Tarea 2 CC4303

- Prof. de Cátedra: José M. Piquer
- Auxiliar: David Miranda
- Estudiante: Máximo Flores Valenzuela

Preámbulo

Este documento resuelve las consultas pedidas, de la 1. a la 5., usando como máximo 5000 caracteres. Cada pregunta está en una única hoja en las páginas de a continuación.

Todos los experimentos que se presentan en este documento se realizaron usando WSL 2 con Debian 12 Bookworm. Las especificaciones del entorno son las siguientes:

- Sistema operativo de 64 bits, procesador basado en x64.
- Procesador 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz.
- 32 GB de RAM instalada.
- Información sobre *threads* por *core*/*cores* por *socket*/*sockets*:

```
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           1
```

- Cachés:

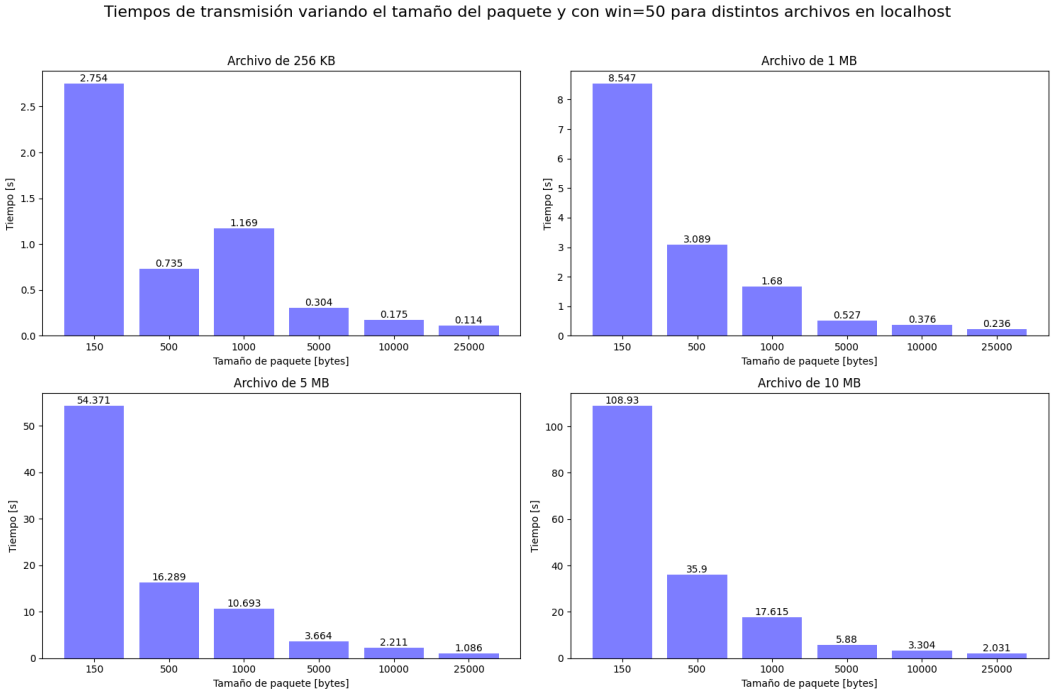
```
Caches (sum of all):
L1d:                   288 KiB (6 instances)
L1i:                   192 KiB (6 instances)
L2:                    7.5 MiB (6 instances)
L3:                    12 MiB (1 instance)
```

Bajo estas condiciones, el valor máximo de `pack_size` que se pudo experimentar en el servidor `anakena.dcc.uchile.cl` fue 1444. Para `pack_size` > 1444, los paquetes nunca eran recibidos por el hilo receptor. Esto fue verificado analizando el tráfico con `tcpdump`.

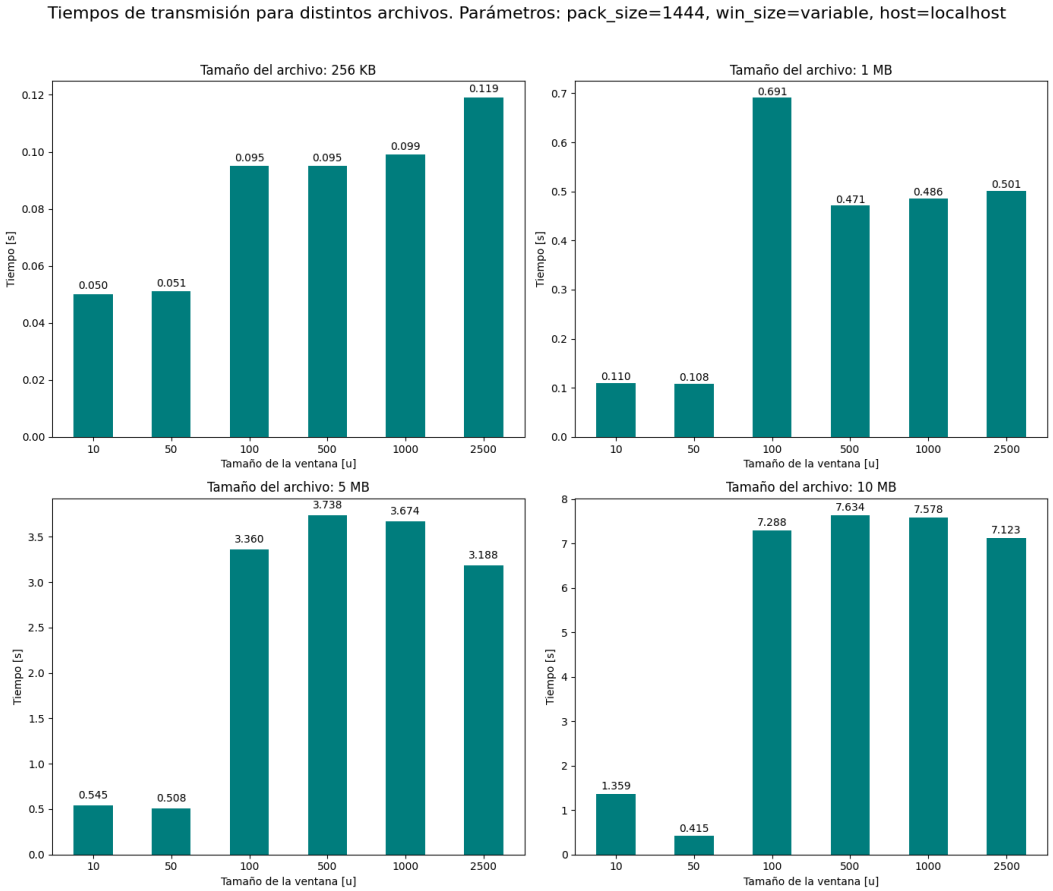
Para todos los experimentos, se usó el protocolo de transmisión *Go-Back-N* y su caso particular *Stop-and-Wait* cuando $N = 1$. No se implementó *Selective Repeat*. También, es importante mencionar que dada la contingencia con el servidor Anakena, los resultados se obtuvieron desde ejecuciones en `localhost`.

Pregunta 1

- Variando `pack_size` con los valores $\{150, 500, 1000, 2500, 10000, 25000\}$ y manteniendo `win_size` $= 50$ (ejecutar *script* `variable_pack_size.sh` para reproducir).



- Variando `win_size` con los valores $\{10, 50, 100, 500, 1000, 2500\}$ y manteniendo `pack_size` en 1444 (ejecutar *script* `variable_win.sh` para reproducir).



De los resultados del experimento anterior, se puede concluir que la mejor combinación de `pack_size` y `win_size` dadas las condiciones del entorno es $\texttt{pack\_size}_{\text{OPT}} = 25000$ y $\texttt{win\_size}_{\text{OPT}} = 50$. Son los valores que proveen mejores resultados en cuanto a tiempos de transmisión, considerando que en todos los casos la transmisión fue de un $100\ \%$, sin duplicaciones ni faltas.

Pregunta 2

El experimento se realizó en `localhost` probando con dos archivos: uno de $1\,\text{MB}$ y otro de $10\,\text{MB}$. Se realizaron 5 iteraciones de cada método (`tcp`: `server_echo_2`, `server_echo_4`, `server_echo_5`; `udp`: `go_back_n`), y se fijaron como variables de interés el tiempo real de transmisión y la tasa de transferencia, es decir, qué tanto recibe el hilo receptor de los datos enviados por el emisor. En todos los casos, se consideró `pack_size` $\leftarrow 1444$, y en el caso particular de *Go-Back-N*, se fijó `win_size` $\leftarrow 50$.

Los resultados se muestran a continuación. Cada entrada de las siguientes tablas es de la forma $(t; \overline{x}; \sigma)$ donde t es la mínima tasa de transmisión de las 5 iteraciones, calculada como $t = \text{bytes recibidos}/\text{bytes enviados}$, y $\overline{x}$ y σ son el promedio y la desviación estándar respectivamente de los tiempos totales de transmisión.

- `localhost` enviando un archivo de $1\,\text{MB}$.

`go_back_n` (`udp`)	`server_echo2` (`tcp`)	`server_echo4` (`tcp`)	`server_echo5` (`tcp`)
$(100\ \%; 0,11; 0,005)$	$(100\ \%; 0,042; 0,006)$	$(100\ \%; 0,067; 0,025)$	$(100\ \%; 0,052; 0,012)$

- `localhost` enviando un archivo de $10\,\text{MB}$.

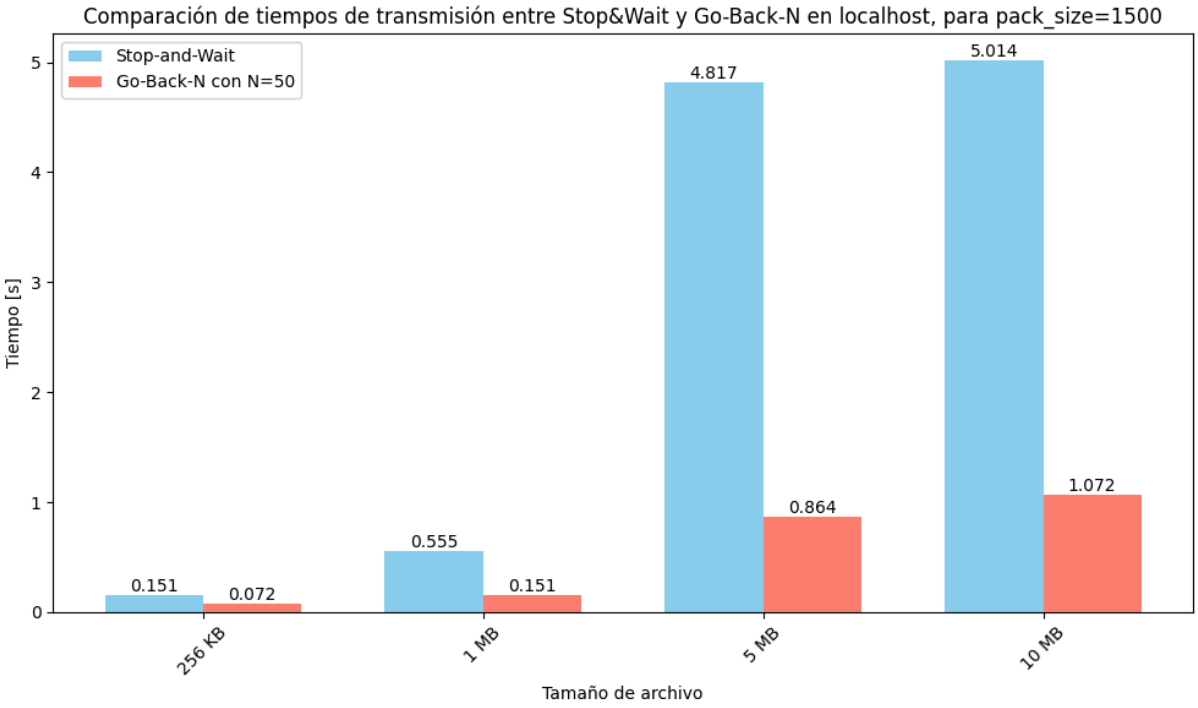
`go_back_n` (`udp`)	`server_echo2` (`tcp`)	`server_echo4` (`tcp`)	`server_echo5` (`tcp`)
$(100\ \%; 0,423; 0,047)$	$(100\ \%; 0,131; 0,006)$	$(100\ \%; 0,126; 0,012)$	$(100\ \%; 0,117; 0,011)$

A partir de estos resultados, se puede comentar que en los escenarios descritos se comporta mejor el protocolo TCP. No hay ninguna pérdida ni duplicación de información en las transmisiones, pero todos los servidores TCP superan levemente a *Go-Back-N* en el tiempo total de transmisión.

En la práctica, el protocolo TCP debería ser más lento, pues es menos propenso a errores. Esto no significa que el experimento esté mal hecho, o el protocolo de transmisión *Go-Back-N* esté mal implementado, pues hay más factores a considerar. Por ejemplo, no se implementó *Selective Repeat* (UDP), entonces no se puede concluir a partir de comparaciones con alternativas al método de transmisión implementado.

Pregunta 3

Para este experimento, se comparará *Stop-and-Wait* y *Go-Back-N* en `localhost`. La implementación de *Go-Back-N* permite implementar *Stop-and-Wait* cuando $N = 1$. A continuación, se muestran los resultados obtenidos (ejecutar *script* `sw_vs_gbn_localhost.sh` para reproducir):



Los resultados sí son coherentes con la teoría. *Stop-and-Wait* debiese ser más lento que *Go-Back-N* en entornos donde hay una probabilidad de pérdida y *delay*. Esto se debiese notar más en `anakena.dcc.uchile.cl`, dado que `localhost` no es representativo de las conexiones que ocurren en el día a día en la realidad.

Pregunta 4

Sí, es posible saberlo creando un experimento que varíe el valor del *timeout* adaptativo. Se probarán los ponderadores $c \in \{1, 2, 2.5, \mathbf{3}, 3.5, 4\}$, donde $\boldsymbol{c = 3}$ es el valor sugerido. En esencia, cada vez que podemos calcular un RTT, el valor del *timeout* adaptativo se actualizará a $c \cdot \mathrm{RTT}$.

Para este experimento, se probarán 5 ejecuciones en `localhost` para un archivo de tamaño 10 MB, con `pack_size` $\leftarrow 150$ y `win_size` $\leftarrow 2500$ (las peores métricas de la pregunta 1 en cuanto a tiempo). Se mirarán los errores de envío del *script*, y para cada valor de c, se calcularán los promedios $\overline{x}[c]$ y desviaciones estándar $\sigma[c]$ sobre esta métrica.

Si se desea reproducir este experimento, basta con cambiar en el código la constante `TIMEOUT_FACTOR` que aparece en las definiciones de las líneas iniciales al valor de c que se desee analizar, establecer `TEST_TRANSMISSION_ERRORS` a `True`, y correr el script `test_transmission_errors.sh`.

Los resultados obtenidos se muestran a continuación:

Ponderador c	$\overline{x}[c]$	$\sigma[c]$
1	$424,6$	$1,02$
2	$423,4$	$0,49$
2.5	$423,2$	$0,4$
$\mathbf{3}$	$\mathbf{423}$	$\mathbf{0}$
3.5	423	0
4	423	0

De estos resultados, se puede concluir que sí hay mayor tasa de retransmisión cuando el ponderador c es menor. Esto es consistente con la práctica, pues los *timeouts* tienen como cota inferior el valor $\mathrm{RTT} + \delta$, donde $\delta \geq 0$ es una constante para asegurar que el paquete haya llegado. A pesar de que las diferencias son minúsculas, se deduce que esto es así porque se está probando en `localhost`. En `anakena.dcc.uchile.cl` existen más retransmisiones, pues es un escenario más cercano a lo real.

Si $c < 1$, entonces el *timeout* de la ventana es menor que un RTT. En dicho caso, siempre habrían retransmisiones, pues no se le da tiempo al paquete para ir y volver. Para probar esto, se estableció $c = 0,01$, y los errores de transmisión aumentaron a más de 1.000, por lo tanto, la teoría es cierta. Cuando $c = 0,05 < 0,1$, la diferencia no se nota tanto. La hipótesis es que esto sucede porque el RTT es variable. Esto último se pudo verificar mediante el proceso de depuración.

Pregunta 5

Después de realizar los experimentos y una investigación sobre cómo se comportan estos protocolos en la práctica, se puede concluir que todos los parámetros son importantes, y dependen del contexto de la red (probabilidad de pérdida, *delay*, etc.).

Por un lado, si el tamaño de los paquetes enviados no es bien elegido, generaremos muchas más transmisiones. Basta pensar que para transmitir un archivo de tamaño T *bytes* se necesitarán al menos $\lceil T/p \rceil$ emisiones, donde $p =$ `pack_size` en *bytes*. Es claro que el cuociente disminuye si p es mayor. En los resultados de la pregunta 1 se puede ver que mientras más grande sea `pack_size`, menor es el tiempo de emisión, pasando en el peor caso de 108 segundos a 2.

Después, si el tamaño de la ventana no es bien elegido, se podrían generar muchas retransmisiones innecesarias en el caso de *Go-Back-N*, dado que retransmite toda la ventana. Esto se nota mucho más cuando hay alta probabilidad de pérdida, pues el receptor necesita aceptar un paquete muy específico (el "sucesor modular" del último aceptado, en términos de números de secuencia).

Por otro lado, *Stop-and-Wait* mostró ser más lento que *Go-Back-N* en el escenario más cercano al óptimo obtenido en la pregunta 1.

Finalmente, el *timeout* también es importante. Si no consideramos que en la vida real el RTT es variable, y tampoco sabemos que esta métrica juega un rol fundamental en el cálculo del *timeout*, podemos generar retransmisiones innecesarias. Esto combinado con una mala elección de parámetros `pack_size` y `win_size` puede generar congestión en la red.

Así, la conclusión obtenida es que no hay algo "más importante". La importancia en la eficiencia se puede medir como una combinación lineal de criterios asociados a cada variable del experimento (`pack_size`, `win_size`, *timeout*, protocolo). Se requeriría experimentar con mayores volúmenes de datos e implementar *Selective Repeat* para poder esclarecer la ponderación que tendría cada variable en la medición matemática de la eficiencia.