Content Distribution Networks

A content distribution network (CDN) is a system of geographically distributed servers that deliver web content, such as images, videos, and documents, to end users. The main goals of a CDN are to improve the performance, availability, and security of web content delivery.

Server Placement Philosophies

One of the key design decisions for a CDN is where to place its servers. Different CDNs may adopt different server placement philosophies, depending on their objectives and constraints. Two common philosophies are:

- Enter deep: This philosophy aims to place CDN servers as close as possible to the end users, usually in the access networks of Internet Service Providers (ISPs). This can reduce the latency and bandwidth consumption of content delivery, as well as increase the resilience to network failures and congestion. However, this philosophy also requires more servers and more cooperation from ISPs. An example of a CDN that follows this philosophy is Akamai.
- Bring home: This philosophy aims to place CDN servers in a few strategic locations within the network topology, such as major Internet exchange points or backbone networks. This can reduce the operational cost and complexity of managing a large number of servers, as well as leverage the existing network infrastructure and peering relationships. However, this philosophy may not provide optimal performance for some end users who are far away from the CDN servers. An example of a CDN that follows this philosophy is Google.

CDN Server Selection Strategies

Another key design decision for a CDN is how to select which server to serve a content request from an end user. Different CDNs may adopt different server selection strategies, depending on their criteria and mechanisms. Two common strategies are:

- DNS-based: This strategy relies on the Domain Name System (DNS) to direct end users to the appropriate CDN server. When an end user requests a web content with a domain name that is managed by a CDN, the CDN's authoritative DNS server responds with the IP address of a CDN server that is selected based on some criteria, such as geographic proximity, network conditions, or server load. The end user then contacts the CDN server directly to retrieve the content. This strategy is simple and scalable, but it may not be very accurate or adaptive, as it depends on the DNS resolution process and the DNS caching behavior. An example of a CDN that uses this strategy is Cloudflare.
- Application-based: This strategy relies on the application layer to direct end users to the appropriate CDN server. When an end user requests a web content with a domain name that is managed by a CDN, the CDN's authoritative DNS server responds with the IP address of a generic CDN server that acts as a redirector. The redirector then analyzes the end user's request and redirects it to another CDN server that is selected based on some criteria, such as content availability, network performance, or user preference. The end user then contacts the redirected CDN server to retrieve the content. This strategy is more flexible and dynamic, but it may introduce more overhead and complexity, as it requires an additional redirection step and more application logic. An example of a CDN that uses this strategy is Netflix.

Examples of CDNs: Netflix and YouTube

Netflix and YouTube are two popular video streaming services that use CDNs to deliver their content to millions of users around the world. However, they have different approaches to building and managing their CDNs.

Netflix has built its own custom global CDN, called Open Connect, which consists of two main components: embedded Open Connect Appliances (OCAs) and settlement-free interconnection (SFI). OCAs are servers that store copies of Netflix videos and are deployed in various locations, such as ISP data centers or Internet exchange points. SFI is a peering arrangement that allows Netflix to connect its OCAs directly to ISP networks without paying any fees. Netflix uses both DNS-based and application-based server selection strategies to direct end users to the best OCA for their requests.

YouTube uses Google's global CDN, which is part of Google's larger infrastructure that supports various Google services. Google's CDN consists of thousands of servers distributed in hundreds of locations around the world, connected by Google's own private network backbone. Google uses mainly DNS-based server selection strategy to direct end users to the closest or best-performing server for their requests. However, Google also uses application-based server selection strategy for some scenarios, such as live streaming or adaptive bitrate streaming.

HTTP 1.x

Key Characteristics:

1. **Text-Based:** HTTP 1.x is text-based, with both requests and responses being human-readable. This simplicity made it easy for developers to debug, but it also resulted in larger message sizes and higher bandwidth consumption.
2. **Stateless:** Each HTTP request-response cycle is independent, and the server doesn't maintain any knowledge of previous interactions. This statelessness simplifies server design but can lead to inefficiencies when multiple requests are needed to load a web page.
3. **Blocking:** HTTP 1.x is blocking in nature, meaning that the client must wait for a response to a request before making the next request, even if there are other resources it could be fetching concurrently. This blocking behavior can cause latency and slower page loading times.
4. **Connection Overhead:** A new TCP connection is typically established for each request-response cycle. This results in connection overhead and can slow down web page loading, especially when many resources need to be fetched.

HOL Problem in HTTP 1.x:

HTTP 1.x, the first generation of the HTTP protocol, uses a simple and straightforward request-response model, where each client request is processed sequentially. While this model is easy to understand, it leads to the Head-of-Line (HOL) problem.

1. Serial Processing: In HTTP 1.x, the client sends a request for a resource, and the server processes it one at a time. The server can't start processing another request until it has completed the current one. This sequential processing of requests and responses creates a bottleneck.
2. Blocked Resources: If a web page includes multiple resources (e.g., HTML, CSS, JavaScript, images, and more), and any of these resources takes a while to load, it blocks subsequent resources from being requested and loaded. For example, if an image resource is slow to load, it can delay the loading of critical resources like CSS or JavaScript files.
3. Impact on Page Load Time: The HOL problem directly impacts the page load time. Since resources are loaded serially, the total time to load a web page can be significantly longer, especially if there are latency issues or slow-loading resources. Users may experience slower page rendering and interactivity.
4. Latency Issues: High network latency further exacerbates the HOL problem. For each resource request, there's a delay due to the round-trip time between the client and server. This latency adds up when multiple resources need to be fetched.

Example:

Imagine a scenario where a web page consists of several resources:

- HTML document
- CSS stylesheet
- JavaScript files

- Multiple images

In an HTTP 1.x scenario, if the HTML document is slow to load due to a large size or network latency, the subsequent resources, such as CSS, JavaScript, and images, will be delayed. The browser has to wait for the HTML document to finish loading before it can parse it and discover other resources that need to be fetched. This results in a situation where even fast-loading resources are held back by slower resources.

This delay has a cascading effect, leading to slower web page rendering and reduced user experience. The HOL problem makes it difficult to fully utilize available network bandwidth and parallelize resource retrieval effectively.

HTTP/2.0, with its multiplexing and concurrent processing capabilities, addresses the HOL problem by allowing multiple resources to be requested and delivered in parallel over a single connection, significantly improving web page loading times and user experience.

HTTP/2.0

Key Features:

1. **Binary Framing:** HTTP/2.0 uses a binary protocol for its messages. Data is divided into smaller frames, which are more efficiently processed by both the client and server. Binary framing allows for better multiplexing and prioritization of requests and responses.
2. **Multiplexing:** Multiplexing is one of the most significant improvements in HTTP/2.0. It allows multiple requests and responses to be sent concurrently over a single connection. This reduces latency, as clients no longer have to wait for one resource to download before requesting another.
3. **Header Compression:** HTTP/2.0 uses header compression techniques to reduce redundancy in header data. This reduces the overhead of sending headers with each request and response, making data transfer more efficient.
4. **Server Push:** With HTTP/2.0, servers can proactively push resources to the client. For example, if a web page requires multiple resources, the server can push some of these resources to the client before the client requests them. This feature improves the efficiency of loading complex web pages.

Binary Frame Structure:

Binary frames in HTTP/2.0 are the fundamental units used to structure and transmit data efficiently over the network. These frames are essential for multiplexing, flow control, and prioritization. Let's dive into the binary frame structure and its components with an example:

A binary frame in HTTP/2.0 consists of three parts:

1. **Frame Header:** This is a fixed-size section of the frame that contains essential information such as the frame length, type, flags, stream identifier, and, in some cases, a priority field.
2. **Frame Payload:** This is the variable-sized portion of the frame that carries the actual data. The structure of the payload depends on the frame type.

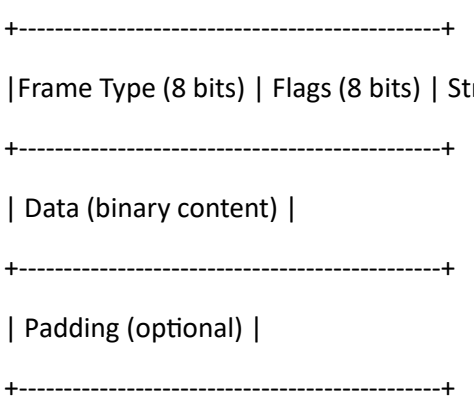
3. **Padding:** Padding can be added to a frame if necessary, but it's optional. Padding is used to increase the size of the frame, for example, to obfuscate the actual payload size.

Example: A DATA Frame

Let's take a look at the structure of a DATA frame in HTTP/2.0 as an example:

- **Frame Type:** DATA (0x0)
- **Flags:** DATA frames have three flags: END_STREAM (0x1), END_HEADERS (0x4), and PADDED (0x8). These flags indicate, respectively, whether this is the last frame for the associated stream, the end of the header block, and if padding is included.
- **Stream Identifier:** A 31-bit identifier indicating which stream this frame belongs to.
- **Payload:** The actual binary data that the frame carries. This is typically the content of an HTTP message or a part of it.
- **Padding:** If the PADDED flag is set, there may be padding included, and the length of the padding is determined by the length of a padding field. The padding field can be up to 255 bytes long.

Here's how a DATA frame might look in binary for



Example: Sending a DATA Frame

Let's say you're the client, and you're sending an HTTP request for an image resource to the server:

1. You initiate a new stream with the server and send a DATA frame with the following structure:
    - Frame Type: DATA
    - Flags: END_STREAM (indicating that this is the last frame for the stream)
    - Stream Identifier: 1 (assuming this is the first stream you've created)
    - Payload: The binary content of the HTTP request, which includes the resource URL, headers, and other request data.
    - Padding: None in this case (PADDED flag is not set).

2. The server receives the DATA frame, processes the request, and responds with a DATA frame of its own, containing the binary content of the image you requested.

The binary frame structure and multiplexing capabilities of HTTP/2.0 allow multiple streams to be managed concurrently, optimizing data transmission and improving the overall performance of web applications.

HTTPS (HTTP Secure)

SSL (Secure Sockets Layer):

1. **Encryption:** SSL was the original protocol designed for securing data transmission between a client and a server. It uses encryption algorithms to ensure that data exchanged over the network is encrypted, making it unreadable to any third parties intercepting the traffic.
2. **Authentication:** SSL also provides a level of authentication. It verifies that the server is indeed the one it claims to be, helping to prevent man-in-the-middle attacks.

TLS (Transport Layer Security):

1. **Encryption Standards:** TLS builds upon SSL but uses more modern and secure encryption standards. It has evolved to address vulnerabilities found in SSL.
2. **Security:** TLS is generally considered more secure and robust than SSL. It's the protocol that is used to secure most HTTPS connections today.
3. **Versions:** SSL has multiple versions, including SSL 2.0 and SSL 3.0, which have known security issues. TLS, on the other hand, has versions like TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3, with each newer version improving security and performance. TLS 1.3, in particular, focuses on minimizing latency.

In summary, HTTP/2.0 brings significant improvements over HTTP 1.x, including binary framing, multiplexing, header compression, and server push. HTTPS, secured by SSL or TLS, is vital for protecting data in transit, with TLS being the more secure and modern option compared to SSL, due to its stronger encryption standards and better security practices.

Exchange of Messages between Host and Server in HTTPs:

Client (Host): Your web browser, for example, Google Chrome.

Server: The web server hosting a website, such as an online shopping site like "ExampleShop.com."

Message Exchange Steps in HTTPS:

1. **Client Hello (TLS Handshake):**
    - **Client**: The client initiates the connection by sending a "Client Hello" message to the server.
    - **Server**: The server receives the "Client Hello" message and processes it.
2. **Server Hello (TLS Handshake):**

- **Server**: The server responds with a "Server Hello" message, which includes information like the selected encryption protocol, a random number, and the server's digital certificate.
- **Client**: The client receives the "Server Hello" message and checks the server's certificate for authenticity. The certificate might be issued by a trusted Certificate Authority (CA). If it's trusted, the client proceeds.

3. **Key Exchange (TLS Handshake):**
    - **Client**: The client generates a random "pre-master secret" and encrypts it with the server's public key (extracted from the server's certificate).
    - **Client**: The client sends this encrypted "pre-master secret" to the server.
    - **Server**: The server decrypts the "pre-master secret" using its private key.
4. **Session Key Derivation (TLS Handshake):**
    - Both the client and server use the "pre-master secret" to independently derive a shared "session key." This key is used for encrypting and decrypting the actual data exchanged during the session.
5. **Finished Messages (TLS Handshake):**
    - Both the client and server send "Finished" messages to confirm that they have successfully established a secure connection.
    - The "Finished" messages are encrypted using the shared session key, ensuring that further communication will be secure.
6. **Data Exchange (HTTP Messages):**
    - With the secure TLS connection established, the client and server can exchange data, such as HTTP requests and responses.
    - **Client**: The client sends an HTTP request, such as requesting the homepage of "ExampleShop.com."
    - **Server**: The server receives the request, processes it, and sends back the requested web page, along with an encrypted HTTP response.
7. **Secure Data Exchange (TLS Encryption):**
    - All data exchanged between the client and server, including the web page content, is encrypted using the shared session key.
    - This encryption ensures that any data intercepted during transmission remains unreadable to eavesdroppers.
8. **End of Session:**
    - When the client has received the data it needs, it may choose to terminate the session. The client and server exchange "goodbye" messages and close the connection.

Example Scenario:

Imagine you're shopping online on "ExampleShop.com." Your web browser establishes a secure HTTPS connection with the server hosting the website. During the TLS handshake, encryption keys are exchanged, ensuring that your sensitive information, such as your payment details, remains confidential. When you make a purchase, your payment details are encrypted and securely transmitted to the server, providing you with a safe and private shopping experience.