

Proyecto final: *Miner Rush*

Autores:

Kevin de la Coba Malam

Marcos Aarón Bernuy

Grupo 2292, pareja 04.

Contenido

Introducción	3
Trabajadores.....	3
Rondas de minado	4
Bloques.....	4
Red	4
Bloques compartidos	5
Concurrencia y semáforos	5
Votación	6
Monitor.....	7
Problemas y errores	8
Ejecución.....	9
Autoevaluación	11

Introducción

El proyecto emula el funcionamiento de la blockchain en la red. La blockchain no es una estructura simple, es muy compleja por lo que emularla supone un reto. Para emularla de manera local se usa memoria compartida, señales, mensajes, tuberías... todo lo aprendido durante el curso.

Invitamos a consultar el código ya que hemos puesto un gran esfuerzo comentarlo de forma útil y abundante. En la votación hay un pequeño “esquema” por el cual si se sigue el orden de los comentarios se podrá entender la forma en la cuál hemos implementado la votación.

Nuestra implementación cumple con todos los requisitos exceptuando el que implica una salida correcta de los mineros cuando uno recibe SIGINT se produce **interbloqueo**. Se ha intentado mitigar el efecto del interbloqueo, pero esto se explicará más adelante.

Trabajadores

El primer paso en el proyecto es el de implementar trabajadores dentro de un minero. Mediante la librería <pthread.h>, se crean un número determinado de threads que buscan la solución. El padre de estos threads (el minero) se queda a la espera de que estos terminen ejecutando la llamada de pthread_join.

Los trabajadores usan una estructura definida por nosotros, la estructura es la siguiente:

```
typedef struct {  
    int starting_index;  
    int ending_index;  
    long int target;  
    long int solution;  
} worker_struct;
```

Para hacer la ejecución de los trabajadores lo más eficiente posible la estructura contiene 2 índices:

- **starting_index** se refiere al índice desde el cual el trabajador debe buscar una solución.
- **ending_index** es el índice donde debe terminar de buscar la solución.

Por ejemplo: Digamos que el minero debe buscar la solución desde el número 0 hasta el número 999, para esta tarea también se especifican 4 trabajadores. Para paralelizar la tarea cada trabajador tendrá unos índices definidos, el primero tendrá desde 0 hasta 249, el segundo desde 250 hasta 499, el tercero desde 500 hasta 749, y el último desde 750 hasta 999. Esto paraleliza la ejecución de los trabajadores y hace que, a más trabajadores, más veloz el proceso de minado (siempre y cuando la CPU permita el número especificado de threads y tenga varios núcleos).

El campo **target** es común en todos los trabajadores ya que es el target que deben buscar, y el campo **solution** solo es modificado por el trabajador que encuentra la solución.

Una vez un trabajador encuentra una solución puede que los demás sigan buscando, para hacer que dejen de buscar se usa una variable global llamada **solution_find**. Inicialmente se establece a 0 de modo que cada vez que un trabajador itere buscando una solución comprueba el valor de esta variable, si el valor es diferente de 0 significa que un trabajador ya ha encontrado una solución y por lo tanto debemos salir del bucle. Esta variable es global para todos los threads, e incluso el minero puede acceder a esta ya que está declarada al comienzo del código como **extern solution_find**.

Para modularizar el código, todas las acciones relacionadas con los trabajadores están declaradas en los archivos **trabajador.c** y **trabajador.h**, también se incluye la declaración de la estructura.

Rondas de minado

El segundo argumento que el minero recibe es el número de rondas de minado a ejecutar, si es 0 o menor se ejecuta indefinidamente. Para esto se crea una variable llamada **infinite** la cual, en caso de que el segundo argumento cumpla las ejecuciones especificadas, se pondrá a 1, obligando al loop a nunca acabar.

Bloques

El siguiente paso en la implementación fue hacer uso de los **bloques**, la estructura en este caso ya se nos entregó hecha y no se hicieron modificaciones al respecto.

El manejo de los bloques se hace de forma dinámica, aunque se puede hacer de forma estática también (y ahorraría más de un dolor de cabeza). Para hacer un uso correcto y eficiente de los bloques se han definido varias **primitivas** las cuales se pueden consultar en **block.h**, ahí se encontrarán todas las primitivas.

El minero lo primero que hace es reservar memoria para un bloque, este bloque no tiene definidos ninguno de los campos necesarios. Como todavía no se ha implementado la red el bloque simplemente establece un target aleatorio entre los números 1-1.000.000. Se hace uso de una variable **last_block** la cuál es un puntero que simplemente se usa para definir el bloque anterior al creado, como en el primer bloque este es null, no tiene gran relevancia. Digamos que ahora estamos con el segundo bloque, *last_block* en este caso apuntará al bloque creado en la anterior iteración y al llamar a la función `int block_set(Block *prev, Block *block)` el bloque nuevo se actualiza y el id es el id del anterior +1, las wallets se copian, el campo *prev* del nuevo bloque apunta a *last_block*, el campo *next* del *last_block* apunta al nuevo bloque y por último el nuevo target es la solución anterior.

Una vez el minero ha inicializado el bloque, este copia el target en la estructura usada por los threads. Una vez se encuentra la solución solo haría falta actualizar la solución del último bloque, cogemos la solución de la estructura del thread y la copiamos en el último bloque.

Para mantener la modularidad todo lo relacionado con el manejo y uso de bloques está en los archivos **block.c** y **block.h**. En este momento de la implementación no se usan bloques compartidos entre procesos, es todo de forma individual.

Red

La red de mineros es la que permite que los mineros estén “conectados” entre ellos. Para el desarrollo de la red se ha usado la estructura ya proporcionada NetData. En una primera implementación la red solo conecta a los mineros, no significa que usen el mismo bloque todos y se ejecuten al mismo ritmo.

Para crear la red se usa memoria compartida, el primer minero en ejecutarse la crea, los demás se van uniendo a la red modificando `pid_t miners_pid[MAX_MINERS]; int last_miner; int total_miners;`. Estos campos también se modifican al salir de la memoria dinámica de forma que cuando el último minero salga dejara la variable **total_miners** a 0, cuando llega a este valor debemos hacer `unlink`.

Para mantener la modularidad todo lo relacionado con el manejo y uso de la red de mineros está en los archivos **net.c** y **net.h**. La red en este momento no resulta muy útil pero más adelante se implementa la red de la forma esperada.

Bloques compartidos

Tras haber implementado la red ya podemos usar bloques de forma compartida. Esto se implementa en los archivos **block.c** y **block.h**. El primer paso para hacer los bloques compartidos es el de definir una estructura con la memoria compartida necesaria:

```
typedef struct {  
    long int target;  
    long int solution;  
    int id;  
    int is_valid;  
    int num_miners;  
    int wallets[MAX_MINERS];  
} shared_block_info;
```

Esta es la estructura usada para definir el bloque compartido entre los procesos. Como se puede apreciar contiene todos los campos de un bloque exceptuando los campos *next* y *prev*, a parte se le añade el campo **num_miners** que nos sirve para saber cuántos procesos se encuentran usando esta región de memoria compartida, al igual que se hace en la red de mineros con la variable **total_miners**.

En esta parte de la implementación los mineros cuando empiezan una nueva ronda cargan el target de la memoria compartida. Como todavía no hay mecanismos de concurrencia se producen en ocasiones condiciones de carrera.

Concurrencia y semáforos

```
typedef struct {  
    int total_miners;  
    int blocked_loosers;  
    sem_t net_mutex;  
    sem_t block_mutex;  
    sem_t mutex;  
    sem_t vote;  
    sem_t count_votes;  
    sem_t update_blocks;  
    sem_t update_target;  
    sem_t finish;  
} Sems;
```

Esta es la estructura usada para definir los semáforos que emplearemos, además de dos campos que serán necesarios. Esta estructura y los semáforos definidos en ella serán necesarios para evitar problemas con la concurrencia que se daba al acceder a la memoria compartida de tanto bloques como la de la red de los mineros. Estos semáforos son semáforos sin nombre, decidimos usar estos ya que simplificaba la implementación (no hay que hacer close y unlink, simplemente destroy). En un principio únicamente teníamos *total_miners*, *net_mutex*, *block_mutex*, y *mutex* los cuales son semáforos de exclusión mutua. Más adelante implementaremos el resto que serán necesarios para programar de forma correcta la votación.

Esta estructura y su implementación junto con la de las funciones que modifican cada elemento de la estructura está en los ficheros **sem.c** y **sem.h** en los cuales se inicializan todos los semáforos anteriormente mencionados y también serán eliminados cuando sea necesario empleando las respectivas funciones. En ellos además para evitar la repetición de código implementamos

sem_down() y *sem_up()* funciones las cuales reducen el tener que realizar un control de errores cada vez que se emplee *sem_wait()* y *sem_post()* en otros ficheros, a parte en *sem_down()* protegemos a la llamada *sem_wait()* de posibles errores si se reciben señales.

Ahora cada vez que se quiere acceder a la región de memoria compartida de la red se usa *net_mutex* para que haya únicamente un proceso leyendo o escribiendo en la memoria, lo mismo pasa con la región de memoria para el bloque compartido, usamos *block_mutex*, y por último tenemos un último *mutex* destinado a proteger las propias variables de la estructura de los semáforos.

Ya que los semáforos se cargan en memoria compartida, los protegemos de la misma forma que protegemos las otras regiones de memoria compartida, tenemos la variable **total_miners** que cuenta el número de procesos que usan esta región para después cerrar correctamente la región y los semáforos.

En esta parte los mineros estaban sincronizados entre ellos y todos tenían un mismo target, actualizaban después la solución, etc. La concurrencia estaba solventada.

Votación

En esta sección en la que implementamos todos los aspectos relacionados con la votación finalmente implementamos el resto de la estructura de Sem (creamos los semáforos necesarios).

El proceso de votación comienza cuando se detecta que hay un ganador, es decir, cuando un minero ha encontrado solución a la operación planteada. Nada más ver esto comprobamos si más de un minero ha encontrado solución y cambiamos el valor *solution* a **0** de la memoria compartida de los bloques. Al modificar la variable si otro minero acaba al mismo tiempo, solo uno podrá cambiar el valor de la variable (el primero que llegue a esa zona ya que está protegida por semáforos). Una vez modificada la variable los demás mineros que acaben se suspenderán ya que verán que alguien ha modificado el valor antes. Los perdedores se quedarán esperando a SIGUSR2.

A partir de aquí los mineros seguirán dos caminos, el de ganador, o el de perdedor.

En el caso de que se trate de un minero **ganador**, lo primero de todo será actualizar la solución. Tras esto, recibirá el quorum, es decir el número de mineros que participan en la votación y que se encuentran activos, este valor será guardado en la memoria compartida de red, en concreto en la variable *total_miners*.

Después el ganador envía la señal SIGUSR2 a todos los procesos registrados en la red usando la función **send_SIGUSR2** y empezaría la votación. Para la votación se realizan *ups* del semáforo **vote**, desbloqueando a los perdedores bloqueados. Se llama a la función *sem_up* tantas veces como procesos haya activos (usando como referencia la variable *quorum*). El ganador pasa a bloquearse con el semáforo **count_votes**.

Una vez la votación ha sido realizada con éxito se cuentan los votos empleando el **voting_pool** de la memoria compartida de red (donde los perdedores han escrito). Cuando hemos contado los votos debemos comprobar que el bloque sea válido, para lo que tiene que haber más de la mitad de los votos positivos teniendo en cuenta el quorum. En caso de que sea así, actualizamos los campos necesarios, y en caso de que no lo sea, se destruye el bloque el bloque local, dejando el target igual a como estaba antes.

Cuando ya se ha hecho la comprobación se reinicia la votación para futuras votaciones, poniendo todos los valores de la *voting_pool* a **-1**. Tras esto tenemos que permitir una vez más que cada minero actualice su bloque, para lo que bloquearemos el proceso ganador hasta que se hayan actualizado como previamente se ha explicado, pero en este caso se emplearán los semáforos

update_target y **update_block**. Por último, antes de liberar el proceso de votación y liberar a los votantes, actualizamos el bloque compartido introduciendo como nuevo target la solución del anterior. Tras esto liberamos los recursos como ya estaba implementado y finaliza el proceso de votación.

En el caso del minero **perdedor**, lo primero que hacen es reservar memoria para un bloque (este será necesario en el futuro), posteriormente nos bloqueamos en el semáforo *vote* a la espera de que el ganador nos desbloquee.

Una vez nos desbloquea, comprobamos si la solución propuesta es correcta o no, después introducimos nuestro voto en la *voting_pool* y comprobamos si somos el último en votar contando cuantos mineros han votado, si hay tantos votos como mineros-1 en la red, es que somos el último en votar, por lo que tenemos que desbloquear al ganador. Los perdedores se bloquean con **count_votes** esperando que el ganador cuente los votos.

Una vez se cuentan los votos es hora de que los mineros actualicen su bloque (el que se allocó nada más recibir SIGUSR2) y tras actualizarlo nos bloqueamos en el semáforo **finish** esperando a que el ganador comience una nueva ronda actualizando la memoria compartida del bloque.

Queremos mencionar que la ejecución de la votación en el caso de los perdedores se hace dentro de el manejador de la señal SIGUSR2. Sabemos que se recomienda que se usen manejadores lo más cortos posibles, pero el hacerlo corto nos dio muchos problemas y decidimos meter esa parte de la ejecución ahí.

Monitor

Para la implementación del monitor en dónde también configuramos la cola de mensajes y realizamos la impresión de los resultados en *blockchain.log*, hemos creado los ficheros **monitor.c** y **monitor.h**. En ellos se encuentran las funciones y procesos necesarios para la monitorización de los mineros.

A parte los mineros nada más terminar su ronda, sean ganadores o perdedores, envían un mensaje al monitor con su bloque actualizado.

El monitor lo implementamos de la siguiente manera. Creamos un proceso padre e hijo empleando la función `fork()`. El padre y el hijo estarán comunicados por un pipeline, por lo que el primer paso del proceso hijo será cerrar el extremo de escritura y el del proceso padre cerrar el extremo de lectura. Tras esto se abrirá el fichero *blockchain.log* para la escritura de todos los bloques en este cada 5 segundos en el proceso hijo. Para escribir cada 5 segundos se usa SIGALRM y un manejador.

De forma paralela el proceso padre se encarga de inicializar los semáforos, unir el monitor a la red e inicializar el buffer circular de bloques. También se encargará de crear la cola de mensajes.

Una vez hemos realizado todos estos pasos previos podemos centrarnos en la gestión del bloque tanto para proceso padre como para proceso hijo.

En la gestión del proceso **padre**, este se bloqueará (o no, si hay un bloque dentro de la cola) esperando la recepción de un bloque. Una vez recibido, comprueba si el bloque se encuentra ya en el buffer con la variable y si está modifica la variable **is_in** (es tratada como un boolean). En el caso de que en efecto si este, se realiza la operación y se imprime el mensaje correspondiente a la operación. En cambio, si no se encuentra, metemos el bloque en el buffer. Posteriormente realizamos una copia del bloque y la escribimos en el pipeline para que llegue al proceso hijo, pero únicamente si no se encuentra en el buffer.

De forma paralela el proceso **hijo** se había quedado bloqueado esperando a recibir bloque por el pipeline y poder hacer la lectura correcta de este. Una vez lo ha leído correctamente, simplemente realiza una copia y lo guarda en la cadena dinámica que este guarda.

Para cerrar de manera correcta, el padre cuando recibe SIGINT envía SIGINT al hijo y espera a que este acabe, cuando este acaba, el padre libera todos sus recursos.

Problemas y errores

Hemos tenido 3 problemas:

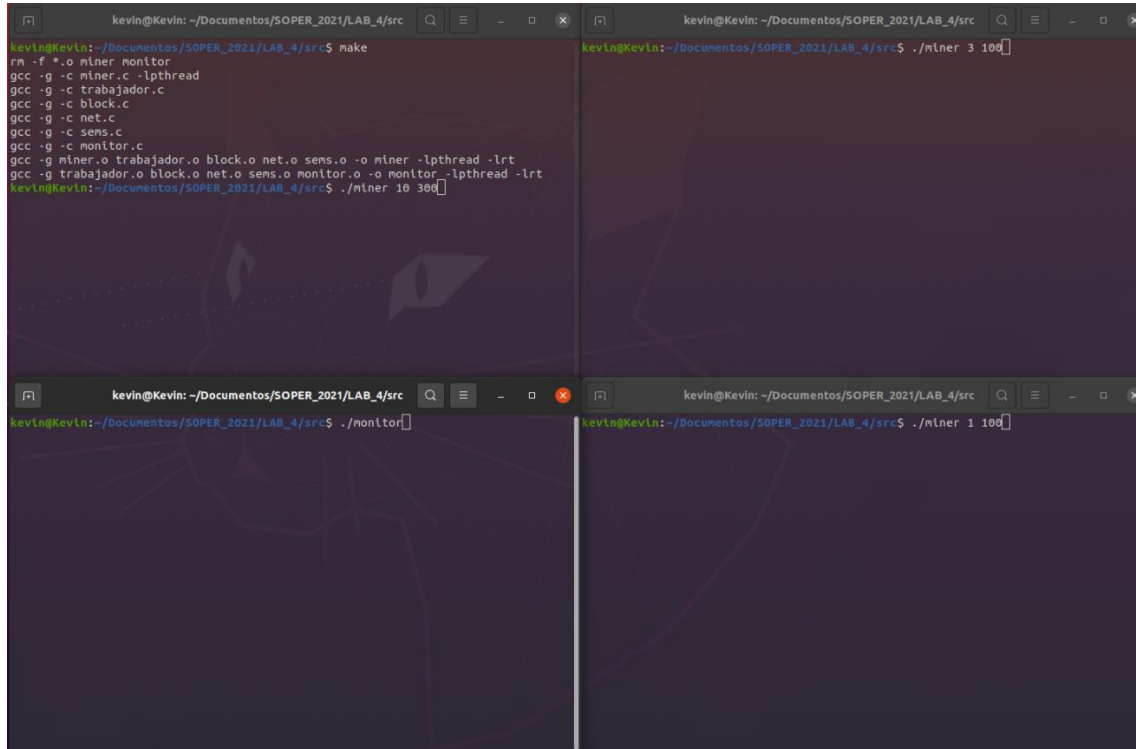
1. SIGUSR2 (solucionado). Cuando implementamos por primera vez el proceso de votación, este se encontraba en el **main**, pero descubrimos que se producían interbloqueos ya que los procesos antes de meterse a hacer la votación comprobaban si habían recibido la señal SIGINT, en cuyo caso terminaban su ejecución. Si terminan su ejecución puede darse el caso de que mientras lo hacen reciban SIGUSR1, por lo que se tendría en cuenta en el quorum a ese proceso que ya no esta y el ganador se quedaría bloqueado.
2. A raíz del problema anterior el ganador se quedaría bloqueado. Quisimos mitigar este problema usando `sem_timedwait` de forma que si pasan 2 segundos el ganador se desbloquea y sigue.
3. El núcleo de todos nuestros problemas es SIGINT, no somos capaces de implementar el requisito que especifica que si un proceso recibe SIGINT, este salga sin que los demás lo noten.

A parte del problema con SIGINT y los mencionados, no hemos detectado ningún otro problema. Los mineros y la red se ejecutan de forma correcta siempre y cuando no se trate de cerrar un minero con SIGINT.

Ejecución

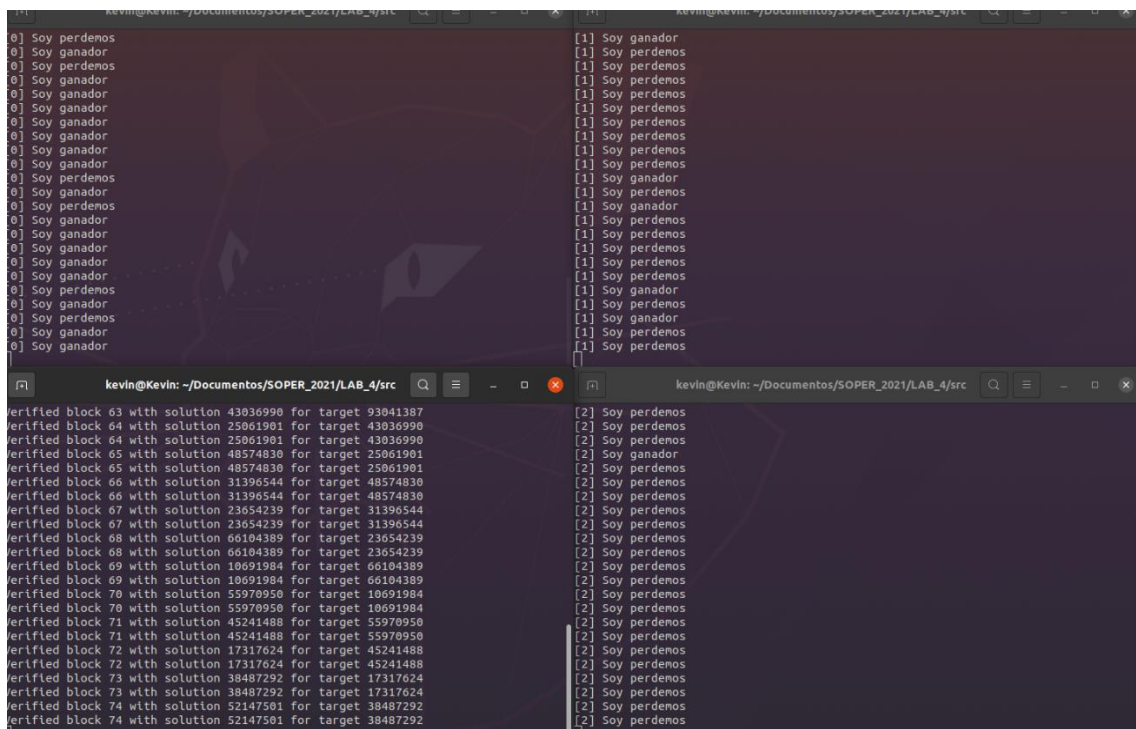
Para ejecutar el programa basta con compilar el código poniendo **make** y posteriormente ejecutar mineros en distintas terminales y ejecutar el monitor en otra. El monitor debe ejecutarse siempre después de un minero, ya que el monitor se **une** a la red, no la crea.

A continuación, se va a hacer una prueba de ejecución donde hay 3 mineros y 1 monitor:



The image shows four terminal windows arranged in a 2x2 grid. The top-left window shows the compilation process using 'make', listing source files (miner.c, trabajador.c, block.c, net.c, sens.c, monitor.c) and the resulting object files and executables. The top-right window shows the execution of a miner: './miner 3 100'. The bottom-left window shows the execution of the monitor: './monitor'. The bottom-right window shows the execution of another miner: './miner 1 100'.

Podemos ver lo primero el índice del minero en la red (en *miners_pid*), después si es ganador o perdedor. En el monitor vemos los bloques que recibe este y si los verifica o no.



The image shows two terminal windows. The left window shows the output of a miner, displaying a list of 'Soy ganador' (I am a winner) and 'Soy perdenos' (I am a loser) messages. The right window shows the output of the monitor, displaying a list of 'Soy ganador' and 'Soy perdenos' messages. Below these, the monitor shows a list of blocks being verified, including block numbers, solutions, and targets.

Si prestamos atención a la salida anterior podemos ver que el minero de la esquina izquierda superior gana más que los otros dos, pero el minero de la esquina derecha inferior pierde más. Esto es debido a que antes de la ejecución se han configurado los mineros con un número diferente de trabajadores, por lo que se puede ver que el minero con más trabajadores gana más veces, y el que tiene menos trabajadores gana menos veces.

Al finalizar la ejecución se crea un archivo *blockchain.log* en el cual tenemos impresa la blockchain entera recibida por el monitor. El monitor imprime lo que tiene cada 5 segundos por lo que el puede haber más de una impresión de la blockchain en el mismo archivo.

BLOCK 226:

is_valid: 1

target: 26366500

solution: 53550853

Wallets:

0: 201 | 1: 24 | 2: 1 |

BLOCK 227:

is_valid: 1

target: 53550853

solution: 16608806

Wallets:

0: 202 | 1: 24 | 2: 1 |

BLOCK 228:

is_valid: 1

target: 16608806

solution: 95013860

Wallets:

0: 203 | 1: 24 | 2: 1 |

Autoevaluación

Hemos tenido debate entre nosotros para decidir cuál sería una autoevaluación correcta. Teniendo en cuenta que se ha implementado todo lo relacionado con la votación correctamente estaríamos en un 9 mínimo y si estuviera perfecto podríamos optar al 10, pero aquí empieza lo dudoso. El problema de implementación tiene que ver con los mineros y por tanto se puede considerar como un error que debería haberse solucionado, aunque es cierto que se trata del punto 15 de mineros por tanto depende mucho de la relevancia dada al error. Junto a esto debemos añadir posibles errores de implementación que no hayamos resuelto de la forma más eficiente. A parte de esto, creemos haber mantenido la modularidad, y no haber cometido fallos que se consideren penalizaciones. En definitiva, partiendo de un 10 reducimos por el error de SIGINT, además de posibles pérdidas de eficiencia y tenemos dos notas propuestas por cada uno de nosotros y la media de ambas resulta en un 9 (uno propone 9.5 y el otro propone 8.5) sin contar con la posibilidad de aumentar dicha nota por una mejora que puede ser tanto por la autoevaluación realizada, como por que el programa muestre un rendimiento excelente.