TP 1: Sincronización de archivos

Taller de Programación

Agustina Barbetta 96528

29 de Marzo
1er. Cuatrimestre 2016

1. Introducción

Este trabajo práctico tiene por objetivo la implementación de una versión reducida y simplificada del algoritmo *rsync*. El algoritmo de *rsync* permite sincronizar directorios locales y remotos. Suponiendo que hay dos computadoras, A y B, conectadas en red y hay un archivo F1 en la computadora A y otro archivo muy similar en la computadora B llamado F2. El algoritmo permitirá reconstruir un archivo F2 en una máquina de A a partir de F1 y de las diferencias entre estos, calculadas por la máquina B. La estrategia consiste primero en que A calcule una serie de *checksums* del archivo F1 y se los envíe a la máquina 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 envía a A quien reconstruye finalmente F2.

2. Implementación

2.1. Socket

Para llevar a cabo la comunicación entre computadoras, se desarrolló el tipo de dato abstracto `socket_t`. A las firmas de las primitivas dadas en clase se les ha agregado el parámetro `struct addrinfo*`, de esta forma se realiza el llamado a la función `getaddrinfo` por fuera y se pasa solo un resultado, logrando así una estructura mas flexible.
Las primitivas de `socket_t` suelen encapsular el uso de las funciones de la librería `sys/socket.h`, exceptuando `socket_send()` y `socket_receive()`, las cuales reciben la cantidad de bytes que deben enviar/recibir y permanecen en un ciclo llamando a *send/recv* hasta que la transferencia se haya completado.
Para el manejo de errores se creó el enumerado `socket_error_t`, sabiendo que todas las funciones de la librería `sys/socket.h` devuelven -1 en caso de error, se compara el resultado de cada llamado con la constante `ERROR_CODE` (La cual es -1) y se devuelve el código de error correspondiente en caso de coincidencia. Si la operación fue exitosa, se devuelve 0.

2.2. Checksum

Para el cálculo de los *checksums*, se implemento la variante del algoritmo *Adler-32* descripta en el enunciado. El mismo se encuentra en un módulo independiente al resto del programa, de esta forma se facilita su reemplazo.

2.3. List

El cliente A recorre su archivo F1, calcula *checksums* de sus bloques y los envía al servidor B. El servidor debe guardarlos para luego notificar al cliente si hubo alguna coincidencia entre

los bloques de F1 y F2. Para cumplir esta tarea, los *checksums* deben ser guardados uno a uno en una estructura fácil de recorrer, de principio a fin, en múltiples oportunidades.
Para cumplir con estas tareas se implementó un tipo de dato abstracto `list_t`. La estructura que representa la lista contiene un vector de punteros `void*`, de manera que la lista pueda contener elementos de cualquier tipo, un `size_t` para su tamaño y otro para su longitud. La diferencia entre estos últimos es que el tamaño es la cantidad de memoria reservada para el vector y la longitud es la cantidad de posiciones del vector que fue utilizada. Se agregan los elementos por medio de la primitiva `list_append()`, la cual los anexa al final de la lista, y se accede a ellos por medio de `list_get()` proveyendo el índice del elemento buscado.
La lista es dinámica y se redimensiona al doble de su tamaño cada vez que esta llena y se intenta agregar un nuevo elemento.
El manejo de errores se planteó diferente al del TDA `socket_t`. En este caso, a mi criterio, no resulta tan importante identificar que fue exactamente lo que falló (especialmente porque los errores solo pueden deberse a parámetros inválidos o falla de `malloc` o `realloc`). Lo importante es poder determinar si una operación falló o no. Es por esto que, en lugar de usar un enumerado con diferentes códigos de error, se utilizaron booleanos. Las primitivas devuelven `true` si todo salió bien, `false` si algo salió mal.
Cabe aclarar que la lista fue también utilizada para guardar los nuevos bytes para F1 leídos en el archivo F2, aprovechando así su dinamismo ya que nunca se sabe cuantos bytes nuevos habrá.

2.4. File handler

Para evitar repetición de código y encapsular la lógica de apertura, lectura, escritura y cierre de archivos se implemento el TDA `file_handler_t`. A diferencia de `list_t`, su primitiva de creación no devuelve un puntero a su estructura, lo recibe, ya que se busca utilizar el *stack* para guardarla, creando y destruyéndola en una misma función.
Se aprovecha el hecho de que la lectura es siempre de a n bytes, recibiendo este n en el constructor, de esta forma no necesita recibirlo cada vez que realiza una lectura.
Con respecto al manejo de errores, se implementa con booleanos de la misma forma que en la estructura `list_t`.

2.5. RSync

A continuación se presentan los TDAs implementados para representar a las computadoras A y B.
En ambos casos, a fin de simplificar el trabajo práctico, no se manejaron errores arrojados por los tipos o estructuras manipuladas. Todas las primitivas devuelven `void`.

2.5.1. RSync local

El TDA `rs_local_t` contiene primitivas para iniciar la comunicación con el servidor y enviar los datos del archivo F1 a sincronizar; Longitud del nombre del archivo F2, nombre del archivo F2, longitud de los bloques a sincronizar, *checksums* de los bloques del archivo F1. Además, cuenta con una función que recibe la respuesta del servidor y actualiza el archivo F1 mediante la creación de un nuevo archivo local. Por último, se debe llamar a la función de destrucción para cerrar el `socket_t` cliente y terminar con la comunicación.
En la estructura del TDA sólo se guardan los datos que serán reutilizados en alguna instancia de la sincronización.
Cabe aclarar que tanto en el cálculo de *checksums* como en la actualización del archivo local viejo, se procesa información y se envía/recibe constantemente, no se procesan todos los datos y luego se envían por completo. Se prioriza así espacio antes que performance.

2.5.2. RSync remote

El TDA `rs_remote_t` contiene primitivas para iniciar la comunicación con el único cliente y recibir los datos del archivo F2 y F1 que envía el mismo.
Además, cuenta con una función que compara los *checksums* recibidos del cliente con los calculados del contenido del archivo F2. Como indica el enunciado, se lee un bloque del archivo, se calcula su *checksum* y se lo busca en la lista del cliente. Si el *checksum* no está, se guarda el primer byte de ese bloque en un *buffer* y se continúa leyendo moviendo el puntero de a uno (Para ello, se deberá retroceder el puntero `block_size - 1` lugares). Si el *checksum* está, se envían las diferencias encontradas hasta el momento y el número de bloque coincidente. Hay un caso especial cuando no se encuentra en *checksum* y se llega al final del archivo en el cual se agrega al *buffer* todo el bloque leído.
Por último, se debe llamar a la función de destrucción para cerrar los `socket_t` aceptor y servidor y liberar la memoria del *heap* reservada para el nombre del archivo remoto a sincronizar F2.

2.6. main

Se implemento una función `main` simple cuyo flujo esta dividido dependiendo del modo en el cual se inicia el programa (cliente o servidor).
Como pide el enunciado, se realiza un simple manejo de errores sobre los argumentos recibidos al ejecutar. Se verifica que la cantidad de parámetros sea la mínima, que el modo sea válido y, finalmente, que la cantidad de parámetros sea la correcta para el modo correspondiente. En caso de error se devuelve 1, si hay éxito, se devuelve 0. Para ello se utilizan las constantes `EXIT_FAILURE` y `EXIT_SUCCESS`, las cuales corresponden a estos valores.
Al finalizar se cierran los *file descriptors* `stdin`, `stdout` y `stderr` para que la corrida con Valgrind sea exitosa.

3. Diagramas

3.1. Tipos de datos abstractos

A continuación se presenta un diagrama que representan los tipos de datos abstractos utilizados y las relaciones entre ellos. Se omitieron las primitivas para facilitar su lectura, las mismas pueden encontrarse en los respectivos archivos de cabecera.

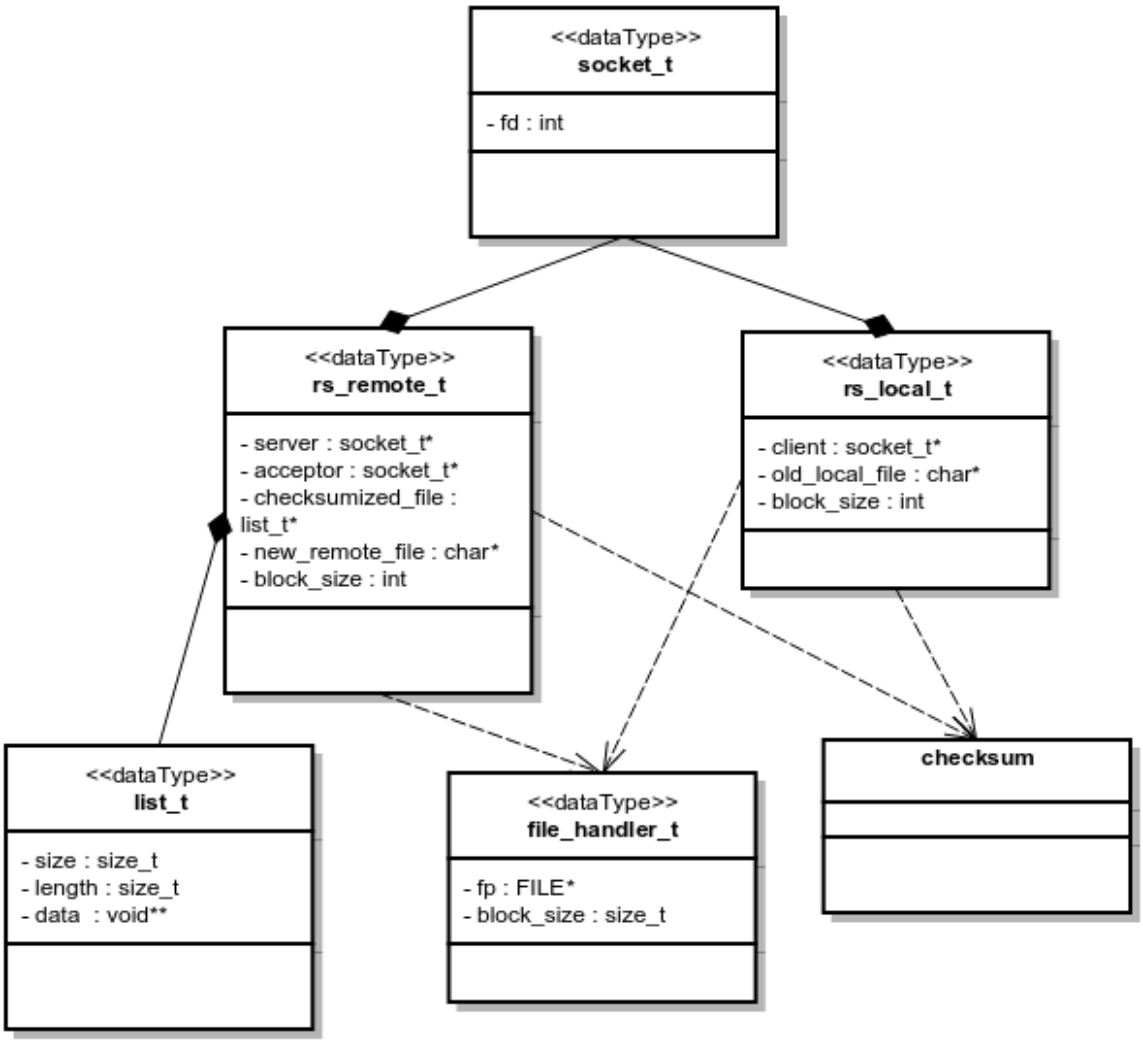

Figura 1: Diagrama de tipos abstractos de datos y sus relaciones

3.2. Secuencias

Los siguientes diagramas de secuencia ilustran la lógica de las funciones más complejas del programa; `remote_file_cmp()` y `local_file_update()`. Los mismos se encuentran simplificados mostrando el contenido de funciones auxiliares y no su llamado.

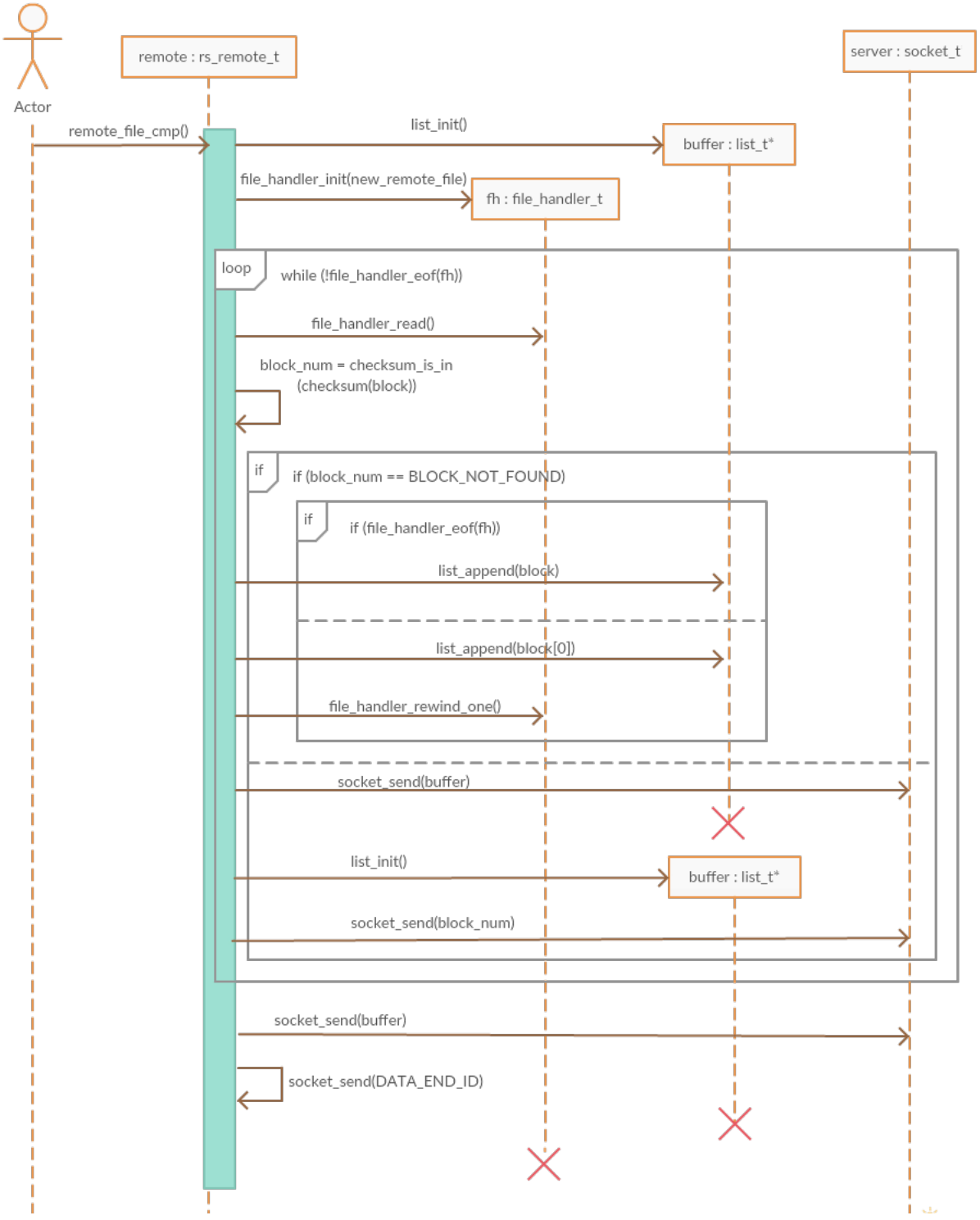

Figura 2: Diagrama de secuencia de la función `remote_file_cmp()`

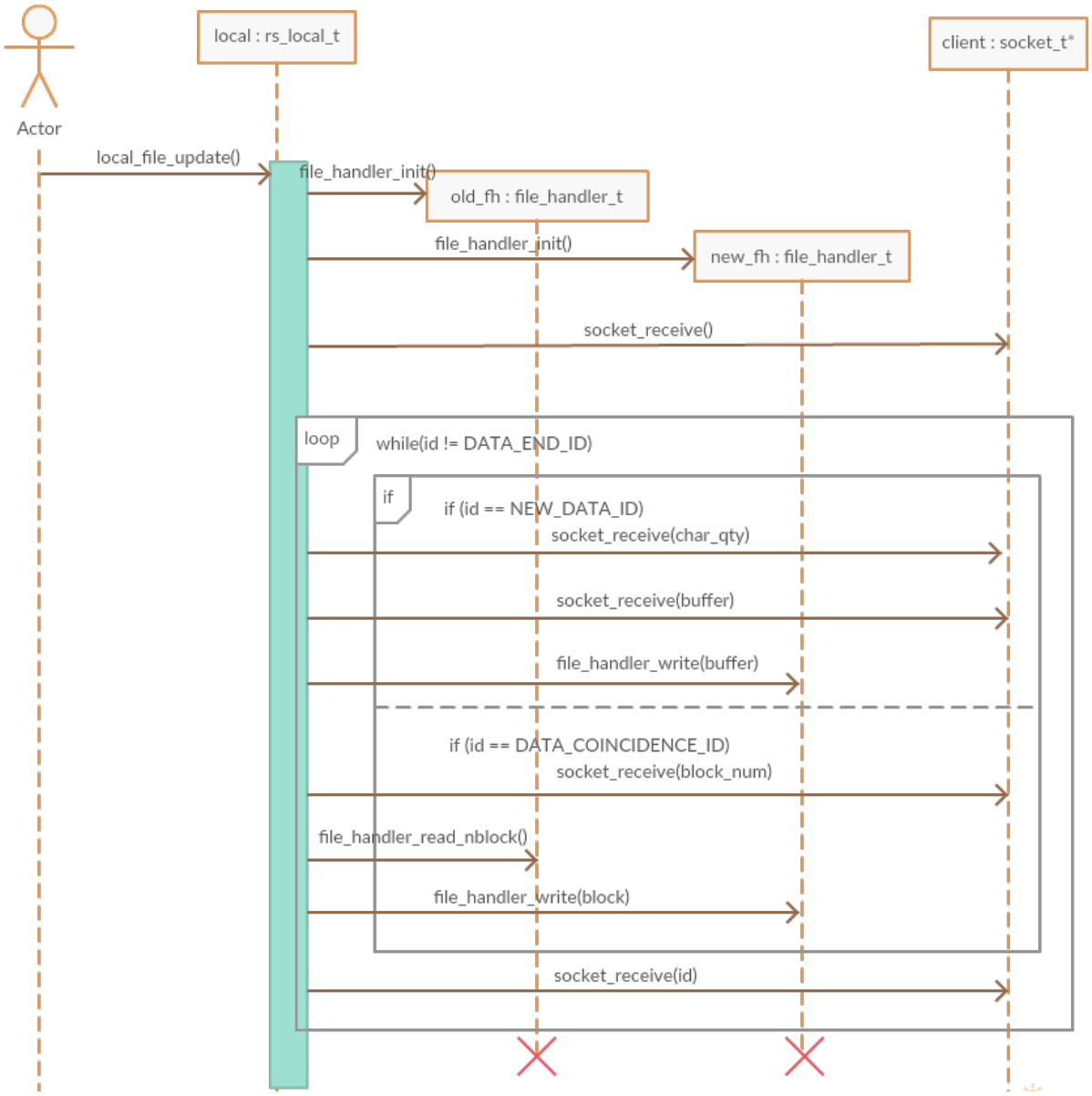

Figura 3: Diagrama de secuencia de la función `local_file_update()`