System design of DNS

Santosh P. Follow 19 min read · Aug 9, 2024

The Domain Name System (DNS) is a hierarchical and decentralized naming system for computers, services, or other resources connected to the Internet or a private network.- Wikipedia.

Internet is a large network of connected devices, in which we have physical / virtual devices, small iot instruments and even the websites, Web applications and web services which are being hosted with their own IP addresses in the network. These ipaddresses are their identity to be reached out. The IP addresses are dynamic in nature and being assigned by DHCP servers. Because of it's dynamic nature, using ip addresses to directly reach out to specific target is not possible, for that we have the static domain names. DNS helps to resolve these domain names by mapping them to respective ipaddresses to reach the appropriate target.

Functional requirement

1) Associate domain name to the IP addresses.

2) Managing entries for faster search.

3) Enables access routing to services to keep HTTP web traffic and network traffic flowing.

4) To perform Load balancing. DNS servers need to be distributed, the requests are distributed from a single-point-of-entry to multiple servers in the background (DNS loadbalancing).

5) DNS should also used as a Firewall. In case of DNS attack, the original IP address could be changed to a fake one and all the users would be redirected to a fraudulent servers which could collect sensitive user information. DNS should able to address this issue.

6) Provided security against vulnerability, defined by DNS-SEC.

7) Helps in automating network provisioning, VM provisioning and service provisioning.

NonFunctional requirement

1) **Fault Tolerant**: DNS should ensure that one server in TLD zone goes down, does not affect the DNS flow.

2) **Scalable**: DNS to be designed in distributed architecture with capability of horizontal scaling.

3) **Highly Available**: DNS system need to ensure it's availability to the user, as hundreds of operators and thousands of services rely on it.

4) **Predictable Performance and Lower latency**. DNS shouldn't have any lag in responding.

Back of the Envelope Estimation

```
Some of the rough estimates are:
-- Over 100K new domains per day.
-- Over 190K domain name servers per month.
-- Over 10K domain registers per month.
-- Over 4K top level domains per month.
-- Over 70B events (DNS queries) per day.
```

Deployment Strategy

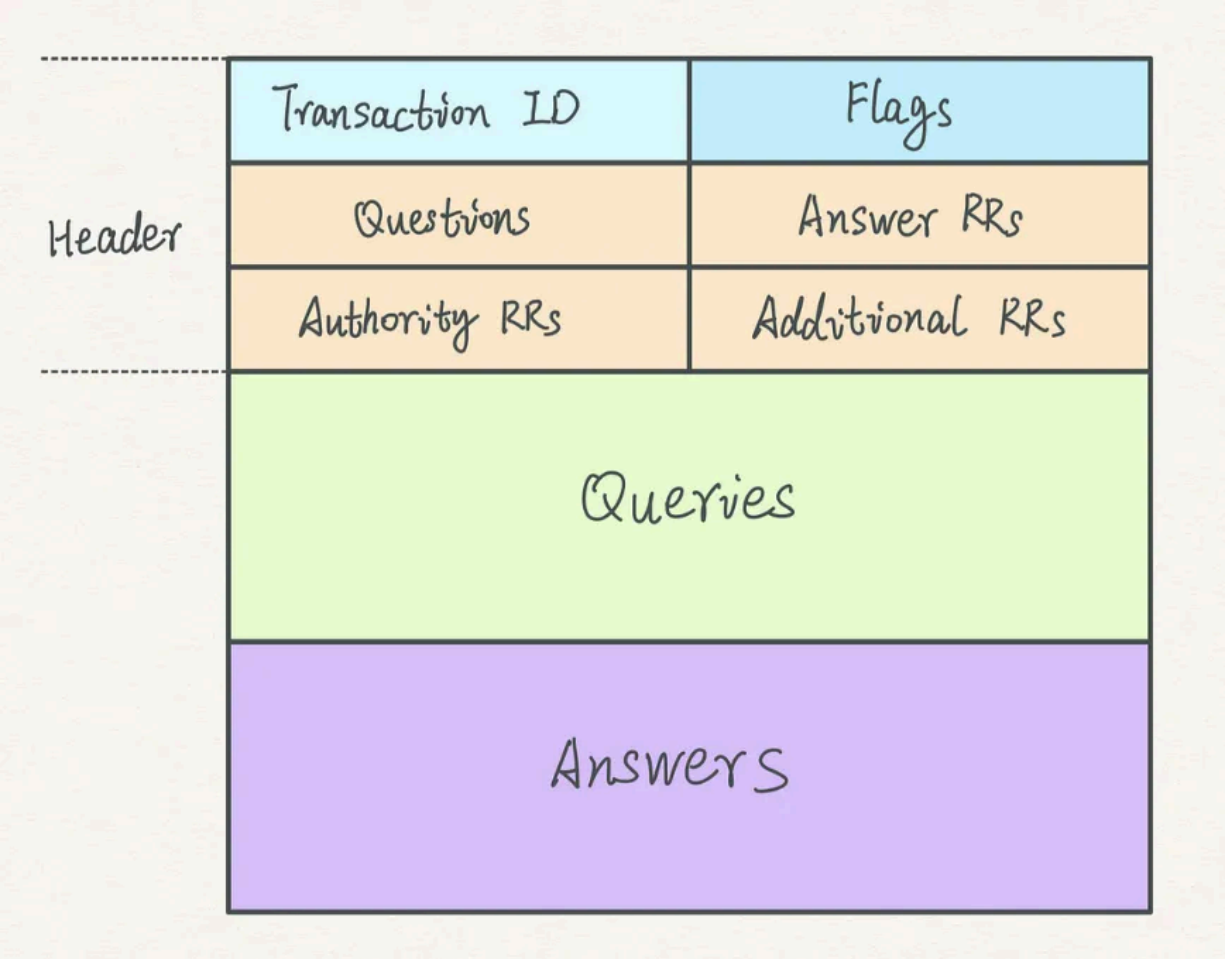
Domain Name Service(DNS) is a distributed system where it has to address customer's request for the services hierarchically distributed across AZR,

spread across multiple data centers located worldwide. The DNS resolution process is a complicated process that involves intricate interactions between **DNS clients**, **recursive servers**, **authoritative name servers** to fetch information stored in the form of **DNS records.**

The DNS name or **Fully Qualified Domain Name (FQDN)** in the form of **DNS query** originate on behalf of DNS client reach a **DNS resolver** which is a crucial component of the internet that helps us to find the ip address associated with a specific domain name. A DNS client has a BIND or local cache for storing some frequently accessed domain name.

> *BIND (**Berkeley Internet Name Domain**) provides a combination of a lightweight resolver library that can be run on DNS clients, such as host operating systems or routers, and a resolver daemon process which can run on a local host. Both communicate over a UDP-based Lightweight Resolver Protocol. The BIND DNS server **resolves and caches successful or failed lookups**.*

A DNS query message is typically composed of the following sections:



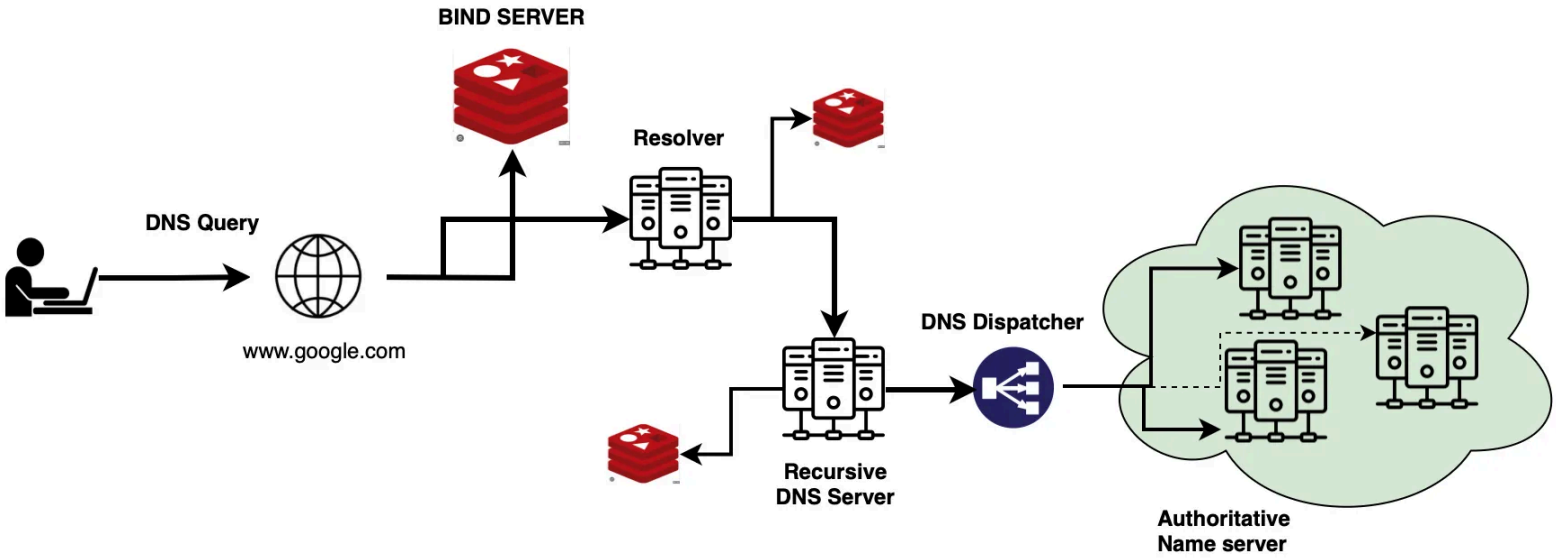
Header: 12 bytes with transaction ID, flags.

Questions section will have **QNAME:** The domain name being queried. QTYPE will have type of DNS record being requested.

```
QNAME for www.example.com

   3www7example3com0

QTYPE:
A (1): IPv4 address.
AAAA (28): IPv6 address.
CNAME (5): Canonical name.
MX (15): Mail exchange record.
PTR (12): Pointer to a canonical name.
```



If the resolver does not have a cached record, it sends a query to a **recursive DNS server**, which then resolves the domain by querying **authoritative name servers.. DNS dispatcher** distribute and loadbalance the DNS queries based on policy information.

DNS record format

A DNS record typically consists of the following fields:

Name	Type	Class	TTL	RDLength	RDATA

The following example shows how dns client fetch the dns information from dns server.

```
import (
    "fmt"
    "github.com/miekg/dns"
)
```

```
func main() {
    // Create a new DNS message
    m := new(dns.Msg)
    m.SetQuestion(dns.Fqdn("example.com"), dns.TypeA)
```

```
// Set the DNS server to use for the query
    dnsServer := "8.8.8.8:53" // Google's public DNS server
```

```
// Perform the DNS query
    c := new(dns.Client)
    in, _, err := c.Exchange(m, dnsServer)
    if err != nil {
        fmt.Printf("Failed to perform DNS query: %v\n", err)
        return
    }
```

```
// Print the response
    for _, ans := range in.Answer {
        if aRecord, ok := ans.(*dns.A); ok {
            fmt.Printf("A record: %s\n", aRecord.A.String())
        }
    }
}
```

*A person that is visiting web sites asks **Recursive DNS servers** for the lookups. Recursive DNS servers then ask the necessary Authoritative Name Server for the*

*answer. **Authoritative DNS servers** store the "maps" of domain names to IP addresses. This domain name to IP mapping is usually configured by system administrators. Then the Recursive name server will give this answer to the person needing the information.*

```
Note: Recursive and authoritative DNS servers are integral parts of the DNS
infrastructure, each playing a unique role. Recursive DNS servers handle
the process of querying multiple DNS servers to resolve domain names for
clients, often caching results to enhance performance. Authoritative DNS
servers provide the definitive answers for domain queries based on the DNS
records they store.
```

DNS record is vital for the proper functioning of domain names on the internet. It map domain names to IP addresses, specify mail servers, provide information for service discovery, and more. Understanding and configuring these records correctly is essential for managing domain name services. **DNS record** encapsulate essential information about the domain. Common record types include:

1. **A (IPv4 Address)**: Associates a domain with an IPv4 address.

```
example.com.    IN    A    192.0.2.1
```

2. **AAAA (IPv6 Address)**: Associates a domain with an IPv6 address.

```
example.com.    IN    AAAA    2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

3. **CNAME** (Canonical Name): Creates an alias for a domain.

```
www.example.com.    IN    CNAME    example.com.
```

4. **MX (Mail Exchange):** Specifies mail servers for a domain.

```
example.com.    IN    MX    10 mail1.example.com.
example.com.    IN    MX    20 mail2.example.com.
```

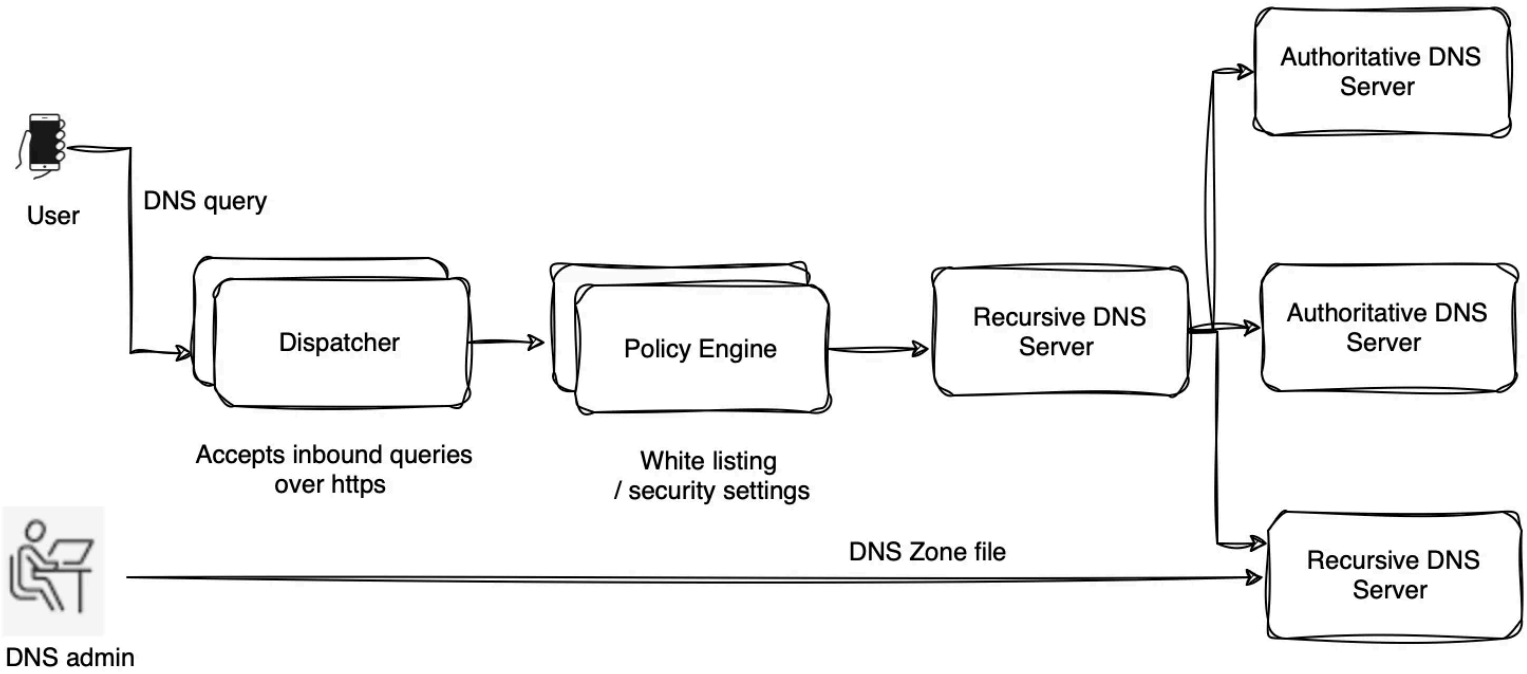
5. **SOA (Start of Authority):** Contains crucial information about a DNS zone.

```
@    IN    SOA    primary-ns hostmaster-email (
                  serial-number
                  refresh-interval
                  retry-interval
                  expire-interval
                  minimum-TTL )
```

DNS caching store query results for specified amount of TTL or time to live. DNS caching can be in **Resolver or in client-side or even in recursive servers.** When a DNS resolver (recursive DNS server) queries DNS records for a domain, it stores (caches) the response in memory for a certain amount of time. Subsequent queries for the same domain can be answered from the cache without querying other DNS servers. With **Client-Side Caching,** Operating systems and browsers cache DNS responses locally. This means if a user visits a domain, and then visits it again shortly after, the system can quickly retrieve the IP address from the local cache.

High Level System Design

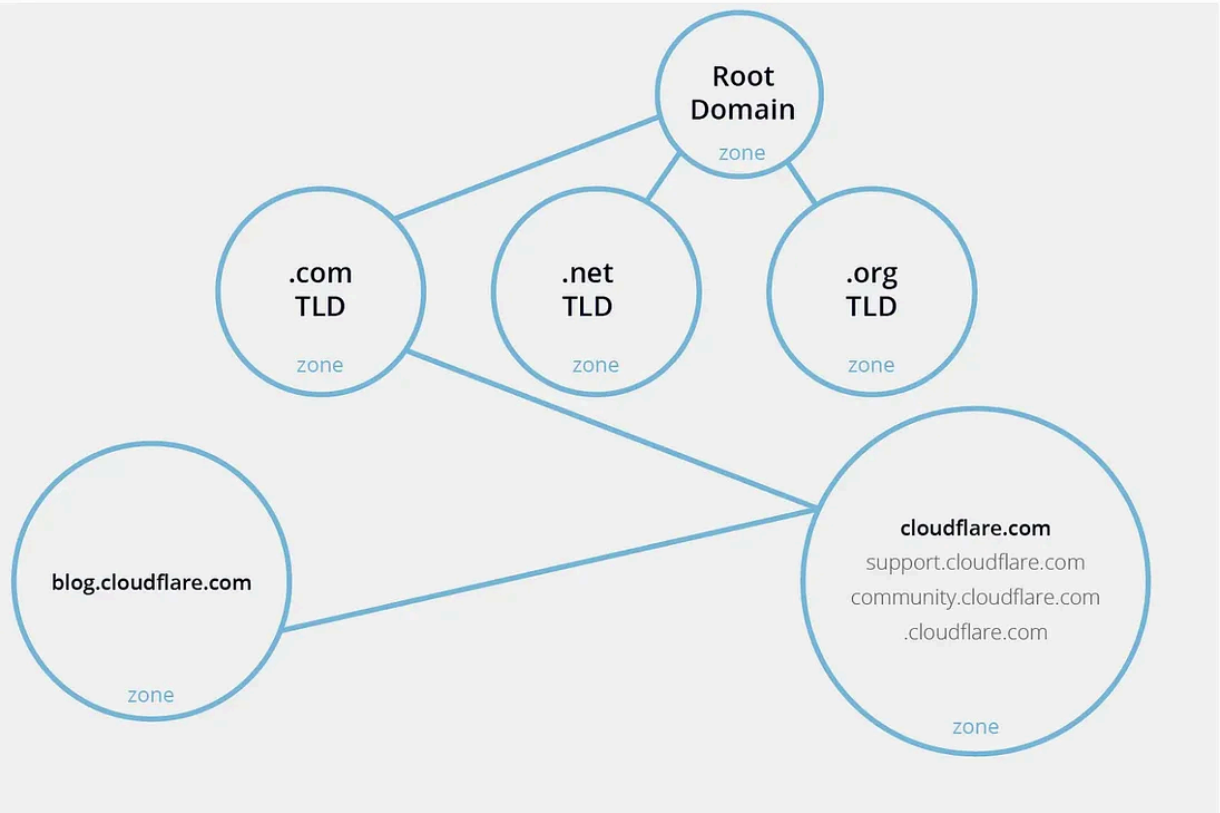
First the DNS query goes to local DNS or internal DNS server (IDS). IDS forwards the request to root level domain server (RLDS) to obtain the ip address. If the RLDS didn't find the ip, then it forwards to top level domain name (TLDS) server, from there to second level domain name server to get the answer.



Once it get the IP address mapped to particular domain name, it returns back to host.

DNS maintains distributed database system for managing host names and their associated Internet Protocol (IP) addresses. DNS data divided into manageable sets of data in the form of **zones**. This distribute the load of maintaining one large DNS server among several DNS servers and improve performance as well as develop a fault-tolerant DNS server.

Different zone will have the different organizational affiliation with appropriate domains, for ex. edu for educational institutions, com for commercial, org for non-profile.



DNS zones are fundamental to the structure and administration of the DNS. They allow for efficient management and delegation of the DNS namespace, ensuring that domain names are resolved accurately and efficiently across the internet.

Each zone maintained a **DNS zone file** which is a text based file that is stored on a DNS name server. This file contains information about mappings between IP addresses, domain names, and other resources, organized in the form of **resource records** (RR). There are two mandatory records which are included at the start of any DNS zone file, they are:

- Start of Authority (SOA) record.
- Global Time to Live (TTL) record.

Apart from these two records, the DNS zone file includes records for all resources described within the zone.

```
$TTL 3600
@    IN    SOA    ns1.example.com. admin.example.com. (
                2024010101 ; Serial
                3600       ; Refresh
                1800       ; Retry
                1209600    ; Expire
                3600 )     ; Minimum TTL

     IN    NS    ns1.example.com.
     IN    NS    ns2.example.com.

     IN    A     192.0.2.1
     IN    AAAA  2001:0db8:85a3:0000:0000:8a2e:0370:7334

www  IN    CNAME example.com.

mail IN    A     192.0.2.2
     IN    MX    10 mail1.example.com.
     IN    MX    20 mail2.example.com.

     IN    TXT   "v=spf1 ip4:192.0.2.1 include:_spf.example.com ~all"
     IN    CAA   0 issue "letsencrypt.org"
```

The DNS (Domain Name System) **root zone** is the top-level DNS zone in the hierarchical namespace of the Domain Name System. It is the starting point for all DNS queries that are not cached or otherwise locally resolved. Root zone of the DNS hierarchy tree is maintained By **IANA** which has database of Top-level domain zone operators. IANA (the Internet Assigned Numbers Authority) manages the global coordination of the DNS (Domain Name System) root, IP addressing, and other Internet protocol resources. One of IANA's responsibilities is managing the DNS root zone, which includes the allocation of top-level domains (TLDs). The TLD name servers that are associated with these domains are distributed into two primary groups:

1. **Generic Top-Level Domains (gTLDs)**: gTLDs are typically used for general purposes and include TLDs like `.com`, `.org`, `.net`, `.info`,

and many others.

2. **Country Code Top-Level Domains (ccTLDs)**: These TLDs are associated with specific countries or territories.

This is called the **DNS Namespace,** which contains DNS records for all the domain names within that zone. Each zone is managed by a specific DNS server or group of servers, allowing for the delegation of DNS administration and the distribution of DNS data across multiple servers.

To ensure reliability, DNS records are often replicated across multiple DNS servers. These servers can be geographically distributed to improve fault tolerance and reduce latency for users around the world.

Authoritative zones which are p**rimary zones** has secondary zone used for backup and redundancy. A **forward zone** is part of recursive zone. A forward lookup zone lets users look up the IP address resolution of a domain from a Domain Name System (DNS) server.

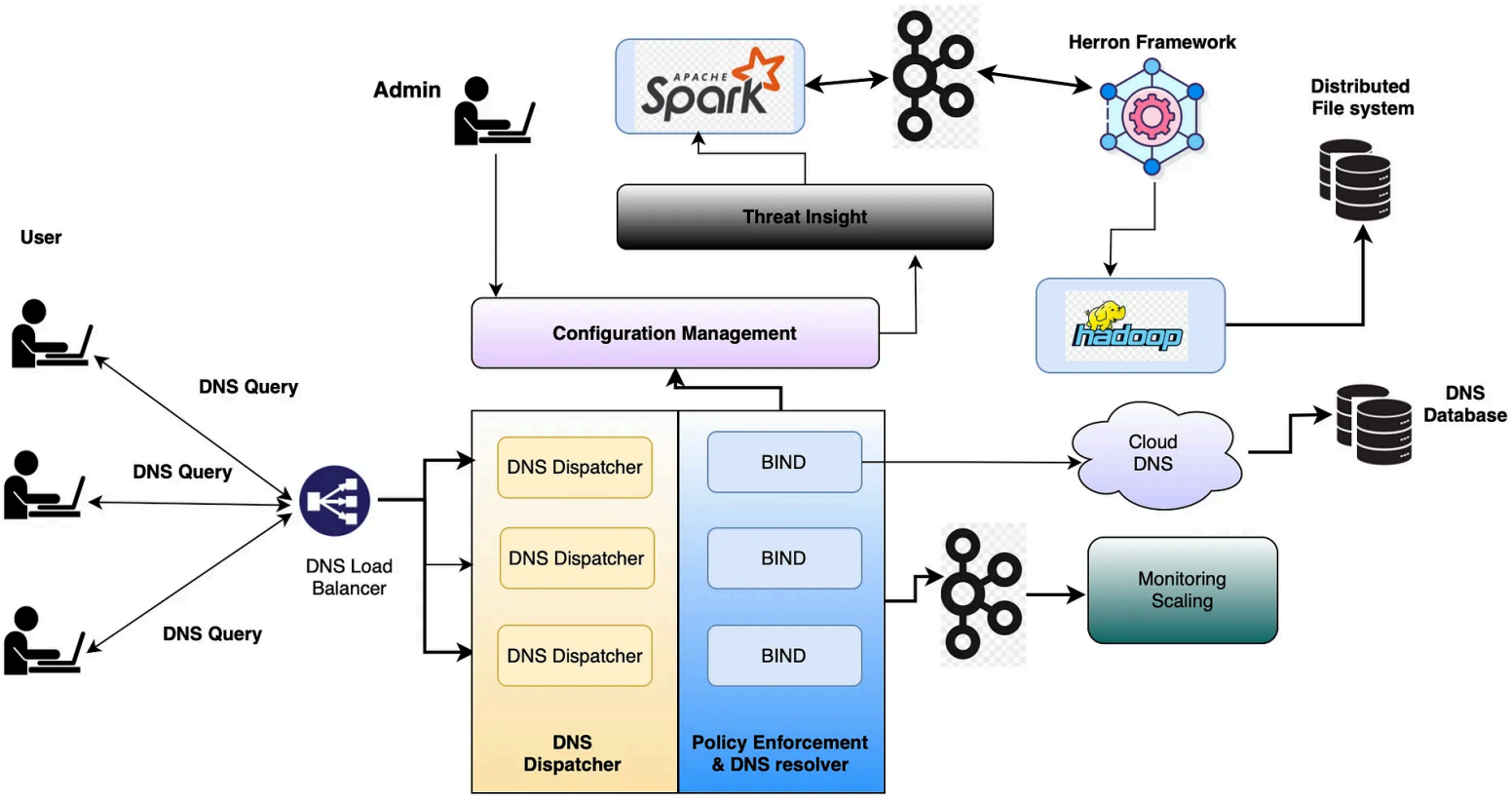
Unbound can do the resolving of DNS queries and cache the results for quicker retrieval. Unbound instances used to handle certain number of policies.

When it comes DNS records shared across multiple domains, **DNS Shared Record Group** has been used to manage and maintain DNS records . This allows for shared management of common DNS records, making it easier to update and maintain consistent configurations across various zones.

Implementing a shared record group involves defining a set of DNS records that can be applied to multiple DNS zones. This way it achieved consistency, efficiency with ease of management of managing DNS records. In BIND, you can use include files to create shared record groups that can be included across multiple zone files.

DNS also maintains a **blacklist ruleset** which has predefined rules used to block or deny access to certain resources, services, or actions based on specific criteria. Blacklists are commonly used in security, network management, and content filtering to prevent malicious activities, unauthorized access, or unwanted content. This is to ensure network/application security with web content filtering.

Low Level design



Lookup for DNS records: In low level design we will look into how the dns records being managed by different domain name servers. **DNS records** are managed using a data structure known as a **Trie** (pronounced “try”),

specifically a variant called a **Radix Tree** or **Patricia Trie**. Trie Structure is efficient for prefix matching. Since domain names are organized hierarchically, a Trie can quickly locate the appropriate node for a given domain name by traversing from the root down to the specific node. DNS zone files, which contain all the DNS records for a domain, can be efficiently represented and managed using a Trie. Since DNS lookups are performed billions of times daily, a data structure like the Trie that supports quick and scalable lookups.

But in the meantime we need a memory efficient solution. To achieve that we have Radix tree, which is a sparse version of a Trie, only uses memory for nodes that actually represent parts of domain names, further optimizing memory usage. It also allows for path compression, where long chains of nodes with single children are collapsed into a single node.

A **DNS load balancer** is a mechanism that distributes network or application traffic across multiple servers using the Domain Name System (DNS).

There are many different schemes being used for load balancing the DNS queries. But selecting proper load balancer depend on it's primary responsibility of load balancing the queries and perform proper health monitoring of DNS servers. For that we have **global server load balancer** , that uses the geographic location of the client to direct them to the closest server or a server in a particular region. This helps reduce latency by directing traffic to a server that is physically closer to the user. With the DNS failover it continuously monitor the health of servers (via health checks). If a server becomes unavailable, the DNS server stops returning its IP address in DNS responses.

DNS can also have combinations of several techniques (geolocation, health checks, and latency-based routing) to ensure traffic is optimally routed based on a variety of factors, including user location, server health, and performance metrics.

Deterministic Traffic Control (DTC) hashing load balancing in DNS is a method used to direct traffic more predictably and efficiently across multiple servers based on a **consistent hashing algorithm.** This method aims to ensure that the same client (or type of request) is always routed to the same server, helping to loadbalance and maintain session consistency.

When a client sends a DNS query for a specific domain, such as `www.example.com`, `t`he DNS server applies a hashing function to a specific piece of data from the client, usually the client's IP address or another identifying factor (such as a session ID). The computed hash value is then mapped to one of the available servers. This is done using a consistent hashing algorithm, which distributes the hash space evenly across the servers. For instance, if there are 4 servers, each server is assigned a range of hash values:

```
Server 1: 0-99
Server 2: 100-199
Server 3: 200-299
Server 4: 300-399
```

This is through range query it finds which server the query need to be sent. The DNS server then responds to the client with the IP address of the mapped server (e.g., `Server 3: 192.0.2.3`).

Each time the same client or request type queries the DNS, the hashing algorithm ensures that it is directed to the same server, as long as the server pool remains unchanged. This maintains session persistence and helps in reducing load imbalance caused by frequent changes. If a server is added or removed, consistent hashing minimizes the impact on the traffic distribution. Only a portion of the clients will be remapped to different servers, rather than all clients being redistributed, which reduces disruption.

In a DNS setup, the relationship between the **Primary (Master) DNS Server** and the **Secondary (Slave) DNS Server** is typically **active-passive** rather than active-active. The primary DNS server is the authoritative source for the DNS zone. It holds the original zone file, which contains the DNS records for the domain. Any changes or updates to the DNS records are made directly on the primary server. The primary server is responsible for sending updates to the secondary servers through a process called zone transfer.

A **DNS zone transfer** is the process of copying the contents of a DNS zone file from one DNS server to another. This is typically done to synchronize DNS records between a primary (master) DNS server and one or more secondary (slave) DNS servers. There are two main types of zone transfers: full (AXFR) and incremental (IXFR).

1. **With the AXFR (Full Zone Transfer) t**ransfers the entire zone file from the primary server to the secondary server.
2. **With IXFR (Incremental Zone Transfer) t**ransfers only the changes (deltas) made to the zone since the last transfer. It's more efficient than AXFR as it reduces the amount of data transferred.

Encryption in DNS zone transfers ensures that the data being transferred between DNS servers remains confidential and secure from eavesdropping or tampering. One of the most effective methods to secure DNS zone transfers is by using **TSIG (Transaction Signature)**, which provides authentication and data integrity through shared secret keys and digital signatures. While TSIG provides authentication and integrity, it doesn't encrypt the data. However, when combined with encrypted channels (e.g., VPN, IPsec, or DNS over TLS), it ensures that the data remains confidential and secure.

The **DNS forwarder** should be thought of as the designated server to which a particular subset of queries (either for external addresses or specific internal

addresses) are forwarded by other DNS servers within the network. Once it receives the response, it forwards it back to the client.

There are two types of DNS forwarder.
1) **Forwarding DNS server**: A DNS server explicitly configured to forward queries that it cannot resolve locally to another DNS server. This configuration is typically done in the DNS server's settings.

2) **Conditional forwarder**: A special type of forwarding DNS server that forwards queries to different DNS servers based on specific domain names.

DNS forwarders can be used in content filtering systems where certain DNS requests are forwarded to filtering servers that block access to certain domains based on policies. DNS forwarding proxy protect our enterprise networks from DNS-based cyber attack. **DNS forwarding proxy** is a specialized type of DNS forwarder that acts as an intermediary between DNS clients (like user devices) and DNS servers. It accepts DNS queries from clients and forwards them to the appropriate DNS servers based on predefined rules or configurations. It can also modify, filter, or log the DNS requests and responses, providing additional functionality compared to a standard DNS forwarder. **Unbound** is an open-source DNS resolver that can act as a DNS forwarding proxy. It supports DNS caching, DNSSEC, and DNS-over-TLS/HTTPS, and is often used in secure network setups.

Bottleneck

Security in DNS (Domain Name System) is critical because DNS is a foundational technology that translates domain names into IP addresses, enabling internet connectivity. DNS can be attacked in many ways, whether through DNS spoofing (cache poisoning) in attackers insert false DNS responses into the cache of a DNS resolver, redirecting users to malicious

sites or DNS Amplification Attacks in which attacker use open DNS resolvers to flood a target with a high volume of DNS response traffic. Attacks on DNS can lead to a wide range of security issues, including man-in-the-middle attacks, data theft, and service disruptions. We can protect DNS in many ways.

DNSsec protect against digitally signed data to ensure data's validity. Signing happens at every level of DNS lookup process. With the DNSsec, DNS should able to filter to find malicious domains and have access control list to allow or deny access to your domain name systems.

Using DNS Policies which is like whitelisting or blacklisting or security categories. One policy per one or more remote office and one policy for all the remote clients.

```
Configuring ACLs on a BIND DNS server.

acl "trusted-nets" {
    192.168.1.0/24;
    10.0.0.0/8;
};
options {
    allow-query { trusted-nets; };
};
```

Using DNScrypt for authenticating the communication between DNS client and full DNS resolver using cryptographic signatures and ensuring secure communication between DNS client and DNS server in an encrypted channel. DNS over HTTPS (DoH) and DNS over TLS (DoT) in which encrypt DNS queries and responses to prevent eavesdropping and tampering. The DNS Resolver module provides user protection for DNS interception and configuration update attacks and improved network performance for DNS resolutions.

DNSSEC Validate (authenticate and integrity check) resource records in signed zones.

```
# Use a DNSSEC tool to generate a key pair for signing the zone.

dnssec-keygen -a RSASHA256 -b 2048 -n ZONE example.com
# Sign the zone file with the generated key.
dnssec-signzone -A -3 randomstring -N INCREMENT -o example.com -t
     example.com.zone
# Add the generated DNSKEY and DS records to the zone file.
example.com. IN DNSKEY 257 3 8 AwEAAb...key...content...
example.com. IN DS 12345 8 1 6db4...hash...content...
# Ensure the DNS server is configured to serve the signed zone.
zone "example.com" {
    type master;
    file "example.com.zone.signed";
};
```

Similarly DNS will be having the **ratelimiter** to address any kind of DDoS kind of attacked that can bring down the entire DNS infrastructure. DNS servers can be configured to limit the number of queries they will respond to from a single IP address within a given timeframe.

```
# Configuring rate limiting on a BIND DNS server.
rate-limit {
    responses-per-second 10;
    window 5;
};
```

CSP (Content Security Policy) is a security feature in web browsers that helps prevent various types of attacks such as cross-site scripting (XSS) and data injection attacks. While CSP itself is not directly related to transferring data, it is often involved in securing the transfer of data by controlling what resources a web page can load and execute.

For transferring data securely, especially in the context of web applications and services, CSP can be part of a larger strategy that includes HTTPS, secure APIs, and proper server configurations. Below, we'll outline how CSP can be used to enhance security during data transfers and provide examples of CSP configurations.

Optimization

> *Response Policy Zone*

When it comes to optimization, we have **RPZ** or **Response Policy Zone**, which is a mechanism used in DNS servers to enforce custom policies on DNS responses. This allows administrators to modify DNS behavior based on predefined rules, often for security purposes such as blocking malicious domains, redirecting traffic, or implementing corporate policies. RPZ is implemented as a specially formatted DNS zone that contains rules for modifying DNS responses. When a DNS server receives a query, it checks the RPZ to see if any policies apply to the domain in question. If a matching rule is found, the DNS server alters the response according to the rule.

> *Reputation scoring in DNS (Domain Name System)*

is a security mechanism used to evaluate and rate the trustworthiness of a domain name based on various factors. The evaluation will be based on: **Domain Age:** Newer domains may be scrutinized more closely as they are often used for short-term attacks. **Domain History:** Checking the domain's past activities, including previous associations with malicious activities. **Content Analysis:** Examining the website's content for signs of phishing, malware, or other harmful activities. **Hosting Infrastructure:** Analyzing the hosting provider, IP addresses, and geographical location. **Behavioral**

Patterns: Identifying patterns that match known attack vectors, such as mass registrations or rapid changes in DNS records.

The scoring of reputation calculated will be by assigning numerical score to each domain based on the collected data and evaluation criteria. We can also apply heuristic rules and machine learning models to improve scoring accuracy over time.

The action will be of blocking or filtering DNS queries to domains with low reputation scores, alerting and notifying administrators or users about potentially harmful domains and placing suspicious domains in a quarantine list for further manual review.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "time"
)

type DomainReputation struct {
    Domain      string  `json:"domain"`
    Reputation  float64 `json:"reputation"`
    IsMalicious bool    `json:"is_malicious"`
}

// Fetch reputation score for a given domain
func getReputation(domain string) (DomainReputation, error) {
    // Simulate fetching data from an external reputation service
    url := fmt.Sprintf("https://reputation.example.com/api/v1/score?domain=%s",
    client := &http.Client{Timeout: 10 * time.Second}
    resp, err := client.Get(url)
    if err != nil {
        return DomainReputation{}, err
    }
    defer resp.Body.Close()

    var reputation DomainReputation
    if err := json.NewDecoder(resp.Body).Decode(&reputation); err != nil {
        return DomainReputation{}, err
    }
```

```
    return reputation, nil
}

func main() {
    domain := "example.com"
    reputation, err := getReputation(domain)
    if err != nil {
        fmt.Printf("Error fetching reputation for domain %s: %v\n", domain, err)
        return
    }

    fmt.Printf("Domain: %s\nReputation Score: %.2f\nIs Malicious: %t\n", reputat

    if reputation.IsMalicious {
        fmt.Println("Warning: This domain is flagged as malicious!")
        // Implement further actions such as blocking the domain
    } else {
        fmt.Println("This domain is safe.")
    }
}
```

This way we enhance security by reducing the risk of phishing attacks, malware infections, and other cyber threats by proactively blocking malicious domains.It also enhances the network performance by Minimizing the impact of malicious traffic on network resources by blocking it at the DNS level. Reputation scoring in DNS is one of the powerful technique for enhancing network security by evaluating the trustworthiness of domains based on various criteria.

> *DNS Query for Threat Hunting*

DNS queries is a powerful tool for threat hunting, allowing security analysts to detect and investigate suspicious domain names and IP addresses. Threat hunting involves proactively searching for threats that may evade automated detection systems. Using DNS queries, we can gather valuable information about domains, such as their associated IP addresses, mail servers, canonical names, and text records.

Create a text file (e.g., `suspicious_domains.txt`) with each suspicious domain on a new line.

```
example.com
malicious-domain.com
suspicious-domain.org
```

```
package main

import (
    "encoding/json"
    "fmt"
    "net"
    "os"
    "strings"
)

type DNSInfo struct {
    Domain       string
    IPAddresses  []string
    MXRecords    []string
    CNAME        string
    TXTRecords   []string
}

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: go run main.go <file_with_domains>")
        os.Exit(1)
    }

    filePath := os.Args[1]
    domains, err := readDomainsFromFile(filePath)
    if err != nil {
        fmt.Printf("Error reading domains from file: %v\n", err)
        os.Exit(1)
    }

    var results []DNSInfo
    for _, domain := range domains {
        info := gatherDNSInfo(domain)
        results = append(results, info)
    }

    output, err := json.MarshalIndent(results, "", "  ")
    if err != nil {
        fmt.Printf("Error marshaling results: %v\n", err)
```

```
            os.Exit(1)
        }

        fmt.Println(string(output))
    }

    func readDomainsFromFile(filePath string) ([]string, error) {
        content, err := os.ReadFile(filePath)
        if err != nil {
            return nil, err
        }
        return strings.Split(strings.TrimSpace(string(content)), "\n"), nil
    }

    func gatherDNSInfo(domain string) DNSInfo {
        var info DNSInfo
        info.Domain = domain

        // Perform A/AAAA record lookup
        ipRecords, err := net.LookupIP(domain)
        if err == nil {
            for _, ip := range ipRecords {
                info.IPAddresses = append(info.IPAddresses, ip.String())
            }
        }

        // Perform MX record lookup
        mxRecords, err := net.LookupMX(domain)
        if err == nil {
            for _, mx := range mxRecords {
                info.MXRecords = append(info.MXRecords, fmt.Sprintf("%s (priority %d
            }
        }

        // Perform CNAME record lookup
        cname, err := net.LookupCNAME(domain)
        if err == nil {
            info.CNAME = cname
        }

        // Perform TXT record lookup
        txtRecords, err := net.LookupTXT(domain)
        if err == nil {
            info.TXTRecords = append(info.TXTRecords, txtRecords...)
        }

        return info
    }
```

Logging and Monitoring

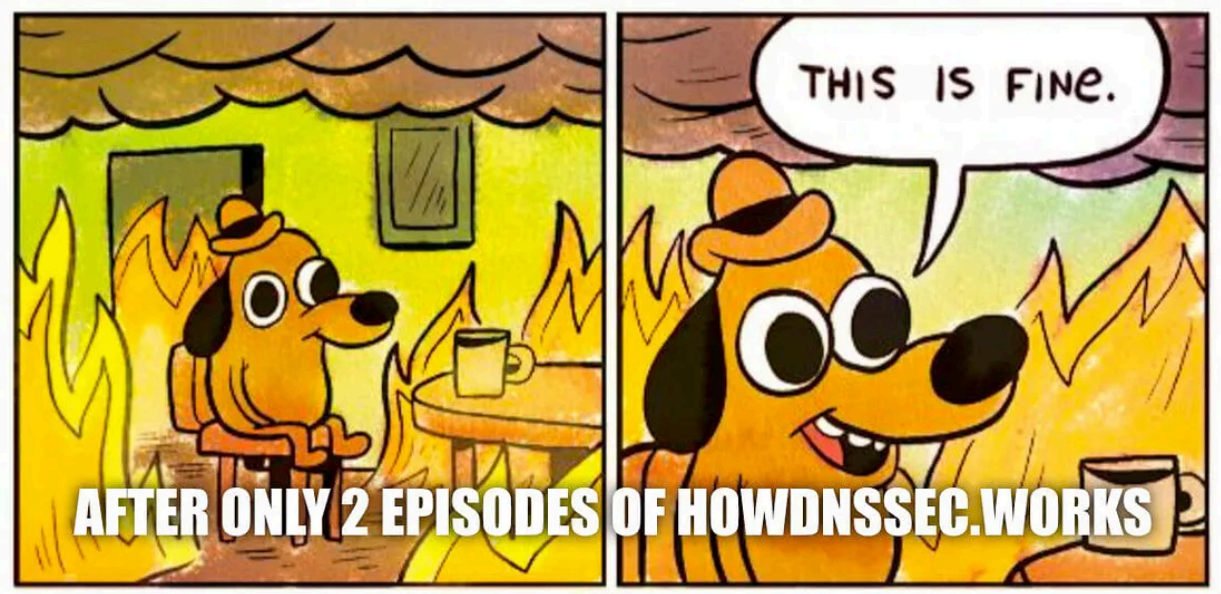
dnstap is a flexible, structured binary logging system for DNS software. It provides a mechanism to log DNS queries and responses in a detailed and structured format, which can be useful for debugging, monitoring, and security analysis.

Dnstap is a method for high-performance, structured logging of DNS server traffic. It allows DNS servers to log details of DNS transactions in a binary format, which can be more efficient and informative than traditional text-based logging.

Generating the dsn query and responses log information is computationally expensive and to maintain the DNS QPS rate at it's peak, DNS Q/R log messages offloaded from the DNS server. Offloading done using dnstap. these logs have being collected and transported from cloud in the form of syslog.

All the DNS requests of log or query has been encrypted when passing between on-premises or could setup.

- DNS servers generate dnstap logs for various events (e.g., received queries, sent responses).
- Transport: Logs can be written to a file, sent to a remote server, or processed in real-time.



Written by Santosh P.

196 followers · 79 following

A Developer | Aspiring Distributed Computing Expert | Leveraging Algorithms & Data Structures for Optimal Performance | Passionate Techie

Follow

No responses yet

Raihanur Rahman

What are your thoughts?