OnPrem CDN Project Overview

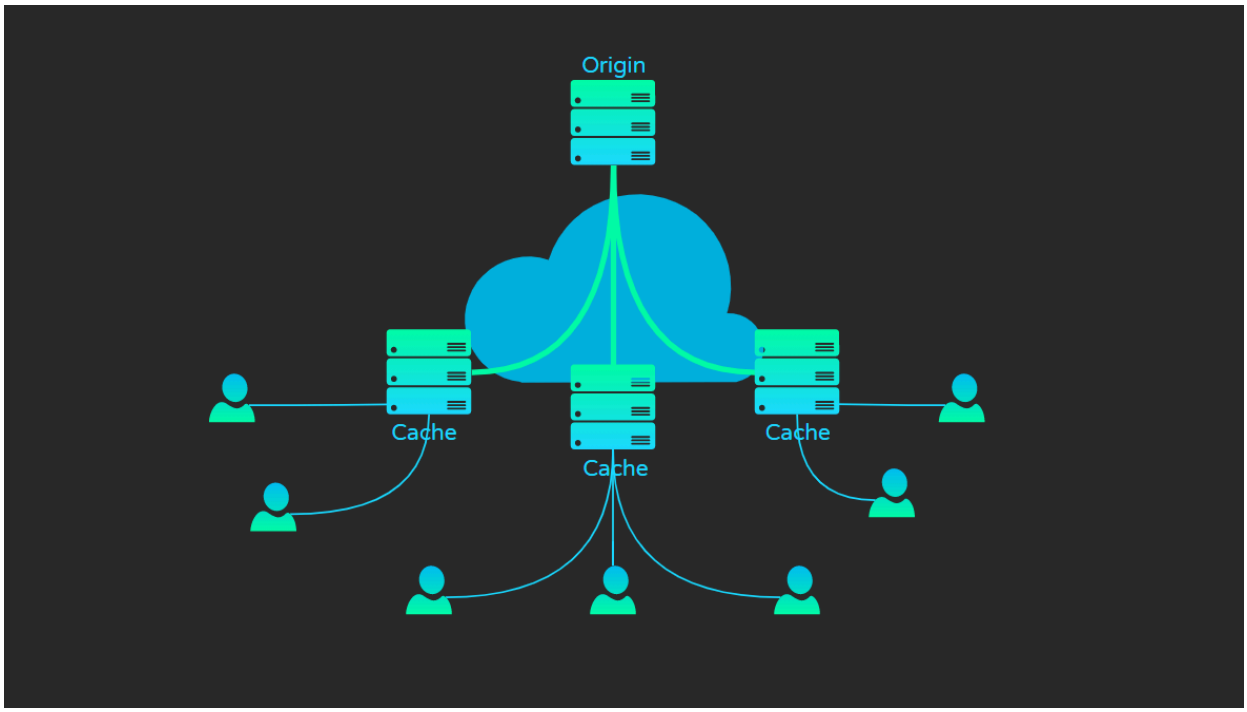
Introduction

The purpose of this project document is to gain a further understanding of how the “internet” works. Specifically, the following content will focus on the impact of Content Delivery Networks (CDN). CDNs currently dominate most of the internet landscape, with most sites utilizing their services or organizations operating their own. While CDNs have matured to provide numerous product offerings, their initial claim to fame was content caching, which improved

the loading times of static and dynamic content. This document will focus on the behavior of how to improve content loading times. Most individuals would leave this kind of burden to the CDN provider, but in this document, a more painful approach will be investigated.

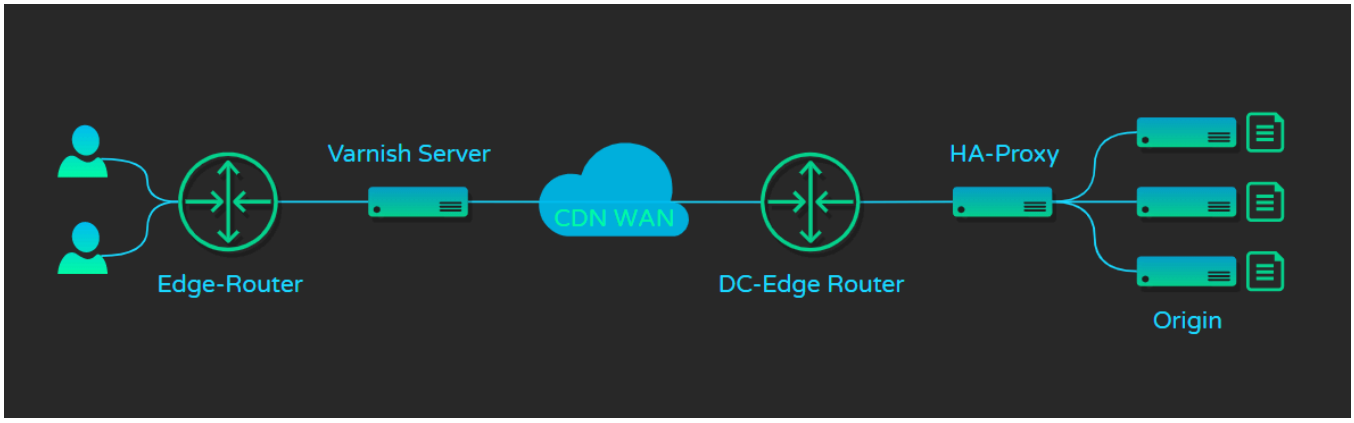
High-Level Design Overview

As mentioned in the introduction, the objective of a CDN is to reduce latency, which is achieved by placing content closer to the end user. Multiple components at play will be reviewed in further detail within their respective sections. This section will serve as a high-level explanation of the topology.



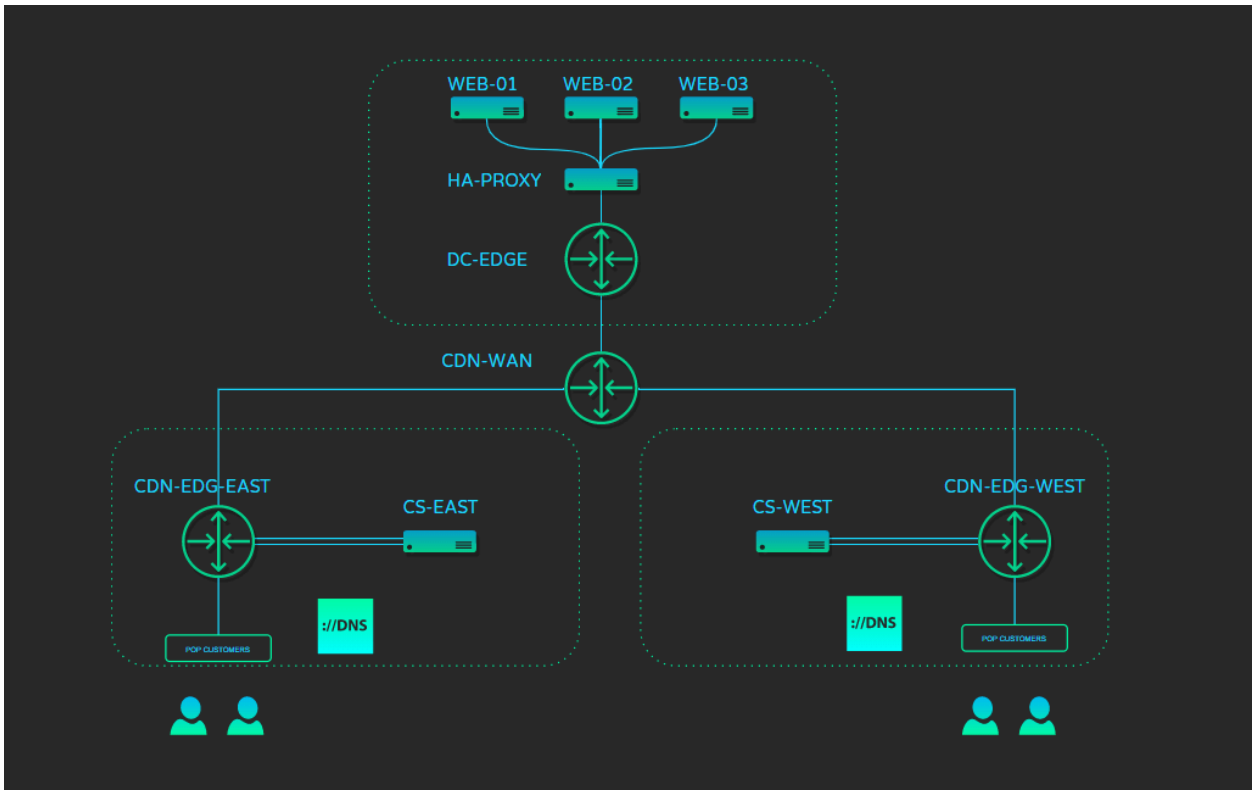
Caching server distribution from the origin

This oversimplified depiction resembles what is seen in the marketing world as a deployment of servers "*at the edge,"* which allows content to be pulled and stored as needed. This reduces latency as customer requests do not need to traverse multiple hops to retrieve content from the origin location. While that lengthy trip can still happen, it is usually only required for content not requested frequently. A CDN is comprised of thousands of nodes dispersed strategically at high-traffic locations. This lab will focus on the network, DNS, cache server, and load balancing requirements at a functional level.



End-to-end connectivity from end user to origin

Reviewing the diagram above, the flow becomes easier to understand, albeit grossly simplified compared to reality. The first server encountered by the client is a cache server (After DNS resolution). Varnish is an open-source software that enables HTTP caching. The following server is an HA-Proxy server that performs simple load balancing. Customer-originated traffic may never make it this far into the network, depending on the loaded page. The cache would hopefully already have stored all the needed information. Finally, all routers used in this lab are VyOS VMs. The VyOS node allows high throughput without limitation and enables WAN emulation features (Latency and Error Rate).



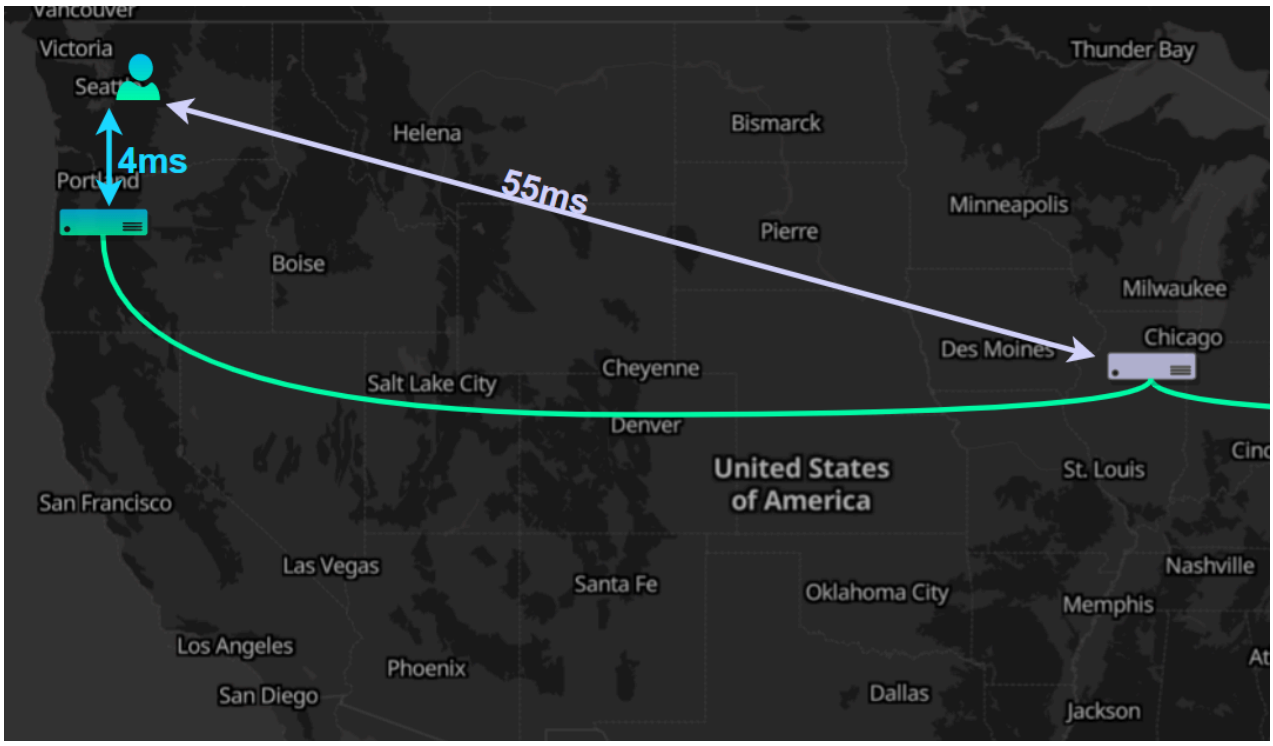
Regional node distribution

The lab will consist of two regions, east and west, with one data center to house the origin content. The website found on WEB-01 through 03 will be an elementary NGINX-hosted page with large image and video files to demonstrate the benefit of caches. The DNS servers are Lubuntu machines running FRR and bind9. In the coming sections, each component and its responsibility will be investigated in greater detail.

Network

Overview

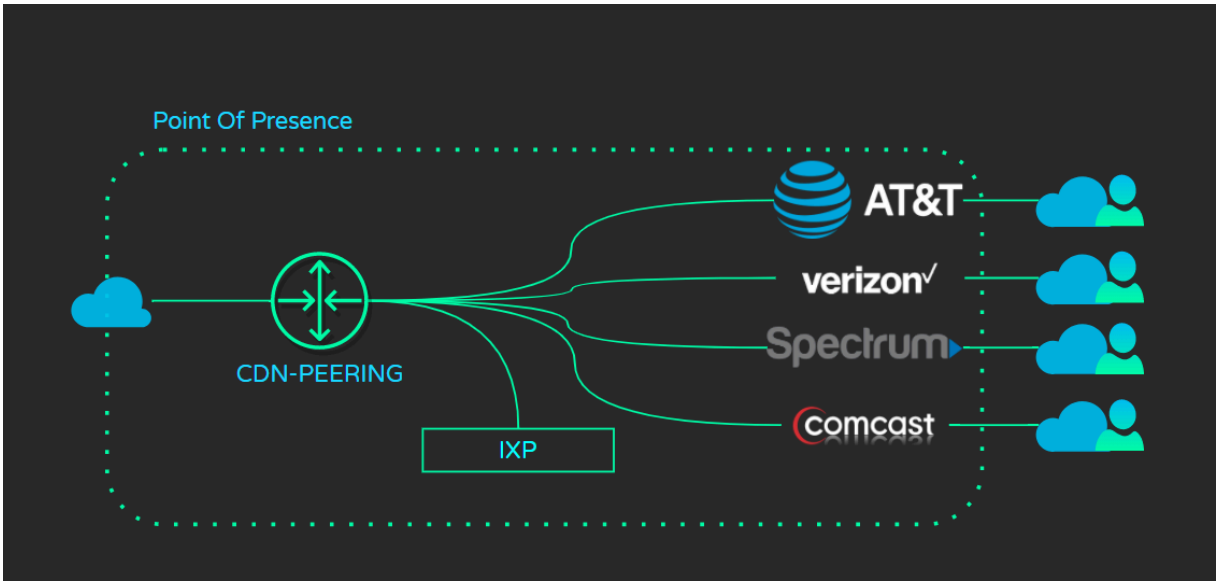
CDN networks are globally connected, highly available, and designed for fast and efficient traffic flow. While this may sound like the objective of many networks today, CDN networks do this very well at scale. A CDN network's value begins at the edge, where it relies on presence to provide the closest available ingress to its network. The diagram below visualizes the importance of edge presence within CDN operation.



Latency mitigation via CDN

Without using a CDN, the end user must retrieve all information from the origin in Chicago, which is a 55ms trip from Seattle. The more content this user has to retrieve, the longer they will continue to wait for page load completion. The second option is to utilize cached information at a local point of presence, reducing the trip time to 4ms. The ping values used for this discussion were retrieved via https://wondernetwork.com/pings. A point of presence, or “POP” is the prevailing architecture of content providers in the modern network landscape. A POP is an

almost slang-like term for what would commonly be referred to as a data center. The data centers that CDNs use for points of presence will usually be large and well-connected.

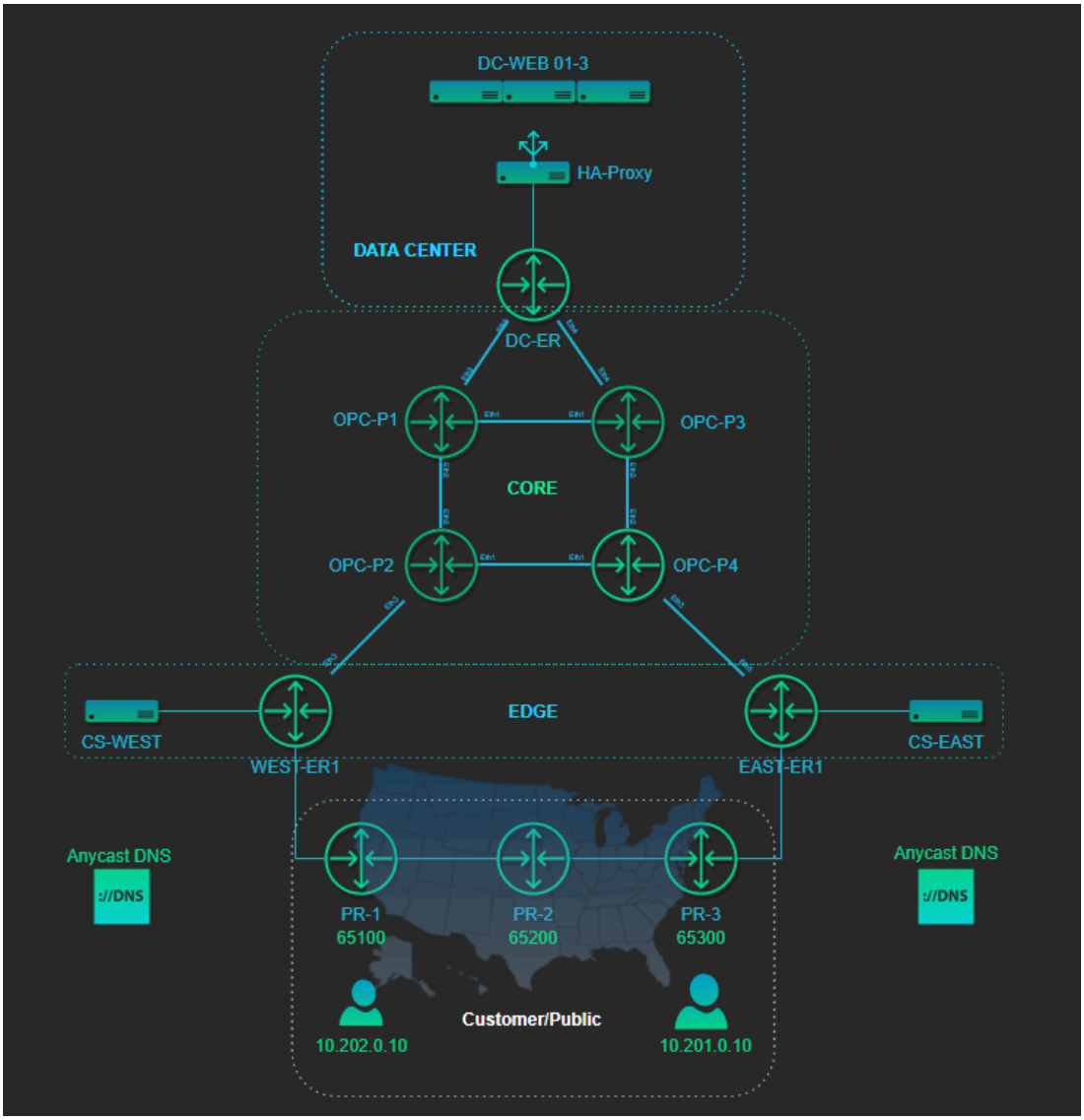


CDN strategic peering

The appeal of large data centers such as Equinix locations is the ease of interconnection. CDNs can connect directly to some of the largest ISPs and extend their reach through Internet Exchange Point (IXP) peering. Not all providers peer happily within the traditional IXP framework, hence the reason for the distinction within the diagram. With this configuration in play, CDN traffic can go directly to the network needed. By peering purposefully, CDN traffic can take the shortest path when receiving and delivering content to/from end users.

The last major network component is how clients get directed to the nearest POP. In this initial lab, simplified Anycast DNS will be utilized. This method relies on the Internet landscape to properly direct clients to the closest name server, which will hold the record most appropriate for that region. In the *real world,* this is more complex. CDNs allow the use of GeoLocation IP databases to direct these requests further. This will be beyond the scope of this document, but Amazon Web Services Route 53 is an excellent example of this option. This article also lays out the DNS portion very well. With the fundamental objective outlined, the focus can move toward the implementation side. This lab has many moving pieces that may not be overly familiar to network engineers; at least, that is what I told myself as I struggled to build it.

Architecture

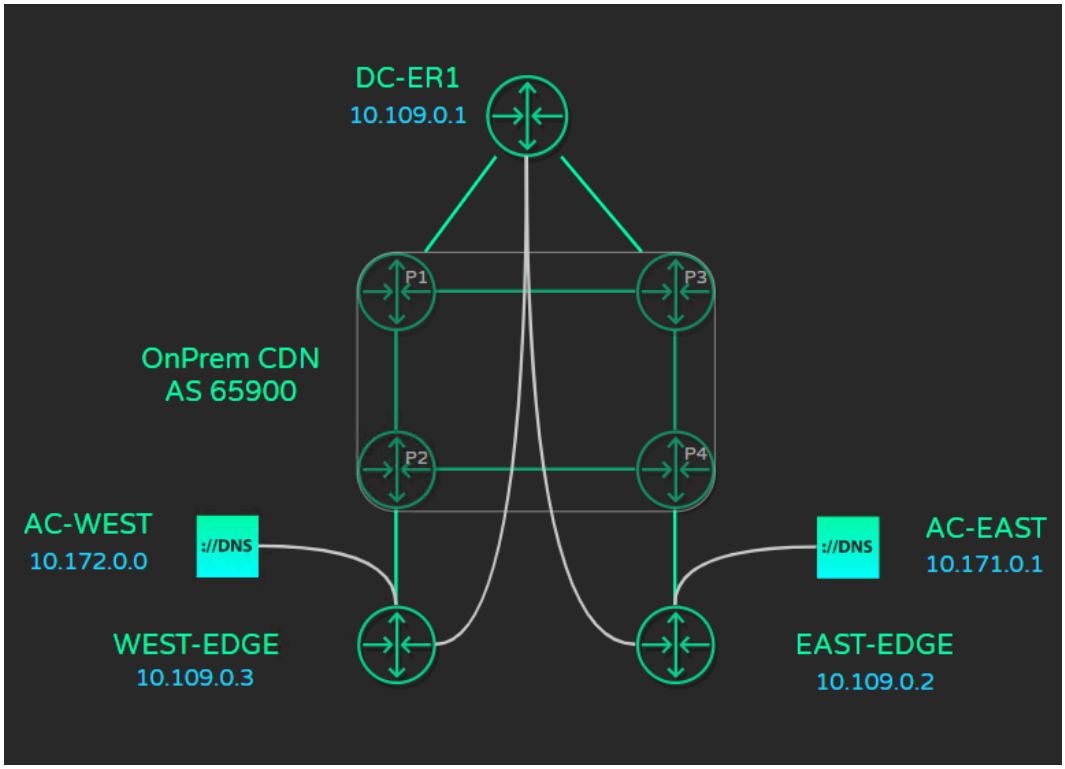


Lab Architecture

The CDN internal network will run a BGP-free core using MPLS LDP labeling and OSPF as the chosen IGP. BGP peering relationships will be formed between the edge units and the DC router (WEST-ER1, EAST-ER1, and DC-ER1). The DNS servers will also be BGP enabled to aid in the anycast function and will peer to DC-ER1(DC1 also operates as a route-reflector). Southbound of the edge nodes will be handled by vanilla eBGP for the emulated public network. Two customer networks are advertised into the public realm.

Routing

The CDN network will operate under one BGP AS of 65900. OSPF will be utilized as the IGP to provide loopback reachability for iBGP connections spanning edge locations. The DC router will operate as a route-reflector for the edge devices.



Reviewing the ~~Juniper~~… VyOS router configuration displays a basic OSPF and BGP configuration.

```
vyos@DC-R1:~$ sh configuration commands | match bgp
set protocols bgp neighbor 10.109.0.2 address-family ipv4-unicast route-reflector-client
set protocols bgp neighbor 10.109.0.2 remote-as '65900'
set protocols bgp neighbor 10.109.0.2 update-source 'lo'
set protocols bgp neighbor 10.109.0.3 address-family ipv4-unicast route-reflector-client
set protocols bgp neighbor 10.109.0.3 remote-as '65900'
set protocols bgp neighbor 10.109.0.3 update-source 'lo'
set protocols bgp neighbor 10.171.0.1 address-family ipv4-unicast route-reflector-client
set protocols bgp neighbor 10.171.0.1 remote-as '65900'
set protocols bgp neighbor 10.171.0.1 update-source 'lo'
set protocols bgp neighbor 10.172.0.0 address-family ipv4-unicast route-reflector-client
set protocols bgp neighbor 10.172.0.0 remote-as '65900'
set protocols bgp neighbor 10.172.0.0 update-source 'lo'
set protocols bgp system-as '65900'
```

DC-ER BGP Configuration

```
vyos@OCP-EAST-EDGE:~$ sh configuration commands | match bgp
set protocols bgp neighbor 10.109.0.1 address-family ipv4-unicast nexthop-self
set protocols bgp neighbor 10.109.0.1 remote-as '65900'
set protocols bgp neighbor 10.109.0.1 update-source 'lo'
set protocols bgp system-as '65900'
```

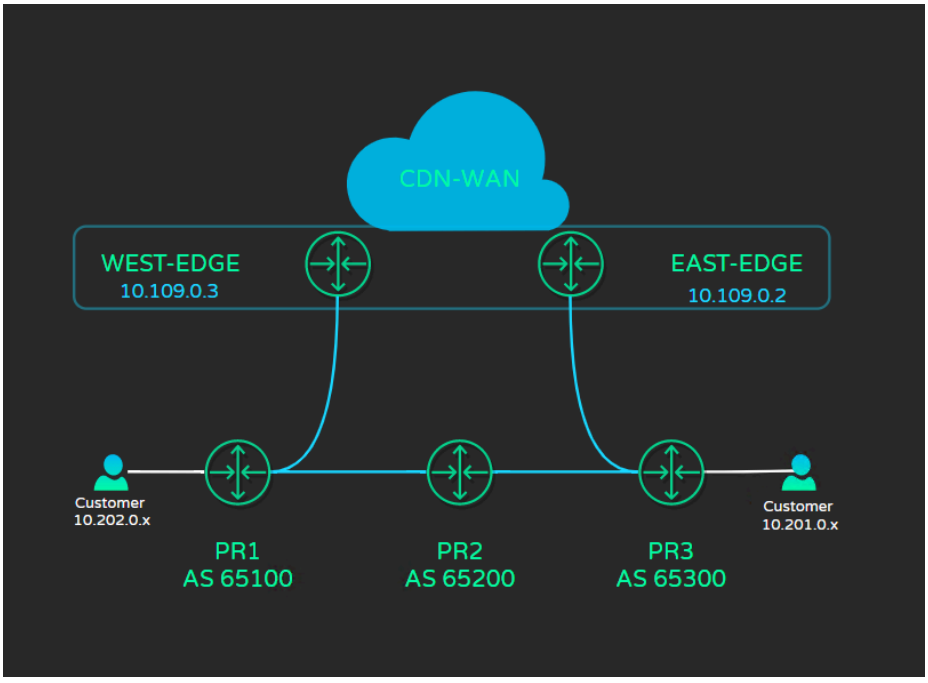
Edge Node BGP Configuration

All core-facing interfaces will run in point-to-point mode for OSPF, and the loopbacks will be advertised into OSPF for BGP. For MPLS, simply enable it on all core and core-facing interfaces.

```
vyos@OPC-P2:~$ show configuration commands | match ospf
set protocols ospf interface eth1 area '0'
set protocols ospf interface eth1 network 'point-to-point'
set protocols ospf interface eth2 area '0'
set protocols ospf interface eth2 network 'point-to-point'
set protocols ospf interface eth4 area '0'
set protocols ospf interface eth4 network 'point-to-point'
set protocols ospf interface lo area '0'
set protocols mpls interface <if#>
set protocols mpls ldp interface <if#>
set protocols mpls ldp discovery transport-ipv4-address '10.109.0.2'
set protocols mpls ldp router-id '10.109.0.2'
```

OSPF Interface Specific Configuration

The internal connectivity does not hold many surprises as this lab is more focused on software use rather than network nerd knobs. Focusing attention outside the CDN network, a simple three autonomous system chain has been created to provide AS PATH simulation on the public network.



Public Routing Topology

A single link spans between the BGP peers, allowing all routes to pass. PR1 and PR2 house the external customer connectivity and advertise their respective networks.

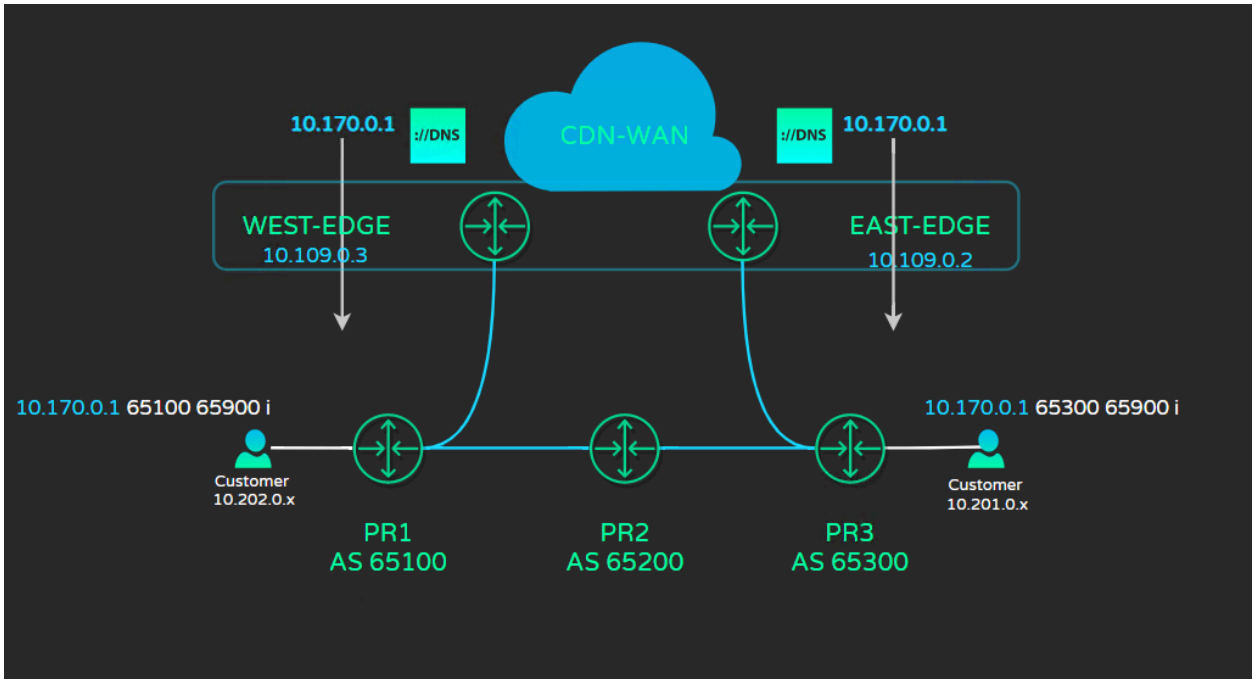
```
vyos@PR1:~$ show configuration commands | match bg
set protocols bgp address-family ipv4-unicast network 10.202.0.0/24
set protocols bgp neighbor 10.60.0.2 address-family ipv4-unicast
set protocols bgp neighbor 10.60.0.2 description 'AS65200_PEER'
set protocols bgp neighbor 10.60.0.2 remote-as '65200'
set protocols bgp neighbor 10.191.0.1 address-family ipv4-unicast
set protocols bgp neighbor 10.191.0.1 description 'OPC_WEST_PEER'
set protocols bgp neighbor 10.191.0.1 remote-as '65900'
set protocols bgp system-as '65100'
```

Public Router Config

The few internal assets within the CDN network are advertised via OSPF passive interfaces. Then, the necessary prefixes are picked up via eBGP network commands and placed into the public realm.

DNS Server BGP (FRR)

The implementation that may be unfamiliar to the everyday network engineer (me) is running BGP on a DNS server. This becomes crucial for the anycast ability. With BGP, the server can actively advertise the anycast IP and withdraw itself upon failure. This allows traffic to continue to flow to the next available path. CDN anycast nodes are plentiful and highly redundant.



DNS BGP Advertisement Topology

BGP will distribute both routes and allow the local router to determine the best path based on the Best Path algorithm or locally configured policy. Everything in the lab is in the default state, leaving the routing decision solely determined by AS PATH.

```
vyos@PR1:~$ sh ip bgp ipv4 unicast 10.170.0.1
BGP routing table entry for 10.170.0.1/32, version 9
Paths: (2 available, best #1, table default)
  Advertised to non peer-group peers:
  10.60.0.2 10.191.0.1
  65900
        10.191.0.1 from 10.191.0.1 (10.109.0.3)
        Origin IGP, valid, external, best (AS Path)
        Last update: Wed Apr 17 16:41:53 2024
  65200 65300 65900
        10.60.0.2 from 10.60.0.2 (10.60.0.1)
        Origin IGP, valid, external
        Last update: Wed Apr 17 16:39:28 2024
```

Public Router 1 Received Anycast Routes

This configuration is all made possible via Free Range Routing (FRR) for Linux. FRR makes this deployment painless and enables a familiar CLI to configure. I also attempted to utilize BIRD routing, another Linux BGP daemon, resulting in frustration and lost time. With FRR, I completed the installation, peering, and route advertisement in about 10 minutes. While this topic teeters the network/software line, the installation will be demonstrated below.

The Linux distro in use for the DNS server is the latest Lubuntu release (22.04.2).

```
sudo apt update
```

```
sudo apt install frr
```

Depending on your management preference, this server can get away with a single interface that connects to the internal CDN network. An additional loopback is needed to act as the anycast address interface. The loopback can be added using the command below.

```
ifconfig lo:1 170.0.0.1 netmask 255.255.255.255 up
```

The last thing to do before configuring the router is to enable the correct daemons, which can be done via the configuration file.

```
sudo nano /etc/frr/daemons
```

Change the bgpd option to yes.

```
bgpd=yes
```

Restarting FRR will cause the new daemon to load.

```
systemctl restart frr
```

To drop into the FRR command line, run the following command.

```
 sudo vtysh
```

The CLI is familiar as it operates almost identically to Cisco IOS.

```
ispadmin@AC-DNS-EAST:~$ sudo vtysh

Hello, this is FRRouting (version 8.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

AC-DNS-EAST#
```

In this lab, a static route was needed to inform the server which interface it would find its BGP peer. Once that direction is in place, a simple BGP configuration can be implemented.

```
AC-DNS-EAST# sh run bgpd
Building configuration...

Current configuration:
!
frr version 8.1
frr defaults traditional
hostname AC-DNS-EAST
log syslog informational
service integrated-vtysh-config
!
router bgp 65900
 bgp router-id 10.171.0.1
 neighbor 10.109.0.1 remote-as 65900
 !
 address-family ipv4 unicast
  network 10.170.0.1/32
  neighbor 10.109.0.1 next-hop-self
 exit-address-family
exit
!
```

FRR BGP Configuration

With FRR the Linux host networking is transformed into a full-function router. This saved fighting with Linux commands and provided the exact functionality needed.

```
AC-DNS-EAST# sh ip bgp
BGP table version is 9, local router ID is 10.171.0.1, vrf id 0
Default local pref 100, local AS 65900
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

   Network          Next Hop            Metric LocPrf Weight Path
*>i10.10.0.0/24     10.109.0.2               0    100      0 65500 i
*>i10.53.0.0/24     10.109.0.2               0    100      0 65500 i
*>i10.160.0.0/31    10.109.0.2               0    100      0 i
*>i10.160.0.2/31    10.109.0.3               0    100      0 i
  i10.170.0.1/32    10.172.0.0               0    100      0 i
*>                  0.0.0.0                  0         32768 i
*>i10.201.0.0/24    10.109.0.2               0    100      0 65300 i
*>i10.202.0.0/24    10.109.0.3               0    100      0 65100 i
```

FRR show ip bgp

Ensure the routing configuration is saved with the *wr* command before exiting FRR. All the routes learned via the specific daemon operating in FRR will also be visible in the Linux shell.

```
ispadmin@AC-DNS-EAST:~$ ip route
default via 100.64.100.1 dev ens160 proto static metric 100
default via 10.171.0.2 dev ens192 proto static metric 101
10.10.0.0/24 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
10.53.0.0/24 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
10.109.0.0/24 via 10.171.0.2 dev ens192
10.160.0.0/31 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
10.160.0.2/31 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
10.171.0.0/24 dev ens192 proto kernel scope link src 10.171.0.1 metric 101
10.201.0.0/24 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
10.202.0.0/24 nhid 16 via 10.171.0.2 dev ens192 proto bgp metric 20
100.64.100.0/24 dev ens160 proto kernel scope link src 100.64.100.214 metric 100
```

Linux routing table receiving routes from FRR

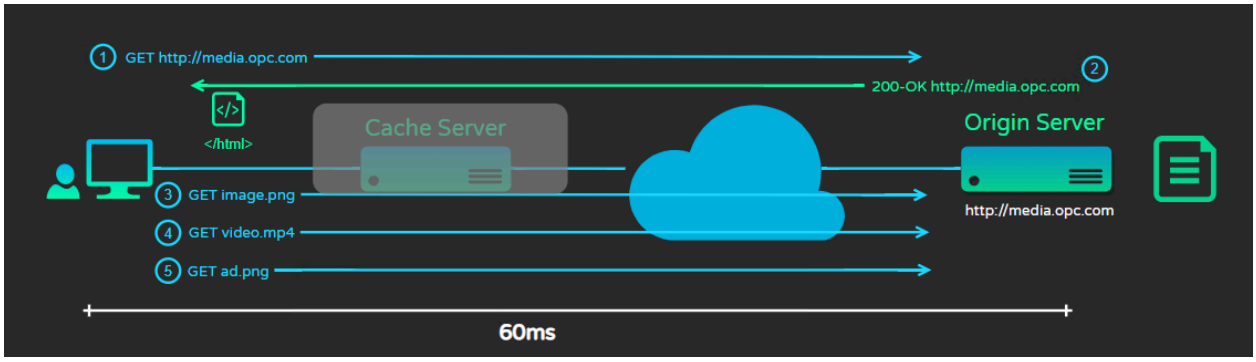
This concludes the core networking discussion. Beyond some specific methods, the lab network is nothing special and allows the demonstration of cache server impact.

Software

Overview

In a CDN network, the server hardware and accompanying software provide the majority of the benefits the end user will experience. Three software components will be examined in the following sections. Those three are bind DNS, HA-Proxy load balancing, and finally, the Varnish caching software. While the first two do not require much discussion, the caching server warrants further investigation.

Before diving into the specifics of particular software, the operations of a generic cache server will be examined. First, a refresher on HTTP and life before cache servers.

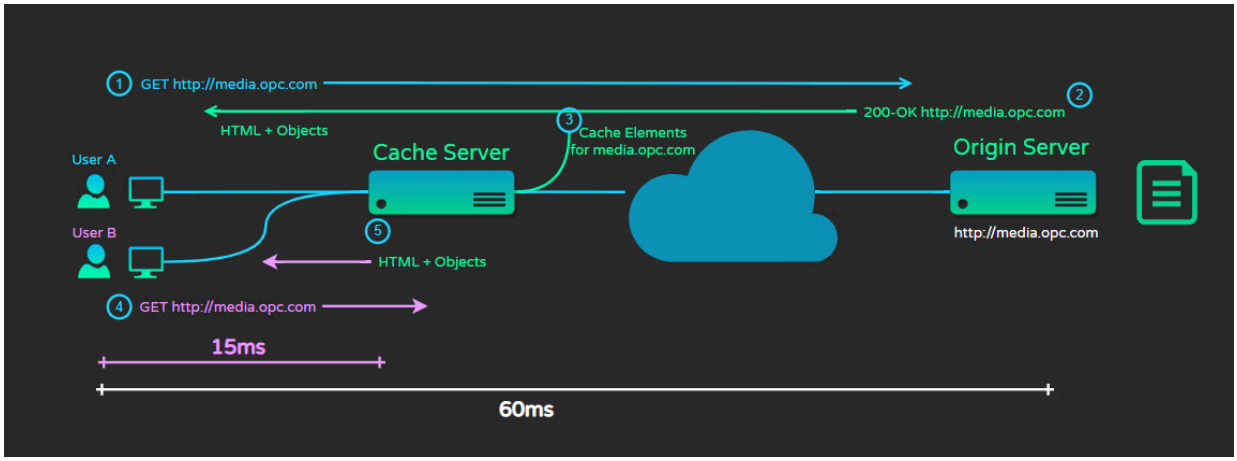


HTTP default operation

Referring to the diagram above, with a round trip time of 60ms and no cache server in use, this may become problematic in specific situations. After the initial GET, the user's browser is handed an HTML file, laying out everything the page will display. The browser will then begin pulling all those resources via HTTP GET calls. As those resources are received, the page will start to load. With modern websites, this would feel like revisiting the days of 6Mbps ADSL. Just navigating to Delta.com triggered 171 requests (This was with local caching disabled).

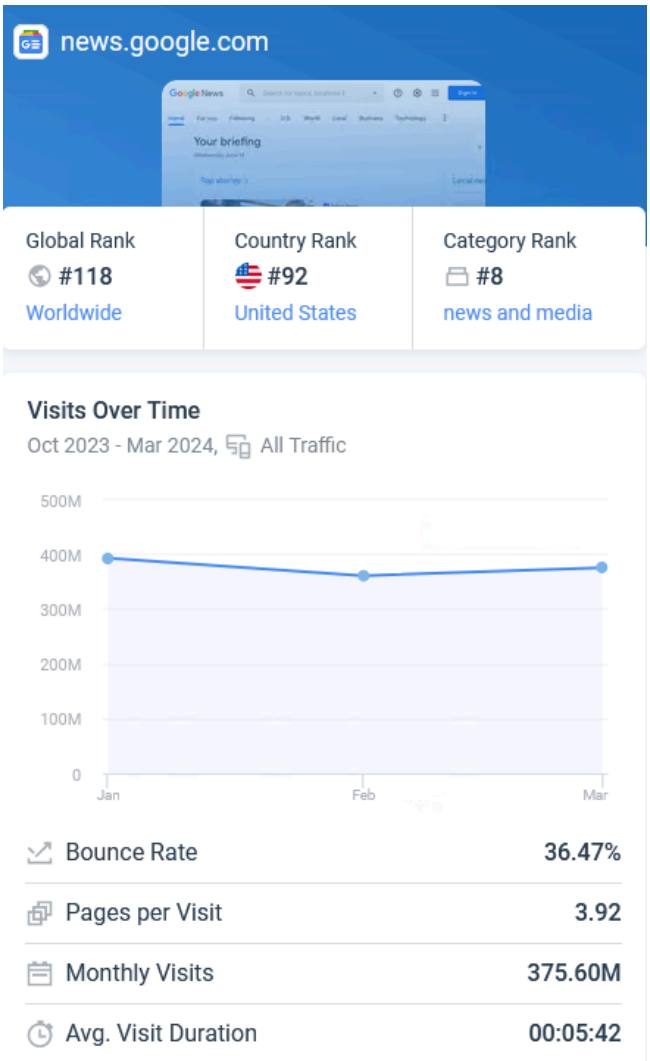
Browser Console Network Statistics

To clarify, the user does not have to wait for every request to finish before the page is usable. Luckily, this scenario is quite rare in today's landscape. Altering the diagram, the impact of a cache server can be further realized.



With the cache server placed back into the topology, initial requests are sent through the cache server and ultimately terminate at the origin server from which content is fetched. Upon the return of content, the cache server will store those elements locally for a certain period. When User B sends a GET request to media.opc.com the cache server can respond with the elements locally. This action cuts down page load times drastically, which improves user experience.

One metric website owners will use to gauge user experience is *bounce rate*. This percentage value equates to users who visit the page but take no action. Higher bounce rates have been observed to rise as page load times increase. To think beyond the network, loading times can cause more than user complaints and lead to loss of revenue. SimilarWeb has a great browser extension and website to view site statistics, including bounce rate.



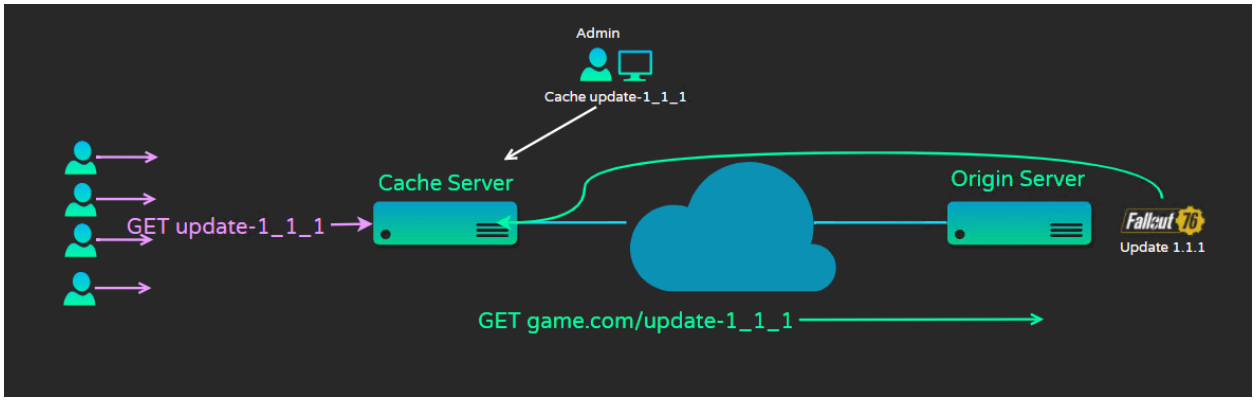
SimilarWeb extension for news.google.com

The implementation of a cache server alleviates multiple issues throughout the entire stack. The backend server does not handle all requests, while WAN and Data Center bandwidth is preserved. With a CDN architecture, the site becomes distributed and highly available by

inherent design. CDN providers also provide various security benefits, including web application firewalls, geo security, and DDoS mitigation.

Focusing more on the server software, a cache server works as a proxy between the public internet and the backend server. A server administrator has granular control over what is and is not cached. For example, if a site houses substantial data files, an administrator may not want to use that valuable space on the cache server. Conversely, some organizations may need to do just that. For example, game installs and updates can be large and would benefit from cache servers storing that information. As always, it depends.

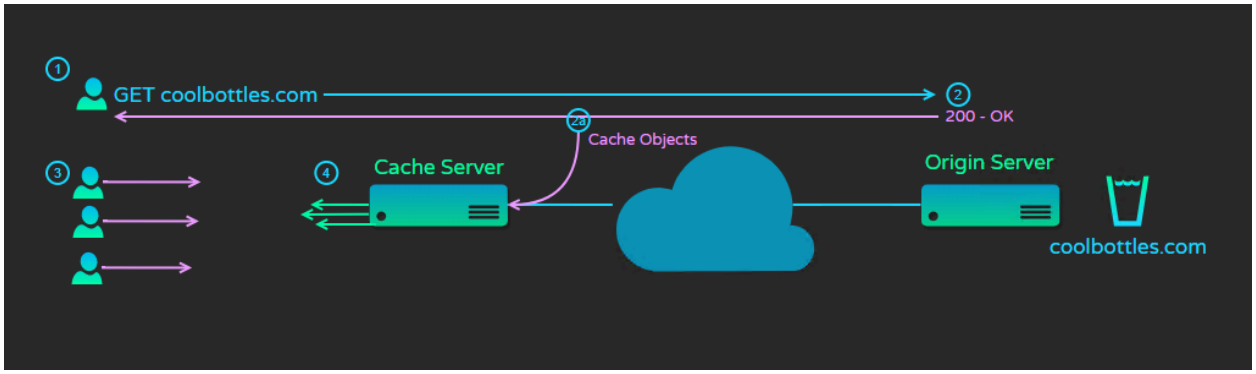
Cache servers within a CDN have different modes of operation. The previously mentioned example of a game update would benefit from a push-based cache. A push-based CDN allows administrators to push content down to the cache server, allowing them to specify the exact content the server will hold. This allows a proactive approach ahead of a significant release or expected increase in site traffic.



Push Based CDN

The administrator can cache the update before its official release, ensuring an improved user experience. This also places the traffic influx solely on the purpose-built CDN, whereas the origin servers may not handle the stress.

A pull-based CDN operates within the lab environment, and while it is not as predictive, the design handles spikes and abnormal traffic loads very well. For example, if a water bottle goes viral on TikTok and propels a brand into a new realm, a pull-based CDN can aid in the influx of traffic to a once low-volume site.

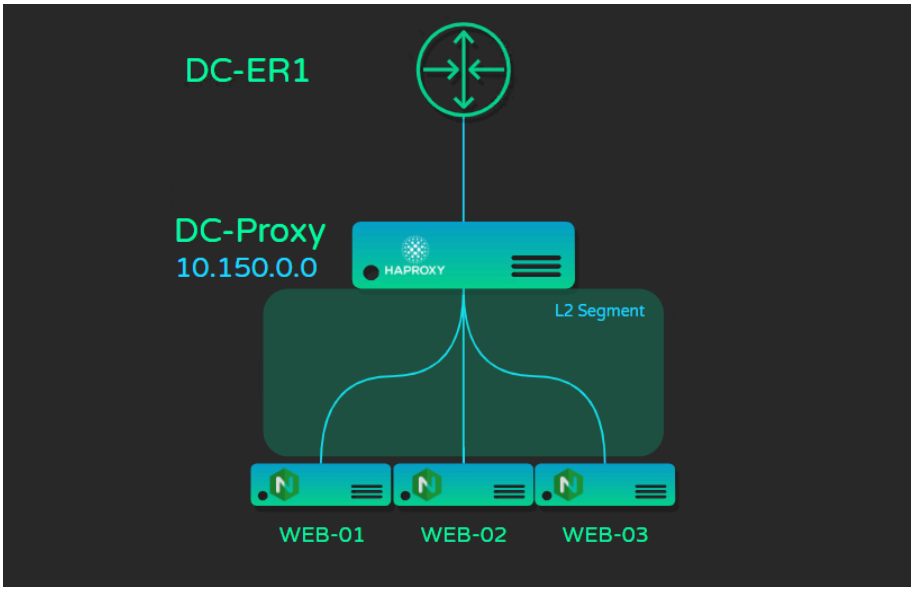


Pull Based CDN

The benefit of this configuration is that it adapts well and requires less administrative overhead. A pull-based server still retains options on what to cache and for how long, but that initial cache will happen due to an incoming request and not an explicit configuration. The pull-based model is susceptible to longer load times if an end user is unlucky in requesting uncached content. With that overall idea of cache servers explained the upcoming sections will dive into configuring the particular components used in the lab.

NGINX web servers

The “data center” of the lab will house three NGINX web servers that are being load-balanced by HAProxy. These are all running Ubuntu 22.04.3. The diagram below displays the topology that has been implemented.



To get started with NGINX update and install

```
sudo apt update
```

```
sudo apt install nginx
```

The site root will need to be created. This is where HTML files and associated site objects can be stored.

```
sudo mkdir /var/www/onpremcdn
```

```
sudo mkdir /var/www/onpremcdn/html
```

With the folder structure in place, an HTML file must be dropped in the directory. This will act as the index. A simple template was pulled off a website, and minor adjustments were made to call for the specific objects needed.

Create the HTML file with the following.

```
sudo nano index.onpremcdn.html
```

```
<html>
<head>
<title></title>
</head>
<body data-gr-ext-installed="" data-new-gr-c-s-check-loaded="8.912.0" data-new-gr-c-s-loaded="8.912.0">
<body style="background-color:#292929;">

<!--TOP BANNER-->
<img src="SplashGIF.gif"
height="250"
width="250"
class="center">
<!--TOP BANNER-->

<video src="Epcot92.mp4" controls width="300">
  This browser does not support HTML video.
</video>

<img src="OPLOGO.png">
<img src="EPC.png">
<img src="EPC_COM.png">
</body>
</html>
```

With that in place, NGINX will need site-specific details including what port to listen on and what

file to generate the site from. These files are stored in the sites-enabled directory.

```
cd /etc/nginx/sites-enabled
```

```
sudo nano onpremcdn
```

```
server {
	listen 8080;
	listen [::]:8080;

	server_name _;

	root /var/www/onpremcdn/html;
	index index.onpremcdn.html;

	location / {
		try_files $uri $uri/ =404;
	}
```

With the files created and placed in the respective directories, NGINX can be reset, and a curl command can be issued to test the site quickly (Curl may need to be installed).

```
sudo systemctl restart nginx
```

```
sudo apt install curl
```

```
ispadmin@WEB-01:/etc/nginx/sites-enabled# curl -I 127.0.0.1:8080
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Fri, 19 Apr 2024 14:38:36 GMT
Content-Type: text/html
Content-Length: 1409
Last-Modified: Mon, 15 Apr 2024 18:33:17 GMT
Connection: keep-alive
ETag: "661d72ed-581"
Accept-Ranges: bytes
```

The curl command confirms that the site is up. Removing the “ -I “ argument will display the complete response, which will be the HTLM file from /var/www/onpremcdn/html/. This configuration must be dispersed across the many nodes used to serve the site.

HAProxy

HAProxy will perform simple load balancing on the previously created NGINX servers. To begin run an apt update and install HAProxy.

```
sudo apt update
```

```
sudo apt install haproxy
```

Once installed, a single configuration file will need to be edited. This file will specify the frontend and backend details while enabling a GUI monitoring page.

```
sudo nano /etc/haproxy/haproxy.cfg
```

The specific configuration will be placed under the defaults section.

```
frontend public_109
      bind 10.150.0.0:8080
      default_backend DC_WEB

backend DC_WEB
      balance leastconn
      server web-01 172.16.40.11:8080 check
      server web-02 172.16.40.12:8080 check
      server web-03 172.16.40.13:8080 check

listen stats
      bind  100.64.100.228:9000
      stats enable
      stats uri /monitoring
      stats auth username:password
```

The frontend specifies what IP and port HAProxy will listen for incoming connections. This section also identifies the backend pool for this IP and Port combination. The backend specifies a load balance algorithm of least connected and each server's details. The *check* command allows health checks to be used to monitor server availability.

Finally the listen section allows GUI monitoring for HAProxy.

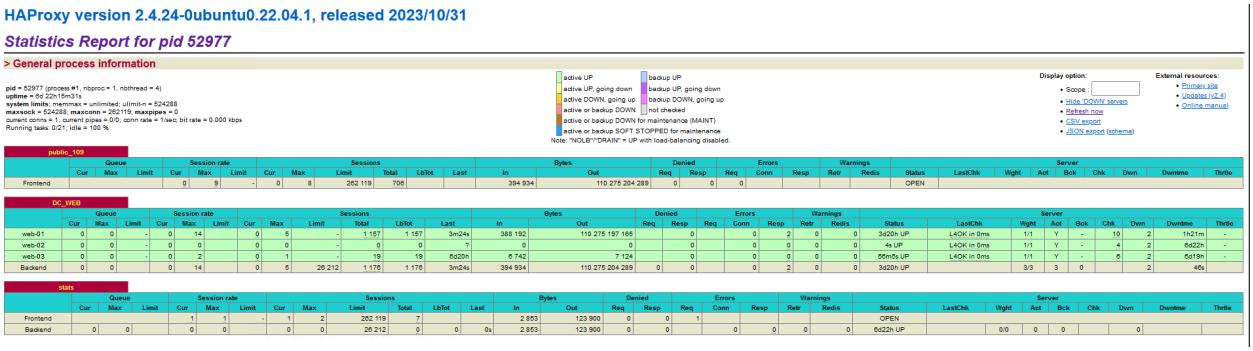
The service can be restarted and tested.

```
sudo systemctl restart haproxy
```

A curl command can be utilized to test whether the proxy serves the correct content from the backend members. Running the command with no arguments will display the HTML file, making verifying that the proper content is being served easier.

```
curl 10.150.0.0:8080
```

The GUI page can also be reviewed by navigating to the IP that was bound in the listening portion of the configuration. The GUI will provide uptime, traffic statistics, and other valuable insights for both the frontend and backend.



HAProxy Web Based Monitoring

With the HA proxy in place, we can now move on to the cache server's final supporting piece, Bind DNS.

Bind9

Bind DNS services will be utilized to allow the Anycast DNS architecture.

```
sudo apt update
```

```
sudo apt install bind9
```

A few configuration files need to be altered for essential bind operation.

```
sudo nano /etc/bind/named.conf.options
```

The listening IP will be specified within this initial configuration file, and who can query that IP.

```
options {
      listen-on { 10.170.0.1; };
      allow-query { any; };
```

The zone will then need to be specified in the named conf local file.

```
sudo nano /etc/bind/named.conf.local
```

```
zone "opc.com" {
type master;
file "/etc/bind/db.opc.com";
};
```

Finally, the zone file will store the actual DNS records. This is the db file previously referenced in the conf.local file.

```
sudo nano /etc/bind/db.opc.com
```

```
;
; BIND data file for local loopback interface
;
$TTL  604800
opc.com.    IN    SOA   ns1.opc.com. ispadmin.opc.com. (
                           2           ; Serial
                      604800      ; Refresh
                      86400       ; Retry
                      2419200           ; Expire
                      604800 )    ; Negative Cache TTL
;
@     IN    NS    ns1.opc.com.
ns1   IN    A     10.170.0.1
com82   IN        A     10.160.0.0
```

With changes to the DNS record, a server process must be reset for these to take effect.

```
sudo rndc reload
```

An NSlookup now reveals that the server is returning the configured mapping.

```
ispadmin@AC-DNS-EAST:/etc/bind$ nslookup com82.opc.com 10.170.0.1
Server:           10.170.0.1
Address:          10.170.0.1#53

Name:   com82.opc.com
Address: 10.160.0.0
```

This configuration will be enough for the demonstration, but it should be noted that the initail lab is tricking the system into acting similarly to a CDN. In the *real world,* there is a lot more going on. This blog post is fantastic for getting a better understanding.

Varnish (Cache Server)

Varnish is an open-source caching software that also has an enterprise-grade offering. This software runs in the lab on Ubuntu Desktop (22.04.3). This requires no fancy build process asit can be pulled via the apt install command.

```
sudo apt update
```

```
sudo apt install varnish
```

Before jumping into configuration changes, it is good practice to stop the application.

```
sudo systemctl stop varnish
```

The first configuration file edit that will be needed is the varnish.service file. Here, the frontend listening port and cache storage size will be defined.

```
sudo nano /lib/systemd/system/varnish.service
```

```
ispadmin@CS-WEST:~$ cat /lib/systemd/system/varnish.service

ExecStart=/usr/sbin/varnishd \
```

```
            -j unix,user=vcache \
            -F \
            -a :808 \
            -T localhost:6082 \
            -f /etc/varnish/default.vcl \
            -S /etc/varnish/secret \
            -s malloc,6G
ExecReload=/usr/share/varnish/varnishreload
ProtectSystem=full
ProtectHome=true
PrivateTmp=true
PrivateDevices=true
```

The text colored in green above is the only necessary change to get started. The -a denotes which port varnish will listen for frontend communication. The malloc storage specified after the -s argument is the cache size allotted. This caching is done in memory to retain the needed speed, and it is recommended that this value not exceed 75% of the available capacity. After making changes to the service file, the daemon must be reloaded.

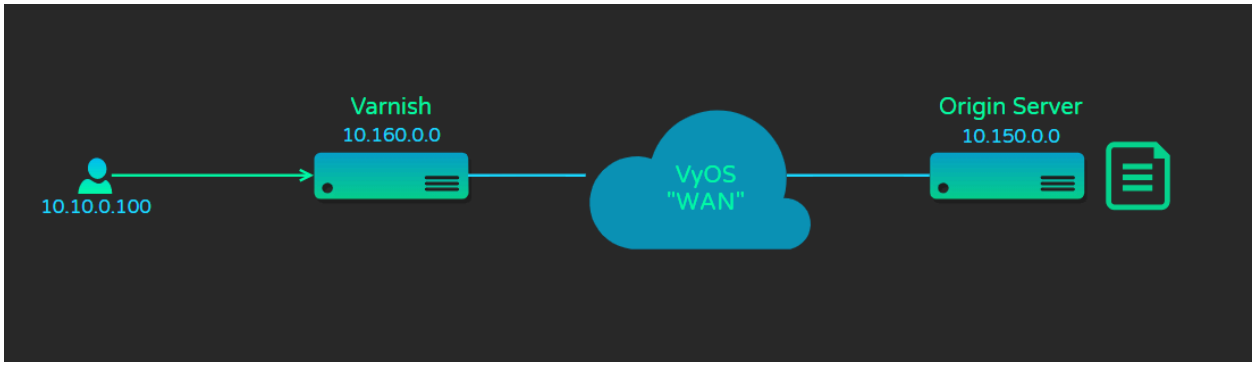
```
sudo systemctl daemon-reload
```

The next step will be to define the backend servers. This will be the HAProxy frontend that was previously configured.

```
sudo nano /etc/varnish/default.vcl
```

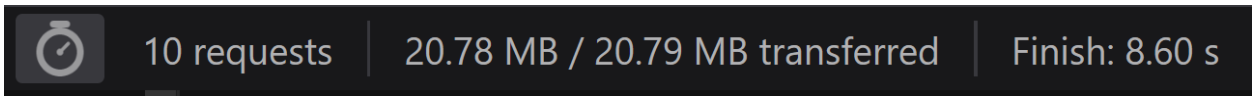
```
backend default {
      .host = "10.150.0.0";
      .port = "8080";
}
```

The file is written in a language specific to varnish called Varnish Configuration Language (VCL).
Given that the correct routing is in place, Varnish should now be able to serve and cache content. A quick test can be conducted to prove this.



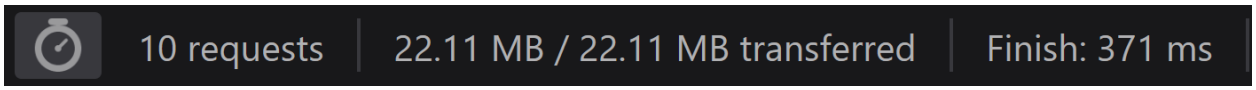
Demonstration Topology

The end user will pull a webpage with large pictures and a video file. The developer console will be utilized within the browser to display details.



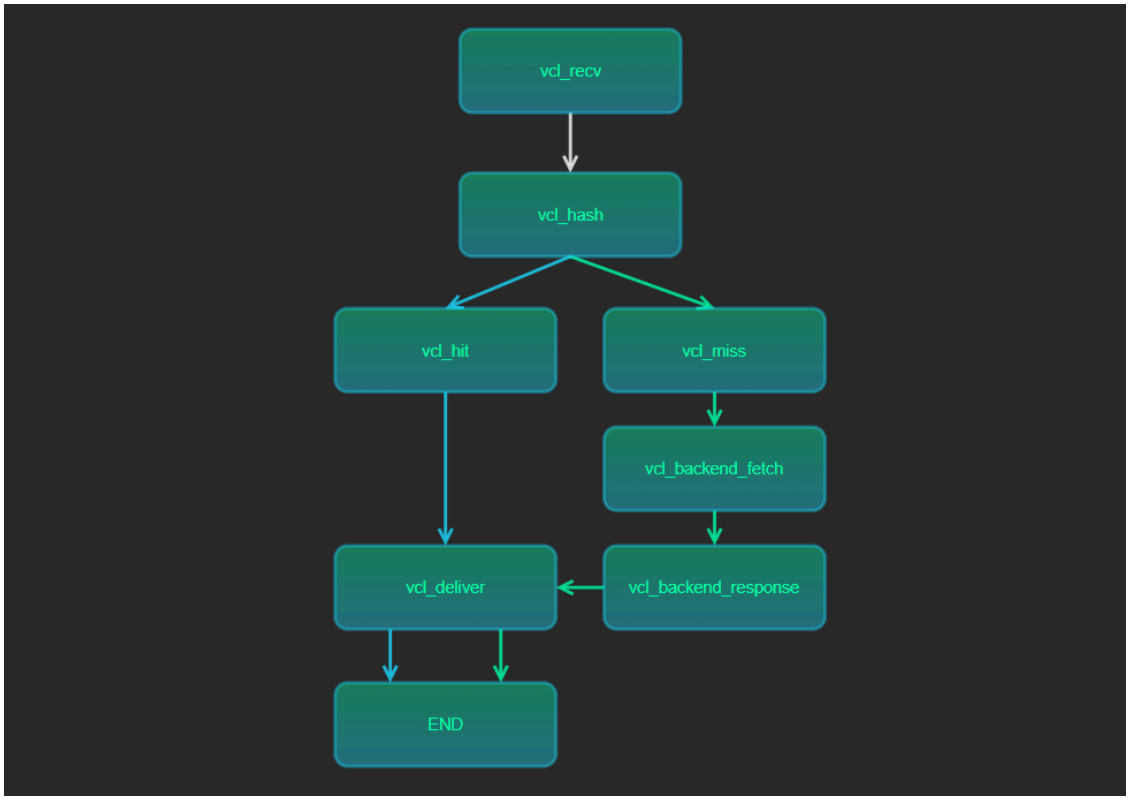
Transaction Where Varnish Pulled From Origin

The content previously requested by Varnish has now been cached, making the subsequent request noticeably faster.



Varnish Cached Response

This is the simple gist of the operation, proven by the timings. To understand the cache object flow, the finite state machine (FSM) will be the topic of quick review.



Varnish FSM stages

When a request is made, a content hash is calculated. The hash value is then used to search the local cache contents, and if a match is found, this results in the vcl_hit state. Content reaching the hit state can then proceed directly to the delivery stage. If the hash value can not be found, the state drops into vcl_miss, and a backend fetch is initiated. The server will receive the needed content during the backend response and move on to the vcl_deliver state, which pushes content to the end user and stores the object locally for a set time.

Running the following command will enable live logging on the varnish server.

```
sudo varnishlog
```

The log output provides abundant detail on both the client and server. Displayed below are the finite state machine definitions discussed above in action.

```
-   VCL_call       HASH
-   VCL_return     lookup
-   VCL_call       MISS
-   VCL_return     fetch
```

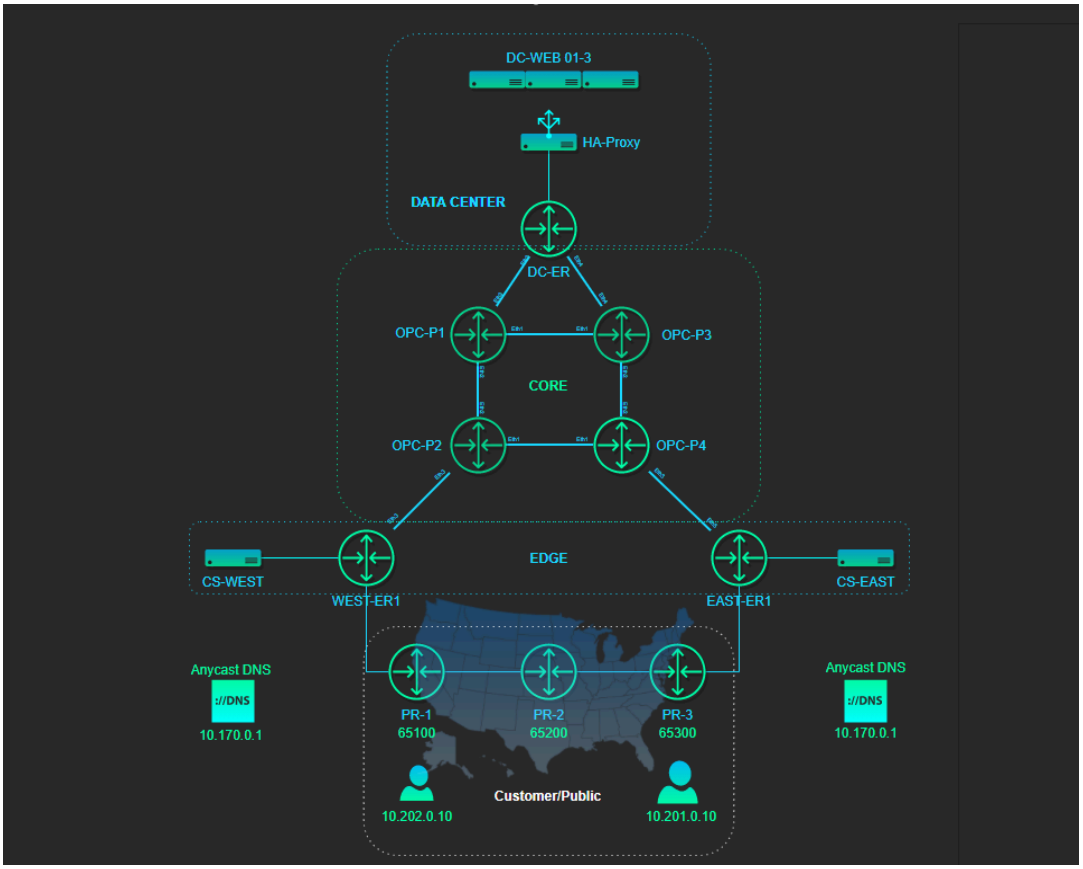
Once the server has run through the finite states and cached content, the subsequent request to the server will result in a hit and deliver. This is where the latency reduction occurs. The delivery process is streamlined with distributed nodes, cached objects, and faster memory.

```
-   VCL_call       RECV
-   VCL_return     hash
-   ReqUnset       Accept-Encoding: identity
-   VCL_call       HASH
-   VCL_return     lookup
-   Hit            229390 113.111034 10.000000 0.000000
-   VCL_call       HIT
-   VCL_return     deliver
```

This concludes the varnish server configuration.

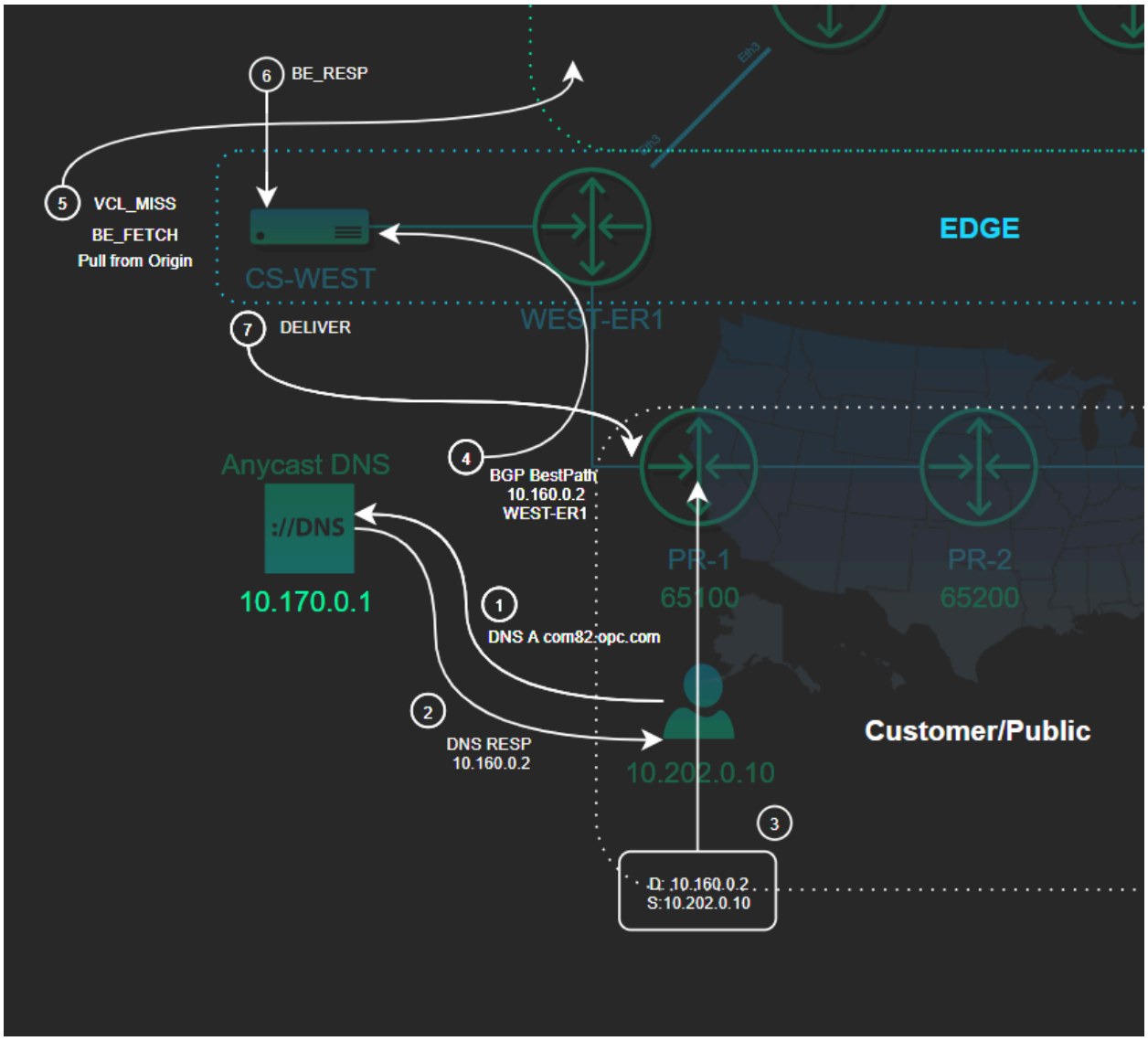
Operation

With all network and software components in place, the operations of the infrastructure as a whole can be investigated.



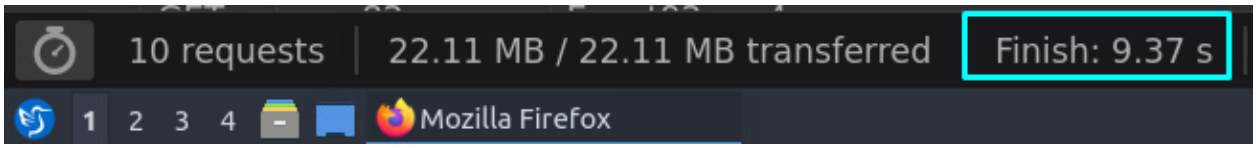
Simple Cache Comparison

In this operational example, the two hosts residing on the public internet will point directly to the 10.170.0.1 DNS address and be routed to the nearest point. From the closet nameserver, they receive the record for com82.opc.com. Each server has a unique value for that region's server (West = 10.160.0.2, East = 10.160.0.0). Host 10.202.0.10, which sits within the west region, will send the initial request. The West DNS server will answer this request and direct the client to the west cache server.

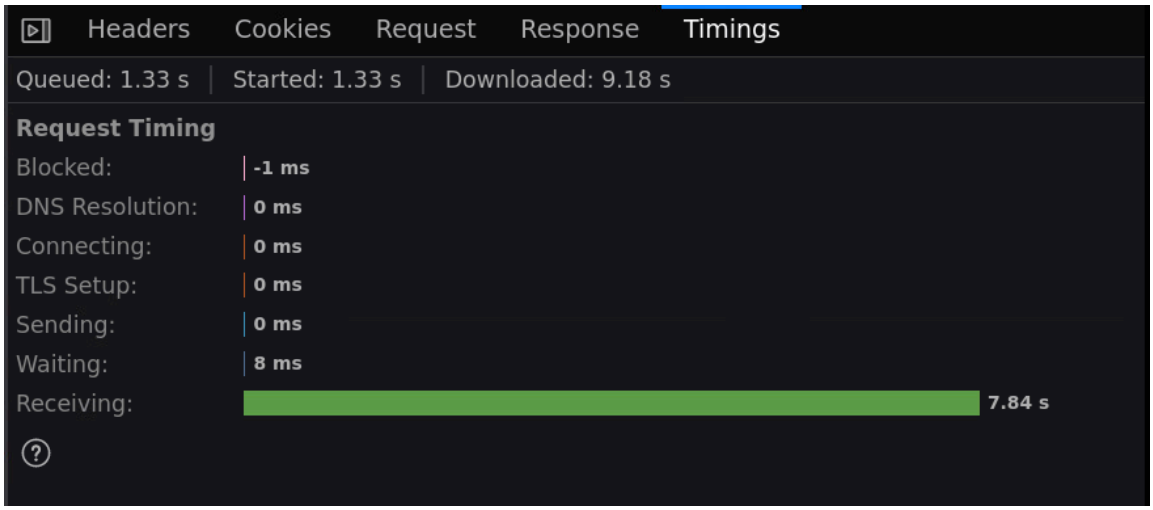


The flow of the first demonstration

The initial load time of the page and its associated assets is 9.37s.



Most of this time was spent pulling the video object from the origin server.



**When encountering this beyond a lab, Varnish does have some settings that can be configured to tune the delivery of large files. These are referred to as transit buffers.

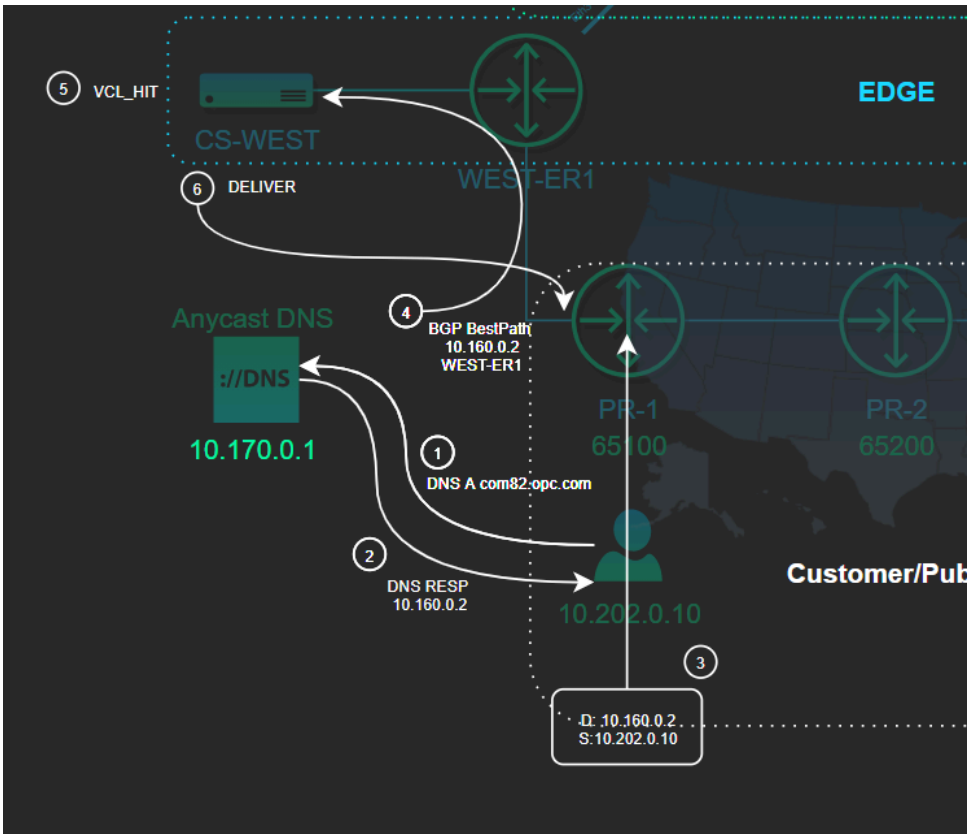
After this initial request, the server should now have cached all of the site's objects, as no restrictions are in place. Running the stats command will confirm this theory.

```
sudo varnishstat
```

The SMA.s0.g_bytes indicates the space utilized from the allotted storage. Currently, the varnish serve is caching 579MB of content.

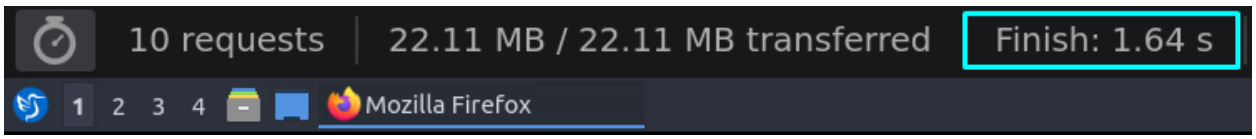
```
SMA.s0.g_alloc                                                4
SMA.s0.g_bytes                                          579.83M
SMA.s0.g_space                                            5.43G
VBE.boot.default.bereq_hdrbytes                           4.80K
VBE.boot.default.beresp_hdrbytes                          3.12K
VBE.boot.default.beresp_bodybytes                         1.17G
VBE.boot.default.req                                         14
```

A slight change in the diagram was made to reflect the process after Varnish cached the specific site content.

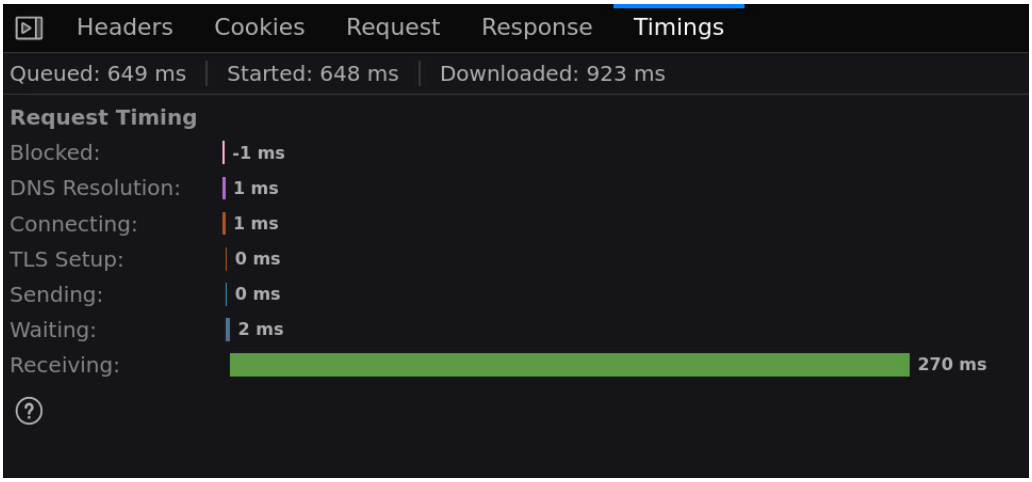


The flow of the second demonstration

The second request with the cached objects has a load time of 1.64s.



The video transfer timing was also drastically reduced, which took 923ms to retrieve.



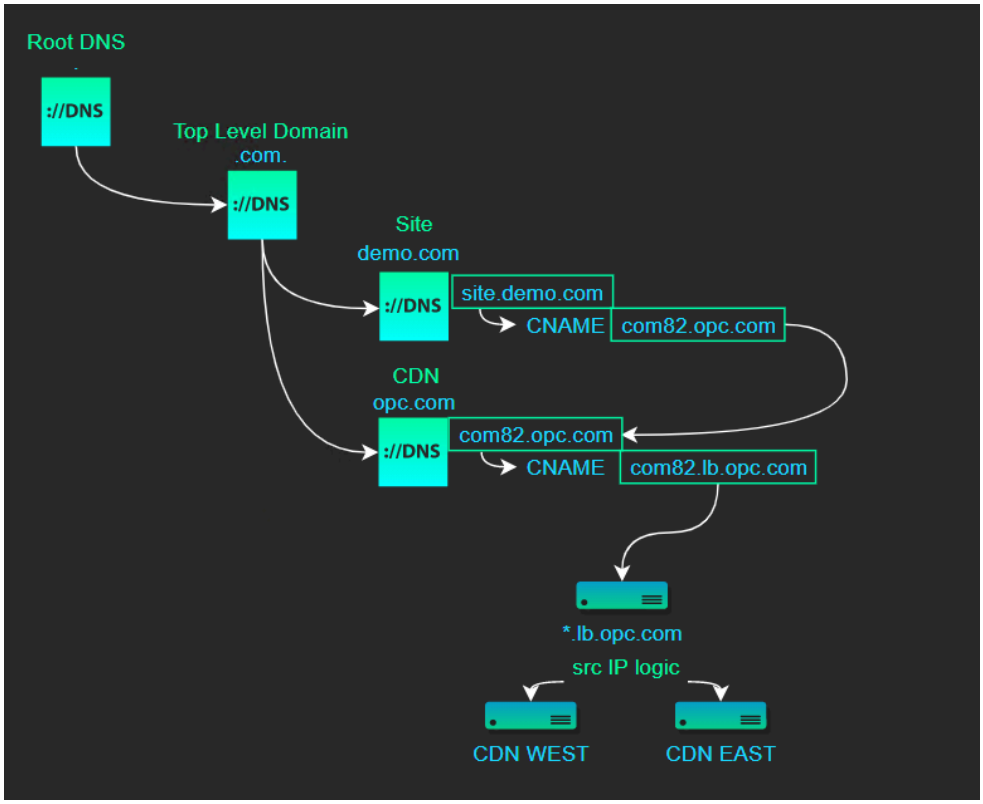
It is important to note that this is not the entire video file. As the video is scrubbed, additional GETs can be observed via the console. The browser will pull the necessary section of the video in smaller blocks. The status code of these transactions is 206 - Partial Content.

206	GET	com82.op...	Epcot92.mp4	media	mp4	415.58 kB	415.19 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	383.72 kB	383.33 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	333.04 kB	332.65 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	512.59 kB	512.21 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	123.08 kB	122.69 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	99.91 kB	99.53 kB
206	GET	com82.op...	Epcot92.mp4	media	mp4	212.86 kB	212.47 kB

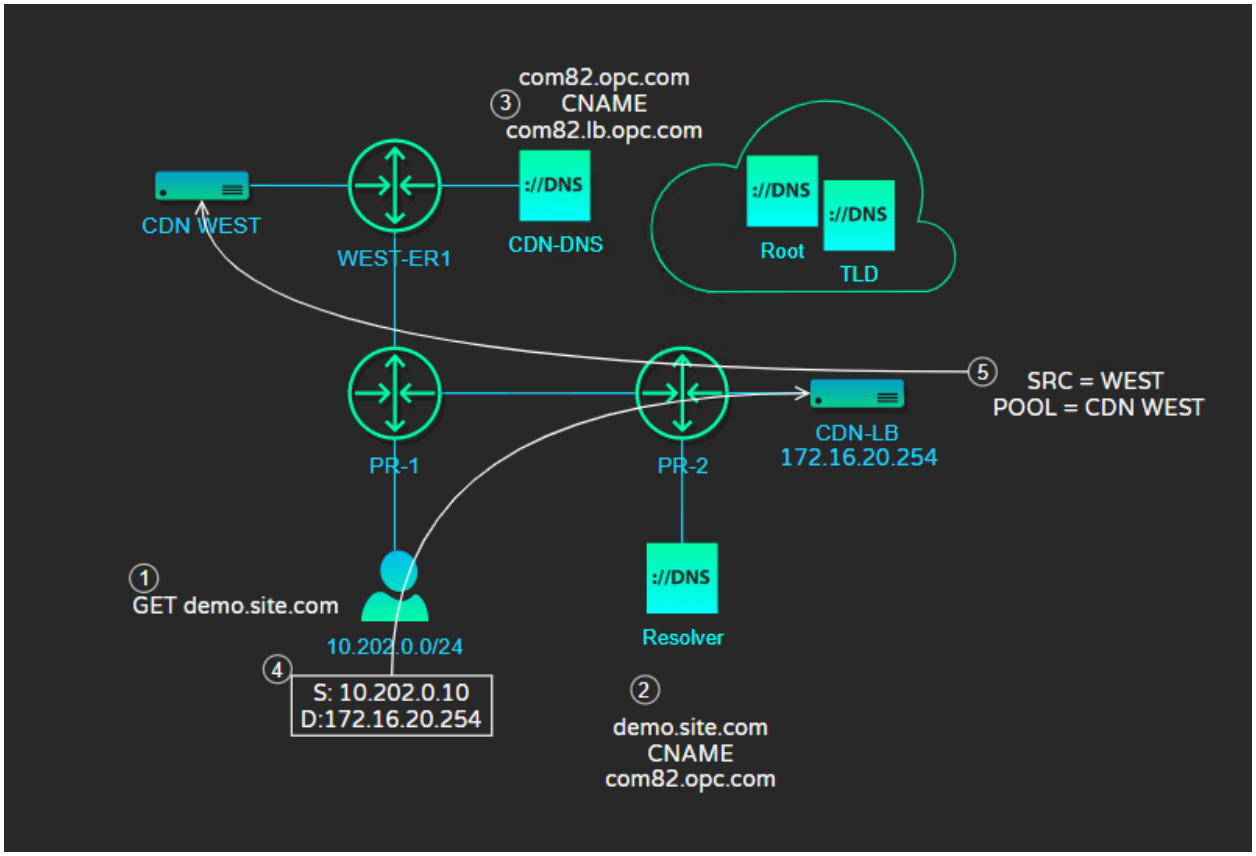
Any subsequent requests from any customer accessing the static content will benefit from the cached resources.

Complex DNS Example

Something kept bugging me to not “cheap out” on the DNS implementation and do it correctly (mostly). With this example, Root, TLD, and site-specific DNS servers were created. Another HAProxy was created as the CDN magic box, which distributes requests to the nearest ingress point via source IP examination.



Rather than investigate every detail of the DNS Servers, I will drop those configurations in a GitHub repo for anyone interested. The Root and TLD servers are Lubuntu VMs running bind9, while the demo.com server is a Windows Server 22 VM running the DNS feature. East and West public internet clients point toward the Windows device as their recursive resolver. Setting up a root server was a veiled mystery. There are few write-ups on the correct way to achieve it. My config was based on a GitHub DNS lab repo. This example breaks from reality as the HAProxy is a traditional load balancer and establishes a backend connection rather than returning the closet cache IP. The emulation comes from some backend pool selection logic based on source IP.



While this method works, I could still not get an exact operation replica. The very **scientific* process of distributing via region IP in the lab is as follows:

```
frontend cdnlb
      bind 172.16.20.254:80
      use_backend cdnwest if { src 10.202.0.0/24 }
      use_backend cdneast if { src 10.201.0.0/24 }

backend cdnwest
      option forwardfor
      server west 10.160.0.2:808 check

backend cdneast
      option forwardfor
      server east 10.160.0.0:808 check
```

HAProxy East/West redirect logic

A quick follow of a request will solidify this final flow. A dig command from the west host reveals that all the records are resolved correctly. The final destination, 172.16.20.254 is the front-end IP of the CDN-LB.

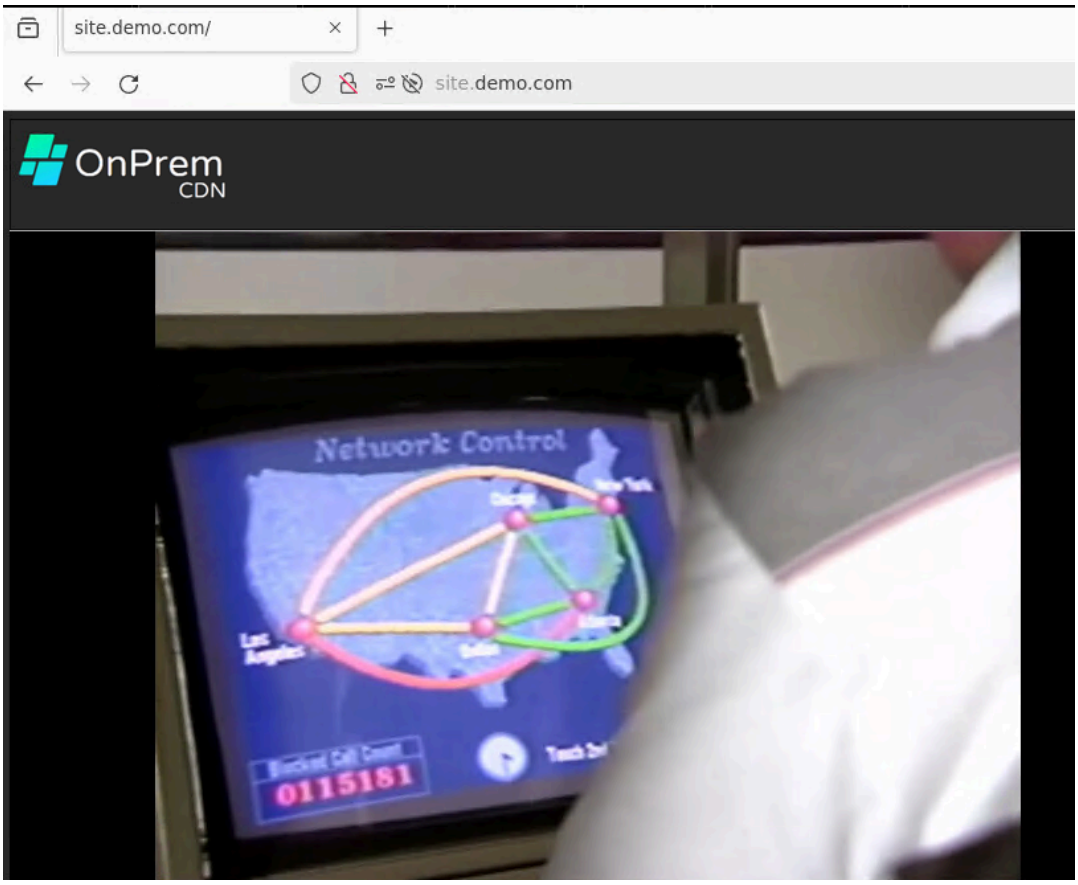
```
lubuntu@lubuntu:~$ dig site.demo.com

; <<>> DiG 9.18.18-0ubuntu0.22.04.2-Ubuntu <<>> site.demo.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43860
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;site.demo.com.                 IN      A

;; ANSWER SECTION:
site.demo.com.          3600    IN      CNAME   com82.opc.com.
com82.opc.com.          50      IN      CNAME   com82.lb.opc.com.
com82.lb.opc.com.       50      IN      A       172.16.20.254

;; Query time: 4 msec
;; SERVER: 172.16.20.220#53(172.16.20.220) (UDP)
;; WHEN: Mon Apr 22 20:18:36 UTC 2024
;; MSG SIZE  rcvd: 105
```

When a request is made the West Varnish server logs the request, and HAProxy has added the X-Forwarded-For header, making it easy to identify that this has in fact come from the West host.

```
*   << Request  >> 131077
-   Begin          req 131076 rxreq
-   Timestamp      Start: 1713817673.870180 0.000000 0.000000
-   Timestamp      Req: 1713817673.870180 0.000000 0.000000
-   VCL_use        boot
-   ReqStart       172.16.20.254 58052 a0
-   ReqMethod      GET
-   ReqURL         /Epcot92.mp4
-   ReqProtocol    HTTP/1.1
-   ReqHeader      host: site.demo.com
-   ReqHeader      user-agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:122.0) Gecko/20100101 Firefox/122.0
-   ReqHeader      accept: video/webm,video/ogg,video/*;q=0.9,application/ogg;q=0.7,audio/*;q=0.6,*/*;q=0.5
-   ReqHeader      accept-language: en-US,en;q=0.5
-   ReqHeader      range: bytes=606240768-
-   ReqHeader      referer: http://site.demo.com/
-   ReqHeader      accept-encoding: identity
-   ReqHeader      x-forwarded-for: 10.202.0.10
-   ReqUnset       x-forwarded-for: 10.202.0.10
-   ReqHeader      X-Forwarded-For: 10.202.0.10, 172.16.20.254
```

Varnish will rewrite the header to contain both the original host and the CDN LB address which the request was sourced from. This information can be utilized how the individual organization sees fit. A real-life varnish example will be reviewed and the amount of added headers is overwhelming.

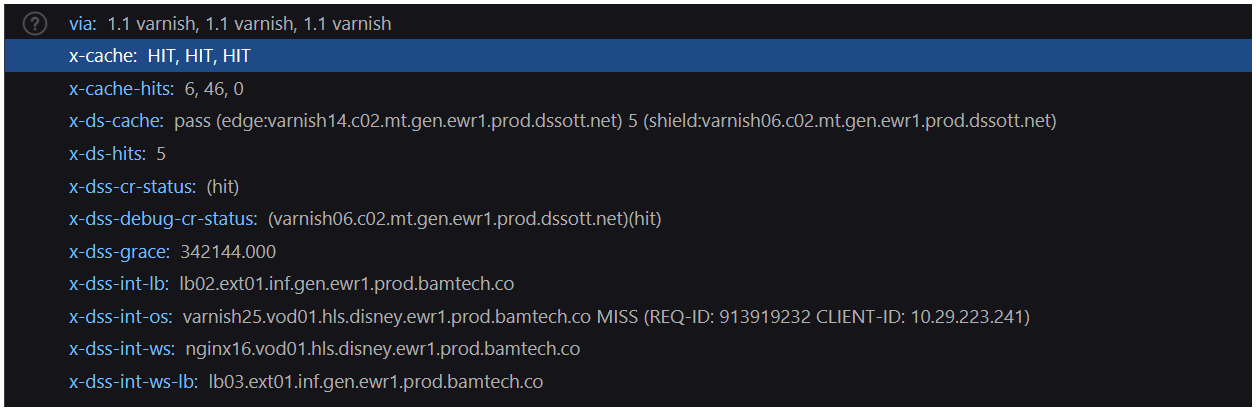
At the host, the page loads as expected with no issues despite the DNS acrobatics.



This wraps up the more involved example, which demonstrated a realistic flow of traffic and how a customer will be directed to the CDN ingress.

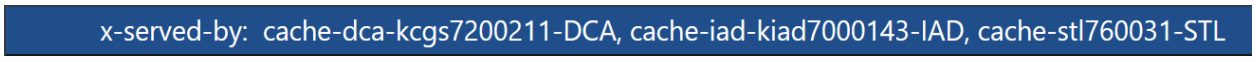
Real-World Example

I stumbled upon some real-life varnish implementations while getting familiar with the browser console. Disney+ streaming is utilizing varnish heavily in its content distribution.

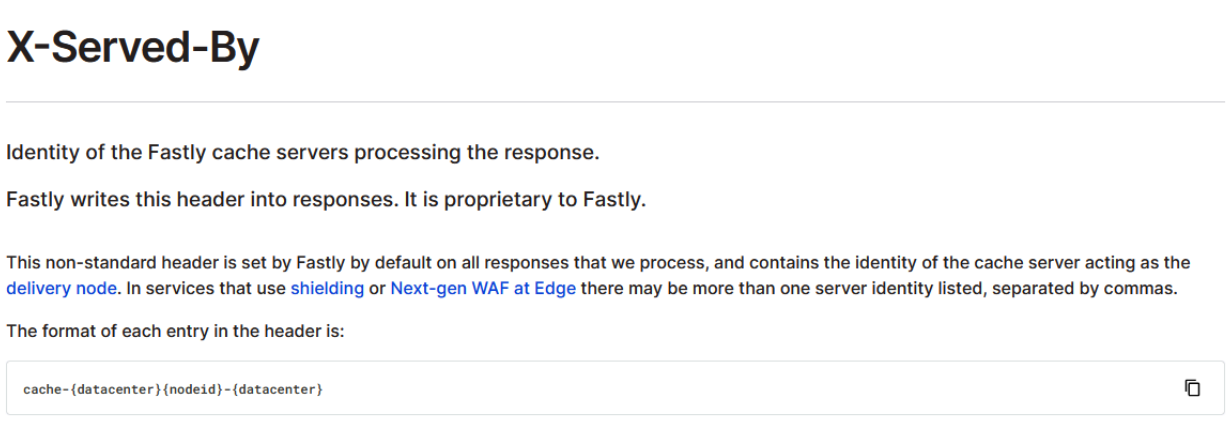


Added headers

The via response header indicates the use of 3 varnish instances. There is also an abundance of customer headers in use, which I imagine are needed to keep track of flows in a vast network such as a leading streaming provider. Some values that can be gleaned are the load balancers used, an indication of the varnish FSM status, and some web server information. In this particular example, I was able to identify a Fastly proprietary header, but I also saw Akamai hosts in the mix when pulling other content.



Fastly specific header



X-Served-By documentation

The last header seen indicates a pre-cache as in a push-based operation.

Pre-cache added header

The three values used are all options within the varnish configuration. This deployment would be essential for a video platform where content is hyped for weeks/months before the actual release date. This section served little purpose for any implementation needs but was intended to link some of the simple examples seen in the lab to real-world implementation.

Summary

This was a “just because” document. I have found that taking a deeper look into architecture has helped me understand the bigger picture of services. Over-the-top networks enabled by CDNs are a massive part of today's internet landscape. As subscriber-based DOCSIS/Sattelite/IPTV offerings continue to dwindle, modern streaming will only become more prevalent, which relies on CDNs to aid in content distribution. I hope something in the document was of use, and I thank you if you have read it this far.