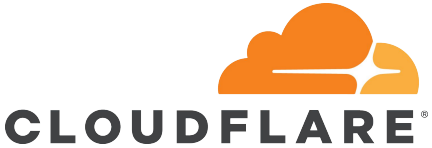


The road to the Ultimate Stub-resolver

Olafur Gudmundsson & Pavel Odintsov

Short history of stub resolvers?

It came to live as _res as part of BSD 4.1 OS, it was a simple lookup mechanism that could look up records. Later “higher” level calls like getaddr() came. Stub resolvers traditionally come as part of “standard” library of the operating system.

Capabilities: almost none

- Looks up recursor from configuration
- Sets RD bit on query
- Waits for answer
- may retry

Assumptions: One of everything

★ network connection
★ namespace
★ Address
★ Location

```
extern struct __res_state _res;

      int res_init(void);

      int res_query(const char
*dname, int class, int type,
                  unsigned char
*answer, int anslen);

      int res_search(const char
*dname, int class, int type,
                  unsigned char
*answer, int anslen);

      int res_querydomain(const char
*name, const char *domain,
                  int class, int type,
unsigned char *answer,
                  int anslen);

      int res_mkquery(int op, const
char *dname, int class,
                  int type, const
unsigned char *data, int datalen,
                  const unsigned char
*newrr,
                  unsigned char *buf,
int buflen);

      int res_send(const unsigned
char *msg, int msglen,
                  unsigned char
*answer, int anslen);
```

CLOUDFLARE

Over the years: better interface

Slightly higher level calls (POSIX)

- One RRset at the time
- Limited number types supported

Object oriented interfaces

- Generally did not support unknown types

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const
char *name);

#include <sys/socket.h>       /* for
AF_INET */
struct hostent *gethostbyaddr(const
void *addr,socklen_t len, int type);

void sethostent(int stayopen);
void endhostent(void);

void herror(const char *s);

const char *hstrerror(int err);

       /* System V/POSIX extension */
struct hostent *gethostent(void);

       /* GNU extensions */
struct hostent *gethostbyname2(const
char *name, int af);
int gethostent_r( struct hostent *ret,
char *buf, size_t buflen, struct
hostent **result, int *h_errnop);

int gethostbyaddr_r(const void *addr,
socklen_t len, int type,
struct hostent *ret, char *buf, size_t
buflen, struct hostent **result, int
*h_errnop);

int gethostbyname_r(const char *name,
struct hostent *ret, char *buf, size_t
buflen, struct hostent **result, int
*h_errnop);

int gethostbyname2_r(const char *name,
int af, struct hostent *ret, char
*buf, size_t buflen,struct hostent
**result, int *h_errnop);
```

CLOUDFLARE

Around 2000
Language specific libraries

- DNSjava, DNSpython
- Got more functionality
- Interactive operations

⇒ did not influence standard OS libraries

Explosion in DNS “servers”
BIND-9, Nominum, MS, DNSmasq ……..

Few new resolvers

CLOUDFLARE

20xx more libraries: server and application building blocks

ldns

Libunbound

getDNSapi

Miekg Go

.....

CLOUDFLARE

What is the problem?

Almost nothing

No caching of answers

No memory of upstreams

- Repeated queries
- First address gets all questions
  - Blocking

Lots of stupid apps send queries to a.root-servers.net

CLOUDFLARE

DNS stub crapware

Built on top

- of bad API’s
  - No support for “modern” types only A, MX, AAAA, NS, SOA, TXT
  - SRV usage is not feasible

Living in the past i.e. hostages of old mistakes

Makes live horrible for applications!!!

CLOUDFLARE

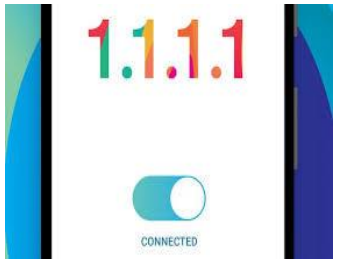
AAAA rollout to OS’s is almost complete
→ type defined in 1995

SPF failed due to some mail servers not being able look up SPF type

Not all devices equal

Mobility

Setup Complexity



CLOUDFLARE

Resolving at speed

Problem: Make services faster

Cloudflare CDN is reverse proxy:

Web Caching, DNS, SSL, optimizations, DoS protection, WAF, ...

Eyeballs → CF metals

CF metals → Customer Origins

React fast to changes:
Max TTL applied

Recover fast from failures:
Short TTL used

Dyn Attack was a disaster for shared customers

CLOUDFLARE

Problem: Make services faster

Cloudflare DNS metals resolution:

V0: 8.8.8.8

v1 : PowerDNS resolver on each metal

V2: Unbound on each metal

V3: Tiered Unbound

V4: <wait>

V0 easy and simple but slow

V1 worked for a while

V2: worked well for small sites; scaling issues

V3: Scaled and ,uch faster

V4: more reliable, scaleable and faster

CLOUDFLARE

Recursive resolver on each host

Problems:

- Very poor cache locality
- prefetch “unused”
- Complex maintenance (all machines resolve all domains)
  - Inconsistent behaviors
  - Hard to debug customer problems

Some TLD’s are bad in certain geographical area’s

ccTLD’s NS’s “far” from sites using them.

Not everyone is using Anycast

Routing is strange

Network providers are sometimes unreliable

Rate Limits triggered

CLOUDFLARE

General DNS issues to overcome

Frequently in “discovery” mode

- Infrequent queries to Authorities with large NS sets ⇒ no RTT history established

Hostage of slow Authority or “parental” domains

- Question of “safety” vs “fast”

Scatter of queries result of Kaminski bug defenses

Does resolver have multiple addresses to use as query addresses ?

Until recently no user choice between operating modes

Tiered Unbound Recursor

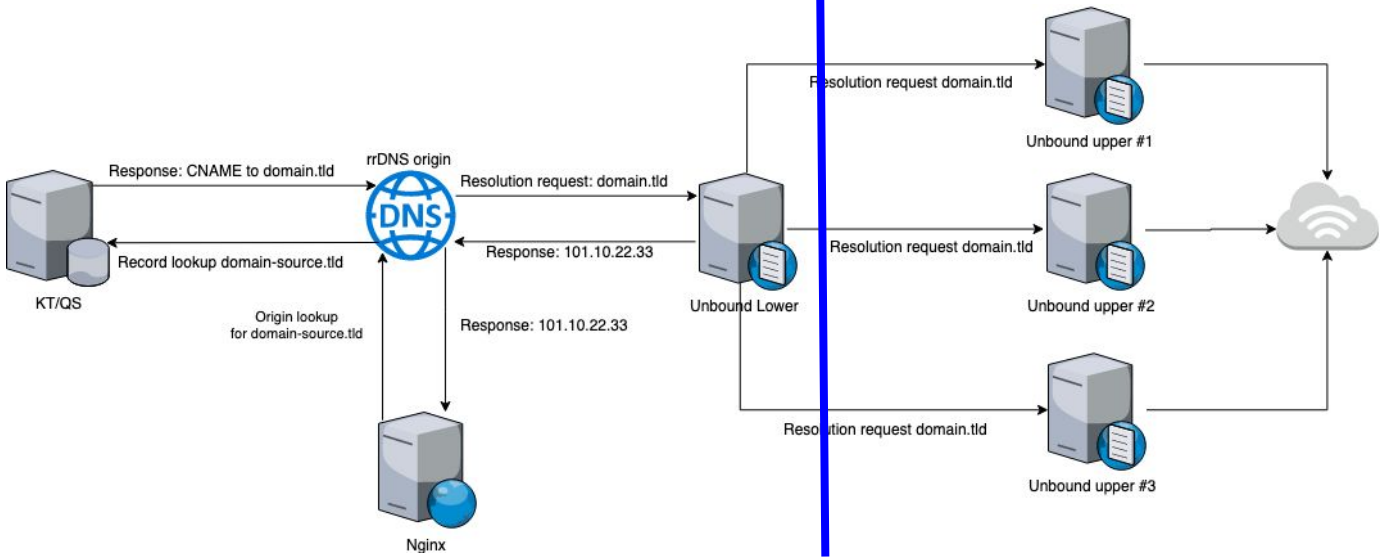
- 3 metals (unbound-Upper) in recursion mode others in forward-first mode
  - For safety reasons
- Uppers answer on anycast addresses
- Unbound-upper has higher maxTTL than -lower
  - To encourage prefetching

CLOUDFLARE

Tiered setup

Metal

Network



CLOUDFLARE

Tiered Unbound deployed

Needed to bring down site to enable

Instant speedup
Less inconsistency

One customer complained

CLOUDFLARE

Issues encountered #1

Lack of upstream monitoring/health checks

Unpredictable

- upstream selection logic
- blacklisting for upstreams in case of issues (SERVFAIL / timeout / Refused)
- behaviour with "forward-first" when -Upper server(s) blacklisted

Addressed many issues by adding reporting

Started using the command interface to unbound more and more

CLOUDFLARE

Issues encountered #2

- Lack of clear logging about reasons for SERVFAIL
  - No ways to distinguish Upstream or Auth DNS failures (SERVFAIL for both)
- Unbound reuses same timeouts for forward servers as for auth DNS.
  - Not suitable for LAN

Added PCAP interface that feeds queries to our logging infrastructure

Still hard to debug SERVFAIL (14 different locations set return code)

Issues encountered #3

Anycast can be hard to debug

Using NAT to forward 127.0.0.1/53 to 127.0.0.1/5353 was a bad idea

Upgrades required taking POP off-line

Integrating into Logging infrastructure was a challenge

    solved by Pcap

Cache hit rate not as high as we hoped

We made mistakes

NAT must die

All applications should talk directly to Stub only system services should use system-stub

Logging and metrics !!!

Debug tools

KSK roll in Oct 2017 might have been a failure due singular reliance on RFC5011 and restarts of Upper on random machines

The good

- Unbound is a great resolver for external resolution
  - in particular on “hostile” networks !! it never gives up.
- It hid from us all the EDNS0 breakage at some performance cost; due to our short maxTTL’s

CLOUDFLARE

V4: DNSdist → Unbound-upper

Selection process

- **Unbound is not suitable as metal DNS provider**
- **Looked at Bind, Knot, PowerDNS ⇒ same issues**
-
- **Write our own ?**
- **dnsdist was different**
-
- **Undo Anycast**
- **Easy to apply policies**

Desires:

- Upper selection
- Work around failures
- Good logging
- Reliable
- Robust
- DNS compliant
- Open Source
- Not Abandon ware
-

CLOUDFLARE

dnsdist good features

- Query distribution by name
- Quick reaction to failures
- Does not confuse timeout vs Servfail
- Fast, reliable, easy to extend
- Policy interface
- DoT, load balancing, rate limiting

Missing:

- Prefetching, using TTL overwrite to overcome
- DNSSEC: we trust Unbound-Upper to do it so not needed
-

CLOUDFLARE

Thinking outside the box

Dnsdist sold as DNS Load Balancer

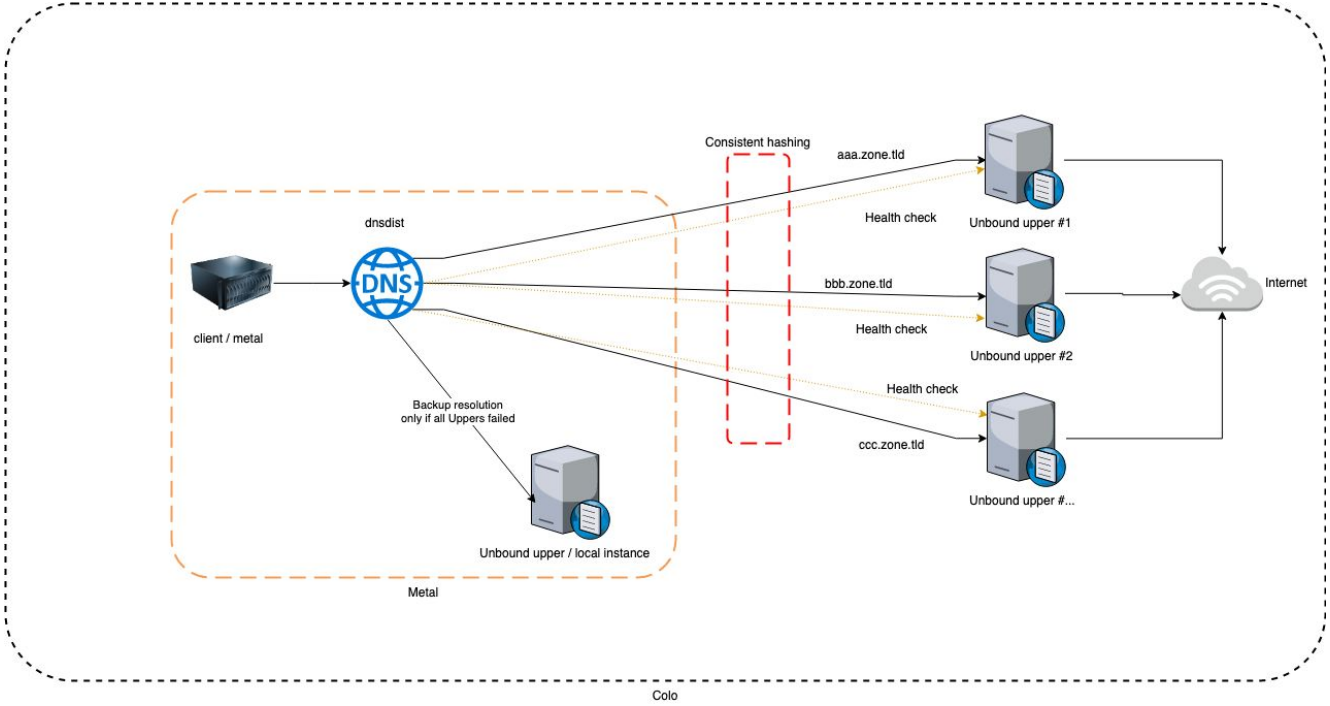
But it is basically Traffic steering with caching ⇒ what we wanted from -lower

Exactly what all stub resolvers should have!

Perfect as local client on all systems that forward queries to resolvers

CLOUDFLARE

Design for CF colo’s



Scaled Uppers from 3 to fraction of colo size

CLOUDFLARE

Deployment process

1. Run in test colo on select metals
2. Integrate into infrastructure
   a. Prometheus
   b. Logging
   c. Dashboards
3. Deploy to small location
4. Deploy to “hostile” location
5. Global rollout small .. large sites

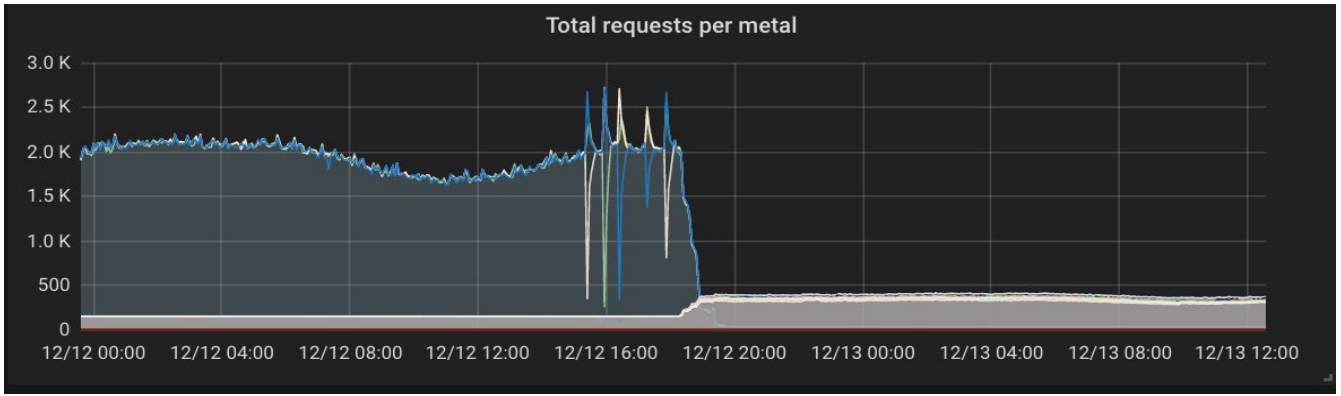
Compare results
Logging and metric parity
Worked with PowerDNS Developers to extend and fix issues we uncovered

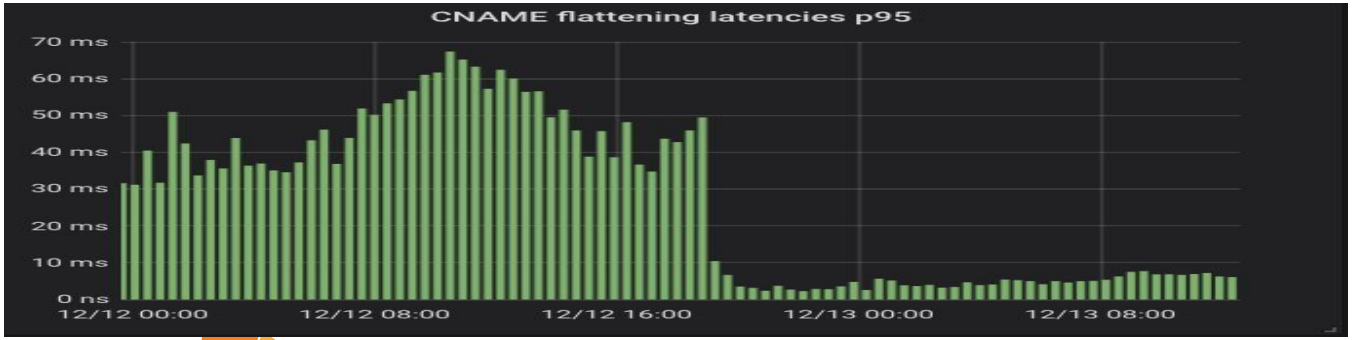
Conservative deployment plan,

Deployment done a quiet periods

CLOUDFLARE

Switching to DNSdist #1



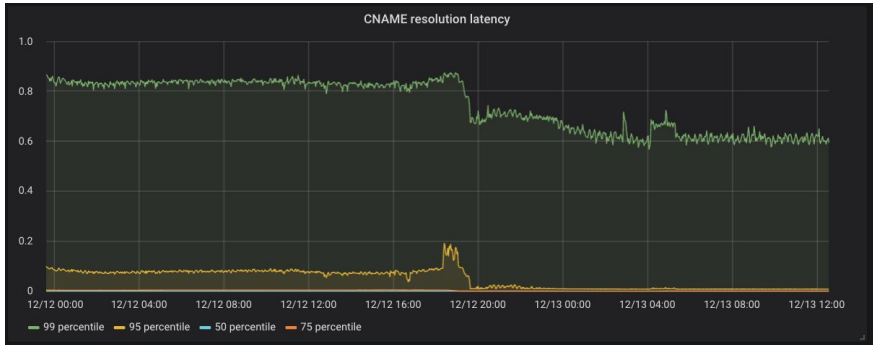


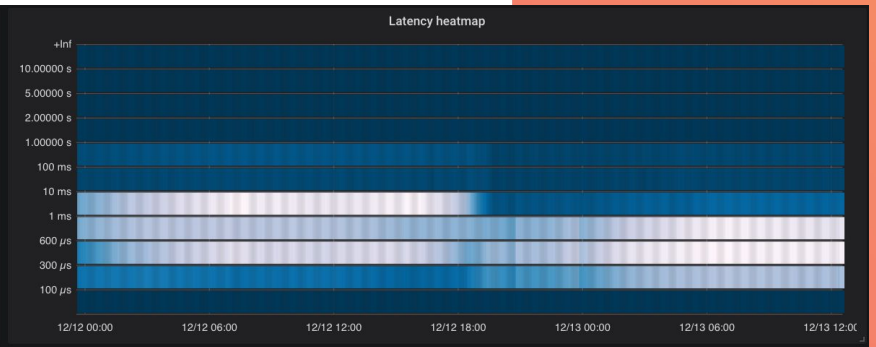
CLOUDFLARE

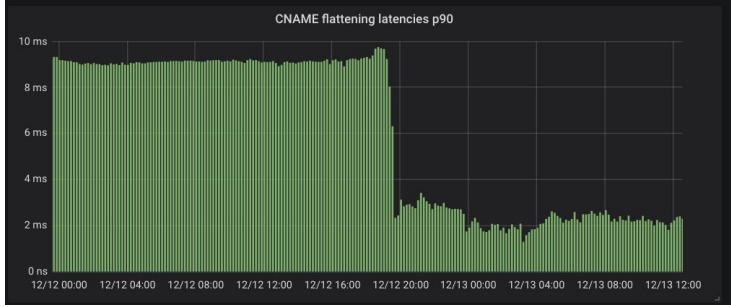
Effect on Upper

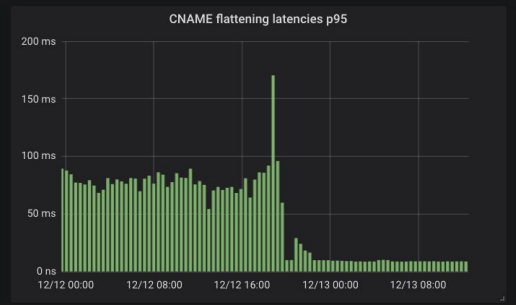
CLOUDFLARE

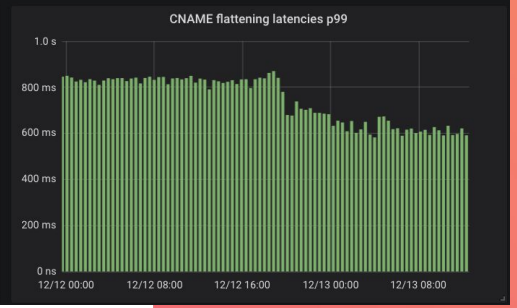
Effect on RRDNS





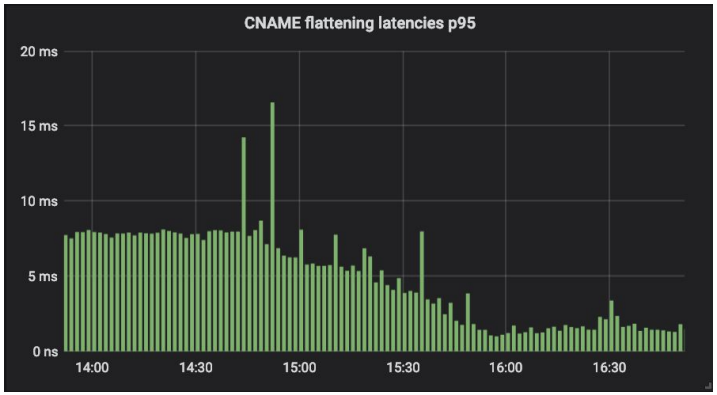


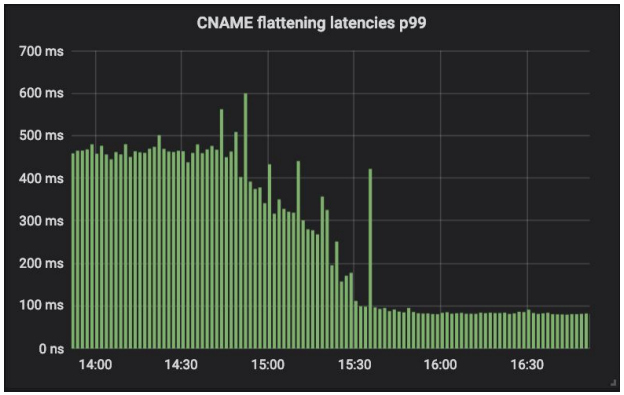


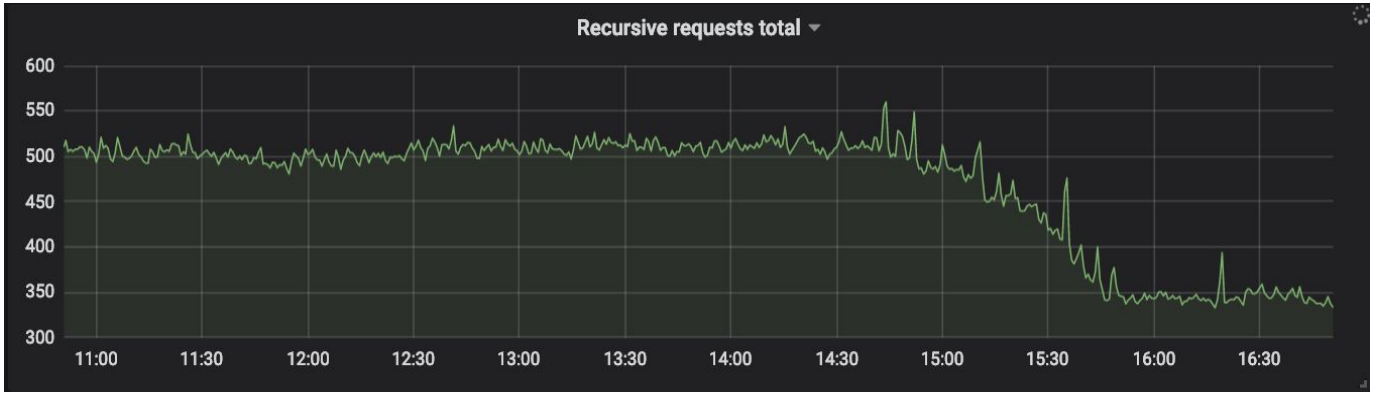


CLOUDFLARE

MaxTTL change 90 → 3607







3607 is a prime selected to maximize prefetching

The result

- Faster
- Fewer problems

Unbound and dnsdist are both great at what they do well !!!

CLOUDFLARE

?

CLOUDFLARE