Control 1
Redes
Plazo de entrega: 10 de abril de 2024

José M. Piquer

Intro

El control se responde en forma individual, sin preguntas entre Uds o a terceros. Todas las dudas pregúntenlas en el foro de U-cursos y el equipo docente les responderá. Entreguen las tres preguntas en un PDF con su nombre por U-cursos. Pueden (y deben) buscar material en Internet para responder, pero citen siempre las fuentes que utilizan. Pueden usar chatGPT para ayudarlos, pero, no pueden entregar su respuesta directamente, deben explicar y analizar todo lo que les propone. Recuerden que chatGPT se equivoca, alucina y miente con mucha convicción. Siempre recuerden mencionarlo si lo utilizan.

En épocas de chatGPT estamos obligados a evaluar mal respuestas que antes habríamos aceptado como casi correctas. Si Uds entregan un párrafo de argumentos correctos pero no responden nunca la pregunta específica con argumentos técnicos que corresponden a ese caso, hoy en día tienen cero puntos.

Un ejemplo, decir algo como:"existen algoritmos propuestos con mejoras para TCP para enlaces de alto delay como CUBIC y BBR, que buscan resolver este problema" antes les habrían dado algún puntaje (significaba que habían hecho una búsqueda en google al menos). Hoy en día vale cero.

Las preguntas de los controles las chequeamos con chatGPT antes de ponerlas, y la idea es que chatGPT no logre sacarse un 4.0.

Entonces, la recomendación es: contesten la pregunta específica del enunciad. Las frases periféricas que buscan adornar sin ir al detalle técnico no aportan en nada, de hecho más bien perjudican, por que mientras más hablan sin ir al punto, más generan la impresión en el corrector que no conocen la respuesta (y suenan igual que chatGPT).

P1: sockets

Para la Tarea 1, propusimos usar el servidor de eco que viene en el código de ejemplos: `server_echo_udp2.py` para que probaran localmente.

Sin embargo, en esos mismos ejemplo viene un servidor mucho más simple: `server_echo_udp.py`

Analice las diferencias entre ambos y pruebe si la segunda versión también serviría para la Tarea 1 (en ambos casos, hay que modificar el tamaño máximo del mensaje que se recibe desde el socket).

Responda las siguientes preguntas:

1. ¿Cuál de las dos es mejor solución para eco?

   El servidor simple es mejor: mucho más eficiente y liviano, y eco no necesita ningún tipo de estado que almacenar para un cliente específico, podemos responder cada paquete que llega inmediatamente.

2. En un servidor general (no de eco), ¿Cuándo debiéramos usar un estilo o el otro de solución?

   El servidor con threads es una buena solución cuando el diálogo con el cliente es largo y tiene etapas (estado en el servidor), por ejemplo si es un protocolo con autentificación.

3. Si en el caso de la Tarea 1 el servidor debe atender muchos clientes simultáneos enviando archivos muy grandes, ¿cambia su selección de mejor solución? ¿Por qué?

   No, sigue siendo mejor el servidor simple. Al haber muchos clientes con archivos grandes, simplemente aumenta el tráfico para el servidor. Por lo tanto la solución simple es aun mejor en este caso.

4. El cliente de eco que Uds tienen que modificar para la Tarea 1 envía un mensaje inicial y espera la respuesta (para descartarlo) antes de entrar al ciclo principal. ¿Por qué hace eso? ¿Sería necesario para ambos tipos de servidores?

   Como el servidor complicado crea un thread que se hace cargo del socket asociado a ese cliente, hay un momento entre el primer paquete y el resto en que el servidor no está preparado aun para recibir esos paquetes. En el caso del servidor simple no es necesario, ya que ese está siempre preparado para recibir paquetes de todos los clientes.

5. Si ahora uso un socket TCP (SOCK_STREAM) en vez de UDP, ¿qué pasa con las dos versiones de servidores discutidas aquí?

En TCP el servidor simple no tiene sentido. Siempre debo hace un `accept()` *en el socket de conexión para atender un nuevo cliente, por lo que siempre tendré un socket por cliente. El servidor complicado es muy parecido, pero el simple no sirve.*

P2: NTP

En NTP, existe una versión del servidor donde simplemente hace *broadcast* hacia la red, enviando un paquete UDP con la hora actual a todos los dispositivos que están conectados en la red local en ese momento y están escuchando en el puerto NTP.

Esta versión no es la favorita hoy en día, por que no permite mantener mucha precisión en el ajuste de los relojes.

Responda las siguientes preguntas:

1. ¿Qué limita la precisión en este caso?

   El problema de base es que el mensaje sale con la hora de origen, pero toma un tiempo dentro de la red antes de llegar a su destingo. Por lo tanto, siempre su hora está un poco atrasada. No puedo ser más preciso que el retardo entre el origen y el destino.

2. ¿A qué precisión puedo aspirar?

   La mejor precisión posible será el mínimo retardo de red que tengo con el emisor.

3. Proponga ideas para mejorar esa limitación

   Cualquier estimación que el receptor pueda hacer del retardo con el emisor serviría para mejorar. Simplemente le sumo el retardo a la hora recibida. Podríamos mantener una comunicación paralela con el emisor y medir el tiempo de ir y volver y usar la mitar de eso como estimación del retardo. O incluso un archivo de configuración donde el administrador ponga un retardo estimado para sumarle a la hora recibida.

4. ¿Existen escenarios en que el esquema de *broadcast* igual sea mejor?

   Una red local con bajo retardo, donde hay muchos dispositivos muy simples (tipo IoT) puede ser un buen escenario, ya que no requiere ninguna configuración ni algoritmo en el receptor.

5. Si recibo una hora que es anterior a la mía (mi reloj está "adelantado"), ¿puedo retroceder el reloj?

   La idea es nunca retroceder el reloj del dispositivo. Muchos programas pueden verse afectados e incluso fallar en este escenario. La recomendación es hacer que el reloj avance más lento, de modo que recupere el tiempo sin nunca retroceder.

P3: Arquitectura de Servidores

Una empresa de sistemas de vigilancia quiere generar un servidor de video que permita mirar muchas cámaras dentro de una organización, desde Internet. Para esto, necesitan programar un servidor que recibe clientes desde Internet que miran el conjunto de las cámaras, y el mismo servidor es cliente de múltiples cámaras IP (ver diagrama).

Para esto, el servidor de video debe manejar muchos sockets conectados a las cámaras (digamos, más de 20) y aceptar conexiones de Internet de clientes que quieren ver esas cámaras en tiempo real.

El problema es que estas cámaras soportan un sólo cliente TCP a la vez y envían datos que sólo transmiten los cambios en el video (las zonas que se mueven), para ahorrar capacidad y ancho de banda.

Entonces, el servidor debe ser el único cliente que se conecta a las cámaras, y arma en memoria el video a transmitir para cada una. Pero, hacia Internet debe ser capaz de atender múltiples clientes. Mientras más clientes se puedan tener en Internet, mejor.

Tienen cuatro propuestas de implementación que les han traído sus ingenieros para atender a los clientes de Internet:

1. Usar procesos pesados: cada cliente es atendido por un servidor dedicado, con un socket TCP (SOCK_STREAM) para cada uno.

   Este esquema no me sirve, ya que las cámaras soportan un sólo cliente a la vez. No puedo tener ningún esquema que requiera que las cámaras tengan múltiples conexiones. Al hacer fork() del servidor, las cámaras tendrán nuevos clientes siempre. Por otro lado, al ser procesos pesados, tendrían que duplicar los videos en memoria, lo que es demasiado ineficiente para múltiples clientes.

2. Usar threads: cada cliente es atendido por un thread dedicado, con un socket TCP (SOCK_STREAM) para cada uno.

   Este es un buen esquema en este caso. Cada cámara tendrá un video en memoria en el servidor, y el thread tendrá que obtenerlo para armar

su video consolidado. Los threads deberán sincronizar el acceso a estos videos con threads que atienden las cámaras. Los problemas son los clásicos de threads: difícil de programar, si se cae un thread se caen todos juntos, etc.

3. Usar select: todos los clientes son atendidos por el mismo proceso que usa select() para definir a qué socket atender, cada cliente tiene un socket TCP (SOCK_STREAM) propio.

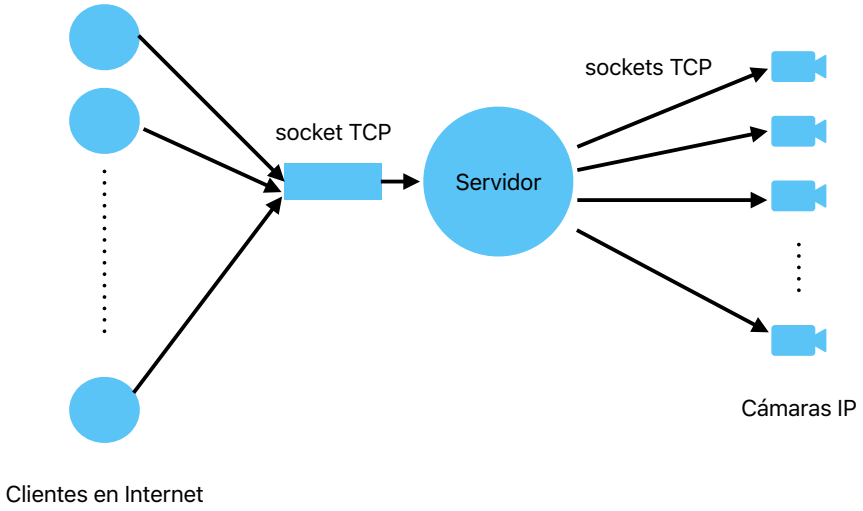
   Esto se puede hacer funcionar, pero parece demasiado complejo: lo que hay que hacer para cada cliente (obtener el video de cada cámara, consolidarlos en un sólo video común que enviar al cliente) parece difícil de escribir en bloques de código que estarán en las diversas partes del select. Por otro lado, hay limitaciones a la cantidad máxima de sockets para select() que puede llegar a ser un problema.

4. Socket común: todos los clientes comparten el mismo socket UDP (SOCK_DGRAM) para enviar sus requerimientos, el servidor es un proceso que va atendiendo los requerimientos uno por uno.

   Depende del protocolo: si cada requerimiento del cliente tiene una respuesta simple (por ejemplo, un frame que contiene todos los frames actuales de las cámaras), puede ser buena idea. De la misma forma, si normalmente estoy enviando una secuencia de paquetes UDP con el video a todos los clientes (sin necesitar respuesta de ellos) y el cliente solo envía paquetes para terminar o cosas esporádicas, puede ser muy buena solución. Hay que enfrentar el problema de las pérdidas, pero en stream de video normalmente esto no es demasiado difícil de soportar.

Analice las distintas alternativas y discuta sus pro y contras. Elija la alternativa que Ud usaría en un caso así y justifique su elección.

En este caso particular, creo que la alternativa 2 es la que yo usaría: es flexible para implementar un protocolo avanzado entre el cliente y el servidor, por ejemplo recodificando los videos. Si en realidad el protocolo es muy simple, y el flujo del video hacia el cliente es simplemente una secuencia de paquetes UDP que no requieren ninguna respuesta del cliente, la alternativa 4 es muy buena.


sockets TCP
socket TCP
Servidor
Cámaras IP
Clientes en Internet