Escalabilidad y virtualización

Ingeniería Web, 2019

La escala y sus problemas

Al crecer vamos a encontrarnos con problemas.

Si crecemos **mucho**, van a ser **muchos** problemas.

Y no todos los problemas son técnicos!

Problemas no técnicos

Cómo hacemos para que 1.000.000 personas pueda interactuar en un mismo lugar sin que sea un caos?

No hay una fórmula. Pero podemos discutir algunas ideas útiles.

Ruido y filtros

No podemos esperar que el usuario vea tooodo junto y filtre con su mente. Se cansaría muy rápido y dejaría la comunidad.

- Filtrar por **contenido**: Subforos, categorías por temas, secciones, o por tópicos que sabemos que le interesan al usuario (ej: reddit).
- Filtrar por **relaciones** entre usuarios: mostrar lo de las personas que el usuario sigue, amigos, grupos a los que se une (ej: facebook).
- Filtrar por **geolocalización**: mostrar contenido relevante en su región (ej: twitter trending topics).

Anonimato vs identificación

El **anonimato** facilita el abuso, trolling, spam, etc. Con muchos usuarios, esto crece mucho.

Identificar a los usuarios mitiga el problema.

- Requerir un nombre o identificación real (ej: Airbnb pidiendo DNI). Pero en pocos casos es justificable.
- Asociarlo con su identidad digital: su email, sus contribuciones y perfil dentro del sitio, su cuenta de redes sociales, reputación, etc (ej: Stack Overflow).
- ...

Velocidad vs calidad

La velocidad y facilidad de respuesta fomenta las peleas en caliente, las respuestas poco medidas, etc.

Podemos intentar mitigar esto artificialmente:

- Limitar **cadencia** de respuestas.
- **Detectar** lenguaje ofensivo, esconder y más tarde pedir **confirmación**.
- Mostrar respuestas **solo a pocos usuarios** primero. Si la reacción es mala: rechazar el post. Si es buena: mostrarlo a todos.
- ...

Usuarios abusivos

Hoy el bullying, acoso y otras actitudes dañinas destruyen comunidades, hacen perder usuarios, etc.

Se puede tratar de controlar:

- **Normas claras** respecto a comportamiento no tolerado.
- Facilidad de **denuncia**, y buenas herramientas para **moderar** denuncias.
- **Consecuencias consistentes** para los usuarios abusivos (bans sin importar fama, etc).
- Facilidades para que los usuarios puedan **ignorar, bloquear, etc** (ej: mutear respuestas de un tweet, bloquear a listas conocidas de malos usuarios, etc).
- ...

Es un problema grande y aún no resuelto.

Problemas técnicos

Cómo hacemos para que 1.000.000 personas pueda acceder a mi sitio diariamente y no me explote el servidor?

Es un problema bastante más resuelto!

Las partes del problema

- Capa de **encriptación** (si usamos SSL)
- Servicio **HTTP** (servidor web que recibe las conexiones, encola, responde y loguea las requests, etc)
- Capa de **aplicación** (el código de nuestro sitio web)
- Capa de **persistencia** (el motor de base de datos, y almacenamiento de blobs/archivos)

Una máquina vs muchas máquinas

En un sitio pequeño: una máquina ejecuta todas las capas.

Qué pasa si se empieza a quedar chica?

Compramos una máquina más grande? O separamos las capas en varias máquinas?

Una máquina vs muchas máquinas

Dato super importante:

Una máquina grande, siempre es más cara que N máquinas chicas que sumen la misma potencia.

Una máquina vs muchas máquinas

Vamos por la opción de separar las capas del sitio en varias máquinas…

Es más estable?

Una máquina vs muchas máquinas

No!

Si una máquina tiene 95% de uptime, y separamos las capas en 3 máquinas, necesitamos **las 3 al mismo tiempo** para que el sitio funcione...

95% * 95% * 95% = nuestro sitio tendría entonces **85% de uptime**!

Una máquina vs muchas máquinas

Es muy diferente distribuir **partes del sistema en distintas máquinas**, que tener **varias máquinas "iguales" que se repartan la carga de una misma parte del sistema**.

- Lo primero mejora performance pero reduce estabilidad.
- Lo segundo mejora performance y puede mejorar estabilidad!

Al escalar vamos a tener que hacer ambas cosas: va a haber máquinas diferentes para cada capa, pero cada capa puede tener N máquinas iguales repartiéndose la carga.

Capa de persistencia

Si queremos N máquinas en esta capa:

- Muchas máquinas con la misma DB, replicada? complicación grande: sincronización. Sobrecarga la red, es costosa, compleja de mantener.
- Distintas DBs? Complicaciones grandes: hacer consultas que combinan datos de varias DBs, capa intermedia que decide a qué DB ir según el dato que buscamos, etc.

Pero recordemos: una máquina grande va a ser más caro que N máquinas chicas. Acá si podemos, conviene gastar.

Capa de persistencia

Componente de HW más importante?

Capa de persistencia

La RAM!

No el disco. Una DB bien configurada, evita ir a disco todo lo que pueda. Más RAM ⇒ menos lecturas a disco.

Segundo componente de HW importante: CPU, para resolver consultas complejas (joins, etc).

Capa de aplicación

Esta capa recibe requests, y arma y devuelve responses ejecutando nuestro código.

Hay algo que nos complique tener muchas máquinas iguales ejecutando el sitio web y respondiendo requests?

Capa de aplicación

Depende: dónde se guardan las sesiones?

- En RAM? Entonces sí complica: si una máquina tiene la sesión de fisa, y otra máquina recibe una request de fisa?
- En DB? Entonces cualquier máquina puede responder requests de cualquiera. Peeero… demasiada carga de trabajo para la DB.
- Ideal: server de sesión aparte (Redis, Memcached, etc).

Capa de aplicación

Si hicimos bien, entonces podemos tener N máquinas ejecutando el sitio independientes, sin comunicación entre ellas. Cualquiera puede responder requests.

Re fácil de escalar!! Con un “molde”, replicamos máquinas y listo.

Hay que “declararlas” en alguna parte, para que la capa de arriba les reparta las requests equitativamente. Un **balanceador de carga**.

Capa de aplicación

Componente de HW más importante?

Capa de aplicación

CPU.

No deberían necesitar mucha RAM: no tienen data persistente, y cada request debería requerir poca RAM para responderse.

No deberían necesitar buenos discos: no tienen la DB.

La CPU va a determinar cuántas requests por segundo van a poder responder.

Servicio de HTTP

En techs un poco más viejas, esto está “unido” con la capa de aplicación: el server HTTP ejecuta también el código del sitio (ej: Apache+PHP, IIS+.net).

Hoy se tiende a separarlos, es super importante poder hacerlo (ej: nginx + cualquier lenguaje).

Servicio de HTTP

Si están separados, esta capa está expuesta al mundo y **se pone en el medio** entre el cliente y la capa de aplicación. La escondemos detrás.

- Recibe y encola las requests.
- Pide a la capa de aplicación que construya las responses para esas requests.
- Balancea la carga repartiendo las requests equitativamente entre las máquinas de la capa de aplicación.
- Responde a los clientes con las responses armadas por la otra capa.

Servicio de HTTP

Normalmente es **mucho** más rápida que nuestra aplicación, y más liviana. Puede estar encolando y respondiendo millones de requests a la vez.

El factor de HW más limitante va a ser **latencia de red**!

Y en segundo plano, la CPU y RAM van a determinar cuánto podemos responder por segundo y cuánto podemos tener encolado a la vez.

Disco tiene cero impacto, no tenemos nada persistente (logs?).

Capa de encriptación

Lo puede hacer el mismo **servicio de HTTP**, y eso es lo usual.

Es 100% CPU.

Si hace falta, se puede separar en máquinas aparte.

Incluso hay hardware específico para esto, “cajitas de SSL” que ponemos entre el servicio HTTP y el exterior.

Cuántas máquinas necesitamos?

Difícil de calcular (depende del caso), pero algunas cosas se pueden estimar.

Aproximadamente, un core responde **10 requests dinámicas por segundo**.

Los micros no fueron mejorando?

Sí, y los sitios se fueron complejizando.

Hay que hacer **benchmarks**!

Balanceo de carga

Si una capa tiene varias máquinas, cómo les repartimos el trabajo? (requests a responder, o queries a ejecutar, etc).

Objetivos:

- Distribuir carga a las máquinas más libres.
- Si una máquina falla, que no sea catástrofe.
- Que sea transparente para el usuario (bookmarks, etc).

Hay dos balanceos que nos interesan: de queries, y de requests web.

Balanceo de carga

El balanceo de **queries** se hace en la capa de persistencia:

- Si es una máquina, se va a balancear en el manejador de procesos de la máquina (cada query va a ser un proceso).
- Si tenemos muchas máquinas de db, alguien que mande las conexiones a cada máquina.

Frente a errores:

- Si falla la única máquina? Tenerla duplicada/replicada!
- Si falla una de las N máquinas?
    - Todas tenían la misma DB? Tener una extra preparada igual.
    - Tenían partes de la DB? Cada parte duplicada/replicada!

Balanceo de carga

El balanceo de **requests** se hace en el servicio de HTTP.

Tenemos dos opciones: por DNS, o con un balanceador de carga.

Balanceo de carga

El balanceo de **requests** por DNS: cada vez que un usuario pide entrar a “misitio.com”, resolvemos a una IP diferente (ciclando por un grupo de ips). Cada ip es un server.

Feo. Problemas!

- Los clientes cachean la ip resuelta. Si un cliente hace 10.000 requests, todo ese tráfico llega al mismo server, no se reparte. Ejs: una empresa con un router, o todos los clientes de fibertel!
- Si se muere un server, los clientes con ip cacheada siguen intentando hablarle a ese server. Ej: todos los clientes de fibertel!
- No podemos tener lógica de balanceo “custom” (ej: por región geográfica, etc).

Balanceo de carga

El balanceo de **requests** por balanceador de carga interno: los clientes hablan siempre al mismo server, el server “le pide” la response a los servers de aplicación, rotando a quién le pide en cada request.

Mucho mejor!

- Se resuelven los problemas de la solución anterior.
- Muy resistente a fallas de un server!
- Completamente transparente al usuario, y no nos pueden sobrecargar un server en particular.
- Nos permite tener lógica de balanceo custom.
- Los web servers lo saben hacer muy bien (nginx, etc).

Balanceo de carga

Y qué pasa si nos falla el balanceador de carga???

Tener otro siempre preparado.