

HTTP Web Proxy Server & Content Delivery Networks

Prepared for:

Dr. Le,

CS 4390 Computer Networks

Prepared by:

Stephen Brooks, Alan Padilla

DATE

July 26, 2016

HTTP Web Proxy Server

a) Completed Code

```
from socket import *
import sys
import re

if len(sys.argv) <= 1:
    print 'Usage : "python ProxyServer.py server_ip"\n[server_ip : It is the IP Address Of Proxy Server]'
    sys.exit(2)

# Create a server socket, bind it to a port and start listening
tcpSerPort = 8888
tcpSerSock = socket(AF_INET, SOCK_STREAM)
tcpSerSock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
# Fill in start.
tcpSerSock.bind(("",tcpSerPort)) #listening port
tcpSerSock.listen(1)
# Fill in end.
while 1:
    # Start receiving data from the client
    print 'Ready to serve...'
    tcpCliSock, addr = tcpSerSock.accept()
    print 'Received a connection from:', addr
    message = tcpCliSock.recv(1024) # added code
    print message
    # Extract the filename from the given message
    print message.split()[1]
    requestType = message.split()[0]
    filename = message.split()[1].partition("/")[2]
    print filename
    fileExist = "false"
    filetouse = "/" + filename
    print filetouse
```

```

try: # Check whether the file exist in the cache
    f = open(filetouse[1:], "r")
    outputdata = f.readlines()
    fileExist = "true"
    # ProxyServer finds a cache hit and generates a response
message

    tcpCliSock.send("HTTP/1.0 200 OK\r\n")
    tcpCliSock.send("Content-Type:text/html\r\n")
    # Fill in start.
    for data in outputdata:
        tcpCliSock.send(data)
    # Fill in end.
    print 'Read from cache'
    continue

# Error handling for file not found in cache
except IOError:
    # if fileExist == "false":
    print "Cache file does not exist"

# Create a socket on the proxyserver
# Fill in start.
c = socket(AF_INET, SOCK_STREAM)
# Fill in end.
hostn = filename.replace("www.", "", 1).partition("/")[0]
print 'going to connect to: ', hostn

try:
    # Connect to the socket to port 80
    # Fill in start.
    c.connect((hostn, 80))
    # Fill in end.
    # Create a temporary file on this socket and ask port 80
    #for the file requested by the client
    fileobj = c.makefile('r', 0)
    subfile = filename.partition("/")[2]

```

```

        if requestType == 'GET':
            fileobj.write("GET "+"http://" + hostn + "/" + subfile + "
HTTP/1.0\n\n")
        else:
            fileobj.write("POST "+"http://" + filename + " HTTP/1.0\n\n")
            msg_body = re.split("\r\n",message, maxsplit=1)[1]
            fileobj.write(msg_body)

        # Read the response into buffer
        # Fill in start.
        tmpBuffer = fileobj.readlines()
        # Fill in end.
        # Create a new file in the cache for the requested file.
        # Also send the response in the buffer to client socket
        # and the corresponding file in the cache
        tmpFile = open("./" + filename,"wb")
        # Fill in start.
        for data in tmpBuffer:
            tmpFile.write(data)
            tcpCliSock.send(data)

        tmpFile.close()
        # Fill in end.
    except IOError as e:
        print ""
        # HTTP response message for file not found
        # Fill in start.
        try:
            print 'Server Response:',tmpBuffer[0].split("\n", 1)[0]
            tmpBuffer = ""
        except Exception as e:
            print " "
        # Fill in end.
        # Close the client and the server sockets
        tcpCliSock.close()
        # Fill in start.

```

```
c.close()  
# Fill in end.
```

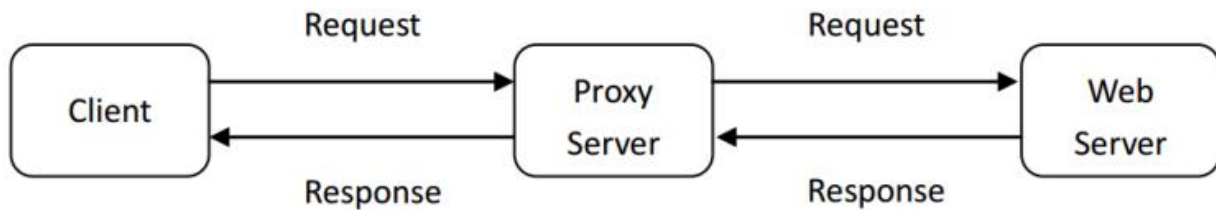
b) Background and Summary

HTTP is the protocol that runs the internet. A web page is loaded by a series of request from a client computer being the user and the web server that sends the data that loads the web page through the clients' web browser. This proxy server basically acts like a middle man in this exchange between the proxy server and client server. Proxy servers can help reduced load time by providing cached copies of internet web pages and also provide an extra level of security and filter between the client and the internet. The proxy server for this project first creates a socket that allows it to accept a connection from a client. It then prints the message and depending on whether it's a get request or post request it performs several actions. For any other type of request, the code ignores it as it is in the category of an unhandled request type. For the post request and get request it stores copy of the payload and sends it to the server. If the file is already in the cache, then it reads it from the file. The proxy is acting as the middle man taking request from client sending the request and then storing the response in its cache.

c) Hardware setup and configuration

The proxy server was run from our local machine. For testing purposes we used a tool known as Curl. Curl for our testing purposes mimics a web browser and sends out GET and POST request with a simple command request. This made debugging easier and also served as a demo of the proxy server. Curl is the client that send the request to the proxy server the proxy server then sends the request to the specified webpage. From there the proxy gets the response from web server and pushes that response to the client, being Curl for us. Curl also allows to continue if a redirected is sent and is able to show that our proxy indeed did pass the 200 OK message back to curl.

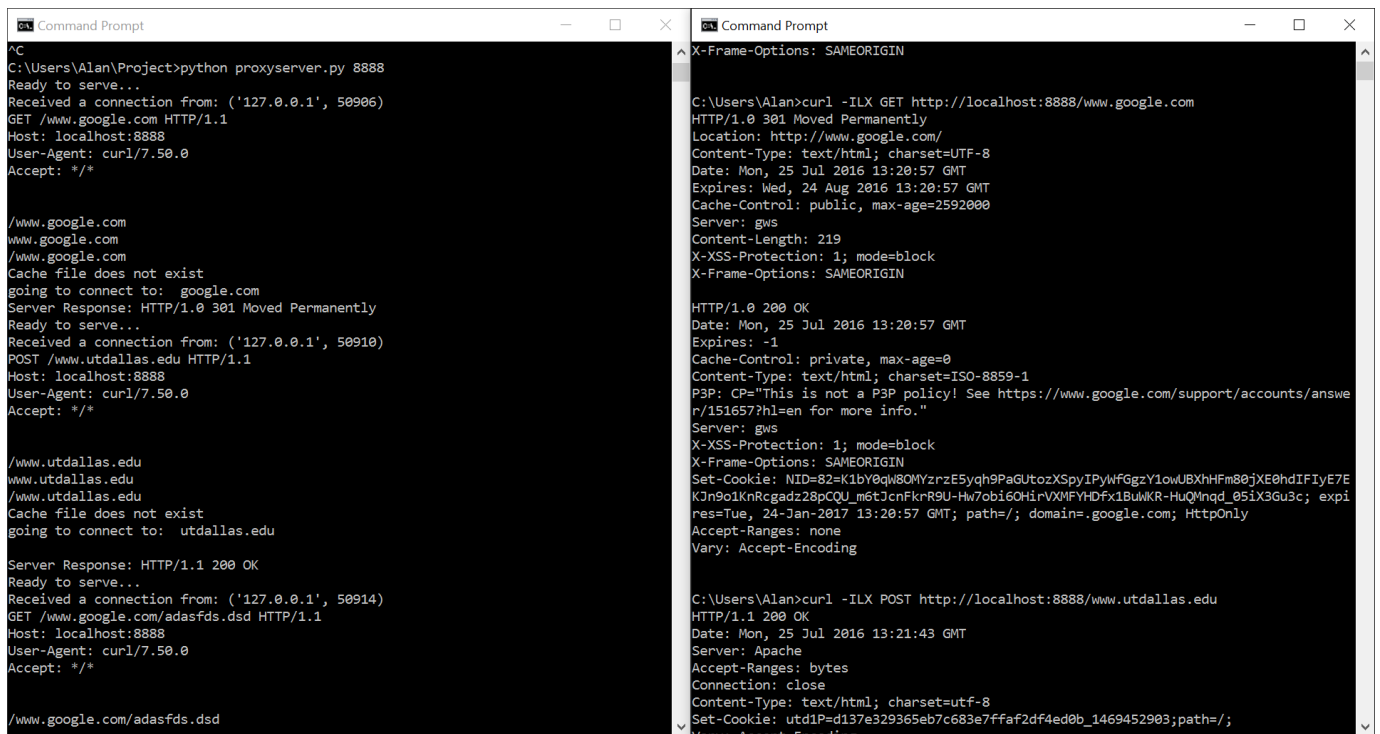
d) Design



Because the framework of the code was given in the included documents most of the code is self-explanatory through the comments. A server socket is created then the server socket begins to listen for any request. Once it does receive a connection from a client user it checks if it's checks first what type of request it is. With it being either a GET or POST request. For a post request, if not cached it would write the POST request just as similar to a GET, however it would then get the message body of such POST and write that as well in the cache. After this then it would send the request to the webserver. For the 404 Error handling we read the reply back from the web server and determine what type of response we received. For GET and POST responses that are not caught in an exception are sent back to client and cached. If caught, we state the 404 Error message. In the end our final design allowed for the statement of all message response from web servers because it printed directly the response in the proxy output.

e) Screenshots

The first screenshot below shows two command prompts. The one on the left is our proxy first stating the “Ready to serve...” message to the client. The one on the right is a curl command, “curl -ILX GET <http://localhost:8888/www.google.com>”. As seen Curl shows the two responses received from the proxy server the first being a redirect and the second being a 200 OK. The second curl request is “curl -ILX POST <http://localhost:8888/www.utdallas.edu>”. In the proxy tab we see the response is 200 OK.



```
Command Prompt
C:\Users\Alan>python proxyserver.py 8888
Ready to serve...
Received a connection from: ('127.0.0.1', 50906)
GET /www.google.com HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.50.0
Accept: */*

/www.google.com
www.google.com
/www.google.com
Cache file does not exist
going to connect to: google.com
Server Response: HTTP/1.0 301 Moved Permanently
Ready to serve...
Received a connection from: ('127.0.0.1', 50910)
POST /www.utdallas.edu HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.50.0
Accept: */*

/www.utdallas.edu
www.utdallas.edu
/www.utdallas.edu
Cache file does not exist
going to connect to: utdallas.edu

Server Response: HTTP/1.1 200 OK
Ready to serve...
Received a connection from: ('127.0.0.1', 50914)
GET /www.google.com/adasfds.dsd HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.50.0
Accept: */*

/www.google.com/adasfds.dsd

Command Prompt
X-Frame-Options: SAMEORIGIN

C:\Users\Alan>curl -ILX GET http://localhost:8888/www.google.com
HTTP/1.0 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Mon, 25 Jul 2016 13:20:57 GMT
Expires: Wed, 24 Aug 2016 13:20:57 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

HTTP/1.0 200 OK
Date: Mon, 25 Jul 2016 13:20:57 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See https://www.google.com/support/accounts/answer/151657?hl=en for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=82=K1by0qW8QMYzrzE5yqh9PaGutozXSpyIPyWfGgzY1owUBXhHfM80jXE0hdIFIyE7E
KJn9o1KnRcgadz28pCQU_m6tJcnFkrR9U-Hw7obi60HirVXMFYHDFx1BuwKR-HuQmnd_05iX3Gu3c; expi
res=Tue, 24-Jan-2017 13:20:57 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding

C:\Users\Alan>curl -ILX POST http://localhost:8888/www.utdallas.edu
HTTP/1.1 200 OK
Date: Mon, 25 Jul 2016 13:21:43 GMT
Server: Apache
Accept-Ranges: bytes
Connection: close
Content-Type: text/html; charset=utf-8
Set-Cookie: utdIP=d137e329365eb7c683e7ffaf2df4ed0b_1469452903;path=/;
Vary: Accept-Encoding
```

The second screenshot highlights the handling of a 404 error message. We sent the request “curl -ILX <http://localhost:8888/www.google.com/adasfds.dsd>”, which is obviously a fake URL and a clear indication that it should receive a 404 response from the web server. Curl receives an empty response from the proxy. However, this is intentional and not lost in the proxy because we show that the response from the webserver is clearly a 404 error message.

```
Command Prompt
/wwww.google.com
Read from cache
Ready to serve...
Received a connection from: ('127.0.0.1', 50956)
POST /www.utdallas.edu HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.50.0
Accept: */*

/wwww.utdallas.edu
www.utdallas.edu
/wwww.utdallas.edu
Cache file does not exist
going to connect to: utdallas.edu

Server Response: HTTP/1.1 200 OK
Ready to serve...
Received a connection from: ('127.0.0.1', 50959)
GET /www.google.com/adasfds.dsd HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.50.0
Accept: */*

/wwww.google.com/adasfds.dsd
www.google.com/adasfds.dsd
/wwww.google.com/adasfds.dsd
Cache file does not exist
going to connect to: google.com

Server Response: HTTP/1.0 404 Not Found
Ready to serve...
^C
C:\Users\Alan\Project>
```

```
Command Prompt
C:\Users\Alan>curl -ILX GET http://localhost:8888/www.google.com
HTTP/1.0 200 OK
Content-Type:text/html
HTTP/1.0 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Mon, 25 Jul 2016 13:30:55 GMT
Expires: Wed, 24 Aug 2016 13:30:55 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

C:\Users\Alan>curl -ILX POST http://localhost:8888/www.utdallas.edu
HTTP/1.1 200 OK
Date: Mon, 25 Jul 2016 13:31:33 GMT
Server: Apache
Accept-Ranges: bytes
Connection: close
Content-Type: text/html; charset=utf-8
Set-Cookie: utd1P=8fec6daec6e5853878498a4c03a9b457_1469453493;path=/;
Vary: Accept-Encoding

C:\Users\Alan>curl -ILX GET http://localhost:8888/www.google.com/adasfds.dsd
curl: (52) Empty reply from server

C:\Users\Alan>
```

f) Issues

When we did our first attempt in just trying to handle a GET request correctly we ran into several issues in where the connection was established and the webpage was correctly cached however the proxy server would send out an error 404 message although it clearly was not. At this point we have not yet tried testing the code with the

```
Command Prompt - python proxyserver.py 8888
Ready to serve...
Received a connection from: ('127.0.0.1', 63667)
GET /www.google.com HTTP/1.1
Host: localhost:8888
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,es;q=0.6

/wwww.google.com
www.google.com
/wwww.google.com
google.com
404: File Not Found
Ready to serve...
Received a connection from: ('127.0.0.1', 63669)
GET /favicon.ico HTTP/1.1
Host: localhost:8888
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36
Accept: */*
Referer: http://localhost:8888/www.google.com
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,es;q=0.6

/favicon.ico
favicon.ico
/favicon.ico
favicon.ico
404: File Not Found
Ready to serve...
```

tool Curl. We discovered that using an actual web browser as an interface with the proxy server was cumbersome and led to additional request being sent to proxy that was not in the scope of this project. Curl was designated as the interface for testing and demoing.

Once we started using curl the lack of a 404 error response from the web server was also an issue. After some debugging we found that when submitting a URL that contained info beyond the main page of a website the GET request written would become corrupt. This was caused because hostn was unable to create the socket unless we partitioned the filename. This was supposed to be handled by the skeleton code provided however we found that we needed to modify it in order to properly work. We solved this issue by partition the actual URL with a string called subfile. The subfile would be added to the get request and if the subfile was empty, meaning a URL with nothing beyond the main page, then it would just add an empty sting not affecting the get request in any way.

Other issues encountered where standard syntax and required extensive debugging through the use of an IDE. Figuring out proper partitioning methods that would not throw out of bounds errors was a challenge at first given our lack of experience in python. In the end all of the issues encountered where fixed leading to the code presented in this project.

g) Video Demo

The image displays two terminal windows side-by-side, both showing the output of a netcat listener on port 8080. The left window shows a successful connection from 127.0.0.1, receiving an HTTP GET request for www.google.com. The response is an HTTP 200 OK with headers indicating it's a text/html response from a Google server, with a cache-control of 'private, max-age=0'. The right window shows a similar request but with a more complex User-Agent string and an 'Accept' header. The response is an HTTP 302 Found, redirecting to a specific Google page.

Content Delivery Networks

Introduction

Internet traffic has increased exponentially since its inception and will continue that growth trend for the foreseeable future as mobile devices continue to gain popularity all over the world. With the explosion of the tech industry in the United States and Eastern Europe in the 1990s serving content to users on the internet became a problem, because before Content Delivery Networks, Content Providers used their own Web Servers to serve users the all the data that they had requested. A The reason that this is a problem is that a Content Provider is not necessarily well-equipped to manage the increased load on their servers and network and users that were across the country and across the globe were suffering. Content Delivery Networks, as the name suggests, is a network of machines which delivery content, which may be static or dynamic data on the web [1]. Content delivery Networks are able to solve this issue very effectively. As the worldwide internet traffic continues to grow, Content Delivery Networks are becoming more and more important. Global IP traffic will increase nearly threefold over the next 5 years, and will have increased nearly a hundredfold from 2005 to 2020. Overall, IP traffic will grow at a compound annual growth rate (CAGR) of 22 percent from 2015 to 2020 [6]. In 1998 the first CDNs begin to appear as a response to the explosion of internet traffic in the 1990s. Only two years later, the CDN market generates \$905 million, projected to reach \$12 billion by 2007. In 2002, ISPs begin deploying CDNs of their own. By 2004 more than three-thousand companies use CDNs spending more than \$20 million per month [2]. The success of the CDN market allows for huge benefits to much of the internet, increasing user experience. In less than ten years, CDNs became an integral part of the internet. In this paper we will discuss the structure of Content Delivery Networks, their motivations, history, Akamai (a major player in the industry) and their solutions and contributions to CDN architecture, and future trends for CDNs.

Motivations

As global IP traffic grows, so does the size of objects that are fetched. For example, in the infancy of the internet, a web page was typically HTML only and mostly just text. There may have been a few low quality images, but certainly not tens of megabytes per page like we see today. Now, web pages are typically much larger in size and can show very large objects such as high quality images, videos, games, etc. and this proposes a problem for the user that may have to fetch the data from a server that is hundreds or thousands of miles away which almost always means that there are many hops through the network to reach that server. What a CDN can do to improve the user experience is replicate the data that this/these user(s) are requesting on a server that is much closer to the user(s). For example, if a user that lives in New York City wants to visit a website showcasing an artist's work that is hosted on a web server in Southern California, and the artist's web hosting provider subscribes to a CDN service, that CDN could replicate the website's content on a server that is located in San Francisco. In this case, the user doesn't have to download the images from servers that are nearly three-thousand miles away, but are within tens of miles, decreasing the RTT by a significant amount. Not only is this a benefit for the Content Provider and the user, but this is a benefit to the internet itself, because this scheme decreases the amount of network traffic, thus decreasing overall network congestion.

We have seen a major bump in global internet traffic by the increase in mobile devices mostly since the deployment of third generation mobile data networks. The number of devices connected to IP networks will be three times as high as the global population in 2020. There will be 3.4 networked devices per capita by 2020, up from 2.2 networked devices per capita in 2015. Accelerated in part by the increase in devices and the capabilities of those devices, IP traffic per capita will reach 25 GB per capita by 2020, up from 10 GB per capita in 2015 [6]. These

predictions by Cisco indicate an even stronger reason to continue using and evolving CDN technologies as the population demands higher quality content at the click of a button.

CDNs are even more helpful to Content Providers (CP) and the internet when these CPs are serving content to millions of users (CNN, Netflix, etc.), because CDNs are able to use load balancing techniques to spread the traffic across multiple nodes and/or datacenters in an effort to lighten the load on the network which can help prevent queuing delay and also add redundancy for outages. Metrics for redirecting requests could include network proximity derived from network routing tables (e.g. border gateway protocol), topological proximity (i.e. depending on region), load balancing with servers (i.e. finding out the server with less load in a particular region). In case the respective server finds that content is not actually present in nearby locations, it sends a request to origin server and gets content [1]. CDNs will typically deploy multiple datacenters to a region for this reason. One practical solution for this in the past was using a proxy server to cache frequently requested web content. While this does help speed up object requests, proxies cannot be deployed on a global scale and really are only efficient for internal use. Proxy servers also do not have some of the more advanced capabilities of CDNs that we will discuss in the next section.

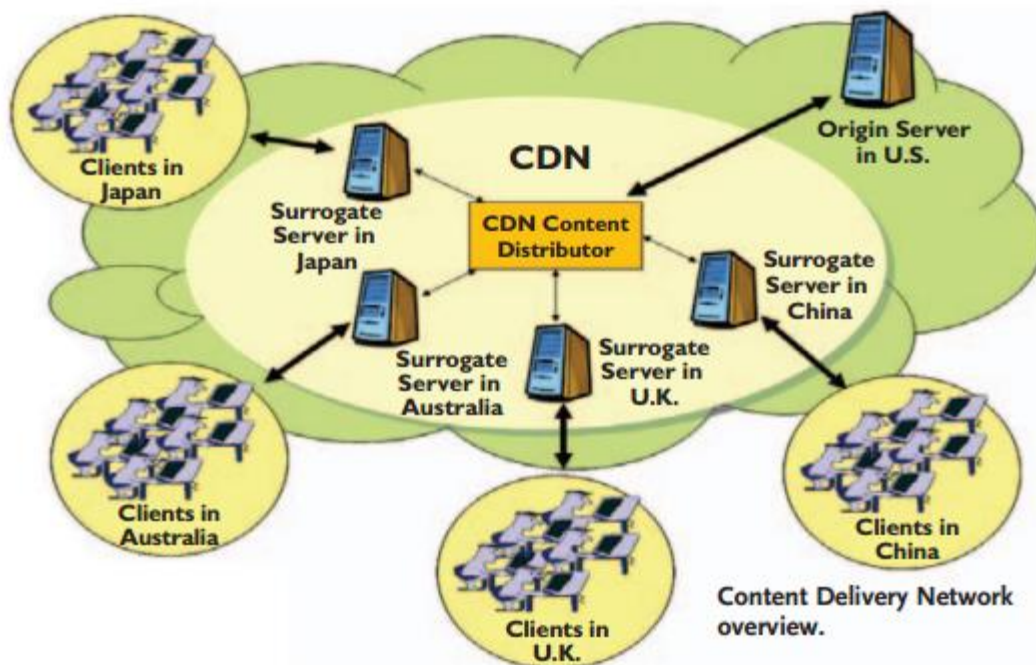
Architecture

For Content Delivery Network companies, the first and most important challenge they face in designing their infrastructure is to maximize efficiency of their network and services. Two major factors are at play here: The first is where exactly geographically will the CDNs' satellite datacenters that users access content going to be located? The second is what content are the satellite servers going to host? The geographical placement of the datacenters is possibly the most important question, because deploying a datacenter is extremely expensive and somewhat

permanent while deciding what content to replicate is easily changed, but can take time and a lot of compute resources during the process.

To choose the geographical locations of datacenters, CDNs typically use some graph algorithms to analyze Web traffic statistics. One choice for datacenter locations are nearby Internet Service Providers (ISPs). This makes a lot of sense, because the ISP's network is usually close to the users. It is possible to get even closer to the users, so this method is not the de facto location for CDNs, but we have seen many Telecommunications companies like AT&T deploying the own CDNs because of this. Another service that CDNs have to consider is international web traffic. Netflix, for example, is a US-based company, but provides service all over the globe. What CDNs can do is cache Netflix's content in other countries or near the international gateway routers. Figure 1 illustrates the ideal CDN infrastructure.

To answer the question of how a CDN can use their resources most efficiently, a CDN



company will employ several different methods depending on the type of content they will serve.

Figure 1 [2]

Zipf's Law should be considered here. This law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. It can be applied to many things including web content. For web content, what a CDN can do is cache objects that are most frequently accessed. These objects will also have a time to live (TTL) which will dictate if the object should be replaced by some other object that is more in demand at the time, much like cache memory in a computer. CDNs are also known to analyze browsing statistics to employ a prefetching technique. This involves predicting what objects users will most likely request based on their usage patterns. For example, if a user is on an e-commerce website that is having a sale, the CDN might prioritize hosting the sale items rather than the non-sale items.

Not all content delivery networks are created equally and are able to serve all types of content efficiently, especially with how fast IP traffic is growing even CDNs are often overwhelmed so some peering solutions have been proposed and implemented. Content Delivery Network Interconnection is one of these solutions. One study has found that CDNi can reduce intra-ISP Internet Exchange Point (IXP) traffic by 16.2% to 29% [3]. CDNi allows collaborative peering between CDNs to avoid hops through the IXP. During times when one CDN cannot handle spikes in content requests, CDNi can allow sharing of the popular content between CDNs located in different Autonomous Systems (AS), thus drastically reducing IXP traffic. One problem with this technology, however, is the added overhead of making these connections. If the traffic is for small objects, the overhead is a statistically significant consideration, however, current trends in content popularity is showing that size of objects is increasing, thus increasing the efficiency of CDNi.

Akamai Technology

Akamai Technologies is an industry leader in CDN services and has been a pioneer of advances in CDN technology. Akamai's technologies and architecture are very efficient and have been extremely successful because of it (more than \$2B in revenue per year). Akamai has recognized that the internet being a best-effort network, it does not provide the reliability that Akamai aimed to achieve. Specifically, congestion at peering points and BGP present some major flaws in efficiency and reliability. What Akamai has done to overcome these problems is deploy their datacenters as close to the end-users as possible to avoid peering congestions and outages and they also have created a virtual network on top of the internet. This virtual mapping of the internet allows Akamai to

basically re-engineer the internet to add reliability and decrease latency. What Akamai are able to do is use multiple best-effort routing paths of the physical internet network and use their virtual network overlay to conduct races in order to detect the best possible path for content. This virtual overlay also allows for transmitting on multiple paths for redundancy to substantially increase reliability for applications that require it. The virtual overlay also

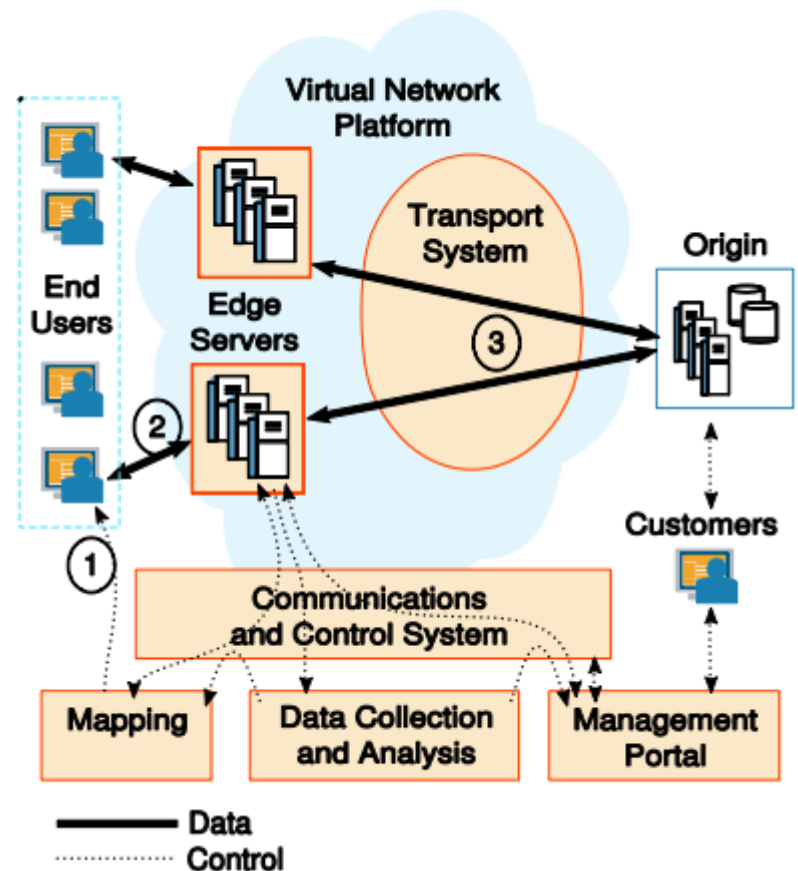


Figure 2 [4]

allows for optimization of the transport layer for Akamai-to-Akamai communications. Figure 2 shows a simplified diagram of Akamai's topology.

Akamai has engineered and employs a proprietary transport protocol in their network that improves on several aspects of TCP. This proprietary protocol eliminates overhead of establishing and destroying sessions, more aggressive window sizing using routing data obtained by their virtual network, and more aggressive retransmission timers. Because of the scale of Akamai, they have run into an issue with generating logs of all things. Logs alone generate 100 TB of data per day and these are the primary means of billing for the company. What they have done is created a way to compress the log files and distribute them to specific clusters to process them.

Some notable achievements for Akamai are: Protecting the U.S. Government networks from DDoS attacks in July 2009. The attack generated eight years' worth of traffic in a single day and Akamai's network was able to absorb 200 Gbps of traffic targeted at the government websites allowing regular visitors to continue nearly unaffected. During the height of MySpace's popularity, Akamai was able to increase the overall speed of content delivery for MySpace by 6 times and handle 98% of traffic. Using Akamai's services, the New York Post was able to increase the performance of its website by twenty times.

Conclusion

Content Delivery Networks have become an integral piece of the global internet infrastructure. CDNs have continually shown to be an effective way to improve the best-effort network of the internet and drastically improve user experience. As we see in Cisco's predictions, internet traffic will continue increase exponentially on a global scale and as less developed countries begin to match the number of users in developed parts of the world, the importance of CDNs will only increase as well. Streaming content has become extremely popular in the last few years and new protocols and network topologies are being researched to

improve streaming performance as well. Mobile networks are the most complex networks and because of handoffs between networks, this is a hurdle that CDN companies are also attempting to handle better as well.

Reference List

- [1] HCL Technologies, “Content Delivery Networks: An Introduction.” Internet: <http://userpages.umbc.edu/~dgorin1/451/caching/contentdel.pdf>, May 2002 [July 23, 2016].
- [2] G. Pallis, A. Vakali, “Insight and Perspectives for Content Delivery Networks,” *Communications of the ACM*, pp.101-106, January 2006. [Online] Available: <https://dl.acm.org/citation.cfm?id=1107462>
- [3] G. Nam, K. Park, “Analyzing the Effectiveness of Content Delivery Network Interconnection of 3G Cellular Traffic,” in *CFI '14 Proceedings of The Ninth International Conference on Future Internet*, Tokyo, Japan, 2014, Article No. 1. [Online] <https://dl.acm.org/citation.cfm?id=2619297>
- [4] E. Nygren, R. Sitaraman, J. Sun, “The Akamai Network: A Platform for High-Performance Internet Applications,” *ACM SIGOPS Operating Systems Review*, Volume 44, Issue 3, July 2010 pp.2-19 [Online] <https://www.akamai.com/it/it/multimedia/documents/technical-publication/the-akamai-network-a-platform-for-high-performance-internet-applications-technical-publication.pdf>
- [5] L. Kontothanassis, *et al*, “A Transport Layer for Live Streaming in a Content Delivery Network,” *Proceedings of the IEEE*, Volume 92, Issue 9, Sept. 2004 pp.1408-1419 [Online] http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1323289&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxppls%2Fabs_all.jsp%3Farnumber%3D1323289
- [6] Cisco Systems, “Cisco Visual Networking Index: Forecast and Methodology, 2015–2020.” Internet: <https://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf> [July 23, 2016].