Redes de ordenadores
Transmisión libre de errores de un fichero

1 Introducción

El propósito de este trabajo es el desarrollo de una aplicación original para la transferencia de un fichero, utilizando como servicio de transporte el protocolo UDP. La aplicación resultante deberá ser capaz de adaptarse a escenarios de red diversos, manteniendo unas buenas prestaciones.

La aplicación consistirá en dos procesos independientes, uno actuando de emisor de la información (fuente), y el otro de receptor (sumidero). Diferentes escenarios de red se emularán con la ayuda de un tercer programa capaz de alterar la secuencia de los paquetes recibidos, de introducir errores en la transmisión, retardos de propagación... Este emulador de red ya está implementado, y podéis descargarlo de `https://github.com/RedesdeOrdenadores/ShuffleRouter` o de `https://snapcraft.io/shufflerouter`.[1]. Para ver las opciones, podéis ejecutar *shufflerouter -h*

2 Los tres procesos

2.1 El emulador de red

Se trata de una sencilla aplicación -que ya se os da hecha- que debe recibir, a modo de paso intermedio obligatorio, todo paquete transmitido por ambos extremos de la comunicación (fuente y sumidero), reenviándolos a su destino natural tras retrasarlos un tiempo aleatorio entre `min_delay` y `min_delay` + `rand_delay`. Como consecuencia del retardo introducido, los paquetes pueden abandonar este proceso fuera de orden.

El emulador de red también puede descartar paquetes con probabilidad `drop`.

El emulador de red debe recibir el identificador de la aplicación destino (dirección IP y puerto), a donde se dirige un paquete que lo atraviesa, en los primeros 6 octetos de la cabecera de aplicación que habrá incorporado el extremo transmisor correspondiente. Los primeros cuatro octetos contendrán la dirección IP destino, mientras que el quinto y sexto, el número de puerto, ambos en formato de red (big-endian). A modo de ejemplo, los primeros seis bytes de un paquete con IP y puerto destino (127.0.0.1, 2000) serían, en notación hexadecimal, 0×7f 0×00 0×00 0×01 0×07 0×d0. El emulador modificará estos seis octetos al reenviar el paquete al destino con la dirección IP original y el puerto de la aplicación de origen.

2.2 La aplicación emisora

Su objetivo es leer un fichero y transmitirlo, de forma fiable y con eficiencia, a la aplicación receptora. Su invocación por línea de comandos requiere cinco argumentos:

```
ro2021send input_file dest_IP dest_port emulator_IP emulator_port
```

Para hacer su trabajo, la aplicación emisora del fichero deberá implementar un algoritmo de retransmisión (ARQ), y necesitaréis definir un protocolo de comunicación para que ambos

[1] Recomendamos instalar la versión basada en snap, ya que se actualizaría automáticamente en caso de que detectemos algún fallo/defecto en su funcionamiento.

extremos se entiendan. Las características del algoritmo ARQ, y la definición de los mensajes que implica su operación, se dejan libres a vuestra elección. Eso sí, tened en cuenta que el emisor no podrá conformar paquetes mayores de 1472 bytes.

Una vez finalice la transmisión (y todos los datos hayan sido asentidos por el receptor), la aplicación emisora debe acabar automáticamente.

2.3 La aplicación receptora

La ejecución de la aplicación receptora del fichero requiere dos argumentos en la línea de comandos:

```
ro2021recv output_file listen_port
```

Este extremo de la comunicación podrá conocer la dirección IP y el puerto donde escucha el emulador de red inspeccionando los metadatos asociados a los datagramas recibidos. No olvidemos que todo paquete, recibido y transmitido, tiene que atravesar el emulador de red.

La aplicación receptora debe finalizar su ejecución tras recibir todo el fichero transmitido. Y una posibilidad para detectar dicho final es la utilización de un paquete de datos de tamaño cero para que la fuente señalice al receptor tal eventualidad.

3 Estrategias ARQ

La implementación más sencilla de una estrategia ARQ se corresponde con **Parada y Espera** ya que no implica multiplexación de eventos y puede seguir la misma lógica del sencillo programa de suma que ya realizamos, sin más que añadir un bucle en el emisor. El emisor transmite un paquete y se queda a la espera de recibir un ACK. Tras recibirlo envía un nuevo paquete y así hasta terminar el fichero. Si en algún momento de la espera, el ACK tardase demasiado y vence el temporizador, el paquete debe retransmitirse. Llamamos a este temporizador *Retransmission TimeOut* (RTO).

Este RTO puede fijarse a un valor constante o adapatarse de forma dinámica a las variaciones que se van produciendo en el RTT, de forma que ante RTT bajos el RTO tendría un valor inferior y se podría retransmitir antes, y ante RTT altos el RTO aumentaría y no se realizarían retransmisiones innecesarias. Resulta obvio que la implementación de un RTO dinámico da lugar a mejores prestaciones que si es estático. Ver sección 6.

La implementación de Parada y Espera con RTO dinámico permite llegar hasta **7 puntos**; y nuestra **recomendación** es que este sea vuestro objetivo inicial.

Para obtener una calificación superior será necesario lograr mejores prestaciones y ello implicará necesariamente la implementación de una estrategia de **envío continuo**, es decir, usar una ventana en transmisión.

Podemos implementar una estrategia Go-Back-N modificando sólo el emisor y usando el mismo receptor de Parada y Espera. En este caso, cuando se pierde un mensaje, éste será siempre el primero de la ventana de transmisión y habrá que retransmitir la ventana completa.

El problema de usar envío continuo es que se requiere multiplexar eventos de envío y recepción en el emisor que no sean bloqueantes. Ver sección 4.

Go-Back-N permite llegar hasta **9 puntos.**

Para obtener la **máxima calificación** habría que mejorar las prestaciones complicando el receptor mediante el uso de una **ventana** también en **recepción**, de forma que el receptor sólo acepta un paquete si cae dentro de la ventana y, en ese caso, se mueve el inicio de la ventana al

primer hueco (paquete todavía no recibido). Tal y como vimos en clase, tenemos dos posibilidades de asentimiento y retransmisión a) asentimiento simple y acumulativo (por bloques) b) asentimiento y Retransmisión selectivos

Damos libertad al alumno que desee ir a por la máxima calificación (o asegurar una buena puntuación) para implementar el mecanismo que desee.

4 La multiplexación de eventos

Para la implementación de cualquier estrategia de retransmisión de envío continuo, la fuente de la información no debe bloquearse mientras su ventana de transmisión se lo permita y siga teniendo datos que enviar. Esto lleva a que un programa que implemente esta estrategia deba estar siempre a la espera de tres sucesos: la llegada de un asentimiento, el vencimiento de una temporización y la oportunidad de transmitir un nuevo paquete. En los lenguajes de programación habituales, las operaciones de E/S relacionadas con los sockets (`send/receive`; `read/write`) son, por defecto, bloqueantes. Ello implica que un receive bloqueará al proceso que lo invoca mientras no se reciba un nuevo paquete.[2] En la lógica que debe seguir el emisor de nuestro proyecto, este debe intercalar lecturas del fichero a transmitir, para conformar nuevos paquetes e introducirlos en la red, con el procesamiento de los asentimientos que envía el lado receptor. Obviamente, procesar un asentimiento debe hacerse si hay un asentimiento que procesar, es decir, si ya ha sido recibido y aguarda en la cola de recepción del socket correspondiente. En ese caso, un receive del proceso que maneja el socket extraería el asentimiento sin demora. Pero si nuestra fuente invoca un receive sin que haya todavía un ACK, esta se bloquearía, rompiendo la continuidad que exige nuestro algoritmo ARQ. Esto impide poder simultanear la espera por un asentimiento con la transmisión de tráfico.

En definitiva, se necesita un mecanismo que permita conocer si hay datos pendientes de recepción sin necesidad de bloquear el proceso. En los sistemas UNIX esto se realizaba tradicionalmente con la función select, que permite comprobar, incluso trabajando simultáneamente con más de un socket a la vez, el estado de cada uno de ellos, decidiendo el programador el no bloquearse, o esperar, un tiempo máximo configurable, por algún evento que involucre a alguno de los elementos controlados; y, en especial, por el evento que indica la llegada de nuevo tráfico. De este modo, el programador puede llamar a receive sin riesgo de interrumpir el flujo del programa. Desafortunadamente, las clases del paquete java.net, que nos aporta las clases `DatagramSocket/DatagramPacket` que ya conocemos, no poseen ninguna funcionalidad equivalente.[3]

Para poder realizar operaciones *no bloqueantes* con sockets en Java se requiere utilizar el paquete java.nio; y manejar —en nuestro caso— tres de sus clases: `DatagramChannel`, `Selector` y `ByteBuffer`. En el siguiente enlace tenéis ejemplos de cómo utilizar este nuevo paquete: `http://tutorials.jenkov.com/java-nio/index.html`.

Un objeto de la clase `DatagramChannel` es análogo a la clase DatagramSocket que ya conocéis. Con `Selector` podréis comprobar el estado de la cola de recepción del socket ligado al canal (método `SelectNow()`), y gestionar las retransmisiones por vencimiento de los tempo-

[2] Del mismo modo un send no finalizará hasta que los niveles inferiores (de la arquitectura de comunicaciones local) tengan espacio para acomodar más datos en el flujo de salida habitual, aunque en muchos casos esto no supone un problema, pues es posible transmitir un paquete en una red actual en una fracción de milisegundo.

[3] El método setSoTimeout que utilizasteis en la tarea inicial no admite un valor que indique "espera cero", y no nos vale utilizar el valor mínimo que sí admitiría (1 ms). Destacar, no obstante, que para la implementación de "Parada y Espera" sí es suficiente, pues existe un único temporizador y el emisor solo puede encontrarse en dos estados: esperando un asentimiento temporizado o transmitiendo un nuevo paquete.

rizadores de retransmisión (método `Select(timeout)`). Usaréis el primero si queréis transmitir más datos y el segundo si no podéis transmitir hasta que se asienta un nuevo paquete por haber agotado la ventana de transmisión. Por último, la clase ByteBuffer os resultará muy útil para construir/procesar fácilmente los paquetes de la aplicación (con su cabecera particular), gracias a sus variados métodos `get` y `put`. Os aconsejamos utilizarla igualmente en Parada Y Espera (y en ambos extremos de la comunicación). En cualquier caso, en el extremo receptor del fichero, donde cualquier acción (generación de asentimientos, escritura en disco, etc.) es provocada siempre por la recepción de un paquete de datos, no se necesitan estos nuevos recursos, y un simple receive bloqueante es más que suficiente.

5 Los números de secuencia

Debéis tener también en cuenta que, a la hora de implementar la estrategia de retransmisión, debéis usar un rango de números de secuencia mayor que el explicado en teoría. Para obtener este último, se suponía que todas las transmisiones en el canal ocurrían en orden y que, por tanto, no podía ocurrir que una retransmisión "adelantase" a la primera transmisión de un paquete. Si eso sucede, y el número de secuencia llega a reutilizarse, puede ocurrir que dicha primera transmisión sea aceptada como un paquete de la nueva ventana de transmisión por el receptor.[4]

La manera más sencilla de evitar esto es, aún manteniendo un tamaño máximo de ventana, usar números se secuencia de rango muy grande, de modo que se garantice que estas transmisiones antiguas retrasadas han abandonado la red (han sido rechazadas por estar fuera de secuencia en el receptor) cuando el número de secuencia sea válido. Os recomendamos numerar las tramas con enteros de 32 bits para evitar el problema.

6 El temporizador de retransmisión

El valor del temporizador de retransmisión extremo a extremo (RTO) debe ajustarse dinámicamente para adaptarse tanto a las características del camino entre fuente y destino (enlaces y número de nodos de conmutación intermedios), como a la impredecible carga o congestión en la red, que hará que el tiempo invertido en las colas de los routers atravesados pueda variar sensiblemente a lo largo del tiempo de vida de una comunicación.

Para conseguir dicho ajuste dinámico, la fuente debe medir continuamente (utilizando cada paquete de datos transmitido, y su correspondiente asentimiento) el valor del RTT. Un sencillo sello temporal en el paquete de datos, que será devuelto en el ACK que genere, es lo que requiere el emisor para calcular el valor puntual del tiempo de ida y vuelta.

Para evitar que el RTO fluctúe demasiado es necesario estabilizarlo, para lo que se suele emplear una media móvil. Relacionadas con el protocolo TCP, existen varias alternativas para su implementación, siendo la siguiente la más utilizada:

$\widehat{rtt}$ **(Smoothed Round-Trip Time)** es la media ponderada entre el rtt (medido) y el último $\widehat{rtt}$ calculado:

Medida 1: $\widehat{rtt}^1 = rtt^1$

Medida k: $\widehat{rtt}^k = (1-\alpha) \times \widehat{rtt}^{k-1} + \alpha \times rtt^k, \alpha < 1$ (Valor estándar $\alpha = \frac{1}{8}$)

[4] En la redes que tienen varias rutas posibles entre fuente y destino, puede ocurrir que paquetes que van por una ruta adelanten a paquetes que se han transmitido antes y que han recorrido una ruta más congestionada. En la práctica ocurre cuando el retardo que introduce el *shufflerouter* a los paquetes no es constante.

Desviación del RTT ($\widehat{\sigma_{rtt}}$)

Medida 1: $\widehat{\sigma_{rtt}}^1 = \frac{rtt^1}{2}$

Medida k: $\widehat{\sigma_{rtt}}^k = (1-\beta) \times \widehat{\sigma_{rtt}}^{k-1} + \beta \times |\widehat{rtt}^k - rtt^k|, \beta < 1$ (Valor estándar $\beta = \frac{1}{4}$).

Temporizador de Retransmisión (Algoritmo de Jacobson/Karels)

$RTO = \widehat{rtt} + 4 \times \widehat{\sigma_{rtt}}$

Cada paquete de la ventana de transmisión tendrá su correspondiente temporizador de retransmisión. La gestión eficiente de todos ellos puede hacerse invocando cada vez el select con un horizonte temporal variable igual al tiempo que falta en ese momento para que venza el RTO de vencimiento más inminente.

7 Comprobación de los argumentos de entrada

Ambos programas solo tendrán que comprobar que el número de argumentos de entrada es correcto. En caso de que este chequeo falle, el correspondiente programa finalizará su ejecución, mostrando previamente un mensaje sugiriendo la sintaxis de invocación correcta.

8 Entrega y evaluación

El código fuente podrá desarrollarse en C, C++, Python 3, Java o Rust. En cualquier caso, las versiones requeridas para probar/evaluar el programa tienen que ser las ya instaladas en los laboratorios de la Escuela.

Los dos programas a desarrollar deben llamarse `ro2021send` y `ro2021recv`, respectivamente.

En caso de programar en C o C++, es obligatorio incluir un `Makefile` que automatice la compilación del proyecto. Si el desarrollo es en Java, la clase principal tiene que residir en el paquete por defecto (*default package*).

La fecha límite para la entrega del código, que deberá ser **original** y **meridianamente legible**, se publicará en el portal de la materia.

Se entregará un único fichero ".zip" que contenga una carpeta donde resida vuestro trabajo. El nombre de dicha carpeta (y, en consecuencia, la versión comprimida a entregar) será el DNI del estudiante (e.g.: 12345678.zip). Las entregas que no sigan este formato no serán admitidas.

Utilizaremos un procedimiento de evaluación que se apoye en las prestaciones del producto, midiendo, para diferentes escenarios, el tiempo invertido y el número de bytes transmitidos para la **correcta** transferencia de un fichero. En la tabla se muestran las pruebas que se realizarán para obtener la calificación indicada en la primera columna. Más adelante se completará la tabla indicando los valores máximos que se deben obtener para el tiempo de transferencia y la cantidad total de bytes transmitidos, para enviar correctamente el fichero. Huelga decir que a mayor sofisticación del algoritmo ARQ implementado, mejor nota.

Puntuación	File size (MB)	Min. delay(ms)	Rand. delay (ms)	Drop rate	Tiempo (s)	Bytes tx. (MB)
5	10	5	0	0.1%	250	30.000.000
+0.5	10	5	0	0.1%	100	12.000.000
+0.25	5	20	0	0.1%	200	7.500.000
+0.25	5	5	15	0.1%	300	20.000.000
+0.25	5	5	15	1%	120	8.000.000
+0.5	10	5	0	10%	110	15.000.000
+0.25	5	20	0	0%	200	6.000.000
+0.5	100	5	5	0.1%	600	1.000.000.000
+1	200	50	0	0.01%	300	2.000.000.000
+0.5	200	100	0	0.1%	150	1.000.000.000
+0.5	200	0	100	1%	300	250.000.000
+0.5	200	0	100	10%	300	1.000.000.000

Cuadro 1: Relación entre prestaciones y calificación final.