Tarea 1 Redes

Entrega: 29 de Septiembre 2014

1. Objetivos

Esta tarea persigue construir una capa de transporte simple sobre UDP, que permita un protocolo confiable usando Stop-and-Wait.

Para esto, se les entregan los fuentes en C de un cliente y una implementación de Stop-and-Wait correcta, usando números de secuencia en los paquetes y en los ACKs, entre 0 y 1. Esta implementación no funciona bien frente a errores, por que los timeouts son fijos y altos, para asegurarnos de no tener falsos timeouts.

Una forma de mejorar este comportamiento, es usar timeouts variables, que se calculan en base a los RTT (round-trip-time) reales de la conexión entre el cliente y el servidor. Se les pide modificar la capa de datos para implementar un manejo de timeouts óptimo, mejorando la velocidad del traspaso de archivos frente a errores.

2. Aplicación

Es un cliente y un servidor que intercambian un archivo para medir ancho de banda en ambas direcciones. Se les provee el fuente del cliente. Uds deben modificar el archivo que contiene la capa de datos (transporte).

3. Capa Datos provista

La capa datos original usa paquetes con un header de 3 bytes: el primero indica la conexión a la que pertenece (entre 0 y 255), el segundo indica el tipo de paquete ('C', 'D' o 'A') y el tercero es un número de secuencia (entre 0 y 1).

- Número de conexión

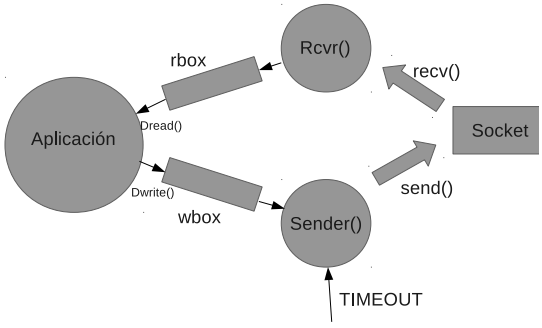
  Es un identificador de conexión, entero de un byte entre 0 y 255.

- Tipo de Paquete

  1. Conexión 'C': el cliente, inicialmente envía un paquete con sólo el header, con tipo conexión, número de conexión 0 (byte en cero) y número de secuencia 0 (byte en cero). El servidor responde con un paquete de tipo ACK, el número de conexión asignado (0 - 255) y número de secuencia 0. Este paquete se debe re-enviar 10 veces, esperando 3 segundos por el ACK, antes de abandonar.
  2. Datos 'D': es un paquete de datos normal. Se espera 1 segundo el ACK correspondiente y se re-intenta 10 veces antes de abandonar.

3. ACK ’A’: es un paquete de 3 bytes (sólo header), que indica qué paquete de la conexión fue recibido OK, lleva el número de secuencia al que corresponde.

La implementación de esta capa es un poco complicada, por que se necesita que opere en paralelo con la aplicación, recibiendo datos desde el socket y re-transmitiendo mientras la aplicación puede estar haciendo cualquier otra cosa. El diseño que sigue esta implementación se puede ver en este dibujo:



4. Capa Datos pedida

Se les pide modificar la capa datos anterior de modo de tener un timeout adaptable, que evite las retransmisiones inútiles, pero a la vez aproxime lo mejor posible el RTT.

Se les entrega un servidor ejecutables que ya implementa esto para que lo prueben contra su cliente. Si ejecutan el cliente original contra el servidor modificado, pueden ver la diferencia de velocidad de transferencia al agregarle pérdida a la conexión. La transferencia desde el cliente al servidor es responsabilidad de Uds, la otra es responsabilidad del servidor.

Para calcular bien los timeouts se les sugieren los siguientes algoritmos:

- RTT: marcar hora del envío y calcular la diferencia con la hora de llegada del ACK respectivo
- Descartar retransmisiones: no deben actualizar el RTT cuando reciben un ACK para un paquete retransmitido (puede ser del primer envío)
- El timeout es: $1{,}1 * RTT$
- Timeout nunca debe ser mayor que 3s ni menor que 0.005s

Cualquier otra idea o aporte que hagan será bienvenido, mientras logre mejor ancho de banda utilizado sin saturar la red con paquetes inútiles.

5. Ejecución

Para compilar la tarea, recomendamos hacer un Makefile. Mantengan la separación de las funciones de comunicación de la aplicación para que puedan reusar la aplicación para las próximas tareas.

La versión original es `bwc-orig` (cliente) y les proveo el ejecutable del servidor adaptable `bws`. Para probarlo deben correr el servidor primero (con opción de debug):

```
% ./bws -d
```

y luego el cliente:

```
% ./bwc-orig -d archivo-in archivo-out ::1
```

Se les proveen todos los fuentes para compilar el cliente:

```
% gcc bwc.c jsocket6.4.c Dataclient.c bufbox.c -o bwc-orig -lpthread -lrt
```

La versión modificada la pueden probar con el servidor ejecutable (`bws`) que viene con los archivos, y usar localhost (`::1` o `127.0.0.1`).

Para probar con pérdidas, fuercen a localhost para que genere pérdidas aleatorias. En linux, usen ”netem”, usualmente basta hacer como superusuario (instalar paquete kernel-modules-extra):

```
% tc qdisc add dev lo root netem loss 20.0%
```

Y tienen 20 % de pérdida. Para modificar el valor, deben usar replace en vez de add en ese mismo comando.

Cualquier duda o pregunta o reporte de bugs, dirigirse al foro de U-cursos.

6. Entrega

A través de U-cursos, no se aceptan atrasos. La evaluación será en función de que logren mejorar las tasas de transferencia con un 20 % de pérdida, sin hacer retransmisiones inútiles. Se hará una lista relativa de velocidad lograda contra retransmisiones

inútiles y la mejor tarea tendrá un 7.0. La velocidad la pueden ver en el número que da el cliente, las retransmisiones inútiles en el DUP que va dando el servidor en modo debug. Una tarea que tiene la misma performance del original tiene un 1.0. El código entregado DEBE compilar y ejecutar algo.

Al código, acompáñenlo de un archivo LEEME.txt donde explican los algoritmos utilizados y las mediciones que hicieron para validar que funcionan bien.