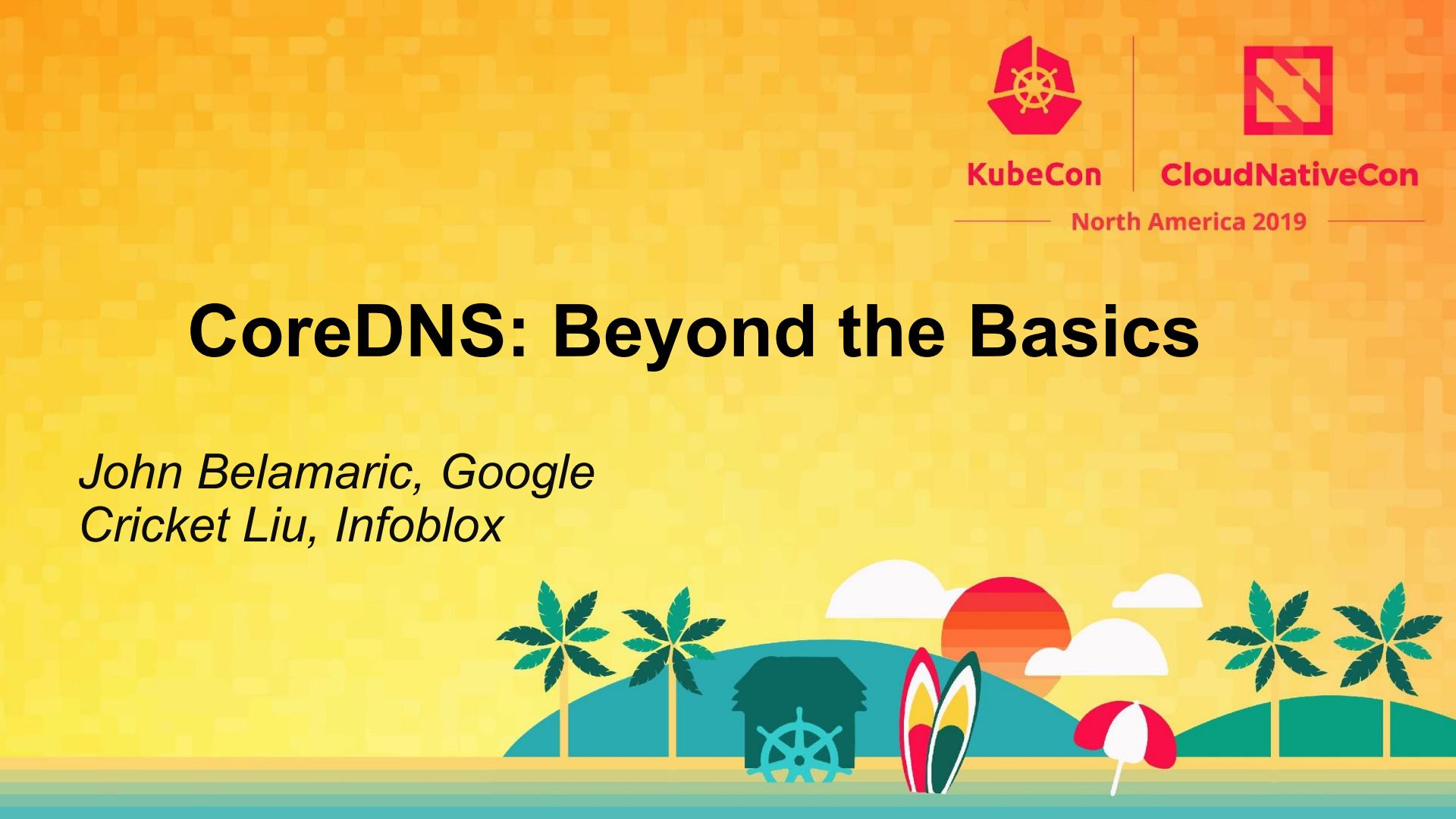
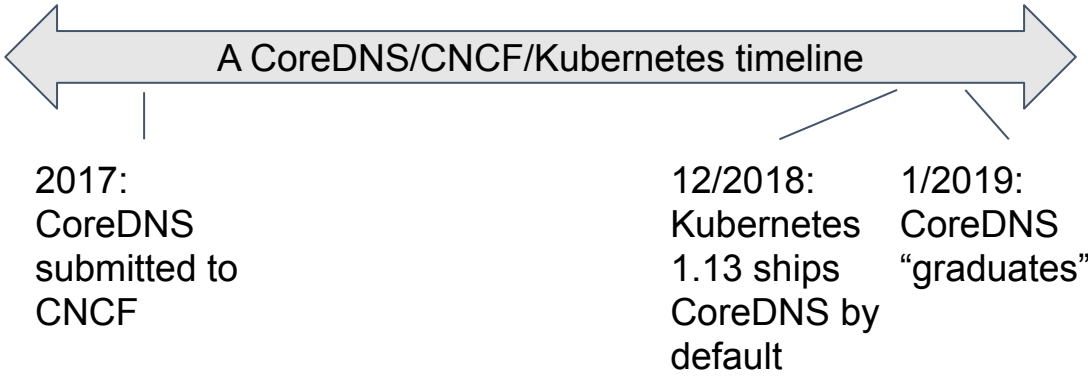


CoreDNS: Beyond the Basics

John Belamaric, Google
Cricket Liu, Infoblox

A Little CoreDNS History

- Project started by Miek Gieben, author of a popular Go language DNS library
  - Miek had written a couple of DNS servers used in containerized environments, SkyDNS and SkyDNS2
  - Miek admired Caddy, a Go-based web server
  - He used Caddy as a framework for CoreDNS



Why CoreDNS?

1. Written in Go, so memory-safe
2. Built-in, straightforward Kubernetes integration
3. Plugin architecture
    a. Many plugins implementing diverse functionality
    b. More being written all the time
    c. Easy to write your own plugin

DNSSEC

DNSSEC? Is That Still a Thing?

- DNSSEC is the DNS Security Extensions, a set of extensions to DNS to digitally sign DNS zone data and support
  - Origin authentication
  - Integrity checking
- “Signing” is the process of calculating signatures and adding them to zone data in the form of a new type of resource record
- Most of the top-level zones in the Internet’s namespace are signed
  - E.g., the root, *com*, *net*, etc.
- More operators of recursive DNS services do DNSSEC validation
  - E.g., Google Public DNS, Quad9, Cloudflare’s 1.1.1.1

CoreDNS Supports Two “Varieties” of DNSSEC Signing

1. Signing “on-the-fly” using the *dnssec* plugin
2. Signing by adding DNSSEC’s resource records to a zone data file is supported using the *sign* plugin

Signing “On-the-Fly” with the *dnssec* Plugin

1. Generate a Common Signing Key or a Key-Signing Key and Zone-Signing Key with BIND 9’s *dnssec-keygen* program. We recommend using ECDSA as an algorithm because it produces shorter signatures.

```
dnssec-keygen -a ECDSAP256SHA256 -f KSK foo.example
```

   This will create files called something like *Kfoo.example.+013+19815.key* (containing the public key) and *Kfoo.example.+013+19815.private* (containing the private key). (“13” is the algorithm number and “19815” is the “key tag.”)

Signing “On-the-Fly” with the *dnssec* Plugin

1. Generate a Common Signing Key or a Key-Signing Key and Zone-Signing Key with BIND 9’s *dnssec-keygen* program. We recommend using ECDSA as an algorithm because it produces shorter signatures.

```
dnssec-keygen -a ECDSAP256SHA256 -f KSK foo.example
```

   This will create files called something like *Kfoo.example.+013+19815.key* (containing the public key) and *Kfoo.example.+013+19815.private* (containing the private key). (“13” is the algorithm number and “19815” is the “key tag.”)

Signing “On-the-Fly” with the *dnssec* Plugin

2. Add the *dnssec* plugin to the server block that handles queries you want to sign. Note that you can sign responses generated through other plugins, such as the *kubernetes* plugin.

```
foo.example {
    file db.foo.example
    dnssec {
        key file Kfoo.example.+013+19815
    }
}
```

Signing “On-the-Fly” with the *dnssec* Plugin

3. Et voila!

```
% dig @127.0.0.1 soa foo.example. +dnssec +norec

; <<>> DiG 9.10.3 <<>> @127.0.0.1 soa foo.example. +dnssec +norec
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60812
;; flags: qr aa; QUERY: 1, ANSWER: 2, AUTHORITY: 3, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;foo.example.                    IN      SOA

;; ANSWER SECTION:
foo.example.            3600    IN      SOA     ns1.foo.example. root.foo.example. 2019041900 3600 600 604800 600
foo.example.            3600    IN      RRSIG   SOA 13 2 3600 20191113233529 20191105203529 19815 foo.example.
7sYyOqo0nfOtKG5iAL1M0xXEoQiaoTuy2bhk6rqU/O0i/e9M1e7Wzkq5 jFEpIpUhGxrAndQPREnp5CSYxelx6Q==

;; AUTHORITY SECTION:
foo.example.            3600    IN      NS      ns1.foo.example.
foo.example.            3600    IN      NS      ns2.foo.example.
foo.example.            3600    IN      RRSIG   NS 13 2 3600 20191113233529 20191105203529 19815 foo.example.
0dtDR1BGHMT13PpWyoZAyXREkyw2E/CWFBpg3MgzzB/i7z7Xlm5R3MH4 3uvL364gE4Gefw2q8SY6mpsHnreQAw==

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Tue Nov 05 15:37:43 PST 2019
;; MSG SIZE  rcvd: 434
```

Signing Zone Data Files with the *sign* Plugin

- The *sign* plugin supports automatic signing (i.e., adding DNSSEC resource records) to static zone data files.
  - Hence it's used with the *file* and *auto* plugins
- *sign* only signs zones using a Common Signing Key, or CSK

Signing Zone Data Files with the *sign* Plugin

1. Generate a Common Signing Key with BIND 9's *dnssec-keygen* program.
2. Add the *sign* plugin to the server block that handles queries you want to sign.

```
bar.example {
    root coredns-files
    file db.bar.example.signed

    sign db.bar.example {
        key file Kbar.example.+013+49537
        directory .
    }
}
```

Signing Zone Data Files with the *sign* Plugin

3. Et voila!

```
% more db.bar.example.signed
bar.example.    100     IN      SOA     ns1.foo.example. root.foo.example.
1573062869 3600 600 604800 200
bar.example.    100     IN      RRSIG   SOA 13 2 100 20191208175429
20191105235423 49537 bar.example.
kxh08gNzcsfa/GCxI2MTUNjR+Nt6z8YTgFR7RjOy0XHIliKq7tMNzH95BXukz+cxD3bpRJ6QI+aj7F6
zWWTxew==
bar.example.    3600    IN      NS      ns1.foo.example.
bar.example.    3599    IN      NS      ns2.foo.example.
bar.example.    100     IN      RRSIG   NS 13 2 100 20191208175429
20191105235423 49537 bar.example.
6w/kw1ak/msCzWHtKwdIqbPCR2i6QIx1frWSmsJodkqBO1rww8TTes2mlLjdn+nznQPRaK+csOj8sp/
DJ0jq5w==
bar.example.    100     IN      RRSIG   DNSKEY 13 2 100 20191208175429
20191105235423 49537 bar.example.
QHrSGEdhnDIhNH1wyj4pcNgQrq1FMfY4iJ0y0kBXFDblDMFvZ8QVWcHQqcoAh2FTRAuSHxvRZqoTt77
lz5lVvg==
```

DNS Over TLS (DoT)

Handling Queries Over TLS

- CoreDNS can handle queries received over TLS
- Just use the “tls://” prefix in your server block

```
tls://. {
    forward . 8.8.8.8 8.8.4.4
    cache
}
```

or

```
tls://foo.example {
    root /corefiles
    file db.foo.example
}
```

Creating and Configuring a TLS Certificate

1. To handle queries over TLS, CoreDNS needs a TLS certificate. You can create one using *openssl*:

```
% openssl req -new -newkey rsa:4096 -x509 -sha256 -days 365 -nodes
-out MyCertificate.crt -keyout MyKey.key
```

2. Then configure CoreDNS to use the new certificate and key:

```
tls://. {
    tls MyCertificate.crt MyKey.key
    forward . 8.8.8.8 8.8.4.4
    cache
}
```

Creating and Configuring a TLS Certificate

3. Et voila!

```
% echo | openssl s_client -connect '127.0.0.1:853' | grep -B 2 -A 5 "Certificate chain"
depth=0 C = US, ST = California, L = Santa Clara, O = Infoblox, OU = Engineering, CN = faith.inca.infoblox.com, emailAddress = cricket@infoblox.com

verify error:num=18:self signed certificate
verify return:1
depth=0 C = US, ST = California, L = Santa Clara, O = Infoblox, OU = Engineering, CN = faith.inca.infoblox.com, emailAddress = cricket@infoblox.com
verify return:1
DONE
CONNECTED(00000003)
```

Sending Queries Over TLS

- Naturally, CoreDNS can also send queries over TLS.
- For example, to query a forwarder over TLS:

```
tls://. {
     forward . tls://8.8.8.8 tls://8.8.4.4
     cache
}
```

- You can even specify client authentication, CA and TLS servername:

```
tls://. {
     tls CERT KEY CA
     tls_servername dns.google
     forward . tls://8.8.8.8 tls://8.8.4.4
     cache
}
```

Managing Zone Data with Git

KubeCon | CloudNativeCon

North America 2019

Managing Zone Data with Git

- Many of you are familiar with Git
- Wouldn’t it be nice to be able to use Git to manage DNS zone data?
  - Distributed management
  - Version tracking
- With the *auto* plugin and a tool called *git-sync*, you can!

Managing Zone Data with Git

1. First, set up *auto* to serve data from any zone data file in a directory:

```
. {
    auto . {
        directory /etc/coredns db\.(.*) {1}
        reload 1m
        transfer to 10.0.1.1
    }
}
```

Managing Zone Data with Git

2. Get *git-sync:*

   https://github.com/kubernetes/git-sync

3. Use *git-sync* to periodically synchronize zone data from your repo:

```
% docker run -d -v /etc/coredns:/tmp/git registry/git-sync \
--repo=https://github.com/myzonedata --branch=master --wait=30
```

~~Haeks~~ Fancy Tricks

Manipulating Queries and Responses

- Various plugins can manipulate queries or responses
- Simple: *loadbalance*
  - Shuffles the order of A, AAAA, and MX records in responses
- Fancy: *template*, *rewrite*, *firewall* (external), *metadata*

Template

- ip-1-1-1-1.example.com A? 1.1.1.1
- ip-1-1-1-1.example.com AAAA? No data, NOERROR
- foo.example.com A? NXDOMAIN

```
example.com:5300 {
  errors
  log

  template IN A {
    match (^|[.])ip-(?P<a>[0-9]*)-(?P<b>[0-9]*)-(?P<c>[0-9]*)-(?P<d>[0-9]*)[.]example[.]com[.]$
    answer "{{ .Name }} 60 IN A {{ .Group.a }}.{{ .Group.b }}.{{ .Group.c }}.{{ .Group.d }}"
    fallthrough
  }
  template IN ANY {
    match (^|[.])ip-(?P<a>[0-9]*)-(?P<b>[0-9]*)-(?P<c>[0-9]*)-(?P<d>[0-9]*)[.]example[.]com[.]$
    rcode NOERROR
    fallthrough
  }
  template IN ANY {
    rcode NXDOMAIN
  }
}
```

Rewrite

- .org => .com on the way in
- .com => .org on the way out, IF original was rewritten
- Looks normal to client

```
example.com:5300 example.org:5300 {
  errors
  log

  rewrite {
    name regex (.*).example.org {1}.example.com
    answer name (.*).example.com {1}.example.org
  }

  ...templates are the same as before...
}
```

Metadata + Firewall

- *metadata* - it exposes data from one plugin to others
- *metadata_edns0* - unpacks EDNS0 options and puts them in metadata
- *firewall* can operate on request data AND metadata!

```
example.com:5300 {
  errors
  log

  metadata
  metadata_edns0 {
    secret 65200 bytes
  }

  firewall query {
    allow [metadata_edns0/secret] == 'C0reDNSR0cks!'
    refuse true
  }

  ...templates are the same as before...
}
```

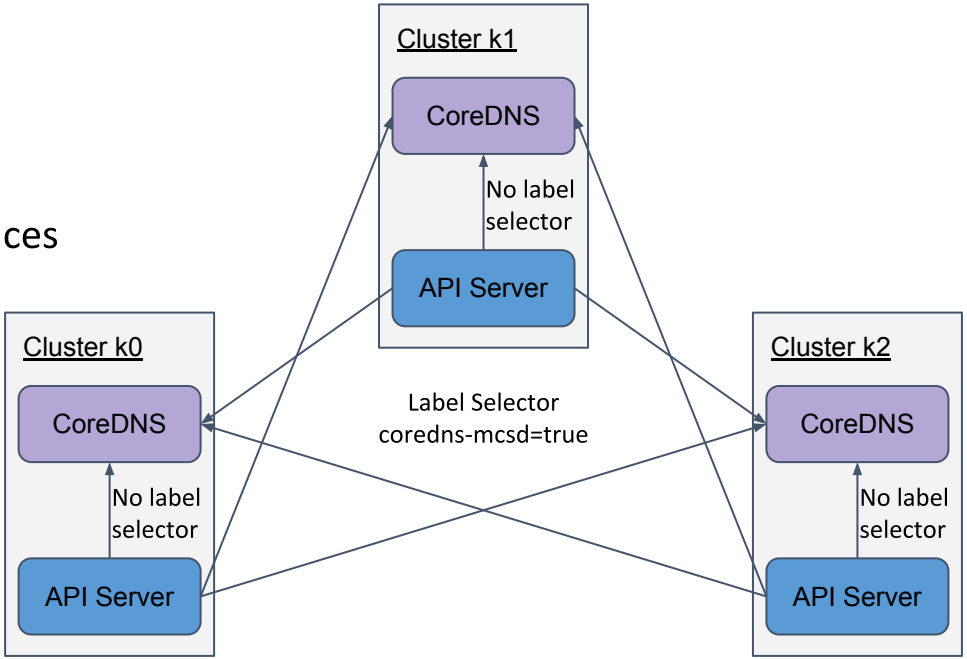
Rewrite, Firewall, and Template

Demo

Multicluster Service Discovery

Multicluster Service Discovery

- https://github.com/coredns/multicluster-dns
- Use the kubernetai plugin
- Each CoreDNS instance:
  - Talks to every cluster API server
  - Serves up all local services
  - Serves up explicitly labeled remote services
- Service names
  - Can be shared across clusters
  - Can have unique cluster zones
- Only makes sense for headless services
  - For now...



Multicluster Service Discovery

- Example Corefiles for k0 - each cluster will have a different one
- With same service name, there is a fixed ordering of clusters

Same service name

```
.:53 {
    …standard log, errors, health, etc…

    kubernetai cluster.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k0
        fallthrough in-addr.arpa ip6.arpa cluster.local
    }
    kubernetai cluster.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k1
        labels coredns-mcsd=true
        fallthrough in-addr.arpa ip6.arpa cluster.local
    }
    kubernetai cluster.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k2
        labels coredns-mcsd=true
        fallthrough in-addr.arpa ip6.arpa cluster.local
    }
}
```

Cluster-specific service name

```
.:53 {
    …standard log, errors, health, etc…

    kubernetai cluster.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k0
        fallthrough in-addr.arpa ip6.arpa
    }
    Kubernetai k1.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k1
        labels coredns-mcsd=true
        fallthrough in-addr.arpa ip6.arpa
    }
    kubernetai k2.local in-addr.arpa ip6.arpa {
        kubeconfig /root/kubeconfig k2
        labels coredns-mcsd=true
        fallthrough in-addr.arpa ip6.arpa
    }
}
```

Multi-cluster Service Discovery

Demo


KubeCon
CloudNativeCon
North America 2019
Q & A