

# Pauta Control 1

## Redes

**Plazo de entrega: 27 de septiembre 2022**

*José M. Piquer*

### 1 P1: SOCKETS

Cuando escribimos un servidor que recibe peticiones por sockets (tanto UDP como TCP) vimos que podemos atender a cada cliente con procesos pesados, procesos livianos (threads) o incluso con select().

A Ud acaba de contratarlo una empresa que desarrolla un nuevo servidor web, que espera ser muy estable y manejar muchos clientes sin problemas. Le piden diseñar la mejor forma de atender los sockets de los clientes en una forma que ayude a la estabilidad y escalabilidad del servicio. ¿Qué esquema recomendaría? Justifique y analice los escenarios posibles.

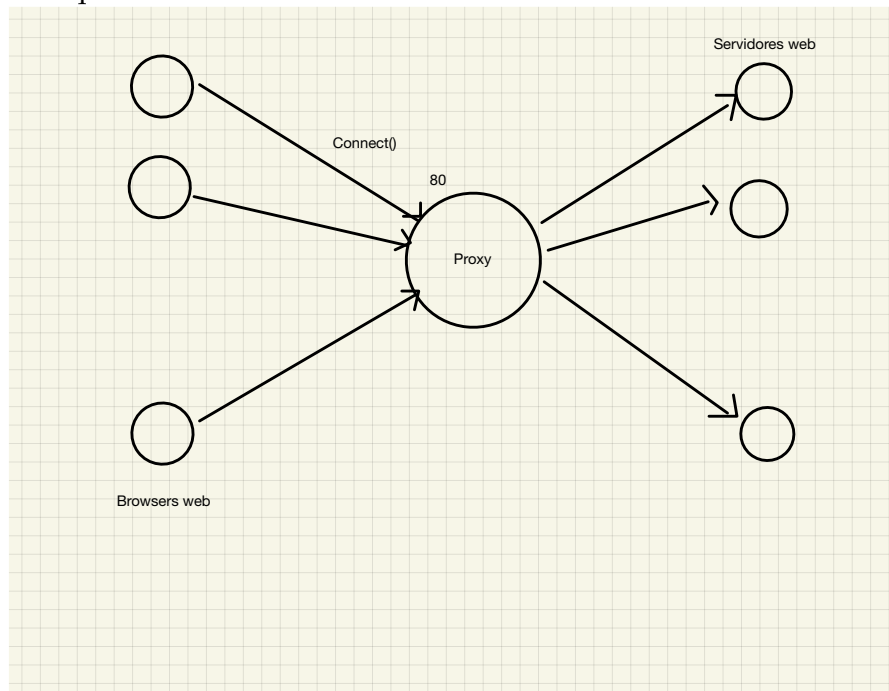
*La forma más estable y segura de manejar múltiples conexiones TCP (sockets HTTP o HTTPS) es siempre usando procesos pesados, uno que atienda cada cliente en forma independiente. En un servidor grande, deberíamos poder tener miles de servidores web corriendo en paralelo sin problemas, y el que cada uno tenga un socket y un cliente, nos permite asegurar que un error matará sólo a ese servidor y a ese cliente y el resto del sistema seguirá disponibles sin problemas. Todos los otros sistemas (threads, select) aumentan el riesgo de matar a todo el servicio al mismo tiempo, y son mucho más frágiles (aunque más eficientes también). La solución de fork() con procesos pesados es un poco menos eficiente y lenta, pero mucho más segura y estable.*

### 2 P2: APLICACIONES

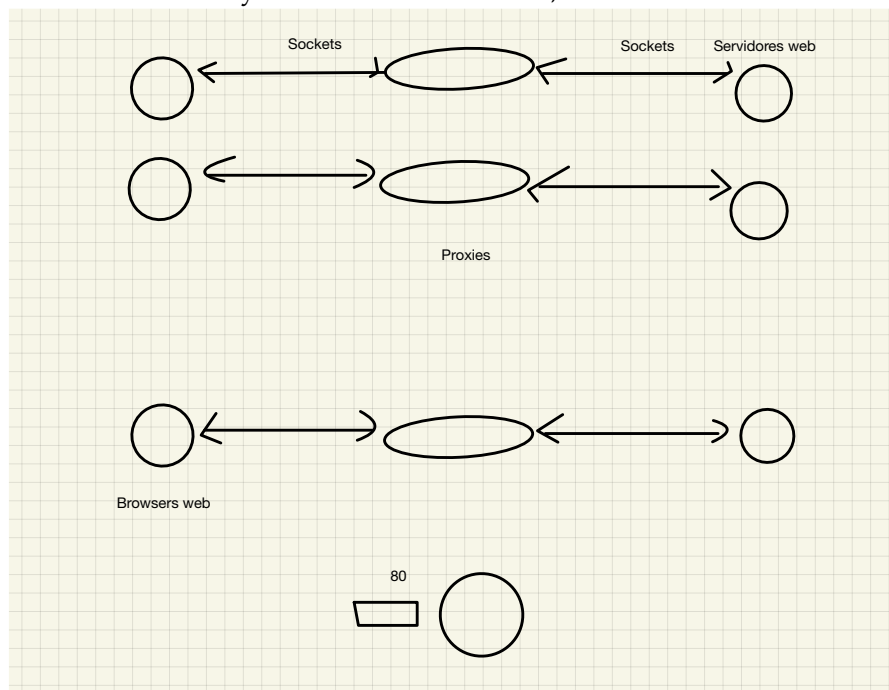
En las actividades propuestas del curso, vimos cómo programar un proxy, que conectaba un cliente con un servidor poniéndose él al medio. A una ingeniera de redes se le ocurrió que podríamos usar esta misma idea para balancear carga entre servidores. En la red interna de la corporación podemos instalar unos 20 servidores idénticos e independientes corriendo en 20 computadores grandes, incluso en redes distintas. Luego, instalamos el proxy como único servidor público hacia afuera, y él va conectando los clientes a medida que llegan, a los servidores reales, uno por uno. Cada cliente conectado al proxy se atiende con un proceso "pesado" (usando `fork()` en Unix)

independiente y, si ya tenemos más de 20 clientes conectados, vamos re-utilizando servidores.

El esquema de conexión sería:



Y con los clientes y servidores conectados, sería:



¿Es buena o mala idea? ¿Es factible de implementar? ¿Cree Ud que ayuda a balancear carga realmente? ¿Qué ventajas y desventajas le ve? ¿Hay algún tipo de servidor web que no se pueda instalar así?

*Es buena idea en general. Si tengo muchos computadores que me permiten correr servidores web idénticos en todos ellos, con los mismos archivos y los*

*misimos datos, esta es una buena forma de distribuir la carga entre todos ellos. Es factible de implementar, aunque requiero de un proxy que quede corriendo a lo largo de toda la conexión (además del servidor web correspondiente). Si el proxy tiene la información de los servidores que están ocupados y los que no, se puede hacer un buen balanceador de carga. Si no, es difícil, por que podemos estar sobrecargando unos sobre otros, sobre todo si no sabemos cuánto tiempo va a durar cada conexión. La principal ventaja es que es una forma simple de balancear carga entre varios servidores. La desventaja es la dependencia del proxy que tiene que mantenerse funcionando a lo largo de toda la conexión. Si el servicio web es básicamente acceso a una base de datos central, o a un ambiente compartido (un juego de video compartido, por ejemplo), este balanceo no ayudará mucho, ya que el cuello de botella estará en el espacio a compartir.*

### 3 P3: DNS

Hay dos formas de darle redundancia al DNS: aumentar el número de servidores del dominio (así cualquiera de ellos que responda, está bien) y usar *anycast*, que parece un sólo servidor de nombres pero que se replica en varias máquinas distribuidas por el mundo y la red hace que responda siempre el más cercano geográficamente al cliente (veremos cómo se implementa más adelante en el curso, su implementación concreta no es relevante para esta pregunta).

En general, dominios importantes, como .CL, usan una mezcla de ambas soluciones: varios servidores y algunos de ellos son *anycast*. ¿Por qué no usar simplemente un servidor *anycast* replicado en decenas de lugares del mundo? ¿Qué gano con tener más servidores en la lista? Y, si es bueno tener muchos servidores, ¿por qué mejor no tener cientos de servidores en la lista?

Es importante ver que la pregunta no es sobre implementación de *anycast*. Suponga que funciona perfecto: con un nombre de servidor, siempre me conecta con el servidor más cercano a mí.

*El principal riesgo de usar un sólo anycast como nombre de servidor es que tengo un sólo servidor del que dependo. Es decir, si ese servidor (que está cerca mío geográficamente) puede estar sobrecargado, o incluso dejar de responder en su socket DNS, y yo seguiré preguntándole a él, ya que no hay más alternativas. Al tener más servidores en la lista, puedo seguir intentando con otros servidores que tal vez estén más lejos, pero que puede ser que sí me respondan. El tener demasiados servidores en la lista tampoco es tan buena idea, por que me demoraré mucho en descubrir que nadie responde y tendré que estar manejando la lista de servidores siempre para ir decidiendo a cuál me conviene preguntar ahora. Y, quiero que las respuestas con las listas de servidores no sean demasiado largas, o se vuelven ineficientes.*