UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

ALGORITMOS DE CONSENSO Y LA BLOCKCHAIN

T E S I S

QUE PARA OBTENER EL TÍTULO DE
LICENCIADO EN MATEMÁTICAS

P R E S E N T A

YUGUO XIE

T U T O R

DR. ARMANDO CASTAÑEDA ROJANO

CIUDAD UNIVERSITARIA, CDMX, 2022

Índice

Resumen

En esta tesis, primero vamos a dar una introducción breve sobre los sistemas distribuidos, el modelo Maquina de Estado Replicada (MER), la imposibilidad de FLP y los algoritmos de consenso clásicos, por ejemplo Paxos y PBFT. Después hablaremos de la Blockchain, Sistemas Descentralizados, el protocolo de Bitcoin, y su análisis matemático incluyendo a PoW (proof of work).

1. Introducción

1.1. Maquina de Estado Replicada

La replicación de servicio es clave para proporcionar alta disponibilidad y tolerancia de fallas en los sistemas distribuidos. Básicamente consiste en múltiple servidores para dar un servicio como si fuera un solo servidor para los clientes. La replicación aumenta la disponibilidad por usar más recursos (CPU, Memoria, etc), y la tolerancia de fallas está garantizada por tener la redundancia necesaria en el sistema.

Maquina de Estado Replicada (MER) es un método general para implementar un servicio tolerante a fallas replicando servidores y coordinando las interacciones del cliente con las réplicas del servidor, ofreciendo un marco para diseñar los protocolos del manejo de replicación.

Una MER consiste en *variables de estado* y *comandos*, los clientes envían comandos a las replicas para que los ejecuten, y devuelven los resultados. Por ejemplo, un banco puede tener un grupo de servidores y cajeros automáticos para dar los servicios bancarios, cuando depositamos una cantidad x de dinero a un cajero automático, el cajero automático tiene el rol de un cliente en el modelo *servidor-cliente*, mandando un comando $\langle \text{DEPOSITA}, x, \text{CUENTA} \rangle$ a los servidores, los servidores ejecutan este comando, devuelven los resultados al cajero automático y actualizan sus estados, luego el cajero automático nos muestra que la operación fue exitosa.

Las MERs cumplen tres propiedades [2]:

- Estado inicial: todas las réplicas correctas comienzan en el mismo estado.
- Determinismo: todas las réplicas correctas que reciben la misma entrada en el mismo estado producen la misma salida y el estado resultante.
- Coordinación: todas las réplicas correctas procesan la misma secuencia de comandos.

El sistema tiene que satisfacer:

- *Seguridad* (Safety): todas las réplicas correctas ejecutan la misma secuencia de operaciones.
- *Viveza* (Liveness): Se ejecutan todas las operaciones correctas de los clientes.

Una replica del sistema será considerada *fallida* cuando no se comporta de forma consistente con su especificación. Las clases representativas de fallas más comunes y estudiadas son:

- Fallas Bizantinas: la replica tiene comportamiento malicioso y arbitrario.
- Fallas de tipo paro: la replica se detiene cuando ocurre falla.

Un sistema que consiste en un conjunto de replicas distintas es f-tolerante si cumple con sus especificaciones siempre que no más de f de esas replicas están fallidas durante algún intervalo de interés: un sistema f fallas tolerante puede continuar funcionando correctamente si ocurren más de f fallas, pero ya no se puede garantizar que la operación sea correcta.

1.2. Modelo de Interacción

En un sistema distribuido es difícil establecer límites en el tiempo que se puede tomar para la ejecución de un proceso, la entrega de mensajes o el desvío del reloj de una computadora (clock drift). Hay dos modelos simples y son de los más estudiados: síncrono y asíncrono. El primero tiene una fuerte suposición de tiempo y el segundo no hace suposiciones sobre el tiempo.

Un sistema distribuido síncrono es el que se define los siguientes límites:

- El tiempo para ejecutar cada paso de un proceso tiene límites inferiores y superiores conocidos.
- Cada mensaje transmitido por un canal se recibe dentro de un tiempo límite conocido.
- Cada proceso tiene un reloj local cuya tasa de desvío del tiempo real tiene un límite conocido.

Un sistema distribuido asíncrono es aquel en el que no hay límites en:

- Velocidades de ejecución del proceso: por ejemplo, un paso del proceso puede llevar solo un nano segundo y otro un siglo, todo lo que se puede decir es que cada paso puede tomar un tiempo arbitrariamente largo.
- Retrasos en la transmisión de mensajes: por ejemplo, un mensaje del proceso A al proceso B puede entregarse en un tiempo insignificante y otro puede demorar varios años. En otras palabras, se puede recibir un mensaje después de un tiempo arbitrariamente largo.
- Tasas de desvío del reloj: nuevamente, la tasa de desvío de un reloj es arbitraria.

1.3. Problema de Consenso

En los sistemas distribuidos prácticos, los clientes se comunican con las MERs mediante el *paso de mensajes* (message passing), pero los mensajes pueden ser retrasados, duplicados etc, entonces puede pasar que las MERs distintas tienen secuencias de operaciones distintas o incompletas, algunas MERs pueden ser fallidas. En esos escenarios, para garantizar la seguridad y la viveza de los sistemas, las máquinas de estado replicadas no fallidas ejecutan la misma secuencia de operaciones con todos los comandos de los clientes correctos. Para resolver este problema tenemos que resolver un sub-problema:

Todas las MERs no fallidas ejecutan el mismo i-ésimo comando.

Para eso, las máquinas de estado replicada no fallidas tienen que consensuar el valor del *i-ésimo* comando.

Podemos ver una máquina de estado replicada como un proceso, este problema se puede expresar más formalmente:

El problema de consenso exige un acuerdo entre una colección de procesos $\{P1, P2, \cdots, Pn\}$ *sobre un valor después de que uno ó más procesos proponen valores de entrada. Para llegar al consenso, cada proceso* P_i *empieza con el estado indeciso y propone un valor* $v_i \in D$ $(i = 1, 2, \ldots, N)$*, donde* D *es el conjunto de los valores de las decisiones (en el contexto de MER,* D *es conjunto de comandos que se pueden ejecutar). Los procesos se comunican, intercambian los valores, y después cada proceso establece el valor de su variable de decisión* d_i*. una vez que el valor de* d_i *se ha establecido, entonces los estados ya están en estado decidido.*

Este problema se llama *el problema de consenso*, es uno de los dos problemas más fundamentales y estudiados en la área de sistemas distribuidos (otro es el problema de consistencia). Un enfoque dominante en los sistemas distribuidos es buscar los algoritmos de consenso para resolver este problema bajo modelos distintos. Los requisitos de un algoritmo de consenso son las siguientes condiciones que se deben de cumplir para su cada ejecución [1]:

1. Terminación: eventualmente cada proceso correcto establece su variable de decisión.
2. Acuerdo: el valor de decisión de todo los procesos correctos es lo mismo.
3. Integridad: La decisión de un procesos es la propuesta de algún proceso.

Existen algoritmos de consenso con el modelo síncrono [3], pero en el mundo real los sistemas distribuidos síncronos son difíciles de implementar en general [4]. A principió de los años 80, la gente trataba resolver el problema de consenso con el modelo

asíncrono, en el siguiente capítulo veremos un resultado teórico fundamental para este problema.

2. Imposibilidad de FLP

A mediados de la década de 80, la investigación sobre sistemas distribuidos se convirtió más popular, cuando la gente trataba resolver el problema de consenso bajo la condición asíncrona, Tres investigadores Michael J. Fischer, Nancy Lynch y Mike Paterson, publicaron un resultado sorprendente:

En un sistema completamente asíncrono, no existe un algoritmo correcto de consenso que pueda tolerar una o más procesos con fallas de tipo paro.

Este resultado a veces se le llama teorema de imposibilidad de FLP que lleva el nombre de los autores a quienes se les otorgó el Premio Dijkstra en el 2001 por este importante trabajo.

Es decir, los algoritmos de consenso, incluyendo Paxos y PBFT, que discutiremos posteriormente, no son completamente asíncronos. Por ejemplo, la selección del líder proponente en Paxos tiene que ser síncrona, y el modelo de PBFT tampoco es totalmente asíncrono al usar sincronía para garantizar la viveza parcialmente. Sin embargo, FLP no nos dice que nunca se pueda llegar a un consenso: simplemente que, bajo los supuestos del modelo, ningún algoritmo siempre puede llegar a un consenso en un tiempo limitado. En la práctica, es muy poco probable que ocurra.

Explicamos brevemente la idea básica de la demostración de este teorema: primero demuestra que un algoritmo de consenso puede tener su estado inicial "incertidumbre", que los procesos con valores iniciales de entradas de $\{0, 1\}$ pueden tener ambos valores 0 y 1 como posibles candidatos de consenso, es decir, dependiendo la ejecución posterior, los procesos pueden llegar a decidir 0 o 1. Después la prueba demuestra que un estado incertidumbre puede llegar a otro estado que es incertidumbre también por el retraso de un mensaje. Así que el estado del algoritmo puede estar incertidumbre indefinidamente si los mensajes se demoran todo el tiempo durante de la ejecución, demostraremos que este escenario es posible que se ocurra, ya que el algoritmo no va a llegar a tener un valor de consenso.

2.1. Modelo y Definiciones

Cualquier algoritmo o teorema distribuido tiene sus supuestos sobre el escenario del sistema, que se denomina modelo del sistema. FLP se basa en los siguientes supuestos:

- Comunicación asíncrona: implica que no hay reloj, no hay sincronización horaria, no hay tiempo de espera, no hay detección de fallas, los mensajes pueden retra-

sarse arbitrariamente ó llegar a su destino en un orden distinto al que fueron enviados.

- El sistema de mensajes es confiable: entrega todos los mensajes correctamente y exactamente una vez.
- No hay fallas Bizantinas (que los procesos y la comunicación en el sistema no comportan como maliciosos ni arbitrarios).
- A lo sumo un proceso falla.

Vamos a usar un protocolo de consenso simplificado y universal para tener un resultado más fuerte:

Sea $\mathcal{P}$ un protocolo de consenso en un sistema asíncrono $\mathcal{S}$ de N procesos ($N > 2$). Cada proceso p tiene un registro de entrada x_p de un bit, un registro de salida y_p con valores en $\{b, 0, 1\}$ y una cantidad ilimitada de almacenamiento interno, donde b representa *bivalente*, es decir, el proceso puede llegar a elegir 0, 1, o sigue siendo bivalente, por momento actual, es indeciso. Los valores en los registros de entrada y salida, junto con el contador del programa y el almacenamiento interno, comprenden el estado interno. Los estados iniciales prescriben valores iniciales fijos para todos menos el registro de entrada; en particular, el registro de salida comienza con el valor b. Los estados en los que el registro de salida tiene un valor 0 o 1 se distinguen como *estados de decisión*. p actúa de manera determinista de acuerdo con una función de transición. La función de transición no puede cambiar el valor del registro de salida una vez que el proceso ha alcanzado un estado de decisión; es decir, el registro de salida *se escribe una sola vez*. Todo el sistema $\mathcal{S}$ está especificado por las funciones de transición asociadas con cada uno de los procesos y los valores iniciales de los registros de entrada. $\mathcal{P}$ cumple las tres propiedades de terminación, acuerdo y integridad que mencionamos anteriormente.

Los procesos se comunican enviándose mensajes unos a otros. Un mensaje es un par (p, m), donde p es el nombre del proceso de destino y m es un *valor de mensaje* de un universo fijo $\mathcal{M}$. El sistema de mensajes mantiene un multiconjunto de mensajes, llamado búfer de mensajes, esto se quitaría que tienen enviado pero aún no entregado. El búfer soporta dos operaciones abstractas:

enviar(p, m): se pone (p, m) en el búfer de mensajes;

recibir(p, m): se elimina algún mensaje (p, m) del búfer y devuelve m, en el que en caso de que digamos (p, m) se entrega, o devuelve el marcador nulo especial 0 y deja el búfer sin cambios.

Por lo tanto, el sistema de mensajes actúa de manera no determinista, sujeto solo a la condición de que si recibir(p) se realiza infinitamente muchas veces, después cada

mensaje (p, m) en el búfer de mensajes finalmente se entrega. En particular, el sistema de mensajes puede devolver 0 un número finito de veces en respuesta a recibir(p), aunque haya un mensaje (p, m) en el búfer. De hecho este sistema de mensajes es para simular la comunicación asíncrona.

Una *configuración* del sistema consiste en el estado interno de cada proceso, junto con el contenido del búfer de mensajes. Una *configuración inicial* es aquella en la que cada proceso comienza en un estado inicial y el búfer de mensajes está vacío.

Un *paso* lleva una configuración a otra (se cuenta por el cambio del estado de un solo proceso). Sea C una configuración. El paso ocurre en dos fases. Primero, *recibir*(p) se realiza en el búfer de mensajes en C para obtener un valor $m \in \mathcal{M} \cup \{\varnothing\}$. Luego, según el estado interno de p en C y en m, p ingresa a un nuevo estado interno y envía un conjunto finito de mensajes a otros procesos. Como los procesos son deterministas, el paso está completamente determinado por el par $e = (p, m)$, que llamamos *evento*. (Este "evento" debe considerarse como la recepción de m por p.) $e(C)$ denota la configuración resultante, y decimos que e puede aplicarse a C. Tenga en cuenta que el evento $(p, \varnothing)$ siempre puede ser aplicado a C, por lo que siempre es posible que un proceso tome otro paso.

Una *calendarización* (schedule) desde C es una secuencia finita o infinita σ de eventos que se pueden aplicar, en orden, comenzando desde C, osea si σ consiste en e_1,...,e_n, $\sigma(C) = e_n(...(e_1(C)))$. La secuencia de pasos asociada se llama una *corrida* (run). Si σ es finita, dejamos que $\sigma(C)$ denote la configuración resultante, que se dice que es *accesible desde C*. Una configuración *accesible* desde alguna configuración inicial se dice que es accesible. De aquí en adelante, que todas las configuraciones mencionadas son accesibles.

Una configuración C tiene un *valor de decisión* v si algún proceso p está en un estado de decisión con $y_p = v$, eso implica que todos los demás procesos también deciden v, pues el protocolo soluciona el problema del consenso; si ningún proceso tiene un valor de decisión, la configuración sería *bivalente*. Un protocolo de consenso es *parcialmente correcto* si cumple dos condiciones:

1. Ninguna configuración accesible tiene más de un valor de decisión.
2. Para cada $v \in \{0, 1\}$, alguna configuración accesible tiene un valor de decisión v.

Un proceso p no es $fallido$ en una corrida si ejecuta infinitos pasos y, de lo contrario, es fallido. Una corrida es *admisible* siempre que, como máximo, un proceso sea fallido y que todos los mensajes enviados a procesos no fallidos serán recibidos eventualmente.

Una corrida es *decidida* siempre que algún proceso alcance un estado de decisión en esa corrida. Un protocolo de consenso $\mathcal{P}$ es *totalmente correcto a pesar de una falla* si es parcialmente correcto, y cada corrida admisible es una corrida decidida. El

teorema principal muestra que cada protocolo parcialmente correcto para el problema de consenso tiene una corrida admisible que no es una corrida decidida. Es decir, se puede pasar que protocolo nunca este en un estado de decisión si la ejecución siempre solo cae en esta corrida indecisa.

2.2. Demostración de Imposibilidad de FLP

LEMA 1 (**Conmutabilidad**). *Suponga que desde alguna configuración C, las calendarizaciones σ_1, σ_2 conducen a las configuraciones C_1, C_2, respectivamente. Si los conjuntos de procesos que toman pasos en σ_1 y σ_2, respectivamente, son disjuntos, entonces σ_2 puede aplicarse a C_2 y σ_1 pueden aplicarse a C_2, y ambos conducen a la misma configuración C_3.*

Demostración: Este resultado deduce directamente de las definiciones anteriores. □

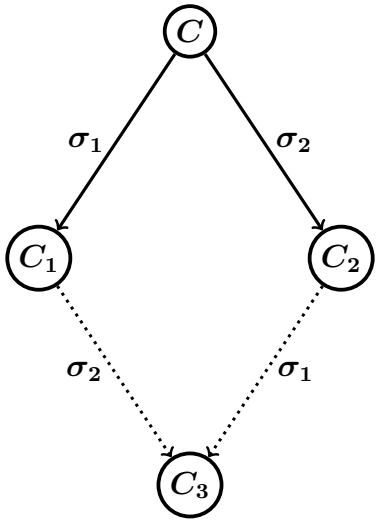

Figura 1: La conmutabilidad

LEMA 2. *$\mathcal{P}$ tiene una configuración inicial bivalente.*

Demostración: Vamos a usar el método de contradicción. Supongamos que $\mathcal{P}$ no contiene una configuración bivalente. Entonces $\mathcal{P}$ debe tener configuraciones iniciales 0-valente y 1-valente por la supuesta corrección parcial. Llamemos a dos configuraciones iniciales adyacentes si difieren solo en el valor inicial x, de un solo proceso p. Cualquier

dos configuraciones iniciales están unidas por una cadena de configuraciones iniciales, cada una adyacente a la siguiente (se forman un grafo, vea la figura 2). Por lo tanto, debe existir una configuración inicial 0-valente C_0 adyacente a una configuración inicial 1-valente C_1. Sea p el proceso en cuyo valor inicial difieren.

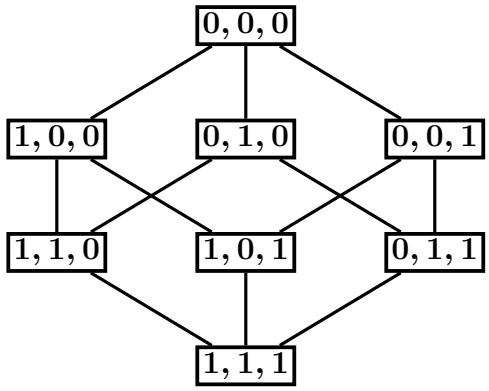

Figura 2: Configuraciones adyacentes con tres procesos

Ahora considere alguna corrida decisiva y admisible desde C_0 en las que el proceso p no toma pasos, sea σ ser la calendarización asociado. Entonces σ también puede aplicarse a C_1, y las configuraciones correspondientes en las dos corridas son idénticas, excepto por el estado interno del proceso p. Se muestra fácilmente que ambas corridas eventualmente alcanzan el mismo valor de decisión. Si el valor es 1, entonces C_0 es bivalente; de lo contrario, C_1 es bivalente. Cualquiera de los casos contradice la supuesta inexistencia de una configuración inicial bivalente. □

El lema anterior revela la esencia de la comunicación asíncrona, que después de empezar con la configuración inicial, por retraso de mensajes, el valor de consenso puede ser 0 o 1 (bivalente). De hecho sería más fácil de entender si la expresamos equivalentemente como:

$\mathcal{P}$ tiene una configuración inicial bivalente cuando potencialmente hay un proceso fallido.

LEMA 3. *Sea C una configuración bivalente de $\mathcal{P}$, y sea $e = (p, m)$ un evento aplicable a C. Sea $\mathscr{E}$ el conjunto de configuraciones accesibles desde C sin aplicar e, y sea $\mathscr{D} = e(\mathscr{E}) = \{e(E) \mid E \in \mathscr{E}$ y e es aplicable a $\mathrm{E}\}$. Entonces, $\mathscr{D}$ contiene una configuración bivalente.*

Demostración: Como e es aplicable a C, entonces, por definición de $\mathscr{E}$ y el hecho

de que los mensajes pueden retrasarse arbitrariamente, e es aplicable a cada $\mathscr{E}$.

Ahora suponga que $\mathscr{D}$ no contiene configuraciones bivalentes, por lo que cada configuración $D \in \mathscr{D}$ es univalente. Procedemos a derivar una contradicción.

Sea E_i una configuración i-valente accesible desde $C, i \in \{0,1\}$. (E_i se existe ya que C es bivalente). Si $E_i \in \mathscr{E}$, sea $F_i = e(E_i) \in \mathscr{D}$. De lo contrario, e se aplicó para llegar a E_i, por lo que existe $F_i \in \mathscr{D}$ desde el cual E_i es accesible. En cualquier caso, F_i es i-valente ya que F_i no es bivalente (ya que $F_i \in \mathscr{D}$ y $\mathscr{D}$ no contiene configuraciones bivalentes) y uno de E_i y F_i es accesible desde el otro. Por $F_i \in \mathscr{D}, i = 0, 1, \mathscr{D}$ contiene las configuraciones 0-valente y 1-valente.

Llamamos a dos configuraciones como *vecinos* si uno resulta desde otro en un solo paso. Se existen dos vecinos $C_0, C_1 \in \mathscr{E}$ tal que $D_i = e(C_i)$ es i-valente. Sin pérdida de generalidad, $C_1 = e'(C_0)$ donde $e' = (p', m')$:

Por la suposición, $\mathscr{D}$ no contiene una configuración bivalente, y $e(C) \in \mathscr{D}$, sea $e(C)$ i-valente, $i \in \{0,1\}$, como C es bivalente, y $\mathscr{D}$ solo contiene ambas configuraciones 0-valente y 1-valente, entonces existe una calendarización $\sigma_0 = (e_1, .., e_k)$ tal que $e(\sigma_0(C)) = e(e_k(..(e_1(C)))))))$ es j-valente, donde $j \in \{0,1\}$, $e \notin \{e_1, ..e_k\}$ y $j \neq i$, ya que $e(C)$, $e(e_1(C))$,..$e(e_k(..(e_1(C))))$ forman una cadena de configuraciones donde $e(C)$ es i-valente y $e(e_k(..(e_1(C))))$ es j-valente, por lo tanto, existe $e_l \in \{e_1, ..e_k\}$ tal que $e(e_{l-1}(..(e_1(C))))$ es i-valente, y $e(e_l(e_{l-1}(..(e_1(C)))))$ es j-valente. ya que $e' = e_l$, $C_0 = e_{l-1}(..(e_1(C)))$, $C_1 = e_l(e_{l-1}(..(e_1(C))))$.

Caso 1. Si $p' \neq p$, por Lema 1, $D_1 = e'(D_0)$, esto es imposible ya que cualquier sucesor de una configuración 0-valente es 0-valente. (Ver Figura 2.)

Caso 2. Si $p' = p$, entonces considere cualquiera corrida decisiva finita desde C_0 en la que p no tome pasos.

Sea σ la calendarización correspondiente, y sea $A = \sigma(C_0)$. Por Lema 1, σ es aplicable a D_i, y conduce a una configuración i-valente $E_i = \sigma(D_i), i = 0, 1$. También por Lema 1,$e(A) = E_0$ y $e(e'(A)) = E_1$. (Ver Figura 3.) Por lo tanto, A es bivalente. Pero esto es imposible ya que la corrida hacia A es decisiva (por hipótesis), por lo que A debe ser univalente.

En cada caso, llegamos a una contradicción, por lo tanto, $\mathscr{D}$ contiene una configuración bivalente. □

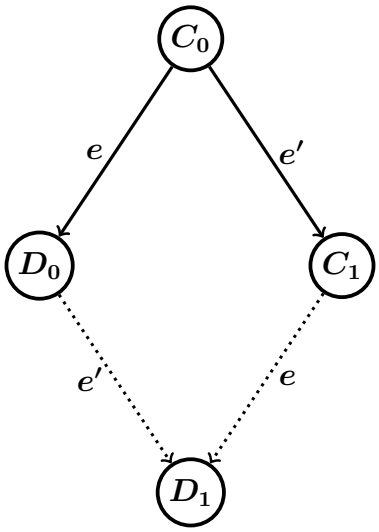

Figura 3

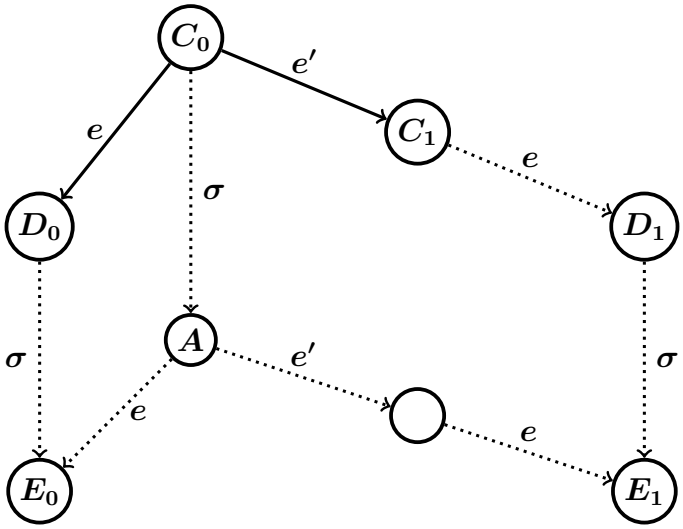

Figura 4

Estos tres lemas deducen el teorema de FLP: cualquiera corrida decisiva desde una configuración inicial bivalente pasa a una configuración univalente, por lo que debe haber un solo paso que vaya desde una configuración bivalente a una configuración univalente. Tal paso determina el valor de decisión eventual. Ahora mostramos que

siempre es posible ejecutar el sistema de una manera que evite tales pasos, lo que lleva a una corrida no decisiva admisible.

La corrida se construye en etapas, comenzando desde una configuración inicial. Nos aseguramos que la corrida sea admisible de la siguiente manera. Se mantiene una cola de procesos, inicialmente en un orden arbitrario, y el búfer de mensajes en una configuración se ordena de acuerdo con con el momento en que se enviaron los mensajes, el primero mensaje que envió sería el primero en el búfer. Cada etapa consiste de uno o más pasos de proceso. La etapa finaliza con el primer proceso en la cola del proceso, tomando un paso en el cual, si su cola de mensajes no estaba vacía al comienzo de la etapa, se recibe su primer mensaje. Este proceso se mueve al final de la cola del proceso. En cualquiera secuencia infinita de tales etapas, cada proceso toma infinitos pasos y recibe cada mensaje que se le envía. La corrida es por lo tanto admisible. El problema, por supuesto, es hacerlo de tal manera que se evite que se tome una decisión.

Sea C_0 una configuración inicial bivalente cuya existencia está garantizada por Lema 2. La ejecución comienza en C_0 y nos aseguramos de que cada etapa comience desde una configuración bivalente. Supongamos que la configuración C es bivalente y que el proceso p encabeza la cola de prioridad. Sea m el primer mensaje a p en el búfer de mensajes de C, si lo hay, y $\varnothing$ en caso contrario. Sea $e = (p, m)$. Según Lema 3, hay una configuración bivalente C accesible desde C mediante una calendarización en el que e es el último evento aplicado. La secuencia de pasos correspondiente define la etapa.

Dado que cada etapa termina en una configuración bivalente, cada etapa tiene éxito en la construcción de la calendarización infinito. La corrida resultante es admisible, y nunca se llega a una decisión. Se deduce que $\mathcal{P}$ no es totalmente correcto. □

3. Algoritmos de consenso clásicos

3.1. Introducción

En esta sección presentaremos dos algoritmos de consenso representativos, el algoritmo Paxos [6] y el algoritmo PBFT [8]. Todos son algoritmos basados en el modelo de máquina de estado replicada. El algoritmo Paxos solo es aplicable a *fallas de tipo paro*, y el algoritmo PBFT también es aplicable a *fallas bizantinas*.

La diferencia entre los dos también se refleja en la sincronía, el primero es un algoritmo síncrono, tanto para su *seguridad* y su *viveza*, y la *seguridad* de PBFT es asíncrona, pero su viveza es síncrona.

3.2. Algoritmo Paxos

El algoritmo Paxos fue publicado por Leslie Lamport a finales de la década de los 90, para resolver el problema de consenso de un sistema distribuido basado el modelo de comunicación de envío de mensajes. Este modelo es asíncrono parcial y sin fallas Bizantinas.

Los procesos pueden ser fallidos, lentos, ó reiniciar después de una falla, los mensajes pueden ser perdidos, retrasados, ó repetidos, pero no van a ser saboteados. El algoritmo Paxos puede garantizar la corrección del valor de los nodos que quieren consensuar, si siempre hay una mayoría (vea la definición a continuación) de nodos correctos (no se detectan los nodos fallidos).

3.2.1. Trasfondo del algoritmo Paxos

Para describir el algoritmo Paxos, Lamport ha utilizado una ciudad-estado griega llamada Paxos. La isla formula leyes de acuerdo con el modelo político de la democracia parlamentaria, pero nadie quiere dedicar todo su tiempo y energía a tales cosas. Por lo tanto, ni el parlamentario, ni el orador ni el asistente que entregaron la nota pueden prometer que alguien aparecerá cuando lo necesiten, ni pueden prometer el momento de aprobar la propuesta o entregar un mensaje. Pero aquí se supone que no hay fallas Bizantinas (es decir, aunque un mensaje se puede entregar dos veces ó con retraso, pero nunca habrá un mensaje falsificado), siempre que se espere el tiempo suficiente, el mensaje se entregará. Además, los parlamentarios de la isla de Paxos no se opondrán a las propuestas de otros parlamentarios.

En correspondencia con el sistema distribuido, los parlamentarios corresponden a cada nodo, y las leyes formuladas corresponden al estado del sistema. Cada nodo necesita entrar en un estado consistente. El requisito de coherencia corresponde a que solo puede haber una versión de las disposiciones legales. La incertidumbre de los congresistas corresponde a la falta de fiabilidad de los nodos y canales de mensajería.

3.2.2. Conceptos Básicos

Una *mayoría* por lo menos $\lfloor \frac{n}{2} \rfloor + 1$, dónde n es el número de los nodos del sistema.

El algoritmo especifica tres roles:

1. *Proponente*, que propone las propuestas; puede haber varios.
2. *Aceptador*, que recibe las propuestas y puede aceptarlas bajo algunas condiciones.
3. *Aprendiz*, que aprende las propuestas aceptadas (de los otro nodos).

Se permite a un nodo tener más de un rol.

Una propuesta es *elegida* cuando haya una mayoría de aceptadores que la aceptan; tiene dos campos:

1. *Valor*, este valor puede ser cualquier dato (codificado), por ejemplo, una operación *Incrementa la variable x por 1.*
2. *Número de propuesta*, se puede entender como la "versión" de la propuesta; las propuestas distintas tienen que tener números de propuesta distintos.

Con los roles y definiciones ya se puede definir el algoritmo, para obtener el algoritmo, se propone tres requisitos:

$R1$. Un *valor* solo puede ser elegido después de ser propuesto por los proponentes.

$R2$. Solo un *valor* puede ser elegido en una ronda de ejecución de Paxos.

$R3$. Los aprendices solo aprender un *valor* elegido.

El algoritmo fue obtenido mediante la satisfacción de los requisitos (especialmente $R2$) en los escenarios del sistema síncrono.

3.2.3. Descripción General del Algoritmo Paxos

El algoritmo se puede dividir en dos etapas.

1. Etapa de *preparación*

   a) El proponente elige un número de propuesta m y envía la solicitud de preparación a los aceptadores.

   b) Después de que el aceptador reciba el mensaje de preparación, si el número de propuesta es mayor que todos los mensajes de preparación a los que ya ha respondido (el mensaje de respuesta indica aceptación), el aceptador responderá a la propuesta que aceptó la última vez y se comprometerá a no responder a propuestas con numeraciones menores de m.

2. Etapa de *aprobación*:

   a) Cuando una propuesta recibe una respuesta de una mayoría de los aceptadores para prepararse, entra en la etapa de aprobación. Envía una solicitud de aceptación a los aceptadores que responden a la solicitud de preparación, incluido el número n y un valor v, donde v es el valor de la propuesta con el número más alto entre las respuestas, o es cualquier valor si las respuestas no informaron propuestas.

   b) En la premisa de no violar su compromiso con otras propuestas, el aceptador aprueba la solicitud después de recibir la solicitud de aceptación.

3.2.4. Un Ejemplo

Vamos a dar un ejemplo muy simple (figura 5), bajo en un escenario que solo hay un proponente, tres aceptadores, sin aprendiz.

Cada aceptador tiene una variable local x, y el objetivo es que todos los aceptadores escriban el mismo valor en ella.

- Etapa 1: El proponente manda $prepara(x, m)$ a los tres aceptadores, diciendo que va a escribir la variable x, el número de propuesta del proponente es m.
- Etapa 1.2: Los aceptadores comparan los mensajes recibidos con sus propio contenido guardado, descubre que no tienen esta variable, tampoco han recibido las propuestas, después mandan mensajes al proponente, y recuerdan la variable x y *prepara* internamente.

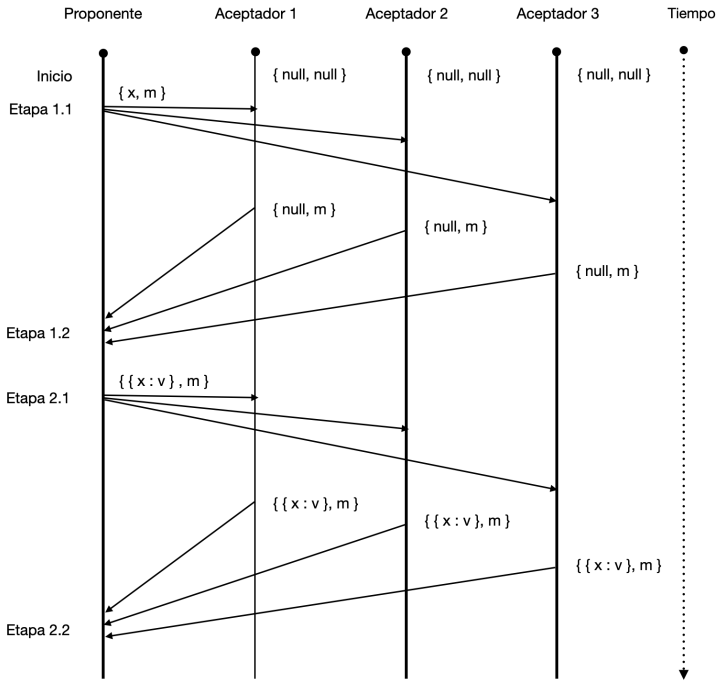

Figura 5: Elegir un valor

- Etapa 2.1: El proponente recibe los mensajes de los tres aceptadores, diciendo: x *no esta asignado con un valor, puede escribirla.* El proponente confirma que hay una mayoría de los aceptadores que están acuerdos de la escritura de x, empieza mandar la propuesta Pr_m con la operación de escribir $acepta(x, v, m)$, que asigna v a x.
- Etapa 2.2: Los aceptadores reciben la propuesta Pr_m con operación $acepta(x, v, m)$, se comparan el número de propuesta de Pr_m con el numero de solicitud *prepara* que recibieron anteriormente que son iguales, asignan x con v, confirman con $aceptado(x, v, m)$.
- Etapa final: El proponente recibe una mayoría de mensajes de confirmación, ya que v está elegido para x.

3.2.5. Aprender El Valor Elegido

Cuando un valor se ha elegido, los aceptadores deben avisarleslo a los aprendices. La manera más intuitiva de hacer esto es mandando mensajes a todos los aprendices, pero este método conducirá a un volumen excesivo de mensajes.

Suponiendo que no haya fallas Bizantinas, los aprendices pueden obtener el valor

que ha ido aprobado por otros aprendices. Por lo tanto, los aceptadores solo necesitan enviar el mensaje de aprobación a un aprendiz específico, y otros aprendices solicitan el valor que se ha aprobado. Este método reduce la cantidad de mensajes, pero la falla del aprendiz especificado hará que el sistema falle.

Por lo tanto, los aceptadores deben enviar mensajes a un subconjunto de aprendices y luego esos aprendices notifican a todos los aprendices.

Sin embargo, debido a la incertidumbre de los mensajes, puede que no haya noticias de que el aprendiz haya sido aprobado por la resolución. Cuando los aprendices necesitan saber el estado de aprobación de una propuesta, pueden solicitar un proponente para volver a hacer una nueva propuesta. Tenga en cuenta que un aprendiz también puede servir como un proponente.

3.2.6. Progreso

De acuerdo con el proceso anterior, cuando un proponente encuentra que existe una propuesta con un número mayor, finalizará la propuesta. Esto significa que una propuesta con un número mayor terminará el proceso de propuesta anterior. Si ambas propuestas en este caso recurren a una propuesta con un número mayor, pueden caer en un bloqueo mutuo, violando los requisitos de progreso.

La solución en este caso es elegir un líder y solo permitir que el líder haga propuestas. El teorema de imposibilidad de FLP implica que un algoritmo confiable para elegir un proponente debe utilizar la aleatoriedad o el tiempo real, por ejemplo, mediante el uso de tiempos de espera. Sin embargo, la seguridad está garantizada independientemente del éxito o el fracaso de la elección.

3.2.7. Implementar Una Maquina de Estado

Una manera simple para implementar un sistema distribuido es usar *MER* (Maquina de estado replicada), cada servidor se comporta como un *MER* y tiene las propiedades que hemos mencionado en la sección 1.2.

Para garantizar que todos los servidores ejecuten la misma secuencia de comandos de máquina de estado, implementamos una secuencia de instancias separadas del algoritmo de consenso de Paxos, el valor elegido por la i-ésima instancia es el comando de máquina de estado i-ésimo en la secuencia. Cada servidor desempeña todos los roles (proponente, aceptador y aprendiz) en cada instancia del algoritmo. Por ahora supongamos que el conjunto de servidores es fijo, por lo que todas las instancias del algoritmo de consenso usan los mismos conjuntos de agentes.

3.2.8. Optimizaciones

Supongamos que un aceptador recibe una solicitud de *preparación* numerada n, pero ya ha respondido a una solicitud de preparación numerada mayor que n, prometiendo que no aceptará ninguna nueva propuesta numerada n. Entonces no hay razón para que el aceptador responda a la nueva solicitud de preparación, ya que no aceptará la propuesta numerada n que el proponente desea emitir. Por lo tanto, hacemos que el aceptador ignore dicha solicitud de preparación. También tenemos que ignorar una solicitud de preparación para una propuesta que ya ha aceptado.

Cuando una propuesta encuentra que otro proponente ha hecho una propuesta con un número mayor, es necesario interrumpir el proceso. Por lo tanto, para optimizar, en el proceso de preparación anterior, si un aceptador encuentra que hay una propuesta con un número mayor, debe notificar el proponente de la propuesta para recordarle que interrumpa proponer su propuesta.

3.3. Algoritmo PBFT

3.3.1. Introducción

El algoritmo Tolerancia Práctica a Fallas Bizantinas (PBFT) [8] fue publicado por Castro M. y Liscov B. en el año 1999, para resolver el problema de la ineficiencia de los algoritmos precedentes tolerantes a fallas Bizantinas en los sistemas semi síncronos. Este fue el primer algoritmo factible para tolerar las fallas Bizantinas.

Los autores implementaron un servicio NFS (Network File System) tolerante a fallas Bizantinas utilizando su algoritmo y midieron su rendimiento. Los resultados mostraron que su servicio es solo un 3 % más lento que un NFS estándar no replicado.

El algoritmo garantiza *viveza* y *seguridad* proporcionadas como máximo $\lfloor \frac{n-1}{3} \rfloor$ de un total de n réplicas que son simultáneamente fallidas. Se utiliza una sola ronda de comunicación de ida y vuelta (*round-trip*) para ejecutar operaciones *solo de lectura* (read-only) y dos para ejecutar operaciones de *lectura* y *escritura* (read-write). Además, se utiliza un esquema de autenticación eficiente basado en códigos de autenticación de mensajes (*MAC*) durante la operación normal: criptografía de clave pública [10][11].

El artículo [8] tiene las siguientes contribuciones:

- Se describe el primer protocolo de replicación de máquina de estado que tolera correctamente fallas Bizantinas en los sistemas semi síncronos.

- Se describe una serie de optimizaciones importantes que permiten que el algoritmo funcione bien para que pueda usarse en sistemas reales.
- Se describe la implementación de un sistema de archivos distribuido tolerante a fallas Bizantinas.
- Proporciona resultados experimentales que cuantifican el costo de la técnica de replicación.

3.3.2. Modelo de Sistema

Asumimos un sistema distribuido donde los nodos están conectados por una red. La red puede fallar en la entrega de mensajes (puede ser que se pierdan pero solo un número acotado de mensajes se pierden), retrasarlos, duplicarlos o entregarlos desordenadamente. Usamos un modelo de falla Bizantina, los nodos fallidos pueden comportarse de manera arbitraria.

Usamos técnicas criptográficas para evitar falsificaciones y repeticiones y para detectar mensajes corruptos. Nuestros mensajes contienen las firmas de clave pública, los códigos de autenticación de mensajes y los resúmenes (digests) de los mensajes producidos por las funciones hash que son resistentes a colisiones. Todas las réplicas conocen las claves públicas de los demás para verificar las firmas.

3.3.3. Propiedades de Servicio

Se puede usar el algoritmo PBFT para implementar cualquier *servicio* replicado determinista con un *estado* y sus *operaciones*. Las operaciones no están limitados solo para las lecturas simples o las escrituras de las partes del estado del servicio; pueden realizar computaciones deterministas arbitrarias usando los argumentos del estado y de las operaciones. Los clientes envían peticiones al servicio replicado para invocar operaciones y se bloquea mientras esperan una respuesta. El servicio replicado está implementado por n réplicas.

Suponiendo que no más de $\lfloor \frac{n-1}{3} \rfloor$ réplicas sean fallidas, el algoritmo proporciona:

- *Seguridad* (safety): El servicio replicado satisface la *linealizabilidad* [9]: se comporta como una implementación centralizada que ejecuta operaciones atómicamente una a la vez. Se requiere una cantidad limitada de las réplicas fallidas porque una réplica fallida puede comportarse arbitrariamente.

- *Viveza* (liveness): Los clientes eventualmente reciben las respuestas de sus peticiones. El algoritmo no se basa en la sincronía para proporcionar seguridad. Por lo tanto, debe depender en la sincronía para proporcionar viveza; de lo contrario, podríamos usarlo para implementar el consenso en un sistema asincrónico, eso no es posible.

La seguridad está proporcionada independientemente de cuántos clientes fallidos estén utilizando el servicio (incluso si se confabulan con réplicas fallidas): todas las operaciones realizadas por clientes fallidos son observadas de manera consistente por clientes no fallidos. En particular, si las operaciones de servicio están diseñadas para preservar algunos invariantes en el estado de servicio, los clientes fallidos no pueden romper esos invariantes.

Garantizamos la viveza, si no más de $\lfloor \frac{n-1}{3} \rfloor$ réplicas sean fallidas y el *retraso*(t) no crezca más rápido que indefinidamente. Aquí, *retraso*(t) es el tiempo entre el momento t cuando se envía un mensaje por primera vez y el momento que es recibido por su destino (suponiendo que el receptor sigue retransmitiendo el mensaje hasta que el mensaje está recibido). Este es un supuesto de sincronía bastante débil que probablemente sea cierto en cualquier sistema real, dado que las fallas de cualquiera red estarán reparados eventualmente, así nos permite evitar el resultado de la imposibilidad en [9].

La resiliencia del algoritmo es óptimo: $3f + 1$ es el número mínimo de réplicas que permiten que un sistema asíncrono proporcione las propiedades de seguridad y viveza cuando fallan hasta f réplicas. Se necesitan tantas réplicas porque el sistema debe poder procederse después comunicarse con $n - f$ réplicas ya que las réplicas f pueden estar fallidas y no responder. Sin embargo, es posible que las réplicas f que no respondieron no sean fallidas y, por lo tanto, f de las que respondieron podrían estar fallidas. Aun así, debe haber suficientes respuestas para que las de las réplicas no fallidas superen en número a las de las fallidas, es decir, $n - 2f > f$. Por lo tanto $n > 3f$.

3.3.4. El Algoritmo

El algoritmo PBFT es una forma de replicación de máquina de estado, el servicio se modela como una máquina de estado que se replica en diferentes nodos en un sistema distribuido. Cada réplica de máquina de estado mantiene el estado del servicio e implementa las operaciones de servicio. Denotamos el conjunto de réplicas por $\mathcal{R}$ e identificamos cada réplica usando un número entero en $\{0, \ldots, |\mathcal{R}| - 1\}$, Por simplicidad, suponemos que $|\mathcal{R}| = 3f + 1$ donde f es el número máximo de réplicas que pueden ser fallidas. Aunque podría haber más de $3f + 1$ réplicas, pero las réplicas adicionales

degradan el rendimiento (dado que se intercambian más y más mensajes) sin proporcionar una capacidad de elasticidad mejorada. La razón de esta conclusión es porque el número máximo de réplicas que pueden ser fallidas no es un parámetro del algoritmo, es decir, una vez se establece este número ya no se debe de cambiarlo.

Las réplicas se mueven a través de una sucesión de configuraciones llamadas *vistas*. En una vista, una réplica es la *primaria* y las otras son respaldos. Las vistas están numeradas consecutivamente. La primaria de una vista es la réplica p tal que $p = v \bmod |\mathcal{R}|$, donde v es el número de la vista. Los cambios de vista se llevan a cabo cuando parece que la primaria ha fallado.

El algoritmo funciona a grandes rasgos de la siguiente manera:

1. Un cliente envía una petición para invocar una operación de servicio a la primaria.
2. La primaria multidifunde (multicasts) la petición a los respaldados.
3. Las réplicas ejecutan la petición y envían una respuesta al cliente.
4. El cliente espera respuestas de $f+1$ de diferentes réplicas con el mismo resultado, este es el resultado de la operación.

Imponemos dos requisitos por el uso de las técnicas de la máquina de estado replicada:

- Las replicas tienen que ser deterministas.
- Las replicas tienen que empezar desde mismo estado inicial.

3.3.5. Operación de Caso Normal

La Figura 2 muestra la operación del algoritmo en el caso normal sin fallas de la primaria. La replica 0 es la primaria y el sistema está en un período de sincronía, la replica 3 es fallida y C es el cliente.

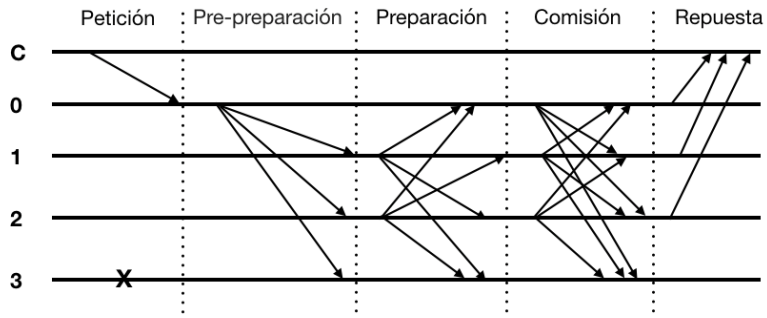

Figura 6: Operación de Caso Normal

3.3.6. Cambios de Vista

El protocolo *cambio de vista* proporciona viveza al permitir que el sistema progrese cuando falla la primaria. Los cambios de vista se activan mediante tiempos de espera (timeout) que evitan que los respaldos esperen indefinidamente a que se ejecuten las peticiones. Un respaldo está en el estado *está-esperando* para una petición si recibió una petición válida y no la ha ejecutado. Un respaldo inicia un temporizador (timer) cuando recibe una petición y el temporizador aún no se está corriendo. Detiene el temporizador cuando ya no está esperando ejecutar la petición, pero lo reinicia si en ese momento está esperando ejecutar otra petición.

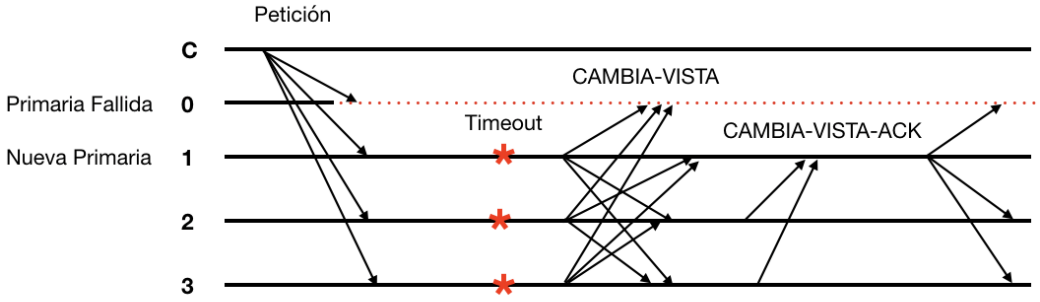

Figura 7: Cambio de Vista

4. La Blockchain

4.1. Historia

En el año 1983, el criptógrafo David Chaum publicó un sistema criptográfico monetario electrónico llamado eCash [13], e implementó DigiCash en 1995, que usaba la criptografía para que las transacciones sean anónimas. Aunque este sistema no era completamente descentralizado, permitió que la criptomoneda no fuera rastreable por el banco emisor, el gobierno o cualquier tercero.

En 1996, la NSA (Agencia de Seguridad Nacional) publicó una investigación titulada *How to make a Mint: the Cryptography of Anonymous Electronic Cash* que describía un sistema de criptomoneda[14].

En el año 1998, el concepto o idea de criptomoneda fue descrita por primera vez por Wei Dai [15], donde propuso la idea de crear un nuevo tipo de dinero descentralizado que usara la criptografía como medio de control.

En 2009, Satoshi Nakamoto implementó por primera vez en la práctica una moneda descentralizada llamada Bitcoin [16], se logró administrar la propiedad a través de la criptografía de clave pública con un algoritmo de consenso conocido como "PoW" (proof of work) para realizar un seguimiento de quién posee las monedas. Las transacciones de Bitcoin están organizadas en una estructura de dato especial llamada Blockchain. Han aparecido otras criptomonedas después de Bitcoin, por ejemplo Namecoin (para descentralizar el sistema de nombres de dominio DNS [17], lo que haría muy difícil la censura de Internet), Peercoin, que utiliza un esquema híbrido PoW/PoS (proof of stake) .. etc.

En los últimos diez años la criptomoneda ha prosperado sin precedentes, la mayoría de las criptomonedas usan la tecnología de Blockchain directamente o indirectamente. La Blockchain está dando un gran impacto para en la área de economía, finanza, y seguridad digital. En este capítulo la vamos a discutir más detalladamente.

4.2. Blockchain de Bitcoin

Una criptomoneda descentralizada debe de poder evitar las transacciones:

1. Reversibles.
2. Con doble gastos.

Para implementar una criptomoneda que satisface esos dos requisitos, en lugar de usar los libros de contabilidades centralizados y privados como los bancos, se necesita un libro de contabilidad distinto que sea:

1. Público.
2. Descentralizado.
3. Inmodificable.

La blockchain de bitcoin funciona como un libro de contabilidad que cumple esas características, la cuál es una cadena de los bloques, cada bloque contiene un cierta cantidad de transacciones (la cantidad puede tener un rango pequeño de variación). Se empieza con un bloque "génesis", después cada bloque se encadena con su bloque anterior y su bloque posterior, formando una cadena de bloques.

Una red de nodos de comunicación que ejecutan el software de Bitcoin mantiene la blockchain. Las transacciones tienen la forma: *el pagador A envía b bticoins al pagador C*. Las transacciones se transmiten en la red usando el software de Bitcoin.

Los nodos de red pueden validar transacciones, agregarlas a su copia del libro de contabilidad y luego transmitir estas adiciones del libro de contabilidad a otros nodos. Para lograr una verificación independiente, cada nodo de la red almacena su propia copia de la blockchain. A intervalos variables de promedios de tiempo hasta cada 10 minutos, se crea un nuevo grupo de transacciones aceptadas llamado bloque, se agrega a la blockchain y se publica rápidamente en todos los nodos, sin requerir supervisión central. Esto permite que el software de bitcoin determine cuándo se gastó una cantidad de bitcoin en particular, lo cual es necesario para evitar el doble gasto. Un libro de contabilidad convencional registra las transferencias de facturas reales o pagarés que existen aparte de él, pero la blockchain es el único lugar donde se puede decir que existe bitcoins en forma de *salidas de transacciones no gastadas* (UTXO - unspent transaction output).

4.3. Transacciones

Una transacción es una transferencia de valor de Bitcoin que se transmite a la red y se recopila en bloques. Una transacción normalmente hace referencia a las salidas de transacciones anteriores como nuevas entradas de transacciones y dedica todos los valores de Bitcoin de entrada a nuevas salidas. Las transacciones no están encriptadas, por lo que es posible navegar y ver todas las transacciones recopiladas en un bloque.

Todas las transacciones son visibles en la blockchain y se pueden ver con un editor hexadecimal.

Desde un punto de vista técnico, el libro de contabilidad de una criptomoneda como Bitcoin se puede considerar como una máquina de estados, donde hay un "estado" que consiste en el estado de propiedad de todos los bitcoins existentes y una "función de transición de estado" que toma un estado y una transacción para generar un nuevo estado que es el resultado. En un sistema bancario estándar, por ejemplo, el estado es un balance general, una transacción es una solicitud para mover X de A a B, y la función de transición de estado reduce el valor en la cuenta de A en X y aumenta el valor en la cuenta de B en X. Si la cuenta de A tiene menos de X en primer lugar, la función de transición de estado devuelve un error. Por tanto, se puede definir formalmente:

$$APLICAR(S, TX) \rightarrow S' \text{ ó } ERROR$$

La figura 8 ilustra un ejemplo de 4 transacciones: A envía 100 BTC a C y C genera 50 BTC. C envía 101 BTC a D, y necesita enviarse a sí mismo algún cambio. D envía el 101 BTC a otra persona, pero aún no lo ha canjeado. Solo la salida de D y el cambio de C se pueden gastar en el estado actual (ignoramos el costo por realizar cada transacción para la simplificación, entonces la cantidad real de cada salida de la transacción es más pequeño que la cantidad puesta).

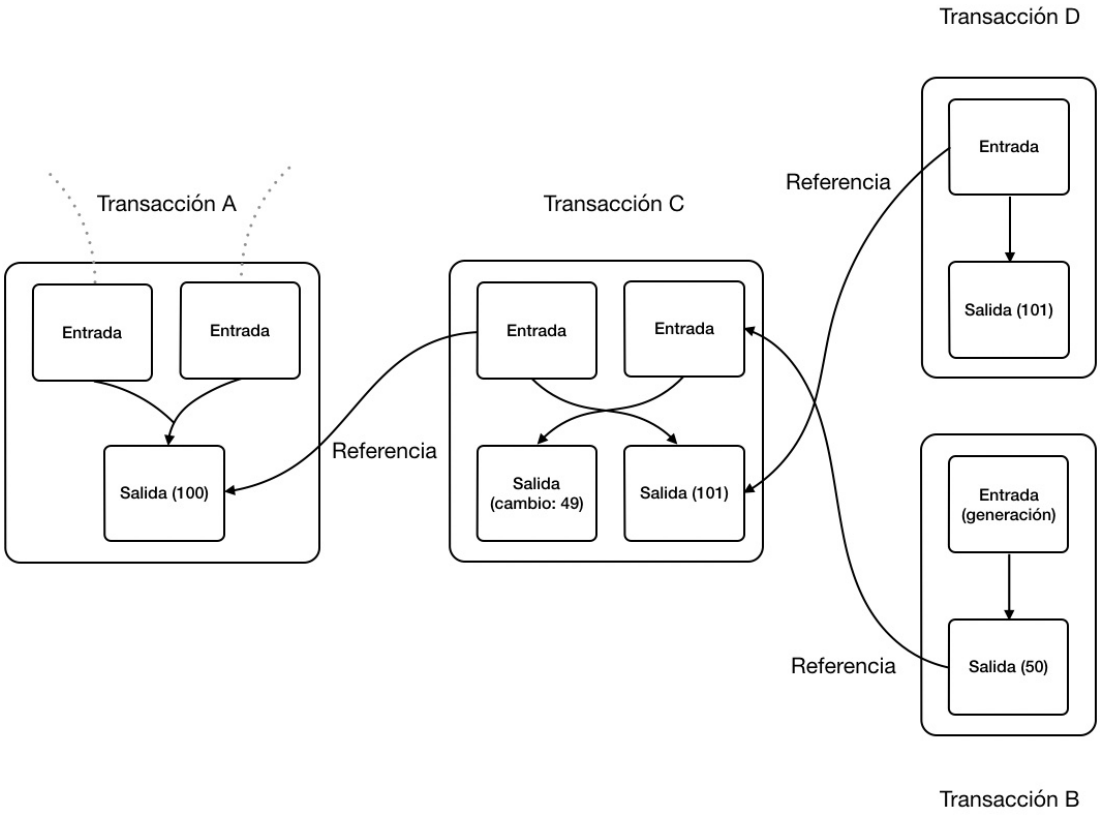

Figura 8: Transacciones

4.4. Árbol de Merkle

El árbol de Merkle ó también se conoce como árbol de hash, es un árbol binario que se construye mediante operación de hash (SHA256), sus hojas son los datos originales. En caso que el árbol es de un bloque de la blockchain, las hojas serían las transacciones del bloque.

El árbol de Merkle de un bloque se genera recursivamente por la siguiente regla:

$$PADRE = HASH(HIJO\_IZQ + HIJO\_DER)$$

En caso que el hijo derecho es nulo, la regla sería:

$$PADRE = HASH(HIJO\_IZQ + HIJO\_IZQ)$$

La figura 6 ilustra la generación del árbol de Merkle con 4 hojas.

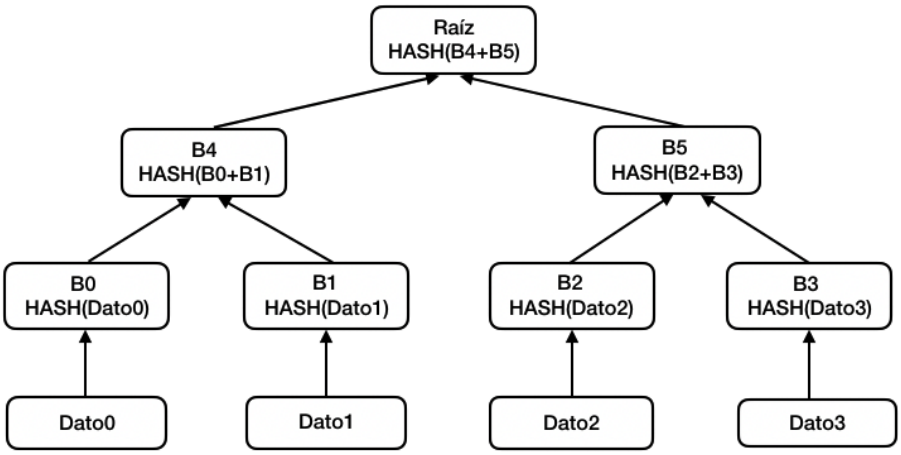

Figura 9: Generación de árbol de Merkle

El árbol de Merkle tiene tres características importantes:

1. Alta sensibilidad a cambio de datos: si una hoja fuera modificada, la raíz será totalmente distinta.

2. Rápidamente se puede ubicar una modificación: por ejemplo, en la figura 10, si los datos en Dato1 están modificados, afectarán a B1, B4 y la raíz. Cuando se encuentre que el valor hash del nodo raíz cambia, siga la ruta Raíz $\rightarrow$ B4 $\rightarrow$ B1 como máximo hasta el tiempo $O(\log_2 n)$ para ubicar rápidamente el tiempo real la transacción modificada Dato1.

3. Prueba con conocimiento cero: significa que un agente al que llamamos comprobador puede convencer a otro agente al que llamamos verificador de que una determinada afirmación es correcta sin proporcionar ninguna información útil al verificador. Por ejemplo, ¿cómo demostrar que alguien es propietario de Dato0 en la figura 10? El propietario crea un árbol de Merkle como se muestra en la figura 10, y luego anuncia B1, B5 y Raíz al exterior. En ese momento, el propietario de Dato0 genera B0 a través de Hash, luego genera B4 de acuerdo con el B1 anunciado, y luego genera raíz de acuerdo con el B5 anunciado; si el valor de hash de raíz generado es el mismo que el de raíz publicado, entonces el propietario ha podido probar la posesión de Dato0 sin necesidad de publicar los datos reales de Dato1, Dato2 y Dato3.

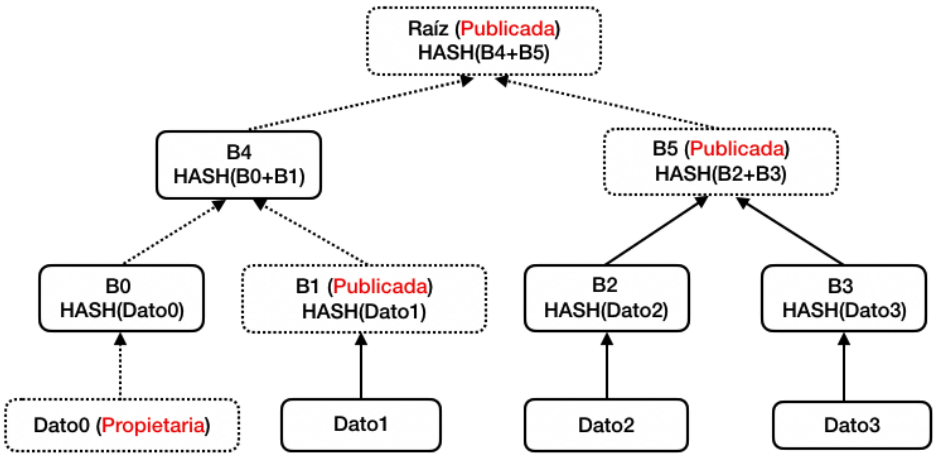

Figura 10: Verificar Existencia de Transacción

En Bitcoin, el árbol de Merkle se utiliza para resumir todas las transacciones en un bloque y, al mismo tiempo, generar una firma digital de todo el conjunto de transacciones, y proporciona una forma eficiente, llamada *ruta de Merkle* para verificar si existe una transacción en el bloque.

Suponiendo que hay 16 transacciones en un bloque, de acuerdo con la fórmula $O(\log_2 n)$ mencionada anteriormente, para encontrar cualquier transacción TX en este bloque, solo se necesita al máximo 4 veces de comparación, la *ruta de Merkle* de TX guardará 4 valores hash.

El uso del árbol de Merkle permite un nodo descargar solo el encabezado del bloque, y luego rastreando una ruta de longitud logarítmica de Merkle desde un nodo completo, se puede verificar la existencia de una transacción sin la necesidad de almacenar o transmitir una gran cantidad de bloques. La figura 11 ilustra la búsqueda de TX3 en la blockchain de Bitcoin.

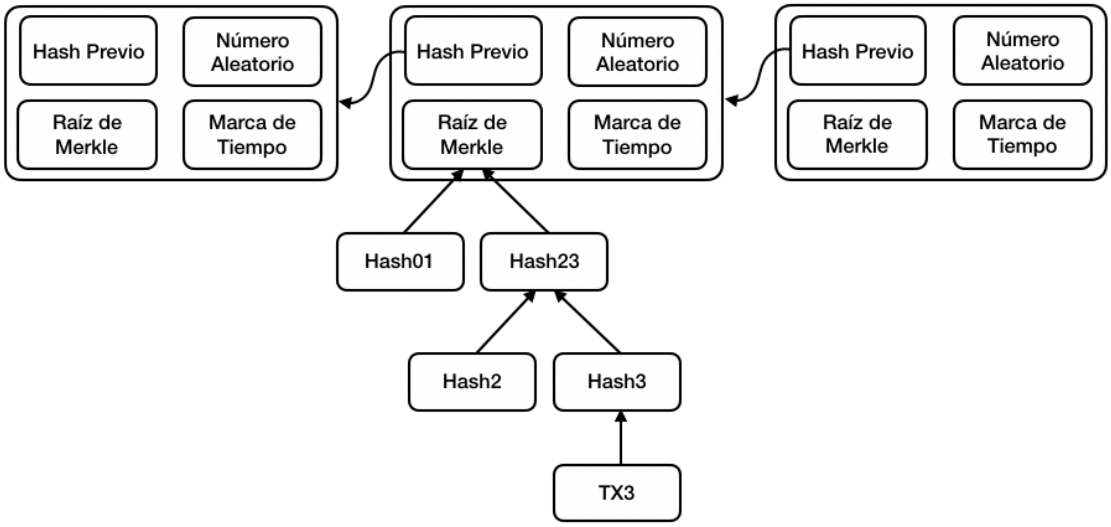

Figura 11: Ruta de Merkle

4.5. Minería y PoW

La minería es el proceso de agregar las transacciones a la blockchain de Bitcoin, que está diseñada intencionalmente para que sea difícil y requiera mucho recurso computacional, de modo que la cantidad de bloques que los mineros encuentran por día se mantenga estable. Los bloques individuales deben contener una prueba de trabajo (PoW) para ser considerados válidos. Esta prueba de trabajo será verificada por otros nodos mineros. Bitcoin utiliza el algoritmo *hashcash* [18] para la prueba de trabajo.

El propósito principal de la minería es establecer el historial de las transacciones de manera que sea computacionalmente impráctico de modificar por cualquier entidad. Al descargar y verificar la blockchain, los nodos pueden llegar a un consenso sobre el orden de los eventos en Bitcoin.

La minería de Bitcoin se llama así porque se asemeja a la minería de otras minas naturales: requiere esfuerzo y poco a poco obtener nuevas unidades a disposición de cualquiera que desee participar. Una diferencia importante es que el suministro de la minería de Bitcoin es más o menos estable porque el sistema de Bitcoin puede autorregular la dificultad de la minería.

Para generar un nuevo bloque, un minero puede empaquetar alrededor de 4000 transacciones, junto con el encabezado del bloque que incluye un número aleatorio (nonce), el hash del encabezado debe de comenzar con un cierto número de ceros, la probabilidad de que este hash comience con muchos ceros es muy baja, por lo que se deben hacer muchos intentos incrementando el nonce del encabezado. Una vez que el minero encuentra el nonce adecuado, se crea una transacción especial de la creación del bloque y difunde el nuevo bloque para que los otros nodos de Bitcoin lo verifiquen y lo agreguen a su blockchain local.

Cuando se descubre un nuevo bloque, el minero puede otorgarse una cierta cantidad de bitcoins mediante la transacción de la creación del bloque, que es acordada por la mayoría de los nodos de Bitcoin. Actualmente, esta recompensa es de 6.25 bitcoins, este valor se reducirá a la mitad cada 210,000 bloques.

Además, el minero recibe las tarifas que pagan los usuarios que envían transacciones. La tarifa es un incentivo para que el minero incluya la transacción en su bloque. En el futuro, a medida que disminuya la cantidad de nuevos bitcoins que los mineros puedan crear en cada bloque, las tarifas constituirán un porcentaje mucho más importante de los ingresos de los mineros.

4.6. Modelo Matemático de Blockchain

4.6.1. El Protocolo Backbone de Bitcoin (PBB)

La sección anterior hemos dado una presentación general de la blockchain, en esta sección, presentaremos su modelo matemático para dar un análisis solido. Este capítulo está basado en la tesis de *The Bitcoin Backbone* [21].

El protocolo Backbone de Bitcoin es ejecutado por los mineros que construyen una blockchain siguiendo el código fuente de Bitcoin[19] y permite a un conjunto de mineros mantenerla de forma distribuida. El protocolo está parametrizado por tres funciones externas $V(\cdot)$, $I(\cdot)$, $R(\cdot)$ que llamamos *predicado de validación de contenido*, *función de contribución de entrada* y *función de lectura en cadena*, respectivamente.

En un nivel alto, $V(\cdot)$ determina la estructura adecuada de la información que está almacenada en la blockchain, $I(\cdot)$ especifica cómo los mineros forman el contenido de los bloques, y $R(\cdot)$ determina cómo se supone que una blockchain debe interpretarse en el contexto de la aplicación.

Para poder analizar el protocolo Backbone de Bitcoin, adoptamos las siguientes suposiciones:

- La red de comunicación de los participantes es síncrona, además la cantidad de los participantes es fija.
- Los participantes no pueden autenticar el uno a otro y, por lo tanto, no hay forma de saber la fuente de un mensaje.
- Los mensajes eventualmente se entregan y todos los participantes en la red pueden sincronizar en el transcurso de una ronda.
- Todos los participantes involucrados tienen permitido el mismo número q de consultas a una función de hash en una sola ronda (llamamos este modelo como modelo “plano”).

La ultima suposición puede facilitar el análisis, de hecho los “mining pools” en Bitcoin pueden pensarse en tales agregaciones de participantes de modelo plano. El adversario mismo representa tal grupo ya que controla $t < n$ mineros; por esta razón, la cuota del adversario por ronda es $t \cdot q$ veces de consulta de hash.

En el análisis del protocolo Backbone de Bitcoin formalizamos y probamos dos propiedades que posee. Las propiedades se cuantifican mediante el parámetro f que

representa la probabilidad de un cálculo exitoso de PoW por un participante honesto durante una ronda de ejecución del protocolo.

La propiedad de prefijo común. Demostramos que si $\frac{t}{n-t}$ está adecuadamente acotado por debajo de 1, entonces $n-t$ blockchains mantenidas por los mineros honestos poseerán un gran prefijo común. Más específicamente, si un participante honesto "poda" (es decir, corta) k bloques del final de su cadena local, la probabilidad de que la cadena podada resultante no sea un prefijo de la cadena de otro participante honesto cae exponencialmente en el parámetro de seguridad.

La propiedad de calidad de la cadena. Demostramos que la proporción de bloques en la cadena de cualquier minero honesto que son contribuidos por los mineros maliciosos está delimitado por $\frac{t}{n-t}$.

Comenzamos presentando la notación de blockchain. Sean $G(\cdot)$, $H(\cdot)$ funciones hash criptográficas con salida en $\{0,1\}^\kappa$. Un bloque es cualquier triple de la forma $B = \langle s, x, ctr\rangle$ donde $s \in \{0,1\}^\kappa, x \in \{0,1\}^*$, ctr $\in \mathbb{N}$ son tales que satisfacen el predicado $\mathbf{validarbloque}_q^T(B)$ definido como

$$(H(ctr, G(s,x)) < T) \wedge (ctr \leq q)$$

El parámetro $T \in \mathbb{N}$ también se denomina *nivel de dificultad* del bloque. El parámetro $q \in \mathbb{N}$ es un límite que en la implementación de Bitcoin determina el tamaño del registro ctr, permitimos que esto sea arbitrario y lo usamos para indicar el número máximo permitido de consultas hash en una ronda. En nuestro análisis ctr está restringido al rango $0 \leq ctr < 2^{32}$ y q sea independiente de ctr.

Una blockchain es una secuencia de bloques. El bloque más a la derecha es la cabeza de la cadena, denominada head($\mathcal{C}$). Tenga en cuenta que la cadena vacía ε también es una cadena; por convención establecemos head(ε) = ε. Una cadena $\mathcal{C}$ con head($\mathcal{C}$) = $\langle s', x', ctr'\rangle$ se puede extender a una cadena más larga agregando un bloque válido $B = \langle s, x, ctr\rangle$ que satisfaga $s = H\left(ctr', G\left(s', x'\right)\right)$. En el caso $\mathcal{C} = \varepsilon$, por convención, cualquier bloque válido de la forma $\langle s, x, ctr\rangle$ puede extenderlo. En cualquier caso, tenemos una cadena extendida $\mathcal{C}_{\text{new}} = \mathcal{C}B$ que satisface cabeza head $(\mathcal{C}_{\text{new}}) = B$.

La longitud de una cadena len($\mathcal{C}$) es su número de bloques. Dada una cadena $\mathcal{C}$ que tiene longitud len($\mathcal{C}$) = $n > 0$ podemos definir un vector $\mathbf{x}_\mathcal{C} = \langle x_1, \ldots, x_n\rangle$ que contiene todos los valores de x que se almacenan en la cadena de modo que x_i es el valor del i-ésimo bloque.

Considere una cadena $\mathcal{C}$ de longitud m y cualquier entero no negativo k. Denotamos por $\mathcal{C}^{\lceil k}$ la cadena resultante de la "poda" de los k bloques más a la derecha. Tenga en cuenta que para $k \geq$ len($\mathcal{C}$), $\mathcal{C}^{\lceil k} = \varepsilon$. Si $\mathcal{C}_1$ es un prefijo de $\mathcal{C}_2$ escribimos $\mathcal{C}_1 \preceq \mathcal{C}_2$.

El protocolo Backbone de Bitcoin se especifica como algoritmo 4. Antes de descri-

birlo en detalle, primero presentamos los tres algoritmos de soporte del protocolo.

El primer algoritmo, llamado **validar**, realiza una validación de las propiedades estructurales de una cadena $\mathcal{C}$ dada. Se dan como entrada los valores q y T, una función hash $H(\cdot)$. Está parametrizado por el predicado de validación de contenido $V(\cdot)$. Para cada bloque de la cadena, el algoritmo comprueba que la prueba de trabajo esté debidamente resuelta, que el contador ctr no supere q y que el hash del bloque anterior esté debidamente incluido en el bloque. Además, recopila todas las entradas de los bloques de la cadena y las ensambla en un vector $\mathbf{X}_{\mathcal{C}}$. Si todos los bloques se verifican y $V(\mathbf{x}_{\mathcal{C}})$ es verdadero, entonces la cadena es válida; invalida de lo contrario. Deliberadamente dejamos el predicado $V(\cdot)$ indeterminado como mencionamos anteriormente.

El segundo algoritmo, llamado **maxvalidado**, es encontrar el "mejor posible" cadena cuando se le da un conjunto de cadenas. El algoritmo es sencillo y está parametrizado por una función $\max(\cdot)$ que aplica algún orden en el espacio de cadenas. El aspecto más importante es la longitud de las cadenas, en cuyo caso $\text{máx}(\mathcal{C}_1, \mathcal{C}_2)$ devolverá el más largo de los dos. En el caso de $\text{len}(\mathcal{C}_1) = \text{len}(\mathcal{C}_2)$, $\text{máx}(\cdot, \cdot)$ siempre devolverá el primer operando.

El tercer algoritmo llamado **pow**, toma como entrada una cadena e intenta extenderla resolviendo una prueba de trabajo. El algoritmo está parametrizado por dos funciones hash $H(\cdot)$, $G(\cdot)$ (que en nuestro análisis se modelarán como oráculos aleatorios), así como dos números enteros positivos q, T; q representa el número de veces que el algoritmo intentará aplicar fuerza bruta a la desigualdad de la función hash que determina la instancia de POW, y T determina la "dificultad" del POW. El algoritmo funciona de la siguiente manera: dada una cadena $\mathcal{C}$ y un valor x que se insertará en la cadena, aplica un hash a estos valores para obtener h e inicializa un contador ctr. Posteriormente, incrementa ctr y comprueba si $H(ctr, h) < T$; esta es la única invocación de $H(\cdot)$ que está sujeta al límite q. Si se encuentra un ctr adecuado, entonces el algoritmo logra resolver el POW y extiende la cadena $\mathcal{C}$ en un bloque insertando x así como ctr (que sirve como POW). Si no se encuentra un ctr adecuado, el algoritmo simplemente devuelve la cadena inalterada.

Dados los tres algoritmos anteriores, ahora estamos listos para describir el protocolo Backbone de Bitcoin como el cuarto algoritmo. Este es el protocolo que ejecutan los mineros y que se supone que se ejecuta "indefinidamente" (nuestro análisis de seguridad se aplicará cuando el tiempo total de ejecución sea polinomio en $\mathcal{C}$). Está parametrizado por dos funciones, la función de contribución de entrada $I(\cdot)$ y la función de lectura en cadena $R(\cdot)$, que se aplica a los valores almacenados en la cadena.

Cada minero comienza una ronda con una cadena local $\mathcal{C}$ (decimos que el minero tiene la cadena $\mathcal{C}$ en esta ronda) y verifica su cinta de comunicación RECIBIR() para ver si se ha recibido una cadena "mejor" y en tal caso la adopta la cadena $\tilde{\mathcal{C}}$ (decimos que el minero adopta la cadena $\tilde{\mathcal{C}}$ en esta ronda). La elección de la cadena $\tilde{\mathcal{C}}$ se realiza mediante

la función **maxvalidada**; tenga en cuenta que podría ser que $\mathcal{C} = \tilde{\mathcal{C}}$. Entonces, el minero intenta extender $\tilde{\mathcal{C}}$ ejecutando el algoritmo POW pow descrito anteriormente.

El valor que el minero intenta insertar en la cadena está determinado por la función $I(\cdot)$. La entrada a $I(\cdot)$ es el estado st, la cadena actual $\mathcal{C}$, el contenido de la cinta de entrada del minero ENTRADA() (recuerde que pueden ser escritos por el entorno $\mathcal{Z}$ al inicio de cualquier ronda) y la cinta de comunicación RECIBIR(), así como el número *ronda* de ronda actual. El protocolo espera dos tipos de entradas en la cinta de entrada, LEER y (INSERTAR, *valor*); otras entradas se ignoran.

Como mencionamos, dejamos intencionalmente las funciones $I(\cdot)$, $R(\cdot)$ indeterminadas en la descripción del protocolo Backbone, ya que sus características específicas variarán según la aplicación. Cuando la entrada x está determinada por $I(\cdot)$, el protocolo intenta insertarla en la cadena $\mathcal{C}$ invocando pow. En caso de que la cadena local $\mathcal{C}$ se modifique durante los pasos anteriores, el protocolo transmite ("difunde") la nueva cadena a las otras partes. Finalmente, en caso de que haya un símbolo de LEER en la cinta de comunicación, el protocolo aplica la función $R(\cdot)$ a su cadena actual y escribe el resultado en la cinta de salida SALIDA(). La ronda finaliza cuando el algoritmo difunde un mensaje ($\perp$ en caso de que no se difunda ningún mensaje).

Algoritmo 4. El protocolo Backbone de Bitcoin, parametrizado por la función de contribución de entrada $I(\cdot)$ y la función de lectura en cadena $R(\cdot)$. Al comienzo se asume que "inicio = True".

1: **if** (*inicio*) **then**
2: $\mathcal{C} \leftarrow \varepsilon$
3: $st \leftarrow \varepsilon$
4: $ronda \leftarrow 1$
5: $inicio \leftarrow$ False
6: **else**
7: $\tilde{\mathcal{C}} \leftarrow$ **maxvalidada**($\mathcal{C}$, cualquiera cadena $\mathcal{C}'$ encontrada en RECIBIR())
8: **if** ENTRADA() contiene LEER **then**
9: **escribe** $R(\tilde{\mathcal{C}})$ a SALIDA() ▷ Produce salida antes de POW .
10: **end if**
11: $\langle st, x\rangle \leftarrow I(\text{st}, \tilde{\mathcal{C}}, \text{ronda}, \text{ENTRADA}(), \text{RECIBIR}())$ ▷ Determina el x-valor.
12: $\mathcal{C}_{\text{nuevo}} \leftarrow \mathbf{pow}(x, \tilde{\mathcal{C}})$
13: **if** $\mathcal{C} \neq \mathcal{C}_{\text{nuevo}}$ **then**
14: $\mathcal{C} \leftarrow \mathcal{C}_{\text{nuevo}}$
15: DIFUNDIR($\mathcal{C}$) ▷ Difundir la cadena en caso de adopción/extensión.
16: **else**
17: DIFUNDIR($\perp$) ▷ Señala la finalización de la ronda a la funcionalidad difusa.
18: **end if**
19: $ronda \leftarrow ronda + 1$
20: **end if**

4.6.2. Análisis de PBB

Primero introducimos los parámetros en nuestro análisis. Enteros positivos n, t, L, s, ℓ, T, k, κ donde $\log T$ está relacionado linealmente con κ; reales positivos f, ϵ, δ, μ, τ, λ, donde $f, \epsilon, \delta, \mu \in (0, 1)$:

λ: parámetro de seguridad
κ: longitud de la salida de la función hash (asumimos $\kappa = \Omega(\lambda)$).
n: número de partes mineras; t de los cuales son controlados por el adversario
T: el valor hash objetivo utilizado por las partes para resolver los POWs
t: número de partidos controlados por el adversario
δ: ventaja de las fiestas honestas, $(t/(n-t) \leq 1-\delta)$

f: probabilidad de que al menos un partido honesto tenga éxito en encontrar un POW en una ronda
ϵ : calidad de concentración de variables aleatorias en ejecuciones típicas
k: número de bloques para la propiedad de prefijo común
ℓ: número de bloques para la propiedad de calidad de la cadena
μ: parámetro de calidad de la cadena
s: número de rondas para la propiedad de crecimiento de la cadena
τ: parámetro de crecimiento de la cadena
L: el tiempo de ejecución total del sistema

A continuación, definimos las dos propiedades principales del protocolo Backbone y las probamos. La primera propiedad se llama *propiedad de prefijo común* y está parametrizada por un valor $k \in \mathbb{N}$. Considera un entorno arbitrario y un adversario en el entorno de configuración de q-delimitado, y se mantiene mientras se eliminen k bloques de la cadena de un partido honesto y se obtenga un resultado prefijo de la cadena de otra partido honesto.

Propiedad de prefijo común. La propiedad de prefijo común Q_{cp} con parámetro $k \in \mathbb{N}$ establece que para cualquier par de mineros honestos P_1, P_2 adoptando las cadenas $\mathcal{C}_1, \mathcal{C}_2$ en las rondas $r_1 \leq r_2$ en $\text{VISTA}^{t,n}_{\Pi,\mathcal{A},\mathcal{Z}}$ respectivamente, se cumple que $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$.

La segunda propiedad, que llamamos *propiedad de calidad de la cadena*, tiene como objetivo expresar la cantidad de contribuciones de un minero honesto que están contenidas en un partido suficientemente largo y continua de la cadena de un minero honesto. Específicamente, para los parámetros $\ell \in \mathbb{N}$ y $\mu \in (0, 1)$, la tasa de bloqueo adversario contribuciones en una parte continua de la cadena de una parte honesta de longitud al menos l está acotado por $1 - \mu$.

Propiedad de calidad de la cadena. La propiedad de calidad de la cadena Q_{cq} con parámetros $\mu \in \mathbb{R}$ y $\ell \in \mathbb{N}$ establece que para cualquier partido honesto P con la cadena $\mathcal{C}$ in $\text{VISTA}^{t,n}_{\Pi,\mathcal{A},\mathcal{Z}}$, se cumple que para cualquier l bloques consecutivos de $\mathcal{C}$ la proporción de bloques honestos es al menos μ.

Y la tercera propiedad que conviene considerar como conjunción de las dos anteriores.

Propiedad de crecimiento de la cadena. La propiedad de crecimiento de la cadena Q_{cg} con parámetros $\tau \in \mathbb{R}$ y $s \in \mathbb{N}$ establece que para cualquier partido honesto P que tenga una cadena $\mathcal{C}$ en vista $\text{VISTA}^{t,n}_{\Pi,\mathcal{A},\mathcal{Z}}$, cumple que después de cualquiera s rondas consecutivas se adopta una cadena que es al menos $\tau \cdot s$ bloques más larga que $\mathcal{C}$.

Llamaremos exitosa a una consulta de una parte si envía un par (ctr, h) tal que $H(ctr, h) \leq T$ Para cada ronda $i, j \in [q]$ y $k \in [t]$, definimos las variables aleatorias

booleanas X_i, Y_i y Z_{ijk} de la siguiente manera. Si en la ronda i una parte honesta obtiene un POW, entonces $X_i = 1$, de lo contrario $X_i = 0$. Si en la ronda i exactamente una parte honesta obtiene un POW, entonces $Y_i = 1$, de lo contrario $Y_i = 0$. Respectando al adversario, si en la ronda i, la j-ésima consulta de la k-ésima parte corrupta es exitosa, entonces $Z_{ijk} = 1$, de lo contrario $Z_{ijk} = 0$. Defina también $Z_i = \sum_{k=1}^{t} \sum_{j=1}^{q} Z_{ijk}$. Para un conjunto de rondas S, sea $X(S) = \sum_{r \in S} X_r$ y defina de manera similar $Y(S)$ y $Y(S)$. Además, si $X_i = 1$, llamamos i una *ronda exitosa* y si $Y_i = 1$, una *ronda excepcionalmente exitosa*.

Suposición de Mayoría Honesto. Un número t de n partidos son corruptos de manera que $t \leq (1-\delta)(n-t)$, donde $3f + 3\epsilon < \delta \leq 1$. Observa que $3f + 3\epsilon < \delta$ implica $(1+\epsilon)(1+f) < (1-\epsilon)(1-f)(1+\delta) < (1-\epsilon)(1-f)/(1-\delta)$ and $f, \epsilon < \frac{1}{3}$.

Lema de crecimiento en cadena. *Suponga que en la ronda r una parte honesta tiene una cadena de longitud l. Entonces, por la ronda $s \geq r$, cada parte honesta ha adoptado una cadena de longitud al menos $\ell + \sum_{i=r}^{s-1} X_i$*

Este lema se demuestra mediante el método de inducción sobre $s - r \geq 0$ y suponiendo que cada parte honesta recibirá $\mathcal{C}$ en la ronda r.

Operaciones sobre blockchain Una *inserción* ocurre cuando, dada una cadena $\mathcal{C}$ con dos bloques consecutivos B y B', un bloque B^* creado después de B' es tal que B, B^*, B' forman tres bloques consecutivos de una cadena válida. Se produce una *copia* si el mismo bloque existe en dos posiciones diferentes. Se produce una *predicción* cuando un bloque extiende uno que se calculó en una ronda posterior.

Definición de Ejecución Típica. Una ejecución es (ϵ, λ)-*típica*, para $\epsilon \in (0,1)$ y entero $\lambda \geq 2/f$, si para cualquier conjunto S de al menos λ rondas consecutivas, se cumple lo siguiente:

(a) $(1-\epsilon)\mathbb{E}[X(S)] < X(S) < (1+\epsilon)\mathbb{E}[X(S)]$ y $(1-\epsilon)\mathbb{E}[Y(S)] < Y(S)$

(b) $Z(S) < \mathbb{E}[Z(S)] + \epsilon\mathbb{E}[X(S)]$

(c) No se produjeron inserciones, copias ni predicciones.

Teorema de Ejecución Típica. *Una ejecución es típica con probabilidad* $1 - e^{-\Omega\left(\epsilon^2 \lambda f\right)}$.

Las partes (a) y (b) se demuestra directamente por *ligado de Chernoff estándar*, y calculando la probabilidad de que sucede una colisión que es $\binom{L}{2} 2^{-\kappa+1} = e^{-\Omega(\kappa)}$ (donde L es la cantidad máxima de consulta de hash) para demostrar la parte (c).

Teorema de Crecimiento de La Cadena. *En una ejecución típica, la propiedad de*

crecimiento de la cadena se mantiene con los parámetros $\tau = (1-\epsilon)f$ and $s \geq \lambda$.

Demostración. Supongamos que en una ronda r y un partido honesto P tiene una cadena $\mathcal{C}$ de longitud ℓ. Según el Lema de crecimiento de la cadena, después de s rondas, P ha adoptado una cadena $\mathcal{C}'$ de longitud al menos $\ell + X(S)$, donde $S = \{i : r \leq i < r + s\}$. Por suposición, $|S| \geq \lambda$ y se aplica la Suposición de Mayoría Honesto, ya que en una ejecución típica, $X(S) > (1-\epsilon)f|S| = \tau \cdot s$. □

Lema de Prefijo Común. *Suponga una ejecución típica y considere dos cadenas* $\mathcal{C}_1$ y $\mathcal{C}_2$. Si $\mathcal{C}_1$ *es adoptado por una parte honesta en la ronda r, y* $\mathcal{C}_2$ *es adoptado por una parte honesta en ronda r o difundida en la ronda r y tiene* $\text{len}(\mathcal{C}_2) \geq \text{len}(\mathcal{C}_1)$, then $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C})_2$ and $\mathcal{C}_2^{\lceil k} \preceq \mathcal{C}_1$ for $k \geq 2\lambda f$.

Este lema se demuestra con método de prueba por contradicción y las propiedades de *ronda excepcionalmente exitosa.*

Las siguientes dos teoremas principales también son basados de ejecución típica:

Teorema de Prefijo Común. *En una ejecución típica, la propiedad del prefijo común se mantiene con el parámetro* $k \geq 2\lambda f$.

Teorema de Calidad de La cadena. *En una ejecución típica, la propiedad de calidad de la cadena se mantiene con los parámetros* $\ell \geq 2\lambda f$ *y* $\mu = 1 - \left(1 + \frac{\delta}{2}\right) \cdot \frac{t}{n-t} - \frac{\epsilon}{1-\epsilon} > 1 - \left(1 + \frac{\delta}{2}\right) \cdot \frac{t}{n-t} - \frac{\delta}{2}$.

4.6.3. Sugerencia de Nakmoto y Libro de Contabilidad de Transacciones Públicas

Comenzamos analizando la sugerencia de Nakamoto para resolver BA, describimos su solución (llámelo protocolo $\Pi_{\text{BA}}^{\text{nak}}$)) a través del protocolo Backbone especificando las funciones $V(\cdot), I(\cdot), R(\cdot)$ de forma adecuada:

1. **Predicado de validación de contenido** $V(\cdot)$: $V(\langle x_1, \ldots, x_n \rangle))$ es verdadero si y solo si se cumple que $v_1 = \ldots = v_n \in \{0,1\}, \rho_1, \ldots, \rho_n \in \{0,1\}^\kappa$ donde $x_i = \langle v_i, \rho_i \rangle$, o $n = 0$.

2. **Función de lectura en cadena** $R(\cdot)$ (parametrizada por k): Si $V(\mathbf{x}_\mathcal{C}) =$ Verdadero y $\text{len}(\mathcal{C}) > k$, el valor de $R(\mathcal{C})$ es el valor único v que está presente en cada bloque de $\mathcal{C}$, mientras que no está definido si $V(\mathbf{x}_\mathcal{C}) =$ Falso o $\text{len}(\mathcal{C}) \leq k$.

3. **Función de contribución de entrada** $I(\cdot)$: Si $\mathcal{C} = \emptyset$ y (INSERTAR, v) está en la cinta de entrada, entonces $I(st, \mathcal{C}, round, \text{ENTRADA}())$ es igual a $\langle v, \rho \rangle$ donde $\rho \in \{0,1\}^\kappa$ es un valor aleatorio; de lo contrario (es decir, el caso $\mathcal{C} \neq \emptyset$)),

es igual a $\langle v, \rho \rangle$ donde v es el valor único con $v \in \{0,1\}$ que está presente en $\mathcal{C}$ y $\rho \in \{0,1\}^{\kappa}$ es un valor aleatorio. El estado st siempre permanece ϵ.

Como mostramos ahora, el acuerdo se sigue fácilmente de la propiedad del prefijo común. De hecho, siempre que haya un prefijo común (independientemente de su longitud), se garantiza que cuando se defina $R(\cdot)$, todas los partidos honestos producirán el mismo resultado.

Lema 18 (Acuerdo). Bajo la Suposición de Mayoría Honesto, mantiene que $\Pi_{\mathrm{BA}}^{\mathrm{nak}}$ parametrizado con $k = \lceil 2f\lambda \rceil$ ejecutando para un número total de rondas $L \geq 2k/f$, satisface el acuerdo con probabilidad de al menos $1 - e^{-\Omega\left(\epsilon^2 f \lambda\right)}$.

Demostración: Primero tenga en cuenta que después de las rondas L, todas los partidos honestos tienen una cadena con más de k bloques. Esto se sigue de la propiedad de crecimiento de la cadena, ya que $\tau L \geq 2(1-\epsilon)k > k$ (donde $\epsilon < 1/2$ se sigue de la Suposición de Mayoría Honesto).

Observe que las cadenas contienen valores únicos (ignorando los nonces), por lo tanto, un desacuerdo entre partidos honestos implica que dos partidos tienen cadenas disjuntas. De la propiedad del prefijo común se deduce q ue dos cadenas de longitud mayor que k que son completamente disjuntas no existen en una ejecución típica. □

Por otro lado, es fácil ver que la validez no se puede garantizar con una probabilidad aplastante a menos que el poder de hash del adversario sea insignificante en comparación con los partidos honestos, es decir, t/n es insignificante. Esto se debe a que, en caso de que el adversario encuentre una solución primero, entonces cada partido honesto extenderá la solución del adversario y cambiará a la entrada del adversario, por lo tanto, abandonará la entrada original. Aunque uno todavía puede demostrar que la validez se puede garantizar con una probabilidad distinta de cero (y, por lo tanto, el protocolo falla bajo la suposición de mayoría honesto), $\Pi_{\mathrm{BA}}^{\mathrm{nak}}$ se queda corto en proporcionar una solución para BA.

El tesis original demostró que existe un protocolo $\Pi_{\mathrm{BA}}^{1/3}$ construido sobre el Backbone de Bitcoin resuelve BA binario con (1/3)-limitado adversario si cambiamos a siguientes dos condiciones:

1. Los partidos nunca abandonan su entrada original, sino que insisten en insertarla en la cadena de bloques.

2. Después de la ronda L, generan la mayoría de su prefijo local de k-longitud.

Ahora llegamos a la aplicación para la que se diseñó Backbone de Bitcoin: mantener

un libro de contabilidad de transacciones públicas.

Un libro de contabilidad de transacciones públicas se define con respecto a un conjunto de libros de contabilidad válidos $\mathcal{L}$ y un conjunto de transacciones válidas $\mathcal{T}$, cada uno de los cuales posee una prueba de membresía eficiente. Un libro de contabilidad $\mathbf{x} \in \mathcal{L}$ es un vector de secuencias de transacciones $\mathrm{tx} \in \mathcal{T}$. Cada transacción tx puede estar asociada con una o más cuentas, denominadas $c_1, c_2, \ldots$ etc.

Los partidos del protocolo Backbone, denominados *mineros* en el contexto de esta sección, procesan secuencias de transacciones de la forma $x = \mathrm{tx}_1 \ldots \mathrm{tx}_e$ que se supone que están incorporados en su cadena local $\mathcal{C}$. La entrada insertada en cada bloque de la cadena $\mathcal{C}$ es la secuencia x de transacciones. Por tanto, un libro de contabilidad es un vector de secuencias de transacciones $\langle x_1, \ldots, x_m \rangle$, y una cadena $\mathcal{C}$ de longitud m contiene el libro de contabilidad $\mathbf{x}_\mathcal{C} = \langle x_1, \ldots, x_m \rangle$ si la entrada del j-ésimo bloque en $\mathcal{C}$ es x_j. La posición de la transacción tx_j en el libro de contabilidad $\mathbf{x}_\mathcal{C}$ es el par (i, j) donde $x_i = \mathrm{tx}_1 \ldots \mathrm{tx}_e$.

Podemos determinar las tres funciones $V(\cdot), I(\cdot), R(\cdot)$ que convertirán el protocolo Backbone en Π_{LCP}, un protocolo que realiza un libro de contabilidad de transacciones públicas:

1. **Predicado de validación de contenido** $V(\cdot)$: $V\left(\langle x_1, \ldots, x_m \rangle\right)$ es verdadero si y solo si el vector $\langle x_1, \ldots, x_m \rangle$ es un libro de contabilidad válido, es decir, $\langle x_1, \ldots, x_m \rangle \in \mathcal{L}$.

2. **Función de lectura en cadena** $R(\cdot)$: $V\left(\langle x_1, \ldots, x_m \rangle\right) =$ Verdadero, el valor $R(\mathcal{C})$ es igual a $\langle x_1, \ldots, x_m \rangle$; indefinido de lo contrario.

3. **Función de contribución de entrada** $I(\cdot)$: $I(st, \mathcal{C}, \textit{round}, \mathrm{ENTRADA}())$ funciona de la siguiente manera: si la cinta de entrada contiene (INSERTAR, v), analiza v como una secuencia de transacciones y retiene la subsecuencia más grande $x' \preceq v$ que es válida con respecto a $\mathbf{x}_\mathcal{C}$ (y cuyas transacciones aún no están incluidas en $\mathbf{x}_\mathcal{C}$). Finalmente, $x = \mathrm{tx}_0 x'$ donde tx_0 es una transacción neutral con nonce aleatoria.

4.7. Nueva Generación de BlockChain

La Blockchain fue diseñada por Satoshi Nakamoto y el Bitcoin está construido sobre esta base y es el comienzo real de la digitalización y descentralización de la moneda. Ya han surgido muchas criptomonedas similares a Bitcoin, como Namecoin, Litecoin, Metacoin, etc. Pero estas criptomonedas no tuvieron muchas innovaciones

en comparación con Bitcoin. Hasta la aparición de Ethereum, su blockchain no solo registra transacciones y genera Ether, sino que también soporta el llamado contrato inteligente, que es su innovación más importante, porque los contratos inteligentes permiten a los usuarios crear reglas de transacción más complejas, y fortalecen la ecología de la Blockchain.

Ethereum tiene un lenguaje de programación de contrato inteligente, de hecho, el guión de Bitcoin también puede entenderse como un lenguaje de programación, pero es relativamente simple; igual que Ethereum, la ejecución del guión de Bitcoin sucede en su máquina virtual.

En comparación con Bitcoin, Ethereum introdujo los siguientes tres cambios principales:

Primero, Ethereum es *Turing completo*, la ventaja de Turing completo es que tiene la capacidad de expresión más fuerte. Por lo tanto, el grado de libertad de experimentación que proporciona Ethereum es mucho mayor que el de Bitcoin, razón por la cual existen tantos proyectos experimentales en Ethereum.

El segundo, Ethereum introdujo el mecanismo de *gas*, este mecanismo garantiza que exista un límite superior de pasos para cualquier ejecución. Esto no solo resuelve el problema de la parada (halting problem), sino también limita el desperdicio de recursos, por lo que el costo de ejecución del guión tiene una relación proporcional con el costo que va a pagar.

Tercero, los contratos en Ethereum tienen un estado explicito, el guión en Bitcoin no puede leer directamente el estado existente en la cadena. Además, la cadena de bloques de Ethereum adopta un árbol patricia en lugar de un árbol Merkle, que tiene la ventaja de reducir en gran medida el uso de almacén. Su blockchain también incluye los nodos tíos, eso mejora el mecanismo de incentivo de la minería.

5. Conclusión

A través de nuestra discusión anterior, podemos ver que el consenso es una de las partes más importantes de un sistema distribuido. La maquina de estado replicada es el concepto central del mecanismo de consenso.

Debido a la imposibilidad de FLP, todos los algoritmos introducidos en este artículo son soluciones bajo algunas condiciones limitadas, podemos observar que la aplicación de sistemas distribuidos se ha vuelto cada vez más extensa y tiene escenas más complejas. La criptomoneda ha llevado esta complejidad a una nueva altura en los últimos años.

Blockchain proporciona un nuevo mecanismo de consenso basado en la prueba de trabajo. A través de discusiones entre Bitcoin y Ethereum, ha demostrado la universalidad de su modelo, pero no es perfecto, mucho desperdicio de energía, por ejemplo.

Las nuevas tecnologías traerán nuevas ideas o desafíos al consenso. Si no hubiera Internet, es difícil imaginar el surgimiento de la criptomoneda, porque requiere una red descentralizada a gran escala como Internet. Ahora podemos imaginar el consenso en la era de la computadora cuántica, debido a su estado y modelo de cálculo, el mecanismo de consenso podrá ser muy diferente, tal vez la maquina replicada de estado ya no será su base.

El consenso está en ascenso.

Referencias

[1] George Coulouris; Jean Dollimore; Tim Kindberg; Gordon Blair, Distributed Systems: Concepts and Design (5rd Edition), p.678, ISBN 978-0-13-214301-1

[2] Fred B. Schneider; Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.

[3] Berman, Piotr; Juan A. Garay (1993). "Cloture Votes: n/4-resilient Distributed Consensus in t + 1 rounds". Theory of Computing Systems. 2. 26: 3–19.

[4] Aguilera, M. K. (2010). "Stumbling over Consensus Research: Misunderstandings and Issues". Replication. Lecture Notes in Computer Science. 5959. pp. 59–72.

[5] Leslie Lamport; Time, clocks, and the ordering of events in a distributed system, Communications of the ACM.

[6] Leslie Lamport; The Part-Time Parliament, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.

[7] Leslie Lamport; Paxos Made Simple, 01 Nov 2001.

[8] Castro Miguel, Liskov Barbara: Practical Byzantine fault tolerance[C], OSDI. 1999, 99(1999): 173-186.

[9] M. Herlihy and J. Wing. Axiomsfor Concurrent Objects. In ACM Symposium on Principles of Programming Languages, 1987.

[10] D. Malkhi and M. Reiter. A High-Throughput Secure Reliable Multicast Protocol. In Computer Security Foundations Workshop, 1996.

[11] M. Reiter. Secure Agreement Protocols. In ACM Conference on Computer and Communication Security, 1994.

[12] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[13] David Chaum (1982), "eCash", Archivada en www.hit.bme.hu/buttyan/courses/BMEVIHIM219/2009/Chaum.BlindSigForPayment.1982.PDF

[14] Law, Laurie; Sabett, Susan; Solinas, Jerry (11 January 1997). "How to Make a Mint: The Cryptography of Anonymous Electronic Cash". American University Law Review. 46 (4). Archived from the original on 12 January 2018. Retrieved 11 January 2018.

[15] Wei Dai (1998). "B-Money", Publicada en www.weidai.com/bmoney.txt

[16] Satoshi Nakamoto, 2008, "Bitcoin: A Peer-to-Peer Electronic Cash System"

[17] Loibl, Andreas. "Namecoin"

[18] Adam Back. Hashcash, May 1997. Publicada en www.cypherspace.org/hashcash/

[19] Satoshi Nakamoto. Bitcoin open source implementation of p2p currency. http://p2pfoundation.ning.com/forum/topics/bitcoin-open-source, February 2009.

[20] https://ethereum.github.io/yellowpaper/paper.pdf

[21] Juan A. Garay, Aggelos Kiayias, Nikos Leonardos, 2020, The Bitcoin Backbone Protocol: Analysis and Applications