Tarea 3: Cliente eco UDP con Stop-and-Wait para medir performance
Redes
Plazo de entrega: 9 de mayo 2025

José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente UDP con 2 threads que hicieron en la T2, para que ejecute un protocolo Stop-and-Wait para corregir los errores en la conexión UDP. El servidor que usaremos para la medición es el mismo de la T2: `server_udp3.py` (se provee como material docente, no está en el código estándar del curso).

Igual que en la T2, el cliente usa un archivo de entrada y otro de salida como archivos binarios (así pueden probar con cualquier tipo de archivo), y recibe como argumento el tamaño de las lecturas y escrituras que se harán (tanto de/desde el socket como de/desde los archivos).

Igual que en la T2, se debe terminar el envío con un paquete UDP vacío (cero bytes) que hace de EOF. Cuando el receptor detecta este paquete, debe terminar la ejecución.

Deben definir un timeout máximo de espera de 3 segundos en el socket para el receptor, con `settimeout()`. Cuando ocurre este timeout deben terminar con un error, pero esto no debiera ocurrir nunca si el protocolo está bien implementado. Este valor no es lo mismo que el timeout de retransmisión, que se recibirá de parámetro (y obviamente debe ser inferior a 3s).

Pueden medir el tiempo de ejecución y usar el tamaño del archivo de salida como la cantidad de bytes transmitidos.

El cliente que deben escribir recibe el tamaño de lectura/escritura, el timeout de retransmisión, el archivo de entrada, el de salida, el servidor y el puerto UDP.

```
./client_bw.py size timeout IN OUT host port
```

Para medir el tiempo de ejecución pueden usar el comando `time`.

El servidor para las pruebas pueden correrlo localmente en local (usar localhost o 127.0.0.1 como "host"). Dejaremos uno corriendo en anakena también, en el puerto 1818 UDP.

Para enviar/leer paquetes binarios del socket usen send() y recv() directamente, sin pasar por encode()/decode(). El arreglo de bytes que usan es un bytearray en el concepto de Python (al ser binarios, no son strings).

2. Protocolo Stop-and-Wait

Este caso tiene algo muy distinto y raro con respecto a las implementaciones clásicas: Uds están haciendo un protocolo que conversa entre el thread enviador y el thread receptor que corren en el *mismo cliente*. El servidor no participa del protocolo, simplemente hace eco de los paquetes que Uds le envían. El envío y recepción usarán la misma lógica habitual de Stop-and-Wait (número de secuencia, esperar hasta que se reciba, retransmitir en caso de timeout). Pero, no usaremos ACKs como mensajes, ya que no son necesarios si estamos en el mismo proceso: basta con que el receptor le informe al enviador que recibió bien un paquete.

La forma más sencilla en Python es usar `Condition` y una variable compartida para esto.

Se les pide implementar el protocolo según las especificaciones siguientes (se revisará la implementación para que lo cumpla):

1. Números de secuencia: 000-999 como caracteres de largo fijo (3) y se reciclan cuando se acaban (el siguiente a 999 es 000). Todo paquete enviado/recibido va con estos tres caracteres de prefijo, incluso el paquete vacío que detecta el EOF (que consiste en sólo el número de secuencia, un paquete de largo 3 bytes).

2. Timeout de emisión: si pasa más de ese timeout sin haber recibido el paquete de vuelta, lo retransmitimos.

3. Cálculo de pérdidas: al retransmitir un paquete, incrementamos un contador de errores

4. Al terminar el proceso, imprimimos un mensaje con el total de paquetes, los errores, y el porcentaje de error.

Un ejemplo de medición sería (suponiendo que tengo un servidor corriendo en anakena en el puerto 1818):

```
% time ./client_bw.py 1400 0.1 anakena.dcc.uchile.cl 1818 /etc/services  OUT
sent 486 packets, lost 2, 0.411522633744856%
4.38 real         0.17 user         0.14 sys
```

Un esquema del protocolo va en la página siguiente.

3. Mediciones

El cliente sirve para medir eficiencia. Lo usaremos para ver cuánto ancho de banda logramos obtener y también probar cuánto afecta el largo de las lecturas/escrituras y el timeout en la eficiencia.

Para probar en localhost necesitan archivos realmente grandes, tipo 0.5 o 1 Gbytes (un valor que demore tipo 5 segundos). Para probar con anakena, basta un archivo tipo 500 Kbytes.

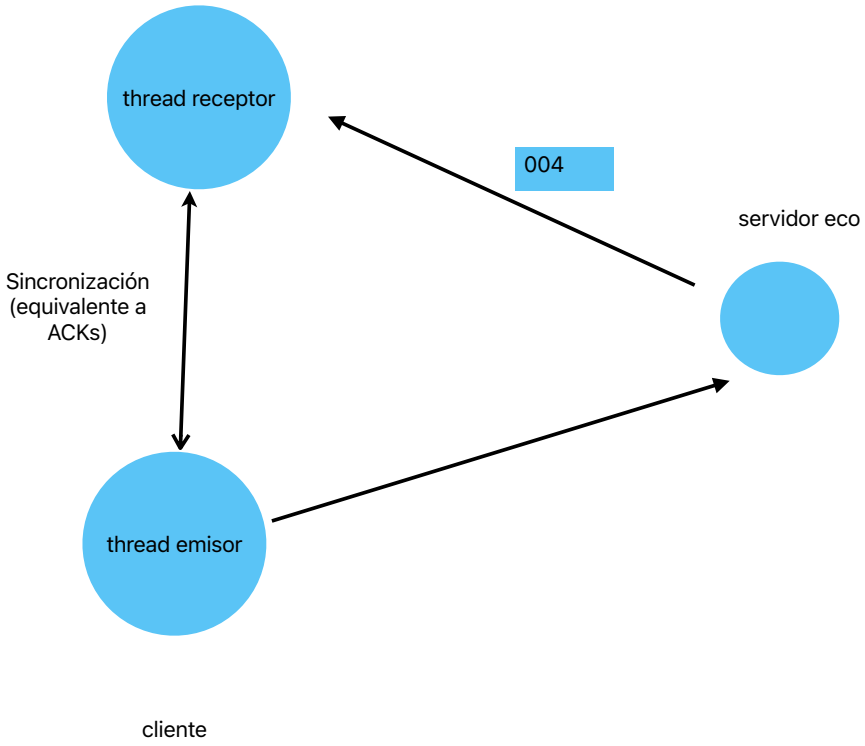
Midan y prueben en las mismas condiciones que probaron la T2 pero no es necesario ahora correr varios threads en paralelo. Reporten sus resultados y las diferencias que obtengan.

Responda las siguientes preguntas (se pueden basar en los valores medidos en la T1 como referencia de los que son valores "correctos", la T2 no sirve porque se perdían muchos datos):

1. Si el archivo de salida es más pequeño que el de entrada, ¿puede ocurrir aunque mi protocolo esté bien implementado?

2. Los valores de ancho de banda medidos en esta tarea son muy distintos a los de la T1 y T2. ¿cuál es la causa principal de estas diferencias?

3. Dijimos que un protocolo bien implementado no debiera nunca morir por el timeout de 3 segundos puesto en el socket de recepción. ¿Por qué es esto así?


thread receptor
004
servidor eco
Sincronización
(equivalente a
ACKs)
thread emisor
cliente

4. Entregables

Básicamente entregar el archivo con el cliente que implementa el protocolo, una descripción de los experimentos y un archivo con los resultados medidos, tanto para localhost como para anakena, una comparación con la T1, la T2 y sus respuestas a las preguntas.