is formed from the type code of the node and the base32 encoding of the BLAKE2b hash of the binary content.

The content of leaf nodes (type 1) is just the binary data associated with the chunk. Each inner node (type 2) is a sequence of 64-byte "pointer" records containing a binary form of child nodes' DNS labels, along with a length for each child. This allows random-access retrieval of subranges of each file.

Weirdly, this is all completely standards-compliant use of DNS. The only place I'm pushing the limits is the large-ish TXT records, effectively mandating use of DNS's fallback TCP mode.

Serving the data: stick it in SQLite, tiny server, job done

Any ordinary DNS server can be used to serve the records from a domain under one's control.

However, I decided to hack together my own little one that could serve records straight out of a SQLite database. I used it as an excuse to experiment with Golang again for the first time in more than a decade.[2]

I cobbled together a program called `nscdnd` which uses

- github.com/miekg/dns as a DNS codec and server,
- github.com/mattn/go-sqlite3 to access the SQLite database, and
- github.com/sirupsen/logrus for logging.

Similarly, a little tool called `nscdn` allows insertion (`nscdn add`) and deletion (`nscdn del`) of files from a database, plus retrieval and reassembly of a file from a given domain name (`nscdn get`).

The bulk of the interesting code is in

- leastfixedpoint.com/nscdn/pkg/hashtree (Merkle tree),
- leastfixedpoint.com/nscdn/pkg/chunking/ronomon, (CDC chunking), and
- leastfixedpoint.com/nscdn/pkg/nscdn, (retrieval, integrity-verification, and reassembly of records from DNS).

Okay, but is this a good idea?

¯_(ツ)_/¯ It works surprisingly well in the limited testing I've done. I doubt it's the most efficient way to transfer files, but it's not wildly unreasonable. The idea of getting the chunks cached by caching resolvers between clients and the authoritative server seems to work well: when I re-download something, it only hits the authoritative server for the short-lived CNAME and the root-node TXT record. The other nodes in the tree seem to be cached somewhat locally to my client.

Try it out yourself!

You can retrieve files from the demo server on `demo.nscdn.org`, as previously mentioned:

```
    docker run -i --rm leastfixedpoint/nscdn \
        nscdn get SEKIENAKASHITA.DEMO.NSCDN.ORG. > SekienAkashita.jpg
```

You can also run an `nscdnd` instance for a domain you control. All the following examples use docker, but you can just check out the repository and build it yourself too.

To run a server:

```
    docker run -d -p 53:53 -p 53:53/udp \
        -v `pwd`/store.sqlite3:/data/store.sqlite3 \
        --env NSCDN_ROOT=your.domain.example.com \
        --name nscdnd \
        leastfixedpoint/nscdn
```

and add files to the store:

```
    docker run -i --rm \
        -v `pwd`/store.sqlite3:/data/store.sqlite3 \
        leastfixedpoint/nscdn \
        nscdn add /data/store.sqlite3 SOMEFILENAME  < SomeFilename.bin
```

Then add an `NS` record pointing to it:

```
your.domain.example.com. 86400 IN NS your.nscdnd.server.example.com.
```

and retrieve your files:

```
    docker run -i --rm leastfixedpoint/nscdn \
        nscdn get SOMEFILENAME.your.domain.example.com
```

You can also `dig +short -t txt your.domain.example.com` to get information about the running server.

For the `demo.nscdn.org` server, it looks like this:

```
$ dig +short -t txt demo.nscdn.org
"server=nscdnd" "version=v0.3.1" "SPDX-License-Identifier=AGPL-3.0-or-later" "source=https://gitlab.com/tonyg/nscdn"
```

Finally, you use `dig` to retrieve CNAME and TXT records without using the `nscdn` tool:

```
$ dig +short -t any SEKIENAKASHITA.DEMO.NSCDN.ORG
2-NPNQZ4OQGZCXTLGFT6SGXO5TFLVZWMCFOXAV6GEYNDPYL67QSMPQ.DEMO.NSCDN.ORG.
"\000\000\000\000\000\000\000\001\000\000\000\000\000\000W^\000\000\000...
```

Future directions

Compression of individual chunks. At the moment, chunks are served uncompressed. It'd be a friendly thing to do to compress the contents of each TXT record.

Experimenting with chunk sizes. Is the distribution of chunk sizes with the current parameters reasonable? Are smaller chunks required, operationally, given we're kind of pushing DNS to its limits here? Could larger chunk sizes work?

How does it perform? How could performance be improved?

Garbage-collection of chunks in a store. At present, running `nscdn del` just removes the CNAME link to a root chunk. It doesn't traverse the graph to remove unreferenced chunks from the store.

Incremental downloads, partial downloads, recovery/repair of files.

Statistics on sharing of chunks in a store. Say you used a store to distribute multiple releases of a piece of software. It'd be interesting to know how much sharing of chunks exists among the different release files.

Download and assemble files in the browser? So DNS-over-HTTPS is a thing. What happens if we use, for example, the Cloudflare DoH API to retrieve our chunks?

```
JSON.stringify(await (await fetch(
    "https://cloudflare-dns.com/dns-query?name=demo.nscdn.org&type=TXT",
    { headers: { "accept": "application/dns-json" } })).json(), null, 2)
→ {
      "Status": 0,
      "TC": false, "RD": true, "RA": true, "AD": false, "CD": false,
      "Question": [{
          "name": "demo.nscdn.org",
          "type": 16
        }],
      "Answer": [{
          "name": "demo.nscdn.org",
          "type": 16,
          "TTL": 61,
          "data": "\\"server=nscdnd\\"\\"version=v0.3.1\\"\\"SPDX-License-Identifier=AGPL-3.0-or-later\\"\\"source=https://
        }]
}
```

Hmm. Promising. Unfortunately, it doesn't seem to like the TXT records having binary data in them (bug? certainly TXT records are allowed to hold binary data, see here and then here), so it might not Just Work. Perhaps a little redesign to use Base64 in the TXT record bodies is required.

Encryption of stored files. Perhaps something simple like age.

Insertion of data across the network. While Dynamic DNS is a thing, it's not quite suitable for this purpose. Perhaps some way of inserting or deleting files other than the current ssh-to-the-server-and-run-a-command would be nice.

Think more about Named Data Networking. Remember Named Data Networking (NDN)? It's an alternative Internet architecture, initially kicked off by Van Jacobson and colleagues. (It used to be known as CCN, Content-Centric Networking.) This DNS hack is a cheap-and-nasty system that has quite a bit in common with the design ideas of NDN, though obviously it's desperately unsophisticated by comparison.

The End

Anyway, I had a lot of fun hacking this together and relearning Go, though I was a little embarrassed when I found myself spending a lot of time at the beginning browsing for a domain to buy to show it off…

1. The distribution of chunk sizes is pretty strange though. I don't know in what sense the "average" actually *is* an average; the minimum and maximum cut-offs are enforced by the algorithm though. For more detail, see the FastCDC paper. ↩

2. So I have, uh, *criticisms* of go. But it's nothing that hasn't been said before. Let's just say that the experience reminded me of programming JavaScript in the bad old days before TC39 really got going. Overall the experience was "ok, I guess". ↩