Control 1
Redes
Plazo de entrega: 13 de abril 2022
Pauta de Corrección

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 Netflix en el año 2010 (cuando estaba comenzando), y tienen un problema grave: el proceso servidor principal de streaming de películas, que usa UDP, falla bastante seguido. El principal problema es que se les cae el programa servidor (escrito en Python) y eso mata todo el servicio de streaming hasta que alguien vuelve a levantar el proceso que atiende los sockets. No sabemos nada más, le encargan averiguar lo que está pasando y levantar soluciones de emergencia lo antes posible. Imaginen el caos que es para una empresa emergente, que está empezando a tener éxito, que se corten todas las películas que la gente está viendo en ese momento y tener que esperar que el servidor vuelva a reiniciar.

Explique qué haría, qué podría estar pasando y qué cambios de emergencia uno podría hacer en el código del servidor para ayudar. ¿Por qué un servidor UDP que atiende múltiples clientes simultáneos puede caerse por completo y no atender ningún cliente más? Concéntrense en el problema de software y no en el de hardware (o sea, no vale decir que instalaremos 100 computadores más o cosas así). ¡Tampoco vale reclamar que nadie escribía servidores en Python el año 2010 o recomendar cambiar de lenguaje!

Si el servidor se cae completo, y mata todos los streamings en curso, probablemente es un servidor que está atendiendo a los clientes con threads o con select(). Si esto es así, la primera medida sería modificar el código del servidor para crear un proceso pesado por cliente. De esta forma, el error que viene de antes (que mataba todo) seguirá existiendo, pero sólo matará el stream de ese cliente en particular y dejará el resto funcionando bien. Esta solución es más ineficiente en performance, pero permite estabilizar el servidor

Otro camino sería tratar de distribuir los paquetes UDP que llegan hacia procesos pesados ad-hoc, y transformar el servidor principal en un distribuidor

de flujos: él recibe todos los paquetes desde los clientes y los re-envía al proceso correspondiente, puede ser vía memoria compartida, socket local, etc.

Lo importante es que cada cliente tenga un proceso propio que lo atienda. (2.0)

2 P2: APLICACIONES

Pensando en el protocolo de la tarea 1, analice si soporta o no archivos binarios: ¿puede haber un contenido del archivo que "engañe" al protocolo? (por ejemplo, una línea con sólo una 'E' que se confunda con fin de archivo). Argumente su respuesta.

El protocolo soporta sin problemas el envío de archivos binarios, ya que el primer byte indica que acá vienen "datos" ('D') y luego sabemos cuántos bytes vienen (del largo del paquete UDP). De la misma forma, un paquete que dice 'E' indica el fin del archivo y no puede confundirse con datos del archivo, ya que si fuera así, llegaría un paquete que diría 'DE', indicando que es un dato que dice 'E'. (2.0)

3 P3: DNS

Cada dominio de Internet tiene una lista de servidores de nombres. Todos esos servidores me responden con la misma autoridad sobre ese dominio, pero tengo que escoger a uno para preguntarle. Cuando yo soy un *resolver*, me interesa minimizar los tiempos de respuesta y quisiera favorecer al que me responde más rápido. Entonces, puedo recordar esas listas por un tiempo y ordenarlas por tiempo de respuesta, así voy usando el más rápido siempre.

Pero, esto es dinámico, un servidor que era rápido hace unas horas puede estar sobrecargado ahora. Entonces, quiero ir probando con los otros de la lista de vez en cuando, para darles oportunidad de mejorar sus tiempos.

Proponga un algoritmo para un *resolver* que comienza cuando recibo la lista de servidores para un dominio por primera vez (no he oído hablar de ninguno de esos servidores, no tengo ninguna información al respecto) y luego va aprendiendo en el tiempo, mientras sigue recibiendo preguntas sobre ese dominio que debemos enviar a uno de esos servidores. Digamos que podemos usar un timeout de 3 segundos, que indica que un servidor no está respondiendo.

Trate de ser óptimo, en el sentido de no dejar esperando a nuestro cliente innecesariamente.

Puede escribir el algoritmo en un pseudo-código o describir su comportamiento.

La propuesta es:

- *Recibo: (dominio, lista_servidores)*
- *elijo s1, servidor al azar*

- *envío query a s1*
- *anoto t1, tiempo de respuesta para s1*
- *si da timeout, busco otro y al loop*
- *nuevas preguntas para dominio:*
- *elijo servidor con mejor tiempo de respuesta para enviar query*
- *cada cierto tiempo, elijo al azar otro servidor de la lista y envío query en paralelo al mejor (para evitar esperas inútiles)*