# *Sincronización de archivos*

## *Ejercicio N° 1*

| | |
|---|---|
| **Objetivos** | • Buenas prácticas en programación de Tipos de Datos Abstractos (TDAs)<br>• Modularización de sistemas<br>• Correcto uso de recursos (memoria dinámica y archivos)<br>• Encapsulación y manejo de Sockets |
| **Instancias de Entrega** | **Entrega 1:** clase 4 (28/3/2016). **Entrega 2:** clase 6 (12/4/2016). |
| **Temas de Repaso** | • Uso de structs y typedef<br>• Uso de macros y archivos de cabecera<br>• Funciones para el manejo de Strings en C<br>• Funciones para el manejo de Sockets |
| **Criterios de Evaluación** | • Criterios de ejercicios anteriores<br>• Cumplimiento de la totalidad del enunciado del ejercicio<br>• Ausencia de variables globales<br>• Ausencia de funciones globales salvo los puntos de entrada al sistema (*main*)<br>• Correcta encapsulación en TDAs y separación en archivos<br>• Uso de interfaces para acceder a datos contenidos en TDAs<br>• Empleo de memoria dinámica de forma ordenada y moderada<br>• Acceso a información de archivos de forma ordenada y moderada |

# Índice

# Introducción

Supongamos que hay dos computadoras, A y B, conectadas en red. Supongamos que en la computadora A hay un archivo F1 y en la B hay un archivo *muy similar* a F1 llamado F2.
¿Cómo se puede hacer para que la computadora A pueda tener una copia de F2?

Una solución trivial es simplemente conectarse a B y copiar todo el contenido de F2 en A. Pero si el canal de comunicación entre A y B es lento o el tamaño del archivo F2 es enorme, es una solución demasiado ineficiente.

Una alternativa más inteligente consiste en enviar a A solo las *diferencias* entre F1 y F2. De esa manera A puede reconstruir F2 en su maquina a partir de F1 y las diferencias. Pero esto requiere que en la maquina B exista una copia de F1, de otro modo no podrá calcular las diferencias entre F1 y F2.

El algoritmo de rsync [1] permite sincronizar las dos versiones, reconstruyendo F2 en la maquina de A a partir de F1 y de las diferencias entre estos, calculadas por la maquina B.
La estrategia consiste primero en que A calcule una serie de *checksums* al archivo F1 y se los envíe a la maquina B.
En B, se utilizan estos *checksums* para saber qué partes del archivo F2 tiene en común con F1 y qué partes son distintas calculando efectivamente las diferencias entre estos.
Esta información se le envia a A quien reconstruye finalmente F2.

En este trabajo práctico se implementará una versión reducida y simplificada del algoritmo para sincronizar dos archivos.

# Descripción

## Calculo del Checksum

El *checksum* propuesto en [1] es una variante del conocido *checksum* Adler32. Para un bloque cuyos bytes son $X_0, X_1, \ldots, X_{B-1}$ su cálculo consiste en los siguientes pasos:

1- Calcular los valores intermedios *lower* y *higher* según:

$$lower = \left( \sum_{i=0}^{B-1} X_i \right) \mod M$$

$$higher = \left( \sum_{i=0}^{B-1} (B-i) X_i \right) \mod M$$

Donde $B$ es el tamaño del bloque (**block_size**), $M$ es la constante 0x00010000 ($2^{16}$) y $X_i$ es el byte numero $i$ del bloque.

2- Finalmente calcular el *checksum* según:

$$checksum = lower + \left(higher * 2^{16}\right)$$

Una propiedad interesante de este *checksum* es que permite lo que se conoce como *rolling checksum* y permite obtener un *checksum* a partir de otro de manera muy eficiente bajo ciertas condiciones.
Suponga que $C_1$ es el *checksum* de la secuencia de bytes $X_0, X_1, \ldots, X_{B-1}$,

$$C_1 = lower_1 + \left(higher_1 * 2^{16}\right)$$

Sea el bloque continuo movido un solo byte, esto es entonces, $X_1, X_1, \ldots, X_B$ (misma secuencia sólo que el primer byte $X_0$ se removió y entró uno nuevo por derecha $X_B$), entonces su *checksum* $C_2$ puede calcularse a partir del *checksum* previo, del byte que salió y del byte que entró:

$$lower_2 = \left(lower_1 - X_0 + X_B\right) \mod M$$
$$higher_2 = \left(higher_1 - B * X_0 + lower_2\right) \mod M$$

Finalmente:

$$C_2 = lower_2 + \left(higher_2 * 2^{16}\right)$$

## Protocolo del request del cliente al servidor

El cliente tiene acceso a un archivo local **old_local_file** y desea actualizarlo con una versión más nueva, un archivo **new_remote_file** que se encuentra en el servidor.
Para ello se conectara al servidor y le enviara

- El nombre del archivo remoto (**new_remote_file**)
- El tamaño del bloque usado en el *checksum*
- Una lista de *checksums*, uno *checksum* por cada bloque.

Para enviar el nombre del archivo remoto el cliente le enviará **4 bytes** al server que representan un **int** con la longitud **L** en bytes del nombre; acto seguido le envía el nombre del archivo.
Del lado del server, el server espera por los primeros **4 bytes**, los interpreta como la longitud **L** del nombre y luego espera por los siguientes **L bytes** que representan el nombre del archivo.

Para enviar el tamaño del bloque, el cliente enviará **4 bytes** representando ese tamaño. El server sólo necesita recibir **4 bytes** e interpretarlos.

Para enviar la lista de *checksums* la situación es ligeramente más compleja. El cliente leerá **block_size bytes** del archivo, le calculará su *checksum* y le enviará al server **un byte** con el numero **0x01** seguido de **4 bytes** que tendran el *checksum* del primer bloque. Luego el cliente leerá los siguientes **block_size bytes** y volverá a enviarle al server **un byte** con el número **0x01** seguido de **4 bytes** con el segundo *checksum*, repitiendo este proceso hasta llegar al final del archivo.
Si el cliente llega a leer menos bytes que **block_size** porque alcanzó al fin de archivo antes de poder completar el último bloque, este se descarta.
Finalmente le envía un mensaje el server de fin de lista. Esto se hará enviando al server **un byte** con el número **0x02**.
Del lado del server, este ira obteniendo *checksum* a *checksum* mientras que cada mensaje empiece con el

identificador **0x01**; el server sabrá que la lista de *checksum* terminó cuando reciba el identificador **0x02**.

***Nota:*** *el algoritmo original de rsync[1] usa dos tipos de checksums: una variante del Adler32 y el MD4 o MD5. Este segundo checksum se usa para reducir la probabilidad de falsos positivos, esto es, dos bloques distintos pero que por casualidad tienen el mismo valor en el primer checksum. Con fines de simplificar el trabajo práctico no se implementará el segundo checksum y, como puede verse a continuación, ni siquiera se contempla la posibilidad de usarlo.*

## Protocolo de update del servidor al cliente

Como se describió previamente, el servidor el archivo a leer (**new_remote_file**), el tamaño de los bloques para el *checksum* (**block_size**) y una lista de *checksums* (**checksum_list**) con los *checksums* de cada uno de los bloques del archivo original (**old_local_file**) del lado del cliente en secuencia.

***Nota:*** *la lista de checksums será usada por el servidor reiteradas veces como se verá a continuación. Una implementación eficiente en la búsqueda de un checksum en particular sería lo ideal. Sin embargo, con fines de simplificar el trabajo práctico, se puede usar cualquier estructura de datos.*

Ahora, el server esta en posición de abrir el archivo **new_remote_file** y de buscar en cualquier posición del archivo un bloque de **block_size bytes** cuyo *checksum* esté en la lista **checksum_list**.

- Si hay una coincidencia,
    - Si hay bytes leídos pero que no fueron enviados al cliente y que están fuera del bloque actual, se los envía. Estos son los bytes que están en el archivo **new_remote_file** pero no en el **old_local_file**.

  Luego, le envía al cliente el número de bloque encontrado **NB**. Esto significa que el bloque de datos está presente en ambos archivos, **new_remote_file** y **old_local_file**, pero en vez de enviarles esos datos sólo le envía el número de bloque.
  Seguido de esto, el server lee los siguientes **block_size bytes** .

- Si no hay coincidencia, el servidor continúa su búsqueda, moviéndose de a un byte a la vez.

El servidor continúa hasta llegar al fin del archivo. En ese punto envía todos los bytes faltantes que no fueron enviados previamente y luego le notifica al cliente el fin de archivo.

Para enviar una serie de bytes (nuevos para el cliente), el server envía **un byte** con el número **0x03** seguido de **4 bytes** que codifican la longitud **L** de bytes a transferir y luego envía **L bytes** con los datos propiamente dichos.
El cliente leerá esos **L bytes** y los escribirá en el archivo **new_local_file**.
El cliente imprimirá por pantalla el mensaje

```
RECV File chunk L bytes
```

Para enviar el número de bloque **NB** que ambos tienen en común, el server envía **un byte** con el número **0x04** seguido de **4 bytes** con el número del bloque **NB**.
El cliente leerá **NB**, luego buscará dicho bloque del archivo **old_local_file** y escribirá el contenido del mismo en el archivo **new_local_file**. Nótese cómo este algoritmo evita que el server envíe datos que el cliente puede obtener de forma local.
El cliente imprimirá por pantalla el mensaje

```
RECV Block index NB
```

Finalmente, para notificarle al cliente que se llegó al fin del archivo, el server envía **un byte** con el número **0x05**.
El cliente imprimirá por pantalla el mensaje

```
RECV End of file
```

Para este punto el cliente habrá escrito en **new_local_file** una copia exacta del **new_remote_file** a partir de datos copiados del **old_local_file** y de datos transferidos desde el server.

En este punto tanto el cliente como el servidor finalizan.

# Formato de Línea de Comandos

Hay dos formas de ejecutar el programa: una en modo servidor que tendrá la versión actualizada de un archivo en particular; el otro, modo cliente, que tiene una versión desactualizada de un archivo y se conectará al server para actualizarlo.

Para ejecutar en modo servidor:

```
./tp server port
```

Donde **port** es el puerto o servicio en donde escuchara el socket.

Para ejecutar en modo cliente:

```
./tp client hostname port old_local_file new_local_file new_remote_file block_size
```

Los parámetros **hostname** y **port** son el hostname/IP y el servicio/puerto del server remoto.
Donde **old_local_file** es el archivo que se desea actualizar y que el cliente tiene acceso (es local a la maquina); **new_local_file** es el nombre del archivo que el cliente va a crear (este no existe aún); **new_remote_file** es el nombre del archivo que tiene la versión actualizada pero este archivo no se encuentra en la maquina del cliente sino en la del server (es remoto).
El parametro **block_size** es es tamaño en bytes de los bloques usados en el cálculo del *checksum*.

# Códigos de Retorno

El programa retorna **1** si hubo un error en los parametros. En todo otra situación, retorna **0**.

# Entrada y Salida Estándar

No se lee nada de la entrada estándar pero si se imprimen en la salida estándar tal como se lo describe en la sección *Protocolo de update del servidor al cliente.*
No está permitida la escritura de otros mensajes, aún bajo condiciones de falla. En estas situaciones, el programa debe cerrar ordenadamente todos los recursos y retornar el código 0 silenciosamente.

# Ejemplos de Ejecución

Supongamos que tenemos el archivo old con el siguiente contenido

```
aabbcc
```

Y tenemos el archivo remote con

```
xaazzzccy
```

Si lanzamos el servidor:

```
./tp server 8081
```

Y luego lanzamos el cliente:

```
./tp client 127.0.0.1 8081 old new remote 2
```

donde ambos programas corren en la misma maquina y **2** fue elegido como **block_size.** Entonces, al finalizar la ejecución, se debería crear un archivo **new** con el contenido igual al de **remote**.
Además, el cliente debería imprimir por pantalla lo siguiente:

```
RECV File chunk 1 bytes
RECV Block index 0
RECV File chunk 3 bytes
RECV Block index 2
RECV File chunk 1 bytes
RECV End of file
```

Veamos que sucedió paso a paso.

En primer lugar el cliente se conectó al servidor y le envio lo siguiente (los números se escriben aquí en hexadecimal, big endian y el resto se escribe en ASCII para su fácil lectura. Además, se resaltan en distintos colores):

```
00 00 00 06 r e m o t e 00 00 00 02
```

Como puede verse, se envían 4 bytes con la longitud del nombre del archivo remoto, seguido de éste y luego de 4 bytes más con el tamaño del bloque.

El cliente ahora lee el archivo **old** de a bloques de 2 bytes y le calcula el *checksum* a cada uno:

```
aa          bloque 0 y checksum 0x012300c2
bb          bloque 1 y checksum 0x012600c4
cc          bloque 2 y checksum 0x012900c6
```

Esta informacion se la envía al servidor junto con el fin de lista:
01 01 23 00 c2 01 01 26 00 c4 01 01 29 00 c6 02

Ahora el servidor sabe que archivo tiene que usar (**remote**), cual es el tamaño del bloque (**2**) y la lista de

*checksums*:

bloque 0 con checksum 0x012300c2
bloque 1 con checksum 0x012600c4
bloque 2 con checksum 0x012900c6

Ahora es el turno del servidor, abriendo el archivo **remote** y cargando el primer bloque (se escribe a continuación el archivo completo, pero sólo lo que está entre corchetes es lo que está en memoria):

```
[xa]azzzccy
```

El *checksum* del bloque es 0x015100d9 y no esta en la lista de *checksums*, asi que el la ventana de búsqueda se desplaza un byte:

```
x[aa]zzzccy
```

El *checksum* del bloque es 0x012300c2 y coincide con el bloque número 0.

***Nota:*** *este segundo checksum se puede calcular de dos maneras: calculando el checksum del bloque por definición o actualizando el checksum del bloque anterior usando la propiedad de rolling checksum. Si bien la segunda es deseable por ser más eficiente, usted puede optar por cualquiera de ambas.*

Bien, hemos encontrado una coincidencia con el bloque número 0, pero como tenemos bytes que fueron leídos pero no fueron enviados al cliente y están por fuera de nuestro bloque (esto es, el byte `x`), hay que enviar este chunck de datos primero:

```
03 00 00 00 01 a a
```

El cliente escribe en el archivo **new** `aa` y imprime por pantalla

```
RECV File chunk 1 bytes
```

Seguido de esto el servidor envía el número de bloque coincidente (0):

```
04 00 00 00 00
```

Y el cliente lee del archivo **old** el bloque número 0 (`aa`) y lo copia en el archivo **new**. Luego imprime por pantalla

```
RECV Block index 0
```

El servidor ahora no mueve la ventana de búsqueda un byte sino que la mueve **block_size** bytes (**2**).

```
xaa[zz]zccy
```

Su *checksum* es 0x016e00f4, no hay coincidencias. Nos movemos un byte:

```
xaaz[zz]ccy
```

Su *checksum* es 0x016e00f4, no hay coincidencias. Nos movemos un byte:

```
xaazz[zc]cy
```

Su *checksum* es 0x015700dd, no hay coincidencias. Nos movemos un byte:

```
xaazzz[cc]y
```

Tenemos una coincidencia con el bloque nuúmero 2. Ademas tenemos bytes sin transferir (zzz) asi que

enviamos:

03 00 00 00 03 z z z

Y luego:

04 00 00 00 02

A lo cual el cliente escribe en **new** `zzz` e imprime:

```
RECV File chunk 3 bytes
```

Para luego escribir el bloque número 2 (`cc`) en **new** e imprimir:

```
RECV Block index 2
```

El servidor ahora lee los siguientes **block_size** bytes (2) y alcanza el fin de archivo. Asi que transmite el último chunk de datos y luego un fin de archivo:

03 00 00 00 01 y 05

El cliente recibe el último chunk y lo escribe en **new** y tambien recibe la notificación del fin de archivo. Imprime por pantalla:

```
RECV File chunk 1 bytes
RECV End of file
```

# Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C (C99) / ISO C++98.
2. Está prohibido el uso de variables globales y funciones globales (salvo el punto de entrada ‘main’).

# Referencias

[1] El algoritmo de rsync: https://rsync.samba.org/tech_report/tech_report.html
[2] Referencia C/C++: http://www.cplusplus.com/reference/