DNS Proxy Server Design Document

Overview

The DNS Proxy Server is a specialized application designed to enhance network resilience abandoning all DNS functions that lead to system fragility. Its core functionality revolves around caching DNS query responses, serving stale data during upstream DNS outages, and periodically checking the health of the upstream DNS server. This document outlines the application's architecture, components, functionalities, and operational logic.

Purpose

The primary purpose of the DNS Proxy Server is to ensure continuous DNS resolution within a network, particularly in scenarios where the upstream DNS server might become temporarily unavailable. By serving cached and potentially stale DNS data during these outages, the application aims to minimize the impact of DNS failures on end-users and dependent services.

Functional Requirements

DNS Query Handling: Intercept and resolve DNS queries from clients by forwarding them to an upstream DNS server.

Caching: Cache DNS query responses along with their Time to Live (TTL) values to reduce latency and load on the upstream server.

Serve Stale Data: In the event of an upstream DNS failure, serve stale cached data beyond its original TTL, up to a predefined extended period.

Health Checks: Periodically check the availability of the upstream DNS server and refresh the cache with updated data when possible.

Logging: Maintain logs for monitoring DNS query handling, cache operations, and health check results. (not incorporated yet)

Architecture

Standalone:

The DNS Proxy Server is implemented as a single, standalone application with the following key components:

DNS Query Listener: Listens for DNS queries on port 53 (both UDP and TCP) from client devices.

Cache Manager: Manages the storage, retrieval, and expiration of cached DNS query responses.

Upstream Query Forwarder: Forwards DNS queries to the configured upstream DNS server and handles responses.

Health Checker: Periodically sends test queries to the upstream DNS server to check its availability.

Logger: Captures and stores logs related to DNS query processing, caching operations, and health check outcomes. (not implemented yet)

Kubernetes Implementation of DNS Proxy Server

The DNS Proxy Server is deployed to a Kubernetes cluster using a Deployment resource. This approach allows the application to be managed as a scalable and updateable entity within the Kubernetes ecosystem. The Deployment manages pods that run the DNS Proxy Server containers, maintaining the desired number of replicas to ensure availability and load distribution.

Key Components:

Pods: Run instances of the DNS Proxy Server container. Each pod encapsulates the application, running it in an isolated environment while sharing the same network namespace.

Deployment: Manages the creation, scaling, and updating of pods. It ensures that a specified number of pod replicas are running at any given time, facilitating rollouts and rollbacks as necessary.

Service: Defines a logical set of pods and a policy by which to access them. For the DNS Proxy Server, a Service of type ClusterIP (default) or LoadBalancer for external access is used to expose the application on the necessary DNS port (53) to other services within the cluster or external clients.

Configuration and Management

ConfigMaps and Secrets:

ConfigMaps: Used to store non-sensitive configuration options for the DNS Proxy Server, such as upstream DNS server addresses. This allows for easy configuration changes without the need to rebuild the container image.

Health Checks:

Kubernetes supports liveness and readiness probes to monitor the health of applications running in pods. For the DNS Proxy Server, these probes can be configured to perform health checks, ensuring the application is functioning correctly and is able to serve DNS queries.

Liveness Probe: Ensures the DNS Proxy Server is running; if the probe fails, Kubernetes restarts the pod automatically.

Readiness Probe: Checks if the DNS Proxy Server is ready to handle DNS queries. Traffic is only routed to the pod when the readiness probe passes.

Scalability and High Availability

Kubernetes inherently supports scaling and high availability

Network Policies

Network Policies: Define how pods are allowed to communicate with each other and with other network endpoints. For the DNS Proxy Server, network policies ensure secure and controlled network traffic flow, preventing unauthorized access to DNS services.

Operational Flow

DNS Query Reception: The DNS Query Listener receives a DNS query from a client.

Cache Lookup: The Cache Manager checks if the query's response is cached.

If a valid cached response exists (i.e., within its TTL), it is immediately returned to the client.

If a cached response exists but its TTL has expired, the application checks the upstream DNS server's availability.

Upstream Forwarding:

If the upstream DNS server is available, the query is forwarded, and the response is cached with a new TTL.

If the upstream server is unavailable, the application decides based on the stale data serving policy:

Serve stale data if within the extended stale period.

Return a failure response if beyond the stale period or no cached data exists.

Periodic Health Checks: Independently, the Health Checker periodically queries the upstream DNS server to assess its availability. The outcome influences future decisions on serving stale data and refreshing the cache.

Technologies

Programming Language: Implemented in Go for its excellent support for concurrent network and I/O operations.

External Libraries: Uses miekg/dns for handling DNS protocols and operations.

Challenges and Considerations

Security: Implementing DNSSEC validation to enhance security and prevent cache poisoning.

Scalability: Ensuring the application and its caching mechanism can scale to handle high volumes of DNS queries.

Stale Data Accuracy: Balancing the benefits of serving stale data with the risks of using potentially outdated information.

Health Check Strategy: Designing an efficient and reliable health check mechanism that minimally impacts the upstream DNS server.

Future Enhancements

Adaptive Caching: Dynamically adjusting caching strategies based on query patterns and server load.

Load Balancing: Introducing load balancing capabilities for queries among multiple upstream servers.

Advanced Monitoring: Enhancing the logging and monitoring features to provide real-time insights into application performance and DNS query trends.

Logging implementation, code written but use case is bespoke per environment (one size does not fit all)

Conclusion

The DNS Proxy Server is designed to provide a resilient and efficient DNS query resolution service, minimizing the impact of upstream DNS server outages and ensuring continuous network operation. Through its caching and stale data serving capabilities, coupled with periodic health checks, the application aims to deliver a seamless and reliable DNS experience.

<code>

package main

import (

"github.com/miekg/dns"

"log"

"sync"

"time"

)

var (

upstreamDNS = "8.8.8.8:53" // Example: Google's Public DNS

isUpstreamAvailable bool // Global flag to indicate upstream DNS status

cache \*DNSCache

)

type CacheEntry struct {

Response \*dns.Msg

Expires time.Time

StalePeriod time.Time

}

type DNSCache struct {

cache map[string]\*CacheEntry

mu sync.RWMutex

}

func NewDNSCache() \*DNSCache {

return &DNSCache{

cache: make(map[string]\*CacheEntry),

}

}

func (c \*DNSCache) Set(key string, response \*dns.Msg, ttl time.Duration) {

c.mu.Lock()

defer c.mu.Unlock()

now := time.Now()

c.cache[key] = &CacheEntry{

Response: response,

Expires: now.Add(ttl),

StalePeriod: now.Add(24 \* time.Hour), // Using a fixed 24-hour period for stale data

}

}

func (c \*DNSCache) Get(key string) (\*dns.Msg, bool) {

c.mu.RLock()

defer c.mu.RUnlock()

entry, found := c.cache[key]

if !found {

return nil, false

}

now := time.Now()

if now.Before(entry.Expires) || (now.Before(entry.StalePeriod) && !isUpstreamAvailable) {

return entry.Response, true

}

return nil, false

}

func handleDNSQuery(w dns.ResponseWriter, r \*dns.Msg) {

key := r.Question[0].Name + dns.TypeToString[r.Question[0].Qtype]

if response, found := cache.Get(key); found {

response.SetReply(r)

w.WriteMsg(response)

return

}

if !isUpstreamAvailable {

dns.HandleFailed(w, r)

return

}

// Forward the query to the upstream DNS server

c := new(dns.Client)

response, \_, err := c.Exchange(r, upstreamDNS)

if err != nil {

log.Printf("Failed to reach upstream DNS: %v", err)

isUpstreamAvailable = false

dns.HandleFailed(w, r)

return

}

cache.Set(key, response, 5\*time.Minute) // Cache with a TTL of 5 minutes

w.WriteMsg(response)

}

func checkUpstreamDNS() {

c := new(dns.Client)

m := new(dns.Msg)

m.SetQuestion(dns.Fqdn("example.com"), dns.TypeA)

\_, \_, err := c.Exchange(m, upstreamDNS)

if err != nil {

log.Println("Upstream DNS check failed:", err)

isUpstreamAvailable = false

} else {

log.Println("Upstream DNS is available")

isUpstreamAvailable = true

}

}

func scheduleHealthCheck(interval time.Duration) {

ticker := time.NewTicker(interval)

go func() {

for {

select {

case <-ticker.C:

checkUpstreamDNS()

}

}

}()

}

func main() {

// Initialize the DNS cache

cache = NewDNSCache()

// Schedule upstream DNS health checks every 5 minutes

scheduleHealthCheck(5 \* time.Minute)

// Setup DNS server

dns.HandleFunc(".", handleDNSQuery)

server := &dns.Server{Addr: "127.0.0.1:53", Net: "udp"}

log.Println("Starting DNS proxy server...")

if err := server.ListenAndServe(); err != nil {

log.Fatalf("Failed to start server: %v\n", err)

}

}

Unimplemented Logging code:

func handleDNSQuery(w dns.ResponseWriter, r \*dns.Msg) {

// Extract client's IP address and query details

clientIP := w.RemoteAddr().String()

queryName := r.Question[0].Name

queryType := dns.TypeToString[r.Question[0].Qtype]

// Log the received query

log.Printf("Received query from %s: %s %s\n", clientIP, queryName, queryType)

key := queryName + queryType

if response, found := cache.Get(key); found {

// Log cache hit

log.Printf("Cache hit for %s %s\n", queryName, queryType)

response.SetReply(r)

w.WriteMsg(response)

return

}

if !isUpstreamAvailable {

// Log serving stale if applicable

log.Printf("Serving stale (if available) for %s %s\n", queryName, queryType)

dns.HandleFailed(w, r)

return

}

// Forward the query to the upstream DNS server

c := new(dns.Client)

response, \_, err := c.Exchange(r, upstreamDNS)

if err != nil {

log.Printf("Failed to reach upstream DNS for %s %s: %v\n", queryName, queryType, err)

isUpstreamAvailable = false

dns.HandleFailed(w, r)

return

}

// Log successful upstream query

log.Printf("Upstream resolved %s %s\n", queryName, queryType)

cache.Set(key, response, 5\*time.Minute) // Cache with a TTL of 5 minutes

w.WriteMsg(response)

}

Dockerfile  
  
# Builder stage

FROM golang:1.18 as builder

# Create app directory

WORKDIR /app

# Copy the go mod and sum files

COPY go.mod go.sum ./

# Download Go modules

RUN go mod download

# Copy the source code

COPY . .

# Build the application

RUN CGO\_ENABLED=0 go build -o dnsproxy .

# Use `setcap` to allow the binary to bind to well-known ports as a non-root user

# Note: This step might fail in environments where `setcap` is not available or has restricted functionality.

# If you encounter issues, you may need to adjust your base image or set capabilities at runtime or deployment.

RUN apt-get update && apt-get install -y libcap2-bin && setcap 'cap\_net\_bind\_service=+ep' /app/dnsproxy

# Final stage

FROM gcr.io/distroless/base-debian11

# Copy the binary from the builder stage

COPY --from=builder /app/dnsproxy /app/dnsproxy

# Non-root user configuration

USER nonroot:nonroot

# Expose DNS port

EXPOSE 53/udp

EXPOSE 53/tcp

# Run the application

ENTRYPOINT ["/app/dnsproxy"]