Tarea 2: medidor ancho de banda UDP confiable usando Go-Back-N
Redes
Plazo de entrega: 29 de mayo 2023

José M. Piquer

1. Descripción

Su misión, en esta tarea, es generar un cliente que use un socket UDP que permita medir ancho de banda total para enviar datos entre él y un servidor aplicando un protocolo de corrección de errores eficiente. Este cliente lee un archivo de entrada y lo envía completo al servidor (quien lo recibe), usando un protocolo corrección de errores con ventanas GoBackN. Luego, el cliente avisa que terminó (como un fin de archivo) y el servidor envía hacia el cliente la cantidad de bytes que recibió como parte de su último ACK. Una vez terminado, el cliente toma los bytes recibidos por el servidor y los divide por el tiempo transcurrido desde el inicio hasta la recepción del resultado desde el servidor (usar la función `time.time()`), entregando un número de Megabytes por segundo (1 Megabyte = 1024*1024 bytes).

Siendo un socket UDP, habrá pérdida de paquetes, por lo que su cliente debe corregir y retransmitir la ventana cuando sea necesario. Se les provee el código de un cliente en Python que implementa Stop-and-Wait con el servidor, y que es muy ineficiente en su medición. Pueden tomar ese código y modificarlo para implementar GoBackN o partir de cero, como prefieran.

Un tema importante en los sockets UDP es qué tamaño de buffer usar cuando uno invoca `send()` y `recv()`. En teoría es mejor usar paquetes grandes, envían más información por menos costo, y debieran ser más eficientes. Pero el tamaño puede afectar también la probabilidad de pérdida, y puede no ser tan bueno.

De la misma forma, ahora importan otros parámetros: el tamaño de la ventana (que debe aproximar el BDP del enlace, pero si hay pérdidas implica retransmitirla completa), el procentaje de pérdida, el valor del timeout (que debe aproximar el máximo RTT). El cliente proporcionado (Stop-and-Wait)

recibe el tamaño de paquete a proponer, el valor del timeout (en ms), pérdida (en porcentaje), archivo a enviar, servidor, puerto:

```
./bwc-sw size timeout loss filein host port
```

El cliente provisto trae funciones para enviar y recibir con probabilidad de pérdida. En pruebas reales, con anakena, habrán además pérdidas reales, por lo que siempre el porcentaje real será un poco mayor.

El que Uds deben construir debe recibir además el tamaño máximo de la ventana a usar:

```
./bwc-gbn size timeout loss win_size filein host port
```

Deben implementar Go-Back-N con una ventana de paquetes del tamaño especificado. Deben implementar ACKs acumulativos: si reciben ACK para el paquete n debe implicar que recibieron ACK para todos lo paquetes j que estén en la ventana y cumplan con $j \leq n$. (ojo que la ventana es "circular", por lo que no siempre esto es obvio. Yo uso la función `between()` para eso:

```
# Revisar si x está en la ventana: preguntar primero si Win_sz > 0!!!
def between(x, min, max):      # min <= x < max
    if min <= max:
        return min <= x and x < max
    else:
        return (min <= x or x < max)
```

En esta implementación uso min como el número de secuencia del primer paquete en la ventana y max como el número de secuencia del próximo paquete a entrar a la ventana (por eso uno es menor o igual y el otro es menor estricto).

Esta tarea busca probar distintos valores para los parámetros, de modo que estudien la sensibilidad a ellos y busquen una propuesta óptima para el caso que están probando.

Para poder probar eso, el archivo de entrada debe leerse en bloques de bytes, del tamaño del paquete máximo, para ser enviados. En Python, pueden usar archivos binarios (de bytes) para eso y la función `read()`.

Hay un servidor corriendo en anakena.dcc.uchile.cl puerto 1819 UDP con este protocolo. En el 1818 sigue corriendo el servidor de la T1 para que puedan comparar. Uds tienen que implementar el cliente usando GoBackN (el servidor para Stop-and-Wait y Go-Back-N es el mismo). El código del servidor también se les entrega para que puedan probar en localhost su

desarrollo (pero es mucho mejor usar anakena para probar en condiciones reales).

En muchos ejemplos, verán que se usan threads para permitir flujos bidireccionales en Go-Back-N, pero en este caso no es necesario ya que el cliente sólo envía datos y recibe ACKs (no hay datos que fluyan desde el servidor hacia el cliente). Lo único complicado es manejar los bloqueos y condiciones, ya que el ciclo de envío debe revisar varios casos distintos y bloquearse en el recv() esperando el timeout de vez en cuando:

- ventana con paquetes: debo revisar si han llegado ACKs por el socket y procesarlos
- timeout: si ha pasado más tiempo que el timeout desde que envié el paquete más antiguo en la ventana, debo retransmitirla entera. Después de 100 re-intentos de la misma secuencia, deben abortar la conexión y terminar con error.
- espacio en la ventana: si la ventana no está en su máximo, debe leer paquetes desde el archivo, agregarlos a la ventana y transmitirlos (anotando el tiempo de su envío)

Pueden usar `sock.settimeout(val)` para programar el timeout del próximo `sock.recv()` que hagan. Si ponen `val=0` el socket se vuelve no-bloqueante y pueden preguntar si hay datos sin bloquearse.

El cliente nuevo debe invocarse como:

```
% ./bwc-gbn.py 5000 20 5 10 /etc/services anakena.dcc.uchile.cl 1819
propuse paquete: 5000
recibo paquete: 5000
bytes enviados=677977, bytes recibidos=670000, time=0.09839272499084473, bw=6.
```

los parámetros son: paquetes de 5.000 bytes (más el header), 20 ms de timeout, 5 % de pérdida y ventana máxima de 10 paquetes.

2. Protocolo

1. Cliente: envía 'C00xxxx'
2. Servidor: responde 'A00yyyy'

3. Cliente: envía paquetes con todo el contenido de filein, siempre comienzan con 'D' y dos bytes que codifican el número de secuencia (entre 00 y 99) y son de tamaño máximo yyyy bytes más el header utilizar (o sea, resultan paquetes UDP de yyyy+3 máximo):

   ```
   'D01kkkkkkkkkk'
   'D02jjjjjjjjjj'
   ....
   ```

4. Servidor: envía ACKs con el número de secuencia del último paquete recibido bien. Usa una ventana de recepción de tamaño 1.

   ```
   'A01'
   'A02'
   ....
   ```

5. Cliente: al terminarse el archivo envía 'E' con su número de secuencia en un paquete sólo y con eso marca el fin de la transmisión

   ```
   'E81'
   ```

6. Servidor: envía 'Annzzz..zz' donde nn es el número de secuencia y zzz..zz es un string (largo variable) que contiene el número de bytes recibidos por el servidor en total. El ACK al EOF es el único que no es de largo 3 bytes e incluye información extra al final.

   ```
   'A81623154'
   ```

   Es un ACK al EOF número 81 e informa que recibió 623154 bytes del archivo.

En Python, estas secuencias son *bytearray*. Los números xxxx yyyy son los tamaños de datos máximo, codificados como *bytearray* de números como 4 caracteres. Los números de secuencia son números de dos caracteres. Por ejemplo, si el cliente quiere usar un tamaño de 900 bytes, enviará su paquete de conexión (siempre con el número de secuencia 00):

```
C000900
```

(pueden ver que el tamaño máximo de paquete que se puede usar es 9999 y el número máximo de secuencia es 99)

Como son paquetes UDP en el socket, no necesitan fin de línea después.

El protocolo resiste archivos binarios (ejecutables, imágenes) como entrada y debiera funcionar bien. Esto quiere decir que lo que viene después de la 'D' inicial Uds no pueden suponer que es un string, sino que es una secuencia de bytes binarios (que es lo que en Python es un bytearray). Esto debe ser así al enviarlo y al recibirlo, tanto del socket como del archivo.

Como es UDP, pueden perderse paquetes en la red. Pero, si el protocolo está bien implementado, los bytes del archivo siempre deberían corresponder en forma exacta a lo que reporta el servidor.

Al abortar por exceso de intentos, no deben medir el ancho de banda, sólo reportar el error.

3. 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, ventana, timeout y pérdidas y haga una recomendación de valores a utilizar para las distintas pérdidas. Compare con la Tarea1 y con el cliente stop-and-wait. Grafique sus resultados.

2. Discuta si la medición de ancho de banda obtenida es "válida": al haber retransmisiones, ¿estamos midiendo mal? Los resultados obtenidos: ¿son coherentes? ¿se parecen a la realidad? Argumente con sus propias mediciones realizadas en el punto anterior.

3. Revise si ocurre desorden de paquetes: ¿recibe siempre los ACKs en orden? Si no es así, ¿el protocolo debiera funcionar igual? ¿En algún caso extremo de desorden podría fallar?

4. Les pedimos implementar Go-Back-N con ACKs acumulativos. Si uno no quisiera eso, y quisiera esperar los ACKs uno por uno, ¿funcionaría el protocolo con este servidor? ¿Se puede hacer que funcione?

4. Strings y bytearrays en sockets

Una confusión clásica en los sockets en Python es la diferencia entre enviar un string y/o un bytearray. Partamos por los strings, que Uds conocen mejor: los strings no son simplemente arreglos de bytes (alguna vez lo fueron, pero hoy pueden contener hasta alfabetos asiáticos y árabes), son codificaciones en un *encoding* particular. Los sockets no soportan strings, es decir, Uds no pueden llegar y enviar/recibir un string por el socket, deben convertirlo a un bytearray, que es una colección de bytes binarios primitivos, no se interpretan. La forma de convertir un string a un bytearray es aplicando la función *encode* y un bytearray a un string, con la función *decode*. Ojo que si se aplican a cualquier cosa, pueden fallar, particularmente decode falla si uno le pasa cualquier cosa en el bytearray. Entonces, si quiero enviar/recibir el string 'niño' hago:

```
enviador: s = 'niño'
          sock.send(s.encode('UTF-8'))      # UTF-8 es el encoding clásico hoy
receptor: s = sock.recv().decode()          # recibe s == 'niño'
          print(s)
```

En cambio, si leo un archivo cualquiera y quiero enviarlo, no sé si hay strings o no dentro, entonces mejor es no transformarlo y siempre enviar bytes puros:

```
data = fd.read(MAXBYTES)
sock.send(data)
```

En esta tarea hay una mezcla de ambas cosas: los 'comandos' ('C', 'E', etc) son letras, que deben ser transformadas a bytearrays antes de enviar al socket. En Python, ord('C') transforma la letra a bytearray, y chr(b) transforma un bytearray en letra. El string con el total de bytes recibidos desde el servidor es un string codificado a bytes que deben decodificar. Pero todo el resto (lo que viene después de las 'D' que uds envían en cada paquete de datos) es un bytearray puro que Uds obtuvieron desde el archivo, no deben tratar de codificarlo a string.