Tarea 2: Cliente eco UDP para medir performance
Redes
Plazo de entrega: 21 de octubre 2024

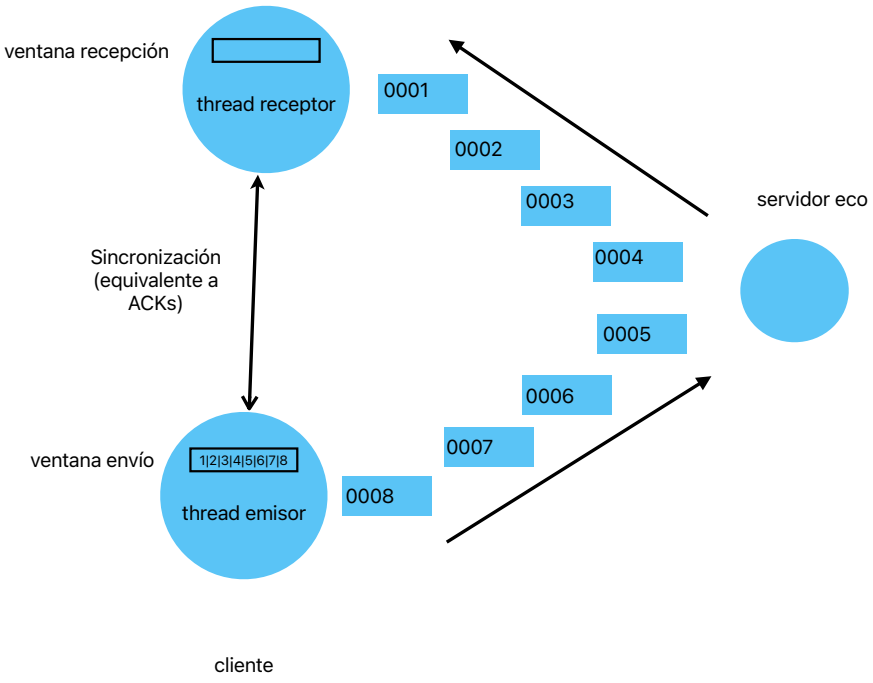
José M. Piquer

1. Descripción

Su misión, en esta tarea, es modificar el cliente con threads de la tarea 1 para hacer que funcione usando UDP y corrigiendo los errores de transmisión. Usaremos el mismo servidor de eco UDP normal y solo modificaremos el cliente.

Ahora los dos threads del cliente se sincronizarán como si fueran un emisor y un receptor en un protocolo clásico de ventana corredera. La idea es que implementen el mejor protocolo posible: si hacen un Stop-and-Wait y no se pierde nada tienen un 4.0, si hacen un Go-Back-N bueno tienen un 5.5, si hacen un Selective-Repeat bueno tienen un 7.0.

La estructura de la solución se parece a la de la Tarea 1, pero ahora los dos threads deben además sincronizarse para control de flujo: en Stop-and-Wait deben esperar la recepción de cada paquete, en Go-Back-N y en Selective-Repeat deben detenerse si la ventana de envío se llenó. Deben mantener una ventana de envío en el thread emisor y una de recepción en el thread receptor. La única diferencia importante con un protocolo normal, es que no usarán ACKs, ya que pueden compartir memoria entre emisor y receptor, por lo que el receptor le puede avisar al emisor qué paquetes ha recibido e incluso los threads pueden mirar la ventana del otro (con mutex y esas cosas eso sí). Ver diagrama.


ventana recepción
thread receptor
0001
0002
0003
0004
0005
0006
0007
0008
servidor eco
Sincronización
(equivalente a
ACKs)
ventana envío
1|2|3|4|5|6|7|8
thread emisor
cliente

Para poder implementar eso, se les pide usar números de secuencia en los paquetes, con 2 bytes binarios que representan el número de secuencia (0-65535), representación *big endian*. Los números de secuencia cuentan los paquetes enviados. Todos los paquetes que envíen deben venir con esos dos caracteres, y así los reciben igual. Cuando definimos el tamaño máximo de paquete a usar, ahora debe incluir esos 2 caracteres, por lo que deben leer paquetes de datos desde el archivo de entrada que sean (tamaño máximo de paquete)-2.

Como siempre: Stop-and-Wait usa ventanas de tamaño 1, Go-Back-N usa una ventana de envío de tamaño N y una de recepción de tamaño 1, Selective-Repeat usa ambas ventanas de tamaño N.

El tamaño de la ventana (N) es un parámetro que deben recibir (salvo en el caso de Stop-and-Wait) y va entre 1 y 32767.

La idea es que la mayoría de las pérdidas de datos se producen cuando el emisor va demasiado rápido y no da tiempo a que el paquete de respuesta llegue. Por eso, es importante en esta tarea permitir que los dos threads ejecuten en paralelo y el receptor pueda ir recibiendo paquetes lo más rápido posibe, mientras el emisor debe detenerse si la ventana de envío se le llena con paquetes que no han sido recibidos aún. Para eso, manejen los mutex de forma que los bloques protegidos sean lo más pequeños posibles y asegúrense de soltarlos de vez en cuando para que el otro thread pueda entrar.

Usaremos entonces los argumentos: el tamaño de paquete que se usará y el tamaño de la ventana (N) cuando no usemos Stop-and-Wait.

Para retransmitir, se les pide implementar un timeout adaptativo: parten con un timeout de 0.5s, pero luego deben calcular el tiempo de ida y vuelta de los paquetes (RTT), es decir el tiempo transcurrido entre el envío de un paquete y su recepción de vuelta. Deben ignorar los paquetes retransmitidos. Cada vez que logren calcular un RTT, definan el timeout como $RTT * 3$.

Como ahora es UDP, no tenemos un cierre de conexión, por lo que, al terminar el envío, deben enviar un paquete de tamaño cero (es decir, los 2 números de secuencia correspondientes y nada más). Al recibir ese paquete, saben que es el último de la secuencia.

Si están en Stop-and-Wait o Go-Back-N, recibir el último paquete les permite terminar todo. Si están en selective-repeat, no es tan simple, por que pueden haber paquetes pendientes de recepción aun. En ese caso, sólo pueden terminar cuando sacan el paquete vacío de la ventana de recepción.

El cliente que deben escribir recibe el tamaño de paquete a proponer, la ventana, servidor y puerto. En la entrada estándar el archivo a enviar y en la salida estándar el archivo que se recibe. Al terminar escriba en la salida de errores el tamaño de paquete y ventana y los errores corregidos:

retransmisiones y paquetes recibidos fuera de orden.

```
./copy_client.py pack_sz win  host port <filein >fileout
```

Hay un servidor de eco corriendo en anakena.dcc.uchile.cl puerto 1818 UDP. Pero, puedan correrlo localmente para pruebas con localhost también.

Un ejemplo de ejecución sería:

```
% ./copy_client.py 8000 10000 anakena.dcc.uchile.cl 1818 </etc/services > OUT
Usando: pack: 8000, maxwin: 10000
Errores envío: 2
Errores recepción: 2
```

Escriban ese mismo tipo de salida de errores en sus clientes, para que podamos corregirlos fácilmente.

Si su programa está bien escrito, el archivo de entrada y el de salida deben quedar con contenido idéntico.

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.

Para generar números de secuencia binarios, usen las funciones primitivas de python: `int.to_bytes()` y `int.from_bytes()`

Para el desarrollo del programa, les recomiendo ir en orden: implementar primero un stop-and-wait, luego hacer un Go-Back-N y luego un Selective-Repeat. De esa forma, van asegurando nota en la tarea si se demoran más de lo esperado (estas tareas son más complicadas en la práctica de lo que parecen en la teoría).

De la misma forma, para cada protocolo, les recomiendo probar primero en localhost hasta que todo funcione, y ahí pueden ir a probar a anakena, donde todo será más difícil.

2. Entregables

Básicamente entregar el archivo con el cliente que implementa el protocolo.

En un archivo aparte responder las preguntas siguientes (digamos, unos 5.000 caracteres máximo por pregunta):

1. Genere algunos experimentos con diversos tamaños de paquete y ventanas (con anakena). Mida el tiempo total que toma la transferencia y recomiende los mejores valores para su caso según sus resultados.

2. Compare sus resultados con lo logrado en la Tarea1. Comente las diferencias.

3. Si implementó más de un protocolo, compare Stop-and-Wait, Go-Back-N y Selective-Repeat. Pruebe en localhost y en anakena. ¿Son coherentes sus resultados con la teoría?

4. En esta tarea pusimos que el timeout es 3 veces el último RTT medido. ¿Podemos saber si ese valor es a veces demasiado pequeño y generó retransmisiones innecesarias?

5. De toda esta experiencia, ¿qué cree Ud que es más importante para ser eficientes?: ¿tamaño paquete, tamaño ventana, protocolo, timeout?