Sistemas Operativos

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico II

Algoritmos en Sistemas Distribuidos

Integrante	LU	Correo electrónico
Alvarez Mon, Alicia	224/15	aliciaysuerte@gmail.com
Ansaldi, Nicolas	128/14	nansaldi611@gmail.com
Suárez, Romina	182/14	romi_de_munro@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El objetivo de este trabajo práctico es ganar conocimientos sobre los sistemas distribuidos mediante el envío de mensajes utilizando la interfaz MPI.

El concepto a desarrollar es una lista enlazada de bloques o blockchain, la cual tratará de ser minada por distintos nodos. Dichos nodos buscarán ser el dueño de la mayor parte de los bloques dentro de la lista y deberán llegar a un consenso sobre que bloques se encuentran en la misma. Nosotros utilizamos la implementación MPICH2 y la version 3.4.

SI NO HAY NODO QUE MINE UNA CADENA, LA CADENA EXISTE?

2. Desarrollo

En esta sección describiremos la implementación de nuestro código, explicando el funcionamiento del mismo y cómo las decisiones realizadas nos permiten respetar el objetivo especificado.

Cada nodo representa un proceso que se encarga de minar el siguiente bloque de una cadena. Cada uno de ellos existe en una máquina en la que no existe otro nodo, sin embargo la cadena es común para todos por lo que necesitan información acerca de la misma. Para solucionar el problema de compartir información utilizamos la interfaz MPI, necesaria para solucionar problemas de sincronización y coherencia característicos de este tipo de entornos. Además, para este caso en particular, deberemos respetar el protocolo, que nos indica que decisiones debemos tomar en los conflictos que se nos pueden presentar durante el cálculo de la cadena.

El código inicializa el entorno MPI para la cantidad de procesos pasados por parámetros, y luego llama a Node, que es la función que se encarga de manejar el funcionamiento del nodo.

2.1. Función Node

La función inicializa el nodo. Para esto le pregunta al MPI, que mantiene el canal global MPI_COMM_WORLD, cuál es su mpi_rank que vendría a ser su identificación con respecto a los otros nodos. Luego define un bloque inicial que, si bien no pertenece a la cadena, sirve para que el nodo tenga de donde comenzar a minar.

Luego, nuestro nodo se dedica a dos tareas durante el curso de nuestro programa. Por un lado, debe minar bloques para colocarlos en la cadena, siendo su objetivo conseguir agregar la mayor cantidad posible. Por otro lado debe escuchar los mensajes de los otros nodos que le avisan si ellos minaron un bloque, ya que nuestros nodos deben seguir una misma cadena, y se ponen de acuerdo seguir la cadena más larga. Recibiendo los mensajes de otros nodos, pueden decidir si deben quedarse en su cadena actual o cambiar a otra cadena.

Para poder realizar estas dos tareas en paralelo, la función nodo crea un thread "hijo", que intentará minar bloques en la función proof_of_work. Mientras que el thread "padreïntentará escuchar los mensajes recibidos.

A la hora de recibir mensajes usamos la función MPI_Recv que es bloqueante, esto lo hacemos porque queremos tomar acción inmediata acorde al mensaje recibido. Pero como podemos recibir distintos mensajes, como por ejemplo que otro nodo minó un bloque o algún nodo nos está pidiendo una lista de bloques porque se retrasó, usamos la función MPI_Iprobe para poder discriminar entre ellos. Esto lo hacemos porque si nos bloqueamos para recibir un mensaje no podemos recibir otro distinto al que estamos esperando. La razón para usar la función MPI_Iprobe, que es no bloqueante, frente a MPI_Probe es por temas de terminación de la función nodo.

Si el tag es de nuevo bloque (TAG_NEW_BLOCK) llamamos a la función que valida los nodos y, de ser necesario, esta llama a la función que migra de cadena.

Validar el bloque puede potencialmente modificar la cadena que cada nodo esta siguiendo. Por esta razón, no puede ocurrir que un nodo sea minado y enviado mientras valido otro nodo (podrían ser el mismo nodo). Estas dos operaciones modifican la variable last_block_in_chain y como son 2 hilos o procesos, se puede producir una race codition. Para evitar esto pusimos un mutex entre la validación del nuevo nodo y la parte de proof_of_work que se encarga de actualizar y enviar el bloque minado.

Para poder terminar incluimos un tag (TAG_FIN) para poder informar a los nodos que alguno de ellos alcanzó la cantidad máxima de bloques a minar. En este caso prendemos un flag para finalizar proof_of_work, cerramos los threads y terminamos.

Si el tag es de pedido de cadena, recibimos el nodo al cual debemos copiar parte de nuestra cadena y el hash del bloque a partir del cuál se requiere conocerla. Luego mandamos un arreglo de bloques de tamaño igual a VALIDATION_BLOCK, es decir, la cantidad máxima de posiciones que una cadena revisa para ver si existe un punto de convergencia con su cadena actual. En la primera posición del arreglo está el bloque que se corresponde con el hash que recibimos en primer momento. En los siguientes bloques del arreglo están los anteriores a este hasta el primer bloque de la cadena o un número igual a VALIDATION_BLOCKS. Para mandársela utilizamos MPI_Isend, ya que no queremos bloquearnos mientras mandamos la cadena.

2.2. Función Validate new block

Esta función se encarga de decidir que hacer a partir del bloque recibido, de manera de seguir el protocolo acordado.

Primero hace una comprobación de validez del nodo por seguridad, y en caso que este sea válido (el hash del block recibido es el mismo que la función de hash generaría con ese block, y el tiempo de creación no es anterior a cierto umbral de tiempo), lo agrega al diccionario de bloques del nodo (que tiene todos los nodos validos que le llegaron al nodo, estén o no en la cadena de éste)

Luego, ve en que caso nos encontramos:

1. Si el índice del bloque recibido es 1, y el índice de mi último bloque es 0, yo no mine ningún bloque aún, y la cadena del otro nodo es más larga, por lo que agrego ese nodo como mi último y vuelvo (no debo migrar cadena porque todavía no miné nada).

2. Si el índice recibido es el siguiente a mi último bloque y tenemos el mismo bloque anterior, el otro nodo minó un bloque más pero estamos siguiendo la misma cadena, por lo que actualizo mi último nodo a éste (modifico last_block_in_chain) y vuelvo.

3. Si el índice recibido es el siguiente al mio, pero no tenemos el mismo bloque anterior, significa que hay dos cadenas, y la del otro nodo es más larga que la mía, por lo que llamo a migrar cadena para pasarme a esa cadena con el bloque recibido.

4. Si el índice recibido es el mismo que el mio, hay dos cadenas, pero ninguna más larga que la otra, así que yo mantengo mi cadena y descarto ese bloque, ya que quiero priorizar que sea yo quien mine la mayor cantidad de nodos posibles, y si produzco el siguiente nodo antes que el otro, mi cadena será más larga y contará con más bloques creados por mi.

5. Si el índice recibido es menor al mio descarto el bloque porque la cadena que yo sigo es más larga.

6. Si el índice recibido es mayor al mio por más de uno, la cadena del otro nodo es más larga, por la que llamo a migrar cadena para cambiarme de cadena con el bloque recibido.

2.3. Función Verificar y migrar Cadena

Esta función cambia de cadena a la del bloque recibido, si los chequeos de seguridad terminan siendo válidos. Para ello, empieza con un MPI_Isend que le pide , al nodo que le mandó el bloque, la cadena de bloques anteriores. Usamos un send no bloqueante porque podría pasar el caso de que el nodo al cual le estamos enviando el mensaje, termine y nos quedemos esperando que le llegue nuestro mensaje. Luego, nos ponemos a esperar recibir un mensaje de vuelta. En esta circunstancia nos pueden llegar dos mensajes de otros nodos que nos interesan, un mensaje con TAG_CHAIN_RESPONSE del nodo al que le enviamos la solicitud con la cadena pedida, o un mensaje con TAG_FIN de cualquier nodo avisando que este terminó de minar el último nodo de la cadena. Entonces, entramos a un ciclo donde recibimos mensajes (con recv bloqueante) de cualquier tag y fuente, y solo procedemos con la función si el mensaje es una TAG_CHAIN_RESPONCE del nodo correspondiente.

1. En el caso de que nos llegue un TAG_FIN, seteamos una variable que señaliza que el otro thread (proof of workk) de nuestro nodo debe terminar, y que nuestra thead principal tiene que hacer join y terminar cuando validate_new_block deslockee el mutex, y salimos de la función. (que siempre es llamada por validate_por_block, y que la hace terminar)

2. Si nos llega un mensaje de nodo nuevo, o un TAG_CHAIN_RESPONSE de otro nodo que no sea el fuente (esto puede ocurrir si dos nodos esperan una cadena de un nodo) lo ignoramos.

3. Si nos llega un TAG_CHAIN_RESPONSE del nodo al que le pedimos la cadena, salimos del ciclo y procedemos a verificarla.

Primero verificamos que el primer nodo sea el mismo (en hash e índice) del que mandamos nosotros para pedir la cadena.

Luego, verificamos que la función block_to_hash genere el mismo hash que tenia el bloque original.

Para seguir, verificamos que los nodos enviados estén bien enlazados y con los índices en orden correspondiente. (osea, que se trate de una cadena valida). Como las cadenas de bloques son siempre de VALIDATION_BLOCKS, pero puede ser que la cadena enviada tenga menos posiciones, nos encargamos de sólo leer más bloques si no llegamos al principio de la cadena (el índice del bloque sea 1) .

Por último, nos fijamos que alguno esté dentro del diccionario o el último nodo válido de la blockchain tenga índice 1.

Si todo esto se cumple, agregamos todos los nodos al diccionario y actualizamos last block in chain como el primer nodo de la cadena enviada. Sino, descartamos la cadena por seguridad y nos quedamos con la nuestra.

2.4. Función Proof of work

Esta función se encarga del hilo de ejecución del bloque que mina nuevos bloques en la cadena. Mientras no termine empieza a crear un bloque con el índice siguiente al su último bloque, y si todavía no superó la longitud máxima de la cadena, busca un nonce y hash que cumplan la dificultad y se las asigna al nuevo bloque creado.

Al lograr esto, se dispone a broadcaestear y agregarlo como último de su cadena. Pero para hacer esto primero debe fijarse que la cadena no haya cambiado en el proceso de crear el nodo, y además asegurarse que no cambie luego de haber hecho la comprobación, o mientras está transmitiendo el nodo. Por eso, colocamos el mutex antes de fijarnos si la cadena cambió, y no permitimos recibir mensajes de nuevos bloques hasta que, o bien compruebe que la cadena cambió, o en caso de que no, hasta que termine de broadcastear. Para brodcastear llamamos a la funcion broadcast_block.

Cuando la cadena haya llegado a su longitud máxima y el nodo deba terminar, al minar y ver que el índice siguiente supera la longitud máxima, enviará a los otros nodos un mensaje de terminación con TAG_FIN, con send bloqueante porque quiero asegurarme que el mensaje de terminación llega, y setteara variables para avisarle al thread principal que debe hacer join (y terminar), y luego saldrá.

2.5. Función Broadcast block

Broadcast_block actualiza el valor del nodo actual y mediante un while envía a todos los nodos, excepto él mismo, el nuevo bloque que mino. Utilizamos rank_a_mensajear de modo que se calcule el modulo y los mensajes se envíen en forma de "mesa redonda"partiendo del siguiente del actual y llegando al anterior al mismo. De esta manera, cada nodo enviara su bloque en un orden distinto, lo cual es requerido para que no exista un nodo que reciba todos los mensajes último, o todos los mensajes primero, condiciones que generarían injusticia en la propagación del mensaje. El MPI_Isend , usado para enviar los mensajes, es no bloqueantes, pues consideramos que al nodo no le interesa bloquearse a esperar si los demás lo recibieron. Es su prioridad minar más nodos. (Además, si algún nodo terminara mientras hace el send se quedaría bloqueado infinitamente)

2.6. Análisis del protocolo

- ¿Puede este protocolo producir dos o más blockchains que nunca converjan?
- Si, se pueden dar casos en los que esto pase: Cuanto menos dificultad tengan en minar un bloque, más minaran cada uno sus propios bloques llevando una carrera paralela de los mismos que resultará en descartar los bloques de los demás. Luego podría pasar que 2 o mas nodos minen en paralelo y nunca migren su cadena a la de otro nodo.
  También, podría pasar que la velocidad de los mensajes afecte el resultado de las migraciones de cadena. Si un nodo recibe con mucho retraso los mensajes, y mina con poca frecuencia, se generará un hueco. Si este supera los 5 nodos para la validación de seguridad, ese nodo nunca migrará su cadena. Y nunca convergerá a otra. Lo que llevaría a que un nodo termine de minar mientras que los otros terminen sin haber minado el máximo de bloques. Si estos casos poco frecuentes no se dieran, las cadenas convergen normalmente.
- ¿Cómo afecta la demora o la pérdida en la entrega de paquetes al protocolo?

La demora de envíos le dará mas tiempo a un nodo a minar sus propios bloques, que en un futuro, si le llegaran los mensajes retrasado, quizas provoque que el indice sea superado por mucho a su actual y tenga que migrar su cadena. Además, la perdida de mensajes, con este protocolo, genera que los demás nodos no definan el bloque perdido dentro de sus diccionarios. Provocando migraciones de cadenas invalidas, ya que las comprobaciones se basan en lo definido en estos mapas, o fallas al encontrar un punto de convergencia.

- En nuestra implementación, la perdida de paquetes puede producirse cuando un nodo esta esperando el pasaje de una cadena pedida. En este momento, los avisos de nuevos bloques minados de otros nodos se reciben e ignoran. La consecuencia es que, en un contexto donde se tarda mucho en recibir el mensaje de la cadena en comparación con el de nuevos bloques (cuando la dificultad es baja), al terminar de migrar cadena la principal (o principales) hayan avanzado más de la distancia aceptable en la validación, y el nodo nunca converja su cadena con las líderes.
- ¿Cómo afecta el aumento o la disminución de la dificultad del Proof-of-Work a los conflictos entre nodos y a la convergencia?
- A menor dificultad, mas rápido minan los nodos, y mas coaliciones se producen. A si mismo menos convergen a una sola lista pues cada uno esta minando su propio bloque y lo logra, llevando cadenas paralelas. Cosa que pasa con menos regularidad si aumenta la dificultad y pocos tienen la posibilidad de minar.

3. Conclusiones

Pudimos ver las dificultades que conlleva la sincronización de procesos que no comparten la misma computadora, y aprendimos a usar lo básico de la interfaz MPI para enviar y recibir mensajes. Vimos y experimentamos con distintos mecanismos de comunicación tales como bloqueantes y no bloqueantes y como podíamos recibir un mensaje para trabajar con él. Todo esto relacionado a las bitcoins y las blockchains. Otra conclusión que sacamos fue que la relación entre la dificultad de minar y el tiempo que tardan las comunicaciones influye a la hora de migrar una cadena o no, produciendo que los nodos, en peor caso, sigan cada uno su cadena.