DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Bitcoins - Sistemas distribuidos

9 de diciembre de 2019

Sistemas Operativos
2do Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Giudice, Carlos Rafael	694/15	carlosr.giudice@gmail.com
Olmedo, Dante	195/13	dante.10bit@gmail.com
Rosende, Federico Daniel	222/16	federico-rosende@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Palabras Clave

Bitcoins, Sistemas distribuidos, MPI

Índice

1. Resumen

Para este trabajo practico nos proponemos implementar y analizar un problema determinado en un sistema distribuido.

El problema que resolveremos es el de la creación de una cadena de bloques (de aquí en adelante nos referiremos a la misma como blockchain) distribuida, utilizada en la implementación del sistema de pagos virtual Bitcoin, popular por ser una solución que anula la necesidad de third-parties en las transacciones, y provee transacciones inmutables gracias al proceso de consenso, del cual se hablará mas adelante.

Para la implementación, se utilizará OpenMPI para el envío de mensajes entre nodos del sistemas distribuidos y C++ como lenguaje para el código.

2. Introducción

2.1. Motivación de la creación del Bitcoin

El comercio en internet confía casi exclusivamente de instituciones financieras que hacen de third parties de confianza para procesar pagos electrónicos. Esto funciona bastante bien, pero carece de ciertas posibilidades.

Transacciones completamente irreversibles no son posibles, pues las instituciones financieras no pueden evitar mediar. El valor de la mediación entre partes incrementa el costo de una transacción, con lo que el tamaño mínimo redituable de una transacción se incrementa, y anula la posibilidad de transacciones mas pequeñas.

El Bitcoin se plantea como una solución a este problema, bajo la premisa de ser un sistema de pagos "peer-to-peer" inmutable, anulando la necesidad de la antes citada third-party. [1]

Este protocolo, propuesto originalmente por Satoshi Nakamoto [2], está implementado usando la tecnología de blockchain.

2.2. Blockchain

El problema a resolver es el desarrollo de una blockchain, la cual va a almacenar en un ambiente distribuido las transacciones realizadas en la red. En el contexto de nuestro trabajo práctico no van a existir las transacciones como tal, solo nos vamos a enfocar en la creación de nuevos bloques. Los mineros van a tener como objetivo agregar a la blockchain la mayor cantidad de nodos creados por ellos. En el caso de la minería en bitcoins, el agregado de nuevos bloques a la cadena se traduce en una compensación económica pero en el trabajo se deja de lado ese aspecto.

El procedimiento por el que deben pasar los mineros para poder anunciar nuevos bloques en el sistema es conocido como proof-of-work. Las ventajas de ese procedimiento para el sistema y su papel serán descriptas en su correspondiente sección. Para poder validar la autenticidad de un bloque, el mismo necesita tener suficiente información para garantizar las ciertas condiciones que se desean del sistema de transacciones (no sólo que pueda almacenar las operaciones realizadas, sino también que sea inmutable, o dicho de otra forma, que ningún bloque pueda ser adulterado para modificar una transacción).

En el caso del trabajo presentado, se cuenta con bloques que tienen la siguiente información:

1. *Índice:* Indica la posición del bloque en la blockchain.
2. *Identificador del dueño del bloque:* Está dado por el número asignado del nodo en el sistema distribuido.
3. *Dificultad:* Determina la complejidad del problema planteado para proof-of-work. En el caso de este trabajo, indica la cantidad de ceros que se requieren al comienzo del hash del bloque.
4. *Nonce:* Variable aleatoria que se utiliza para superar la prueba de proof-of-work. [3]
5. *Hash del bloque previo*: Cada bloque tiene el hash del bloque anterior, por lo que esta enlazado, mas allá de la lista como estructura, con el anterior.
6. Hash del propio nodo: También importa que el bloque no haya sido modificado desde su creación, esto puede ser corroborado con este campo.
7. Tiempo de creacion del bloque: Añade mas información que va a volver al hash del bloque todavía más difícil de falsificar.

Entonces, se tienen bloques con información que dificulta su adulteración, y que al estar distribuido puede tener un chequeo de validez por varias partes, lo que le da fuerza a la afirmación de que un bloque es o no válido.

Hasta aquí se establecieron ciertos aspectos deseables de la blockchain pero no su forma de aplicación. Acá es donde entra en juego el procedimiento de proof-of-work y el consenso entre nodos.

[1] Información extraída de https://bitcoin.org/bitcoin.pdf
[2] https://es.wikipedia.org/wiki/Satoshi_Nakamoto
[3] https://en.wikipedia.org/wiki/Cryptographic_nonce

Proof-of-work

Se afirmó que el tener un campo con un hash que identifica al bloque se traduce en una mayor garantía de seguridad e integridad de la blockchain, pero no se especificó la forma en que esto funciona. Aquí es donde proof-of-work adquiere significado.

Como idea general, el sistema de proof-of-work (o prueba de trabajo) aplicado a el problema de criptomonedas, es un procedimiento que tiene como finalidad evitar la producción masiva de unidades monetarias y ayudar a garantizar su validez. Se requiere que el minero resuelva algún tipo de problema de alto costo computacional pero de fácil verificación. En el caso particular del trabajo, se pide crear bloques cuyos valores de hash asociados comiencen con una cantidad de ceros determinada. Esta cantidad se denomina dificultad y está vinculada con la velocidad de obtención de nuevos bloques.

Una vez obtenido un nuevo bloque que cumple con la dificultad, se tiene un candidato para ser agregado en la cadena. Para que efectivamente se puedan sumar a la misma, es necesario comunicarles al resto de los nodos participantes del sistema que un nuevo bloque ha sido creado. Ahora bien, la decisión que tomen los nodos que reciben el bloque, determina la inclusión o no del bloque en la blockchain a largo plazo.

Aquí aparece la idea de consenso en el sistema y es importante detallar algunos aspectos sobre esto.

Consenso

Al realizar el envío del nuevo bloque obtenido al resto de los nodos, estos tomarán la decisión respecto a la inclusión del bloque en su cadena. Es deseable que los nodos apunten a incluir los bloques que son agregados por otros. El objetivo de esto es aprovechar al máximo el cómputo dedicado al minado de nuevos bloques.

Siempre se busca apostar por la cadena de consenso. Esta idea se conoce como Consenso de Nakamoto y a efectos del trabajo fue adoptada siguiendo las nociones detalladas en los casos de la función *validate_block_for_chain* descripta en la sección de pseudocódigo. Como idea general, se determina si es válido extender la blockchain actual con el nuevo nodo, si es necesario migrarla porque existe una blockchain de otro nodo que es más larga (aquí se evidencia fuertemente la noción de consenso ya que prevalece la cadena válida más larga) o si se descarta porque la blockchain propia está más adelantada que la del bloque que se recibe (si el nuevo bloque no extiende la blockchain propia, el índice del mismo no es mayor al del último que se tiene).

En la práctica fuera de este trabajo, uno de los principales beneficios obtenidos al aplicar esta noción de consenso, es la posibilidad de evitar el doble gasto (o double-spend). Al realizar transacciones, uno podría intentar usar la misma unidad monetaria dos veces. Esto se evita al lograr que una de las dos transacciones sea rechazada por los otros nodos de la blockchain y se acepte solamente una.

Notar que varios nodos pueden tener distintas blockchains. Esta idea de consenso, apunta a minimizar las diferencias entre las mismas y a que, a largo plazo, prevalezca una sobre las demás (la cadena válida más larga). Esto se explorará aún más en la sección de análisis al hablar sobre convergencia de blockchains.

3. Implementación

3.1. Explicación

A la hora de realizar la implementación de blockchain se completaron las siguientes funciones:

1. *send_block_to_everyone*: que realiza un broadcast a todos los otros nodos al crear un nuevo bloque.
2. *proof_of_work*: en la que se realiza el minado de bloques.
3. *validate_block_for_chain*: que verifica si un bloque recibido puede ser agregado a la cadena.
4. *verify_and_migrate_chain*: determina si es posible adoptar una cadena adelantada que fue recibida.

Adicionalmente, se cuenta con funciones que realizan la inicialización del primer bloque, que envían la cadena de bloques a un nodo que quedó atrás y otras auxiliares que se utilizan a la hora de realizar la migración de cadena.

La idea general del procedimiento es utilizar multithreading para tener dos threads realizando tareas en simultáneo.

- En primer lugar, se cuenta con el thread que recibe los mensajes de los otros nodos. Además, ante la llegada de uno, se encarga de realizar el llamado a *validate_block_for_chain*en caso de recibir un bloque o a *send_blockchain* (que envía la cadena al que la haya solicitado mediante *validate_block_for_chain*) en caso de recibir un pedido de cadena.
- El otro thread, es el encargado de minar bloques y realizar el llamado a **send_block_to_everyone** en caso de extender la cadena de forma exitosa.

Para determinar la finalización de la ejecución se cuenta con una longitud máxima de la blockchain. Al alcanzar la misma, el thread que se encuentra minando bloques, realiza el llamado a *MPI_Abort* que finaliza el procedimiento y les comunica a todos los nodos que deben hacer lo mismo.

3.2. Pseudocódigo

send_block_to_everyone

A continuación se presenta el pseudocódigo de *send_block_to_everyone*

Algorithm 1 send_block_to_everyone

```
1: procedure SEND_BLOCK_TO_EVERYONE(block_to_send)
2:     for i in 0...total_nodes − 1 do
3:         new_rank ← (mpi_rank + i)_(mod total_nodes)
4:         MPI_Send(block_to_send, new_rank)
5:     end for
6: end procedure
```

Obervaciones

La función es sumamente simple ya que solamente se encarga del envío del bloque a todos los nodos. Cabe aclarar que el envío se realiza en un orden diferente para cada nodo. Debido a que cada nodo posee un número *mpi_rank* que lo identifica, se aprovecha esto para comenzar el procedimiento desde el nodo con identificador $mpi\_rank + 1$ y se utiliza la función módulo para lograr el despacho del bloque hasta el nodo $mpi\_rank - 1$.

proof_of_work

Probablemente sea la función más interesante de las implementadas ya que es en la que ocurre el minado de bloques. Su pseudocódigo es el siguiente.

proof_of_work

Algorithm 2 proof_of_work

```
1: procedure PROOF_OF_WORK(block_to_send)
2:     while True do
3:         if blocks_to_mine ≤ last_block_in_chain.index then
4:             MPI_Abort(0)
5:         end if
6:         block ← last_block_in_chain
7:         block.index ← last_block_in_chain.index + 1
8:         block.owner ← mpi_rank
9:         block.difficulty ← default_difficulty
10:        block.created_at ← current_time
11:        block.previous_hash ← last_block_in_chain.block_hash
12:        block.nonce ← random_nonce
13:        hash_str ← block_to_hash(block)
14:        if solves_problem(hash_str) then
15:            _send_mutex.lock()
16:            if last_block_in_chain.index < block.index then
17:                block.hash ← hash_str
18:                node_blocks.insert(hash_str, block)
19:                last_block_in_chain ← block
20:                send_block_to_everyone(block)
21:            end if
22:            _send_mutex.unlock()
23:        end if
24:    end while
25: end procedure
```

Observaciones

- Se cuenta con constantes *blocks_to_mine* y *default_difficulty* que determinan, como sus nombres indican, la cantidad máxima de bloques que se desea en la blockchain previo a la finalización de la ejecución y la dificultad está asociada a la cantidad de ceros que se esperan al comienzo del hash obtenido. Esto último está directamente vinculado con el tiempo que demorará la ejecución en alcanzar la cantidad de bloques minados que se desean.
- Las funciones *block_to_hash* y *solves_problem* simplemente se encargan de obtener un hash asociado al bloque y de verificar que el mismo cumpla con lo pedido por *default_difficulty*.
- Al establecer el tiempo de creación del bloque, en el pseudocódigo se utiliza *current_time* como una forma general de obtener el tiempo actual. En el código, se utilizó la función *time* de C++.
- Se cuenta con un diccionario *node_blocks* de bloques en el que las claves son sus hashes respectivos y se utiliza un mutex para evitar que ambos threads lo modifiquen simultáneamente. Al conseguir el mutex, es necesario verificar que no haya cambiado el último elemento de la blockchain ya que podría haber sido modificado por el otro thread mientras el que ejecuta *proof_of_work* experaba el mutex.
- Al encontrar un bloque nuevo que puede ser agregado a la cadena, se lo comunica a los otros nodos realizando un broadcast con la función *send_block_to_everyone*.

validate_block_for_chain

Esta función es importante porque determina la decisión a tomar al recibir un bloque nuevo de otro nodo. Si bien su accionar es crucial para su correcto funcionamiento, su pseudocódigo es simple.

validate_block_for_chain

Algorithm 3 validate_block_for_chain

```
1: procedure VALIDATE_BLOCK_FOR_CHAIN(block)
2:     node_blocks.insert(block.hash, block)
3:     if (block.index == 1 ∧ last_block.index == 0) ∨ (block.index == last_block.index + 1 ∧
   block.previous_hash == last_block.hash) then
4:         last_block = block
5:         return True
6:     end if
7:     if (block.index == last_block.index + 1 ∧ block.previous_hash ≠ last_block.hash) ∨ (block.index >
   last_block.index + 1) then
8:         return verify_and_migrate_chain(block)
9:     end if
10:    if block.index == last_block.index ∨ block.index < last_block.index then
11:        return False
12:    end if
13:    return False
14: end procedure
```

Observaciones

En esta función se revisan los diferentes casos que pueden darse a la hora de recibir un nuevo bloque.

- Los primeros dos casos se encuentran contemplados en el primer *if*. En un primer caso, se recibe un bloque cuyo índice es igual a 1 y el último bloque de la blockchain del nodo que recibe tiene índice 0. Esto quiere decir, que se recibió un nodo que es el primero de la blockchain de algún otro nodo y todavía no se tiene ninguno en la cadena. El otro caso, es el que se da al recibir un bloque con un índice que supera en uno al del último bloque actual y cuyo atributo vinculado al hash del bloque previo coincide con el del último de la blockchain que se tiene. En ambos casos, se puede agregar al bloque recibido como nuevo último.
- Los segundos dos casos que se consideran en el segundo *if*. En estos, se recibe una un bloque que cumple que tiene un índice que supera por más de uno al del último bloque actual o bien es igual al del último más uno pero su atributo de hash previo no coincide con el del último que se tiene. Aquí, la solución es migrar la cadena mediante la función *verify_and_migrate_chain*.
- Finalmente, los últimos dos casos se observan en el último *if*. En este caso, no es posible agregar el bloque a la blockchain debido a que su índice es igual o menor al del último bloque actual.

verify_and_migrate_chain

Se invoca a esta función producto de haber recibido un bloque que pertenece a una cadena adelantada o cuyo hash previo no coincide con el último bloque que se tiene. Por ello, es necesario migrar la blockchain a la que tiene como último al bloque recibido. Por lo tanto, es necesario solicitar dicha cadena mediante un mensaje al dueño del bloque recibido.

verify_and_migrate_chain

Algorithm 4 verify_and_migrate_chain

```
1: procedure VERIFY_AND_MIGRATE_CHAIN(block)
2:     node_blocks.insert(block.hash, block)
3:     MPI_Send(block, block.owner)
4:     received_blockchain ← MPI_Recv(block.owner)
5:     if check_first(received_blockchain, block) ∧ check_chain(received_blockchain) ∧ −1 < find_block(received_blockchain, block) then
6:         for j in 0...i + 1 do
7:             node_blocks.insert(received_blockchain[j].hash, received_blockchain[j])
8:         end for
9:         last_block ← received_blockchain[0]
10:        return True
11:    end if
12:    return False
13: end procedure
```

Observaciones

A la hora de realizar la migración es necesario, como fue mencionado previamente, solicitar la nueva cadena. Se cuenta con una constante *validation_blocks* que indica la cantidad máxima de bloques que se pueden recibir para la migración. Luego de realizar dicho pedido, se procede a verificar que la cadena obtenida cumpla ciertos requisitos:

1. El primer bloque de la lista obtenida debe contener el hash pedido y el índice del bloque que provocó la invocación a la función. Además, se tiene que cumplir que el hash del bloque coincida con el calculado por *block_to_hash* Esto se realiza en *check_first*.

2. Para el elemento k de la lista tiene que darse que el atributo vinculado al hash del bloque previo sea igual al hash del elemento $k + 1$. Más formalmente, debe cumplirse lo siguiente.

   $$received\_blockchain[k].previous\_hash == received\_blockchain[k+1].hash,\ \forall k \in [0, validation\_blocks - 1)$$

   Y por otro lado se busca una relación similar en base a los índices de cada bloque. Es decir, el $k + 1$ debe tener el índice anterior al del elemento k.

   $$received\_blockchain[k].index - 1 == received\_blockchain[k+1].index,\ \forall k \in [0, validation\_blocks - 1)$$

   Estas condiciones se corroboran en el llamado a *check_chain*.

Y luego se busca en la lista recibida algún bloque que ya estuviera en la cadena que se tiene o en su defecto el primer elemento de la blockchain del dueño del nodo que generó el conflicto. Esto se realiza en el llamado a *find_block*. En caso que la función devuelva -1 se descarta por seguridad. La respuesta -1 indica que no encontró ni el primer elemento de la lista (de índice 1), ni algo en común entre la blockchain propia y la lista recibida. Por lo tanto, no es posible reconstruir la cadena.

Detalles generales sobre el thread encargado de la comunicación

Como fue mencionado previamente, el thread que no se encuentra realizando el minado, está encargado de recibir y enviar mensajes. Para ello, hace uso de las funciones *MPI_Recv* y *MPI_Send*.

Su accionar encuentra dos posibles escenarios:

- En un primer lugar, está la situación en la que llega un nuevo bloque que ha sido minado por otro nodo. En este caso, el thread hace el llamado correspondiente a la función *validate_block_for_chain* con el bloque recibido como parámetro. A partir de aquí, la ejecución continua siguiendo el código de *validate_block_for_chain* y aplicando el caso que corresponda.
- En el otro caso, el thread recibe un pedido de cadena de un nodo que recibió un bloque minado por el thread que se encuentra en *proof_of_work*. Aquí, simplemente realiza el llamado a una función auxiliar *send_blockchain* que se encarga de enviar la lista de bloques al otro nodo utilizando *MPI_Send*. Aquí, se envían bloques hacia atrás a partir del que generó el pedido de cadena y se mandan a lo sumo *validation_blocks* (podrían ser menos si se llega al bloque de índice 1).

En ambos casos, inmediatamente luego de recibir el mensaje, se realiza el pedido del mutex usado en *proof_of_work* y se lo libera después de hacer el llamado a la función correspondiente.

Este procedimiento se realiza constantemente. Luego de liberar el mutex, el thread procede a esperar un nuevo mensaje con *MPI_Recv*.

4. Análisis

4.1. Convergencia de blockchains

Desde un punto de vista teórico, podemos definir una situación hipotética en la cual no ocurra la convergencia. Pensemos en una red de n nodos. Cada uno de estos nodos intentará minar bloques y comunicará al resto en la eventualidad de haberlo logrado. Ahora bien, la aceptación de el bloque minado depende del estado de los nodos que reciben la notificación. Estos solo aceptarán al nuevo bloque si este posee un índice que es mayor al último bloque de las cadenas de ellos pero no lo excede en más de una unidad. Entonces si imaginamos una situación en la cual todos los nodos minan exitosamente bloques exactamente al mismo tiempo, ninguno de los mensajes va a ser aceptado, porque en ningún caso se cumple que una cadena está mas adelantada que otra. Analicemos ahora este fenómeno con una perspectiva más práctica. Si modelamos al éxito del minado como una variable geométrica con probabilidad p, donde p será mas chica cuanto mayor sea la dificultad del minado, vemos que la probabilidad de que todos los nodos resuelvan la ecuación en un momento dado es p^n. Esto representa en si a otra variable geométrica, donde la probabilidad de que en sucesivas iteraciones coincidan los eventos de minado exitoso decrece exponencialmente con el número de iteraciones. De esta manera podemos convencernos de que, en la práctica, utilizar una dificultad no trivial nos garantiza convergencia de las cadenas.

4.2. Pérdida de paquetes y demoras

La demora en los tiempos de envío de paquetes impacta negativamente en los tiempos de ejecución del thread que se encuentra esperando mensajes. Esto se debe a que la función utilizada para recibirlos *RECV* es de carácter bloqueante y, por lo tanto, permanece ahí hasta que suceda algo en el medio que se está escuchando. Así, si se tienen tiempos demasiado grandes de espera, el thread encargado de esta tarea, dedica grandes cantidades de su ejecución a esperar. Además, si la demora fuera lo suficientemente grande como para permitir que se agreguen bloques a la cadena por el thread que se encuentra minando, podría provocar una migración evitable. Es decir, se podría caer en una situación con dos cadenas de longitudes que distan en más de un bloque debido a la tardanza del mensaje. En otras circunstancias, podría evitarse esta situación y simplemente caer en algún otro caso de validación de bloque.

Ahora bien, el peor caso sería que un mensaje se pierda. Esto puede causar una situación similar a la de la migración innecesaria (mencionada en el párrafo anterior). Como idea general, si un mensaje se perdiera, el nodo al que se le enviaba nunca se enterará de su existencia y aquí se tienen dos casos:

- En un primer lugar, está la situación en la que el thread que iba a recibir el mensaje estaba simplemente en el *Recv* del ciclo que se encarga de determinar lo que se hace con el bloque recibido. Aquí la consecuencia es que ese nodo no considerará al bloque enviado y esto puede derivar en migraciones futuras.
- Por otro lado, se tiene el caso en el que el thread se encontraba migrando la cadena y esperaba una subcadena del otro nodo para acoplarla a la suya. En este caso, la consecuencia es mucho más severa puesto que este thread se quedará esperando allí por siempre.

A modo de solución se podría demandar confirmación del receptor. De no recibir la misma, se reenvía el mensaje. Algo similar a un commit de dos fases. Por otro lado, en el caso de la velocidad de tranferencia de mensajes, la solución que surge naturalmente es mejorar el medio de transferencia de los mensajes.

4.3. Conflictos entre nodos respecto a proof of work

El aumento de la dificultad del proof of work provocaría que el tiempo que requiera minar un bloque sea mayor, esto disminuiría la cantidad de migraciones por unidad de tiempo. Esto se debe a que es más fácil que los nodos mantengan sus cadenas en longitudes parejas. Como consecuencia hay un menor solapamiento de los envíos de mensajes. En consecuencia, el medio de transferencia de los mismos está menos congestionado y podría impactar positivamente en la velocidad de transferencia de los mismos. Esto deriva en un aspecto positivo del aumento de la dificultad. Como las cadenas permanecen parejas, se evita desperdiciar tiempo en migraciones y problemas de cadenas adelantadas.

Respecto a la convergencia, así como se aclaro en la primer sección, es altamente improbable que se dé la situación de no convergencia, esto se mantiene al variar la dificultad de proof of work, lo que sí podría verse influenciado por este aumento es a una convergencia mas rápida, pues al haber menos conflictos y migraciones la consistencia entre cadenas de distintos nodos se mantiene más fácilmente. Nuevamente, esto se traduce en una menor cantidad

de migraciones.

En el apartado siguiente, se buscará dar evidencia empírica para la afirmación sobre los tiempos de ejecución (vinculados con el costo del minado) y la dificultad de proof-of-work.

5. Experimentación

5.1. Objetivo

Nos propusimos entender la evolución del costo de la ejecución del algoritmo en relación a los parámetros de entrada. Para esto conjeturamos sobre las implicaciones de las variaciones de los parámetros y luego medimos el rendimiento del programa, pudiendo así arribar a conclusiones empíricas.

5.2. Hipótesis

Nos inclinamos por analizar dos parámetros. En primer lugar, como se estableció en el apartado anterior, es de interés estudiar la dificultad de minar un bloque. Esta está asociada a la complejidad del hash que deben replicar los nodos, con lo que, como fue conjeturado en la sección de análisis, aumentar la dificultad debería aumentar el costo temporal de llegar a una solución. Adicionalmente, vamos a analizar como varía el costo temporal en relación a la cantidad de nodos. El algoritmo termina cuando se ha minado una cantidad determinada de bloques. Esta tarea es cooperativa, o sea que los nodos comparten progreso. Esto debería indicar que una mayor cantidad de nodos contribuirá a la temprana finalización de la ejecución.

5.3. Metodología

Para testear nuestras hipótesis, realizamos ejecuciones del algoritmo variando un parámetro y dejando fijo el otro. Luego variando ambos, probando así distintas combinaciones. Para cada configuración corrimos el algoritmo cinco veces (número elegido empíricamente para reducir la varianza). De esta manera quedó suficientemente caracterizado el comportamiento a observar.

5.4. Resultados

En el siguiente gráfico de dispersión, se utilizaron los ejes para las variables independientes y el color para la variable dependiente.

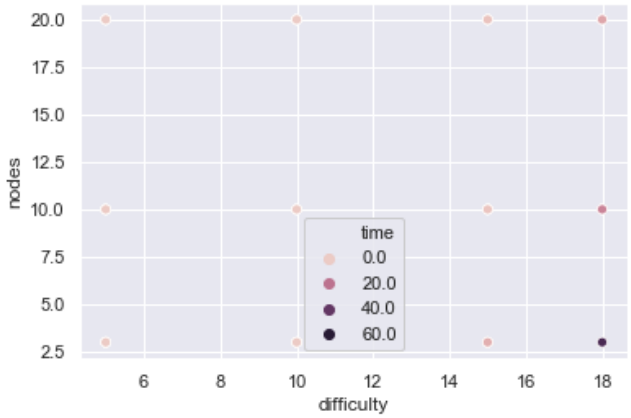

Figura 1: Tiempo en función de la cantidad de nodos y la dificultad de minar un bloque.

Vemos que los patrones que se observan son acordes a las hipótesis arriba enunciadas, siendo la dificultad un factor que ralentiza la ejecución, y la cantidad de nodos un factor que la acelera. Así, se tiene evidencia empírica para respaldar la conjetura sobre la dificultad en relación del tiempo manifestada en la sección de análisis.

6. Conclusiones y trabajo a futuro

A lo largo de este trabajo se estudió el problema de la creación de una blockchain. Se encaró la problemática de desarrollar las principales funcionalidades del protocolo.

A partir de la implementación obtenida de la blockchain, fue posible apreciarla como una poderosa herramienta para llevar registro de un sistema y, en base a esto, tomar decisiones (como por ejemplo la validez de un bloque). En torno a esto, se concluye, además, que el procedimiento de proof-of-work constituye un método interesante y efectivo para abordar el problema de generación masiva de divisas.

Se utilizó MPI para realizar comunicación entre distintos nodos. Esta interfaz permitió la comunicación del trabajo distribuido de forma efectiva y ordenada. Sería interesante poder explorar otros métodos de comunicación entre nodos para contrastarlos con el anteriormente mencionado.

Al estar compartiendo recursos entre distintos threads, fue necesario utilizar técnicas de programación concurrente, sin ellas hubiera sido imposible evitar condiciones de carrera en el comportamiento del algoritmo.

Gracias a las herramientas utilizadas fue posible implementar el algoritmo distribuido y experimentar con este. Las observaciones realizadas sobre el comportamiento del mismo fueron consistentes con las hipótesis propuestas. Consideramos así que queda adecuadamente caracterizado el comportamiento del algoritmo, lo cual nos da la seguridad de poder utilizarlo.

Como trabajo a futuro, sería interesante abordar la problemática de las transacciones y expandir el código agregándole dicha funcionalidad. A partir de esto, habría que determinar el tipo de comunicaciones entre nodos que serían necesarias y analizar el impacto de esto en los tiempos del programa.

Finalmente, se podrían realizar experimentos con la finalidad de determinar el impacto del consenso en torno al comportamiento de las cadenas y su integridad.