Arquitectura - Primeros Pasos

El siguiente material está basado en el capítulo 1 del libro "System Design Interview" de Alex Xu

Versión 1.0

Octubre 2023

Autor: Uriel Soifer

Revisado por: Ezequiel Escobar y Lucas Saclier

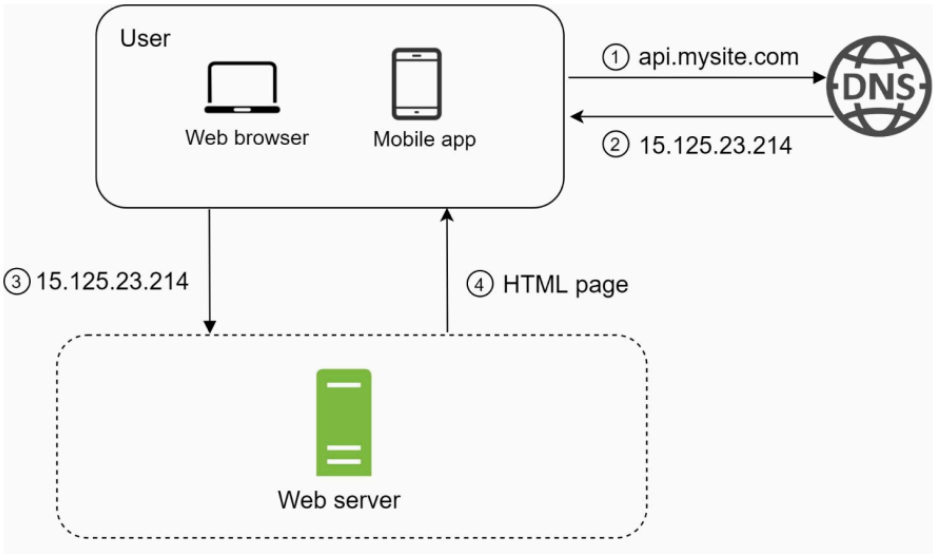
Índice

Diseñando una arquitectura simple

A la hora de diseñar una arquitectura para un determinado sistema son muchas las variables a tener en cuenta. La complejidad radica en que las variables dependen del sistema y de los objetivos que tengamos y decidamos priorizar para el mismo. Además, siempre sucede que el diseño nos embarca en un camino que requiere un continuo perfeccionamiento y una mejora sin fin.

Sin embargo, todos los caminos tienen un comienzo: vamos a plantear una arquitectura simple y complejizarla poco a poco hasta llegar a un sistema que soporte millones de usuarios, evaluando las distintas alternativas posibles junto con las ventajas y desventajas de cada una.

Para comenzar, vamos a plantear una arquitectura simple como mencionamos anteriormente donde tenemos un único servidor que ejecute una: aplicación web, base de datos, caché, etc.



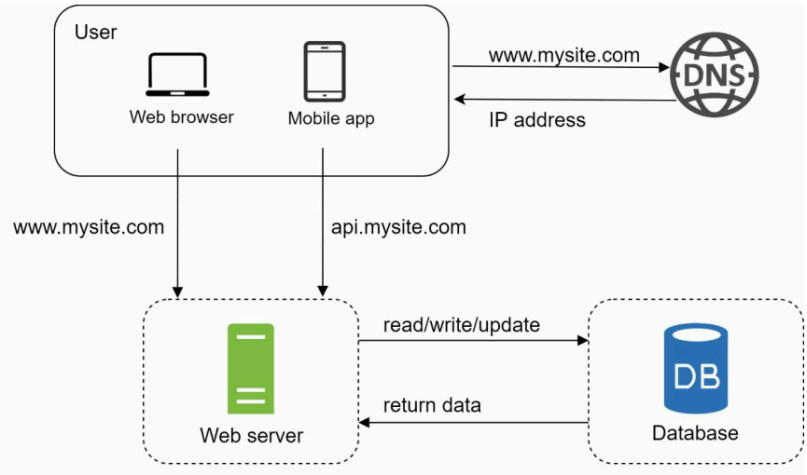
El camino en este caso resulta sencillo:

1. El usuario accede a un DNS, el cual vale la pena mencionar pero por ahora no le vamos a prestar mucha atención ya que este es un sistema independiente y su funcionamiento no depende de nosotros.
2. El DNS retorna una IP
3. El dispositivo utiliza la IP obtenida por el DNS para enviar una request al web server
4. Dependiendo de la request, el web server realiza el procesamiento necesario y puede devolver un: HTML, JSON, CSS, JS, etc.

Base de datos

Cuando el sistema comienza a tener una mayor cantidad de usuarios activos y se vean afectados la performance del servidor o hasta incluso la disponibilidad del mismo; resulta necesario la elaboración de una arquitectura más cohesiva, de forma tal que no todo recaiga en un único servidor.

El primer paso será separar la base de datos para que esta pueda ser escalada de forma independiente. De esta forma, *nuestra arquitectura ya posee dos capas: una para el tráfico web y otra para los datos.*



¿Qué base de datos me conviene utilizar?

Actualmente en el mercado encontramos dos clasificaciones de bases de datos a alto nivel, las cuales a su vez poseen distintas implementaciones:

- Bases de datos relacionales (RDBMS): MySQL, Oracle database, PostgreSQL, entre otras.
- Bases de datos no relacionales: Neo4j (grafos), MongoDB (documentales), Cassandra (wide column), Amazon DynamoDB (clave-valor), entre otras.

Sin profundizar:

- Las bases de datos relacionales son ideales para datos que tienen una estructura clara y relaciones bien definidas entre tablas.
- Las bases de datos relacionales son altamente confiables para el control de transacciones, lo que garantiza que las operaciones se realicen de manera consistente y segura, evitando la corrupción de datos y asegurando la consistencia en caso de fallos.

Mientras que una base de datos no relacional se podría utilizar cuando:

- Los datos no son estructurados: como es el caso de un JSON o información proveniente de sensores, las bases de datos no relacionales son más adecuadas ya que pueden manejarlos de manera más flexible sin la necesidad de definir un esquema rígido.
- Se necesita almacenar una gran cantidad de datos o escalar horizontalmente de manera sencilla.

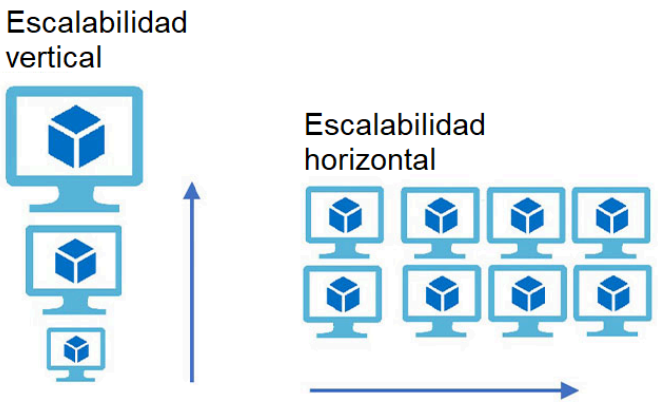
Escalando la capa de tráfico web

Escalabilidad

Cuando un servidor debe atender muchas solicitudes, este se transforma en un cuello de botella y comienza a degradar la performance debido a que se registran tiempos de respuesta muy elevados. Incluso, hasta se puede afectar también la disponibilidad ya que si dicho servidor se cae, todos los clientes quedarían sin poder realizar sus solicitudes.

Escalabilidad horizontal vs Escalabilidad vertical en la capa de tráfico web

Escalabilidad vertical se refiere al proceso de añadir más recursos (CPU, RAM, etc) a los servidores que ya se encuentran en funcionamiento. Existe una limitación física (que la determina el motherboard y el procesador) ya que llegará un punto en el cual no se podrán agregar más recursos al servidor. Por otro lado, este tipo de escalabilidad no soluciona el problema de disponibilidad (ya que el servidor se puede caer igual), sino solamente pretende resolver el problema de performance.

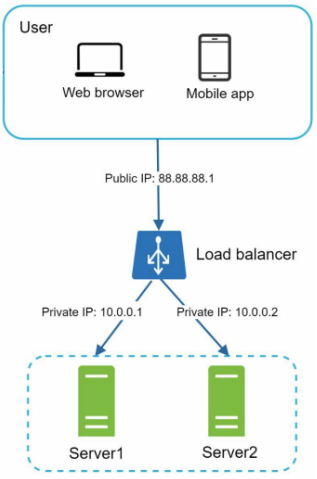


Mientras que la escalabilidad horizontal implica agregar más servidores a la arquitectura. Esta alternativa posee una complejidad extra ya que se debe contar con un balanceador de carga (*load balancer*) para distribuir las solicitudes entre todos los servidores existentes. Por último, este tipo de escalabilidad pretende solucionar el problema de disponibilidad y performance.

Cuando el tráfico es bajo, la escalabilidad vertical es una excelente opción, y la simplicidad de la misma es su principal ventaja. Por otra parte, la escalabilidad horizontal es una mejor opción para aplicaciones a gran escala debido a las limitaciones de la escalabilidad vertical.

Load balancer

En el diseño anterior, los usuarios se conectan directamente al servidor web. Los usuarios no podrán acceder al sitio si el web server está fuera de línea. En otro escenario, si muchos usuarios acceden al servidor web simultáneamente y este alcanza el límite de carga del servidor, los usuarios generalmente experimentan respuestas más lentas o directamente no logran conectarse al servidor.



Un balanceador de carga es la mejor técnica para abordar estos problemas ya que él mismo distribuye el tráfico entrante entre los servidores disponibles. Si bien esta solución suena sencilla y es muy utilizada, trae consecuencias en las que vale la pena detenerse posteriormente.
Como se puede apreciar en la imagen, los usuarios no se conectan directamente con los servidores, sino que lo hacen a través del load balancer. Esto significa que además, estamos sumando una capa extra de seguridad ya que los servidores no tienen IPs públicas. Se comunican entre sí a través de IPs privadas.

Con el load balancer se soluciona el problema de la disponibilidad del web server de la siguiente manera:

- Si se desconecta el servidor 1, el load balancer dirige todo el tráfico al servidor 2. Esto permite que el sitio web siga siendo accesible.
- Si el tráfico del sitio web crece rápidamente y dos servidores no son suficientes para manejarlo, el balanceador de carga puede manejar este problema fácilmente. Si se agregan nuevos servidores a la red, el load balancer comenzará a enviar solicitudes a los nuevos también.

Sin embargo el load puede transformarse en un único punto de falla (single point of failure), ya que ante la falla de éste todos los usuarios experimentarían problemas en el uso del sistema. Para que esto no suceda, es esencial que se cuente con una estrategia de redundancia. Esto implica, por ejemplo, la implementación de duplicados del load balancer o la configuración de un sistema de respaldo para garantizar que, en caso de que un load balancer falle, otro esté listo para asumir la carga de tráfico sin interrupciones.

Sesiones

Una sesión es un mecanismo para mantener información y estado entre diferentes solicitudes y respuestas entre el cliente y el servidor. Cuando un cliente (como un navegador web) se conecta a un servidor web, se crea una sesión única para ese cliente. Durante la sesión, el servidor puede almacenar datos relevantes para el cliente, como información de inicio de sesión por ejemplo. Esto permite que el servidor recuerde al cliente y le brinde una experiencia personalizada.

Cookies

Las cookies son pequeños archivos que se almacenan del lado del cliente. Las cookies se envían automáticamente con cada solicitud al servidor, lo que permite que el servidor identifique al cliente y recupere información relacionada con la sesión. Las cookies son comúnmente utilizadas para mantener la autenticación del usuario, recordar preferencias, etc.

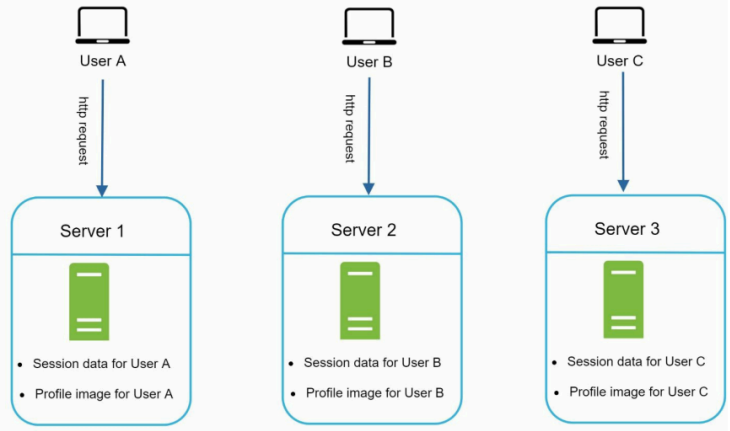
Arquitectura stateless vs arquitectura stateful

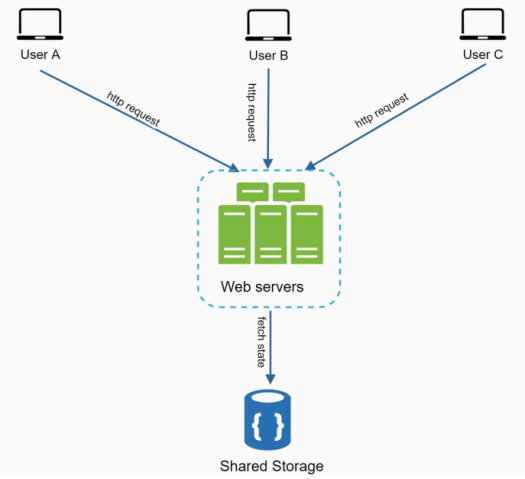
En una arquitectura stateless, el servidor no almacena información sobre la sesión. Cada solicitud que realiza el cliente se trata de manera independiente y no se mantiene ninguna información sobre interacciones anteriores. Esto no significa que del lado del cliente no se almacene información de la sesión, al contrario: cada solicitud debe contener toda la información necesaria para que el servidor comprenda y responda adecuadamente. Esto permite escalar la arquitectura de forma más sencilla, ya que los servidores no necesitan mantener información de estado.

Por otro lado, en una arquitectura stateful el servidor mantiene información sobre el estado del cliente entre solicitudes. Cada solicitud se procesa teniendo en cuenta el contexto y el estado previo. El servidor puede almacenar datos del cliente en la memoria, en una base de datos, en archivos, entre otros medios persistentes, para referencias futuras. Si bien las arquitecturas stateful pueden proporcionar una experiencia más fluida y coherente para el usuario, también pueden requerir una mayor complejidad en el manejo del estado y pueden ser escaladas con mayor dificultad en comparación con las arquitecturas stateless.

Consecuencias del uso de load balancer en una arquitectura stateful

Cuando tenemos múltiples servidores en una arquitectura stateful, el primer problema que se presenta es cuál de todos los disponibles debe guardar la sesión. Para eso podríamos pensar en múltiples soluciones, pero la más simple sería utilizando el algoritmo de ***sticky session***. Este algoritmo consiste en que las request de un cliente sean enviadas siempre al mismo servidor como se ve en la imagen.





Sin embargo, el algoritmo mencionado anteriormente resulta sencillo de implementar pero trae consigo la desventaja de que en el caso de que se caiga el servidor que tenía alojada la sesión para un determinado cliente el mismo experimente problemas en el uso del sitio. Por lo tanto, otra opción podría ser guardar toda la información que respecta la sesión en una BD (puede ser relacional o NoSQL) compartida entre todos los servidores; o bien en archivos compartidos.

Escalando la capa de datos

Escalabilidad horizontal vs Escalabilidad vertical en la capa de datos

Al igual que en la capa de tráfico web, en la capa de datos podemos escalar las bases de datos de forma horizontal o de forma vertical.

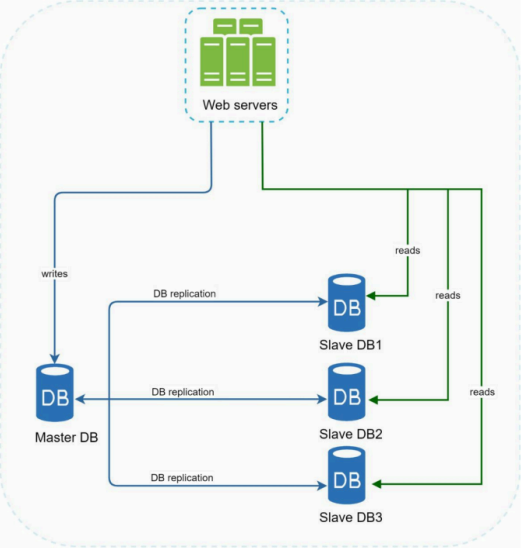
Escalar verticalmente suele ser sencillo ya que simplemente hay que agregarle más recursos a la base de datos. Sin embargo, más allá de la limitación física que tengamos estaríamos generando un nuevo SPOF (Single Point Of Failure). Además, suele ser económicamente más costoso un servidor con mayor capacidad que un nuevo servidor con menos recursos.

Por otro lado, cuando hablamos de escalabilidad horizontal en la capa de datos es un poco más complejo ya que las bases de datos deberían guardar cierto nivel de consistencia entre ellas. Es decir que podemos contar con un load balancer para que distribuya las distintas peticiones que van llegando pero sin algún mecanismo que realice una suerte de actualización entre ellas sería un problema escalar las bases de datos de forma horizontal ya que ante mismas peticiones tendríamos distintos resultados. A continuación veremos dos técnicas para escalar la base de datos de forma horizontal junto con las ventajas y desventajas de cada una.

Replicación de base de datos

La replicación de bases de datos puede ser utilizada en muchos de los sistemas de gestión de base de datos ya existentes. La misma opera teniendo como mínimo dos bases de datos:

- Base de datos master: soporta operaciones de escritura (insert, update y delete)
- Base de datos réplicas: únicamente soporta operaciones de lectura (select)



Para el siguiente ejemplo se proponen 4 bases de datos: 1 master y 3 réplicas, sin embargo esto puede variar dependiendo de las necesidades del sistema, aunque normalmente el número de réplicas suele ser ampliamente mayor.

Normalmente por cada escritura que se hace en el master tendremos 3 escrituras más para mantener la consistencia en las réplicas, aunque esto puede cambiar dependiendo nuestras necesidades. Podríamos optar por replicar la información cada cierto tiempo o cuando se modifiquen ciertos datos. Independientemente de la política que optemos para replicar los datos, resulta pertinente replantearse si es que realmente vale la pena la replicación de base de datos. La respuesta que encontramos es que esta propuesta resulta útil ya que en

muchos de los sistemas existentes la cantidad de operaciones de lectura son muchas más que las de escritura.

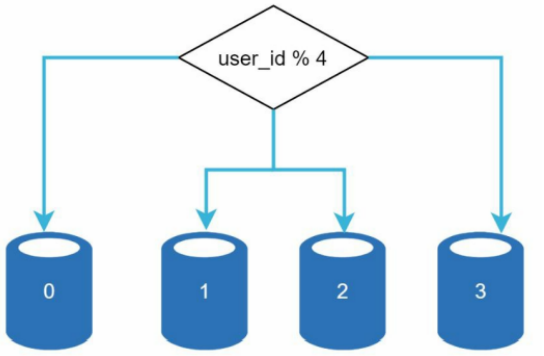
Las ventajas de la replicación de datos son:

- Rendimiento: esta propuesta mejora el rendimiento ya que la cantidad de consultas que se procesan en simultáneo aumenta drásticamente debido a que las consultas se distribuyen en distintas bases de datos y no siempre las atiende la misma (máster)
- Fiabilidad: las bases de datos pueden encontrarse en distintos puntos geográficos, lo que garantiza que ante la pérdida de una por el motivo que fuese encontremos una copia de los datos en otra locación.
- Alta disponibilidad: como se mencionó anteriormente, si las bases de datos se encuentran en distintos puntos geográficos además de fiabilidad se garantiza la disponibilidad ya que el sistema puede seguir en funcionamiento. Lo interesante es discutir cómo actúa el sistema cuando una de las bases deja de funcionar:
    - Si la réplica deja de funcionar las operaciones de lectura serán redirigidas al máster únicamente en el caso de que haya solo una replica
    - Si el master deja de funcionar es un poco más complejo ya que se debe tomar a una réplica y utilizarlo como master pero puede ocurrir que la réplica no esté actualizada por lo que se debe correr scripts de recuperación de datos antes de actuar como master.

Sharding o fragmentación

El proceso de sharding o fragmentación consiste en separar grandes bases de datos en muchas pequeñas que son más sencillas de manejar, donde a cada base de datos se la llama shard o fragmento. Todos los fragmentos tienen el mismo esquema pero no hay datos repetidos entre ninguno de ellos.

Un ejemplo de sharding podría ser teniendo 4 fragmentos para guardar la información de los usuarios. Para determinar en qué shard se encuentra la información de un usuario en particular se utiliza una función de hash para acceder al shard correspondiente. En nuestro caso, la función es *user_id % 4.*

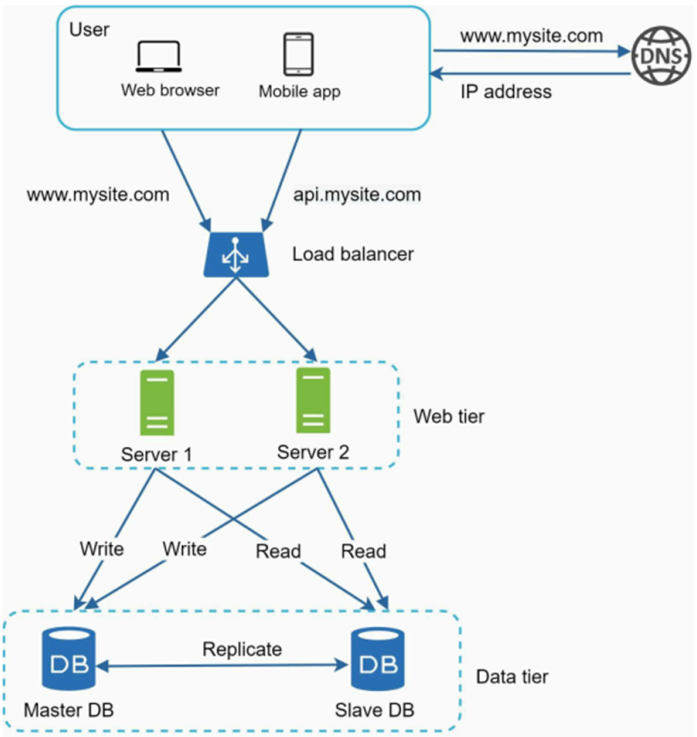


Un punto clave para el correcto funcionamiento del sharding es la clave de fragmentación. La misma es la encargada de dividir los datos entre los shards que se encuentren disponibles, en el ejemplo anterior la clave de fragmentación era el *user_id* y podemos suponer que la misma tenía un valor autoincremental de a 1 lo que permite un equilibrio perfecto en la distribución de datos (sin tener en cuenta que se pueden eliminar registros).

El sharding es una buena técnica para escalar la capa de datos de forma horizontal pero está lejos de ser la solución perfecta ya que introduce nuevas complejidades:

- *Resharding de datos*: por diversos motivos puede ocurrir que sea necesario hacer un resharding de los datos, es decir volver a organizar cómo se reparten los datos entre los distintos shards. Esto sucede cuando: un shard no puede contener más datos o ciertos fragmentos crecieron muy rápido debido a una distribución desigual de datos. Cuando esto ocurre se realiza un resharding actualizando la función de fragmentación. Es un proceso que puede llevar su tiempo y siempre que sea posible es mejor evitarlo.
- *Problema de las celebridades*: esto ocurre cuando datos que se utilizan frecuentemente son colocados en un mismo shard. Este shard tendrá tantos accesos que los usuarios experimentaran una respuesta lenta y hasta en ocasiones una caída del mismo. Es por eso que los datos que necesitan un acceso frecuente no deberían estar en el mismo shard. Incluso, podrían tener un shard exclusivo para alojar cada uno de esos datos en particular.
  Se lo conoce como problema de las celebridades ya que en el caso de una red social, las celebridades no pueden encontrarse en el mismo shard ya que esto provocaría una sobrecarga del mismo.
- *Join y desnormalización*: cuando los datos son fragmentados en distintas bases de datos es difícil realizar operaciones de unión. Una solución común a este problema suele ser la denormalización de la base de datos para que las consultas se puedan realizar en una sola tabla.

Tomando los conocimientos de esta sección y a partir de la arquitectura con la que veníamos trabajando llegamos a lo que se puede apreciar en la imagen. Donde tenemos:

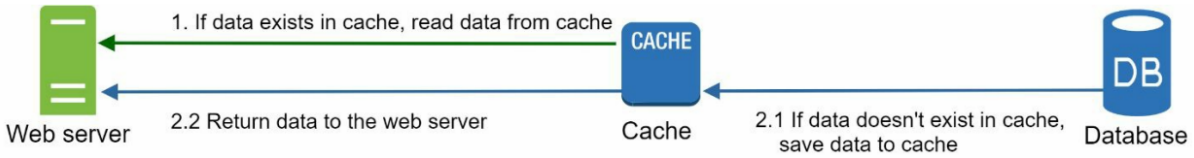


- Dos capas claramente diferenciadas: de tráfico web y de datos. Cada una escalada de forma independiente.
- La capa de tráfico web fue escalada horizontalmente, utilizando una arquitectura stateless. Es por eso que no se agregó una base de datos que mantenga datos de la sesión ni se aplicó el algoritmo de sticky session o alguno que cumpla el mismo objetivo.
- Por otro lado, la base de datos también fue escalada horizontalmente utilizando la técnica de replicación de base de datos. En este caso tenemos un master y una réplica.

Caché

La memoria caché es un almacenamiento temporal de datos cuya principal ventaja es la velocidad de acceso. Es por eso que se utiliza para almacenar datos a los que se acceden muy frecuentemente. La caché puede situarse en varios puntos de la arquitectura, como por ejemplo: entre el cliente y el servidor de aplicación, entre el servidor de aplicación y la base de datos, entre otros.

El momento en que más accesos se suelen hacer a la base de datos suele ser cuando el usuario carga el sitio por primera vez. Para reducir la carga de la base de datos se utiliza una memoria caché como estrategia.



La caché de base de datos funciona de la siguiente manera: cuando el web server recibe una request se fija si la caché tiene almacenada la información suficiente para contestar la request, evita el acceso a la BD. En caso contrario envía un query a la base de datos y almacena el resultado en la caché.

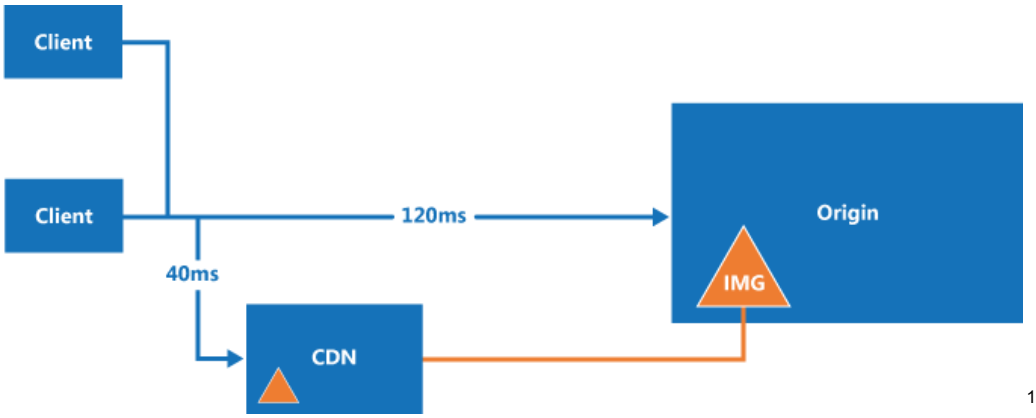
Consideraciones para el uso de memoria caché:

- Su utilización suele ser adecuada cuando tenemos data que es leída frecuentemente pero no es modificada frecuentemente
- No es aconsejable almacenar datos en una caché de forma permanente, se recomienda la implementación de una política de caducidad. Es decir que los datos se borren cada un cierto tiempo, suficientemente largo como para que la cache cumpla su objetivo y lo suficientemente corto como para que los datos en la caché no queden desactualizados
- Se debe tener en cuenta que en el caso de que haya un fallo en la memoria caché, el sistema sepa cómo recuperar los datos de la fuente original ya que sino la memoria caché se transforma en un único punto de fallo (Single Point Of Failure)
- Por último, se debe también pensar en la política de desalojo. Una vez que la caché está llena se puede aplicar algún algoritmo como LRU (Last Recent Used), LFU (Least Frequently Used) o algo mas simple como un FIFO (First In First Out) para determinar cuál es el dato que se puede remover.

Content Delivery Network (CDN)

Una CDN es un tipo de caché. Consiste en una red de servidores dispersos por todo el mundo que se utilizan para proveer contenido estático como: imágenes, videos, css, JavaScript, etc.

La CDN funciona proporcionando el contenido estático del servidor que se encuentre más cercano geográficamente. Por ejemplo: si hay un servidor en San Pablo y otro en Frankfurt la CDN se encargará de enviar el contenido desde el servidor de San Pablo a todos los usuarios que se encuentren en Buenos Aires, mejorando así la velocidad de carga del sitio.



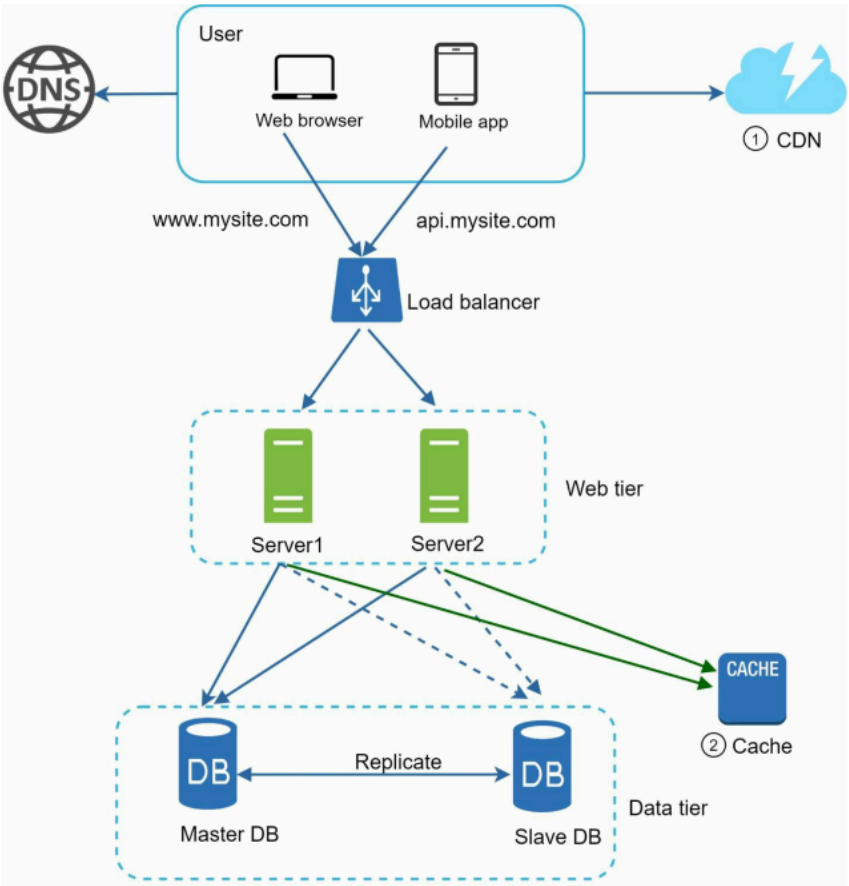
[1]

Como se mencionó en el apartado anterior, el tiempo que los datos (archivos estáticos para el CDN) son almacenados no debe ser ni muy largo ni muy corto. Es por eso que para medir esto se utiliza el TTL (time to live). El TTL nos indica cuánto tiempo estuvo el archivo en el CDN, configurar correctamente el TTL es un punto clave para el buen funcionamiento de CDN.

Consideraciones en el uso del CDN

- Costo: el CDN es mantenido por terceros y esto resulta costoso generalmente. Por lo tanto hay que ser muy cuidadoso con que archivos son almacenados.
- Política de expiración: el seteo correcto del TTL como se explicó anteriormente es clave para el correcto funcionamiento del CDN ya que un tiempo bastante corto invalida su uso y un tiempo largo, el contenido de las respuestas será desactualizado.
- Caída del CDN: el CDN es un sistema complejo que puede tener sus fallas. Por lo tanto el sistema debe ser capaz de detectar que si es que hay una falla para ir a buscar los archivos al lugar de origen.

[1] https://learn.microsoft.com/en-us/azure/architecture/best-practices/cdn



A la arquitectura con la que veníamos trabajando, se le agregó:

1. El uso de un CDN para almacenar todo el contenido estático (JC, CSS, imágenes, etc.). De esta forma el cliente lo busca en el CDN y no en el web server para una mejor performance.
2. Por otro lado, también se redujo la carga a la base de datos agregando una caché.

Cola de mensajes

Una cola de mensajes es un componente de software que admite la comunicación de forma asincrónica entre dos componentes de Software. La arquitectura básica de una cola de mensajes consiste en un productor/publicador que crea mensajes y los publica en la cola. Luego, otros servicios o servidores llamados consumidores/suscriptores se suscriben a la cola y comienzan a recibir mensajes de la misma que básicamente se trata de procesos a ejecutar. Es importante aclarar que los mensajes se procesan una sola vez por un único consumidor.

Las colas de mensajes se pueden usar para desacoplar procesos pesados y diseñar arquitecturas más escalables. Con la cola de mensajes, el productor puede enviar mensajes a la cola cuando el consumidor no está disponible para procesarlos (ya que el mensaje persiste en la cola hasta que un consumidor lo tome para procesarlo) y el consumidor puede leer mensajes cuando el productor está ocupado. Como consecuencia, esto significa que tanto el productor como el consumidor pueden escalar de forma independiente.

Logging, métricas y automatización

Cuando trabajamos con pequeños sistemas que poseen una baja cantidad de usuarios y recursos, el logging, métricas y automatización suele ser una buena práctica pero no algo necesario. Sin embargo, cuando trabajamos en sistemas de mayor envergadura esto resulta esencial.

- *Logging*: tener un registro de los logs del Sistema siempre es útil cuando queremos identificar un error o mejorar algún componente del sistema. Existen herramientas que nos proveen algunas funcionalidades interesantes para trabajar con los logs ya que en sistemas grandes suelen ser millones.
- *Métricas*: las métricas son útiles para comprender la salud del sistema y nos da una idea si es se están utilizando de forma eficiente los recursos disponibles o si esto puede ser el causante de un problema. Algunas métricas que son interesantes revisar son:
    - Métricas a nivel de host: CPU, memoria, E/S de disco, etc.
    - Métricas a nivel agregado: el rendimiento de la capa de la base de datos, la capa de caché, etc.
    - Métricas clave del negocio: usuarios activos diarios, ingresos, etc.
- *Automatización:* cuando un sistema se vuelve grande y complejo, es necesario hacer uso de herramientas de automatización. Un ejemplo podría ser automatizar el proceso de compilación, testing, despliegue, etc. Esto mejora significativamente la productividad de los desarrolladores y contribuye a la implementación eficiente de prácticas de CI/CD (Integración Continua/Entrega Continua). Con CI/CD, las actualizaciones y cambios en el código se integran de manera continua, se someten a pruebas automáticas y se despliegan de manera automática o semiautomática en entornos de producción, lo que agiliza el ciclo de desarrollo y garantiza una entrega de software más rápida y confiable.

CASO PRACTICO: YouTube

En el siguiente caso vamos a aplicar los conocimientos vistos para pensar una posible implementación de la arquitectura de YouTube. Consiste en un sitio donde los creadores de contenido pueden subir videos y el resto de los usuarios pueden visualizarlos. Si bien parece ser algo simple, se complejiza cuando observamos las estadísticas de YouTube en los últimos años:

- Usuarios activos por mes: 2.000 millones
- Vídeos visualizados por día: 5 billones
- La plataforma se encuentra disponible en más de 100 países, en 80 lenguajes distintos
- Más del 70% de los usuarios ingresan desde dispositivos móviles con una sesión promedio de 40 minutos

Alcance de la arquitectura

- Se debe poder subir videos a la plataforma de forma rápida.
- Se deben poder mirar videos en línea de forma fluida (teniendo la posibilidad de cambiar la calidad).
- Los usuarios pueden comentar y compartir videos, además de crear sus propias playlists.
- Los usuarios se pueden conectar desde la aplicación para teléfono, un navegador o un smart TV
- Para simplificar el sistema vamos a suponer que hay aproximadamente 5 millones de usuarios que utilizan la plataforma de forma diaria y que lo hacen por 30 minutos
- Se debe poder soportar usuarios de todo el mundo
- El tamaño máximo soportado para los videos es de 1GB, sin embargo el tamaño promedio es de 300MB
- Se espera que la arquitectura propuesta priorice la disponibilidad, escalabilidad y seguridad

Estimación

- Se calcula que hay **5 millones** de usuarios activos por día (Daily Active Users)
- Los usuarios ven **5 videos** por día
- El **10%** de los usuarios sube un video por día
- Teniendo en cuenta que el tamaño promedio de vídeo es de **300MB**
- La cantidad minima de espacio de almacenamiento necesaria es
  5 millones * 10% * 300MB = 150TB[2]
- Para el diseño de la arquitectura va a ser necesario un CDN, teniendo en cuenta que el costo del mismo es alto vamos a hacer algunas estimaciones al respecto:
  - Si tomamos como referencia el CDN de Amazon, vemos que el costo promedio por GB es de **$0,02** (el costo se calcula a partir del tráfico saliente)

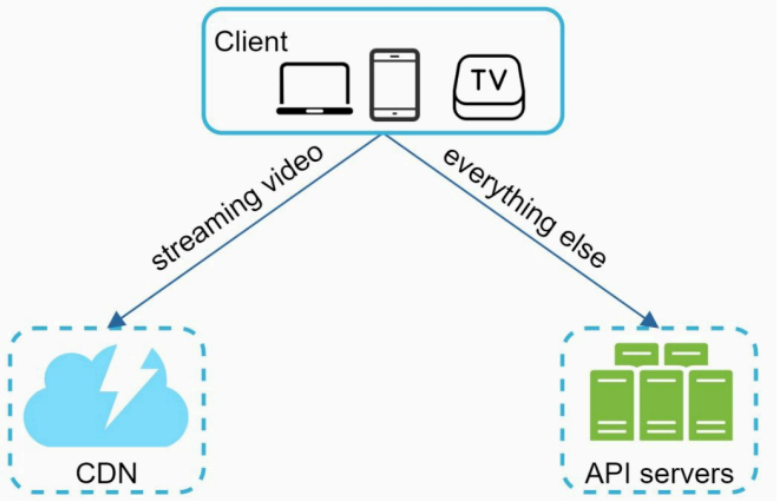
[2] No se tiene en cuenta el almacenamiento del CDN, posibles réplicas, raids, distintas calidades de vídeo, etc.

- El costo diario de utilizar un CDN:
  5 millones * 5 videos * 0,3GB * $0,02 = $1.500.000
- Observamos que el costo del CDN es muy alto, es por eso que luego discutiremos formas de adaptar nuestra arquitectura para reducir este costo

Propuesta de arquitectura a alto nivel

A un alto nivel nuestra arquitectura tiene tres grandes componentes:

- El cliente que puede ser una PC, smartphone o smartTV
- El CDN de donde se reproducen todos los videos
- Los API servers que son para el resto de las acciones como: metadata de videos, registro de usuarios, comentarios, etc.

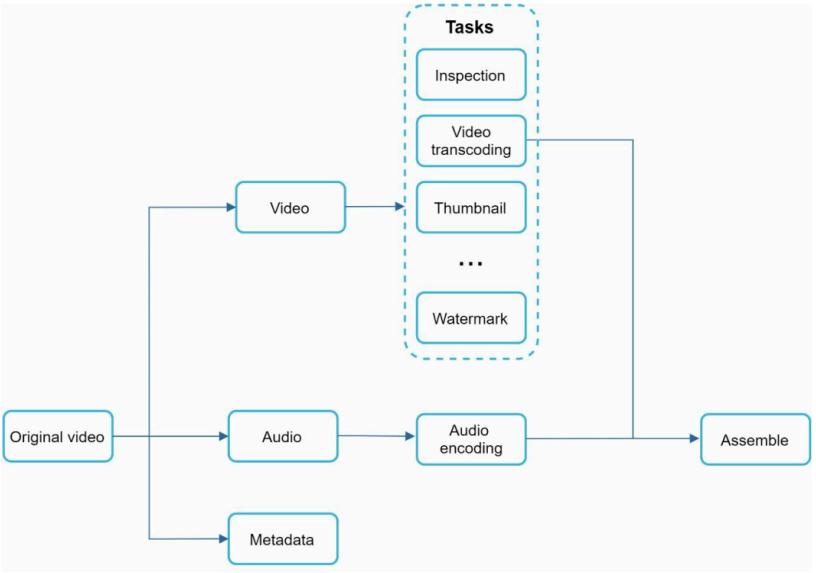


Codificación de videos

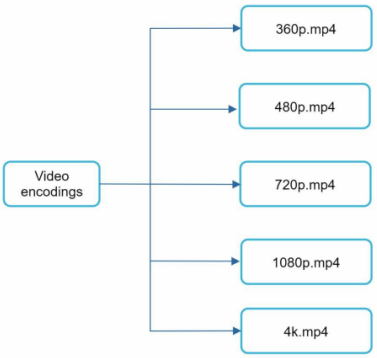
Cuando se graba un video (con un celular o cámara generalmente) obtenemos el mismo en un formato específico, que resulta un problema cuando se quiere reproducir el video en millones de dispositivos ya que lo más probable es que el bitrate y el formato no sean compatibles con otros dispositivos. Es por eso que el video debe ser codificado siempre, lo que trae como consecuencia las siguientes ventajas:

- El espacio de almacenamiento que consume el video es menor
- Muchos dispositivos y navegadores solo admiten ciertos tipos de formatos de video. Por lo tanto, es importante codificar un video en diferentes formatos por razones de compatibilidad.
- Para garantizar que los usuarios vean videos de alta calidad manteniendo una reproducción fluida es una buena opción disponibilizar los videos en varias resoluciones
- Como las condiciones de la red pueden cambiar mientras se reproduce el video, es esencial cambiar automáticamente la resolución del mismo de acuerdo al estado de la red.

La codificación de videos es un proceso computacionalmente caro y que consume un tiempo considerable. A esto se suma que los videos son subidos en diferentes formatos de audio y video. Es por eso que para que el proceso utilice los recursos de forma eficiente se realizan los procesamientos de forma paralela y para llevar esta idea a un siguiente escalón se separa al video original y se lo divide en: audio, video y metadata como se puede ver en la imagen.

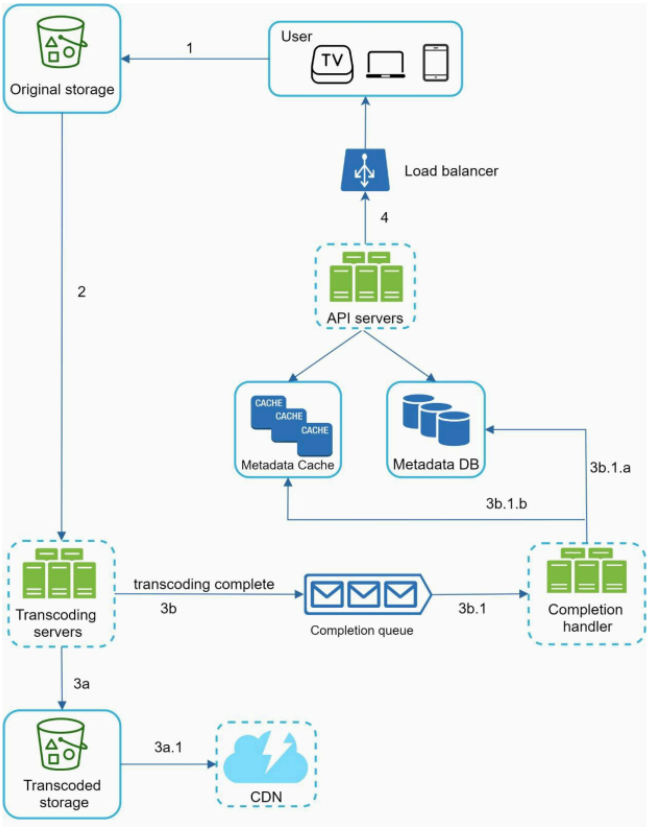


De esta forma algunas de las tareas que podemos realizar en paralelo son:



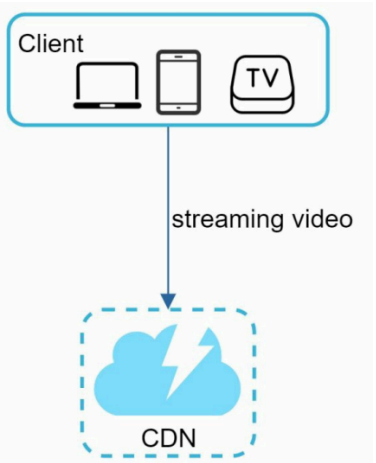
- Video:
    - Inspección: este proceso nos asegura que el video tenga una buena calidad, que no esté corrupto y que respete la política de YouTube.
    - Codificación: los vídeos son convertidos para soportar distintas resoluciones, codecs, bitrates, etc. Como se puede ver en la imagen.
    - Miniatura del video: puede ser subida por el usuario o generada automáticamente por un proceso que se dedique a esto.
    - Marca de agua: se le agrega una marca de agua al video que contiene la identificación del mismo.
- Audio
- Metadata: mientras el audio y el video se encuentran procesando, el cliente en paralelo envía una request al servidor para almacenar la metadata en la base de datos correspondiente.

Flujo de Subida de videos



1. Los videos son subidos en su tamaño y formato original
2. El funcionamiento es similar a una cola, ya que los servidores de conversión toman los videos y los convierten a un formato óptimo para el almacenamiento del mismo y los eliminan del almacenamiento original.
3. Una vez completa la conversión, los siguientes pasos ocurren en simultáneo:
    a. Los videos en el tamaño y formato deseado se envían al almacenamiento correspondiente, donde se encuentran todos los videos del sistema
        i. Los videos son enviados al CDN, de acuerdo a la política que se haya tomado.
    b. Se encola un evento de finalización en la cola de finalización
        i. Los consumidores de la cola toman los procesos
4. Los API servers le informan al cliente que el video fue subido y ya se encuentra disponible para streaming

Flujo de streaming



Cuando miramos un video por YouTube (y en la mayoría de las plataformas de streaming), la reproducción del video es inmediata. No es necesario esperar a que se descargue todo el video para comenzar la reproducción. De hecho esa es la diferencia entre descarga y streaming: la descarga significa copiar el video en su totalidad al almacenamiento local y luego se reproduce, mientras streaming hace referencia a que se recibe el video de a porciones, se almacenan en un buffer y mientras se va reproduciendo

La arquitectura propuesta para el streaming de videos resulta bastante más sencilla que el flujo de subida de videos. Simplemente el CDN más próximo al dispositivo es el encargado de enviar el video ya que esto reduce la latencia.

En el flujo descrito no se tienen en cuenta los videos en vivo, ya que hay muchas cosas que funcionan distinto en este caso.