Universidad Nacional de Rosario

Tesina de grado
para la obtención del grado de
Licenciado en Ciencias de la Computación

Hashchain: una implementación eficiente de Setchain construida sobre Tendermint

Autora:
Gabina Luz Bianchi

Director:
Dr. César Sánchez

Co-director:
Dr. Martín Ceresa

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

23 de septiembre de 2024

Agradecimientos

A Margarita, porque es muy difícil ser tan inteligente sin volverse un pelotudo. A mi familia, a mis amigas y a toda la gente de la LCC.

Resumen

Un aspecto clave en la adopción de tecnologías *blockchain* es el rendimiento, usualmente medido en términos de número de transacciones por segundo. En consecuencia, diversas técnicas que buscan mejorar el rendimiento de las blockchains están siendo estudiadas y desarrolladas. Las blockchains actuales requieren algoritmos de consenso que garanticen que las transacciones, empaquetadas en bloques, estén totalmente ordenadas. Dado que esta imposición de un orden total puede ser innecesaria para algunas aplicaciones, un enfoque prometedor para mejorar el rendimiento se basa en relajar esta exigencia.

Bajo esta idea subyacente nace *Setchain* [14], una estructura de datos concurrente que implementa conjuntos distribuidos que solo crecen (*distributed grown-only sets*). Setchain provee barreras o puntos de sincronización, llamados épocas (*epochs* en inglés), que dotan a la estructura de un orden parcial. Esto implica que las épocas en sí presentan un orden entre ellas, pero que entre los elementos pertenecientes a una misma época no se puede establecer un orden. De esta forma, relaja el requerimiento de orden total buscando lograr mayor rendimiento y escalabilidad.

Las Setchains se pueden usar para aquellas aplicaciones, como los registros digitales, en donde los elementos en la blockchain no necesitan estar ordenados, excepto a través de barreras ocasionales.

Distintos algoritmos distribuidos tolerantes a comportamientos bizantinos que implementan Setchain fueron propuestos, pero no existía al momento ninguna implementacion eficiente compatible con los requerimientos de una *aplicación del mundo real*.

En el presente trabajo se propone una familia de implementaciones de Setchain de mundo real y tolerantes a fallas bizantinas construidas sobre *Tendermint* [12]. Tendermint es una plataforma madura de aplicación de blockchain usada en distintos proyectos de blockchain, como Cosmos [32] o Tezos [26].

La familia de implementaciones que se presenta en este trabajo sigue un enfoque incremental, proveyendo diversas aproximaciones a la solución final y más compleja; consta de tres variantes. Se comienza por la solución más básica de Setchain, llamada Vanilla. Compresschain es una variante intermedia utilizando un algoritmo de compresión. La mayor contribución es Hashchain, la cual explota el poder de compresión de las funciones hash para reducir la comunicación necesaria durante difusión y consenso, comunicando un hash de tamaño fijo en lugar de cientos o miles de elementos. El precio a pagar es un algoritmo distribuido adicional para obtener el conjunto de elementos desde un hash.

0.1. Marco del trabajo

Este documento fue elaborado a partir del trabajo realizado en la pasantía de investigación en el Instituto Madrileño de Estudios Avanzados de Software (IMDEA Software) durante el período enero a julio de 2023. La tesina se desarrolló dentro del grupo de investigación de *síntesis reactiva y verificación en tiempo de ejecución* que allí se desempeña.

0.2. Organización del trabajo

La tesina se organiza en capítulos, cada uno de ellos contando con una pequeña sección inicial describiendo los objetivos del mismo, excepto aquellos en los cuales no se consideró necesario por ser de extensión muy breve. Se estructura según se describe a continuación. En el capítulo 1 se contextualiza el problema sobre el cual se trabaja, detallándolo y presentando brevemente el estado del arte. En el capítulo 2 se presentan los conceptos de las ciencias de la computación que se consideran pertienentes para el análisis y comprensión de las contribuciones de este trabajo. El capítulo 3 ofrece una descripción detallada de la estructura de datos Setchain, comentando su interfaz y las propiedades que satisface. El capítulo 4 aborda las tres implementaciones de Setchain, contribuciones originales de este trabajo. En el capítulo 5 se demuestra formalmente que las tres implementaciones propuestas cumplen las propiedades de Setchain. El capítulo 6 detalla cómo fue llevada a cabo la evaluación empírica de las nuevas implementaciones y se presentan los resultados obtenidos. En el capítulo 7 se discuten las conclusiones de este trabajo y se comentan posibles pasos futuros que se derivan de esta tesina. Finalmente, el capítulo 8 consta de un breve glosario con algunos conceptos utilizados en inglés a lo largo del trabajo, que no cuentan con traducciones adecuadas al español.

Índice general

Capítulo 1

Introducción

1.1. Objetivo del capítulo

En este capítulo se presenta el problema a abordar a lo largo del trabajo, junto con un breve marco teórico, analizando el estado del arte y comentando las principales contribuciones de esta tesina.

1.2. Historia

A lo largo de la historia, el término *bitácora* ha servido para referir a distintos objetos relacionados con la orientación, el orden y el registro. Por primera vez aparecida en un texto escrito en 1538, la palabra *bitácora* oficialmente refiere a un concepto de la navegación: *especie de armario inmediato al timón, en el cual se coloca la brújula* [17]. Desde ahí surge también el conocido *cuaderno de bitácora*, un libro en el que los marinos, durante sus guardias, registraban los datos de lo acontecido, y que se guardaba en el interior de la bitácora para preservarlo de los malos tiempos. El concepto evolucionó y hoy en día la palabra bitácora usualmente se utiliza para hablar de registros metodológicos de un suceso particular, ya sea un viaje, una construcción, etc. Naturalmente, también se transformó la forma en que estos registros se escriben y dónde se guardan, y existe hoy la noción de *bitácora virtual*. En esos casos, son las computadoras, las redes y los protocolos, los que permiten el acceso a esos registros mediante internet; la potencia de la metáfora garantizando que continúe siendo un concepto de la navegación.

En Ciencias de la Computación, las *bitácoras distribuidas* son un tipo de base de datos que se comparte alrededor de mútiples lugares, países o instituciones, y que típicamente es de acceso público. Los registros pueden ser guardados usando distintas estructuras, y solo pueden agregarse cuando los participantes logran un *quorum*. En contraste con un sistema centralizado, las bitácoras distribuidas no requieren un administrador central y, en consecuencia, no tienen un punto de falla central. Un caso particular de bitácora distribuida, en donde los registros se empaquetan en forma de bloques, es la *cadena de bloques* o *blockchain*.

Las blockchains tomaron popularidad con la implementación de Bitcoin [36], una tecnología propuesta por Nakamoto en 2009. Allí se presentó como un método para eliminar terceras partes confiables en sistemas de pago electrónico.

Las versiones más modernas de blockchains incorporan contratos inteligentes o *smart contracts* [40, 46], los cuales son programas de estado inmutable alojados en la blockchain. Dichos programas describen la funcionalidad de las transacciones, incluyendo

el intercambio de criptomonedas. Los contratos inteligentes permiten describir funcionalidades sofisticadas, habilitando diversas aplicaciones en finanzas descentralizadas[1], gobierno decentralizado, Web3, etc.

Conceptualmente, la blockchain es un *objeto distribuido* que contiene las transacciones realizadas en nombre de los usuarios, empaquetadas en bloques, y totalmente ordenadas [25, 24]. En entornos reales, el objeto blockchain es mantenido por múltiples servidores sin una autoridad central, usando *algoritmos de consenso* que son resilientes a los *ataques bizantinos*.

1.3. El problema

Actualmente, uno de los principales obstáculos para la adopción rápida y generalizada de las tecnologías blockchain es su límite en la *escalabilidad*, debido principalmente a los límites de *rendimiento* inerentes a los algoritmos de consenso bizantinos [43, 19].

En términos generales, la escalabilidad de un sistema es su habilidad para gestionar una cantidad creciente de trabajo. Usualmente se habla de escalabilidad *horizontal* cuando se añaden nodos de trabajo a un sistema, en contraposición a la escalabilidad *vertical* que refiere a aumentar los recursos de un nodo particular.

En el caso específico de la tecnología blockchain, la escalabilidad horizontal consiste en añadir nuevos nodos a la red que participen del protocolo de consenso. Por otro lado, un ejemplo de problema de escalabilidad vertical en este contexto es la creciente cantidad de datos que requiere almacenar un nodo en la blockchain para guardar transacciones desde el bloque más reciente hasta el bloque *génesis* (inicial).

En este trabajo nos enfocaremos en la escalabilidad de blockchains desde el punto de vista de su *rendimiento*, medido en términos del número de transacciones añadidas por segundo. Algunos números representativos de hoy en día ilustran el estado actual de situación. Al sistema Bitcoin [36] le toma 10 minutos o más confirmar transacciones, logrando un rendimiento máximo de 7 transacciones por segundo. Ethereum [46], una de las blockchains más populares, admite en promedio 20 transacciones por segundo. Estos números son muy bajos comparados con los que observamos en otros campos de pago convencionales, como PayPal que soporta hasta 200 transacciones por segundo, o Visa, que confirma y procesa 2000 transacciones por segundo en promedio [34]. Por lo tanto, el rendimiento de las blockchains es un aspecto clave en la adopción de esta tecnología.

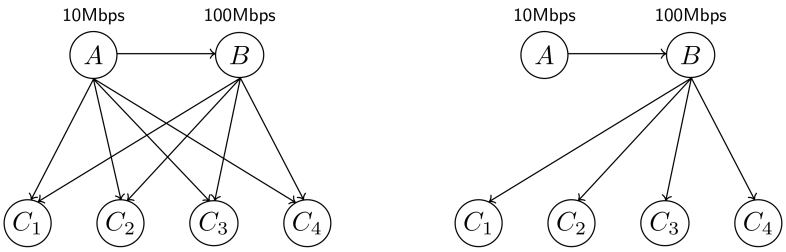
1.4. Estado del arte

Diversas técnicas están siendo desarrolladas para incrementar el rendimiento de las blockchains y aumentar la cantidad de transacciones añadidas por segundo. A continuación se exploran distintas propuestas, intentando abarcar una gama de enfoques novedosos.

1.4.1. Desarrollo de algoritmos más rápidos

En los sistemas blockchains existen múltiples parámetros que se tienen en cuenta a la hora de querer calcular o mejorar su rendimiento. FastChain [44] es un enfoque

[1] En diciembre de 2021, el valor monetario alojado en finanzas decentralizadas (o *DeFi*, abreviación de *decentralized finance*) estaba estimado en alrededor de $100B, de acuerdo a Statista `https://www.statista.com/statistics/1237821/defi-market-size-value-crypto-locked-usd/`.



(a) Política aleatoria. (b) Política de FastChain.

Figura 1.1: Selección de vecinos. Ejemplo motivador para FastChain

para mejorar la escalabilidad de los sistemas blockchains (originalmente diseñado para aquellos que utilizan *proof-of-work*[2]) que se basa en reducir el *tiempo de propagación de los bloques*. El algoritmo propuesto por FastChain trabaja sobre la red de pares de la blockchain.

En la Figura 1.1 se muestra un ejemplo motivador para la lógica detrás de FastChain. El nodo A tiene 10Mbps de ancho de banda. El nodo B tiene 100Mbps. Todos los enlaces tienen 200ms de latencia. La Figura 1.1a corresponde a una política de selección de vecinos aleatoria. Los nodos A y B tienen 5 vecinos. La Figura 1.1b muestra una política de selección de vecinos con información de ancho de banda. El nodo A se conecta únicamente con el nodo B, mientras que el nodo B se conecta a todos los otros. Supongamos que el nodo A mina un bloque de tamaño 10^6 bytes. En la Figura 1.1a, el ancho de banda del nodo A se reparte entre las 5 conexiones. Cada conexión tiene un ancho de banda de 2Mbps. Por lo tanto, a sus vecinos les toma 4.2s recibir el nuevo bloque. En la Figura 1.1b, el nodo A primero le transmite el bloque a B en 1s. El nodo B luego transmite el bloque al resto de los nodos en 0.52s. El tiempo de propagación de bloque promedio en la topología de FastChain es 1.416s, casi 3 veces menor que su contraparte para la topología de selección de vecinos aleatoria. El nodo A es el cuello de botella para la propagación de bloques. Mientras el nodo A está transmitiendo el bloque, otros nodos con mayor ancho de banda están ociosos.

FastChain propone un algoritmo de selección del mejor vecino para reducir el tiempo de propagación de los bloques. De esta forma, los nodos se desconectan de los vecinos con ancho de banda limitado y favorecen a los nodos con mayor ancho de banda.

1.4.2. *Inter-Blockchain Communication*

La tecnología blockchain está creciendo masivamente; el número de plataformas y aplicaciones decentralizadas se incrementó rápidamente en los últimos años. Sin embargo, la mayoría de las redes de blockchain operan en entornos autónomos aislados de los demás. La interoperabilidad de blockchains es la habilidad de conectar mútiples redes de blockchain entre sí, lo cual puede constituir un enfoque para mejorar la escalabilidad en las plataformas. De esta forma múltiples blockchains paralelas pueden interoperar, manteniendo las propiedades de seguridad de las mismas.

Una arquitectura novedosa de redes de blockchains es Cosmos [32], que conecta diversas blockchains independientes, llamadas *zonas*. Las zonas funcionan con Tendermint

[2]La noción de *proof-of-work* (PoW) refiere a situaciones en las que un *probador* demuestra a un *verificador* que ha realizado una cierta cantidad de trabajo computacional en un intervalo de tiempo especificado. Los PoWs han servido como base de diversos protocolos de seguridad en la literatura [27]. Su traducción al español de conoce como *prueba de trabajo*.

Core [12], un motor de consenso de alto rendimiento. El algoritmo de consenso del Tendermint Core es adecuado para escalar blockchains públicas que trabajan con modelos *proof-of-stake*[3]. Sin embargo, blockchains con modelos de consenso *proof-of-work*, como Bitcoin o Ethereum, se pueden conectar a la red de Cosmos utilizando adaptadores de zonas.

La primera zona de Cosmos se llama *Cosmos Hub*. Conecta varias blockchains (o zonas) mediante un protocolo de comunicación entre blockchains, conocido en inglés como *inter-blockchain communication* (IBC). Funciona como un protocolo TCP para blockchains[4]. Gestiona numerosos tipos de *tokens* y mantiene un registro del número total de los mismos en cada zona conectada. Los tokens pueden transferirse de una zona a otra de forma segura y rápida sin necesidad de un intercambio *líquido* entre zonas. En su lugar, todas las transferencias de tokens inter-blockchains pasan a través del Cosmos Hub. En la Figura 1.2 se ilustra un ejemplo simplificado de esta arquitectura.

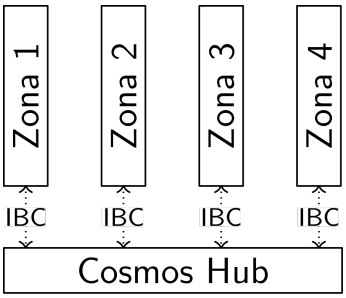


Figura 1.2: Arquitectura simplificada de Cosmos.

La interaoperabilidad de blockchains puede contribuir a la escalabilidad del sistema, debido a que la carga de procesamiento se distribuye, evitando cuellos de botella en una sola red. A su vez, al operar varias blockchains en paralelo, el número total de transacciones que pueden ser procesadas simultáneamente aumenta significativamente.

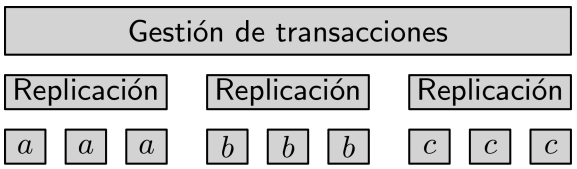
1.4.3. Paralelismo

Los sistemas de base de datos tradicionales logran escalabilidad dividiendo los estados de la base de datos en fragmentos independientes (o particiones). Al distribuir la carga sobre múltiples particiones, la capacidad global del sistema se incrementa. La técnica de *sharding* [21] requiere cierta coordinación para lograr propiedades básicas de atomocidad, consistencia, aislamiento y durabilidad para aquellas transacciones que acceden a múltiples fragmentos. En la Figura 1.3 se representa la idea de *sharding* como la combinación entre réplica y partición. Cada partición se replica sobre múltiples réplicas, y su contenido se mantiene consistente mediante protocolos de consenso.

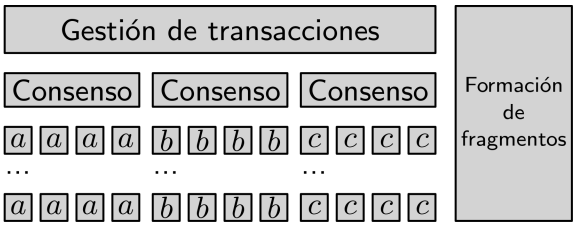
En el contexto de blockchains, el enfoque de *sharding* se basa en aplicar este concepto originario de la escalabilidad de bases de datos, dividiendo la red de blockchain en comités más pequeños de modo de reducir la sobrecarga de los protocolos de consenso.

[3]El concepto de *proof-of-stake* (traducido al español como *prueba de participación*) surge como oposición a los esquemas de prueba de trabajo. En estos tipos de protocolos, en lugar de probarse que se realizó un esfuerzo computacional lo suficientemente costoso, el *probador* demuestra la propiedad de algún bien valioso, en general, alguna cryptomoneda [31].

[4]El protocolo de control de transmisión conocido como TCP (proveniente de *Transmission Control Protocol*) es uno de los protocolos fundamentales de internet, perteneciente a la capa de transporte.



(a) Bases de datos distribuidas.



(b) Blockchains fragmentadas.

Figura 1.3: Protocolos de fragmentación.

Sin embargo, este concepto no puede ser directamente extendido a los sistemas de blockchain, debido a diferencias fundamentales en los modelos de fallas que se consideran en las bases de datos y en las blockchains. Tradicionalmente, las bases de datos asumen modelos de fallo por caída, en los cuales un nodo que falla simplemente deja de enviar y responder peticiones. Por el otro lado, los sistemas de blockchain operan en ambientes más hostiles, asumiendo modelos de fallas más fuertes, conocidos como bizantinos, que tienen en cuenta a atacantes malintencionados. La Figura 1.3 remarca las diferencias entre las bases de datos distribuidas y las blockchains fragmentadas, mostrando la necesidad de un protocolo de formación de particiones.

1.4.4. Capa 2

Los enfoques para incrementar el número de transacciones por segundo en las blockchains pueden ser categorizados según la *capa* que involucran. Las técnicas de capa 0 (*layer 0*) son aquellas que trabajan sobre la instraestructura de la red. Por ejemplo, apostando a decrementar la latencia de la misma. La capa 1 (*layer 1*) es la que cuenta con más propuestas en la literatura, abarcando todas aquellas que mejoran los algoritmos de consenso. Por último, la capa 2 (*layer 2*), también conocida como *off-chain* (fuera de la cadena), incluye aquellos protocolos que funcionan con una interacción mínima con la blockchain[29].

Existen diferentes enfoques dentro de la capa 2. Uno de los principales consiste en la computación *off-chain* de pruebas de *conocimiento cero* (*Zero-Knowledge proofs*) [10], que solo necesitan ser validadas dentro de la cadena. La adopción de funcionalidades limitadas (pero útiles) de canales como *Lightning* [37] es otra técnica perteneciente a la capa 2. Por último, tecnologías conocidas como *optimistic rollups* (por ejemplo, *Arbitrum* [30]) se basan en evitar ejecutar contratos inteligentes en los servidores, manteniendo la mínima sincronización requerida con la cadena para preservar las garantías de la blockchain (por ejemplo, cuando se necesitan anotar reclamos o resolver disputas).

1.4.5. Setchain

Las blockchains actuales requieren de algoritmos de consenso que garanticen que las transacciones, empaquetadas en bloques, estén totalmente ordenadas. Esta imposición de un orden total puede ser innecesaria para algunas aplicaciones.

Un enfoque prometedor para escalar la cantidad de transacciones agregadas por segundo es *Setchain* [14], un tipo de datos concurrente que implementa conjuntos de solo crecimiento distribuidos, proveyendo barreras o puntos de sincronización (llamados épocas). Setchain relaja el requerimiento de orden total y, por lo tanto, logra mayor rendimiento y escalabilidad. Las Setchains se pueden usar para aquellas aplicaciones, como los registros digitales, en donde los elementos en la blockchain no necesitan estar ordenados, excepto a través de barreras ocasionales.

Una descripción más detallada del funcionamiento de Setchain y sus propiedades se da en el capítulo 3.

1.5. Contribuciones

En este trabajo se realiza una contribución al estudio de la escalabilidad en blockchain mediante una familia de implementaciones robustas de Setchain. Si bien distintos algoritmos bizantinos distribuidos que implementaban Setchain fueron propuestos, no existía al momento ninguna implementacion eficiente compatible con los requerimientos de una aplicación del mundo real. Es decir, una aplicación capaz de soportar un *deployment* con miles de nodos[5] distribuidos geográficamente en todos los continentes, con clientes enviando peticiones para añadir nuevas transacciones a los servidores constantemente.

Las implementaciones prototípicas previas de Setchain[6] fueron construidas a partir de distintos componentes básicos: difusión confiable bizantina o *Bizantine Reliable Broadcast* [11, 38], difusión atómica bizantina o *Byzantine Atomic Broadcast* [22], conjuntos distribuidos que solo crecen bizantinos o *Byzantine Distributed Grow-only Sets* [16] y consenso bizantino de conjunto o *Set Byzantine Consensus* [18]. En este trabajo se toma un enfoque diferente, haciendo hincapié en la construcción de implementaciones que cumplan con la especificación de Setchain pero que al mismo tiempo sean compatibles con los requerimientos del mundo real. Es por eso que cada una de las soluciones originales propuestas en este trabajo está enteramente construida sobre la plataforma de aplicación de blockchain *Tendermint* [12].

Tendermint es una plataforma novedosa y popular para la replicación segura y consistente de aplicaciones en distintas máquinas. *Segura* en este contexto significa que Tendermint funciona incluso si a lo sumo un tercio de las máquinas falla de formas arbitrarias, brindando información conflictiva a las diferentes partes del sistema. *Consistente* refiere a que toda máquina correcta verá el mismo lote de transacciones y computará el mismo estado. Tendermint es un abordaje robusto de las blockchains implementado en *Go* [23], que presenta una separación clara entre las capas de bajo nivel de la blockchain, tales como un protocolo *gossip* y un algoritmo de consenso, y los conceptos de alto nivel relacionados a la estructura de datos que la blockchain mantiene[7].

[5] Si bien es difícil saber con exactitud cuántos nodos existen en la red de Blockcahin o Ethereum, se estima que este número varía entre aproximadamente 10000 y 50000 nodos.

[6] Presentadas en [14].

[7] Al momento de la realización de las implementaciones y la evaluación empírica de las mismas, Tendermint 0.34 era la versión de la plataforma vigente y más utilizada. Sin embargo, actualmente, la

La familia de implementaciones que se presenta en este trabajo sigue un enfoque incremental, proveyendo diversas aproximaciones a la solución final y más compleja; consta de tres variantes:

- *Vanilla*, una primera solución básica de Setchain, en donde cada elemento añadido a la Setchain se traduce como una transacción en la blockchain.
- *Compresschain*, una variante intermedia que usa un algoritmo de compresión, en la cual los elementos enviados por los clientes son agrupados en un lote, que se comprime antes de transmitirse como una transacción. Por lo tanto, las transacciones en la blockchain son lotes comprimidos de elementos.
- *Hashchain*, la contribución principal de este trabajo, una solución a Setchain usando funciones hash. Sigue una lógica similar a Compresschain, en donde los elementos se agrupan en lotes y se aplica una función hash antes de transmitirlos como una única transacción de tamaño fijo.

Hashchain explota el poder de compresión de las funciones hash para reducir la comunicación necesaria durante difusión y consenso, comunicando un hash de tamaño fijo en lugar de cientos o miles de elementos. El precio a pagar es un algoritmo distribuido adicional para obtener el conjunto de elementos desde un hash, garantizando tolerancia a servidores bizantinos.

La hipótesis fundamental en la que se basa Hashchain es que intercambiar latencia por un aumento en el ancho de banda resulta en una mejora significativa en el rendimiento. Es decir, hacer consenso sobre hashes de tamaño fijo conlleva un incremento en la cantidad de elementos añadidos a la Setchain por segundo, incluso si esto significara un aumento en la latencia, debido al algoritmo distribuido necesario para obtener el conjunto de elementos desde un hash.

versión más utilizada es un *fork* de Tendermint llamado *CometBFT* [3]. El desarrollo planteado a lo largo de este trabajo podría en principio portarse a la nueva versión sin demasiado esfuerzo.

Capítulo 2

Preliminares

2.1. Objetivo del capítulo

En este capítulo se presenta una breve descripción de los conceptos principales necesarios para la comprensión y discusión sobre las contribuciones de este trabajo.

2.2. Sistemas distribuidos: donde todo comenzó

La computación distribuida nació a fines de la década de 1970 cuando investigadores y profesionales comenzaron a tener en cuenta las características intrínsecas de los sistemas distribuidos físicamente. El campo luego emergió como un área de investigación especializada distinta al de redes, al de sistemas operativos, y al de computación paralela [38].

La computación distribuida surge cuando se tiene que resolver un problema en términos de entidades distribuidas (usualmente llamadas procesos, nodos, actores, agentes, pares, etc) en donde cada entidad tiene conocimiento parcial sobre los diversos parámetros involucrados en el problema que se tiene que resolver. Es usual, en este contexto, hacer hincapié en dos tipos distintos fundamentales de procesos: los *clientes* (o procesos clientes) y los *servidores* (o procesos servidores). Así, los servidores son aquellos que proveen determinado recurso o servicio, mientras que los clientes son quienes los demandan.

Es clave notar que el hecho de que las entidades (y la información que consume cada una de ellas) estén distribuidas no es una propiedad que se encuentre bajo el control de los programadores, sino que es una imposición dada por las características propias del problema. En la Figura 2.1 se expresa esto mediante un punto de vista de la arquitectura del sistema, en donde cada par (p_i, e_i) denota una entidad informática p_i y su entrada asociada e_i.

Como se ilustra en la Figura 2.1, un sistema distribuido está hecho de una colección de unidades de computación (llamadas también unidades informáticas) distribuidas, cada una abstraída a través de la noción de un proceso, interconectada a través de un medio de comunicación. Se asume que los procesos cooperan por un objetivo común, lo cual significa que intercambian información de una manera u otra. Es interesante notar que el hecho de que las entidades cooperen no significa la asunción de que todas las entidades se comportarán de la forma acordada, ya sea por un error (por ejemplo, en la programación o en una red) o por un comportamiento malintencionado. Se profundizará en este punto en la siguiente sección.

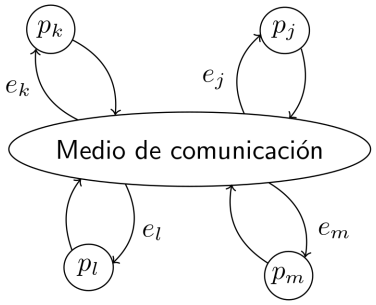


Figura 2.1: Estructura básica de la computación distribuida

2.2.1. Liveness y safety

Existen dos grandes clases de propiedades que son interesantes de enunciar cuando se describen sistemas (sean distribuidos o no): las propiedades de *liveness* y las propiedades de *safety*. Establecer estas propiedades ayuda principalmente en dos aspectos. Por un lado, al entendimiento del sistema. Por el otro, facilita la demostración de correctitud de los algoritmos que pretenden implementarlo. A lo largo de este trabajo veremos distintas propiedades que pueden ser clasificadas dentro de uno de estos dos conjuntos. Usualmente estas propiedades se definen en términos de las ejecuciones de un sistema o de un programa.

Informalmente, las propiedades de liveness son aquellas que establecen que *cosas buenas* ocurren en cada ejecución o, equivalentemente, describen algo que debe obligatoriamente pasar durante cada ejecución. Una vez satisfechas, lo son en cualquier extensión de la ejecución.

Por su parte, las propiedades de safety son aquellas que proscriben *cosas malas* durante la ejecución, caracterizando lo que está permitido al declarar lo que está prohibido. Una vez violadas, lo son en cualquier extensión de la ejecución.

2.3. Algoritmos de consenso

El problema de consenso es uno de los más celebrados en los sistemas de computación distribuida tolerante a fallas. Abstrae diversos problemas en donde, de una manera u otra, los procesos deben ponerse de acuerdo.

Los problemas de consenso requieren acuerdo entre un número de procesos para determinar un valor particular. Algunos de los procesos pueden fallar o ser no confiables de diversas maneras, por lo que los protocolos de consenso deben ser resilientes a fallas. Los procesos deben de alguna forma establecer sus valores candidatos, comunicándose entre ellos, y poniéndose de acuerdo en un valor específico. Los protocolos que resuelven el problema de consenso son diseñados para lidiar con un número limitado de procesos que fallan. Estos protocolos deben satisfacer ciertos requerimientos para ser utilizados. A lo largo de la historia se han desarrollado distintos algoritmos de consenso, que funcionan bajo diferentes escenarios (cantidad de nodos, cantidad de nodos que pueden fallar, formas en las que pueden fallar, etc.)

Si bien a lo largo de este trabajo no se entrará en detalles sobre ningún algoritmo de consenso en particular, es pertienente discutir conceptos y nomenclatura básica.

2.3.1. El problema de los generales bizantinos

Los sistemas de computación confiables deben lidiar con componentes que, por mal funcionamiento, dan información conflictiva a diferentes partes del sistema. La situación puede ser expresada abstractamente en términos de un grupo de generales de la armada bizantina que acampan con sus tropas alrededor de una ciudad enemiga. Comunicándose solo a través de mensajeros, los generales deben ponerse de acuerdo en un plan de batalla común. Sin embargo, uno o más de ellos pueden ser traidores, quienes tratarán de confundir a los otros. El problema es encontrar un algoritmo que asegure que los generales leales lleguen a un acuerdo. Está demostrado que, usando solo mensajes orales, este problema es resoluble si y solo si más de dos tercios de los generales son leales [33].

Con esta analogía de los generales bizantinos se instauraron nuevos conceptos que hoy son ampliamente utilizados. La habilidad de tolerar máquinas que fallan de formas arbitrarias, incluyendo aquellas maliciosas, es conocida como *tolerancia a fallas bizantinas*. A su vez, se inició el estudio de los algoritmos de consenso tolerantes a fallas bizantinas (*Byzantine fault-tolerant algorithms*), también conocidos como algoritmos de consenso BFT (por sus siglas en inglés). Si bien la teoría de BFT es de décadas pasadas, las implementaciones de software se han vuelto populares recientemente, debido al amplio éxito de tecnologías blockchains como Bitcoin [36] y Ethereum [46].

2.4. Firmas digitales

Usualmente, los protocolos BFT hacen un uso muy fuerte de la criptografía, fundamentalmente para autenticar mensajes y decisiones que, a su vez, pueden ser utilizadas por otros para validar criptográficamente los resultados obtenidos gracias al protocolo de consenso.

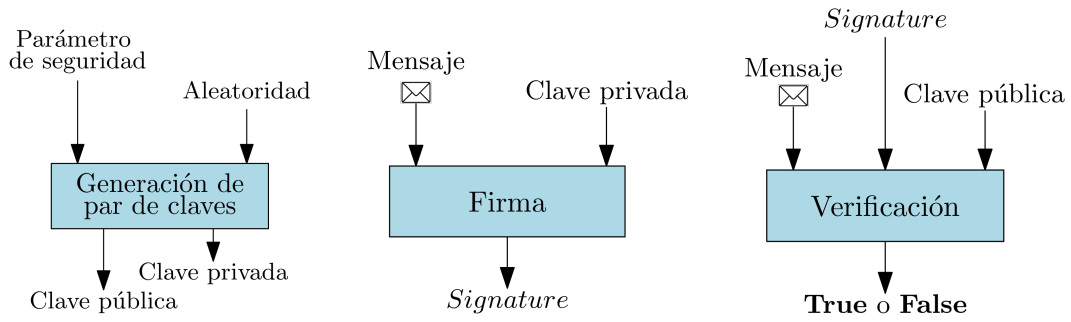
Las firmas digitales son una de las primitivas más poderosas de la criptografía. Son similares a las firmas de puño y letra, en el sentido de que apuntan a cumplir las mismas propiedades. Sin embargo, al ser firmas criptográficas, presentan garantías:

- La firma solo puede ser *utilizada* por su dueño.
- Cualquiera puede *verificar* una firma.

Los *esquemas de firmas digitales* pertenecen al dominio de lo que se conoce como *criptografía asimétrica* o *criptografía de clave pública*. Esta categoría de la criptografía hace uso de distintas claves para distintas funciones (en oposición a una única clave usada en la *criptografía simétrica*), o provee diferentes puntos de vistas a diferentes participantes. En líneas generales esto se puede expresar mediante la existencia de dos claves: una clave pública (conocida por cualquiera) y una clave privada (que debe mantenerse secreta). Estas claves se encuentras conectadas entre sí y tienen roles complementarios.

Un esquema de firmas típicamente consiste en algoritmos diferentes:

- Un algoritmo de generación de pares de claves que un firmante usa para crear una nueva clave privada y una pública.
- Un algoritmo para firmar que toma una clave privada y un mensaje, y produce una firma del mismo.
- Un algoritmo de verificación de firmas que toma una clave pública, un mensaje, y una firma, y determina si la firma fue creada para dicho mensaje con la clave privada asociada a la clave pública.



(a) Generación de claves (b) Firma de mensajes (c) Verificación de firma

Figura 2.2: Interfaz de firma digital.

La clave privada es también conocida como la *clave de firmado* y la clave pública como la *clave de verificación*. En la Figura 2.2 se recapitulan estos tres algoritmos.

Las firmas son utilizadas para autenticar el *origen* de cierta información (que haya sido generada por el dueño de la clave privada asociada a la firma), así como también la *integridad* de la misma (ya que si la pieza de información fuese modificada, la firma sería invalidada).

Existen distintos estándares para los esquemas de firmas digitales, basados en aritmética de módulo sobre grandes números (por ejemplo, RSA-PSS [39]) o en curvas elípticas (como ECDSA [28]). Sin embargo, no se entrará en detalles sobre ellos.

2.5. Los orígenes de la blockchain: Bitcoin

Las blockchains tomaron popularidad con la invención de Bitcoin en 2008, luego de la publicación de un artículo llamado *Bitcoin: A Peer-to-Peer Electronic Cash System*[1], escrito bajo el pseudónimo de Satoshi Nakamoto. Nakamoto combinó diversas invenciones previas como *b-money* [1] y *HashCash* [9] para crear un sistema de dinero electrónico completamente descentralizado que, justamente, no necesita confiar en ninguna autoridad central para la emisión de monedas o la ejecución de transacciones. La innovación clave en Bitcoin fue el uso de un sistema de cómputo distribuido que lleva a cabo una elección global de las nuevas transacciones, permitiendo a la red descentralizada llegar a un consenso sobre el estado de las mismas. Estas transacciones se guardan empaquetadas como bloques en una bitácora distribuida pública conocida como blockchain [8]. Se dice que estas transacciones se encuentran totalmente ordenadas porque es posible determinar para dos transacciones cualesquieras en la blockchain cuál de ellas ocurrió antes.

Si bien la noción de Bitcoin como criptomoneda no es imprescindible para la comprensión de las contribuciones de este trabajo, es el primer caso de uso que da particular importancia a la blockchain como tal y, por lo tanto, a su rendimiento. Por este motivo se considera pertinente presentar un breve resumen sobre su funcionamiento, poniendo foco en el aspecto técnico del mismo (no monetario).

[1] Bitcoin: un sistema de dinero electrónico de par a par.

2.5.1. Cuentas y transacciones

Un usuario de Bitcoin está necesariamente lidiando con primitivas criptográficas. Para comenzar, los usuarios de Bitcoin no tienen un nombre de usuario y una contraseña con la cual inician sesión; en lugar de eso, tienen un par de claves ECDSA (*Eliptic Curve Digital Signature Algorithm*) que generan ellos mismos mediante la interfaz de firmas digitales vista en la sección anterior. El balance de un usuario no es más que una cantidad de BTCs[2] asociada a una clave pública, y por lo tanto, para recibir BTCs, simplemente se debe compartir la clave pública con otros.

Para utilizar los BTCs que tiene disponible, el usuario firma una transacción con su clave privada. De forma simplificada, una transacción se podría pensar tan solo como un mensaje (firmado) del tipo "Envío X BTCs a la clave pública Y".

Una característica fundamental de Bitcoin es que no existe una base de datos real que contenga los balances de cada cuenta. En lugar de eso, los usuarios tienen BTCs disponibles para gastar, llamados *Unspent Transaction Outputs* (UTXOs). Cada usuario puede gastar únicamente sus propios UTXOs (puesto que necesita la clave privada asociada para hacerlo). Cuando una transacción gasta algunos de estos UTXOs, desaparecen y aparecen nuevos, disponibles para gastar para el usuario destino de la transacción.

De este modo, para saber cuántos BTCs tiene una cuenta, se deben contar todos los UTXOs que están asignados a dicha clave pública. En otras palabras, se debe contar todo el dinero que fue enviado a dicha clave pública y aún no fue gastado.

Ahora surgen algunas preguntas. ¿Quién está a cargo de elegir y ordenar las transacciones? ¿Quién mantiene el registro de todas ellas y cómo lo hace?

2.5.2. Prueba de trabajo

Para acordar el ordenamiento de las transacciones, Bitcoin permite a cualquiera proponer una lista de transacciones a incluirse en la siguiente página de la bitácora. Esta propuesta contiene una lista de transacciones llamada *bloque*. Sin embargo, dado que la participación en Bitcoin es abierta a cualquiera, la propuesta de nuevos bloques debe ser restringida de alguna forma, de modo que un único servidor haga una propuesta para el siguiente bloque.

Para lograr esto, el protocolo de Bitcoin establece que para proponer un bloque debe resolverse un problema lo suficientemente difícil. Este mecanismo se conoce como prueba de trabajo (*PoW*, por sus siglas en inglés). En el caso de Bitcoin, el rompecabezas a resolver consiste en que el hash del bloque a proponer tenga una representación binaria que comience con un número dado de ceros.

Las personas que se dedican a resolver el rompecabezas y proponer los nuevos bloques de transacciones son denominadas *mineros*.

2.5.3. La forma de la blockchain

La blockchain es la bitácora pública de Bitcoin, un registro ordenado, y con fecha y hora de las transacciones. Cada nodo completo en la red de Bitcoin guarda independientemente una blockchain que contiene solo bloques validados por ese nodo. Cuando cierta cantidad de nodos tienen los mismos bloques en su propia blockchain se dice que llegaron a un consenso.

[2]BTC es el código asociado a la moneda Bitcoin.

La Figura 2.3 muestra una versión simplificada de una blockchain. Cada bloque tiene una sección de datos en donde se agrupan y alojan una o más transacciones nuevas. Copias de cada transacción se hashean, los hashes luego se emparejan, se hashean, se emparejan de nuevo, y se hashean hasta que queda un único hash: la raíz de un *Merkle tree* [35]. En la Figura 2.4 se muestra un ejemplo de este tipo de árboles. La raíz del Merkle tree se aloja en el encabezado del bloque.

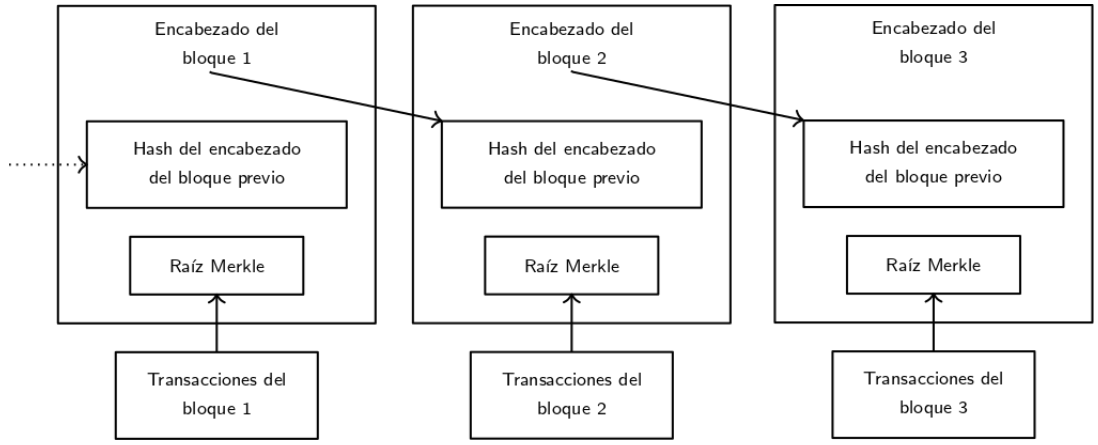


Figura 2.3: Versión simplificada de la blockchain

Cada bloque además contiene el hash del encabezado del bloque anterior, encadenándolos. La blockchain es realmente una sucesión de bloques, donde cada bloque refiere al previo, hasta llegar al primero, conocido como *génesis*. Esto asegura que una transacción no pueda modificarse sin modificar el bloque que la contiene y todos los bloques siguientes [2].

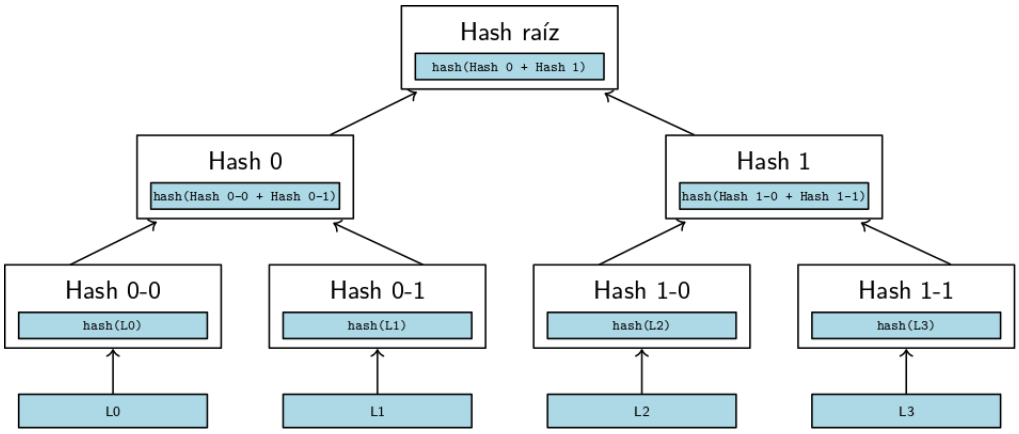


Figura 2.4: Merkle tree

2.6. Tendermint

Tendermint [12] es un motor de replicación de máquinas de estado que tolera fallas bizantinas. Fue uno de los primeros sistemas en adaptar protocolos de consenso clásicos tolerantes a fallas bizantinas al paradigma de blockchain. Es decir, donde el consenso se ejecuta sobre lotes de transacciones vinculados mediante hashes criptográficos (usualmente llamados bloques) en una red pública y abierta.

Tendermint funciona como una capa intermedia de blockchain que soporta la replicación de aplicaciones arbitrarias, escritas en cualquier lenguaje de programación [15]. En la Figura 2.5 se muestra un esquema general de la arquitectura de la replicación de máquinas de estado. Las transacciones se reciben desde el cliente, quien se comunica, mediante una API, con una (o más) de las máquinas en la red. Estas transacciones pasan a través del protocolo de consenso, quien es el encargado de ordernar dichas transacciones en bloques, mediante una serie de pasos que involucran comunicación entre todas las máquinas (señalizada en la figura con líneas punteadas). Una vez ordenadas, las transacciones son ejecutadas en cada una de las máquinas, modificando el estado de cada una de ellas. De esta forma, se replican lotes de transacciones enviadas por los distintos clientes, que son ejecutadas en el mismo orden en cada una de las máquinas en la red, resultando en un mismo estado a lo largo de múltiples máquinas. Las figuras azules representan máquinas. Las líneas punteadas representan la comunicación entre máquinas para llevar a cabo el protocolo de consenso para ordernar transacciones.

Como motor de replicación de máquinas de estado, Tendermint es un sistema distribuido compuesto por un conjunto dinámico de procesos o nodos. Estos nodos varian con el tiempo, a medida que se unen o abandonan la red subyacente del sistema. Un subconjunto de estos nodos son quienes ejecutan el algoritmo de consenso de Tendermint. A estos nodos se los denomina *validadores*. Los nodos que no ejecutan el algoritmo de consenso de Tendermint pueden, sin embargo, participar activamente en otros módulos del sistema.

2.6.1. Arquitectura de capas de Tendermint

Tendermint consiste en dos componentes técnicos principales: un motor de consenso de blockchain, y una interfaz de aplicación genérica. El motor de consenso, llamado *Tendermint Core*, asegura que las mismas transacciones sean registradas en cada máquina en el mismo orden. La interfaz de aplicación, llamada *interfaz de aplicación de blockchain* (ABCI)[3], habilita que las transacciones sean procesadas en cualquier lenguaje de programación.

La Figura 2.6 presenta la estructura básica de Tendermint como un sistema de capas. En este sistema de capas se incluyen los componentes fundamentales mencionados en el párrafo anterior. La capa inferior se encarga de la comunicación entre pares, y provee comunicación para los principales módulos de la blockchain. Éstos son los módulos que se presentan en la capa inmediata superior, el Tendermint Core: mempool, consenso, evidencia, sincronización rápida, y sincronización de estado. Para el presente trabajo, de la capa Tendermint Core, solo serán pertinentes los módulos de mempool y de consenso. En la capa inmediata superior se encuentra la ya mencionada interfaz de aplicación, que es quien se encarga de comunicarse con la aplicación en sí. La aplicación reprsenta la capa superior de este sistema. Como se puede observar en la figura, el cliente se comunica con la capa de Tendermint Core, por medio de los módulos de mempool o consenso.

[3]Proveniente de *Application BlockChain Interface* (ABCI) en inglés.

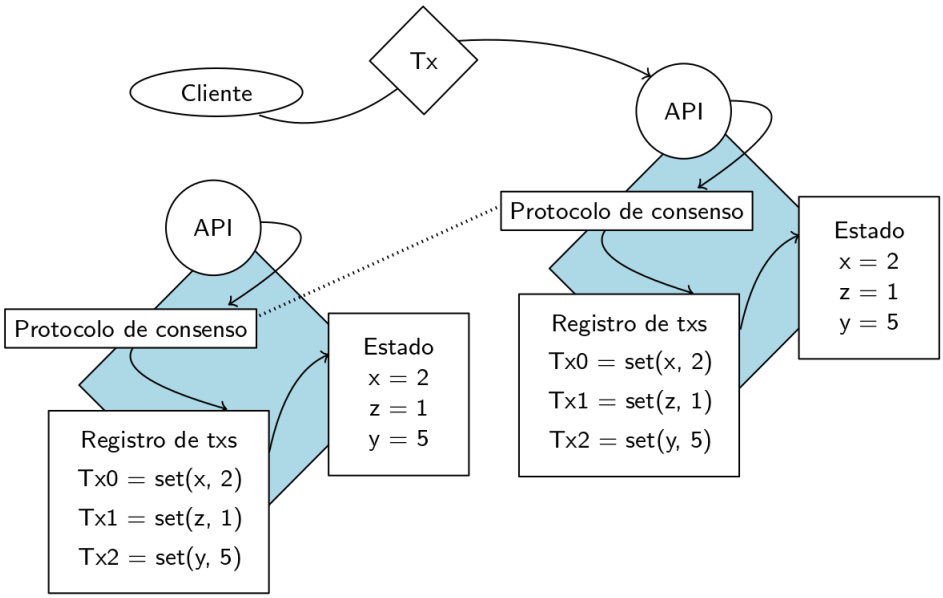


Figura 2.5: Esquema simplificado de la replicación de máquinas de estado.

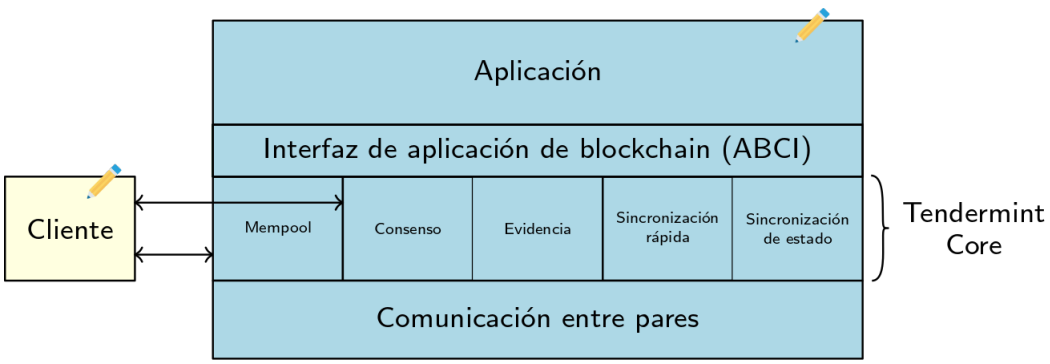


Figura 2.6: Arquitectura de capas de Tendermint.

Los lápices presentes tanto en el cliente como en la capa de aplicación indican que esos son los módulos creados por la programadora. Tendermint funciona como una caja negra que resuelve consenso para la creación de aplicaciones arbitrarias.

Tendermint Core

La capa Tendermint Core tiene diversos módulos, de los cuales se presentarán brevemente dos: mempool y consenso. El módulo de mempool es un punto de entrada de Tendermint. Recibe, valida, almacena y difunde transacciones enviadas por los clientes. Los nodos exponen una interfaz para recibir dichas transacciones de los clientes, mediante llamadas RPC (*Remote Procedure Call*)[4]. Por su parte, el módulo de consenso se encarga de ordenar e intermediar la ejecución de las transacciones, por medio de la interfaz de aplicación. Es responsable de decidir el siguiente bloque de transacciones a añadir a la blockchain.

En la Figura 2.6 se ilustra mediante flechas la comunicación entre el cliente y el Tendermint Core. Simplificadamente, el cliente interactúa con el sistema mediante RPC al enviar transacciones que (en caso de ser válidas) son añadidas al módulo de mempool, y en general recibe respuestas generadas por el módulo de consenso. Las transacciones enviadas por clientes que llegan al módulo de mempool se difunden a todos los nodos, y en algún momento son recibidas por el nodo validador responsable de proponer el siguiente bloque como parte del algoritmo de consenso. El módulo de consenso de este nodo es el responsable de recuperar la lista de transacciones pendientes para construir el bloque que será propuesto como siguiente.

Application BlockChain Interface

La interfaz de aplicación de blockchain es la interfaz entre el Tendermint Core y la aplicación replicada. Un nodo de Tendermint mantiene tres conexiones ABCI con la aplicación replicada.

La *conexión de consenso* se utiliza solo cuando se hace *commit* sobre un nuevo bloque, y comunica toda la información del bloque mediante una serie de peticiones: `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Esto significa que, cuando un bloque es acordado mediante consenso, Tendermint envía una petición `BeginBlock` seguida por una lista de peticiones `DeliverTx` (una por cada transacción en el bloque), que finalizan con las peticiones `EndBlock` y `Commit`, en ese orden.

La *conexión de mempool* es usada por el protocolo de reserva de transacciones [5] para validar las transacciones enviadas por los clientes en relación al estado de la aplicación. Esta conexión solo admite peticiones `CheckTx`. Las transacciones se chequean (mediante `CheckTx`) en el mismo orden en que fueron recibidas por el nodo validador. Si `CheckTx` retorna `OK`, la transacción se mantiene en memoria y se retransmite a otros pares en el mismo orden en que fue recibida. En caso contrario, se descarta. Es parte de la lógica de la aplicación definir cuándo una transacción es válida o no, e incluso la validación es opcional. De esta manera, queda claro que Tendermint es agnóstico de la semántica de las transacciones.

La *conexión de consulta* permite recuperar información de la instancia local de la aplicación, usada por distintos módulos de Tendermint (tales como el módulo de filtrado de pares). Se utiliza para consultar la aplicación sin forzar consenso. Se expone mediante

[4] Las llamadas a procedimiento remoto son una forma de interacción entre clientes y servidores.
[5] *Transaction pool protocol* en inglés.

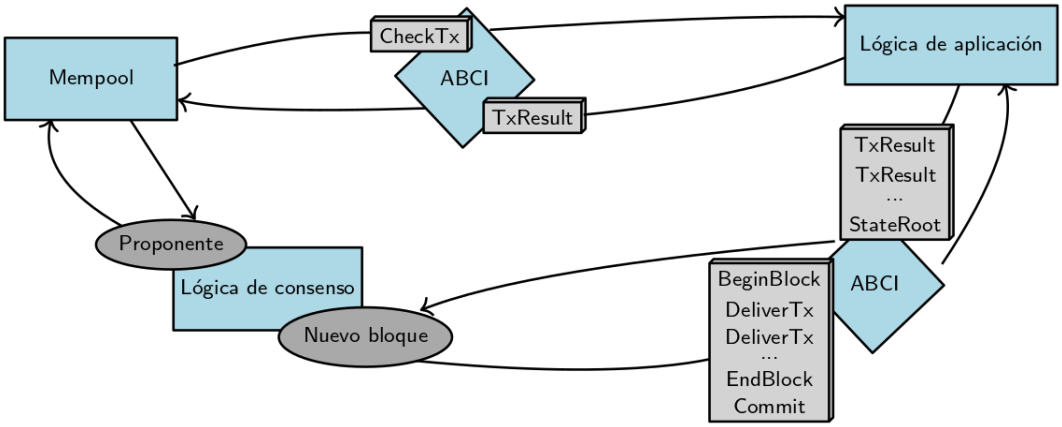


Figura 2.7: Flujo de mensajes mediante la ABCI.

el servidor RPC del Tendermint Core, de modo que los clientes pueden consultar el estado de la aplicación sin exponer un servidor de la aplicación en sí mismo.

La Figura 2.7 muestra el flujo de mensajes mediante las conexiones de consenso y de mempool. Las flechas indican los sentidos de la comunicación. La interacción entre la mempool y la aplicación se da mediante la conexión de mempool de la ABCI. Cuando una transacción llega a la mempool, debe chequearse su validez contra el estado de la aplicación. Esto se realiza mediante la petición `CheckTx` de la ABCI. La respuesta a dicha petición es generada por la lógica de la aplicación, quien se encarga de determinar la validez de una transacción. Por otro lado, la comunicación entre el módulo de consenso y la aplicación se da mediante la conexión de consenso. Cuando a un nodo validador le toca el turno de ser proponente del próximo bloque a añadir en la blockchain, consulta a la mempool para recuperar la lista de transacciones que se utilizarán para la construcción dicho bloque. Una vez que la lógica de consenso resuelve cuál es el nuevo bloque, se comunica con la aplicación para dar a conocerlo. Esto se hace mediante una secuencia de peticiones: `BeginBlock`, [`DeliverTx`, ...], `EndBlock` y `Commit`. La lógica de la aplicación es la encargada de actualizar su estado adecuadamente en función de las transacciones del nuevo bloque y de generar las respuestas a dichas peticiones.

2.7. Modelo de computación

Habiendo presentado los conceptos preliminares básicos para este trabajo, se describe el modelo de computación que utilizaremos de aquí en adelante.

El modelo involucra un sistema distribuido consistente de procesos (clientes y servidores) con una red de comunicación subyacente en donde cada proceso puede comunicarse con cualquiera de los otros. Los clientes representan a aquellos procesos cuyo objetivo es inyectar u obtener elementos en la Setchain, mientras que los servidores representan a los procesos que se encargan de implementar la Setchain como un tipo de datos distribuido con ciertas propiedades.

La comunicación entre procesos se realiza mediante el paso de mensajes. Cada proceso computa independientemente y a su propia velocidad, y los aspectos internos de cada proceso permanecen ocultos para el resto. Los retrasos en la transferencia de mensajes son arbitrarios pero finitos y también permanecen desconocidos para los procesos.

Los procesos pueden fallar arbitrariamente, pero el número de servidores que fallan

(bizantinos) está limitado por f, y el número de servidores total, n, es al menos $3f+1$. Se asumen *canales confiables* entre procesos no bizantinos (correctos), por lo tanto ningún mensaje se pierde, se duplica o se modifica. Cada proceso (cliente o servidor) tiene un par de claves públicas y privadas. Las claves públicas fueron disribuidas confiablemente entre todos los procesos que pueden interactuar con otros. Por lo tanto, se descarta la posibilidad de procesos espurios. Se asume que los mensajes son autenticados, de modo que mensajes corruptos o fabricados por procesos bizantinos son detectados y descartados por los procesos correctos. Como resultado, la comunicación entre procesos corrrectos es confiable pero asíncrona.

Capítulo 3

Setchain

3.1. Objetivo del capítulo

En este capítulo se presenta una descripción detallada de la estructura de datos Setchain, ejemplos básicos de uso y sus propiedades más importantes.

3.2. Presentación

Setchain [14] es una estructura de datos concurrente y distribuida que implementa conjuntos distribuidos que solo crecen, con barreras de sincronización llamadas épocas.

Las barreras imponen un orden entre elementos pertenecientes a diferentes épocas pero no entre elementos de la misma época. Por lo tanto, Setchain relaja el requerimiento de orden total impuesto por las blockchains y, en consecuencia, logra mayor rendimiento y escalabilidad.

La idea básica de funcionamiento de la Setchain involucra tanto nodos clientes como servidores. Los clientes se comunican con algún nodo servidor (quien mantiene la Setchain) para solicitar añadir un nuevo elemento. A su vez, los procesos clientes pueden pedirle a los servidores que les retornen el estado actual de la Setchain.

3.3. La interfaz de Setchain

Sea U el universo de elementos que los procesos clientes pueden inyectar en la Setchain. Se asume que los servidores pueden chequear la validez de los elementos localmente. Llamamos $V \subseteq U$ al subconjunto de elementos válidos. Luego, una Setchain se define como una estructura de datos distribuida en donde cada nodo servidor (correcto) mantiene:

- un conjunto $\texttt{the\_set} \subseteq V$ de elementos agregados;
- un número natural $\texttt{epoch} \in \mathbb{N}$;
- una función $\texttt{history} : [1..\texttt{epoch}] \to \mathcal{P}(V)$[1] que describe los conjuntos de elementos que fueron estampados con un número de época.

Si bien `history` está definido formalmente como una función, también se utilizará notación de conjuntos sobre dicho símbolo. Es decir, utilizaremos $e \in \texttt{history}$ para

[1] $\mathcal{P}(V)$ denota el conjunto potencia de V.

referirnos a que existe algún i en el dominio de `history` para el cual $e \in \texttt{history}(i)$. Inicialmente, tanto `the_set` como `history` son vacíos y `epoch` $= 0$ en todo servidor correcto.

Cada nodo servidor v soporta dos operaciones, `add` y `get`, disponibles para todos los procesos clientes. Se usa notación de punto para invocar estas operaciones, así como también para hacer referencia a las versiones de `the_set`, `epoch` y `history` que dicho servidor mantiene. De este modo:

- v.`add`(e): solicita agregar el elemento e a v.`the_set` (a la versión de `the_set` que v mantiene).
- v.`get`(): retorna los valores de v.`the_set`, v.`history`, y v.`epoch`.

En un servidor v, el conjunto v.`the_set` contiene el conocimiento de v sobre los elementos que fueron *añadidos*, incluyendo aquellos que aún no fueron estampados con un número de época. Con *elementos añadidos* nos referimos a elementos válidos para los cuales algún cliente invocó previamente `add`. Por otro lado, v.`history` contiene solo aquellos elementos (válidos) que ya han sido estampados con un número de época.

En principio, los clientes pueden intentar agregar elementos tanto válidos como inválidos. Sin embargo, los servidores pueden chequear la validez de los elementos y descartar aquellos que sean inválidos. Por lo tanto, en las siguientes secciones se hace referencia exclusivamente a elementos válidos.

3.3.1. Incrementos de época

Cuando se impone una nueva barrera de sincronización, los nodos que mantienen la Setchain colaborativamente deciden cuáles elementos añadidos son estampados con la época actual, y se incrementa el número de época. Es decir, se inicia un proceso en el cual se recolectan aquellos elementos, aún no estampados con un número de época, pertenecientes al conjunto `the_set` mantenido por cada servidor. Luego se decide cuáles de ellos son estampados con la época actual, para dar paso a la siguiente época. Estos eventos se llaman *incrementos de época*.

En la interfaz original de Setchain los nodos servidores también proveen una operación `epoch_inc`(h), donde debe cumplirse $h = \texttt{epoch} + 1$, que puede invocarse por parte de un servidor para forzar un incremento de época. En este trabajo se asume que las barreras de sincronización se lanzan periódicamente y, por lo tanto, que siempre existe un instante futuro en el cual un nuevo incremento de época ocurre. Por este motivo no se hará uso de la operación `epoch_inc` en el presente informe.

3.3.2. Flujo de trabajo

Informalmente, un proceso cliente p invoca v.`get`() sobre un servidor v para obtener la terna (v.`the_set`, v.`history`, v.`epoch`): el punto de vista de v sobre la Setchain, donde v.`history` tiene dominio [1...v.`epoch`]. El proceso p invoca v.`add`(e) para insertar un nuevo elemento e en v.`the_set`, con la intención de que en el futuro sea estampado con un número de época.

Un flujo de trabajo típico desde el punto de vista del cliente es como sigue: un cliente invoca `add`(e) en uno (o más) servidores para insertar un nuevo elemento e en la Setchain. El elemento e será propagado hacia los servidores, y cuando un incremento de época ocurra, los servidores intentarán incluirlo en la nueva época. Después de esperar cierto

tiempo, el cliente invoca `get` en uno (o más) servidores para chequear que el elemento fue efectivamente agregado y estampado con una época.

3.4. Ejemplo

En la Figura 3.1 se muestra un breve ejemplo del funcionamiento de la Setchain. Sea v un servidor correcto que mantiene la Setchain. En la Figura 3.1a se muestra un estado posible para la Setchain desde el punto de vista de v. En dicho estado, la invocación $v.\texttt{get}$ por parte de un cliente resulta en la terna $(v.\texttt{the\_set}, v.\texttt{history}, v.\texttt{epoch})$, donde:

$$v.\texttt{the\_set} = \{x, y, z, w, m, n\},$$

$$v.\texttt{history} = \{1 : \{x, y, z\}, 2 : \{w\}, 3 : \{m, n\}\},$$

$$v.\texttt{epoch} = 3.$$

En la Figura 3.1b y la Figura 3.1c se presentan los resultados de un cliente invocando $v.\texttt{add}(\text{a})$ y $v.\texttt{add}(\text{b})$, respectivamente. Luego de añadir ambos elementos, se tiene:

$$v.\texttt{get} = (\{x, y, z, w, m, n, a, b\}, \{1 : \{x, y, z\}, 2 : \{w\}, 3 : \{m, n\}\}, 3).$$

Es decir, el único cambio se percibe en el valor de $v.\texttt{the\_set}$ (que ahora incluye a los elementos a y b).

Finalmente, en la Figura 3.1d se muestra el estado de la Setchain luego de ejecutarse un nuevo incremento de época. Este evento tiene como consecuencia la creación de una nueva época que incluye a los elementos hasta entonces pendientes de ser estampados, en este caso a, b y c. Si bien el elemento c no era conocido por el servidor v previamente, los nodos servidores deciden colaborativamente los nuevos elementos que son estampados. En este sentido, se puede suponer que un cliente invocó $w.\texttt{add}(c)$ sobre un servidor correcto w y, por lo tanto, c pertenecía a $w.\texttt{the\_set}$. De este modo, se tiene el nuevo punto de vista del servidor v:

$$v.\texttt{get} = (\{x, y, z, w, m, n, a, b, c\}, \{1 : \{x, y, z\}, 2 : \{w\}, 3 : \{m, n\}, 4 : \{a, b, c\}\}, 4).$$

3.5. Propiedades

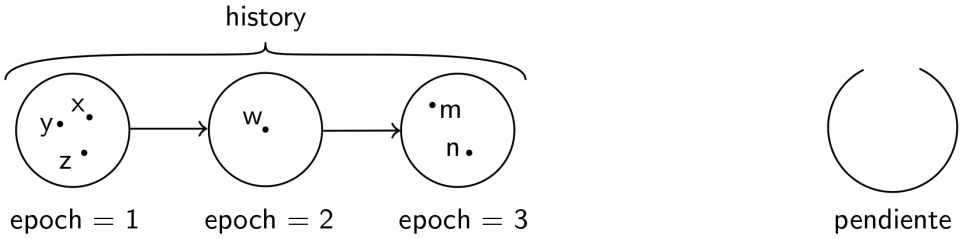
Para asegurar correctitud, las implementaciones de Setchain deben satisfacer ciertas propiedades que proveen garantías eventuales para elementos añadidos y garantía de consistencia entre los servidores correctos. Estas propiedades razonan sobre los servidores correctos, dado que los servidores bizantinos no proveen ninguna garantía.

La primera propiedad establece que las épocas solo contienen elementos que provienen del conjunto de solo crecimiento.

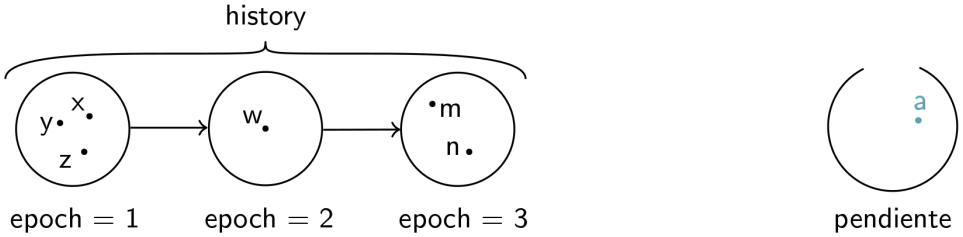
Propiedad 1 (Consistent Sets). Sea $(S, H, h) = v.\texttt{get}()$ el resultado de una invocación a un servidor correcto v. Para cada $i \leq h, H(i) \subseteq S$.

La segunda propiedad declara que todo elemento añadido a un servidor correcto v con el tiempo es retornado en todas las llamadas futuras a $v.\texttt{get}$.

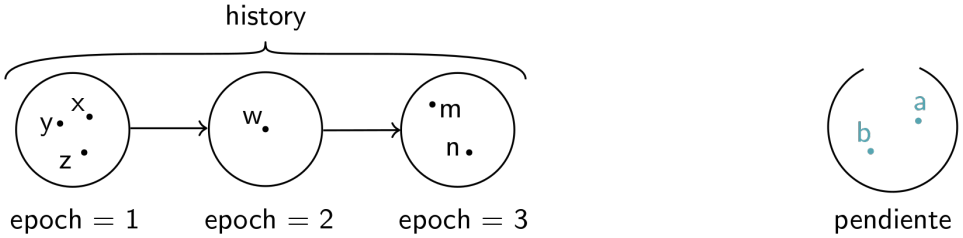
Propiedad 2 (Add-Get-Local). Sea $e \in V$ y $v.\texttt{add}(e)$ una operación invocada en un servidor correcto v. Con el tiempo todas las invocaciones $(S, H, h) = v.\texttt{get}()$ satisfacen $e \in S$.



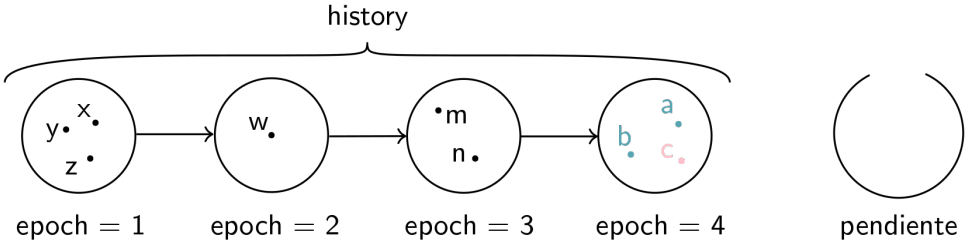
(a) Estado de la Setchain según v



(b) Un cliente invoca v.add(a)



(c) Un cliente invoca v.add(b)



(d) Ocurre un nuevo incremento de época

Figura 3.1: Ejemplo de evolución de la Setchain

La siguiente propiedad establece que los elementos presentes en un servidor correcto son propagados a todos los servidores correctos.

Propiedad 3 (Get-Global). Sean v y w dos servidores correctos, $(S, H, h) = v.\texttt{get}()$ y $e \in U$. Si $e \in S$, tarde o temprano todas las invocaciones $(S', H', h') = w.\texttt{get}()$ satisfacen que $e \in S'$.

Como ya se mencionó anteriormente, a lo largo de este trabajo se asume que en cualquier momento existe un instante futuro en el cual ocurre el próximo incremento de época. Esta suposición es razonable ya que puede ser garantizada usando *timeouts* en cualquier escenario práctico. Luego, la siguiente propiedad establece que todos los elementos añadidos son finalmente estampados con un número de época.

Propiedad 4 (Eventual-Get). Sea v un servidor correcto, $(S, H, h) = v.\texttt{get}()$ y $e \in U$. Si $e \in S$, con el tiempo todas las invocaciones $(S', H', h') = v.\texttt{get}()$ satisfacen que $e \in H'$.

La siguiente propiedad establece que un elemento puede estar en a lo sumo una época.

Propiedad 5 (Unique Epoch). Sea v un servidor correcto, $(S, H, h) = v.\texttt{get}()$, e $i, i' \leq h$ con $i \neq i'$. Se cumple $H(i) \cap H(i') = \emptyset$.

La siguiente propiedad establece que los servidores están de acuerdo en el contenido de las épocas.

Propiedad 6 (Consistent Gets). Sean v y w servidores correctos, $(S, H, h) = v.\texttt{get}()$ y $(S', H', h') = w.\texttt{get}()$, y sea $i \leq \min(h, h')$. Se cumple $H(i) = H'(i)$.

De las dos propiedades anteriores se deriva que ningún elemento puede estar en dos épocas diferentes, incluso si los conjuntos que definen las épocas se obtienen de invocaciones `get` a distintos servidores (ambos correctos).

Finalmente, se requiere que todo elemento en `the_set` provenga del resultado de un cliente añadiendo un elemento.

Propiedad 7 (Add-before-Get). Sea v un servidor correcto, $(S, H, h) = v.\texttt{get}()$ y $e \in S$. Hubo una operación $w.\texttt{add}(e)$ en el pasado en algún servidor w.

Las propiedades 1, 5, 6 y 7 son propiedades de *safety*. Las propiedades 2, 3, y 4 son propiedades de *liveness*.

Capítulo 4

Implementaciones de Setchain

4.1. Objetivo del capítulo

En este capítulo se presenta una familia de implementaciones de Setchain de mundo real construidas sobre Tendermint. En particular, se exponen tres alternativas diferentes, comenzando con una solución simple pero trivialmente correcta y finalizando con un algoritmo complejo que implementa Setchain utilizando funciones hash.

4.2. Consideraciones generales

Para evitar repeticiones, algunas definciones de métodos que permanezcan sin cambios de una versión a la siguiente, no serán re-escritas. Esto será debidamente aclarado al momento de la presentación de los algoritmos.

A su vez, con la intención de mantener consistencia en la nomenclatura, el término *transacción* se utiliza siempre para referirse a las *transacciones de Tendermint*, mientras que *elemento* queda reservado para elementos a agregarse a la Setchain. Dependiendo de la alternativa sobre la que se esté trabajando, una transacción de Tendermint puede contener uno o más elementos a ser agregados.

Las implementaciones correctas de Setchain implementan dos métodos (ver sección 3.2): `add` y `get`, por lo tanto, cada solución provee definiciones para ambas. Por su parte, Tendermint provee dos *endpoints* RPC principales. Nótese que el cliente siempre se comunica con un nodo particular de la red de servidores.

- `Tendermint.Broadcast` se utiliza para enviar transacciones. Cuando una transacción es enviada, se chequea si dicha transacción es válida contra la aplicación (mediante la llamada a `CheckTx`), y en caso afirmativo, se añade a la mempool, se difunde a los otros nodos y en algún momento se incluye en un bloque.
- `Tendermint.Query` se utiliza para consultar el estado de la aplicación.

Por lo tanto, desde el punto de vista del cliente de la Setchain, solo existen dos métodos (`add` y `get`). Sin embargo, hay dos métodos adicionales que se utilizan internamente (es decir, desde el punto de vista de la programadora) para comunicarse con la red de Tendermint subyacente (los ya mencionados `Tendermint.Broadcast` y `Tendermint.Query`).

Finalmente, se asume que hay un predicado definido por el usuario que define cuándo un elemento es válido para ser admitido en el conjunto. En esta sección, dicho predicado se referencia con la función `isValidElement`. De este modo, llamaremos *elementos*

válidos a aquellos elementos e para los cuales `isValidElement`(e) retorne verdadero. En términos de lo presentado en el capítulo 3, podemos pensar que el conjunto de elementos válidos V es $\{e \in U : \texttt{isValidElement}(e)\}$.

A continuación se presentan diferentes definiciones de métodos necesarios para que la aplicación corriendo sobre Tendermint implemente Setchain. Esto involucra, por un lado, los métodos presentados como parte de la API de Setchain y, por el otro, los perteniences a la interfaz de aplicación de blockchain (ver sección 2.6). Si bien el objetivo del capítulo es presentar las soluciones a nivel conceptual, detalles de implementación son dados a lo largo de las secciones cuando se considera oportuno.

La sección de prueba de membresía de elementos (sección 4.3.2) se ubica dentro de la sección de Vanilla únicamente por simplicidad, ya que las ideas allí expuestas aplican de manera similar para todas las soluciones presentadas en este capítulo.

4.3. Primera implementación: Vanilla

Vanilla[1] se presenta como la solución más sencilla a la API de Setchain utilizando Tendermint. La característica principal que le otorga sencillez a esta solución es que un cliente invocando v.`add`(e) sobre un servidor correcto v se traduce en una nueva transacción de Tendermint en la red, que representa a ese y solo a ese elemento a añadir a la Setchain. Que e sea finalmente añadido a la Setchain dependerá de si la transacción asociada al mismo se incluye en un bloque que, a su vez, en algún momento se agrega a la blockchain subyacente de Tendermint.

Otro aspecto fundamental de esta solución es que cada bloque de Tendermint define una única época de Setchain, a la cual pertenecen todos los elementos asociados a las transacciones en dicho bloque.

4.3.1. Flujo de mensajes

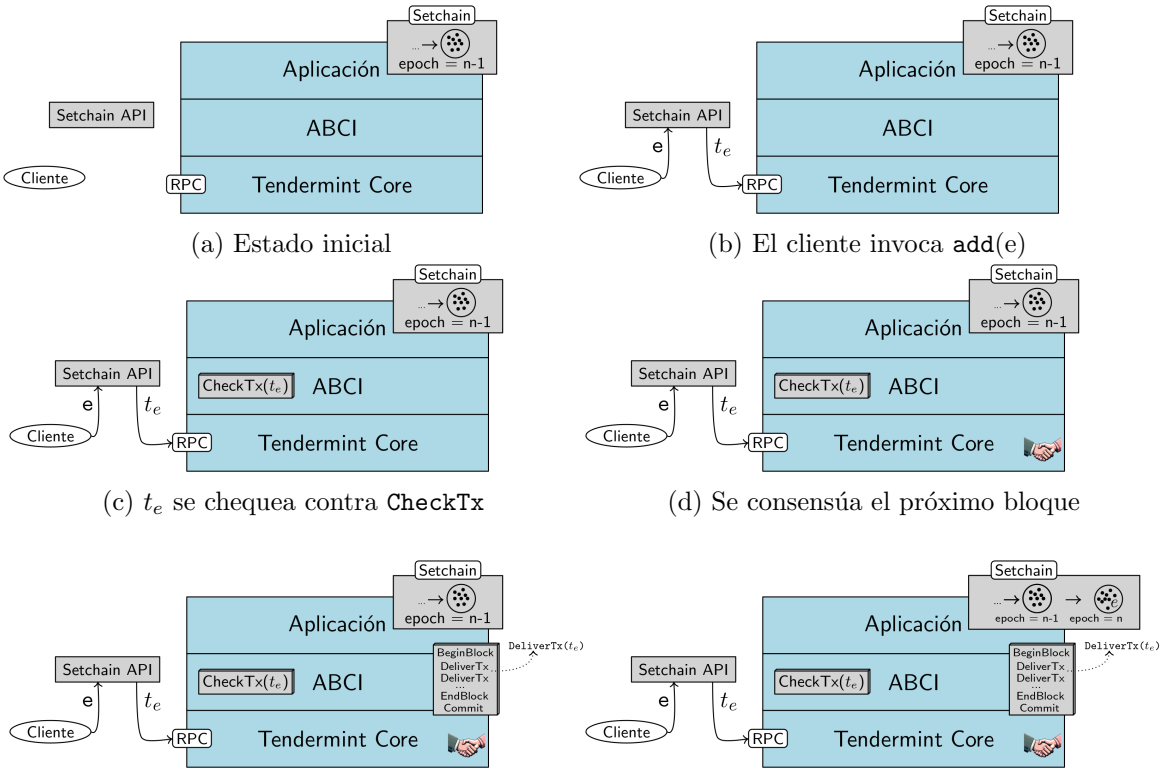
En la Figura 4.1 se muestra el flujo usual de mensajes que se inicia cuando un cliente invoca v.`add`(e) sobre un servidor correcto v.

El sistema comienza en su estado inicial (Figura 4.1a), a la espera de que algún cliente se conecte con ese nodo (parte de la red) para agregar un elemento. El Tendermint Core expone sus puntos de entrada RPC, que serán utilizados por parte de la API de Setchain. Por su parte, la aplicación es quien se encargará de acceder a la base de datos persistente que mantiene la estructura distribuida que conforma la Setchain.

En la Figura 4.1b se observa el resultado de un cliente invocando `add`(e): el cliente, mediante la API de Setchain, envía un elemento e, y la API de Setchain, a su vez, se comunica con el punto de entrada del Tendermint Core para enviar una transacción t_e que representa a e. Si bien el cliente envía *elementos*, en la red de Tendermint lo que circulan son *transacciones*.

Para determinar si la nueva transacción debe ser difundida en la red, Tendermint chequea por medio de `CheckTx` que sea válida y que el elemento asociado a ella no pertenezca ya a la Setchain. En caso afirmativo, inserta la transacción en la mempool. En caso contrario, la descartará y el flujo de dicha transacción terminará. La petición `CheckTx` se muestra en la Figura 4.1c.

[1] En ciencias de la computación el término *vanilla* describe software o hardware que se utiliza en su forma original, es decir, que no se ha modificado con fines específicos. Es por este motivo que a la primera implementación de Setchain le llamamos así, puesto que utiliza Tendermint en su forma natural.



(a) Estado inicial

(b) El cliente invoca `add`(e)

(c) t_e se chequea contra `CheckTx`

(d) Se consensúa el próximo bloque

(e) Se envían los pedidos de la ABCI a la aplicación

(f) Se agrega la época n a la Setchain

Figura 4.1: Flujo usual de mensajes para Vanilla

Si la transacción t_e llega a la mempool, luego de un tiempo, es esperable que sea añadida a un bloque de Tendermint, propuesto como siguiente bloque en la cadena, como parte del algoritmo de consenso. Una vez que dicho bloque obtiene todos los votos necesarios para ser considerado *commited by the network*, comenzará una nueva instancia de consenso. En la Figura 4.1d se representa que el siguiente bloque fue consensuado por parte del Tendermint Core.

El siguiente paso será que el Tendermint Core envíe la secuencia correspondiente de pedidos a la aplicación: `BeginBlock` para indicar el inicio de un nuevo bloque, una lista de llamadas a `DeliverTx` por cada transacción agregada[2] (en el orden en que fueron consensuadas dentro del bloque), `EndBlock` para indicar la finalización del bloque y `Commit` para señalizar que el nuevo estado puede ser persistido. Si la transacción t_e de Tendermint asociada a e es parte de dicho bloque, entonces con seguridad, una de las llamadas a `DeliverTx` tendrá como argumento a t_e. Esto se visualiza en la Figura 4.1e.

El último paso en este flujo (Figura 4.1f) muestra que una nueva época se agregó a la Setchain, en donde uno de sus elementos es e.

Detalles de implementación

La implementación de la API de Setchain establece comunicación con un nodo mediante el protocolo HTTP. Se utiliza la implementación nativa de Tendermint del cliente HTTP, quien se comunica con el nodo de Tendermint usando JSON RPC. De los distintos clientes nativos que provee Tendermint, el cliente HTTP es la implementación más adecuada de utilizar en un entorno real, y es por eso que se elige para este trabajo.

La aplicación necesita acceso a una base de datos persistente, de modo de ir añadiendo los elementos correspondientes a cada época de la Setchain. Esta base de datos persistente se implementa utilizando el paquete *BadgerDB*[4], que provee una base de datos simple y rápida de tipo clave-valor, escrita completamente en Go. Dependiendo del caso de uso de Setchain se pueden pensar distintas formas de almacenar los elementos y sus respectivas épocas, de modo de hacer más eficiente las búsquedas necesarias. Por ejemplo, si fuera importante determinar rápidamente si un elemento ya es parte de la Setchain y con cuál número de época fue estampado, entonces, al agregar un nueva entrada a la base de datos, las claves podrían ser elementos, y los valores, el número de época al que dicho elemento pertenece. Por otro lado, si se quisiera dar mayor eficiencia a determinar todos los elementos pertenecientes a una época dada, las claves podrían ser números de épocas, y los valores los elementos en ella. Incluso podría utilizarse un sistema híbrido, de modo de hacer eficiente diversas búsquedas.

4.3.2. Prueba de membresía de elementos

Como se mencionó antes, cuando un proceso cliente utiliza la API de Setchain para añadir un elemento o para recuperar el estado actual de la Setchain, se comunica con un nodo particular de la red. Es importante notar que, en este sentido, un proceso cliente no puede saber si se está contactando con un nodo correcto o con un nodo bizantino. Hay en principio dos métodos para asegurar que un elemento será añadido a la Setchain.

La estrategia más sencilla es garantizando que el cliente interactúe con suficientes servidores, de modo de asegurarse que al menos uno de ellos sea correcto. Como ya se mencionó en el modelo de computación (ver Sección 2.7), se asume que una cota superior f de servidores bizantinos es conocida y, por lo tanto, se requiere invocar `add` en $f + 1$

[2]En esta implementación una *transacción* agregada coincide con un *elemento* agregado.

servidores distintos para asegurar que al menos uno es correcto. Análogamente, para conocer el estado actual de la Setchain, no alcanza con invocar `get` en un único servidor, puesto que podría ser bizantino y, por tanto, dar información incorrecta. El cliente debería contactar $2f + 1$ servidores (suponiendo que existen f servidores bizantinos que pueden negarse a responder o hacerlo incorrectamente) y esperar a que $f + 1$ de ellos coincidan en la respuesta (si $f + 1$ servidores están de acuerdo en que un elemento pertenece a la Setchain, efectivamente debe pertenecer).

Por otro lado, una estrategia alternativa es la presentada como *cliente optimista* [14], que usa un sistema de firmas criptográficas. Este es el método que utilizaremos en este trabajo. El mecanismo se introduce a continuación.

Los nodos correctos firman criptográficamente un hash del conjunto de elementos pertenecientes a una época junto con el número de dicha época, e insertan este hash como un elemento en la Setchain. A esos hashes los llamamos *elementos de prueba de época*. Las pruebas de época en principio contienen: la clave pública del validador que firma[3] y la firma de los elementos de la época dada (ordenados de una forma específica y conocida) junto con el número de época al que pertenecen. De esta manera, los clientes solo ejecutan una única petición $v.\texttt{add}(e)$ a un determinado servidor v, esperando que sea correcto. Luego de esperar cierto tiempo, los clientes pueden invocar `get` (también a un único nodo) y chequear si el elemento e pertenece a alguna época y existen suficientes pruebas para dicha época. De este modo, se garantiza que al menos un servidor correcto la firmó.

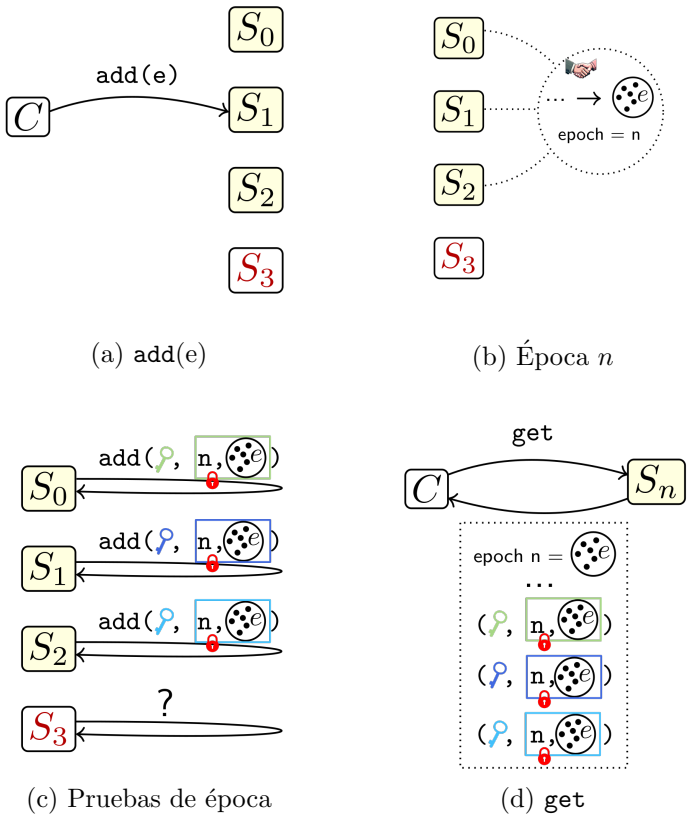
Los clientes pueden verificar si las pruebas de época son válidas generando el hash con el número de época y los elementos que pertenecen a ella, y verificando, a través de la clave pública, si la firma de la prueba es válida para el hash obtenido. Este proceso se corresponde con la verificación de firmas vista en la sección 2.4.

Las claves públicas de los validadores necesitan ser públicamente conocidas para los usuarios. Si un elemento e pertenece a una época que tiene suficientes pruebas, los clientes pueden concluir que la época es correcta y que e fue éxitosamente insertado en la Setchain. En caso de comunicarse con nodos correctos, solo se requieren una llamada a `add` y una llamada a `get` para estar seguros de que el elemento fue añadido a la Setchain.

En la Figura 4.2 se muestra un ejemplo de este mecanismo, en un sistema que cuenta con un cliente y 4 servidores, de los cuales uno es bizantino (señado en rojo). El cliente invoca $\texttt{add}.(e)$ sobre un servidor cualquiera de la red, sin saber si es correcto o bizantino, con la intención de agregar un nuevo elemento a la Setchain. Esto se indica en la Figura 4.2a. Los servidores deciden colaborativamente los elementos pertenecientes a la época n. En la Figura 4.2b se puede ver que e pertenece a dicha época. En la Figura 4.2c se representa que todos los servidores correctos inyectan un elemento de prueba de época que contiene: la clave pública perteneciente a cada nodo (señalizada con una llave), y la firma (señalizada con un candado) del hash de una tupla formada por el número de época (n) y los elementos que pertenecen a ella. Luego de esperar cierto tiempo, el cliente invoca `get` en un servidor cualquiera y, si obtiene suficientes pruebas de época (en este caso, 3) para la época n, entonces puede estar seguro que el elemento e fue efectivamente agregado a la Setchain y que pertenece a la época n.

Las épocas ahora contienen dos tipos de elementos: elementos regulares enviados por un cliente y elementos de prueba de época; significando que cada elemento de prueba de época pertenece a su vez a una época. Sin embargo, los elementos de prueba de

[3]Podría optarse por utilizar simplemente un identificador del validador, dado que las claves públicas son conocidas y podrían inferirse a partir del identificador.



(a) `add`(e) (b) Época n

(c) Pruebas de época (d) `get`

Figura 4.2: Mecanismo de pruebas de época.

época no necesitan ser incluidos como elementos en el hash firmado que es parte de la prueba de época para una época dada. Consecuentemente, es necesario diferenciar entre elementos regulares y elementos de prueba de época. La validez de los elementos regulares se chequea a través de la ya mencionada función `isValidElement`. Sin embargo, un elemento de prueba de época necesita ser validado verificando la firma. Para mantener la simplicidad, consideraremos a partir de ahora una versión extendida de la función `isValidElement` que es capaz de diferenciar entre elementos regulares y elementos de prueba de época, y aplicar la validación correspondiente para cada uno.

Detalles de implementación

El aspecto criptográfico de la implementación de los elementos de prueba de época se hace a través del paquete *crypto* de *go-ethereum*, la implementación del protocolo Ethereum en Golang [4]. Este paquete trabaja con el algoritmo de firma digital ECDSA, uno de los esquemas de firmas más utilizado [45]. Provee una función `Sign` que cacula una firma ECDSA. La firma producida se encuentra en el formato $[R||S||V]$, donde (R, S) son los componentes usuales de una firma ECDSA, y V es 0 o 1. El valor de V es usualmente llamado *bit de recuperación* y permite recuperar inequívocamente, a partir de la firma y de un hash del mensaje firmado, la clave pública de quien realizó la firma. Sin ese bit de recuperación, el proceso de recuperación de clave pública puede retornar potencialmente más de un candidato, motivo por el cual se añade dicho bit.

[4] Disponible en `https://pkg.go.dev/github.com/ethereum/go-ethereum/crypto`.

Algoritmo API Vanilla

```
1: function ADD(transaction)
2:     return Tendermint.Broadcast(transaction)
3: function GET()
4:     return Tendermint.Query()
```

Esta propiedad de recuperación de la clave pública que provee ECDSA es muy útil en contextos con limitaciones de ancho de banda, en donde la transmisión de la clave pública sería muy costosa.

La función `Ecrecover` es la función de recuperación de clave pública que facilita el paquete *go-ethereum*. Los elementos de prueba de época contienen el número de época y la firma en el formato mencionado anteriormente, y los clientes pueden, mediante la propiedad de recuperación, usar la función `Ecrecover` para verificar que la firma dada fue, de hecho, generada por un validador conocido.

4.3.3. Algoritmos

En esta sección se presentan los algoritmos (en pseudocódigo) necesarios para concluir la implementación de la versión Vanilla.

En el Algoritmo API Vanilla se muestra la solución más sencilla a la API de Setchain. Los clientes agregan elementos invocando `add`, cuya implementación llama a `Tendermint.Broadcast`, mientras que pueden consultar el estado actual de la Setchain mediante `get`, quien invoca a `Tendermint.Query`.

Para dar una implementación completa de Setchain utilizando Tendermint, además se deben dar las definiciones de los métodos pertenecientes a la interfaz ABCI (ver sección 2.6.1). El Algoritmo ABCI Vanilla muestra la definición de la ABCI para la versión Vanilla. Solo se muestran las definiciones de `CheckTx`, `DeliverTx`, `EndBlock` y `Query`; el resto tiene implementaciones triviales.

Chequear si una transacción es válida consiste simplemente en chequear que el elemento asociado a ella sea válido y nuevo, dado que las transacciones contienen un único elemento. Puede parecer absurdo definir la función `getElementFromTransaction()` en este caso, debido a que es la función identidad. Sin embargo, la decisión de dar la definición explícita se basa en enfatizar la diferencia conceptual entre una transacción de Tendermint y los elementos a añadir en la Setchain.

El Tendermint Core envía peticiones `DeliverTx` asíncronamente pero en orden una vez por cada transacción en el bloque. Cuando `DeliverTx` se ejecuta, las transacciones ya fueron ordenadas en el consenso global por el protocolo de Tendermint. Para este algoritmo, la única acción a realizar por parte de la ABCI cuando se recibe una transacción es añadir el elemento subyacente a la Setchain siempre y cuando el mismo sea válido y nuevo. Siempre es necesario chequear del lado de la ABCI si una transacción es válida, ya que un nodo bizantino podría añadir transacciones sin chequear su validez previamente.

Tendermint envía peticiones `EndBlock` una única vez por bloque, luego de haber enviado todas las transacciones dentro del mismo. En este algoritmo, la finalización de un bloque desencadena un incremento de época y, por lo tanto, cada bloque de Tendermint define una época distinta en la Setchain, la cual contiene como elementos a las transacciones de dicho bloque. La finalización de un bloque también origina la

Algoritmo ABCI Vanilla

```
1: Init: epoch ← 0, next_epoch ← 1, history ← {}, the_set ← {}
2: function CheckTx(transaction)
3:     return isValidTransaction(transaction)
4: function DeliverTx(transaction)
5:     element ← getElementFromTransaction(transaction)
6:     if isValidTransaction(transaction) then
7:         the_set ← the_set ∪ {element}
8:         history[next_epoch] ← history[next_epoch] ∪ {element}
9:     return
10: function EndBlock()
11:     hash ← Hash(history[next_epoch], next_epoch)      ▷ Calcular el hash de la época.
12:     epochProof ← Sign(hash, PRIVATE_KEY)
13:     add(epochProof)
14:     epoch ← next_epoch
15:     next_epoch ← next_epoch + 1          ▷ Cada bloque de Tendermint define una época.
16:     return
17: function Query()
18:     return (the_set, history up to epoch, epoch)
19: function isValidTransaction(transaction)
20:     element ← getElementFromTransaction(transaction)
21:     return isValidElement(element) and not element in history
22: function getElementFromTransaction(transaction)
23:     return transaction
```

creación de un elemento de prueba de época y la posterior invocación a `add` para añadirlo a la Setchain.

Finalmente, cuando los clientes invocan `get`, indirectamente inician una llamada a `Tendermint.Query`, que retorna los valores de `the_set`, `history` y `epoch`, de acuerdo a la definición de Setchain. En este caso, `the_set` contendrá todos los elementos para los cuales se hizo `DeliverTx`. Esto incluye elementos desde la época 0 hasta la época que se está construyendo actualmente (indicada por la variable `next_epoch`). Nótese que los elementos pertenecientes a dicha época pueden estar siendo agregados a la variable `history` en el momento en que `Query` se invoca, por lo que el valor de `history[next_epoch]` podría ser una visión parcial. Es por esto que se retorna `history` up to `epoch` y no directamente el valor de la variable `history`. De otro modo, podría no cumplirse la propiedad *Consistent Gets* presentada en la sección 3.5. La variable `epoch` retornada indica el máximo número de época que fue agregado de manera completa en `history`.

4.3.4. Conclusión

Si bien los algoritmos descriptos en esta sección implementan Setchain, no se está explotando su idea principal: relajar el orden total entre los elementos. Aunque diversos elementos pueden pertenecer a una misma época, significando que no hay orden total entre ellos, por detrás, el protocolo de consenso de Tendermint sí está decidiendo un orden total entre elementos.

4.4. Segunda implementación: Compresschain

Para acercanos al objetivo de Setchain, en esta sección proponemos una implementación nueva, llamada Compresschain, que explora la relajación de orden propuesta por Setchain, aún ejecutando el algoritmo de consenso de Tendermint por detrás.

En lugar de difundir inmediatamente cada elemento añadido por un cliente como una transacción de Tendermint, una nueva pieza intermedia de software llamada *collector* es responsable de recolectar elementos hasta llegar a un lote lo suficientemente grande, que es difundido como una única transacción. Esta transacción está formada por los elementos enviados (potencialmente) por diferentes clientes, convenientemente codificados. El lote de elementos se comprime antes de inyectarse en la red de Tendermint, de ahí el nombre Compresschain. Una transacción es, entonces, un lote de elementos comprimido.

En este caso, cada una de las transacciones en un bloque de Tendermint define una nueva época de la Setchain, la cual contiene todos los elementos pertenecientes al lote comprimido. Por lo tanto, un mismo bloque de Tendermint involucrará diferentes épocas en la Setchain (una por cada transacción dentro del bloque).

4.4.1. Flujo de mensajes

Al igual que como se hizo para la versión Vanilla, se presentará el flujo de mensajes usual que se inicia cuando un cliente invoca `add`(e). En la Figura 4.3 se muestran los distintos pasos de este flujo en el contexto de Compresschain.

El sistema comienza en su estado inicial. Como se muestra en la Figura 4.3a, detrás de la API de Setchain se encuentra la nueva pieza intermedia *collector*.

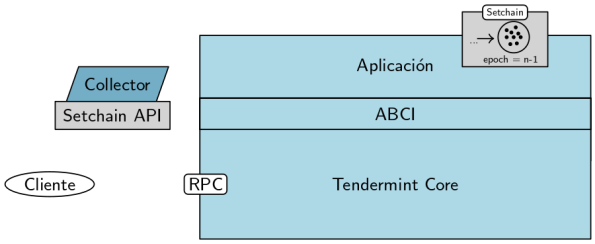
El primer paso se da cuando el cliente envía un elemento e, mediante la invocación a `add`(e), como se ilustra en la Figura 4.3b. Desde el punto de vista del cliente, no hay diferencias en relación a la versión Vanilla, ya que la existencia del *collector* es transparente para él.

En la Figura 4.3c se muestra que el *collector* recolecta elementos que envían los distintos clientes y los agrupa en un lote. Un lote contiene potencialmente varios elementos enviados por uno o más clientes a un mismo nodo servidor que corre el Tendermint Core, la aplicación y el *collector*. Cuando el lote se encuentra listo, el *collector* tiene una nueva transacción para inyectar en la red de Tendermint. Esta transacción se presenta en las figuras con un *símbolo de carpeta* gris.

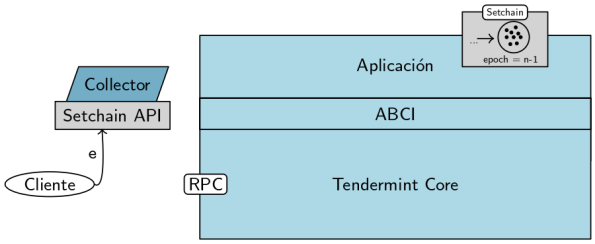
Una vez que el lote fue comprimido, el flujo es el mismo que el explicado en la sección anterior para Vanilla: la transacción que representa al elemento e (y a otros) se chequea contra `CheckTx` para determinar si debe ser insertada en la mempool o descartada. Si llega a la mempool, se espera que luego de un tiempo sea añadida a un bloque de Tendermint y que, tarde o temprano, mediante el algoritmo de consenso, este bloque se considere *commited by the network*. Una vez que el bloque fue decidido en el consenso, entonces el Tendermint Core enviará la secuencia correspondiente de pedidos a la aplicación. Una de las peticiones será `DeliverTx`, entregando la transacción correspondiente al lote que contiene a e. Esto se observa en las Figuras 4.3d. 4.3e y 4.3f.

En el último paso de este flujo se agregan las nuevas épocas a la Setchain. En contraposición con lo visto en Vanilla, acá se agrega una nueva época por cada transacción (cada petición `DeliverTx`). Las épocas contienen potencialmente más de un elemento porque, de hecho, las transacciones en este contexto representan a más de un elemento envíado por los clientes. Estas transacciones sí tienen orden entre ellas, por lo que el

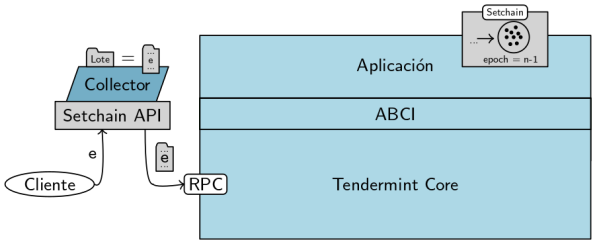
número de época asociada a cada una está determinado por el mismo. En la Figura 4.3g se muestra que la época n se agregó a la Setchain y uno de sus elementos es e. Esta es, justamente, la época asociada a la transacción que contiene al elemento e.



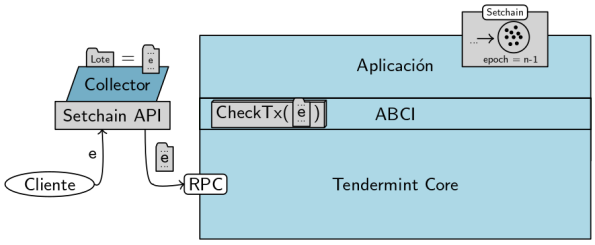
(a) Estado inicial



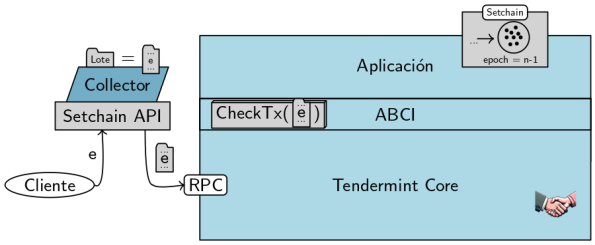
(b) El cliente invoca `add`(e)



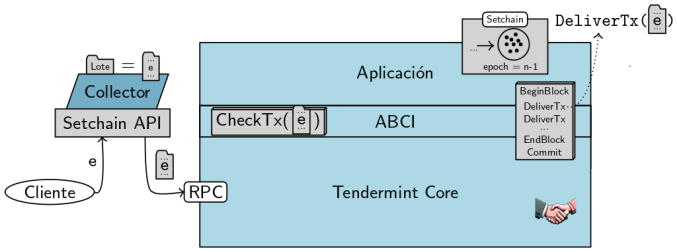
(c) El lote está listo y es difundido



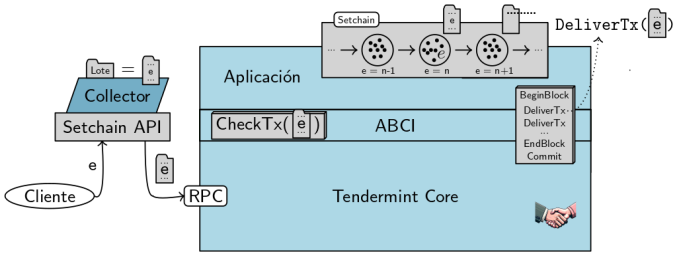
(d) El lote se chequea contra `CheckTx`



(e) Se consensúa el próximo bloque



(f) Se envían los pedidos de la ABCI a la aplicación



(g) Se agregan las nuevas épocas a la Setchain

Figura 4.3: Flujo usual de mensajes para Compresschain

Detalles de implementación

La implementación del *collector* se hace a través de un servidor HTTP RPC que es accesible para los clientes mediante la API de Setchain. Para esto se utilizan los paquetes *http* y *rpc*, ambos de la librería estándar de Golang. El paquete *http* proporciona implementaciones para el cliente y el servidor HTTP. El paquete *rpc* provee acceso a los métodos exportados de un objeto (en este caso, el *collector*) a través de una red u otra conexión de entrada/salida. Un servidor registra un objeto, haciéndolo visible como un servicio con el nombre del tipo del objeto. Después del registro, los métodos exportados

Algoritmo Compress Collector

```
1: Init: batch ← {}
2: function AddElement(element)
3:     if isValidElement(element) then
4:         encoded_element ← RLP.Encode(element)
5:         batch ← batch ∪ {encoded_element}
6:     return
7: when (isReady(batch)) do
8:     compressed_batch ← Brotli.Compress(batch)
9:     Tendermint.Broadcast(compressed_batch)
10:    batch ← {}
```

Algoritmo API Compresschain

```
1: function ADD(element)
2:     return CompressCollector.AddElement(element)   ▷ Usar el componente intermedio.
3: function GET()
4:     return Tendermint.Query()
```

del objeto (en este caso, únicamente `AddElement`) son accesibles de forma remota.

4.4.2. Algoritmos

En Compresschain, en lugar de agregar directamente elementos enviados por los usuarios a la red de Tendermint, primero se recolectan y se comprimen para ser enviados como una única transacción. Siguiendo la práctica actual de Ethereum, se utiliza el algoritmo *Recursive Length Prefix* (RLP) [46] para codificar los elementos individualmente y el algoritmo de compresión Brotli [7] para comprimir el lote[5].

Un lote se considera *listo* para ser enviado una vez que, o bien alcanza un tamaño máximo, o bien una cantidad razonable de tiempo transacurrió desde que el primer elemento llegó. El Algoritmo API Compresschain presenta una nueva definición para la API de Setchain. En el Algoritmo Compress Collector se muestra una implementación posible para un *collector*.

En Compresschain, un cliente que ejecuta `add` invoca indirectamente el método `AddElement` del *collector*. Se realizaron algunas simplificaciones en los algoritmos presentados. Por ejemplo, una condición de carrera podría ocurrir si varias rutinas agregan elementos a la misma instancia del *collector* concurrentemente. Sin embargo, esto puede resolverse sencillamente gracias al uso de candados.

Para completar nuestra definición, se provee el pseudocódigo para la ABCI de Compresschain en el Algoritmo ABCI Compresschain. Existen varias diferencias con respecto a la implementación previa de Setchain. Se elaboran a continuación.

La diferencia principal con la implementación Vanilla radica en que las transacciones contienen potencialmente más de un elemento de usuario. Esto se da porque las transacciones en este contexto son lotes comprimidos de elementos. Más aún, para recuperar elementos dentro de las transacciones se necesita primero descomprimir el lote y, una vez que se tiene el lote original, debe RLP-decodificarse para así obtener los elementos

[5]Otros algoritmos de codificación y de compresión podrían utilizarse de ser necesario.

Algoritmo ABCI Compresschain

```
1: Init: epoch ← 0, next_epoch ← 1, history ← {}, the_set ← {}
2: function CheckTx(batch)
3:     return isValidBatch(batch)
4: function DeliverTx(batch)
5:     elements ← getElementsFromBatch(batch)
6:     newEpoch(elements)
7:     return
8: function Query()
9:     return (the_set, history up to epoch, epoch)
10: function isValidBatch(batch)
11:     elements ← getElementsFromBatch(batch)
                    ▷ Si al menos un elemento en el lote es válido y nuevo, el lote es válido.
12:     for e in elements do
13:         if isValidElement(e) and not e in history then
14:             return True
15:     return False
16: function getElementsFromBatch(batch)
17:     decompressed_batch ← Brotli.Decompress(batch)
18:     elements ← RLP.Decode(decompressed_batch)
19:     return elements
20: function newEpoch(elements)
21:     for e in elements do                ▷ Agregar solo elementos nuevos y válidos.
22:         if isValidElement(e) and not e in history then
23:             the_set ← the_set ∪ {e}
24:             history[next_epoch] ← history[next_epoch] ∪ {e}
25:     hash ← Hash(history[next_epoch], next_epoch)      ▷ Calcular el hash de la época.
26:     epochProof ← Sign(hash, PRIVATE_KEY)
27:     add(epochProof)
28:     epoch ← next_epoch
29:     next_epoch ← next_epoch + 1
30:     return
```

originales enviados por los clientes. Aquellos elementos son los que deben ser agregados a la Setchain. La función `getElementsFromBatch` muestra este comportamiento.

En la definición de Compresschain, las transacciones de Tendermint contienen, en principio, más de un elemento. Se necesita definir un nuevo criterio para determinar cuándo una transacción (es decir, un lote de elementos comprimido) se considera válida. La función `isValidBatch` (en el Algoritmo ABCI Compresschain) implementa este nuevo criterio, permitiendo a las transacciones ser consideradas válidas si al menos uno de sus elementos es válido y nuevo (es decir, no fue agregado a la Setchain previamente). La elección de este criterio se basa en que, si bien se espera que un *collector* correcto genere lotes de elementos válidos (puesto que chequea la validez de los elementos antes de agregarlos), no puede asegurarse que sean nuevos. A su vez, un *collector* bizantino también podría incoporar elementos inválidos en un lote. Por lo tanto, las transacciones se consideran válidas si al menos uno de sus elementos es válido y nuevo. A pesar de que una transacción válida puede contener algún elemento inválido o repetido, como se ilustra en el método `DeliverTx`, solo elementos válidos y nuevos se agregan a la Setchain, mientras que el resto simplemente se descarta.

La última diferencia a mencionar con respecto a Vanilla es que en Compresschain

los bloques no delimitan épocas. Por esta razón no se peresenta una definición para el método `EndBlock`. En este caso, las épocas se definen como transacciones: cada lote de elementos define una época.

El método `Query` se comporta de la misma forma que en Vanilla.

4.4.3. Conclusión

Compresschain es una alternativa construida enteramente sobre Tendermint, pero que agrega una capa intermedia entre el cliente y el Tendermint Core. De forma completamente transparente tanto para el cliente como para la pila de Tendermint, muchos elementos enviados potencialmente por diferentes partes (pero a la misma máquina servidor) son difundidos dentro de la red de Tendermint como una única transacción. El algoritmo de consenso decide el orden de estas transacciones, ignorando por completo cuáles y cuántos elementos hay dentro de ellas. Si bien los elementos del cliente agrupados dentro de un lote naturalmente tienen un orden específico, este orden no se decide como parte del consenso y por eso se ignora cuando los elementos son agregados a una nueva época de la Setchain.

El uso de un algoritmo de compresión previo a la inyección de una nueva transacción es una apuesta para mejorar el rendimiento, medido en cantidad de elementos por segundo, que pueden ser añadidos a la Setchain. Este enfoque trabaja bajo la hipótesis de que hacer consenso sobre elementos agrupados de forma eficiente interpretados como unidad representa una mejora en comparación con el consenso usual, hecho sobre elementos individuales.

4.5. Tercera implementación: Hashchain

Hashchain se alinea con el concepto introducido en Compresschain; sin embargo, usa funciones hash en lugar de usar un algoritmo de compresión. Mientras que el poder de compresión de las funciones hash puede ser enorme, dado que estas funciones mapean datos de longitud arbitraria a valores de tamaño fijo, los hashes son irreversibles. Esto significa que debe proveerse un método no trivial para recuperar el lote original de elementos.

La implementación de Hashcain involucra dos aspectos: una blockchain de Tendermint compuesta por hashes y una estrategia distribuida de inversión de hashes para obtener los lotes originales de elementos.

A menudo en esta sección nos tomaremos la libertad de utilizar la expresión *elementos en un hash H* para referirnos a aquellos elementos pertenecientes al lote B, tal que $Hash(B) = H$.

En lugar de usar el *collector* de Compresschain (ver Algoritmo Compress Collector), ahora utilizamos un *hash collector*, que construye un lote de elementos para luego hashearlo antes de introducirlo a la red de Tendermint. Los lotes hasheados son difundidos como transacciones y compartidos a través de toda la red, definiendo cada uno de ellos una nueva época en la Setchain, de forma análoga al enfoque empleado para los lotes comprimidos en Compresschain.

4.5.1. Flujo de mensajes

Al igual que se hizo para las versiones Vanilla y Compresschain, el objetivo es presentar el flujo usual de la vida de un elemento en el contexto de Hashchain. Retomando

la Figura 4.3 de la sección anterior, podemos pensar que el flujo de mensajes debería ser similar. El cliente se comunica con la API de Setchain, mientras que el *hash collector* colecciona elementos hasta que el próximo lote esté listo para ser hasheado y enviado al Tendermint Core. Una vez allí es sometido a `CheckTx` para determinar si debe ser añadido a la mempool o descartado.

Analicemos con mayor profundidad esta situación. En Compresschain, para decidir si una transacción es válida, debe revertirse el proceso hecho a la misma. Es decir, descomprimir la transacción para así obtener los elementos de cliente que la componen y analizarlos de forma individual. Cuando se piensa en la contraparte para Hashchain, una de las primeras preguntas que surge es cómo se determina si una transacción (un hash) es válida para ser agregada a la mempool, siendo que no es trivial recuperar los elementos originales teniendo un hash.

La misma pregunta surge luego, cuando después de consensuado el siguiente bloque llegan las peticiones de la ABCI. El evento `DeliverTx` dará la indicación de agregar una nueva época a la Setchain, para lo cual también será necesario conocer los elementos originales con los cuales se formó el hash que está siendo entregado.

Por lo tanto, dado que en el escenario propuesto por Hashchain, tanto `CheckTx` como `DeliverTx` reciben hashes como transacciones, el flujo de mensajes implicará resolver el problema de la obtención de elementos originales provenientes del hash. A este proceso lo llamaremos *inversión de hashes*.

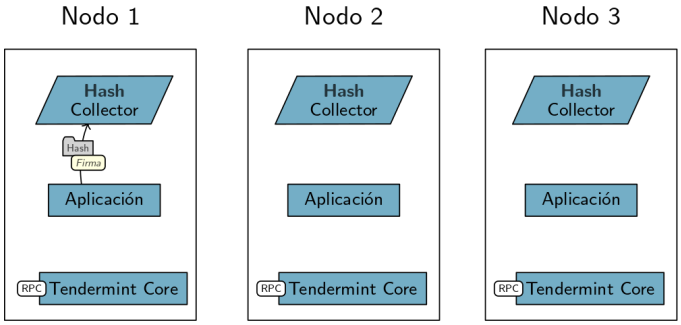
4.5.2. Algoritmo distribuido para la inversión de hashes

Debido a la naturaleza irreversible de un hash, un nodo participando de la Setchain no puede traducir inmediatamente hashes para obtener su lote de elementos original. La ausencia del lote original de elementos hace que `CheckTx` sea incapaz de verificar los elementos en la transacción y que `DeliverTx` no pueda añadir elementos a la Setchain. En este punto, nuestro algoritmo distribuido para la inversión de hashes entra en juego.

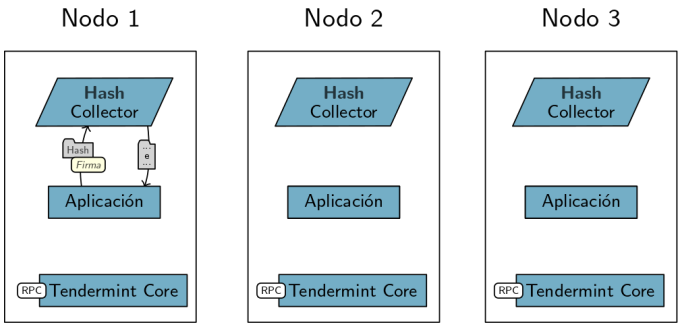
La propuesta que se hace en este trabajo es comunicarse con un nodo de Tendermint que *conoce el hash*, es decir, conoce los elementos del cual proviene. Inicialmente, el único nodo que conoce un hash es el creador del mismo. Para distribuir la información sobre quiénes conocen los datos de un hash, las transacciones contienen no solo el hash del lote, sino además una firma criptográfica. Las firmas que acompañan los hashes indican que un validador específico de Tendermint (quien firma el hash) afirma conocer el hash y, por lo tanto, si es un nodo correcto, tiene el lote original de elementos. De esta forma, las transacciones se representan con una tupla (h, s), donde h denota el valor del hash y s representa la firma obtenida al firmar h con la clave privada del nodo que conoce el hash. Gracias a la firma es posible comunicarse con el nodo que afirma conocer los datos y obtener el lote original (si el nodo se comporta apropiadamente). El proceso de petición de inversión de hashes se corre de forma asíncrona para evitar potenciales retrasos.

El *hash collector* mismo es quien se encarga de conseguir el lote asociado a un hash. En la Figura 4.4 se muestra el flujo de pedidos que se puede dar en el proceso de inversión.

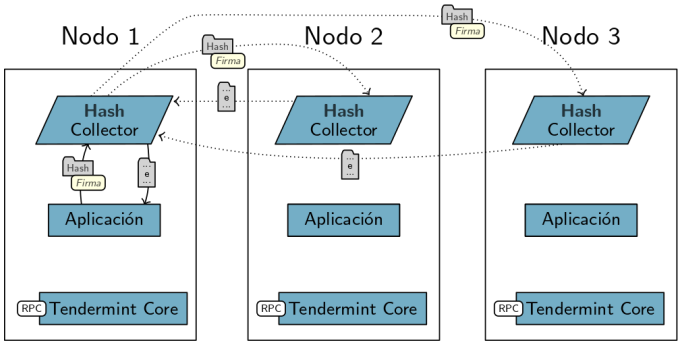
En el momento en que la aplicación necesita invertir un hash, se comunica con el *hash collector* para solicitárselo. Esto se observa en la Figura 4.4a. Si el *hash collector* conoce el lote original del hash (por ejemplo, porque fue el *hash collector* creador del mismo), entonces directamente devuelve los elementos asociados a él a la aplicación. Esto se representa en la Figura 4.4b.



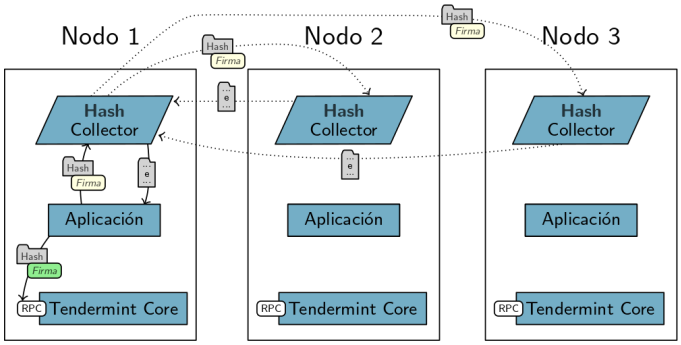
(a) La aplicación requiere la inversión de un hash



(b) El *hash collector* conoce el hash



(c) El *hash collector* no conoce el hash



(d) Se anuncia a la red el conocimiento de un hash

Figura 4.4: Flujo usual de mensajes para la inversión de hashes

Por el contrario, en la Figura 4.4c se muestra el caso en que el *hash collector* no posee el lote original, por lo que debe pedírselo al *hash collector* correspondiente. Gracias a la firma, es posible determinar quién es el *hash collector* que anuncia tener el lote. Una vez que obtiene el inverso, lo almacena para futuras peticiones y lo pasa a la aplicación. De esta forma, es transparente para la aplicación cómo se resuelve el proceso de inversión.

En la Figura 4.4d se muestra el último paso posible de esta comunicación: la aplicación anuncia a la red sobre el conocimiento de un nuevo hash. Esto ocurre solo cuando la aplicación efectivamente toma conocimiento de un nuevo hash, ya que anunciar más de una vez que se conoce un determinado hash generaría tráfico innecesario en la red. Por otro lado, solo se firma el hash si al menos un elemento en el hash es válido y nuevo, como se explicará con más detalle en la siguiente sección. En principio, los pares de claves públicas/privadas que sirven para autenticarse funcionan a nivel nodo: aplicación y *hash collector* comparten una misma clave.

Detalles de implementación

El *hash collector* se implementa, similar a su contraparte en Compresschain, como un servidor HTTP RPC. La comunicación con él por parte del cliente no cambia respecto a lo presentado para el *collector* de Compresschain. La comunicación entre la aplicación y el *hash collector*, necesaria para la inversión de hashes, se hace también de la misma manera. Es decir, la aplicación actúa como cliente para la comunicación cliente-servidor que establece con su propio *hash collector*. A su vez, es también mediante HTTP RPC que un *hash collector* a quien su respectiva aplicación le solicitó revertir un hash, se comunica con el *hash collector* correspondiente, en función de la clave pública asociada a este hash (recordar que los hashes a revertir están acompañados de una firma, de la cual es posible recuperar la clave pública de quien afirma conocer el hash).

La implementación del *hash collector* involucra una base de datos persistente capaz de almacenar las correspondencias entre hashes y lotes. A medida que los clientes envían elementos que terminan en la creación de un nuevo hash, o que nuevos hashes se descubren por medio de otro *hash collector*, se agregan nuevas entradas en esta base de datos. Se implementa como una base de datos de tipo clave-valor, utilizando el paquete *BadgerDB*. Las claves son los hashes y los valores son el lote de elementos asociados.

4.5.3. Validación de hashes

Después de dejar el *hash collector*, se espera que los hashes sean chequeados contra `CheckTx` para, o bien ser añadidos a la mempool, o bien ser descartados. Para determinar la validez de un hash se necesita chequear los elementos en él. Similar a Compresschain, los hashes son considerados válidos si al menos un elemento en el hash es válido y nuevo. Sin embargo, para un nodo dado, en el punto en que `CheckTx` corre, no podemos asegurar que el lote original de elementos sea conocido.

Hashchain es optimista en el sentido de que, ante la imposibilidad de correr el chequeo de las transacciones (debido a la ausencia del lote original), `CheckTx` considera a los hashes como transacciones válidas con la esperanza de que sus lotes originales sean enviados y conocidos más tarde.

La naturaleza optimista de Hashchain tiene consecuencia notables. Como el comportamiento por defecto de `CheckTx` es retornar `True`, se pueden tener transacciones en Tendermint sin conocer los elementos de éstas (solo conociendo los hashes). Aún más, una transacción puede terminar siendo parte de la blockchain de Tendermint sin que

nadie la haya chequeado previamente, pero pasando el chequeo por el comportamiento optimista. De hecho, este es probablemente el caso para la primera vez que un nuevo hash aparece en Tendermint, ya que fue consensuado sin que los nodos hayan chequeado realmente los elementos del lote. Por lo tanto, necesitamos definir un criterio para determinar cuándo es seguro que los elementos en un hash que está en la blockchain de Tendermint sean parte de la Setchain.

Se define un número natural `SIGNATURES_PER_HASH` que establece el número de firmas que un hash tiene que tener para que los elementos del lote asociado a él sean considerados elementos a añadir a la Setchain. Estamos únicamente interesados en hashes firmados por al menos `SIGNATURES_PER_HASH` nodos. A dichos hashes los llamamos *hashes consolidados*.

El número `SIGNATURES_PER_HASH` se define de manera tal que se garantiza que si un hash consolidó entonces al menos un nodo correcto conoce el lote original del cual proviene. Siendo f la cota superior de servidores bizantinos, podemos considerar `SIGNATURES_PER_HASH` $= f + 1$. Los elementos de hashes consolidados son los candidatos a pertenecer a la Setchain. Existen casos en los que algunos de dichos elementos pueden no ser añadidos a la Setchain, por ejemplo, si no son elementos nuevos. Es decir, si esos elementos ya fueron estampados con un número de época.

Cuando un nodo solicita la inversión de un hash, si al menos un elemento en el lote es válido y nuevo, el nodo firma el hash con sus propia clave privada y difunde el hash junto con su propia firma. Al difundir el hash y su firma, el nodo anuncia a la red que conoce el lote detrás de ese hash. De esta forma, los lotes válidos con el tiempo consolidarán, significando que sus elementos serán añadidos a la Setchain.

Para que el algoritmo mencionado funcione, se necesita mantener un registro de cuántas firmas tiene un hash, así como también un mapeo de hashes a lotes. Durante la ejecución de `DeliverTx` se chequea la validez de la firma. Si la firma es válida y nueva para el hash en cuestión, se incrementa el contador de firmas por hash. Además, cada vez que un hash es invertido, se chequea la correctitud del lote original (es decir, $hash = Hash(originalBatch)$). En caso de éxito, el mapeo de hashes a lotes es actualizado con el nuevo descubrimiento.

4.5.4. Consolidación de épocas

En esta implementación las épocas están definidas por todos los elementos en una misma transacción (es decir, un hash), similar a lo presentado para Compresschain. En otras palabras, los elementos que provienen del mismo hash consolidado, en principio, pertenecen a la misma época. Excepción a esto son los elementos de un hash que no son válidos (puesto que no serán estampados con un número de época) o los que no son nuevos, es decir, ya fueron estampados con un número de época distinto.

A diferencia de Compresschain, en donde hubiese sido sencillo implementar las épocas como un bloque de Tendermint completo (como se hizo para Vanilla), en Hashchain dicha implemtanción no sería trivial, debido a que es posible que algunos lotes (transacciones de Tendermint) nunca consoliden. Es decir, si se definiera una época como la unión de todos los lotes en un bloque de Tendermint, la época no podría considerarse completa hasta que todos los lotes en el bloque consoliden. Por lo tanto, un bloque con un lote que nunca consolida, evitaría la conformación de la época, evitando que elementos en condiciones de ser añadidos a la Setchain sean en efecto añadidos. Esta dificultad no está presente en Compresschain, puesto que todos los lotes son trivialmente posibles de descomprimir.

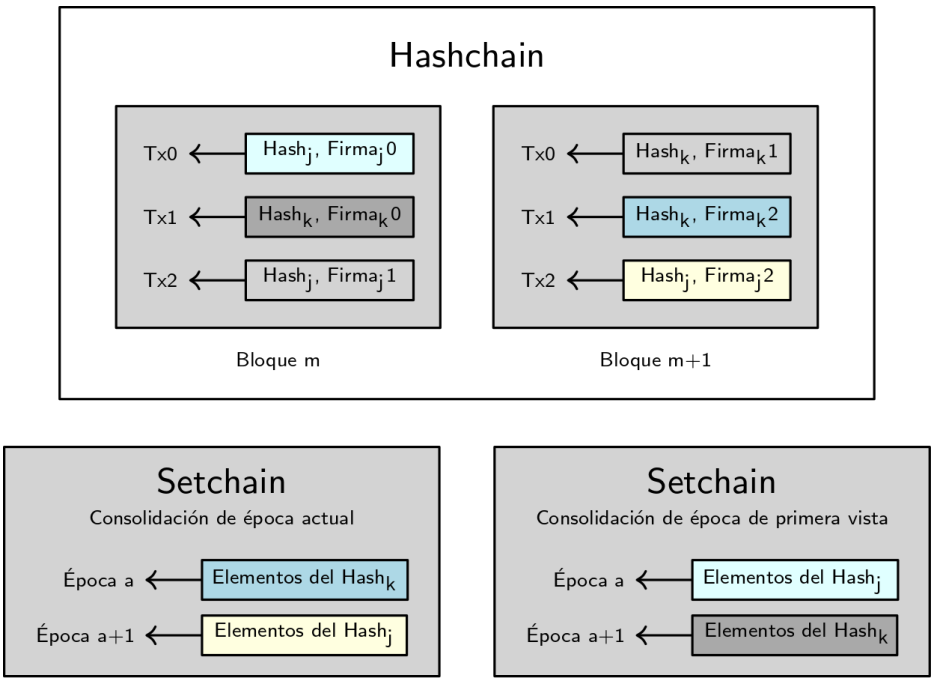


Figura 4.5: Estrategias de consolidación de épocas

En este trabajo se presentan dos alternativas para asignar un número de época a un hash consolidado. La primera se llama *Consolidación de época actual* (CEC), que significa que una vez que un hash consolida, todos sus elementos subyacentes nuevos y válidos son añadidos a la época actual (es decir, a la época en la cual la firma número `SIGNATURES_PER_HASH` es vista). La segunda estrategia, *Consolidación de época de primera vista* (FSEC), asigna la época de acuerdo a cuándo el hash fue visto por privera vez. La asignación de época se lleva a cabo una vez que el hash consolidó. Las transacciones en los bloques de Tendermint están totalmente ordenadas, por lo tanto ambas estrategias pueden determinar cuál hash ocurrió o consolidó primero, y todos los nodos correctos estarán de acuerdo en eso.

Para ver claramente la diferencia entre estas dos alternativas se presenta un ejemplo. En la Figura 4.5 se ilustran estas dos estrategias para asignar un número de época a un hash que consolidó cuando `SIGNATURES_PER_HASH` es 3. Como se mencionó en la Sección 4.5.2, las transacciones se representan con una tupla (h, s), donde h es un hash y s es una firma de dicho hash generada con la clave privada de un validador que afirma conocer los elementos de ese hash. En la figura, el hash j es el primer hash en el bloque m, y el hash k es el hash que primero consolida (es decir, el primero en obtener su tercera firma, denotada como $Firma_k2$). Por un lado, la estrategia de consolidación de época actual asigna la época tan pronto como el hash consolida, estampando los elementos en el hash k con época a primero y luego a los elementos en el hash j con la época $a + 1$. Por el otro lado, la consolidación de época de primera vista no asigna la época al hash k apenas este consolida, porque el hash j fue visto por primera vez antes que k. Dado que el hash j consolida justo después que el hash k, los elementos del hash j son estampados con la época a y los elementos del hash k, con la época $a + 1$.

La variante de consolidación de época de primera vista le otorga a los hashes un período de gracia (medido en número de transacciones ocurridas) en el cual pueden, o bien consolidarse, o bien ser descartados. Dicho período de gracia es necesario; exami-

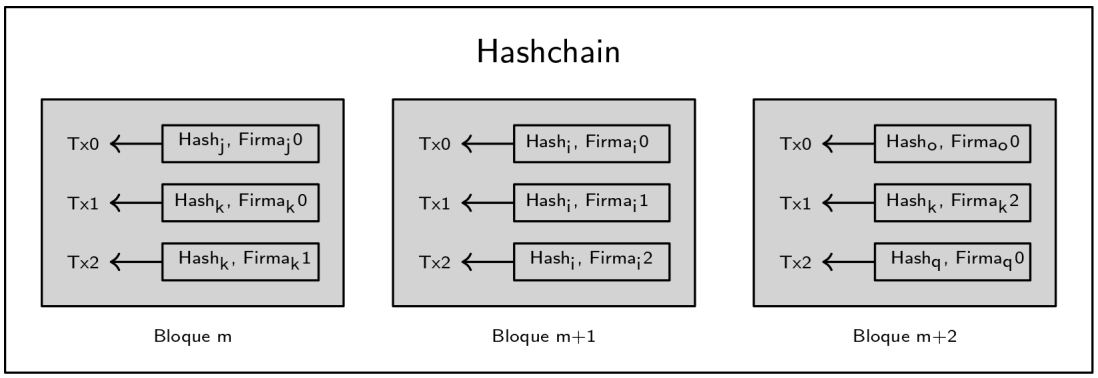


Figura 4.6: Escenario que requiere período de gracia bajo FSEC.

nemos por qué.

Suponiendo que no existe período de gracia, si un hash j apareciera por primera vez antes que el hash k, una vez consolidados, los elementos provenientes del hash j serían estampados con la época E, mientras que los elementos del hash k serían estampados con una época $F > E$. Ahora bien, si el hash j nunca lograse obtener las `SIGNATURES_PER_HASH` firmas para consolidar, el hash k (que es *más nuevo* porque fue visto por primera vez después de j) no podría consolidar, dado que sus elementos subyacentes no podrían ser estampados con una época $F > E$, si la época E no fue definida aún. En la Figura 4.6 se muestra un ejemplo de lo mencionado. Los hashes k e i (e incluso o y q) no pueden consolidarse hasta que j consolide o sea rechazada.

Se concluye entonces que, bajo la estrategia FSEC, un período de gracia es necesario para garantizar la evolución exitosa de la Hashchain. Este período de gracia debe ser medido en términos de transacciones, de modo de que existan garantías de que todos los nodos correctos estarán de acuerdo en aceptar o rechazar un hash. No es posible medir este período de gracia en tiempo (por ejemplo, segundos), siendo que no hay garantías en cuanto al momento exacto en que cada nodo recibe las transacciones. Vale la pena notar también que el aspecto crucial en la consolidación de FSEC radica en la ocurrencia de las `SIGNATURES_PER_HASH` firmas dentro de una ventana equivalente a la duración del período de gracia. Los hashes que son inicialmente rechazados pueden consolidar en el futuro siempre y cuando consigan las `SIGNATURES_PER_HASH` firmas dentro del período de gracia.

4.5.5. Algoritmos

Al igual que se hizo para Vanilla y Compresschain, en esta subsección se presentan los pseudocódigos para los distintos algoritmos.

En el Algoritmo Hash Collector se muestra una implementación posible para un *hash collector*. La definición de `AddElement` es similar a la presentada para Compresschain, con la única diferencia del uso de funciones hash en lugar de funciones de compresión. La definición de `Reverse` se corresponde con lo explicado mediante la Figura 4.4. Al recibir una petición para la inversión de un hash, si el *hash collector* ya conoce el lote asociado al hash (es decir, lo tiene en su base de datos local), lo retorna. Por el contrario, si no lo conoce, se comunica con el *hash collector* correspondiente para obtenerlo.

En el Algoritmo API Hashchain se define la API de Setchain.

Finalmente, en relación a la definición de la ABCI, se muestra una posible imple-

Algoritmo API Hashchain

```
1: function ADD(element)
2:     return HashCollector.AddElement(element)        ▷ Usar el componente intermedio.
3: function GET()
4:     return Tendermint.Query()
```

Algoritmo Hash Collector

```
1: Init: batch ← {}
2: function ADDELEMENT(element)
3:     if isValidElement(element) then
4:         encoded_element ← RLP.Encode(element)
5:         batch ← batch ∪ {encoded_element}
6:     return
7: when (isReady(batch)) do
8:     hash ← Hash(batch)
9:     DATA_BASE.Set(hash, batch)
10:    my_signature ← Sign(hash, PRIVATE_KEY)
11:    Tendermint.Broadcast(hash, my_signature)
12:    batch ← {}
13: function REVERSE(h, s)
14:    if DATA_BASE.Get(h) is not null then
15:        batch = DATA_BASE.Get(h)                    ▷ Obtener el lote de la base de datos local.
16:    else
17:        public_key = RecoverPK(s)
18:        remote_collector = BuildCollector(public_key)
19:        batch = remote_collector.REVERSE(h, s)      ▷ Obtener el lote mediante otro collector.
20:        if Hash(batch) = h then
21:            DATA_BASE.Set(h, batch)                 ▷ Añadir la nueva entrada a la base de datos local.
22:    return batch
```

mentación para Hashchain utilizando la estrategia de consolidación de época actual en el Algoritmo ABCI Hashchain - Parte 1 y ABCI Haschain - Parte 2.

Es importante notar que en el algoritmo el método `Query`, a través de la llamada a `updateHistory`, inicia la construcción de `history` y la difusión de pruebas de época, Aunque este enfoque es empleado por su claridad, un nodo que recibe consultas no muy frecuentemente podría experimentar retrasos en la generación de pruebas de época. Para evitar esto, el proceso de construcción de `history` y las pruebas de época podrían ser ejecutadas periódicamente mediante planificación, independientemente de la petición de consultas.

A diferencia de lo visto para `Query` en Vanilla y Compresschain, en este caso, la diferencia entre `the_set` y `history` no radica en las épocas a medio construir, puesto que el proceso de construcción de épocas es atómico. En este contexto, en `the_set` se encuentran todos los elementos que el nodo firma y difunde, aún antes de ser estampados con un número de época. Notemos que esto es correcto puesto que a la larga todos los nodos correctos van a recibir el lote correspondiente y firmarlo, por lo que con seguridad sus elementos tendrán un número de época asignado en el futuro.

Algoritmo ABCI Hashchain - Parte 1 Consolidación de época actual

```
1: Init: epoch ← 0, next_epoch ← 1, history ← {}, the_set ← {},
2: hash_to_signatures ← {}, epoch_to_hash ← {}, hash_to_batch ← {}
3: function CheckTx(hash, signature)
4:     if isValidSignature(hash, signature) then
5:         if hash in hash_to_batch then
6:             return isValidBatch(hash_to_batch[hash])
           ▷ Si el nodo no tiene el lote original, lanzar una rutina asíncrona que solicita el lote.
7:         spawns tryReverse(hash, signature)
8:         return True                ▷ En caso de ausencia de información, retornar True.
9:     else
10:        return False
11: function DeliverTx(hash, signature)
12:    if isValidSignature(hash, signature) then
13:        hash_to_signatures[hash] ← hash_to_signatures[hash] ∪ {signature}
14:        if not (hash in hash_to_batch) then
15:            spawns tryReverse(hash, signature)
16:        if shouldConsolidateHash(hash) then
17:            epoch_to_hash[next_epoch] ← hash
18:            next_epoch ← next_epoch + 1
19:    return
20: function Query()
21:    updateHistory
22:    return (the_set, history, epoch)
23: function updateHistory()
24:    lastEpochInHistory ← max(history) + 1
25:    for i in (lastEpochInHistory, next_epoch) do
26:        hash ← epoch_to_hash[i]
27:        if hash in hash_to_batch then
28:            elements ← getElementsFromBatch(hash_to_batch[hash])
29:            for e in elements do          ▷ Agregar solo elementos nuevos y válidos.
30:                if isValidElement(e) and not e in history then
31:                    history[i] ← history[i] ∪ {e}
32:            epoch_hash ← Hash(history[i], i)
33:            epoch_proof ← Sign(epoch_hash, PRIVATE_KEY)
34:            add(epoch_proof)
35:        else
36:            epoch ← i - 1          ▷ Guardar el número de la última época completa.
37:            break
```

4.5.6. FSEC vs CEC

En esta sección se comparan las estrategias FSEC y CEC a partir de la idea de ataques de *front-running*. En el contexto de las blockchains, un ataque de front-running [13] refiere a una situación en donde un actor malicioso explota conocimiento sobre transacciones pendientes para obtener una ventaja desleal. Debido a que las transacciones pendientes son públicas, un usuario malicioso puede observar las transacciones que genera una víctima, construir una nueva transacción y luego encontrar una forma de ubicarla antes que la transacción de la víctima. En la red Ethereum, esta "forma" de colocar una transacción antes de otra es usualmente llevada a cabo mediante pagos: o bien pagando impuestos de transacción (*transaction fees*) más altos que la víctima, o haciendo pagos

Algoritmo ABCI Haschain - Parte 2 Consolidación de época actual

```
1: function tryReverse(hash, signature)
2:     original_batch ← my_collector.Reverse(hash, signature)
3:     if Hash(original_batch) = hash then
4:         hash_to_batch[hash] ← original_batch
5:         for e in getElementsFromBatch(original_batch) do
6:             if isValidElement(e) then
7:                 the_set ← the_set ∪ {e}
8:         my_signature ← Sign(hash, PRIVATE_KEY)
9:         if my_signature ≠ signature then        ▷ Difundir la firma únicamente si es nueva.
10:            Tendermint.Broadcast(hash, my_signature)
11:    return
12: function shouldConsolidateHash(hash)
13:    return #hash_to_signatures[hash] = SIGNATURES_PER_HASH
```

directos a ciertos validadores.

Front-running en Hashchain

En esta sección analizaremos qué clase de ataque de front-running puede suceder al nivel de abstracción de Hashchain. Supongamos que un validador v recibe una transacción (h_k, s_k0), donde h_k es un hash, y s_k0 una firma de aquel hash (la primera que el validador recibe). Un validador especulador puede firmar él mismo la transacción y difundirla (el comportamiento esperado), o puede ignorarla (un posible comportamiento malicioso). Si el validador la firma y la difunde, entonces está colaborando para que el hash consolide (y, por lo tanto, los elementos provenientes de él se agreguen a la Setchain). En caso contrario, no contribuirá a que el hash consolide, potencialmente causando retrasos en la consolidación.

Las estrategias de consolidación presentadas en la sección anterior tienen distintos comportamientos. En la estrategia FSEC, si una transacción no obtiene las `SIGNATURES_PER_HASH` firmas dentro del período de gracia, el hash no consolidará en aquella ronda (pero podría hacerlo en el futuro). En la estrategia CEC, el hash no consolidará hasta obtener las `SIGNATURES_PER_HASH` firmas, sin tener ningún período de gracia para ello. En este punto, las estrategias no parecen tener diferencias fundamentales. Sin embargo, el aspecto interesante se da en el escenario alternativo: cuando las transacciones finalmente son añadidas a la Setchain.

Supongamos que v ve una transacción $t_0 = (h_k, s_k0)$ y quiere ejecutar un ataque de front-running. Para lograrlo, ignora la transacción, pero difunde su propia transacción firmada $t_1 = (h_l, s_l0)$, esperando que los elementos en ella sean finalmente estampados con un número de época menor que el número de época con que se estampa a los elementos en t_0.

Bajo el comportamiento FSEC, h_k fue visto por primera vez en un bloque anterior, por lo que si logra las `SIGNATURES_PER_HASH` firmas dentro del período de gracia, consolidará y los elementos en el hash serán estampados con una época menor a la de los elementos en t_1. Notemos que esto es completamente independiente del número de firmas que h_l logre y la velocidad a la cual lo haga. El hash h_l podría incluso consolidar (llegar a su firma `SIGNATURES_PER_HASH`) antes que h_k.

Por el otro lado, bajo el comportamiento CEC la situación es básicamente una carrera

entre h_k y h_l: el primero en obtener las `SIGNATURES_PER_HASH` firmas será el primero en consolidar y, por lo tanto, sus elementos serán estamapdos con una época más temprana.

Como ya se mencionó, la estrategia FSEC toma en consideración no solo el momento en el cual los hashes consolidan, sino también el momento en que las transacciones fueron vistas por primera vez. En principio, esta característica se puede considerar un aspecto valioso en la mitigación de ataques de front-running, dado que las transacciones observadas primero son probablmente las legítimas y originales. Sin embargo, la estrategia FSEC viene con algunos apesctos negativos no triviales que se presentan a continuación.

Dado que la época en FSEC es determinada por el momento en el cual el hash fue visto por primera vez, la estrategia de alguna manera *reserva* el número de época para los elementos en la transacción específica. Si v quiere que un hash h_k consolide únicamente si puede hacerlo antes que otro hash h_l (y no necesita conocer los elementos de h_l desde antes), v podría hacer lo siguiente. Primero, difunde la transacción de modo que todos los validadores reserven el número de época para los elementos en ella. Luego, v no revierte el hash h_k a nadie, porque quiere esperar hasta que el hash h_l aparezca y obtenga suficientes firmas. Una vez que esto sucede, y es seguro que el hash h_l consolidará, v comienza a revertir su hash h_k a todos sus pares, esperando que el hash logre las `SIGNATURES_PER_HASH` firmas dentro del período de gracia. Si es exitoso, luego v habría cumplido su objetivo: los elementos del hash h_k serían estampados con una época menor que los elementos provenientes del hash h_l.

Esta situación es particularmente interesante porque el validador v puede hacer algo suceder en el pasado (elementos del hash h_k se estampan con época e) teniendo información del futuro (elementos del hash h_l se estampan con época $e + n$). A este ataque en el contexto de Hashchain lo llamamos *el ataque del clarividente.*

En principio, intentar hacer algo similar bajo el esquema CEC no sería posible, dado que una vez que el hash h_l consolidó, sus elementos son estampados con un número de época menor a cualquier hash que aún no consolidó.

El ataque del clarividente

Volvamos a este ataque con un ejemplo. Supongamos que hay un juego de apuestas simple en donde las apuestas y el resultado final se publican en la Hashchain. El juego es sencillo: los usuarios pueden votar por A o por B. Se sabe que el resultado ganador (A o B) se publica en determinado momento, y todo aquel que haya votado por el resultado ganador recibirá el premio.

Si v es un validador que quiere ganar este juego haciendo trampa, podría votar tanto por A como por B justo antes del momento en el que se sabe que la compañía publicará el resultado ganador. Dado que v difundió dos transacciones (una votando por A y otra votando por B) antes de que la transacción con el resultado ganador aparezca, todos los validadores reserverán un número de época anterior al del resultado. Después de esto, v no revertirá el hash asociado con estas transacciones hasta que esté seguro de cuál es el resultado ganador (que efectivamente será añadido a la Hashchain). Una vez que se asegura el resultado ganador, solo dará a conocer la inversa del hash que contiene la apuesta por el resultado ganador. Si obtiene las firmas necesarias durante el período de gracia, sus transacciones serían estampadas con un número de época anterior que la transacción de los resultados, y por lo tanto, v ganaría el juego haciendo trampa.

4.5.7. Conclusión

Hashchain emplea la estrategia introducida en Compresschain mediante el uso del *collector*, agregando una capa intermedia entre el cliente y el Tendermint Core. Trabaja también sobre la misma hipótesis: hacer consenso sobre elementos agrupados de forma eficiente interpretados como unidad representa una mejora en comparación con el consenso usual, hecho sobre elementos individuales.

La novedad en Hashchain es que, en lugar de usar un algoritmo de compresión, utiliza funciones hash para reducir la sobrecarga de la red. De esta forma se profundiza la idea original presentada en Compresschain, ya que una función hash puede ser vista como un método de compresión con un ratio potencialmente muy alto (dado que convierte texto plano de longitud arbitraria en un hash de longitud fija).

Aunque reduzca el tráfico de la red (haciendo consenso sobre transacciones pequeñas y de tamaño fijo), el uso de funciones hash requiere una forma de invertirlos para así obtener los elementos originales. Para resovler esto, se diseñó un algoritmo distribuido, tolerante a fallas bizantinas, que trabaja como un objeto distribuido de resolución de hashes.

Capítulo 5

Pruebas formales

5.1. Objetivo del capítulo

En este capítulo se exponen las pruebas de correctitud de los algoritmos presentados para Vanilla, Compresschain, y Hashchain en el capítulo 4. Es decir, se prueba que dichos algoritmos satisfacen las propiedades deseadas para una implementación de Setchain introducidas en 3.5.

5.2. Propiedades de Tendermint

Dado que todas las soluciones presentadas en este trabajo son construidas sobre Tendermint, para probar la correctitud de ellas, será necesario recapitular las propiedades que la plataforma Tendermint garantiza.

5.2.1. Modelo de computación

El modelo de Tendermint considera un sistema de procesos que se comunican mediante el intercambio de mensajes. Los procesos pueden ser correctos o defectuosos, donde un proceso defectuoso se comporta de maneras arbitrarias. Es decir, se consideran procesos bizantinos.

Se asume que cada proceso tiene una cantidad determinada de *poder de voto* (el poder de voto de un proceso puede ser 0). En el contexto de Tendermint, denotamos con n el poder de voto total de los validadores en el sistema, y asumimos una cota superior f en el poder de voto total proveniente de los validadores bizantinos. El algoritmo de consenso de Tendermint asume que $n > 3f$, es decir, requiere que el poder de voto en manos de validadores bizantinos sea menor a $1/3$ del poder de voto total.

Para lograr homogeneidad con el modelo de computación presentado en 2.7, consideramos que el poder de voto de cada validador es igual a 1, y por lo tanto existen exactamente n procesos en total y a lo sumo f procesos bizantinos.

5.2.2. Propiedades nativas

Se presentan dos propiedades fundamentales de Tendermint, debidamente demostradas en [12].

Propiedad 8 (Tendermint-Agreement). Dos procesos correctos nunca deciden valores distintos.

Propiedad 9 (Tendermint-Termination). Todos los procesos correctos deciden finalmente un valor.

A continuación se presenta una propiedad que se desprende de las propiedades anteriores y de la concepción de Tendermint como máquina de replicación de estados.

Propiedad 10 (Tendermint-Global-Requests). Si un sevidor correcto recibe una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`, entonces todos los servidores correctos reciben dichas peticiones en el mismo orden. Aún más, esta serie de peticiones arriban a todos los servidores correctos en el mismo orden con respecto a cualquier otra cadena de peticiones.

5.2.3. Propiedades adicionales

En el contexto de estas demostraciones se asume una mempool de tamaño tal que nunca se satura. Es decir, el ratio de invocaciones `add` por parte de los clientes nunca llena la capacidad de la mempool, de modo que los elementos enviados por clientes nunca son descartados por mempool llena. De aquí se desprenden las siguientes propiedades.

Propiedad 11 (Tendermint-Eventual-CheckTx). Sea t una transacción. Si se invoca `Tendermint.Broadcast`(t) sobre un servidor correcto v, luego v recibirá la petición `CheckTx`(t).

Propiedad 12 (Tendermint-Eventual-Injection). Sea t una transacción en la mempool de un servidor correcto. En algún momento t formará parte de un bloque.

5.3. Pruebas de correctitud

5.3.1. Consideraciones generales

Muchas propiedades de Setchain razonan sobre el resultado (S, H, h) de una invocación a v.`get` sobre un servidor v. Siendo que en todas las implementaciones `get` retorna los valores actuales de v.`the_set`, v.`history` y v.`epoch`, en las demostraciones se razonará directamente sobre la construcción de ellos, omitiendo la invocación a `get`. Esto requiere un tratamiento especial para Hashchain, siendo que allí la definición de `Query` 20 (asociada a `get`) no solo se limita a retornar la terna mencionada sino que además inicia la construcción de `history` y difunde las pruebas de época, a través de la llamada a `updateHistory` 23. Sin embargo, como fue mencionado en la sección 4.5.5 ambos procesos podrían ser ejecutados periódicamente mediante planificación, independientemente de la petición de consultas.

A su vez, para todo servidor correcto v, v.`history` y v.`the_set` son estructuras que solo crecen, por lo que una vez que un elemento fue añadido a ellas, nunca será removido.

Por otro lado, como se presentó en el modelo de computación definido en la sección 2.7, un nodo bizantino no puede generar un elemento válido. Es decir, los elementos válidos son solo creados por clientes (no por procesos bizantinos). Además, suponemos que los elementos solo pueden ser añadidos mediante una llamada a `add`. Por lo tanto, la propiedad *Add-before-get* 7 de Setchain se cumple trivialmente para todas las soluciones presentadas en este trabajo. Notemos que consideramos de la misma manera a los elementos de cliente como a los elementos de prueba de época. Es decir, incluso un elemento de prueba de época e' fue añadido mediante una llamada a w.`add`(e') en algún servidor correcto w, donde el *cliente* que invocó tal llamada fue el mismo proceso que corre la ABCI correspondiente al servidor w.

5.3.2. Vanilla

Lema 1. La implementación Vanilla cumple la propiedad 1 *Consistent Sets* de Setchain, que establece que las épocas solo contienen elementos que provienen del conjunto de solo crecimiento.

Demostración. Esto es trivialmente correcto debido a que la construcción de `the_set` y de `history` se hace a partir de los mismos elementos, como se puede ver en las líneas 7 y 8 en el Algoritmo ABCI Vanilla. □

Lema 2. La implementación Vanilla cumple la propiedad 2 *Add-Get-Local* de Setchain, que declara que todo elemento válido añadido a un servidor correcto v es con el tiempo retornado en todas las llamadas futuras a v.`get`.

Demostración. Sea $e \in V$ y v un servidor correcto, para los cuales un cliente invoca v.`add`(e). Como $e \in V$, sabemos que e es válido (★). Por defínción de `add` en el algoritmo API Vanilla, la invocación v.`add`(e) se traduce en la llamada `Tendermint.Broadcast`(t) sobre el servidor v, donde $t = e$. Por la propiedad 11 *Tendermint-Eventual-CheckTx*, v recibirá con el tiempo la petición `CheckTx`(t). Por ★ se tiene que e es un elemento válido. Si $e \in v$.`history`, entonces, por la propiedad anterior, $e \in v$.`the_set` y la demostración es trivial. Suponemos entonces $e \notin v$.`history` (♣). Por lo cual, siendo $t = e$ y considerando la definición de `CheckTx` en la línea 2 del algoritmo ABCI Vanilla, se deduce que la llamada a `CheckTx`(t) retornará `True` en v, por lo cual t pasará a formar parte de la mempool de v. Luego, por la propiedad 12 *Tendermint-Eventual-Injection* sabemos que la transacción t formará parte de un bloque. Por definición de `DeliverTx`, dado que t es una transacción parte de un bloque, el servidor v tarde o temprano recibirá una petición `DeliverTx`(t), en donde $t = e$. Como se puede observar en la condición `if` de la línea 6 y en la definición de `isValidTransaction` 19 en el Algoritmo ABCI Vanilla, se analizan dos aspectos: que e sea válido y que e no forme parte de v.`history`. Ambas condiciones se cumplen por ★ y ♣, respectivamente. Por lo tanto, e será añadido a v.`the_set` como se indica en la línea 7, y retornado como parte de él en todas las futuras invocaciones a v.`get`. □

Lema 3. La implementación Vanilla cumple la propiedad 3 *Get-Global* de Setchain, que establece que los elementos presentes en un servidor correcto son propagados a todos los servidores correctos.

Demostración. Sean v y w dos servidores correctos y $e \in U$, tal que $e \in v$.`the_set`. Debemos probar que con el tiempo se cumplirá $e \in w$.`the_set`. Si $e \in v$.`the_set`, entonces, por construcción de `the_set`, necesariamente v en algún momento recibió una petición `DeliverTx`(t), donde $t = e$, y e es válido. Si un servidor correcto recibe una petición `DeliverTx`(t), necesariamente lo hace como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Luego, por la propiedad 10 *Tendermint-Global-Requests* todos los servidores correctos, y en particular w, reciben la misma serie de peticiones, incluyendo la petición `DeliverTx`(t). Dado que e es válido, se tienen dos opciones: o bien e es parte de w.`history`, en cuyo caso ya se cumple $e \in w$.`the_set` (ver lema 1), o bien e es un elemento nuevo ($e \notin w$.`history`). En este último caso, e será añadido a w.`the_set`, como se muestra en la línea 7 en el Algoritmo ABCI Vanilla. Por lo tanto, se cumplirá $e \in w$.`the_set`.

□

Lema 4. La implementación Vanilla cumple la propiedad 4 *Eventual-Get* de Setchain, que establece que todos los elementos añadidos en algún momento son estampados con un número de época.

Demostración. Sea v un servidor correcto y $e \in U$. Si $e \in v.\texttt{the\_set}$, entonces por construcción de `the_set`, necesariamente v en algún momento recibió una petición $\texttt{DeliverTx}(t)$, donde $t = e$, y e es válido. Como se puede ver en las líneas 7 y 8 en el Algoritmo ABCI Vanilla, inmediatamente después de agregar un elemento a $v.\texttt{the\_set}$, el elemento se agrega a $v.\texttt{history}$. Por lo tanto, con seguridad, el elemento e será en algún momento agregado a $v.\texttt{history}$. Si un servidor correcto recibe una petición $\texttt{DeliverTx}(t)$, necesariamente lo hace como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. De aquí se desprende entonces que v recibirá una petición `EndBlock` que, como se muestra en la línea 14, incrementa el número de época, provocando que, a partir de ese momento, todas las invocaciones a `get` incluyan la nueva época que contiene a e como parte de ella.

□

Lema 5. La implementación Vanilla cumple la propiedad 5 *Unique Epoch* de Setchain, que establece que un elemento puede estar en a lo sumo una época.

Demostración. Sea v un servidor correcto e $i, i' \leq v.\texttt{epoch}$ con $i \neq i'$. Queremos probar que $v.\texttt{history}(i) \cap v.\texttt{history}(i') = \emptyset$.

Esto es trivialmente correcto debido a que un elemento e se agrega a `history`[j] únicamente si no pertenece a $\texttt{history}[j']$ con $j' \in \{0, 1, ..., j-1\}$. Esto se puede ver en la cláusula `if` de la línea 6 y en la definición de `isValidTransaction` de la línea 19, presentes en el Algoritmo ABCI Vanilla.

□

Lema 6. La implementación Vanilla cumple la propiedad 6 *Consistent Gets* de Setchain, que establece que los servidores están de acuerdo en el contenido de las épocas.

Demostración. Sean v y w dos servidores correctos, e $i \leq \min(v.\texttt{epoch}, w.\texttt{epoch})$. Queremos probar que $v.\texttt{history}(i) = w.\texttt{history}(i)$.

Se demostrará $v.\texttt{history}(i) = w.\texttt{history}(i)$ por inducción fuerte en el número de época.

Caso base. Sea un elemento $e \in v.\texttt{history}(1)$ [1]. Por construcción de `history`, necesariamente v en un momento determinado recibió una petición $\texttt{DeliverTx}(t)$, donde $t = e$, y e es válido. Naturalmente, esta petición $\texttt{DeliverTx}(t)$ arriba a v como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Esto produjo que e fuera añadido a $v.\texttt{history}(1)$, como se muestra en la línea 8 en el Algoritmo ABCI Vanilla. Posteriormente, $v.\texttt{epoch}$ fue incrementado de 0 a 1 como parte del proceso de la petición `EndBlock`, como se muestra en la línea 14. De esta forma, e fue estampado con el número de época 1 en v. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que todos los servidores correctos, y en particular w, reciben la cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` en el mismo orden en que lo hace v. Por lo tanto, de forma análoga, w añadirá e a $w.\texttt{history}(1)$. Posteriormente, $w.\texttt{epoch}$ será incrementado de 0 a 1 como parte del proceso de la petición `EndBlock`. De esta forma, e es estampado con el número de época 1 en w. Así queda demostrado $v.\texttt{history}(1) \subseteq w.\texttt{history}(1)$. El mismo razonamiento se puede seguir para demostrar

[1] Por definición de Setchain, `history` empieza en la época 1.

$w.\texttt{history}(1) \subseteq v.\texttt{history}(1)$. Luego, queda demostrada la propiedad para el caso base: $v.\texttt{history}(1) = w.\texttt{history}(1)$.

Caso inductivo. Sea un elemento $e \in v.\texttt{history}(n+1)$. Análogamente a lo mencionado anteriormente, a v llegó una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` que provocó que e fuera añadido a $v.\texttt{history}(n+1)$ y, posteriormente, que $v.\texttt{epoch}$ fuera incrementado de n a n+1 como parte del proceso de la petición `EndBlock`. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que la misma cadena de peticiones llegó a w. A su vez, por la hipótesis inductiva fuerte, sabemos que $v.\texttt{history}(j) = w.\texttt{history}(j), \forall j \leq n$. Por lo tanto, podemos asegurar que al procesarse la petición $w.\texttt{DeliverTx}(t)$, donde $t = e$, la definición de `isValidTransaction` de la línea 19 retorna `True`, puesto que $e \notin w.\texttt{history}(j) \forall j \leq n$. De este modo, e es añadido a $w.\texttt{history}(n+1)$, y, posteriormente, $v.\texttt{epoch}$ se incrementa de n a n+1 como parte del proceso de la petición `EndBlock`. Así queda demostrado $v.\texttt{history}(n+1) \subseteq w.\texttt{history}(n+1)$. El mismo razonamiento se puede seguir para demostrar $w.\texttt{history}(n+1) \subseteq v.\texttt{history}(n+1)$. Luego, queda demostrada la propiedad para el caso inductivo: $v.\texttt{history}(n+1) = w.\texttt{history}(n+1)$. □

Lema 7. La implementación Vanilla cumple la propiedad 7 *Add-before-Get* de Setchain, que determina que todo elemento en `the_set` proviene del resultado de un cliente añadiendo un elemento.

Demostración. Se cumple trivialmente por lo mencionado en 5.3.1. □

5.3.3. Compresschain

Lema 8. La implementación Compresschain cumple la propiedad 1 *Consistent Sets* de Setchain.

Demostración. Esto es trivialmente correcto debido a que la construcción de `the_set` y de `history` se hace a partir de los mismos elementos, como se puede ver en las líneas 23 y 24 en el Algoritmo ABCI Compresschain. □

Lema 9. La implementación Compresschain cumple la propiedad 2 *Add-Get-Local* de Setchain.

Demostración. Sea $e \in V$ y v un servidor correcto, para los cuales un cliente invoca $v.\texttt{add}(e)$. Como $e \in V$, sabemos que e es válido (★). Por defínción de `add` en el algoritmo API Compresschain, la invocación $v.\texttt{add}(e)$ se traduce en la llamada `CompressCollector.AddElement(`e`)`. Como se puede observar en el algoritmo Compress Collector, dado que e es válido, será codificado adecuadamente y añadido a un lote t que con el tiempo estará listo[2] y se invocará $\texttt{Tendermint.Broadcast}(t)$ sobre el servidor v. Por la propiedad 11 *Tendermint-Eventual-CheckTx*, v recibirá con el tiempo la petición $\texttt{CheckTx}(t)$. Por ★ se tiene que e es un elemento válido. Si $e \in v.\texttt{history}$, entonces, por la propiedad anterior, $e \in v.\texttt{the\_set}$ y la demostración es trivial. Suponemos entonces $e \notin v.\texttt{history}$ (♣). Por lo cual, siendo e un elemento de t y considerando la definición de `CheckTx` en la línea 2 del algoritmo ABCI Compresschain, se deduce que la llamada a $\texttt{CheckTx}(t)$ retornará `True` en v, por lo cual t pasará a formar parte de

[2]Como se mencionó en la sección 4.4.2, un lote se considera listo para ser enviado una vez que, o bien alcanza un tamaño máximo, o bien una cantidad razonable de tiempo transacurrió desde que el primer elemento llegó. Por este motivo podemos asegurar que todo lote en algún momento será considerado listo.

la mempool de v. Luego, por la propiedad 12 *Tendermint-Eventual-Injection* sabemos que la transacción t formará parte de un bloque. Por definición de `DeliverTx`, dado que t es una transacción parte de un bloque, el servidor v a la larga recibirá una petición `DeliverTx`(t). Como se puede observar en la condición `if` de la línea 22 en el Algoritmo ABCI Compresschain, se analizan dos aspectos: que e sea válido y que e no forme parte de `history`. Ambas condiciones se cumplen por ★ y ♣, respectivamente. Por lo tanto, e será añadido a `the_set` como se indica en la línea 23, y retornado como parte de él en todas las futuras invocaciones a v.`get`. □

Lema 10. La implementación Compresschain cumple la propiedad 3 *Get-Global* de Setchain.

Demostración. Sean v y w dos servidores correctos y $e \in U$, tal que $e \in v$.`the_set`. Debemos probar que se cumplirá $e \in w$.`the_set`. Si $e \in v$.`the_set`, entonces, por construcción de `the_set`, necesariamente e es válido y v en algún momento recibió una petición `DeliverTx`(t), donde t es un lote comprimido de elementos, y uno de esos elementos es e. Si un servidor correcto recibe una petición `DeliverTx`(t), necesariamente lo hace como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Luego, por la propiedad 10 *Tendermint-Global-Requests* todos los servidores correctos, y en particular w, reciben la misma serie de peticiones, incluyendo la petición `DeliverTx`(t). Dado que e es válido, se tienen dos opciones: o bien e es parte de w.`history`, en cuyo caso ya se cumple $e \in w$.`the_set` (ver lema 8), o bien e es un elemento nuevo ($e \notin w$.`history`). En este último caso, e será añadido a w.`the_set`, como se muestra en la línea 23 en el Algoritmo ABCI Compresschain. Por lo tanto, se cumplirá $e \in w$.`the_set`. □

Lema 11. La implementación Compresschain cumple la propiedad 4 *Eventual-Get* de Setchain.

Demostración. Sea v un servidor correcto y $e \in U$. Si $e \in v$.`the_set`, entonces por construcción de `the_set`, necesariamente e es válido y v en algún momento recibió una petición `DeliverTx`(t), donde t es un lote comprimido de elementos, y uno de esos elementos es e. Como se puede ver en la definición de `DeliverTx` y en las líneas 23 y 24 de `newEpoch` en el Algoritmo ABCI Compresschain, inmediatamente después de agregar un elemento a `the_set`, el elemento se agrega a `history`. En algún momento la iteración sobre los elementos del lote termina y, como se muestra en la línea 28 se incrementa `epoch`. Por lo tanto, con seguridad, el elemento e será agregado a v.`history` y se producirá el incremento de v.`epoch`, estampando a e con un número de época. □

Lema 12. La implementación Compresschain cumple la propiedad 5 *Unique Epoch* de Setchain.

Demostración. Sea v un servidor correcto e $i, i' \leq v$.`epoch` con $i \neq i'$. Queremos probar que v.`history`$(i) \cap v$.`history`$(i') = \emptyset$.

Esto es trivialmente correcto debido a que un elemento e se agrega a `history`[j] únicamente si no pertenece a `history`$[j']$ con $j' \in \{0, 1, ..., j-1\}$. Esto se puede ver en la cláusula `if` de la línea 22 presente en el Algoritmo ABCI Compresschain.

□

Lema 13. La implementación Compresschain cumple la propiedad 6 *Consistent Gets* de Setchain.

Demostración. Sean v y w dos servidores correctos, e $i \leq \min(v.\texttt{epoch}, w.\texttt{epoch})$. Queremos probar que $v.\texttt{history}(i) = w.\texttt{history}(i)$.

Se demostrará $v.\texttt{history}(i) = w.\texttt{history}(i)$ por inducción fuerte en el número de época.

Caso base. Sea un elemento $e \in v.\texttt{history}(1)$. Por construcción de `history`, necesariamente e es válido y v en un momento determinado recibió una petición $\texttt{DeliverTx}(t)$, donde t es un lote comprimido de elementos, y uno de esos elementos es e. Esto produjo que e fuera añadido a $v.\texttt{history}(1)$ como se muestra en la línea 24. Posteriormente, cuando la iteración sobre los elementos del lote finaliza, $v.\texttt{epoch}$ es incrementado de 0 a 1, como se muestra en la línea 28 en el Algoritmo ABCI Compresschain. De esta forma, e fue estampado con el número de época 1 en v. Naturalmente, esta petición $\texttt{DeliverTx}(t)$ arriba a v como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que todos los servidores correctos, y en particular w, reciben la cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` en el mismo orden en que lo hace v. Por lo tanto, de forma análoga, w añadirá e a $w.\texttt{history}(1)$ y, posteriormente, $w.\texttt{epoch}$ será incrementado de 0 a 1. De esta forma, e es estampado con el número de época 1 en w. Así queda demostrado $v.\texttt{history}(1) \subseteq w.\texttt{history}(1)$. El mismo razonamiento se puede seguir para demostrar $w.\texttt{history}(1) \subseteq v.\texttt{history}(1)$. Luego, queda demostrada la propiedad para el caso base: $v.\texttt{history}(1) = w.\texttt{history}(1)$.

Caso inductivo. Sea un elemento $e \in v.\texttt{history}(n+1)$. Análogamente a lo mencionado anteriormente, a v llegó una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` que provocó que e fuera añadido a $v.\texttt{history}(n+1)$ y que $v.\texttt{epoch}$ fuera incrementado de n a n+1. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que la misma cadena de peticiones llegó a w. A su vez, por la hipótesis inductiva fuerte, sabemos que $v.\texttt{history}(j) = w.\texttt{history}(j), \forall j \leq n$. Por lo tanto, podemos asegurar que al procesarse la petición $w.\texttt{DeliverTx}(t)$, donde t es un lote comprimido de elementos, y uno de esos elementos es e, la cláusula `if` de la línea 22 retorna `True`, puesto que $e \notin w.\texttt{history}(j) \forall j \leq n$. De este modo, e es añadido a $w.\texttt{history}(n+1)$, y $v.\texttt{epoch}$ se incrementa de n a n+1. Así queda demostrado $v.\texttt{history}(n+1) \subseteq w.\texttt{history}(n+1)$. El mismo razonamiento se puede seguir para demostrar $w.\texttt{history}(n+1) \subseteq v.\texttt{history}(n+1)$. Luego, queda demostrada la propiedad para el caso inductivo: $v.\texttt{history}(n+1) = w.\texttt{history}(n+1)$. □

Lema 14. La implementación Compresschain cumple la propiedad 7 *Add-before-Get* de Setchain.

Demostración. Se cumple trivialmente por lo mencionado en 5.3.1. □

5.3.4. Hashchain

Consideraciones generales

Para estas demostraciones consideramos que dos lotes de elementos distintos nunca tienen el mismo hash asociado, es decir, no se considera la posibilidad de colisiones.

Propiedades derivadas

Con el objetivo de simplificar las pruebas, se definen y demuestran tres propiedades derivadas antes de proceder con la demostración de los lemas.

Propiedad 13 (Haschain-Global-Consolidation). Si un hash h consolida en un servidor correcto, a la larga consolidará en todos los servidores correctos.

Demostración. Sea v un servidor correcto. Sea h un hash que consolida en v. Por lo tanto, `shouldConsolidateHash`(h) retornó `True`, es decir, se recolectaron `SIGNATURES_PER_HASH` firmas para h. Dado que en `hash_to_signatures` solo se añaden firmas en la línea 13 de `DeliverTx` tras chequear que la misma sea válida (ver línea 12), concluimos que v recibió `SIGNATURES_PER_HASH` peticiones `DeliverTx` para h con distintas firmas válidas. Dichas peticiones `DeliverTx` arriban como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que la misma cadena de peticiones llegará a todos los servidores correctos. De este modo, todos los servidores correctos recibirán `SIGNATURES_PER_HASH` firmas válidas y distintas para h, de modo que dicho hash consolidará en todos los servidores correctos. □

Propiedad 14 (Haschain-Local-Consolidation). Si un hash h consolida en un servidor correcto v, tarde o temprano todos los elementos asociados a h son añadidos a v.`history`.

Demostración. A continuación se proporciona una descripción general de la demostración. Una demostración por inducción en el número de época se podría realizar siguiendo una estructura similar a la presentada en la demostración del lema 20.

Sea v un servidor correcto. Sea h un hash que consolida en v. Por lo tanto, `shouldConsolidateHash`(h) retornó `True`, es decir, se recolectaron `SIGNATURES_PER_HASH` firmas válidas y distintas para h. Dado que `SIGNATURES_PER_HASH` $\geq f + 1$, podemos asegurar que al menos un servidor correcto (llamemoslo w) firmó h. Con seguridad, dicho servidor correcto conoce el hash y, por lo tanto, garantiza que lo revertirá cuando v procese la petición `DeliverTx` asociada a h y a la firma de w (si es que v aún no tenía el lote asociado a h). De este modo, se asegura que v tiene los elementos asociados a h y, por lo tanto, podrá añadirlos a `history` cuando se procese `updateHistory`. El mismo razonamiento se puede utilizar para las épocas anteriores.

□

Propiedad 15 (Haschain-Broadcast-Consolidation). Sea v un servidor correcto. Sea e un elemento válido tal que $e \in l$, y sea h el hash asociado al lote l. Si se invoca `Tendermint.Broadcast`(h, s) en v, donde s es la firma de h con la clave privada de v, entonces tarde o temprano $e \in v$.`history`.

Demostración. Supongamos que se invoca `Tendermint.Broadcast`(h, s) en v. Por la propiedad 11 *Tendermint-Eventual-CheckTx*, v recibirá la petición `CheckTx`(h, s). Siguiendo la definición de `CheckTx` 3, si la firma es válida (s lo es puesto que v es correcto) se abren dos casos: que v conozca el hash o que aún no lo conozca. Si v no tiene el lote l asociado al hash h, entonces trivialmente retornará `True`. Si v tiene el lote l asociado al hash h, entonces si l pasa `isValidBatch`, retornará `True`. Por el contrario, si l no pasa `isValidBatch`, siendo que $e \in l$ y e es válido, necesariamente tenemos que $e \in v$.`history`. En tal caso, la propiedad queda demostrada.

Suponemos entonces que v.`CheckTx`(h, s) retorna `True`. Por lo tanto, t pasará a formar parte de la mempool de v. Luego, por la propiedad 12 *Tendermint-Eventual-Injection* sabemos que la transacción t formará parte de un bloque. Por definición de

`DeliverTx`, dado que t es una transacción parte de un bloque, el servidor v recibirá una petición `DeliverTx`(t) como parte de una cadena de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que la misma cadena de peticiones llegará a todos los servidores correctos. Sea w un servidor correcto cualquiera. Como parte de la ejecución de `DeliverTx`(t) en w, si aún no tiene el lote l asociado, w intentará revertir h comunicándose con el *hash collector* de v, quien, con seguridad, devolverá el lote asociado a h. Así, como parte de `tryReverse`, w firmará h con su propia clave privada. Esto ocurrirá para todos los servidores correctos. Siguiendo un razonamiento idéntico al ya explicado, podemos concluir que v recibirá peticiones `DeliverTx` para todas las transacciones del tipo (h, s'), donde s' es una firma válida para h generada con la clave privada de un servidor correcto, siempre que w.`CheckTx`(h, s') retorne `True` en todo servidor correcto w. De este modo, el hash consolidará en v y, por la propiedad 14 *Haschain-Local-Consolidation*, todos sus elementos serán añadidos a v.`history` (incluyendo a e). La única excepción a esto se dará en el caso en que w.`CheckTx`(h, s') retorne `False` para algún servidor correcto w, significando que $e \in w$.`history`. En ese caso, un hash h' que contiene a e consolidó en w (puesto que solo se agregan elementos a `history` cuando un hash consolida). Por la propiedad 13 *Haschain-Global-Consolidation*, con el tiempo ese mismo hash consolidará en v. A su vez, por la propiedad 14 *Haschain-Local-Consolidation* se añadirá e a v.`history`. □

Lema 15. La implementación Hashchain cumple la propiedad 1 *Consistent Sets* de Setchain.

Demostración. Sea v un servidor correcto sobre el que se invoca v.`get`. Por definición de `get` en el Algoritmo API Hashchain, esto se traduce como una llamada a `Query` en el Algoritmo ABCI Hashchain - Parte 1. Sea (`the_set`, `history`, `epoch`) el resultado de dicha invocación. Queremos probar que para todo $i \leq$ `epoch`, `history`$(i) \subseteq$ `the_set`. Sea e un elemento $\in$ `history`(i) para algún $i \leq$ `epoch`. Por construcción de `history`, e fue necesariamente agregado en la línea 31 como parte de la ejecución de `updateHistory`. Se deduce entonces que e es un elemento válido proveniente de un lote asociado al hash correspondiente a la época i.

Dado que la variable `hash_to_batch` solo se modifica en la línea 4 como parte de `tryReverse` 1, se concluye que dicho código tiene que haber sido ejecutado previamente, lo cual implica también que e fue añadido a `the_set` con anterioridad. Por lo tanto, se tiene $e \in$ `the_set`, concluyendo que `history`$(i) \subseteq$ `the_set` para todo $i \leq$ `epoch`. □

Lema 16. La implementación Hashchain cumple la propiedad 2 *Add-Get-Local* de Setchain.

Demostración. Sea e un elemento válido y v un servidor correcto, para los cuales un cliente invoca v.`add`(e). Por defínción de `add` en el algoritmo API Hashchain, la invocación v.`add`(e) se traduce en la llamada `HashCollector.AddElement(`e`)`. Como se puede observar en el algoritmo Hash Collector, e será codificado adecuadamente y añadido a un lote l que a la larga estará listo, se hasheará, se firmará, y se invocará `Tendermint.Broadcast`(t) sobre el servidor v, en donde $t = (h, s)$ será una tupla conformada por el hash del lote que contiene a e y una firma válida para el mismo (★). A su vez, la correspondencia entre el hash y el lote será añadido a una base de datos local.

Por la propiedad 11 *Tendermint-Eventual-CheckTx*, v recibirá la petición `CheckTx`(t). Consideremos la definición de la línea 3 en el Algoritmo ABCI Hashchain - Parte 1. Dado

que la firma s es válida por ★, se abren dos casos: que el lote asociado a h ya sea conocido o no.

Si el lote asociado a h ya es conocido, es decir, si h `in hash_to_batch`, entonces con certeza previamente se recibió una petición `CheckTx` o `DeliverTx` asociada al hash h, h fue revertido, y se añadió su lote asociado[3] a `hash_to_batch`. Esto únicamente se hace como parte de `tryReverse` 1, en donde, como se puede ver en la línea 7, e tuvo que haber sido añadido a `the_set`, puesto que es un elemento válido por hipótesis.

Si, por el contrario, el lote asociado a h no es aún conocido, es decir, si `not` h `in hash_to_batch`, entonces se invocará a `tryReverse`. De este modo, se comunicará con su propio *hash collector* para revertir el hash (la firma s está asociada a él). Con seguridad, el *hash collector* tendrá el lote asociado a dicho hash en su base de datos local (puesto que fue el *hash collector* creador de dicho hash) y lo retornará, de manera que el elemento e (parte del lote) será añadido a `the_set` de acuerdo a la línea 7 en el Algoritmo ABCI Haschain - Parte 2.

Por lo tanto, todas las invocaciones a v.`get` con el tiempo tendrán al elemento e como parte de `the_set`. □

Lema 17. La implementación Hashchain cumple la propiedad 3 *Get-Global* de Setchain.

Demostración. Sean v y w dos servidores correctos y $e \in U$, tal que $e \in v$.`the_set`. Debemos probar que se cumplirá $e \in w$.`the_set`. Si $e \in v$.`the_set`, entonces, por construcción de `the_set`, necesariamente v en algún momento ejecutó `tryReverse`(h, s) 1, donde h es el hash asociado a un lote que contiene a e y s es una firma válida para ese hash. De este modo, revirtió h correctamente y, como consecuencia, añadió a e a `the_set` (ver línea 7 en el Algoritmo ABCI Haschain - Parte 2). A su vez, como parte de `tryReverse`, v difundió h con su propia firma s' (a menos que s ya fuera la firma de v, en cuyo caso ya lo había difundido previamente). Esto se observa en las líneas 9 y 10. En cualquier caso, podemos asegurar que se invocó `Tendermint.Broadcast`(h, s') en v. Por la propiedad 15 *Haschain-Broadcast-Consolidation*, sabemos que a la larga $e \in v$.`history`. Esto significa que un hash h' (no se puede asegurar que $h' = h$) asociado a un lote que contiene a e consolidó en v. Por la propiedad 13 *Haschain-Global-Consolidation*, sabemos que h' consolidará en todos los servidores correctos y, en particular, en w. Luego, gracias a la propiedad 14 *Haschain-Local-Consolidation* podemos asegurar que $e \in w$.`history`. Finalmente, por el lema 15 ya demostrado, tenemos que $e \in w$.`the_set`. □

Lema 18. La implementación Hashchain cumple la propiedad *Eventual-Get* 4 de Setchain.

Demostración. Sea v un servidor correcto y $e \in U$. Si $e \in v$.`the_set` y $e \in$ `history`, luego la propiedad vale trivialmente. Suponemos que $e \notin$ `history`. Si $e \in v$.`the_set`, entonces por construcción de `the_set`, necesariamente e es válido y v en algún momento ejecutó `tryReverse` 1 para un hash h, donde el lote asociado a h contenía a e como elemento. Como parte de `tryReverse`, v difundió h con su propia firma s' (a menos que s ya fuera la firma de v, en cuyo caso ya lo había difundido previamente). En cualquier caso, podemos asegurar que se invocó `Tendermint.Broadcast`(h, s') en v. Luego, por la propiedad 15 *Haschain-Broadcast-Consolidation* podemos asegurar que tarde o temprano $e \in v$.`history`. □

[3]Por lo mencionado en las consideraciones generales de Hashchain 5.3.4, dicho lote es l, puesto que no se considera la posibilidad de colisiones.

Lema 19. La implementación Hashchain cumple la propiedad *Unique Epoch* 5 de Setchain.

Demostración. Sea v un servidor correcto e $i, i' \leq v.\texttt{epoch}$ con $i \neq i'$. Queremos probar que $v.\texttt{history}(i) \cap v.\texttt{history}(i') = \emptyset$.

Esto es trivialmente correcto debido a que un elemento e se agrega a `history`[j] únicamente si no pertenece a `history`$[j']$ con $j' \in \{0, 1, ..., j-1\}$, como se puede ver en la cláusula `if` de la línea 30 en el Algoritmo ABCI Hashchain - Parte 1. □

Lema 20. La implementación Hashchain cumple la propiedad *Consistent Gets* 6 de Setchain.

Demostración. Sean v y w dos servidores correctos, e $i \leq \min(v.\texttt{epoch}, w.\texttt{epoch})$. Queremos probar que $v.\texttt{history}(i) = w.\texttt{history}(i)$.

Se demostrará $v.\texttt{history}(i) = w.\texttt{history}(i)$ por inducción fuerte en el número de época.

Caso base. Sea un elemento $e \in v.\texttt{history}(1)$. Por construcción de `history`, necesariamente e es válido y v recibió al menos `SIGNATURES_PER_HASH` peticiones `DeliverTx`(h, s), donde h es un hash asociado a un lote que contiene a e, y s es una firma válida de h (generada con la clave privada de un servidor). Esto produjo que `hash_to_signature`$[h]$ acumulara firmas como se muestra en la línea 13 de la definición de `DeliverTx`, y que en algún momento `shouldConsolidateHash`(h) retorne `True`, asignándole el número de época 1 al hash h (ver línea 17 en el Algoritmo ABCI Hashchain - Parte 1). Luego, en la posterior ejecución de `updateHistory`, e es añadido a $v.\texttt{history}(1)$ como se muestra en la línea 31 (necesariamente h fue revertido correctamente en algún momento anterior). Naturalmente, estas peticiones `DeliverTx`(h, s) arriban a v como parte de cadenas de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit`. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que todos los servidores correctos, y en particular w, reciben las cadenas de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` en el mismo orden en que lo hace v. Por lo tanto, de forma análoga, h consolidará en w. A su vez, dado que `SIGNATURES_PER_HASH` $\geq f + 1$, podemos asegurar que al menos un servidor correcto firmó h. Con seguridad, dicho servidor correcto conoce el hash y, por lo tanto, garantiza que lo revertirá cuando w procese su petición. De esta forma, con certeza se añadirá e a $w.\texttt{history}(1)$. Así queda demostrado $v.\texttt{history}(1) \subseteq w.\texttt{history}(1)$. El mismo razonamiento se puede seguir para demostrar $w.\texttt{history}(1) \subseteq v.\texttt{history}(1)$. Luego, queda demostrada la propiedad para el caso base: $v.\texttt{history}(1) = w.\texttt{history}(1)$.

Caso inductivo. Sea un elemento $e \in v.\texttt{history}(n+1)$. Análogamente a lo mencionado anteriormente, a v llegaron cadenas de peticiones `BeginBlock`, [`DeliverTx`, ...], `EndBlock`, `Commit` que provocaron que un hash h (asociado a un lote l tal que $e \in l$) consolidara, haciendo que e fuera añadido a $v.\texttt{history}(n+1)$. Por la propiedad 10 *Tendermint-Global-Requests* podemos asegurar que las mismas cadenas de peticiones llegaron a w. A su vez, siguiendo el razonamiento explicado en el caso base, podemos asegurar que w es capaz de revertir el hash h y obtener el lote l. Por la hipótesis inductiva fuerte, sabemos que $v.\texttt{history}(j) = w.\texttt{history}(j), \forall j \leq n$. Por lo tanto, podemos asegurar que, una vez que h consolida en w, cuando se procesa `updateHistory` para la época $n+1$, la cláusula `if` de la línea 30 retorna `True`, puesto que $e \notin w.\texttt{history}(j) \forall j \leq n$. De este modo, e es añadido a $w.\texttt{history}(n+1)$. Así queda demostrado $v.\texttt{history}(n+1) \subseteq w.\texttt{history}(n+1)$. El mismo razonamiento se puede

seguir para demostrar $w.\texttt{history}(n+1) \subseteq v.\texttt{history}(n+1)$. Luego, queda demostrada la propiedad para el caso inductivo: $v.\texttt{history}(n+1) = w.\texttt{history}(n+1)$. □

Lema 21. La implementación Hashchain cumple la propiedad *Add-before-Get* 7 de Setchain.

Demostración. Se cumple trivialmente por lo mencionado en 5.3.1. □

5.4. Conclusión

Habiendo demostrado que Vanilla, Compresschain y Hashchain cumplen todas las propiedaes de Setchain presentadas en la sección 3.5, queda formalmente probado que las tres soluciones propuestas en este trabajo son implementaciones correctas de Setchain.

Capítulo 6

Evaluación empírica

6.1. Objetivo del capítulo

En este capítulo se introducen los resultados de la evaluación empírica para las implementaciones prototípicas de las tres alternativas propuestas a lo largo del trabajo: Vanilla, Compresschain, y finalmente Hashchain. La idea de los experimentos realizados es determinar la cantidad de elementos que pueden ser añadidos a la Setchain por segundo, de modo de evaluar las siguientes hipótesis:

- La implementación de Compresschain presenta mejor rendimiento que la de Vanilla.
- La implementación de Hashchain presenta mejor rendimiento que la de Compresschain.

6.2. Configuración

La evaluación fue ejecutada sobre una MacBook Pro con procesador Apple M2 Max con 12 núcleos y 64GB de RAM.

Todas las implementaciones fueron escritas en Golang y residen en un único ejecutable, el cual se encarga de inicializar los siguiente servicios.

- Un nodo de Tendermint adjunto a una versión específica de la aplicación (la cual implementa los métodos de la ABCI). Las versiones disponibles de la aplicación son: Vanilla, Compresschain, y Hashchain. La versión de Hashchain utilizada implementa la estrategia de consolidación de época actual. Todas estas versiones de la aplicación necesitan de una base de datos persistente para alojar la Setchain en sí.
- Para Compresschain y Hashchain, un servidor *collector* RPC. Este servidor se encarga de recibir elementos, armar los lotes adecuados (siguiendo la lógica correspondiente), y difundirlos mediante comunicación con el nodo de Tendermint. Por su parte, el *hash collector* también requiere una base de datos persistente para guardar el mapeo de hashes a lotes de elementos.
- Un cliente, el cual se encarga de leer e interpretar elementos, y enviarlos (comunicándose con el nodo de Tendermint en la versión Vanilla, o con el *collector* correspondiente en los otros casos). Cada cliente se comunica con un único nodo servidor.

Para las bases de datos persistentes se utilizó *BadgerDB*[4]. El servidor RPC del *collector* se implementó utilizando el paquete *net/rpc* de la librería estándar de Go. Para las funciones criptográficas se utilizó el paquete *crypto* de *go-ethereum* (la implementación del protocolo de Ethereum en Golang).

Cada instancia fue empaquetada en un contenedor Docker [6] con uso limitado a 6 núcleos y 32GB de RAM. La cantidad de nodos utilizados varió según cada experimento. Fue utilizada la herramienta Docker-Compose [5] para iniciar y correr los contenedores necesarios, y proveer una red entre ellos de manera sencilla.

6.3. Experimentos

El funcionamiento de los experimentos llevados a cabo sigue la lógica presentada a continuación. En principio, se inician las N instancias que correspondan, según la configuración del experimento. Como ya se mencionó, las instancias involucran tanto el nodo de Tendermint, como el *collector* (en caso de ser necesario), y el cliente.

Una vez inicializados los N contenedores, los clientes comienzan a procesar los elementos asignados a cada uno y a invocar `add` para enviarlos. Cada proceso cliente hace las invocaciones `add` en el servidor que reside en su mismo contenedor. Esto implica (indirectamente) comunicarse con el nodo de Tendermint en el caso de Vanilla, y comunicarse con el *collector* correspondiente en el caso de Compresschain o Hashchain.

Los experimentos funcionan en rondas. Cada experimento ejecuta 5 rondas, de 120 segundos cada una. Los clientes envían elementos cada S segundos (determinado en función del valor de `sleep_time`). Luego de cada ronda, cada cliente envía una petición `get` para registrar cuántas épocas y cuántos elementos fueron efectivamente añadidos a la Setchain. Los distintos servidores pueden diferir en la respuesta (ya que algunos pueden encontrarse atrasados respecto a otros), por lo que se realiza un promedio de estos valores. De este modo se logra estimar la cantidad de elementos añadidos por segundo.

6.3.1. Parámetros

Tendermint posee diversos parámetros que pueden configurarse. Por ejemplo, el tamaño de la mempool (en cantidad de transacciones y en cantidad de bytes) o el tamaño del bloque. El valor de dichos parámetros fue tomado de [15].

Así, los parámetros que se modifican en los experimentos son los siguientes:

- `n_nodes`: cantidad de nodos corriendo
- `sleep_time`: cantidad de tiempo que espera el cliente antes de añadir un nuevo elemento
- `collector_size`: cantidad de elementos que el correspondiente *collector* aloja antes de generar un lote (no aplicable a Vanilla).

La idea es ajustar los valores de los parámetros `sleep_time` y `collector_size`, de modo de obtener, para una cantidad determinada de nodos en la red, la máxima cantidad de elementos que pueden ser añadidos a la Setchain para Vanilla, Compresschain y Hashchain. Es importante notar que la búsqueda del valor máximo siempre debe hacerse sobre ejecuciones en donde la cantidad de elementos enviados y añadidos a la Setchain

se mantenga constante a lo largo de todas las rondas. En caso contrario, los resultados se descartan.

6.3.2. Elementos

Los elementos utilizados por los procesos clientes durante los experimentos son transacciones provenientes de Arbitrum [30], una tecnología de tipo *Optimistic Rollup* [1].

La elección de los elementos fue tomada priorizando que los experimentos sean fieles a un contexto real, respetando el contenido de las transacciones. Fueron descargadas utilizando las API de Alchemy y de Chainstack[2], las cuales permiten conectarse directamente a un nodo específico de Arbitrum que es parte de la red, de modo de interactuar con los datos en la cadena y hacer las peticiones necesrias para descargar las transacciones.

Al analizar el conjunto de transacciones descargadas (alrededor de 3 millones de transacciones) se encontró que el tamaño de estas presentaba algunos valores extremos hacia la derecha. Es decir, mientras que la mediana rondaba los 350 bytes, y la desviación estándar los 600 bytes, los valores mínimos y máximos eran de 110 bytes y 50 mil bytes, respectivamente. Por tal motivo se decidió remover los valores extremadamente altos. Siguiendo los valores modelo presentados en [15], se limitó a utilizar transacciones de hasta 1KB de tamaño. De este modo, analizando aproximadamente 100 mil transacciones provenientes del conjunto de datos usado durante los experimentos (solo transacciones de hasta 1KB de tamaño), se obtuvieron los siguientes resultados.

- Mediana: 340 bytes
- Desviación estándar: 220 bytes
- Min: 110 bytes
- Max: 1024 bytes

6.4. Resultados

A continuación se presentan los resultados obtenidos al ejecutar los experimentos para las tres alternativas, con redes de 4, 10 y 20 nodos. A su vez, los tamaños de *collector* utilizados fueron de 800, 5000 y 10000 elementos.

6.4.1. Comentarios generales

La primera observación que se hizo al analizar los resultados obtenidos es que el ratio al que los clientes envían elementos debe ser elegido de modo de no saturar la mempool. Esto quiere decir que debe evitarse enviar elementos a la mempool más rápidamente de lo que los elementos pueden salir de ella.

Cuando un cliente invoca `add` en un servidor cuya mempool está llena[3], el elemento es rechazado y el cliente recibe un error. A su vez, cuando se recibe un error por mempool

[1] Arbitrum es una cadena lateral que corre en paralelo a la red principal de Ethereum. También conocida como una solución de escalabilidad de capa 2 (L2), Arbitrum mejora la velocidad de las transacciones así como los costos en comparación a la red principal. Arbitrum fue fundada en 2021 y es al momento una de las tecnologías de capa 2 más populares en el mercado.

[2] Disponibles en `https://www.alchemy.com/` y `https://chainstack.com/`, respectivamente.

[3] Siguiendo los valores utilizados en [15] la mempool puede alojar hasta 5000 transacciones con un tamaño máximo total de 1GiB.

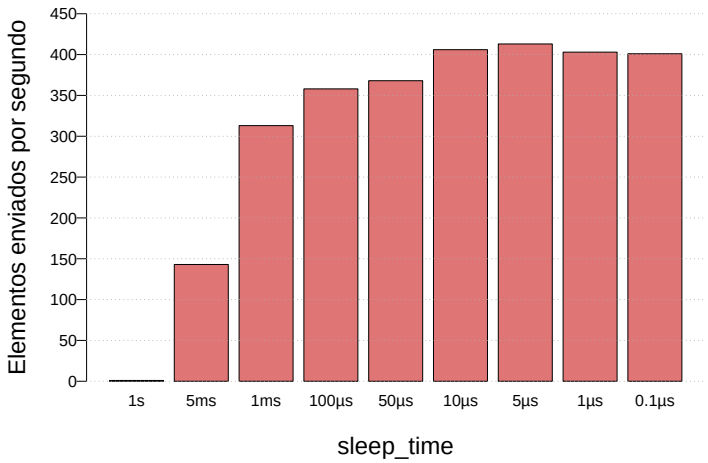


Figura 6.1: Promedio de elementos enviados por nodo variando `sleep_time` en un experimento con 4 nodos para Vanilla.

llena, el cliente espera una cierta cantidad de tiempo para intentar nuevamente el envío del elemento. Esto produce un degradamiento en el ratio de elementos enviados por segundo.

Por su parte, si a medida que pasan las rondas del experimento se envían cada vez menos elementos, menos elementos pueden ser añadidos a la Setchain, generando también un degradamiento en la cantidad de elementos añadidos por segundo.

Es por esto que los resultados estables son aquellos que no llegan a saturar la mempool en ningún momento del experimento. Con *resultado estable* nos referimos a que la cantidad de elementos enviados por parte de los clientes así como también la cantidad de elementos añadidos a la Setchain se mantiente similar en cada una de las rondas y en cada uno de los nodos.

6.4.2. Vanilla

Los experimentos para Vanilla fueron los más sencillos de llevar a cabo debido a que, variando la cantidad de nodos, el único parámetro que debía ajustarse era `sleep_time`.

A modo de ejemplo, en la Figura 6.1 se muestra la cantidad de elementos enviados en promedio por nodo por segundo, al variar el valor de `sleep_time`, durante todas las rondas de un experimento con 4 nodos. Allí se puede observar que no es posible enviar mucho más de 400 elementos por segundo, dado que incluso disminuyendo el tiempo que se espera antes de volver a invocar `add`, la cantidad de elementos enviados por segundo no aumenta. Esto es porque justamente al mandar elementos con demasiada frecuencia, la mempool se satura y los elementos comienzan a ser rechazados, lo que genera una degradación en el ratio de elementos enviados por segundo.

A su vez, es interesante notar que la cantidad de elementos enviados en cada caso no es proporcional a la variación del valor del parámetro `sleep_time`. Esto se explica considerando el hecho de que la invocación a `add` tarda un tiempo no despreciable en comparación con los valores de `sleep_time`.

En la Figura 6.2 se muestra una comparación de la máxima cantidad de elementos añadidos en Vanilla para experimentos con 4, 10 y 20 nodos. Se nota una clara degrada-

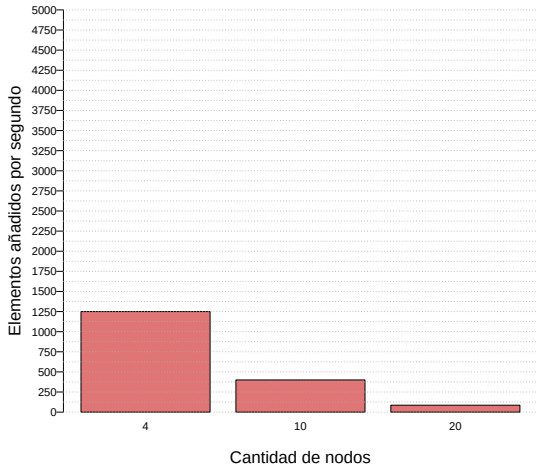


Figura 6.2: Elementos añadidos a la Setchain por segundo según la cantidad de nodos para Vanilla.

ción en la cantidad de elementos añadidos al aumentar la cantidad de nodos corriendo en el experimento. Esta baja es esperable, sin embargo, es importante notar el siguiente punto. Mientras más entidades (clientes y servidores) participen del consenso dentro de la red de Tendermint el experimento se hace más exigente en cuanto a recursos (procesador y memoria RAM). Dado que el límite de recursos destinados a Docker se mantiene constante durante todos los experimentos (independientemente de la cantidad de nodos), se genera una degradación en la cantidad de elementos que los clientes pueden enviar por segundo. Naturalmente, y como ya se mencionó antes, al disminuir el ratio de elementos enviados por segundo, invariablemente también lo hace el ratio de elementos añadidos por segundo.

6.4.3. Compresschain

En los experimentos para Compresschain se utilizaron los tres parámetros ya mencionados: `n_nodes`, `sleep_time` y `collector_size`. La cantidad de nodos varió, al igual que para Vanilla, en 4, 10 y 20 nodos. Para ajustar el parámetro `sleep_time` se utilizó el mismo criterio que para Vanilla, pero se obtuvieron valores menores para los cuales los experimentos resultaron estables y sin saturación de la mempool. Por otro lado, para el *collector* se utilizaron tamaños de 800 y 5000 elementos. No fue posible utilizar un tamaño de 10000 elementos puesto que el lote comprimido de 10000 elementos era más grande que lo que la configuracion usada de Tendermint permitía para una transacción.

Una observación que se hizo al realizar estos experimentos es que, para los mismos valores de `sleep_time`, el cliente es capaz de enviar más elementos en Compresschain que en Vanilla. Esto se atribuye principalmente a dos motivos que se presentan a continuación.

Por un lado, la mempool no se satura tan fácilmente. Esto evita, por ejemplo, que el cliente espere un determinado tiempo extra (más allá del `sleep_time`) para volver a intentar el envío de un elemento cuando recibe un mensaje de error por mempool llena, como fue mencionado en la sección anterior. Decimos que no es tan fácil saturar la mempool puesto que no llegan a ella los elementos directamente enviados por el cliente,

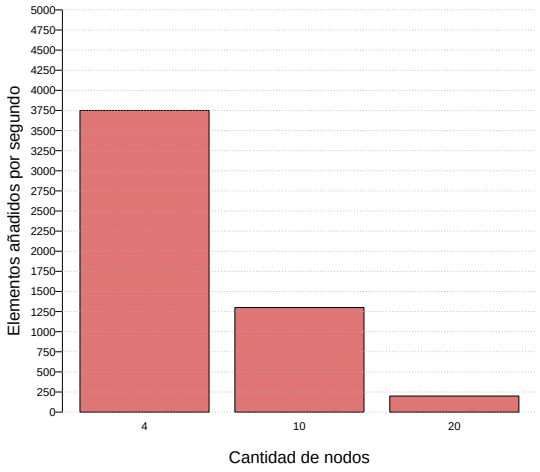


Figura 6.3: Elementos añadidos a la Setchain por segundo según la cantidad de nodos para Compresschain.

sino que llegan los lotes construidos por el *collector*. Esto significa que, aunque el cliente envíe la misma cantidad de elementos en Vanilla que en Compresschain, a la mempool llegan muchos menos (dependiendo del tamaño del *collector* utilizado).

Por otro lado, durante los experimentos se comprobó que las llamadas por parte del cliente al servidor RPC del *collector* son considerablemente más rápidas que las llamadas al servidor RPC nativo de Tendermint. Esto tiene como consecuencia directa que, a mismo `sleep_time`, la cantidad de elementos enviados por parte del cliente sean menores en Vanilla que en Compresschain.

En cuanto al tamaño del *collector*, los resultados arrojados por los experimentos de 4 nodos no mostraron diferencias significativas al usar uno o el otro. Sin embargo, en experimentos con 10 o 20 nodos, el uso de un *collector* de mayor tamaño implica un mayor rango en la cantidad de elementos añadidos a la Setchain por los distintos nodos. Esto es esperable dado que, cuanta más cantidad de elementos contenga un lote, mayor será la diferencia de elementos totales añadidos entre un nodo que ya agregó un determinado lote y un nodo que aún no lo hizo.

En la Figura 6.3 se muestra una comparación de la máxima cantidad de elementos añadidos en Compresschain para experimentos con 4, 10 y 20 nodos.

6.4.4. Hashchain

En el caso de Hashchain, los experimentos fueron ejecutados variando el tamaño del *collector* en 800, 5000 y 10000 elementos. Al igual que en Compresschain, al aumentar la cantidad de elementos que conforman un lote, aumenta el rango en la cantidad de elementos añadidos por los distintos nodos, lo que se hace aún más notable en experimentos con 10 y 20 nodos en la red.

En la Figura 6.4 se muestra una comparación de la máxima cantidad de elementos añadidos en Hashchain para experimentos con 4, 10 y 20 nodos.

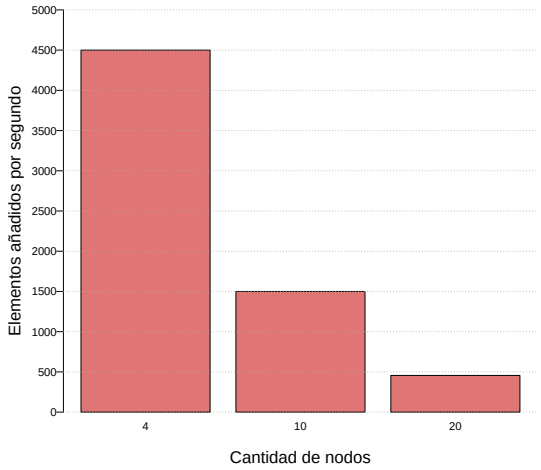


Figura 6.4: Elementos añadidos a la Setchain por segundo según la cantidad de nodos para Hashchain.

6.4.5. Conclusión

En la Figura 6.5 se muestra la comparación de todos los resultados presentados previamente. Allí se puede observar que el rendimiento de Hashchain es mejor que para las otras alternativas en todos los casos.

Cabe señalar que estos resultados corresponden a un entorno controlado de laboratorio, en donde todos los procesos corren en una misma máquina física. En un escenario de red real, donde las máquinas están físicamente distribuidas, los recursos no son compartidos y tanto las latencias como el ancho de banda suelen ser mayores, es posible que Hashchain ofrezca un rendimiento aún más favorable. Esto se debe a que el consenso en Hashchain se realiza sobre fragmentos de datos considerablemente más pequeños en comparación con las otras alternativas, lo que podría, en condiciones reales de fallas en la red, representar mejoras más significativas.

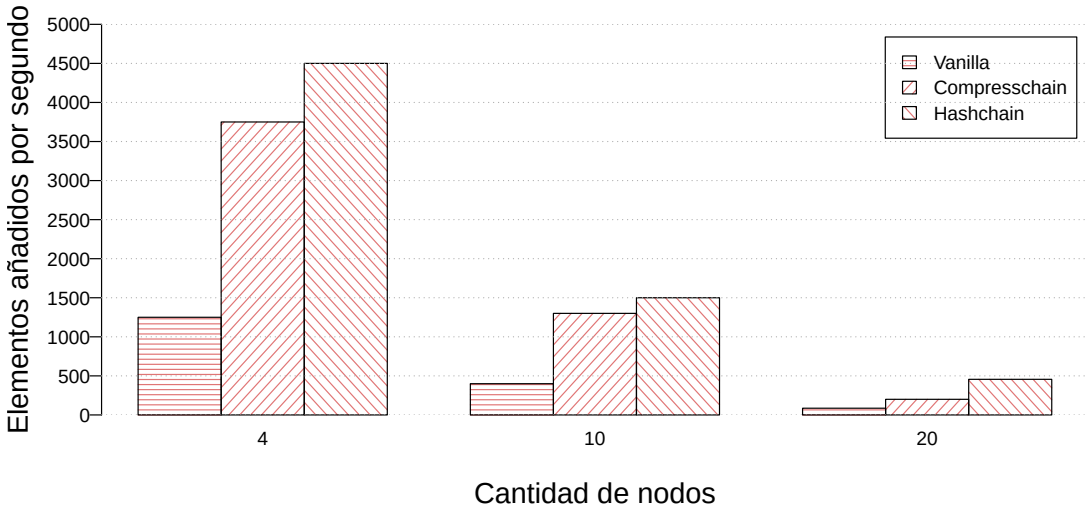


Figura 6.5: Elementos añadidos a la Setchain por segundo según cantidad de nodos y versión.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Las tecnologías blockchain están atravesando un problema de escalabilidad. Comparado a otros sistemas basasdos en transacciones, las blockchains son simplemente demasiado lentas. Existen, por lo menos, dos formas posibles de resolver este problema: o bien diseñando algoritmos de consenso más rápidos, o bien colocando soluciones por encima de la blockchain.

Un ejemplo de lo último es lo que se conoce como soluciones de *Layer-2* (capa 2), donde se tiene una máquina de computación sobre la blockchain que consume menos recursos que si se actuara directamente sobre ella. Un ejemplo de lo primero es Setchain.

Setchain presenta un algoritmo de consenso más rápido que abandona el clásico orden total de las transacciones en las blockchains. El orden parcial de Setchain establece que no es posible comparar los elementos en una misma época. Sin embargo, se puede decir que los elementos de una época ocurrieron antes o después que los elementos de otra. Esto es, Setchain propone un orden relativo, en donde los elementos de distintas épocas se pueden relacionar, pero no es posible relacionar elementos en la misma época.

Aunque Setchain es una idea inteligente, carecía al momento de una implementación de mundo real. Los autores de Setchain dieron una implementación a modo de prueba de concepto, que les permitió observar que, en condiciones de laboratorio, era más rápido que otros algoritmos de consenso.

La implementación de sistemas distribuidos es difícil, por lo que en este trabajo se decidió utilizar una solución prefabricada: Tendermint. Tendermint ofrece una implementación de mundo real de una estructura basada en blockchain, presentando una separación clara entre las primitivas de la blockchain y la interfaz definida por el usuario.

Como contribuciones originales de este trabajo se presentaron tres alternativas posibles para la implementación de Setchain construidas sobre Tendermint. Dado que los algoritmos de Setchain presentados en [14] emplean bloques primitivos como *Set Byzantine Consensus* no es posible replicar los mismos comportamientos. Sin embargo, tomamos un enfoque ligeramente diferente, intentando obtener tanto como se pueda de lo que Tendermint es capaz de ofrecer. Cada alternativa presentada trata de acercarse a la implementación ideal de Setchain, explotando la idea de que cada época es un conjunto de elementos.

La primera solución (ver Sección 4.3) presenta una simple y correcta pero ineficiente implementación de Setchain donde Tendermint solo mantiene un conjunto de solo crecimiento.

La segunda solución (ver Sección 4.4) mejora la alternativa anterior usando un algoritmo de compresión: las transacciones de Tendermint ahora son conjuntos comprimidos de elementos.

Finalmente, la tercera solución (ver Sección 4.5) va más allá de la segunda alternativa y usa funciones hash para reducir la sobrecarga en la red. Aunque reduzca el tráfico haciendo consenso sobre transacciones pequeñas y de tamaño fijo (es decir, sobre hashes), el uso de los mismos requiere de una forma de invertirlos para así obtener los elementos originales enviados por los clientes. Para resovler esto, se diseñó un algoritmo distribuido, tolerante a fallas bizantinas, que trabaja como un objeto distribuido de resolución de hashes. De este modo se conforma una implementación de Setchain de mundo real.

Los números provistos por la evaluación empírica son prometedores, indicando que, efectivamente, Compresschain permite un rendimiento mejor que Vanilla y, a su vez, Hashchain presenta un rendimiento superior a Compresschain. Estos resultados se presentan a favor de la hipótesis de que hacer consenso sobre elementos agrupados de forma eficiente interpretados como unidad representa una mejora en comparación con el consenso usual, hecho sobre elementos individuales.

7.2. Trabajo futuro

7.2.1. Desarrollo de Hashchain

Una posible línea de trabajo futuro es la continuación del desarrollo de Hashchain. Esto se puede abordar, por un lado, implementando la versión de Hashchain que utiliza *consolidación de época de primera vista* (ver Sección 4.5.4), ya que en este trabajo se implementó únicamente la versión de *consolidación de época actual*, con la cual se realizó la evaluación empírica.

Otra posibilidad de continuación de Hashchain se da a través del desarrollo de una *Hashchain pesimista*. Esta nueva versión cambiaría el comportamiento por defecto del resultado de `CheckTx` ante la imposibilidad de correr el chequeo de un hash debido a la ausencia del lote original. Actualmente, el comportamiento por defecto es considerar a un hash como válido. Una Hashchain pesimista consideraría inválidos a los hashes como conducta predeterminada. Esto cambiaría el funcionamiento de la Hashchain ya que, por ejemplo, los elementos serían posiblemente rechazados en un principio, pero una vez consensuados podrían ser agregados a la Setchain quizás sin necesidad de una estrategia de consolidación.

Por último, en la sección 4.3.1 se presentó a la estrategia de almacenamiento de elementos y épocas en la base de datos como un aspecto con múltiples abordajes: dependiendo del caso de uso de Setchain se opta por una forma u otra, con el objetivo de mantener eficientes las búsquedas. Una perspectiva de desarrollo interesante es ir incluso más allá de la definición primitiva de `get` (que se limita a retornar la terna `the_set`, `history`, `epoch`) y tener una Hashchain que provea a los usuarios diversos métodos de hacer consultas simples sobre el contenido de la Setchain. Ejemplos de estos nuevos métodos son:

- *find_element*: dado un elemento retorna un booleano indicando si el elemento pertenece o no a alguna época, junto con una prueba de ello.

- *get_epoch*: dado un número de época, retorna los elementos de la misma.

La idea principal es poder desarrollar métodos de búsqueda *confiables*, es decir, donde además de retornar el objeto pedido, se presenta una prueba de pertenencia que el cliente puede verificar sencillamente mediante el uso de *estructuras de datos autenticadas* [41].

7.2.2. *Deployments* de Hashchain

Otro aspecto de posible trabajo futuro incluye la evaluación empírica con *deployments* de Hashchain similares a los presentados en [15]. Es decir, utilizando un entorno distribuido geográficamente, con nodos desparramados uniformemente por distintas regiones en todos los continentes.

De esta forma podrían analizarse cuestiones que quedaron por fuera del alcance de las evaluaciones realizadas en este trabajo. Por ejemplo, cómo reaccionan las distintas soluciones a problemas en la red (menos probables de ocurrir cuando se está utilizando docker compose en una misma máquina), o cómo se ve afectado el rendimiento cuando los nodos se encuentran a una distancia física considerable.

7.2.3. Aplicación de Hashchain

Una dirección de trabajo adicional comprende el desarrollo de las aplicaciones que motivan la existencia de la Setchain: aquellas que no requieren de un orden total. Por ejemplo, la implementación de mempools y sistemas de *optimistic rollup* de capa 2 a través de Setchains.

A su vez, la Setchain puede ser usada como un mecanismo para mitigar ataques de tipo *front-running*. La mempool almacena transacciones solicitadas por los usuarios, por lo que observar la mempool permite predecir las operaciones futuras. *Front-running* es la acción de inyectar maliciosamente transacciones para que se ejecuten antes que las observadas [20, 42] (pagando una tarifa más alta a los mineros). Setchain puede usarse para detectar ataques de este tipo, dado que puede servir como un mecanismo básico para construir una mempool que sea eficiente y sirva como un registro de solicitudes. Además, las Setchains pueden utilizarse como un bloque de construcción para evitar los ataques de *front-running*, donde los usuarios cifran sus solicitudes utilizando un esquema de cifrado de firma múltiple. Una vez que el orden de las solicitudes fue fijado, los servidores de desencriptación de los participantes desencriptan las solicitudes.

Por último, estudiar cómo equipar las blockchains existentes con una Setchain (sincronizando bloques y épocas) para permitir a los contratos inteligentes acceder a la Setchain como parte de su almacenamiento es otra ruta de trabajo futura.

Capítulo 8

Glosario

A continuación se presentan las definciones de algunos conceptos que fueron utilizados en inglés a lo largo del trabajo, debido a la dificultad de ser traducidos razonablemente.

- *Application Programming Interface (API).* Pieza de código que permite a diferentes aplicaciones comunicarse entre sí, compartiendo información y funcionalidades.

- *Commit.* Acción de guardar o confirmar un nuevo bloque en la blockchain, haciéndolo parte permanente de ella.

- *Commited by the network.* Un bloque se considera así cuando una cantidad necesaria de nodos en la red lo validaron y agregaron a la blockchain mediante el mecanismo de consenso que utilizan, asegurando la inmutabilidad.

- *Deployment.* Mecanismo por el cual un determinado software se disponibiliza para los usuarios en un entorno particular.

- *Distributed grown-only sets.* Conjuntos distribuidos que solo crecen.

- *Endpoint.* Una URL específica asociada a un recurso particular. Cuando se interactúa con una API, en general los endpoints ejecutan actividades específicas, como solicitar datos o iniciar procesos.

- *Fork.* El fork de un proyecto ocurre cuando desarrolladores toman una copia del código fuente de un software particular, y comienzan un desarrollo independiente a partir de él, creando una pieza de software distinta.

- *Front-running attack.* Ataque de adelantamiento. Estrategia maliciosa en la cual un actor observa una transacción que está pendiente de ser confirmada en una blockchain y, basándose en esta información, introduce una transacción propia con el objetivo de adelantarse a la transacción original para obtener un beneficio.

- *Gossip protocol.* Protocolo de comunicación que permite intercambiar información entre entidades de un sistema distribuido, de modo que la información se disemina a todos los miembros de la red, sin necesidad de comunicación directa entre todos ellos.

- *Optimistic Rollup.* Protocolos de capa 2 diseñados para ampliar el rendimiento de la capa base. Reducen el cómputo en la cadena principal al procesar las transacciones fuera de la cadena, ofreciendo mejoras significativas en la velocidad de procesamiento.
- *Proof-of-work.* Prueba de trabajo. Mecanismo por el cual el probador demuestra que realizó un esfuerzo computacional lo suficientemente costoso.
- *Proof-of-stake.* Prueba de participación. Mecanismo por el cual el probador demuestra la propiedad de algún bien valioso, en general, alguna cryptomoneda
- *Recursive Length Prefix (RLP).* Algoritmo de codificación estándar de Ethereum para la transferencia de datos entre nodos en formato eficiente en espacio.
- *Remote Procedure Call (RPC).* Llamada a procesamiento remoto. Es una forma de comunicación inter-procesos, en donde un programa hace que un procedimiento se ejecute en un espacio de memoria diferente (comúnmente en otra computadora en una red compartida). El procedimiento está escrito como si fuese una llamada a un procedimiento local, sin necesidad de que el programador dé los detalles explícitos de la interacción remota.
- *Timeout.* Período de tiempo máximo que un proceso está dispuesto a esperar por una determinada operación o evento. En general tiene asociado una acción a realizar si dicho período de tiempo se termina antes de que la operación o el evento esperado ocurra.
- *Transaction fees.* Impuesto de transacción. Es una pequeña cantidad que se cobra a los usuarios que envían transacciones a la red blockchain.

Capítulo 9

Bibliografía

[1] B-money. http://www.weidai.com/bmoney.txt.

[2] Block chain. https://developer.bitcoin.org/devguide/block_chain.html.

[3] Cometbft repository. https://github.com/cometbft/cometbft.

[4] Dgraph: dgraph-io/badger. https://github.com/dgraph-io/badger.

[5] Docker compose docs. https://docs.docker.com/compose.

[6] Docker docs. https://docs.docker.com/.

[7] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A general-purpose data compressor. 2018.

[8] Andreas M. Antonopoulos. *Mastering Bitcoin. Programming the Open Blockchain.* O'Reilly Media, 2017.

[9] Adam Back. Hashcash - a denial of service counter-measure. 2002.

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive Zero Knowledge for a von Neumann architecture. In *Proc. of USENIX Sec.'14*, pages 781–796. USENIX, August 2014.

[11] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[12] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[13] A. Canidio and V. Danos. Commitment against front-running attacks. https://arxiv.org/pdf/2301.13785.pdf, 2023.

[14] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. Setchain: Improving Blockchain scalability with Byzantine distributed sets and barriers. In *Proc. of the 2022 IEEE International Conference on Blockchain*, pages 87–96. IEEE, 2022.

[15] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. The design, architecture and performance of the tendermint blockchain network. 2021.

[16] Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, Michel Raynal, and Antonio Russo. Byzantine-tolerant distributed grow-only sets: Specification and applications. *CoRR*, abs/2103.08936, 2021.

[17] Joan Coromines. *Breve diccionario etimológico de la lengua castellana.* Gredos, 1990.

[18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483, 2021.

[19] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *Financial Crypto. and Data Security*, pages 106–125. Springer, 2016.

[20] Philip Daian, Steven Goldfeder, T. Kell, Yunqi Li, X. Zhao, Iddo Bentov, Lorenz Breidenbach, and A. Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. *Proc. of S&P'20*, pages 910–927, 2020.

[21] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proc. of SIGMOD'19*, pages 123—-140. ACM, 2019.

[22] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, dec 2004.

[23] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language.* Adison-Wesley, 2015.

[24] Antonio Fernández Anta, Chryssis Georgiou, Maurice Herlihy, and Maria Potop-Butucaru. *Principles of Blockchain Systems.* Morgan & Claypool Publishers, 2021.

[25] Antonio Fernández Anta, Kishori Konwar, Chryssis Georgiou, and Nicolas Nicolaou. Formalizing and implementing distributed ledger objects. *ACM Sigact News*, 49(2):58–76, 2018.

[26] L. M. Goodman. Tezos – a self-amending crypto-ledger. `https://www.tezos.com/whitepaper.pdf`, 2014.

[27] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.

[28] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). 2001.

[29] Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2019:352, 2019.

[30] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*, pages 1353–1370. USENIX Assoc., 2018.

[31] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012.

[32] Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.

[33] L. Lamport, R. Shostak, , and P. Marshall. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[34] Daniela Mechkaroska, Vesna Dimitrova, and Aleksandra Popovska-Mitrovikj. Analysis of the possibilities for improvement of blockchain technology. In *2018 26th Telecommunications Forum (TELFOR)*, pages 1–4, 2018.

[35] Ralph Merkle. A digital signature based on a conventional encryption function. 1987.

[36] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009.

[37] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[38] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach.* 01 2018.

[39] P. Rogaway and M. Bellare. Pss: Provably secure encoding method for digital signatures. 1998.

[40] Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.

[41] Roberto Tamassia. Authenticated data structures. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003*, pages 2–5, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[42] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the Dark Forest: An empirical study of frontrunning on the Ethereum blockchain. In *Proc of USENIX Sec.'21*, pages 1343–1359, 2021.

[43] Shobha Tyagi and Madhumita Kathuria. *Study on Blockchain Scalability Solutions*, page 394–401. ACM, 2021.

[44] Ke Wang and Hyong S. Kim. Fastchain: Scaling blockchain system with informed neighbor selection. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 376–383, 2019.

[45] David Wong. *Real-World Cryptography.* Manning, 2021.

[46] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.