DNS over IPFS

A Very Stupid Idea You Should Not Use

Beka (@beka#somenumbers) and Will (@kernelmethod#5003)

Outline

- W.. why?
- Overview of DNS
- Overview of IPFS
- Using IPFS for DNS
- Improving the Design

W.. why?

W.. why?

- Why not!

W.. why?

- Why not!
- Decouple DNS infrastructure from authorities

W.. why?

- Why not!
- Decouple DNS infrastructure from authorities
  - Domain authorities must have computers running DNS servers

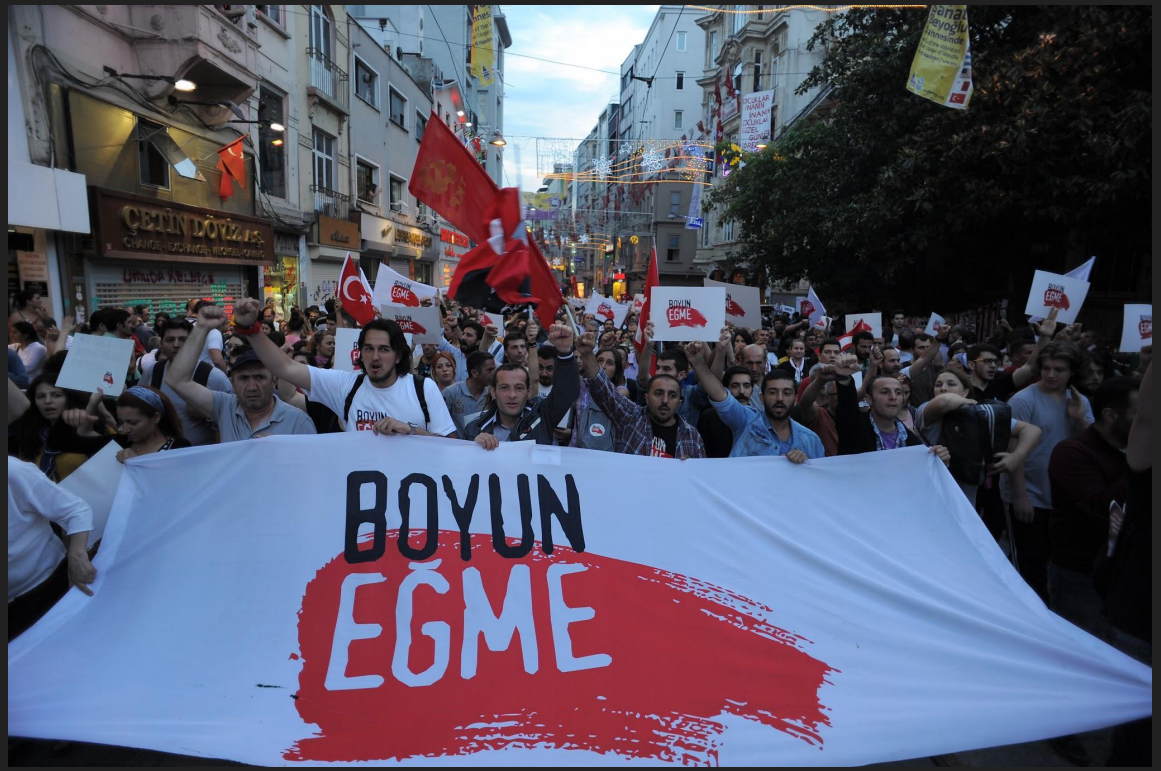
W.. why?

- Why not!
- Decouple DNS infrastructure from authorities
  - Domain authorities must have computers running DNS servers
  - That means the average person doesn’t run their own DNS, they get someone else to do it, such as Google, Tucows, GoDaddy, etc.

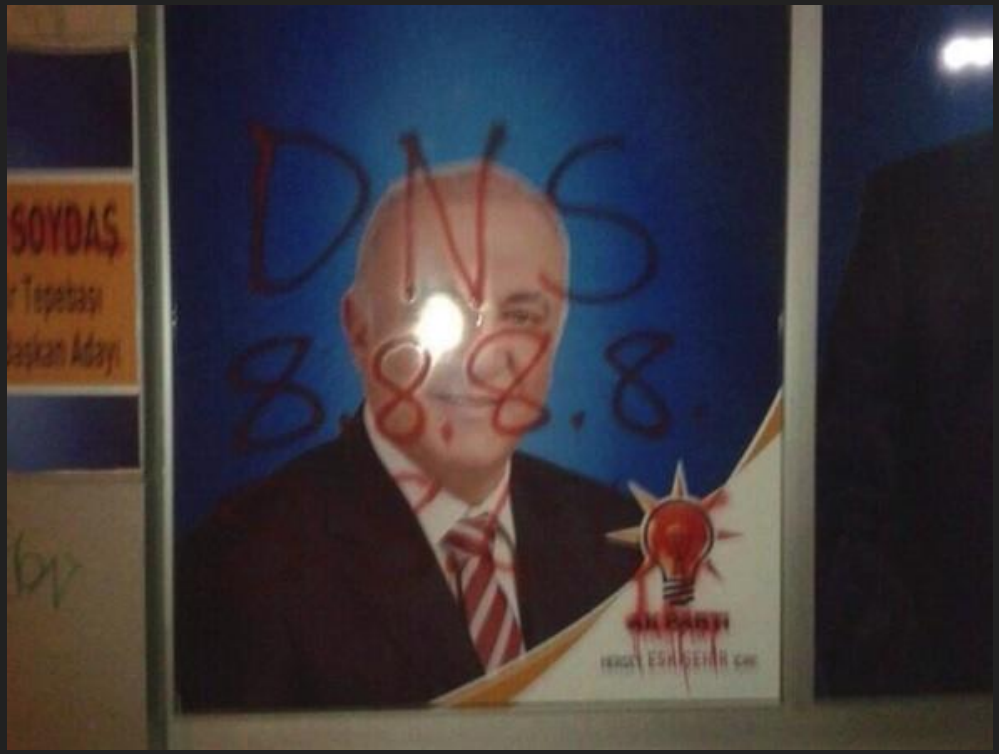
W.. why?

- Why not!
- Decouple DNS infrastructure from authorities
  - Domain authorities must have computers running DNS servers
  - That means the average person doesn’t run their own DNS, they get someone else to do it, such as Google, Tucows, GoDaddy, etc.
- Censorship resistance

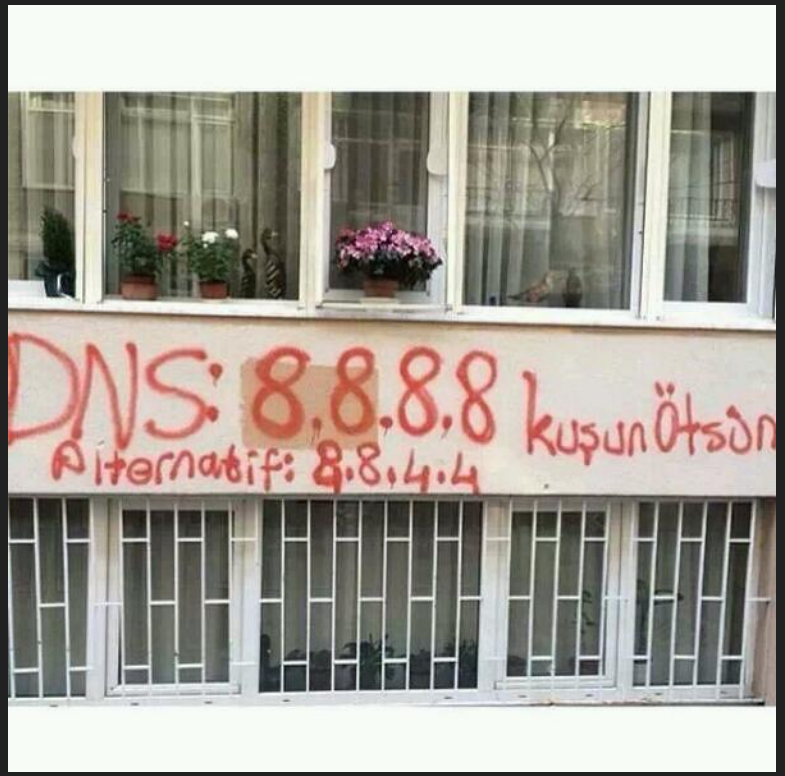
W.. why?



W.. why?



W.. why?



Overview of DNS

Overview of DNS

- Remembering IP addresses sucks

Overview of DNS

- Remembering IP addresses sucks
- What if names?

Overview of DNS

- Remembering IP addresses sucks
- What if names?
- Hierarchical name resolution

DNS Example

- www.google.com

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)
- Ask 5.6.7.8: who knows about `www`?

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)
- Ask 5.6.7.8: who knows about `www`?
- Learn the `www` sub-name server is at IPv4 address 9.10.11.12 (not real)

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)
- Ask 5.6.7.8: who knows about `www`?
- Learn the `www` sub-name server is at IPv4 address 9.10.11.12 (not real)
- Ask 9.10.11.12: where can I get your data?

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)
- Ask 5.6.7.8: who knows about `www`?
- Learn the `www` sub-name server is at IPv4 address 9.10.11.12 (not real)
- Ask 9.10.11.12: where can I get your data?
- Learn the `www.google.com` data is at 13.14.15.16

DNS Example

- www.google.com
- Ask a root name server: who knows about the `com` TLD?
- Learn the `com` name server is at IPv4 address 1.2.3.4 (not real)
- Ask 1.2.3.4: who knows about `google`?
- Learn the `google` sub-name server is at IPv4 address 5.6.7.8 (not real)
- Ask 5.6.7.8: who knows about `www`?
- Learn the `www` sub-name server is at IPv4 address 9.10.11.12 (not real)
- Ask 9.10.11.12: where can I get your data?
- Learn the `www.google.com` data is at 13.14.15.16
- Ask 13.14.15.16 for the data (this is done with HTTP or whatever else)

IPFS Overview

IPFS Overview

- Two Distributed Hash Tables

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)
- Want to download some file? Look up its hash in the CID-based DHT

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)
- Want to download some file? Look up its hash in the CID-based DHT
- Want to download by mutable feed? Look up my Peer ID

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)
- Want to download some file? Look up its hash in the CID-based DHT
- Want to download by mutable feed? Look up my Peer ID
- The CID stuff is called just “IPFS” usually, the Peer ID stuff is called IPNS
    - IPNS = InterPlanetary Name System, a bad name

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)
- Want to download some file? Look up its hash in the CID-based DHT
- Want to download by mutable feed? Look up my Peer ID
- The CID stuff is called just “IPFS” usually, the Peer ID stuff is called IPNS
    - IPNS = InterPlanetary Name System, a bad name
- CIDs are determined by the content, so what content is stored at a CID is immutable and unchangeable forever

IPFS Overview

- Two Distributed Hash Tables
- One maps content ID’s (hashs of content) to Peer IDs who host the data
- Other maps Peer IDs (public keys) to some chosen content ID (CID)
- Want to download some file? Look up its hash in the CID-based DHT
- Want to download by mutable feed? Look up my Peer ID
- The CID stuff is called just “IPFS” usually, the Peer ID stuff is called IPNS
    - IPNS = InterPlanetary Name System, a bad name
- CIDs are determined by the content, so what content is stored at a CID is immutable and unchangeable forever
- IPNS Peer IDs are just public keys, so their values can be changed but only by the owner of the pubkey

Using IPFS for DNS

Using IPFS for DNS

- Each name authority (e.g. the ICANN, who is the root authority, Verisign, who is the `com` authority, Google who is the `google.com` authority, etc.) runs an IPFS node. That gives them a Peer ID.

Using IPFS for DNS

- Each name authority (e.g. the ICANN, who is the root authority, Verisign, who is the `com` authority, Google who is the `google.com` authority, etc.) runs an IPFS node. That gives them a Peer ID.
- At /ipns/$PEERID, the authority stores the CID for DNS records in some format (maybe JSON for lulz) for the domains it controls

Using IPFS for DNS

- Each name authority (e.g. the ICANN, who is the root authority, Verisign, who is the `com` authority, Google who is the `google.com` authority, etc.) runs an IPFS node. That gives them a Peer ID.
- At /ipns/$PEERID, the authority stores the CID for DNS records in some format (maybe JSON for lulz) for the domains it controls

    For example, /ipns/$ICANN might store /ipfs/$ICANNCID, which stores the data

        { “com”: { “NS”: “/ipns/$VERISIGN”, … }, … }

Using IPFS for DNS

For example, /ipns/$ICANN might store /ipfs/$ICANNCID, which stores the data

{ “com”: { “NS”: “/ipns/$VERISIGN”, … }, … }

- To look up the `com` TLD’s information…

Using IPFS for DNS

For example, /ipns/$ICANN might store /ipfs/$ICANNCID, which stores the data

{ “com”: { “NS”: “/ipns/$VERISIGN”, … }, … }

- To look up the `com` TLD’s information…
- Then you query /ipns/$ICANN, and get /ipfs/$ICANNCID…

Using IPFS for DNS

For example, /ipns/$ICANN might store /ipfs/$ICANNCID, which stores the data

{ “com”: { “NS”: “/ipns/$VERISIGN”, … }, … }

- To look up the `com` TLD’s information…
- Then you query /ipns/$ICANN, and get /ipfs/$ICANNCID…
- Then query /ipfs/$ICANNCID, to get the JSON blob…

Using IPFS for DNS

For example, /ipns/$ICANN might store /ipfs/$ICANNCID, which stores the data

{ “com”: { “NS”: “/ipns/$VERISIGN”, … }, … }

- To look up the `com` TLD’s information…
- Then you query /ipns/$ICANN, and get /ipfs/$ICANNCID…
- Then query /ipfs/$ICANNCID, to get the JSON blob…
- Finally look at the “com” key, then the “NS” key, and learn that the `com` TLD’s “nameserver” is located at /ipns/$VERISIGN

Using IPFS for DNS

- Abstractly:

  /ipns/$ICANN points to /ipns/$VERISIGN along the key `(com,NS)`, using some indirection through the IPFS content level

  /ipns/$ICANN —(com,NS)—> /ipns/$VERISIGN

Using IPFS for DNS

- To continue looking up `www.google.com` we just repeat this process starting at /ipns/$VERISIGN

Using IPFS for DNS

- To continue looking up `www.google.com` we just repeat this process starting at /ipns/$VERISIGN
- /ipns/$VERISIGN points to some CID, getting us to /ipfs/$VERISIGNCID

Using IPFS for DNS

- To continue looking up `www.google.com` we just repeat this process starting at /ipns/$VERISIGN
- /ipns/$VERISIGN points to some CID, getting us to /ipfs/$VERISIGNCID
- Loading that, we'll get some DNS records that look like

```
{
    “google”: { “NS”: “/ipns/$GOOGLE”, … },
    “twitter”: {“NS”: “/ipns/$TWITTER”, … },
    …
}
```

Using IPFS for DNS

- Same for the next domain, `www`:

  /ipns/$GOOGLE has /ipfs/$GOOGLECID which stores the content

```
{
    “www”: { “A”: “142.250.191.36”, … },
    “gmail”: { “A”: “142.250.189.238”, … },
    “calendar”: { “A”: “142.250.191.46”, … },
    …
}
```

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Again, the process is like this:

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Again, the process is like this:

/ipns/$ICANN —(com,NS)—> /ipns/$VERISIGN

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Again, the process is like this:

/ipns/$ICANN —(com,NS)—> /ipns/$VERISIGN

/ipns/$VERISIGN —(google,NS)—> /ipns/$GOOGLE

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Again, the process is like this:

/ipns/$ICANN —(com,NS)—> /ipns/$VERISIGN

/ipns/$VERISIGN —(google,NS)—> /ipns/$GOOGLE

/ipns/$GOOGLE —(www,A)—> 142.250.191.36

Using IPFS for DNS

Now you know that `www.google.com` lives at IPv4 address 142.250.191.36!

Again, the process is like this:

/ipns/$ICANN —(com,NS)—> /ipns/$VERISIGN

/ipns/$VERISIGN —(google,NS)—> /ipns/$GOOGLE

/ipns/$GOOGLE —(www,A)—> 142.250.191.36

Improving the Design

Improving the Design

- There’s a massive glaring problem with idea, sadly.

Improving the Design

- There’s a massive glaring problem with idea, sadly.
- I mean, aside from how slow IPFS is.

Improving the Design

- There’s a massive glaring problem with idea, sadly.
- I mean, aside from how slow IPFS is.
- Namely uh…

Improving the Design

- There’s a massive glaring problem with idea, sadly.
- I mean, aside from how slow IPFS is.
- Namely uh…

Literally over 100,000,000 .com domain names

Improving the Design

- There’s a massive glaring problem with idea, sadly.
- I mean, aside from how slow IPFS is.
- Namely uh…
- Literally over 100,000,000 .com domain names
- That’s one chonky JSON object!

Improving the Design

- Option 1: Use prefix tries?

Improving the Design

- Option 1: Use prefix tries?
- /ipns/$VERISIGN points to /ipfs/$VERISIGNCID which contains the `com` TLD’s DNS records

Improving the Design

- Option 1: Use prefix tries?
- /ipns/$VERISIGN points to /ipfs/$VERISIGNCID which contains the `com` TLD's DNS records
- Instead of just a giant blob like

```
{
    "google": …,
    "twitter": …,
    …
}
```

Improving the Design

- Option 1: Use prefix tries?
- /ipns/$VERISIGN points to /ipfs/$VERISIGNCID which contains the `com` TLD's DNS records
- We'll use some indirection:

```
{
    …,
    "g": "/ipns/$VERISIGN_G_DOMAINS",
    …,
    "t": "/ipns/$VERISIGN_T_DOMAINS",
    …
}
```

Improving the Design

- We'll use some indirection:

```
{   …,
    "g": "/ipns/$VERISIGN_G_DOMAINS",
    … }
```

- If we look up /ipns/$VERISIGN_G_DOMAINS, we get some CID /ipfs/$VERISIGN_G_DOMAINS_CID

Improving the Design

- We'll use some indirection:

```
{   …,
    "g": "/ipns/$VERISIGN_G_DOMAINS",
    … }
```

- If we look up /ipns/$VERISIGN_G_DOMAINS, we get some CID /ipfs/$VERISIGN_G_DOMAINS_CID
- And if we look *that* up, we get another layer of the prefix trie

Improving the Design

- If we look up /ipns/$VERISIGN_G_DOMAINS, we get some CID /ipfs/$VERISIGN_G_DOMAINS_CID
- And if we look *that* up, we get another layer of the prefix trie

```
{   …,
    “o”: “/ipns/$VERISIGN_GO_DOMAINS”,
    … }
```

Improving the Design

- And so on and so on, all the way down until /ipns/$VERISIGN_GOOGLE_DOMAINS, etc

Improving the Design

- And so on and so on, all the way down until /ipns/$VERISIGN_GOOGLE_DOMAINS, etc
- At each layer, the object contains single-character fields for each alphabetical letter that needs to be recorded (i.e. if there’s a domain name with that prefix)

Improving the Design

- And so on and so on, all the way down until /ipns/$VERISIGN_GOOGLE_DOMAINS, etc
- At each layer, the object contains single-character fields for each alphabetical letter that needs to be recorded (i.e. if there's a domain name with that prefix)
- Each layer also can store a "records" field, which will have a JSON object that has all the DNS records for the domain name that terminates at that part of the trie

Improving the Design

- And so on and so on, all the way down until /ipns/$VERISIGN_GOOGLE_DOMAINS, etc
- At each layer, the object contains single-character fields for each alphabetical letter that needs to be recorded (i.e. if there's a domain name with that prefix)
- Each layer also can store a "records" field, which will have a JSON object that has all the DNS records for the domain name that terminates at that part of the trie
    - E.g. /ipfs/$VERISIGN_GO_DOMAINS_CID contains

      "records": $RECORDS_FOR_GO_DOT_COM

Improving the Design

- Tries are kind of fine, I guess

Improving the Design

- Tries are kind of fine, I guess
- They can get quite deep if you’re doing only one character at a time, so maybe you want to use two or three characters instead

Improving the Design

- Tries are kind of fine, I guess
- They can get quite deep if you’re doing only one character at a time, so maybe you want to use two or three characters instead
- But we can and should do better!

Improving the Design

- Option 2: Range Trees (or whatever they’re called)

Improving the Design

- Option 2: Range Trees (or whatever they’re called)
- Sort all the registered `.com` domain names and divide them into some number of roughly equally sized groups, maybe 100

Improving the Design

- Option 2: Range Trees (or whatever they’re called)
- Sort all the registered `.com` domain names and divide them into some number of roughly equally sized groups, maybe 100
- Maybe that looks like

    a.com to azimuth.com
    azimuths.com to blond-roast-coffee-rules.com
    blond-roast-coffee-sucks.com to coffee-in-general-sucks-actually.com
    etc.

Improving the Design

- Each of those ranges would be keys in the Range Tree, and the value would point to IPNS IDs for the sub-tree for that range

```
{
    “a.com_azimuth.com”:
        “/ipns/$SUBTREE_FOR_A_TO_AZIMUTH”,
    “azimuths.com_blond-roast-coffee-rules.com”:
        “/ipns/$SUBTREE_RECORDS_FOR_AZIMUTHS_TO_ETC”,
    …
}
```

Improving the Design

- Each of those ranges would be keys in the Range Tree, and the value would point to IPNS IDs for the sub-tree for that range
- Each of those sub-trees would themselves be Range Trees, that further narrow the ranges down

Improving the Design

- Each of those ranges would be keys in the Range Tree, and the value would point to IPNS IDs for the sub-tree for that range
- Each of those sub-trees would themselves be Range Trees, that further narrow the ranges down
- At the leaves, the keys aren’t *ranges*, but rather single domain names, and the values are the DNS records for those names

Improving the Design

- Each of those ranges would be keys in the Range Tree, and the value would point to IPNS IDs for the sub-tree for that range
- Each of those sub-trees would themselves be Range Trees, that further narrow the ranges down
- At the leaves, the keys aren’t *ranges*, but rather single domain names, and the values are the DNS records for those names
- This gives us a nice logarithmic depth. The 100-wide fanout gives us a depth of 4, and the size of each Range Tree node isn’t too big.

Improving the Design

- Options 3+: Lots of other possibilities for data structures!
- Also probably we want to have a better representation of DNS records, this was all just very hacky and quick and dirty

EOF