TP2 - Algoritmos en sistemas distribuidos

Sistemas Operativos - Primer cuatrimestre de 2019

Límite de entrega: Sábado 01 de junio, 23:59 hs.

1. Introducción

A través de este trabajo, se busca consolidar el conocimiento de los sistemas distribuidos, y en particular el envío de mensajes. La implementación se hará en el lenguaje C++ utilizando la interfaz MPI [1].

Deberán implementar un cómputo de forma distribuida, que será realizado por un conjunto de $n \geq 1$ procesos, que se podrán ejecutar en $m \geq 1$ máquinas. A estos procesos los llamaremos **nodos**.

2. Problema a resolver

Se pide desarrollar una cadena de bloques (*blockchain*), concepto que tomó creciente popularidad desde el surgimiento de Bitcoin [2].

Una *blockchain* no es más que una cadena distribuida de bloques enlazados, diseñada para evitar su modificación mediante un proceso de consenso.

2.1. Bloque

Empezaremos definiendo la estructura de un bloque (`Block`) que se encuentra implementada en los archivos `block.h` y `block.cpp`.

- `unsigned int index`: Índice del bloque en la cadena **(empieza en 1)**.
- `unsigned int node_owner_number`: Número del nodo que creó el bloque.
- `unsigned int difficulty`: Dificultad pedida para el bloque (definido más adelante)
- `unsigned long int created_at`: Fecha de creación en formato POSIX [1].
- `char nonce[NONCE_SIZE]`: String para resolver el *proof-of-work* (definido más adelante).
- `char previous_block_hash[HASH_SIZE]`: Hash del bloque anterior (en formato SHA256).
- `char block_hash[HASH_SIZE]`: Hash del bloque (en formato SHA256).

Para facilitar la implementación, se encuentra definido un nuevo tipo de datos de MPI: `MPI_Datatype* MPI_BLOCK`. Tener cuidado que tendrán acceso a la estructura de MPI definida mediante un puntero a `MPI_BLOCK`, por lo tanto a la hora de usar la estructura en las funciones de MPI deberán referenciarlo con `*MPI_BLOCK`.

[1] https://es.wikipedia.org/wiki/Problema_del_ano_2038

2.2. Blockchain

El primer bloque de la cadena será un bloque distinguido con índice 1 y no tendrá bloque anterior (el hash del bloque anterior será 0).

El objetivo de cada nodo será que en la *blockchain* existan la mayor cantidad posibles de bloques que hayan sido creados por el propio nodo.

Para agregar un nodo deberán *minarlo*, es decir, superar una prueba que requiere un cierto costo de cómputo (*POW: Proof-Of-Work*). La prueba es variar el campo `nonce` del bloque hasta que la representación en binario del hash del nodo empiece con, al menos, una cierta cantidad de ceros definida en la constante `DEFAULT_DIFFICULTY`.

El código base de los nodos se encuentra en `node.cpp`. La función de entrada es `int node()` que debe devolver -1 si hubo un error que impida continuar la ejecución, o 0 en caso contrario.

Cada nodo deberá contar con un hilo (*thread*) que trate de minar un nuevo bloque mediante la función `void* proof_of_work(void *ptr)` y transmitirlo a los otros nodos, y con otro hilo que esperará mensajes de los otros nodos (para, por ejemplo, saber si otro nodo ha minado un bloque).

A su vez, cada nodo mantendrá un diccionario (`map<string,Block> node_blocks`) con los bloques minados por él o comunicados a él por los otros nodos donde la clave de cada bloque será su propio hash. Y, por otro lado, un puntero al último bloque de la cadena (`Block * last_block_in_chain`) para poder recorrerla.

3. Consenso

Al recibir un mensaje de otro nodo con un nuevo bloque (`tag = TAG_NEW_BLOCK`), el nodo receptor deberá decidir si lo agrega a la cadena o no.

Es de interés de cada nodo, aceptar en su *blockchain* a los bloques que vayan a ser aceptados por los demás nodos. Esto se debe a que -para minar un nuevo bloque- es necesario invertir cómputo en un proceso que depende de la información del último bloque (ya que el hash depende también del valor del campo `previous_block_hash`). A esta necesidad de apostar por la cadena de consenso, se lo conoce como *Consenso de Nakamoto*.

Por eso, cada nodo debe realizar las siguientes acciones al recibir un mensaje de que un nuevo bloque ha sido minado

- Si el nodo no es válido (su tiempo de creación fue 5 minutos atrás de cuando lo recibí o el hash es inválido), entonces omitirlo. Esta parte ya viene resuelta en la llamada a la función `valid_new_block(rBlock)`.
- Si el índice del bloque recibido es 1 y mi último bloque actual tiene índice 0, quiere decir que todavía no tengo un bloque inicial válido, así que lo agrego como nuevo último (y primero válido).
- Si el índice del bloque recibido es el siguiente a mi último bloque actual, y el bloque anterior apuntado por el recibido es mi último actual, entonces lo agrego como nuevo último.
- Si el índice del bloque recibido es el siguiente a mi último bloque actual, pero el bloque anterior apuntado por el recibido no es mi último actual, entonces hay una blockchain

más larga que la mía. Me conviene abandonar la mía actual y llamar a la función `bool verificar_y_migrar_cadena(const Block *rBlock, const MPI_Status *status)`.

- Si el índice del bloque recibido es igual al índice de mi último bloque actual, entonces hay dos posibles *forks* de la *blockchain*, pero mantengo la apuesta por la mía.
- Si el índice del bloque recibido está más de una posición adelantada a mi último bloque actual, entonces me conviene abandonar mi blockchain actual y llamar a la función `bool verificar_y_migrar_cadena(const Block *rBlock, const MPI_Status *status)`.

Cada nodo debe terminar su ejecución al aceptar una cadena de longitud igual a la constante `MAX_BLOCKS` definido en el archivo nodo.h, que se puede modificar según se considere conveniente conveniente.

4. Verificar y migrar de cadena

Como mencionamos en la sección anterior, cuando el nodo (*alice*) analice el bloque enviado por otro nodo (*bob*), puede ser necesario migrar a la *blockchain* del nodo *bob*.

Para eso *alice* enviará un mensaje con el tag `TAG_CHAIN_HASH` a *bob* para que nos envíe los bloques anteriores al recibido y poder completar la cadena. Este mensaje contendrá el hash del bloque a partir del cuál necesitamos los bloques anteriores.

Cuando *bob* reciba el mensaje `TAG_CHAIN_HASH`, responderá con una lista de bloques anteriores a ese hash. Dicha lista nunca deberá superar la cantidad de bloques definida en `VALIDATION_BLOCKS` por seguridad, y deberá ser enviada con el tag `TAG_CHAIN_RESPONSE`.

Cuando *alice* reciba el mensaje `TAG_CHAIN_RESPONSE` de *bob* procederá a verificar que:

- El primer bloque de la lista contiene el hash pedido y el mismo index que el bloque original.
- El hash del bloque recibido es igual al calculado por la función `block_to_hash`.
- Cada bloque siguiente de la lista, contiene el hash definido en `previous_block_hash` del actual elemento.
- Cada bloque siguiente de la lista, contiene el índice anterior al actual elemento.

Si dentro de los bloques recibidos por *alice* alguno ya estaba dentro de `node_blocks` (o el último tiene índice 1), entonces ya puedo reconstruir la cadena. Agrego todos los bloques anteriores a `node_blocks` y marco el primero como el nuevo último bloque de la cadena (`last_block_in_chain`). De lo contrario, descarto la cadena y los nuevos bloques por seguridad.

5. Ejercicios

1. Completar la función `broadcast_block` para que comunique a todos los demás nodos el nuevo bloque creado con el tag `TAG_NEW_BLOCK`. Cada nodo debe enviar los mensajes en un orden distinto a los demás nodos.
2. Modificar el método `nodo` para que cree un nuevo thread que mine bloques mediante la función `proof_of_work`. Modificar las funciones de manera tal que entre los dos hilos del

proceso no se produzcan condiciones de carrera. Además, mientras se envíe la información de un bloque recién creado a los demás, no se deben procesar los mensajes de nuevos bloques minados por otros.

3. Completar la función `validate_block_for_chain` para que respete las reglas de consenso descritas y no tenga condiciones de carrera con el thread que mina nuevos bloques.

4. Completar la función `verificar_y_migrar_cadena` para que respete el protocolo descrito en la sección correspondiente.

5. **Importante:** Desarrolle un análisis del protocolo descrito en este trabajo que responda, al menos, a las siguientes preguntas:

   - ¿Puede este protocolo producir dos o más blockchains que nunca converjan?
   - ¿Cómo afecta la demora o la pérdida en la entrega de paquetes al protocolo?
   - ¿Cómo afecta el aumento o la disminución de la dificultad del Proof-of-Work a los conflictos entre nodos y a la convergencia? Pruebe variando la constante `DEFAULT_DIFFICULTY` para adquirir una intuición.

6. Ejecución

El sistema se ejecutará indicando por parámetro cuántos procesos se utilizarán.

Para ejecutar el programa compilado debe usarse `mpiexec -np N ./blockchain`, donde `N` es la cantidad de procesos a lanzar.

7. Requerimientos

Para el desarrollo puede usarse MPICH2 [4], OpenMPI [5] o cualquier otra implementación del estándar MPI-2 [1] en versión no-obsoleta.

8. Entregable

Para entregar el TP2, envien un mail a la lista de docentes: so-doc@dc.uba.ar

Con la siguiente información:

únicamente:

- El código fuente debidamente documentado, **con Makefile** (**NO** incluir ejecutables).
- El documento del informe (en **PDF**).

Referencias

[1] *MPI: A Message Passing Interface Standard.* http://www.mpi-forum.org/docs/docs.html

[2] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf

[3] MPI Tutorial @ LNLL (muy recomendable)
https://www.llnl.gov/computing/tutorials/mpi/

[4] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/

[5] OpenMPI, http://www.open-mpi.org/