UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Validación de Transacciones Cross Blockchain

Informe de Proyecto de Grado presentado por

Facundo Torterola, Guzman Martinez

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Guzmán Llambías

Montevideo, 8 de septiembre de 2024

Agradecimientos

Yo, Guzman, quiero agradecer principalmente a mis padres, quienes han sido mi mayor soporte a lo largo de este proyecto y toda mi trayectoria en la Universidad. También quiero agradecer a mi novia, hermanos y amigos por acompañarme en el proceso. Yo, Facundo, quiero agradecer a mi madre, padre, hermanas, novia y amigos, quienes me apoyaron y acompañaron en este camino a lo largo de la facultad. Por último, quiero hacer una mención especial a mi abuela, quien me despertó el interés por la matemática y la ingeniería. Finalmente, los dos miembros del equipo queremos hacer una mención especial, nuestro tutor Guzmán, que fue una pieza indispensable para que el proyecto prosperara. Siempre con excelente actitud y positivismo nos aconsejó, motivó y nos guió. Muchas gracias.

Resumen

Las plataformas de blockchain están consolidándose rápidamente en el ámbito tecnológico, con aplicaciones variadas como criptomonedas, cadenas de suministro, entre otros. A medida que las plataformas de blockchain crecen, la interoperabilidad entre ellas se vuelve crucial. Facilitar la comunicación y transferencia de activos entre diferentes redes es clave para permitir la adopción masiva de la tecnología. Existen dos tipos de blockchain de interés: públicas y privadas. Las publicas permiten que cualquier usuario interactúe con ellas sin necesidad de permisos, mientras que las privadas requieren que los usuarios tengan permisos gestionados por un administrador para poder interactuar. No obstante, estas plataformas fueron diseñadas como sistemas cerrados, sin capacidad de interoperabilidad. El crecimiento de blockchain demanda mecanismos de comunicación entre redes públicas y privadas, que permitan el intercambio de datos y operaciones conjuntas sin comprometer la integridad y consistencia de estos. En particular, se requiere la validación de transacciones.

Este proyecto tiene como desafío desarrollar un mecanismo de validación de transacciones *cross-blockchain* entre *Ethereum* y *Hyperledger Fabric*, garantizando la correcta validación de la información transferida. La solución se basa en una solución de interoperabilidad preexistente de tipo *Gateway*, al que se añadió un mecanismo de validación de transacciones *cross-blockchain*. Asimismo, se diseñó e implementó una prueba de concepto para abordar esta problemática. Para validar la propuesta, se utilizaron tres estrategias: implementación de un escenario de uso, pruebas de rendimiento y análisis de costos. El escenario se enfocó en la compra de automóviles en *Hyperledger Fabric*, generando recompensas en tokens de fidelidad o NFTs en *Ethereum*. Las pruebas de rendimiento indicaron que la solución es viable en cuanto a tiempos de procesamiento, identificando un cuello de botella en la blockchain de Hyperledger Fabric. El análisis de costos mostró que la solución es económicamente viable, permitiendo su aplicación en escenarios de menor valor.

En conclusión, se logró diseñar e implementar un prototipo que interopera una blockchain pública con una privada, específicamente *Ethereum* y *Hyperledger Fabric*, garantizando la validez de las transacciones, con integridad de la información, costos accesibles y tiempos de respuesta óptimos.

Palabras clave: Blockchain, Interoperabilidad, Validación de transacciones, Cross-Blockchain, Ethereum, Hyperledger Fabric, Gateway

Índice general

Capítulo 1

Introducción

En este capítulo se ofrece una visión general del documento y una introducción al trabajo realizado. En la sección 1.1, se expone la motivación que impulsó el desarrollo del proyecto. A continuación, en la sección 1.2, se enumeran los objetivos específicos del proyecto. Posteriormente, en la sección 1.3, se detallan los aportes que este proyecto ha realizado en su campo. Finalmente, en la sección 1.4, se describe la estructura del documento, proporcionando una guía clara para su lectura.

1.1. Motivación

Blockchain es una tecnología de registro distribuido que ofrece una estructura segura e inmutable para almacenar y gestionar datos. Introducida por primera vez en 2008 como la base para la criptomoneda Bitcoin (Nakamoto, 2008), la blockchain ha evolucionado significativamente, expandiéndose a una amplia gama de aplicaciones más allá del ámbito financiero. En su núcleo, una blockchain es una cadena de bloques de información, donde cada bloque contiene un conjunto de transacciones verificadas y está enlazado al bloque anterior mediante criptografía, formando una secuencia continua, segura e inmutable.

La relevancia de blockchain en el desarrollo de software radica en su capacidad para garantizar la integridad y la transparencia de los datos en sistemas distribuidos, eliminando la necesidad de intermediarios confiables. Esta característica ha propiciado su adopción en áreas como la gestión de identidades, la trazabilidad de productos, y la automatización de procesos mediante contratos inteligentes (*Smart Contracts*). Estos contratos son programas autoejecutables que se ejecutan en la blockchain cuando se cumplen ciertas condiciones predefinidas, revolucionando la manera en que se gestionan y ejecutan acuerdos digitales.

Existen dos tipos principales de blockchain: públicas y privadas. Las blockchains públicas, como Bitcoin y Ethereum, son accesibles para cualquier persona, permitiendo realizar operaciones como lecturas, transferencias y validaciones de

forma abierta. Por otro lado, las blockchains privadas son gestionadas por una o varias entidades, y requieren permisos específicos para interactuar con ellas, siendo Hyperledger Fabric un ejemplo destacado en este ámbito.

En términos de interoperabilidad, la tecnología blockchain presenta desafíos significativos. Originalmente concebidas como sistemas cerrados, las blockchains[1] no fueron diseñadas para intercambiar información entre sí, lo que hace que el intercambio de datos entre diferentes cadenas sea un problema complejo. Para abordar este desafío, es necesario desarrollar soluciones que faciliten la transferencia de información entre blockchains de manera segura y confiable. Si bien existen soluciones de interoperabilidad que intentan resolver estos desafíos entre diferentes blockchains, lo cierto es que en general estas están enfocadas en el ámbito industrial, focalizando la atención en las criptomonedas y en blockchains públicas como Bitcoin o Ethereum, entre otras, habiendo escasos avances en cuanto a la interoperabilidad entre blockchains públicas y privadas. Un ejemplo de este último caso es la tesis de grado (Mathías Castro, Emiliano González y Sebastian Pandolfi, 2023), donde se desarrolló un Gateway para interconectar Ethereum y Hyperledger Fabric. A pesar de estos avances, aún persisten desafíos en la validación de transacciones *cross-blockchain*. Uno de los principales retos es garantizar la consistencia entre blockchains, asegurando que las transacciones realizadas en una blockchain se reflejen correctamente en la otra, sin comprometer la integridad de los datos. Además, las diferencias en los mecanismos de consenso entre distintas blockchains, como Proof of Work en Ethereum y el consenso basado en permisos en Hyperledger Fabric, complican la sincronización y validación de las transacciones. Otro desafío significativo es el manejo de diferentes estructuras de datos, ya que las blockchains de Hyperledger Fabric y Ethereum emplean distintos formatos de bloques y transacciones. Actualmente, no existe una solución que aborde de manera integral todos estos aspectos necesarios para una interoperabilidad efectiva entre Hyperledger Fabric y Ethereum.

1.2. Objetivos

El objetivo principal de este proyecto es diseñar e implementar una solución de interoperabilidad entre las blockchains de Hyperledger Fabric y Ethereum, que contemple la validación de las transacciones intercambiadas. Por otro lado, los objetivos específicos son:

- Investigar las soluciones existentes de interoperabilidad entre blockchains, con un enfoque particular en la validación de transacciones.

- Analizar y comprender los mecanismos de validación de transacciones en las blockchains de Hyperledger Fabric y Ethereum.

[1] A lo largo del documento se utilizará el término “blockchains” para referirse al plural de blockchain

- Diseñar, implementar y extender la solución de interoperabilidad ya existente entre Hyperledger Fabric y Ethereum, asegurando la validación de la información intercambiada entre ambas blockchains.

- Evaluar la propuesta mediante la implementación de un caso de uso basado en un escenario de compraventa de automóviles, acompañado de un análisis de costos y pruebas de rendimiento.

- Documentar el proyecto y elaborar un informe técnico detallado que recoja todos los aspectos del desarrollo y las conclusiones del mismo.

1.3. Aportes del proyecto

Los aportes realizados en el proyecto son los siguientes:

- **Relevamiento y análisis de mecanismos de validación de transacciones:** Se realizó un estudio de los mecanismos que emplean las blockchains de Hyperledger Fabric y Ethereum para validar las transacciones, proporcionando una base sólida para entender las diferencias y similitudes en sus procesos de validación.

- **Evaluación de soluciones de interoperabilidad existentes:** Se llevó a cabo un análisis detallado de las soluciones actuales para la interoperabilidad entre blockchains, con un enfoque específico en cómo estas abordan la validación de transacciones, identificando fortalezas y debilidades en sus enfoques.

- **Evolución y mejora de la solución de interoperabilidad previa:** A partir del trabajo desarrollado en la tesis de grado anterior (Mathías Castro, Emiliano González y Sebastian Pandolfi, 2023), se avanzó en el diseño e implementación de una solución mejorada que incorpora la validación de transacciones en el proceso de intercambio de información entre Hyperledger Fabric y Ethereum.

- **Validación mediante un caso de uso práctico:** La efectividad de la solución propuesta se evaluó a través de la implementación de un escenario de compraventa de automóviles, complementado con un análisis de costos y pruebas de rendimiento para garantizar la viabilidad y eficiencia de la solución en un entorno real.

- **Documentación técnica :** Se elaboró una documentación detallada del proyecto, que incluye un informe técnico completo, abarcando todos los aspectos del desarrollo, las metodologías empleadas, los resultados obtenidos y las conclusiones derivadas, asegurando así la claridad y la replicabilidad del trabajo realizado.

1.4. Estructura del Documento

El documento se divide en los capítulos descritos a continuación. El capítulo Marco Conceptual 2 desarrolla los conceptos teóricos necesarios para comprender el trabajo realizado. El capítulo Análisis de Requerimientos 3 analiza los requerimientos del proyecto. El capítulo Análisis de Soluciones existentes 4 desarrolla y analiza tres de las soluciones existentes estudiadas. El capítulo Solución propuesta 5 describe la solución que se propone llevar a cabo en base a los requerimientos. El capítulo Diseño de la solución 6 desarrolla el diseño de la solución llevada a cabo. El capítulo Implementación de la solución 7 ahonda en la implementación y desarrollo del prototipo. En el capítulo Evaluación de la propuesta 8 se exponen las evaluaciones llevadas adelante para validar la propuesta. En el capítulo Gestión de proyecto 9 se muestra como fue la gestión del proyecto. Finalmente, en el capítulo Conclusiones y trabajo a futuro 10 se desarrollan las conclusiones y se proponen posibles mejoras en la solución como trabajo a futuro.

Capítulo 2

Marco Conceptual

En este capítulo se explicarán y analizarán los conceptos más relevantes utilizados a lo largo del documento. El capítulo se estructura en cuatro secciones principales. En la sección 2.1, se presentan los fundamentos de la tecnología blockchain, abordando sus características esenciales, las distintas clasificaciones y una descripción detallada de los contratos inteligentes. La sección 2.2 se centra en el concepto de interoperabilidad, proporcionando definiciones, tipos y su aplicación en el contexto de blockchain.

2.1. Blockchain

En la literatura existen diversas definiciones de blockchain:

- "Blockchain es un libro de contabilidad compartido e inmutable que facilita el proceso de registro de transacciones y seguimiento de activos en una red empresarial"(IBM, 2024).

- "(..) blockchain es un mecanismo avanzado de base de datos que permite compartir información de forma transparente dentro de una red empresarial. Una base de datos blockchain almacena datos en bloques que están enlazados en una cadena. Los datos son coherentes cronológicamente porque no se puede borrar ni modificar la cadena sin el consenso de la red. (..) se puede utilizar la tecnología blockchain para crear un libro de contabilidad inalterable o inmutable para el seguimiento de pedidos, pagos, cuentas y otras transacciones (..)". (AWS, 2024)

Por esto, se puede visualizar a blockchain como una base de datos distribuida, y para que exista un cambio de estado en esta base de datos debe existir un consenso previo en la red. De hecho, blockchain es un tipo de DLT (*Distributed Ledger Technology*), lo que significa que es una base de datos en la cual se guarda la información de forma distribuida, donde existen varios participantes en la red. Estos participantes usualmente son llamados *Peers*. En

concreto blockchain organiza esta información en bloques enlazados que utilizan criptografía para asegurar la inmutabilidad de toda la cadena (Ballamudi, 2016). En la figura 2.1 se observa la estructura de la base de datos. Dado que los bloques están ordenados cronológicamente, es posible recorrer todo el historial de eventos en la blockchain revisando cada transacción en cada bloque. Además de las transacciones, cada bloque contiene en su encabezado un hash[1] de todas las transacciones y un hash del encabezado del bloque anterior. Estos dos elementos aseguran que, en caso de que un nodo intente manipular un bloque, no podría convencer al resto de la red de que tiene el bloque "correcto", ya que los hash no coincidirían.

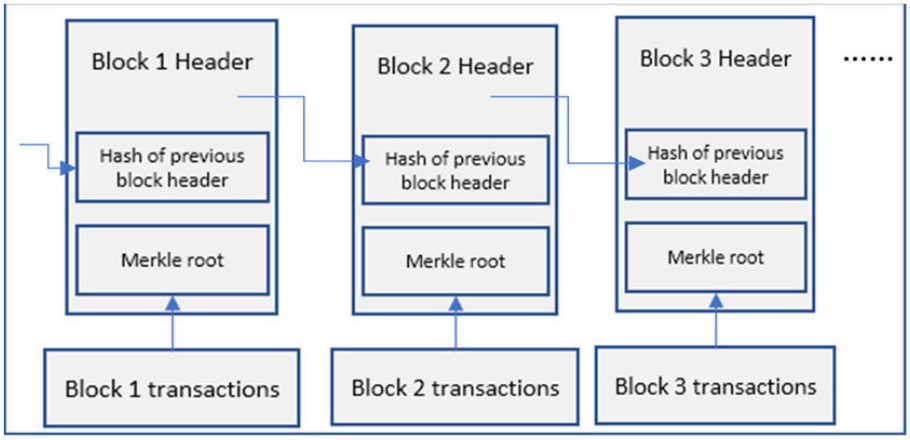


Figura 2.1: Ejemplo de bloques encadenados haciendo uso de criptografía. (Barghuthi y cols., 2019)

2.1.1. Clasificación basada en accesibilidad

Las blockchains se pueden clasificar en dos categorías generales: "Permissioned" y "Permissionless". Estos términos se refieren a la accesibilidad y a los requisitos de identificación y/o permisos necesarios para operar en la blockchain.

Las blockchains **Permissionless** son aquellas a las que cualquier persona puede leer, escribir, operar y participar del protocolo de consenso, realizando transacciones o incluso creando nuevos bloques. Este tipo de blockchain es comúnmente utilizado en criptomonedas, donde no se requiere autorización previa para participar en la red.

Por otro lado, las blockchains **Permissioned** están generalmente controladas por una o más entidades. En estas redes, se necesita obtener permisos de los administradores para interactuar, lo que puede incluir la capacidad de realizar

[1] El termino hash hace referencia al resultado de aplicar una función matemática llamada "función hash"a un conjunto de elementos de entrada

transacciones, acceder a información o participar en la validación de bloques (Parrondo, 2018).

Asimismo, otra posible clasificación es: pública (2.1.1), consorcio (2.1.1) y privada (2.1.1). Estas tres categorías se consideran subcategorías de las categorías permissioned y permissionless. Las blockchain públicas son en general permissionless blockchains, mientras que las de tipo consorcio y privada se consideran usualmente permissioned.

Pública

Este tipo de blockchain permite un acceso total a la red. Permite interactuar con ella, y también participar de esta. Es decir, además de generar transacciones, es posible participar del mecanismo de consenso o del proceso de validación. Este tipo de blockchain en general se consideran totalmente descentralizadas. Ejemplos de estas blockains son Etheureum y Bitcoin (Parrondo, 2018).

Consorcio

A diferencia del anterior, en esta blockchain no cualquier usuario puede participar de ella. Por el contrario, suelen ser un conjunto de entidades u organizaciones las cuales administran esta red. Esto significa que los participantes de la red en el mecanismo de consenso queda limitado concretamente a estas entidades y por ende los permisos de escritura limitados a estos también. En cuanto a los permisos de lectura, dependen de cada caso. Este tipo de blockchain en general se consideran parcialmente descentralizadas.(Parrondo, 2018) Ejemplo de este tipo de blockchain es Hyperledger Fabric.

Privada

En este caso y a diferencia del anterior, los permisos de lecturas no corresponden a un conjunto de entidades, sino que más bien a una única entidad u organización. Respecto a los permisos de lecturas, estos quedan determinados dependiendo del caso. (Parrondo, 2018)

2.1.2. Contratos Inteligentes

Los contratos inteligentes, conocidos comúnmente como *Smart Contracts*, son conjuntos de instrucciones codificadas que se almacenan en una blockchain. Estos contratos definen reglas y consecuencias lógicas para una serie de acciones específicas. Al igual que los contratos tradicionales, un contrato inteligente establece los términos de un acuerdo entre las partes involucradas. Sin embargo, a diferencia de los contratos convencionales que dependen de un sistema legal o intermediario para su ejecución, los contratos inteligentes se autoejecutan según las condiciones establecidas en su código, asegurando así el cumplimiento automático de las reglas acordadas. (Khan, Loukil, Ghedira-Guegan, Benkhelifa,

y Bani-Hani, 2021) En Ethereum, la ejecución de los *Smart Contracts* conlleva un costo asociado denominado *gas*. Cada instrucción dentro de un contrato inteligente requiere una cantidad específica de *gas*, que refleja los recursos computacionales necesarios para su ejecución. Estos contratos se programan principalmente en el lenguaje Solidity y se despliegan en la blockchain, donde se ejecutan de manera descentralizada. En Fabric[2], los contratos inteligentes, denominados *chain-code*[3], pueden escribirse en diversos lenguajes de programación como Typescript, Go y Java. Una *DApp* (Aplicación Descentralizada) es una aplicación que opera sobre una blockchain y utiliza contratos inteligentes para ejecutar su lógica de negocio. Estas aplicaciones son completamente descentralizadas y están compuestas por un front-end que interactúa con la blockchain a través de contratos inteligentes desplegados en ella.

2.1.3. Tokenización

Tokenización es el proceso de convertir un activo o derecho en una representación digital (token) sobre una blockchain. Por lo tanto, un token puede representar cualquier elemento virtualmente.

Tokens fungibles

Los Tokens fungibles es un activo o representación digital, donde la característica principal de estos es que cada token comparte características y valor con otro token. Un ejemplo de token fungible es Ether *ETH*, donde un token Ether es exactamente igual a otro token *ETH*. (*Token ERC20 — Ethereum documentation*, 2024)

Tokens no fungibles (NFT)

Los tokens no fungibles son representaciones digitales únicos individualmente y probablemente no abunden. A diferencia de los anteriores, estos tokens no comparten características ni valor. Esto permite la tokenización de artículos coleccionables, de arte, entre otros. (*NFT — Ethereum documentation*, 2024)

2.1.4. Características

En esta sección se describen las principales características y ventajas de las blockchains. Es importante aclarar que en general la clasificación de blockchains permissioned o permissionless comparten las mismas características; sin embargo, estas lo hacen en mayor o menor medida dependiendo de la característica que se trate. (Tasca y Tessone, 2017)

[2]Se utilizará indistintamente los términos Fabric y Hyperledger Fabric

[3]En el resto del documento, y por simplicidad, se utilizarán los términos Smart Contract y chain-code indistintamente

Descentralización de consenso

El consenso descentralizado en blockchain permite que participantes sin confianza mutua lleguen a un acuerdo. Se basa en reglas (como las transacciones permitidas y la dificultad de minado) y en el historial de transacciones (que permite a los usuarios determinar a quién pertenece cuál activo), permitiendo determinar la propiedad de cada activo. Los nodos locales verifican y agregan transacciones a la blockchain usando la "cadena más larga" como referencia. No se necesita una autoridad central, eliminando un único punto de confianza o fallo.

Transparencia

En blockchain, los registros son auditables por un grupo predefinido de participantes, que puede ser más o menos abierto. En blockchains públicas, cualquier usuario tiene igual acceso a la base de datos, lo que garantiza transparencia y rastreabilidad. Los participantes pueden usar sus derechos individuales (ponderados, por ejemplo, por potencia de cálculo) para actualizar la base de datos y tienen la opción de agrupar sus derechos ponderados.

Seguridad

Las blockchains son bases de datos compartidas, inmutables y replicadas, donde los registros son irreversibles y no pueden ser falsificados gracias a funciones hash criptográficas unidireccionales. La seguridad se logra porque los usuarios solo pueden transferir datos si poseen una clave privada, que genera una firma para cada transacción. Esta firma confirma que la transacción es legítima y evita su alteración después de emitida.

Inmutabilidad

Las blockchains operan bajo el principio de no repudio e irreversibilidad de los registros. Son inmutables porque una vez que los datos se registran en la base de datos, no pueden alterarse sin que la red lo sepa, gracias a su resistencia a manipulaciones. Esto se logra mediante el uso de funciones hash, que convierten cualquier entrada de datos en una "huella digital" de tamaño fijo. Si los datos de entrada cambian, el hash cambia de manera impredecible. Cada bloque incluye el hash del bloque anterior, formando una cadena. La fortaleza de la inmutabilidad depende de la dificultad para cambiar el historial de transacciones. Por ejemplo, en blockchains públicas de prueba de trabajo (PoW)[4] como Bitcoin, la inmutabilidad está relacionada con el costo de un ataque del 51 %. En blockchains privadas, el mecanismo de adición de bloques es diferente, requiriendo la firma de un conjunto definido de participantes. Aunque las blockchains son técnicamente inmutables, desde una perspectiva de gobernanza, esta inmutabilidad no siempre se cumple plenamente.

[4] Por más detalles sobre los mecanismos de consenso en blockchain, consulte el anexo B.2.2.

2.2. Interoperabilidad

En la sección 2.2.1 se ofrece una definición e introducción a interoperabilidad en blockchain. Posteriormente, en la sección 2.2.2 se describen los distintos tipos de interoperabilidad en blockchain. En la sección 2.2.3 se detallan las operaciones permitidas en blockchain. Finalmente, en la sección 2.2.4 se abordan algunas de las soluciones existentes para lograr la interoperabilidad.

2.2.1. Definición

La interoperabilidad en la ingeniería de software es definida por Peter Wegner como "la habilidad de dos o más componentes de software de cooperar a pesar de las diferencias de lenguajes, interfaces, o plataforma de ejecución". (Wegner, 1996).

Blockchain tiene sus particularidades, por ejemplo, es un sistema distribuido, con diferentes estados, y mecanismos de consenso. Por otro lado (Abebe y cols., 2019), define interoperabilidad en blockchain como: " (..) la dependencia semántica entre diferentes blockchains con el fin de transferir o intercambiar datos o valor, con garantías de validez o verificabilidad".

La segunda definición tiene un enfoque que abarca y considera los desafíos que presentan las blockchains a la hora de interoperar. Estos desafíos en gran parte son debido a que estos sistemas distribuidos son cerrados, ya que no aceptan fácilmente información desde otros sistemas. Asimismo, cada blockchain presenta componentes y propiedades únicos, como el consenso, su estado o criptoactivos, todo esto lo convierte un gran desafío la interoperabilidad entre blockchains. (Pillai, Biswas, Hóu, y Muthukkumarasamy, 2022)

Los protocolos de interoperabilidad en blockchain se pueden clasificar en dos categorías: *cross-chain* y *cross-blockchain*. La principal diferencia entre estos tipos radica en la homogeneidad o heterogeneidad de los sistemas que se desean interconectar. En concreto, los protocolos *cross-chain* tienen como objetivo la interoperabilidad entre dos o más blockchains homogéneas, es decir, blockchains que comparten el mismo mecanismo de consenso, arquitecturas equivalentes, entre otros aspectos. Un ejemplo de este tipo de interoperabilidad es la relación entre Ethereum y Polygon, ya que ambas blockchains son compatibles con la Máquina Virtual de Ethereum (EVM). Por otro lado, los protocolos *cross-blockchain* buscan la interoperabilidad entre blockchains heterogéneas, lo que implica que estas blockchains pueden tener diferentes mecanismos de consenso, ambientes de ejecución, tamaños de bloque, entre otras diferencias. Un ejemplo representativo de esto es la interoperabilidad entre Ethereum y Bitcoin. Debido a estas diferencias, los protocolos diseñados para interconectar blockchains heterogéneas suelen ser más complejos en comparación con los protocolos *cross-chain*.(Mohanty, Anand, Aljahdali, y Villar, 2022)

2.2.2. Tipos de interoperabilidad

La interoperabilidad en blockchain puede ser clasificada dependiendo del tipo de blockchains a interoperar y los tipos de sistemas involucrados en dicha interoperabilidad. En el primer caso, dependerá si las blockchains son públicas, privadas, o de consorcio. En el segundo caso, dependerá de los sistemas que estén involucrados. Como se observa en la figura 2.2, se tienen 3 posibilidades. El caso (A), que una aplicación de negocio (como una DApp) requiera interoperar con una o más plataformas de blockchain. El caso (B), donde una plataforma de blockchain requiera operar con una o más plataformas de blockchain. Por último, el caso (C), donde una plataforma de blockchain requiera operar con una o más aplicaciones de negocios (como una o más DApp). (Llambias, González, y Ruggia, 2022)

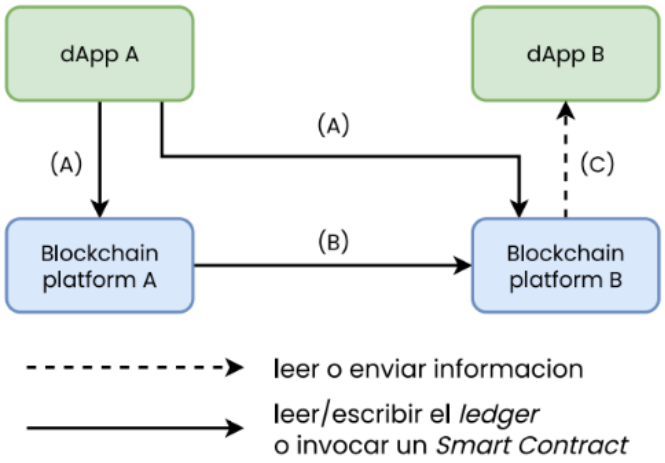


Figura 2.2: Los tres tipos de interoperabilidad entre plataformas blockchain y aplicaciones de negocios DApp. En el presente informe se propone proponer una solución de interoperabilidad del tipo (B)(Ameigenda y cols., 2023)

2.2.3. Operaciones entre blockchain

Cuando se opera con dos plataformas blockchain, se pueden definir tres operaciones principales: *Consulta*, *Comando* y *Coordinación*. Para describir estos términos, asumiremos la existencia de dos blockchains, designando una como la blockchain de origen y la otra como la blockchain de destino.(Llambias y cols., 2022)

Consulta

Esta operación se trata de una *consulta* o también llamada *query* (Operación (a) en la figura 2.3), donde la blockchain origen realiza una petición de solici-

tud de información, mientras que la respuesta incluye la información solicitada junto a una prueba que válida dicha información y que ha sido verificada por el mecanismo de consenso en dicha Blockchain. Finalmente, la información queda guardada en ambas blockchains.

Comando

Se trata de una operación unidireccional de tipo *comando* o también llamada *Command* (Operación (b) en la figura 2.3). En este tipo de operaciones, la blockchain origen envía información junto a una prueba, y basándose en ello, la blockchain destino puede guardar o realizar una transacción. Igual que el caso anterior, la prueba permite validar la información y que verificó el mecanismo de consenso de la blockchain origen. Un ejemplo específico de este tipo de operación es el de "Transfer" o "Transferencia". Este concepto es utilizado en general cuando la información enviada de una blockchain a otra son activos digitales.

Coordinación

Se trata de una operación de tipo *coordinación* o también llamada *Coordinated* (Operación (c) en la figura 2.3). En esta operación es necesario una coordinación entre ambas blockchain para realizar transacciones que sean atómicas y consistentes (ACID)[5]. Para dicha coordinación se requiere que la blockchain origen realice una acción A y envíe un mensaje a la blockchain B. Luego la blockchain B necesita realizar una acción B en consecuencia luego de recibir el mensaje anterior. Ambas acciones deben ejecutar atómicamente. Un caso específico de este tipo de operación es el de "Exchange" o "Intercambio". Este concepto es utilizado en general cuando el intercambio es de assets[6] involucrando acciones en ambas blockchain.

[5] ACID es un conjunto de propiedades (Atomicidad, Consistencia, Aislamiento y Durabilidad) que garantizan la fiabilidad de las transacciones en sistemas de bases de datos.

[6] Un *asset* es un bien o recurso digital que puede representar cualquier valor o propiedad, como criptomonedas, contratos, o cualquier objeto con valor.

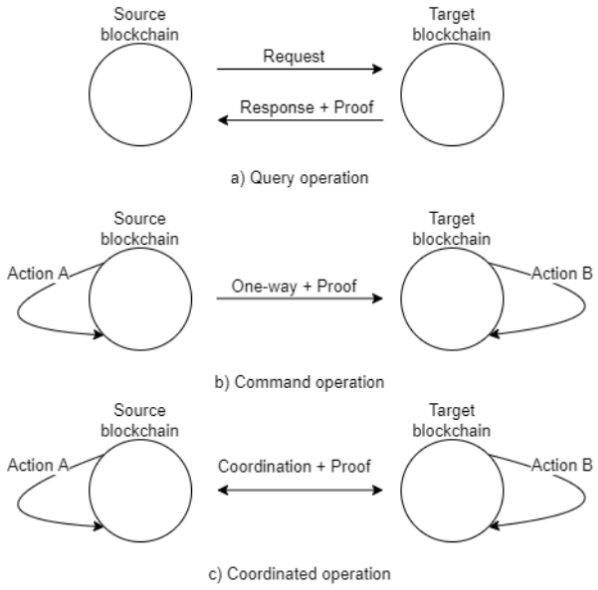


Figura 2.3: Tipos de operaciones entre dos blockchain. (Llambias y cols., 2022)

2.2.4. Soluciones para interoperar

En esta sección se presentan y describen las posibles soluciones de interoperabilidad de blockchain indicando a cuál categoría corresponde. Por tanto, en las secciones conectores públicos, blockchain de blockchain, conectores híbridos se describen las posibles soluciones para las categorías correspondientes.
(Belchior, Vasconcelos, Guerreiro, y Correia, 2021)

Conectores públicos

Usualmente utilizado para interoperar blockchains públicas y homogéneas, sobre todo orientado a criptomonedas. Dentro de este tipo existen tres posibles soluciones: *sidechain*, *notary scheme* y *hash-locks*.

Una *sidechain* es una cadena de bloques independiente que está conectada a una cadena de bloques principal (a menudo denominada *mainchain*) a través de una conexión bidireccional. Este diseño permite la transferencia de activos y datos entre la *mainchain* y la *sidechain*, al tiempo que permite a la *sidechain* operar de forma independiente con sus propias reglas y mecanismos de consenso. Las *sidechains* se utilizan comúnmente para mejorar la escalabilidad y la eficiencia[7]. Usualmente, este tipo de soluciones "escuchan eventos" de cier-

[7]Aquellas sidechain que tienen por objetivo mejorar la escalabilidad y eficiencia de la mainchain son llamadas tambien "soluciones de capa 2" (L2)

tas blockchains y actúan en consecuencia ejecutando transacciones sobre otras blockchains.

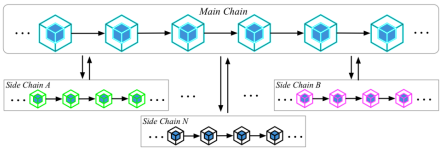


Figura 2.4: Ejemplo abstracto de comunicación entre sidechain y main-chain

Por otro lado, el *notary scheme* es un mecanismo que facilita las transacciones y la comunicación entre múltiples blockchains mediante un intermediario de confianza conocido como notario. Este enfoque mejora la interoperabilidad entre diferentes redes de blockchain. El *notary scheme* actúa como un tercero de confianza que supervisa y verifica las transacciones, garantizando su exactitud y coherencia. Ejemplos conocidos de este tipo de soluciones son Binance o Coinbase (soluciones centralizadas), Uniswap (solución descentralizada).

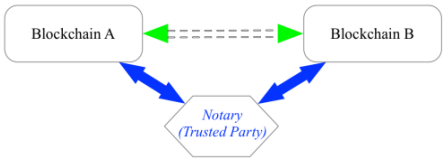


Figura 2.5: Ejemplo abstracto de un notary scheme centralizado como relay

Finalmente, los *hash-locks* son una técnica criptográfica utilizada en la tecnología blockchain para facilitar transacciones seguras sin necesidad de un intermediario de confianza. Esta técnica es especialmente útil para permitir intercambios atómicos (*atomic swaps*) y transacciones entre cadenas.(Wang, 2021) A grandes rasgos, la técnica de *hash-locking* utiliza un sistema de bloqueo por tiempo basado en un hash, aplicando un bloqueo temporal para asegurar la transacción. Solo cuando ambas partes involucradas están de acuerdo con las obligaciones, la transacción se mantendría en un estado bloqueado, lo cual es similar al concepto de una transacción atómica. (Wang, 2021)

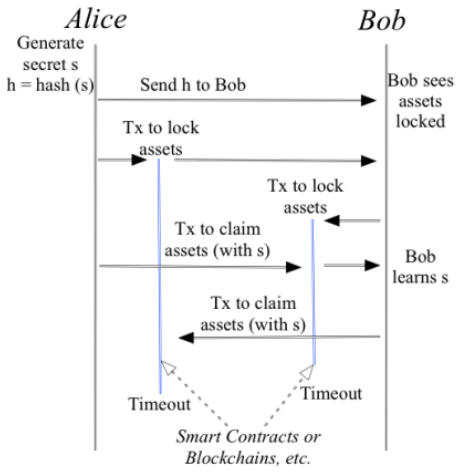


Figura 2.6: Ejemplo abstracto de comunicación hash-locking entre dos clientes usando diferentes blockchains

Blockchain de Blockchain

La interoperabilidad blockchain de blockchain es un *framework* que permite la reusabilidad de información, red, consenso, estructuras de incentivos, y diferentes capas de Smart Contract para facilitar la creación y gestión aplicaciones descentralizadas (dApps) especificas a cada blockchain pero permitiendo interoperabilidad entre ellas. Ejemplos de estos son Palkadot y Cosmos. Estas dos plataformas para interoperar proponen dos protocolos de comunicación (XCMP y IBC respectivamente) para el intercambio de mensajes entre las diferentes blockchains. Este tipo de soluciones en general son implementadas de forma similar a las soluciones de tipo sidechain, donde típicamente existe una blockchain principal que conecta el resto de las blockchains secundarias que se desea interoperar.

Los *bridges* o también llamado "puentes" pueden requerir distintos grados de confianza para operar, dependiendo de si son centralizados o descentralizados. Los puentes centralizados dependen de una única entidad o de un pequeño grupo de entidades para gestionar el puente y validar las transacciones, lo que simplifica el modelo, pero introduce riesgos de centralización. En contraste, los puentes descentralizados utilizan una red de validadores o notarios para garantizar la integridad de las transacciones, sin depender de un único punto de control, mejorando así la seguridad y la descentralización. (Wang, 2021)

Conectores híbridos

Esta categoría compone a todas las soluciones que no son conectores públicos ni blockchain de blockchain. En general, esta categoría surge de la necesidad de

interoperar no solamente blockchains públicas, sino que también privadas. Algunas subcategorías de tipos de soluciones pertenecientes a esta serían: *Trusted Relays*, *Protocolos Agnósticos de Blockchain* y *Migrador Blockchain*.

Los *trusted relays* o relays de confianza son mecanismos diseñados para facilitar la interoperabilidad entre diferentes redes de blockchain, actuando como un bridge que permite la transferencia de información y activos a través de estas redes. Este tipo de soluciones redireccionan las transacciones desde una blockchain origen a una destino, permitiendo a los usuarios implementar su propia lógica de negocio. Este tipo de solución, sin embargo, pueden ser descentralizados. Esto podría ser teniendo entidades llamadas "validadores". Estas entidades estarían a cargo de firmar las transacciones cross-blockchain, que posteriormente únicamente llegando a un cierto cuórum, consideran a la transacción como válida. Un ejemplo de trusted relay puede ser BTC relay. En pocas palabras, este relayer permite que Smart Contracts en Ethereum puedan validar transacciones en Bitcoin sin la necesidad de intermediarios.

Por otro lado, los *Protocolos Agnósticos de Blockchain* permiten la interoperabilidad brindando una capa de abstracción. Este tipo de soluciones permite la interoperabilidad de blockchain a blockchain. Es decir que blockchain a blockchain es una subcategoría de protocolo agnóstico blockchain; sin embargo, existen otras soluciones. Por ejemplo, otra solución es llamada "Interledger Protocol". Esta es un protocolo para enviar de envío de información de pagos. Inspirado en TCP/IP, se envían y redireccionan paquetes de información que contienen valor monetario. Este tipo de soluciones en general no tiene compatibilidad hacia atrás y por ende es necesario cambiar el código fuente de las blockchains para permitir su uso. Finalmente, las soluciones de tipo *Migrador Blockchain* permite a los usuarios finales migrar el estado de una blockchain a otra. Este tipo de solución nace a partir de una necesidad empresarial, en caso de suceso de un desastre o problemas de performance. Algunas de las soluciones que se han estudiado de este tipo, solo proveen portabilidad a un número reducido de blockchains públicas.(Wang, 2021)

Capítulo 3

Análisis de requerimientos

3.1. Introducción

Este capítulo aborda los requisitos funcionales y no funcionales del sistema, proporcionando una descripción detallada de las capacidades y cualidades esperadas. En la sección 3.2, se describen los requerimientos del sistema, mientras que en la sección 3.3, se describen los requerimientos no funcionales. La sección 3.4 detalla el alcance del proyecto y las funcionalidades a implementar en un escenario descrito en la sección 3.5. Por último, en la sección 3.6, se enumeran los casos de uso del escenario.

3.2. Requerimientos Funcionales

En la Tabla 3.1 se presentan los requerimientos relevados en colaboración con el tutor, los cuales se implementarán como parte de la solución.

Identificador	**Descripción**
REQ 1	Actualizar el sistema BIG para validar transacciones en la blockchain de Fabric.
REQ 2.1	Ejecución de una transacción validada cross-blockchain de tipo "Comando" con origen Fabric, y destino Ethereum.
REQ 2.2	Ejecución de una transacción validada cross-blockchain de tipo "Comando" con origen Fabric, generando un NFT en Ethereum.
REQ 3	Habilitar la visualización y ejecución de una transacción validada cross-blockchain de tipo "transferencia" con origen Ethereum, y destino Fabric.
REQ 4	Implementar el escenario propuesto descrito en la sección 3.5.

Tabla 3.1: Requerimientos iniciales del proyecto

Los requerimientos REQ 2.1 y REQ 2.2 que detallan cómo se validarán las

transacciones, se analizarán en profundidad en la sección 5.2.

3.3. Requerimientos No Funcionales

- **Integración con el sistema BIG**: El sistema debe integrarse de manera adecuada y confiable con el sistema BIG, sin necesidad de realizar modificaciones significativas. La integración debe garantizar la comunicación fluida y segura entre ambos sistemas.
- **Interoperabilidad entre Fabric y Ethereum**: El sistema debe ser capaz de enviar y recibir mensajes entre la blockchain de Hyperledger Fabric y la de Ethereum. Esta interoperabilidad es crucial para la funcionalidad del sistema y debe ser implementada de forma eficiente y segura.
- **Extensibilidad**: El sistema debe ser diseñado de tal manera que permita, con mínima dificultad, la integración de otras blockchains en el futuro. La arquitectura del sistema debe ser modular y flexible para acomodar nuevas tecnologías y protocolos blockchain.

3.4. Alcance del Proyecto

Debido a la inexperiencia del equipo, la investigación sobre soluciones existentes y la reutilización de una solución anterior consumió una cantidad significativa de tiempo y recursos. En comunicación con el tutor del proyecto, se decidió que el requerimiento REQ 3 no se podría cumplir en su totalidad. Por lo tanto, el alcance del proyecto se limita a la validación de las transacciones en Ethereum de las transacciones ejecutadas y validadas previamente en Hyperledger Fabric.

3.5. Escenario

La concesionaria “CarToken” utiliza la blockchain privada de Hyperledger Fabric para gestionar la trazabilidad de sus automóviles vendidos. En esta blockchain privada, se registra toda la información relevante sobre los vehículos. Debido a la naturaleza privada de la blockchain de Fabric, solo los usuarios autorizados, como los empleados y socios de la concesionaria, tienen la capacidad de acceder y realizar transacciones en esta red.

Con el objetivo de fomentar la fidelidad de sus clientes, la concesionaria de automóviles “CarToken” ha implementado un sistema de recompensas basado en tecnología blockchain. Cada vez que un cliente adquiere un automóvil en la concesionaria, recibe tokens de fidelidad en la blockchain pública de Ethereum, que pueden ser utilizados en la tienda de “CarToken”. Además, los compradores de automóviles de alta gama tienen la oportunidad de recibir NFT (Non-Fungible Tokens). La presentación de estos NFT permite el acceso a preventas exclusivas de la concesionaria y a eventos reservados para clientes especiales.

El proceso de recompensas y trazabilidad incluye las siguientes actividades:

- **Compra de Automóviles:** Cuando un cliente compra un automóvil en la concesionaria, la transacción se registra en la blockchain de Fabric. Esta transacción incluye detalles como el identificador del vehículo, el comprador y la fecha de la compra.
- **Emisión de Tokens de Fidelidad:** El sistema debe emitir tokens fidelidad en la blockchain de Ethereum. Estos tokens se envían a la dirección de Ethereum del comprador, proporcionando así una recompensa digital por su compra.
- **Emisión de NFTs:** Además de los tokens de fidelidad, el sistema tiene la capacidad de generar y emitir un NFT que representa al automóvil comprado.
- **Uso de Tokens y NFTs:** Los tokens de fidelidad y NFTs recibidos en Ethereum podrían ser utilizados por los compradores para adquirir productos de la marca "CarToken" o acceder a beneficios de la marca.

Este enfoque híbrido combina las ventajas de una blockchain privada (control, privacidad y eficiencia) con los beneficios de una blockchain pública (transparencia, inmutabilidad y acceso a una economía digital más amplia). La implementación de este sistema no solo mejora la fidelidad de los clientes, sino que también proporciona una plataforma segura y transparente para la gestión de los activos digitales y la trazabilidad de los automóviles.

En resumen, el escenario propuesto demuestra cómo la interoperabilidad entre diferentes plataformas blockchain puede crear un ecosistema de valor añadido para los clientes, mejorando la transparencia y la confianza en el proceso de compra y propiedad de automóviles. Este sistema también muestra el potencial de los tokens y NFTs en la creación de nuevas formas de recompensas y propiedad digital, alineándose con las tendencias emergentes en tecnología blockchain y economía digital.

3.6. Casos de Uso

En esta sección se detallarán los casos de uso del sistema, incluyendo los actores y el flujo de eventos correspondiente.

3.6.1. Detalles de los Casos de Uso

Nombre	Ver Automóviles a la Venta
Actores	Usuario
Actividades	Visualizar lista de automóviles
Descripción	El usuario accede a la plataforma y puede ver una lista de automóviles disponibles para la venta.
Pre-condiciones	No tiene.
Post-condiciones	El usuario visualiza los detalles de los automóviles disponibles.

Tabla 3.2: Caso de Uso: Ver Automóviles a la Venta

Nombre	Comprar Automóvil Token
Actores	Usuario, Red Fabric, Red Ethereum
Actividades	Completar la compra de un automóvil
Descripción	El usuario selecciona un automóvil para comprar. La plataforma verifica que el usuario tenga una billetera digital en Ethereum, Registra la compra del automóvil en Fabric y procede con la transferencia de un CarToken.
Pre-condiciones	El usuario debe tener una billetera en Ethereum.
Post-condiciones	Se registra la compra del automóvil del usuario y se transfiere según el automóvil una cantidad de *CarTokens* a su billetera digital de Ethereum.

Tabla 3.3: Caso de Uso: Comprar Automóvil generando Tokens en Ethereum

Nombre	Comprar Automóvil NFT
Actores	Usuario, Red Fabric, Red Ethereum
Actividades	Comprar un NFT del automóvil
Descripción	El usuario selecciona un automóvil para comprar. La plataforma verifica que el usuario tenga una billetera digital en Ethereum asociada, Registra la compra del automóvil en Fabric y procede con la transferencia de un NFT por ese automóvil.
Pre-condiciones	El usuario debe tener una billetera en Ethereum.
Post-condiciones	Se registra la compra del automóvil del usuario y se transfiere el automóvil al usuario y un NFT del mismo a su billetera digital de Ethereum.

Tabla 3.4: Caso de Uso: Comprar Automóvil generando NFT en Ethereum

Capítulo 4

Análisis de soluciones existentes

En este capítulo se exploran algunas tecnologías de interoperabilidad blockchain, proporcionando una comprensión amplia sobre cómo los bridges abordan la interoperabilidad y la validación de transacciones en distintas blockchains. Este análisis permite obtener ideas valiosas para el diseño de una solución propia. A continuación, se presentan las tecnologías de Token Bridge 4.1, YUI 4.2 y Optimism 4.3. Asimismo, en la sección 4.4 se incluyen conclusiones generales acerca del presente capítulo.

4.1. Token bridge

4.1.1. Introducción

El proyecto *Token Bridge*(TokenBridge, 2024) permite a los usuarios transferir datos entre dos cadenas en el ecosistema de Ethereum (EVM-compatible). Los puentes entre cadenas proporcionan conexiones rápidas y seguras entre blockchains, creando escalabilidad y conexión (interoperabilidad) entre las redes de Ethereum.

El funcionamiento básico del *Token Bridge* implica la "conversión" de activos digitales de una cadena a otra. Por ejemplo, si un usuario posee tokens en una red EVM-compatible y desea utilizarlos en otra red EVM-compatible, *Token Bridge* permite transferir esos tokens a través del puente, haciendo que estén disponibles en la otra cadena.

El proceso generalmente involucra depositar los tokens en un contrato en la cadena de origen, que luego de bloqueados dichos tokens, emite los equivalentes en la cadena de destino. Una vez que los tokens están disponibles en la cadena de destino, los usuarios pueden utilizarlos según las reglas y funcionalidades específicas de dicha cadena.

4.1.2. Método de validación

La figura 4.1 ilustra en detalle el proceso de validación de transacciones. Este proceso es llevado a cabo por validadores, entidades externas (*off-chain*) que supervisan los eventos generados por los contratos inteligentes de interoperabilidad de la solución. Su principal función es verificar la exactitud de lo que ocurre en la blockchain. Una vez que los validadores confirman la validez de una transacción, la firman digitalmente. Posteriormente, se espera que se alcance un consenso entre los validadores, determinado por el administrador del puente, mediante un umbral, como la mitad más uno de los validadores. Al alcanzarse este consenso, se procede con la transacción desde la blockchain A hacia la blockchain B.

Otro proceso fundamental es el *Comisión Manager*, que se encarga de calcular las comisiones (fees) y distribuirlas entre los validadores.

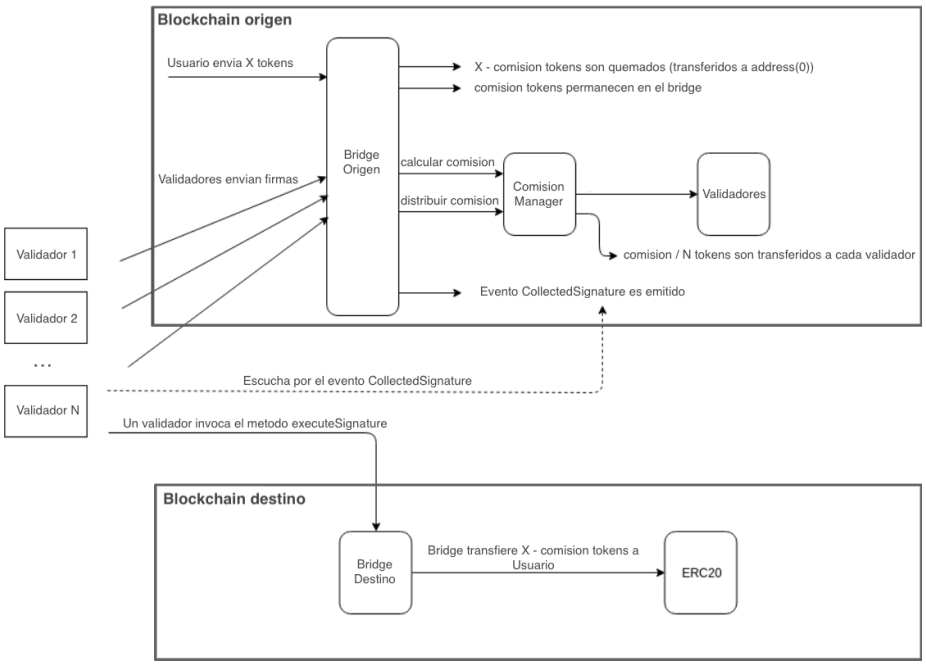


Figura 4.1: Validaciones en Token Bridge (TokenBridge, 2024)

Sin embargo, esta solución implica una cierta dependencia y confianza en los validadores. En caso de que algunos de ellos actúen de manera corrupta, podría conducir a un mal funcionamiento del puente.

4.1.3. Análisis de la solución

Es una alternativa que fue descartada *Token Bridge*, ya que su compatibilidad se limita a blockchains compatibles con EVM. Dado que Hyperledger Fabric no es compatible con EVM, esto lo convierte en una solución inviable para el trabajo planteado. Sin embargo, la utilización de validadores off-chain sobre las transacciones es un aspecto que consideraremos en la solución propuesta.

4.2. YUI

4.2.1. Introducción

El proyecto YUI(*Hyperledger Fabric documentation — YUI*, 2024) de Hyperledger Labs proporciona un marco modular para la interoperabilidad entre diferentes redes de blockchain. Desarrollado para facilitar la comunicación y la transferencia de datos y activos entre diversas blockchains, YUI tiene como objetivo superar las barreras de compatibilidad existentes en el ecosistema blockchain.

El principal objetivo de YUI es permitir que las aplicaciones descentralizadas (dApps) interactúen de manera fluida y eficiente con múltiples redes blockchain. Esto se logra mediante la implementación de estándares y protocolos de interoperabilidad que son compatibles con una amplia gama de tecnologías de blockchain, incluyendo Hyperledger Fabric, Ethereum, Corda, entre otras.

YUI ofrece un conjunto de herramientas y componentes que simplifican el desarrollo de soluciones interoperables. Estos incluyen adaptadores para diferentes blockchains, contratos inteligentes y módulos de comunicación que permiten la transferencia de datos y activos de manera segura y eficiente. Una de sus principales virtudes es que se basa en el estándar de comunicación entre blockchains IBC (*IBC — Inter-blockchain communication protocol*, 2024), al cual extiende para poder soportar proyectos de Hyperledger.

4.2.2. Validación de Transacciones

IBC está compuesto por cinco componentes esenciales: *light clients*, *relayers*, *connections*, *channels* y la *application layer*, los cuales se pueden ver como interactuan en la figura 4.2.

- **Light clients:** Son nodos livianos que almacenan los encabezados de los bloques de la blockchain origen, además se encargan de mantenerse actualizados con su blockchain. Asimismo, tienen la función de validar que cada transacción recibida haya sido efectivamente realizada en la blockchain de origen.

- **Relayers:** Componentes *off-chain* que pueden ser ejecutados por cualquiera (*permissionless*). Su función es retransmitir mensajes entre blockchains.

- **Connections:** Son responsables de conectar los *light clients* a las diferentes blockchains y son específicos para cada cadena.

- **Channels:** Conductos de comunicación que transfieren paquetes entre los módulos de aplicación.

- **Application layer:** Utiliza la capa de transporte de IBC (IBC/TAO) para enviar y recibir mensajes, proporcionando la infraestructura necesaria para establecer conexiones seguras y autenticar paquetes de datos entre blockchains. Aquí, los desarrolladores deben interpretar los datos intercambiados entre diferentes blockchains.

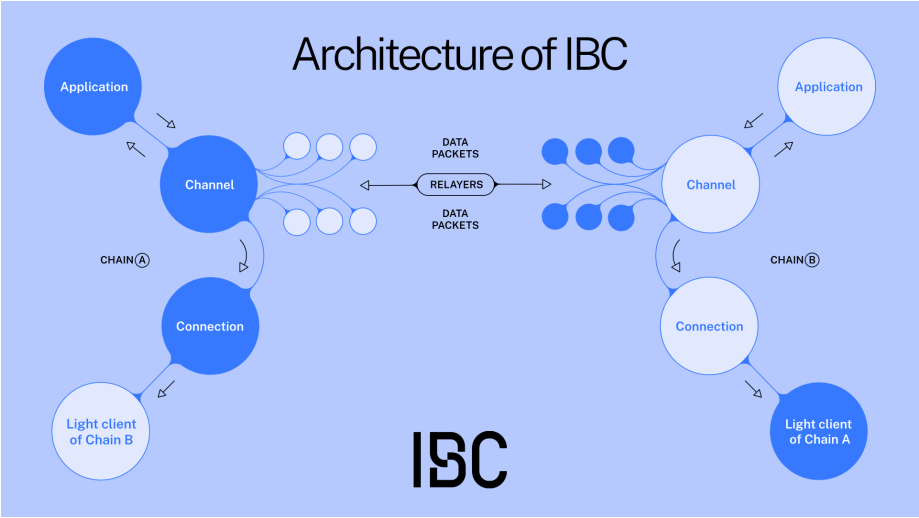


Figura 4.2: Validaciones en IBC(*IBC — Inter-blockchain communication protocol*, 2024)

Los *relayers*, al ser *permissionless* y *off-chain*, permiten la existencia de múltiples instancias sin importar su cantidad. Cada blockchain posee un *light client* que valida transacciones en la blockchain destino. Esto posibilita, por ejemplo, verificar en la blockchain destino que un evento X ocurrió y fue validado en la blockchain origen antes de realizar una acción, como una transferencia de activos. Los *light clients* son livianos en comparación con el resto de clientes que participantes de la red como los Peers, ya que almacenan una menor cantidad de encabezados de bloques para optimizar el consumo de ancho de banda, tiempo y memoria.

4.2.3. Análisis de la solución

YUI se presentó como una herramienta muy adecuada para el proyecto debido a tres aspectos:

1. Su compatibilidad con blockchains de Hyperledger y Ethereum
2. Su utilización de un estándar de interoperabilidad
3. Su utilización de un estándar de interoperabilidad y su capacidad para cumplir con el requisito de la solución fuera *trustless*

Sin embargo, a pesar de que YUI contaba integración con Fabric y Ethereum, esta integracion nunca fue probada ni realizada entre dichas blockchain. Es decir que no existía una integración directa entre Fabric y Ethereum. Se intentó hacer los ajustes e implementaciones para lograr una integración entre ambas blockchains, sin éxito. Asimismo, se intentó contactar a sus creadores para obtener soporte, pero no hubo respuesta. Por lo tanto, se optó por descartarla por falta de madurez y soporte.

4.3. Optimism

4.3.1. Introducción

Optimism (*Optimism Foundation*, 2024) es una solución de escalabilidad de segunda capa (L2) para Ethereum, diseñada para mejorar la velocidad y reducir los costos de las transacciones en la red. Utilizando tecnología de Rollups Optimistas (*Optimistic Rollups*). Optimism permite procesar una gran cantidad de transacciones fuera de la cadena principal de Ethereum, mientras mantiene la seguridad y descentralización inherentes a la red.

Los Rollups Optimistas funcionan agrupando múltiples transacciones ejecutadas off-chain en un único lote, que luego se envía y se registra en la cadena principal de Ethereum. Esta agrupación permite reducir significativamente los costos de transacción y aumentar el rendimiento del sistema, ya que las transacciones individuales dentro del lote no requieren ser procesadas una por una en la cadena principal.

4.3.2. Validacion de Transacciones

La seguridad en Optimism se garantiza mediante un mecanismo de prueba de fraude. En este modelo, las transacciones agrupadas se consideran válidas de manera optimista, pero pueden ser desafiadas por cualquier participante de la red a través de pruebas de fraude. Si se detecta una transacción fraudulenta, se revierte y se penaliza al actor malintencionado, manteniendo así la integridad del sistema. Aunque los validadores operan *off-chain*, en caso de disputas, un contrato inteligente actúa como árbitro, determinando el estado final de la cadena. Esto garantiza la seguridad, ya que la responsabilidad última recae en la seguridad de la blockchain.

4.3.3. Análisis de la solución

Optimism fue rechazado debido a que no se adecuada a los requerimientos del sistema ya que no se limita exclusivamente a ser un puente entre dos blockchain homogéneas, sino que también abarca una solución de capa 2 (L2) para Ethereum. El propósito específico de esta solución es mejorar la escalabilidad de las transacciones por segundo y reducir sus costos.

Debido a que Optimism es una solución especializada para la reducción de costos de transacciones en Ethereum, está restricto en diversos sentidos, tales como la arquitectura, el nivel de compatibilidad, o incluso la posibilidad de reutilizar código. Todo esto se contrapone directamente a los objetivos del proyecto, ya que se desea una solución para interoperar blockchains heterogeneas. Por lo tanto que no se adecua como solución.

4.4. Conclusión del análisis

Las tecnologías mencionadas anteriormente permiten ofrecer un panorama sobre el funcionamiento de los bridges y cómo estos abordan la interoperabilidad y la validación de transacciones en las respectivas blockchains. A pesar de que ninguna de las soluciones existentes resuelve requerimientos del presente proyecto, permiten obtener valiosas ideas para el diseño de una solución.

Estas tecnologías permitieron identificar que componentes son necesarios para la validación de transacciones y los cambios necesarios en el BIG para adaptarlo a las necesidades del proyecto. En resumen, el análisis de estas soluciones brindan una sólida base sobre la cual construir una integración efectiva y personalizada para el proyecto.

Capítulo 5

Solución Propuesta

En el presente capítulo se describe la solución propuesta al problema de validación de transacciones cross-blockchain entre Hyperledger Fabric y Ethereum, teniendo en cuenta el análisis de requerimientos. En la sección 5.1 se expone el objetivo que debe cumplir la solución en aspectos generales. En la sección 5.2 se brinda el suficiente contexto para comprender las posibles alternativas de validaciones. Para esto, se detalla como están compuestos los bloques y las transacciones en Fabric, y las posibles verificaciones que se pueden realizar sobre estos. Asimismo, en la sección 5.3 se evalúan y clasifican las posibles verificaciones en diferentes niveles de seguridad, optando y escogiendo el nivel de seguridad pertinente para en los siguientes capítulos realizar su diseño e implementación.

5.1. Objetivo

El objetivo del proyecto es proveer una solución que permita realizar transacciones cross-blockchain desde Hyperledger Fabric a Ethereum. Esto significa que aquellas transacciones que se realicen en Hyperledger Fabric repercutan y generan como contraparte una transacción en Ethereum. Adicionalmente, esta última transacción debe ser validada en Ethereum, de forma que únicamente sean aceptadas transacciones que hayan sido escritas en el bloque de Hyperledger Fabric. Por esto es importante comprender la formación de un bloque y su composición en Hyperledger Fabric, dado que es la información que se tendrá disponible para lograr validar la autenticidad e integridad de la transacción. Asimismo se busca que la solución sea de tipo *Trusted Relay*, descrito en la sección de 2.2.4, dado que dentro de las soluciones para interoperar, es la que más se ajusta al problema.

5.2. Validación sobre bloque y transacciones de Hyperledger Fabric

Como se muestra en la figura 5.1, un bloque en Fabric puede contener múltiples transacciones dentro del campo *Data*. El campo *payload* (indicado en la figura 5.2 por el número 1) corresponde al *payload* de una transacción específica. Para obtener más información sobre los bloques de Hyperledger Fabric, consulte el anexo B.1.

Figura 5.1: Ejemplo de composición de un bloque Fabric

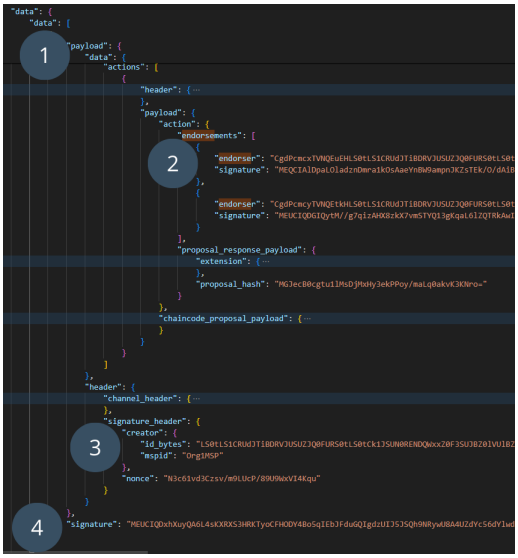


Figura 5.2: Ejemplo de composición de un transacción dentro de un bloque Fabric

Las validaciones pueden llevarse a cabo a nivel de bloque o a nivel de transacción. A continuación, se destacan las validaciones identificadas como factibles de implementar:

5.2.1. A nivel de bloque:

1. *Validación de integridad del bloque*: Esta validación se basa en comparar el *payload* hasheado (Indicado por número 1 en figura 5.1) con el *data_hash* del header (Indicado por número 2 en figura 5.1). Esta validación sirve para saber que el bloque no fue modificado.

5.2.2. A nivel de transacción:

1. *Validación de firma del creador de la transacción*: Esta validación se basa en validar si la firma sobre el *payload* (Indicado por el número 4 en figura 5.2) fue realizada con la clave privada correspondiente a la clave pública *id_bytes* (Indicado por el número 3 en figura 5.2).

2. *Validación de Endorsers*: Esta validación se basa en validar si la firma de los endorsers (Indicado por el número 2 en figura 5.2) está presente y validan con la correspondiente clave pública de dicho Endorser.

3. *Validación de validez transacciones*: Solo aquellos bloques que sean válidos y hayan pasado por el mecanismo de consenso de Fabric deben ser considerados. En la metadata se agrega un booleano el cual denota si un bloque válido o no (Indicado por número 3 en figura 5.1).

5.2.3. Importancia de cada validacion

Las validaciones descritas a nivel de bloque y a nivel de transacción no tienen el mismo nivel de importancia, ya que cada una aborda aspectos diferentes del proceso.

1. La validación más crítica es la de la integridad del bloque, ya que permite detectar si un bloque ha sido modificado. Esta verificación es esencial para una solución que pretende asegurar la validez de las transacciones.

2. La validación de la validez de las transacciones es fundamental, ya que confirma si las transacciones han sido procesadas adecuadamente por el mecanismo de consenso de Hyperledger Fabric. Esta validación también incluye la comprobación de que los *endorsers* han firmado las transacciones.

3. La validación de la firma del creador de la transacción es de importancia intermedia, ya que permite verificar la identidad del creador de la transacción y asegurar que pertenece a una organización reconocida.

4. La validación de las firmas de los *endorsers* también es relevante, ya que identifica qué organizaciones han validado la transacción, pero tiene un impacto menor en comparación con las otras validaciones más críticas.

5.3. Evaluación de posibles soluciones

A partir de las validaciones sobre bloque y sus transacciones descritas en la sección 5.2, el objetivo es clasificar posibles conjuntos de validaciones y asignarle un nivel de confiabilidad y seguridad. Esta asignación será basada en el grado de honestidad que necesitaría el sistema para su correcto funcionamiento. A menor nivel de honestidad, mayor será el grado de seguridad. A continuación se presentan los distintos niveles de seguridad.

5.3.1. Seguridad muy alta

Para este escenario se busca un sistema donde la confianza sea mínima. Se deben validar todos los datos del bloque. Es decir, todas las validaciones mencionadas en la sección 5.2.

5.3.2. Seguridad media

Se califica como un nivel de seguridad medio al contar con las dos validaciones más importantes: la validación de la integridad del bloque y la validación de la transacción como válida (ambas introducidas en la sección 5.2).

Al poder realizar estas ultimas dos validaciones mencionadas, tanto la validación del creador de la transacción como la validación de los *endorsers* ya estarían incluidas en las anteriores, debido a que se tratan de operaciones intermedias del mecanismo de consenso de Hyperledger Fabric. Por lo tanto, las restantes dos validaciones, se consideran de un segundo nivel de importancia.

5.3.3. Seguridad baja

Se clasifica como seguridad baja en cualquier caso que no se tengan las validaciones más importantes vistas en la seccion 5.2.3.

5.3.4. Consideraciones para la selección de nivel de seguridad

Se analizaron dos posibles opciones para llevar a cabo estas validaciones: en la blockchain de Ethereum o en el propio gateway.

Si las validaciones se realizan en la blockchain de Ethereum, el gateway actuaría de manera completamente agnóstica, limitándose a reenviar mensajes. En este caso, Ethereum, mediante múltiples contratos inteligentes, se encargaría de todas las validaciones necesarias. Este enfoque reduce la necesidad de confiar en el gateway, ya que cualquier invocación de los contratos inteligentes sería validada independientemente de la lógica aplicada en el gateway.

La otra opción es realizar las validaciones en el gateway. En este escenario, el gateway sería responsable de todas las validaciones previas y generaría una firma sobre la transacción a realizar en Ethereum. Luego, Ethereum validaría dicha firma, asegurándose de que las validaciones fueron efectuadas correctamente por el gateway.

Desde la perspectiva de la seguridad, el escenario más favorable sería realizar todas las validaciones en Ethereum. De esta manera, se evitaría el riesgo de que el gateway, si fuera alterado o controlado por una entidad desconocida, pudiera causar comportamientos no deseados. Sin embargo, existen varios factores que justifican la opción de realizar las validaciones en el gateway:

- Uno de los principales desafíos fue la serialización de un bloque completo

en Ethereum. Esto se debe a que Solidity (Solidity, s.f.) no ofrece bibliotecas maduras para la conversión de JSON o Protobuf[1].

- Costo elevado de la serialización: Se realizó un análisis de costos basado en la serialización de un mensaje simple en formato Protobuf utilizando la biblioteca `solidity-protoc`. El costo de gas resultante fue prohibitivo.

Además de las razones mencionadas, es importante señalar que Hyperledger Fabric es una blockchain con permisos (permissioned blockchain). Esto implica que se requieren permisos específicos para interactuar con Hyperledger Fabric. Por lo tanto, cualquier gateway que reenvíe transacciones debe ser auditado y autorizado por las organizaciones que gestionan Hyperledger Fabric para interactuar con la blockchain. En consecuencia, una entidad no autorizada no podría reenviar ni validar transacciones de Fabric a Ethereum. En cambio, si ambas blockchains fuesen sin permisos (permissionless), cualquier individuo podría interactuar con ambas, permitiendo la creación de gateways propios. En este último caso, sería indispensable realizar todas las validaciones en Ethereum.

5.3.5. Selección de nivel de seguridad a utilizar

Una vez definidos cómo y dónde se realizarían las validaciones sobre el bloque de Fabric y sus transacciones, se discutirá el nivel de seguridad implementado y las razones detrás de esta elección.

A partir de las pruebas de concepto realizadas para las validaciones, se decidió optar por un nivel de seguridad media. Esta decisión se tomó debido a que una validación específica no se logró ejecutar con éxito: la validación sobre los *endorsers*. Esta validacion aunque confirma que un *endorser* firmó la transacción, no determina ni garantiza que la transacción se haya ejecutado con éxito ni su integridad.

5.3.6. Conclusión

En resumen, se optará por un nivel de seguridad medio, que incluye la validación de la integridad del bloque, la firma del creador del bloque en Fabric, y la validación de cada transacción realizada por *Fabric*. Además, dado que la integración se realiza sobre una blockchain con permisos (*permissioned blockchain*), las validaciones se llevarán a cabo en el gateway, ya que se requieren permisos específicos para interactuar con la blockchain de Hyperledger Fabric. Por limitaciones técnicas previamente mencionadas, no fue posible validar las transacciones directamente en Ethereum.

[1] Protobuf es un protocolo utilizado para serializar información estructurada

Capítulo 6

Diseño de la solución

Este capítulo tiene por objetivo describir, detallar, y dar una visión general de la solución, junto al proceso y transiciones que atravesó la arquitectura para finalmente concluir en la actual solución. Habiendo considerado todos los niveles de seguridad de la solución planteada en el capítulo 5, el diseño se centrará en el nivel de seguridad media descrita en la sección 5.3.2.

En la sección 6.1 se describe a alto nivel los principales componentes y la interacción entre ellos para lograr la validación de transacciones cross-blockchain. De esta manera se le brinda al lector el suficiente contexto para entender los detalles del diseño descritos en la sección 6.2. En esta última sección se detalla la arquitectura a más bajo nivel junto a sus vistas y casos de uso más relevantes. Por último, en la sección 6.3 se listarán las decisiones más relevantes respecto a la arquitectura y su diseño.

6.1. Descripción

En la figura 6.1 se presenta la arquitectura inicial del BIG[1]. La arquitectura presentada se compone de un Router y dos conectores. Estos dos conectores fueron modificados para soportar la validación de transacciones. Una de las principales ventajas del uso de conectores es la separación de responsabilidades que permiten estos. Esto permite agregar, modificar o eliminar conectores de manera sencilla.

[1]BIG por "Blockchain Interoperability Gateway" es una arquitectura propuesta en (Bruno Bradach, Juan Nogueira, 2021)

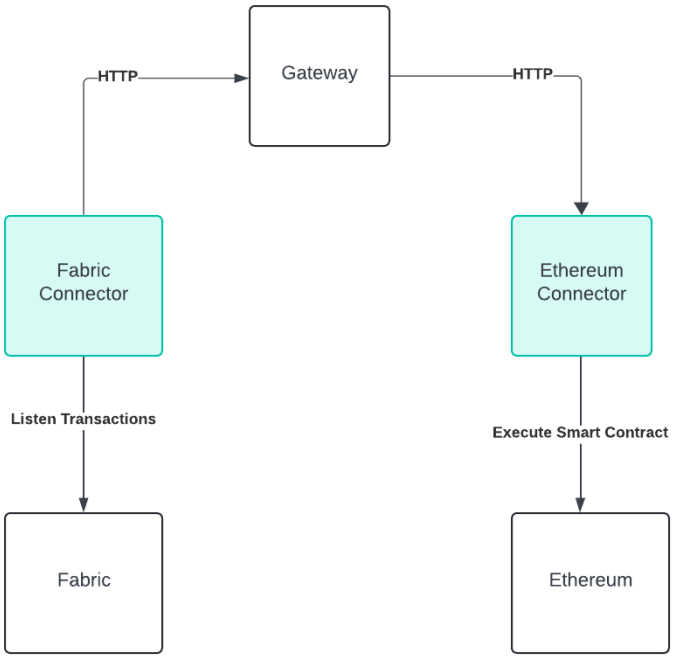


Figura 6.1: Arquitectura BIG. En verde, los componentes que fueron actualizados para poder soportar la validación de transacciones cross-blockchain

En aspectos generales, la solución se puede resumir en cuatro grandes etapas por las que pasa una transacción cross-blockchain.

1. La lectura del bloque y la validación de las transacciones ocurridas en Fabric
2. El parseo[2] y mapeo[3] de cada transacción en Fabric a la transacción equivalente en la blockchain de Ethereum.
3. La generación de una firma sobre la transacción equivalente.
4. Toda transacción que sea ejecutada en Ethereum pasará por el proceso de validación que se basa en validar la firma creada en la etapa anterior.

Estas etapas se pueden observar en la figura 6.2 donde las etapas uno y dos ocurren en Fabric Connector, mientras que la tercera etapa ocurre en el Ethereum Connector. Finalmente, la cuarta etapa ocurre en la blockchain de

[2]El término "parseo" hace referencia al proceso de analizar cierta información de entrada, para extraer su contenido significativo y convertirla en una estructura de datos

[3]El termino "mapeo" o "mapear" hace referencia a la asociación de un elemento con otro, generando una correlación entre ambos

Ethereum, donde se realiza la validación de la firma. Esta validación garantiza que las transacciones de Fabric hayan sido validadas e invocadas desde un Ethereum Connector que haya firmado con la clave privada específica.

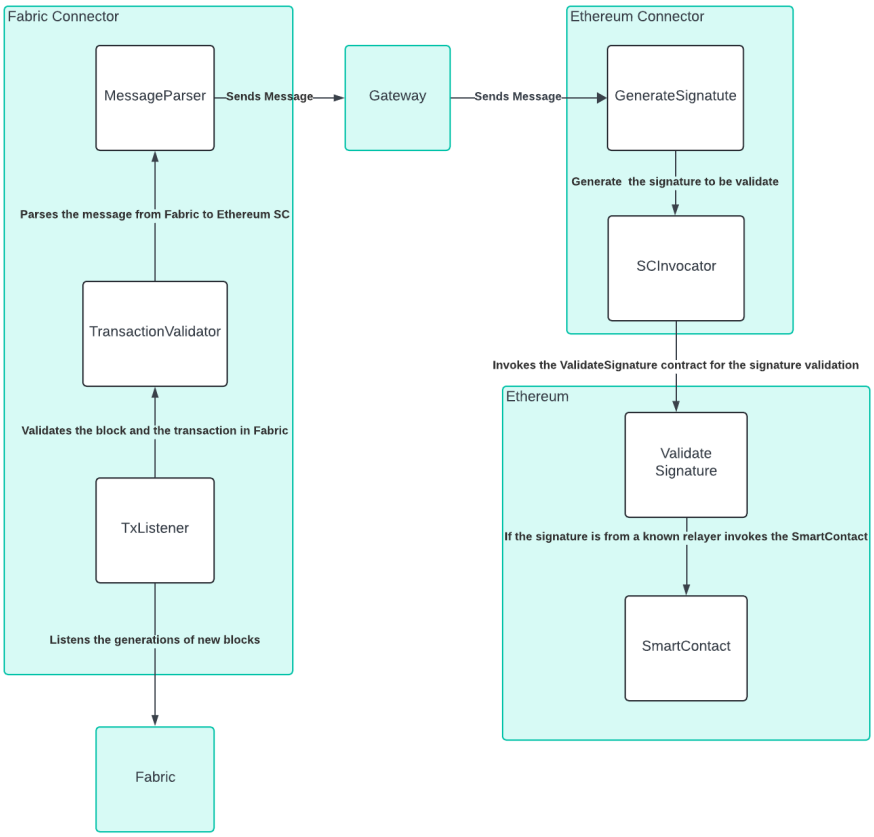


Figura 6.2: Flujo de validación de transacción cross-blockchain

En la figura 6.2 se presenta un flujo de transacción validada desde la blockchain de Fabric hasta la blockchain de Ethereum. El primer componente que se observa es el encargado de la lectura de las transacciones de la blockchain Fabric y se llama *TxListener*. Este se encuentra en el Fabric Connector y se encarga de realizar las lecturas de los bloques que son persistidos en la blockchain de Fabric junto a sus correspondientes transacciones. Este mecanismo de lectura asegura que la transacción haya sido ejecutada y escrita en un bloque de Fabric. Luego el componente *TransactionValidator* se encarga de validar el bloque y las transacciones ocurridas en Fabric. El componente llamado *MessageParser* se encarga de traducir cada transacción ocurrida en Fabric a una

nueva transacción con sus correspondientes características para ser ejecutada en Ethereum. Esta transacción es enviada al router y reenviada por este hacia el Ethereum Connector. Posteriormente las transacciones atraviesan el componente llamado *GenerateSignature* que se encuentra en el Ethereum Connector. Este componente genera una firma sobre la transacción que posteriormente se validará en un Smart Contract en Ethereum. La firma es generada utilizando una clave privada alojada también en el Ethereum Connector. Finalmente, el componente *SCInvocator* se encarga de realizar la invocación al Smart Contract (*ValidateSignature*) de Ethereum. El Smart Contract de *ValidateSignature* tiene dos funciones. La primera es validar la firma de la transaccion que es generada por el componente GenerateSignature utilizando la clave pública del certificado del conector de Ethereum. La segunda función es de enrutar la transacción al Smart Contract correspondiente para realizar la transacción. Para el escenario propuesto, este último Smart Contract podría referirse a *CarToken* o *CarTokenNFT*. Por un lado *CarToken* es un Smart Contract que permite operar con Token's ERC 20, mientras que *CarTokenNFT* permite operar con NFT's ERC 720. En el escenario propuesto se utilizan para transferir y minar estos al usuario.

6.2. Arquitectura de la solución

En esta sección, se presenta una visión general de la arquitectura para el escenario propuesto en la sección 3.5, explicando su funcionamiento y la interacción entre sus componentes, utilizando el modelo 4+1.

6.2.1. Vista de Casos de Uso

En esta sección, se explorará la Vista de Caso de Uso más relevante para la arquitectura. Este se observa en la figura 6.3. Asimismo, a continuación se detalla la ficha del caso de uso.

Nombre	Comprar Automóvil
Actores	Usuario
Descripción	El usuario tiene la intención de adquirir un automóvil. Para ello, navega por el catálogo de automóviles disponibles, elige uno y completa la transacción de compra.
Precondición	El usuario debe contar con una billetera digital en Ethereum.
Flujo Principal	1. El usuario accede a la plataforma CarSeller DApp. 2. La plataforma presenta al usuario el catálogo de automóviles disponibles. 3. El usuario busca y selecciona un automóvil de su preferencia. 4. El usuario solicita la compra del automóvil seleccionado. 5. La plataforma solicita al usuario que indique su cuenta de Ethereum. 6. El usuario ingresa su cuenta de Ethereum en Metamask. 7. El usuario completa el proceso de pago del automóvil. 8. El usuario verifica que los tokens correspondientes se hayan depositado en su cuenta de Ethereum.
Postcondición	El usuario ha realizado la compra del automóvil deseado y los tokens correspondientes se han añadido a su cuenta de Ethereum.

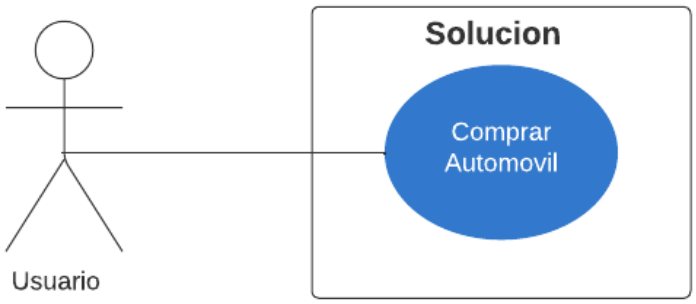


Figura 6.3: Diagrama de Casos de Uso

6.2.2. Vista lógica

En esta sección se describen el caso de uso de comprar automóviles, el cual se describirá mediante un diagrama de secuencia.

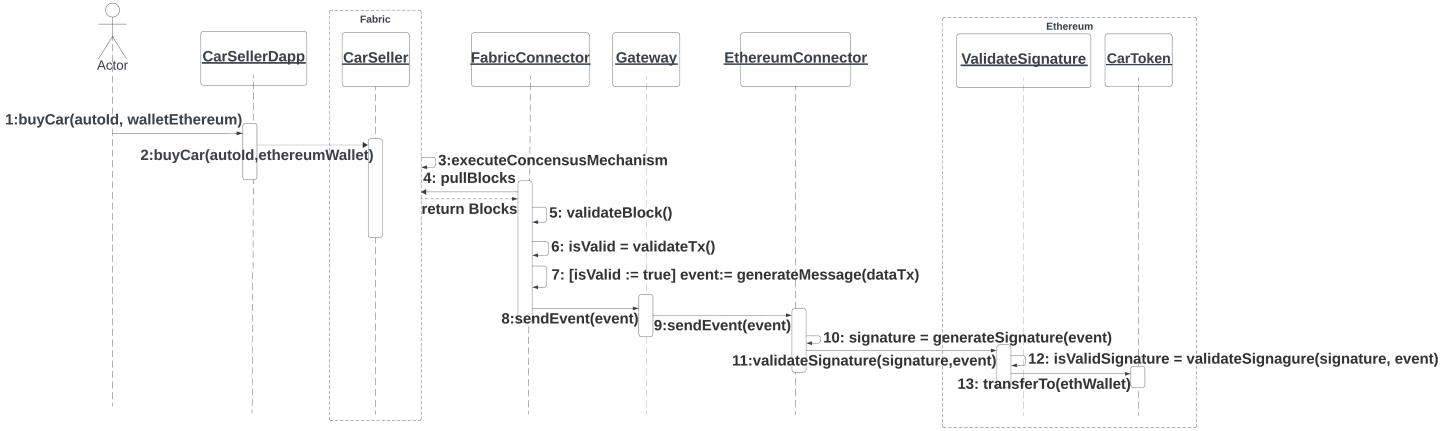


Figura 6.4: Diagrama de Secuencia Comprar Automóvil

Compra de un automóvil

En esta sección se mostrará el proceso que conlleva la ejecución del caso de uso descrito en la sección 3.6.1

El proceso representado en la Figura 6.4 describe los pasos que ocurren cuando un usuario desea adquirir un automóvil a través del sistema. Antes de iniciar este caso de uso, el usuario debe visualizar los automóviles disponibles mediante la *CarSellerDApp* y seleccionar el vehículo de su interés. Para proceder con la transacción, el usuario debe poseer una billetera digital en Ethereum. La interfaz web guiará al usuario para que se conecte a su billetera digital utilizando Metamask (Metamask, s.f.).

El proceso de compra se desarrolla de la siguiente manera:

1. El caso de uso inicia cuando el usuario realiza la compra del automóvil utilizando la funcionalidad *buyCar(carId, walletEthereum)* de la DApp de Fabric. El parámetro "carId" representa el automóvil que desea comprar, mientras que "walletEthereum" es la billetera digital donde se asignarán los tokens con el valor del automóvil.

2. La invocacion de *buyCar(carId, walletEthereum)* desencadena una llamada al desde la DApp al contrato inteligente *CarSell*. El contrato registra que el usuario ha comprado el automóvil "x", el cual tiene un valor determinado. Una cantidad equivalente se convertirá en la cantidad de "CarTokens" que se depositarán en la billetera digital del usuario en Ethereum. La ejecución del contrato inteligente genera una nueva transacción que será analizada posteriormente.

3. Dado que se ha generado una nueva transacción en Fabric, se activa el mecanismo de consenso para decidir la validez de la transacción.

4. El componente *FabricConnector* está continuamente escuchando la blockchain, detectando la llegada de nuevos bloques para su procesamiento.

5. Al recibir un nuevo bloque, *FabricConnector* se encarga de validar tanto la integridad del bloque como la autenticidad de la firma del creador.

6. Adicionalmente, *FabricConnector* verifica la validez de las transacciones contenidas en el bloque. Si la transacción es confirmada como válida, la información de Fabric se transforma para ser compatible con Ethereum.

7. En este paso, se ejecuta la función *generateMessage(dataTx)*, la cual toma los datos de ejecución del Smart Contract en Fabric (entradas y salidas generadas por el Smart Contract) y los transforma a un formato de datos para la invocación al correspondiente contrato inteligente de Ethereum. En este punto, se determina qué contrato inteligente se invocará en Ethereum y se definen los parámetros e información necesarios.

8. La información se envía al *Gateway*, que se encarga de facilitar la comunicación entre los diferentes componentes del sistema.

9. El *Gateway* transmite la información al *EthereumConnector*.

10. En el *EthereumConnector*, se firma la transacción recibida por el *Gateway*.

11. Se procede a llamar al contrato inteligente *ValidateSignature* con la firma generada y la información correspondiente para su propagación en Ethereum.

12. El contrato inteligente *ValidateSignature* es ejecutado, específicamente la función *validateSignature(ValidateSignature, dataTX)*, que valida la firma recibida, asegurando que la información enviada sea la misma que la firmada y que la firma provenga de la fuente esperada.

13. El contrato inteligente *CarToken* es invocado por el Smart Contract *ValidateSignature* para transferir los tokens correspondientes a la billetera digital del usuario que realizó la compra del automóvil.

6.2.3. Vista de Implementación

La Figura 6.5 muestra los componentes principales del sistema y cómo interactúan entre sí.

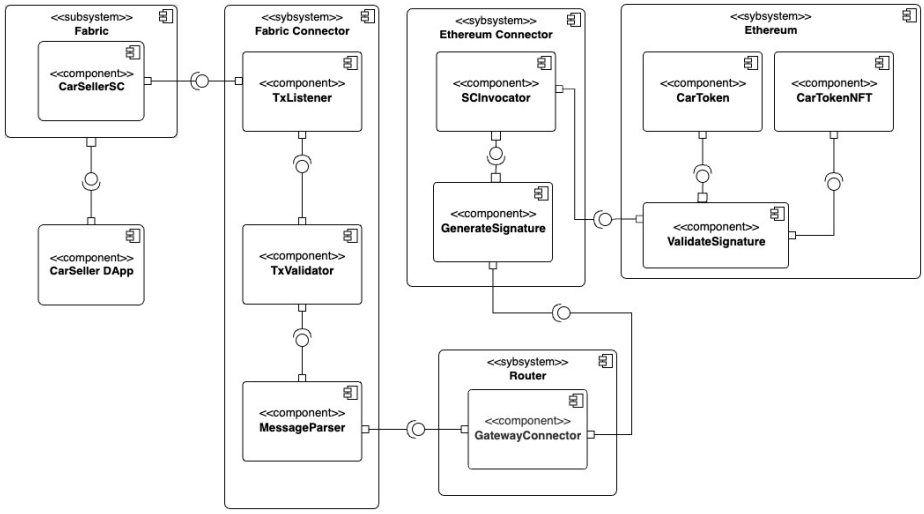


Figura 6.5: Diagrama de componentes

CarSeller DApp

La aplicación web CarSeller DApp interactúa con la blockchain de Fabric a través de su API, mostrando los automóviles disponibles para la venta.

CarSell

CarSell es un Smart Contract desplegado en la red de Fabric, diseñado para registrar las transacciones de compra de automóviles.

Fabric Connector

El *Fabric Connector* tiene como objetivo principal la escucha de los bloques generados en Fabric, su procesamiento y posterior envío al *Gateway*. Está compuesto por tres componentes principales:

- *TxListener*: Este componente intercepta todos los bloques generados por la blockchain de Fabric y filtra las transacciones que corresponden a ejecuciones del Smart Contract *CarSellSC*.
- *TransactionValidatior*: Este componente valida la integridad del bloque interceptado por el TxListener y válida sus correspondientes transacciones.
- *MessageParser*: Este componente se encarga de traducir y mapear las entradas y salidas del Smart Contract *CarSell* de Fabric a un formato adecuado para su ejecución en Ethereum. Finalmente esta traducción es enviada al Gateway. Es esencial destacar la necesidad de un mapeo entre los métodos de CarSell (Smart Contract en Fabric) y CarToken (Smart Contract en Ethereum). Por ejemplo, la compra de un automóvil en Fabric podría traducirse y mapearse en la creación y asignación de un NFT en Ethereum.

Router

El *Router* es un componente parte del *BIG*. Este es responsable de pasar datos de un conector a otro, siendo agnóstico de las blockchains que participan.

Ethereum Connector

El *Ethereum Connector* gestiona la comunicación con la blockchain de Ethereum y consta de dos componentes:

- *GenerateSignature*: Genera la firma para su validación posterior en la blockchain de Ethereum. Para la generación de esta firma se utiliza una clave privada alojada en el conector.
- *SCInvocator*: Es el cliente del Smart Contract *ValidateSignature* de Ethereum.

Ethereum

En Ethereum se despliegan tres Smart Contracts:

- *ValidateSignature*: Verifica que la transacción recibida proviene de un Gateway conocido. Para esto aloja en memoria la clave pública correspondiente a la clave privada utilizada por el componente GenerateSignature del *EthereumConnector*.
- *CarToken*: Implementa un contrato ERC20 que proporciona los tokens del automóvil comprado en Fabric.
- *CarTokenNFT*: Implementa un contrato ERC721 correspondiente a la representación del automóvil con NFT.

6.2.4. Vista de Distribución

En esta sección, exploraremos la infraestructura donde se alojan los componentes de la solución y los protocolos a través de los cuales interactúan.

La Figura 6.6 ilustra el despliegue de la solución, destacando dos componentes principales: *Docker* y *EVM* (Ethereum Virtual Machine).

Dentro del entorno de *Docker*, se despliegan dos redes distintas: la red de la solución, que alberga los contenedores propios de la solución, y la red de Fabric, esencial para el funcionamiento de la blockchain de Fabric.

La red de Fabric debe ser accesible desde el exterior para permitir la interacción de *Fabric Connector* y *CarSellerDApp*. Por consiguiente, esta red debe estar disponible para estos componentes, y la comunicación entre ellos se realiza mediante el protocolo HTTP. Sin embargo, el *Router*, cuya función es facilitar la comunicación entre *Fabric Connector* y *Ethereum Connector*, no requiere acceso desde fuera de la red y, por lo tanto, no necesita conectividad a las blockchain.

Por otro lado, el *Ethereum Connector* requiere acceso a Internet para establecer comunicación con la red externa de *EVM*. La interacción entre *Ethereum Connector* y *Ethereum* se lleva a cabo a través de la API de Ethereum utilizando el protocolo HTTP.

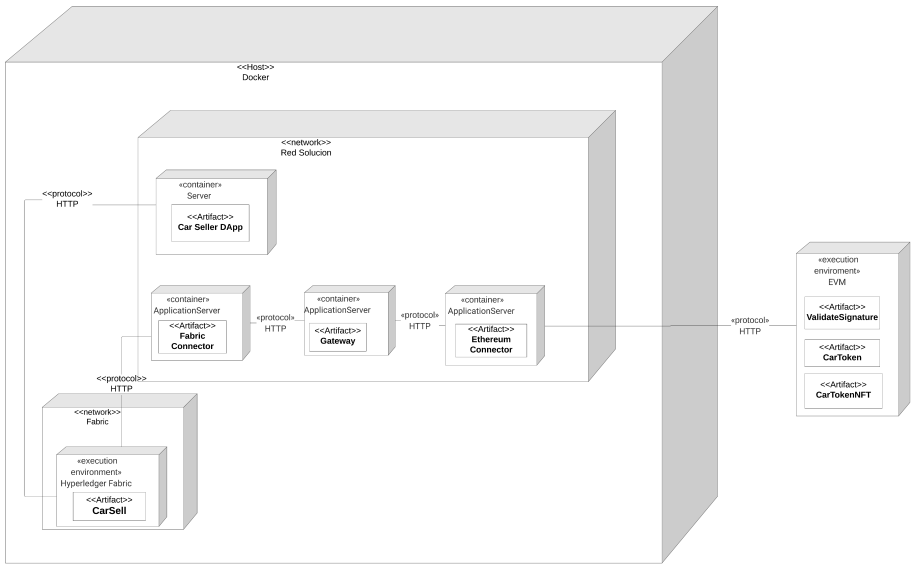


Figura 6.6: Diagrama de despliegue

6.3. Principales decisiones de diseño

En esta sección se detallan las decisiones claves que dieron forma a la solución propuesta. Estas son: la transición de una arquitectura monolítica a microservicios, el mecanismo de validación basado en confianza y la adopción de una arquitectura fundamentada en Token Bridge.

6.3.1. Migración de arquitectura Monolito a Microservicios tipo BIG

La arquitectura descrita en 6.2 no fue la arquitectura inicial implementada. A inicios se implementó una arquitectura de tipo Monolito debido a su sencillez y eficacia. En el Monolito el proceso de lectura de las transacciones en Fabric, validación, generación de firma e invocación de la transacción en Ethereum, ocurre todo dentro del mismo sistema, haciendo este un proceso más seguro. En cambio, en la arquitectura de microservicios se es necesario un sistema de Autenticación y Autorización entre el Ethereum Connector-Gateway y Gateway-Fabric Connector. De otra manera, cualquiera podría invocar directamente al Gateway con los parámetros correspondientes para lograr ejecutar transacciones en Ethereum mal intencionadas. La arquitectura de la solución requería que tuviese la capacidad de sumar nuevas implementaciones de diferentes blockchain sin mayores esfuerzos. Por esto se optó por la arquitectura de microservicios.

6.3.2. Mecanismo de validación con confianza

El sistema de validación de transacciones entre blockchain de esta solución se basa en la confianza. Esto es debido a las características inherentes a la blockchain de Fabric, dado que esta es una blockchain con permiso (permissioned). Esto significa que nadie ajeno a la blockchain puede leer, escribir, o interactuar con esta sin los permisos correspondientes. Es por esto que cualquier entidad para interactuar con Fabric necesitará los permisos correspondientes. Estos permisos implica un cierto grado de confianza sobre dicha entidad. Por lo tanto, cualquier interacción con Fabric requeriría permisos, lo que proporciona un nivel de confianza inherente.

6.3.3. Arquitectura basada en Token Bridge

La arquitectura aquí presente se fundamenta en el concepto de Token Bridge, detallado en la sección 4.1, aunque con ciertas simplificaciones.

La simplificación más notable radica en el número de validadores. En la solución propuesta, se cuenta con un único validador, que es la solución en sí misma, es decir, el conjunto de Fabric Connector-Gateway-Ethereum Connector. En contraste, Token Bridge emplea múltiples validadores, lo que le confiere un modelo de confianza distribuida. En particular, Token Bridge utiliza un mecanismo de consenso en el que una transacción es considerada válida si recibe la aprobación mayoritaria de sus validadores.

En el caso de la solución propuesta, al contar con un único validador, se opera con un nivel de confianza mayor. Si se deseara reducir la confianza en el único validador, sería necesario implementar un mecanismo de consenso con múltiples validadores, similar al utilizado por Token Bridge.

Capítulo 7

Implementación de la Solución

En este capítulo se detallan los aspectos más importantes en cuanto a la implementación de la solución. En la sección 7.1 se describe cuáles fueron las tecnologías utilizadas para toda la solución. En la sección de 7.2 se hace un enfoque en las implementaciones del escenario, contemplando las Dapp, Smart Contracts y Conectores. Por último, en la sección 7.3 se discute sobre las principales decisiones y limitaciones de la implementación realizada junto a posibles alternativas.

7.1. Tecnologías utilizadas

En la figura 7.1 se presenta una visión general de cada componente junto con la tecnología empleada para su implementación. Entre los componentes más importantes se destacan:

- La aplicación descentralizada (Dapp) denominada *CarSellerDApp*, que fue implementada utilizando React (Facebook, 2024) para el front-end.
- Para la API CarSell, se utilizó Node.js (*A JavaScript runtime built on Chrome's V8 JavaScript engine.*, s.f.) junto con el framework Express (Foundation, s.f.).
- Para los conectores (Fabric Connector y Ethereum Connector) se utilizó Node.js y TypeScript (Microsoft, s.f.).
- Los contratos inteligentes (*ValidateSignature*, *CarToken* y *CarTokenNFT*) en Ethereum se implementaron utilizando Solidity.
- En el caso de Hyperledger Fabric, los contratos inteligentes se desarrollaron utilizando TypeScript.
- Finalmente, el Gateway se mantuvo sin modificaciones, utilizando Node.js.

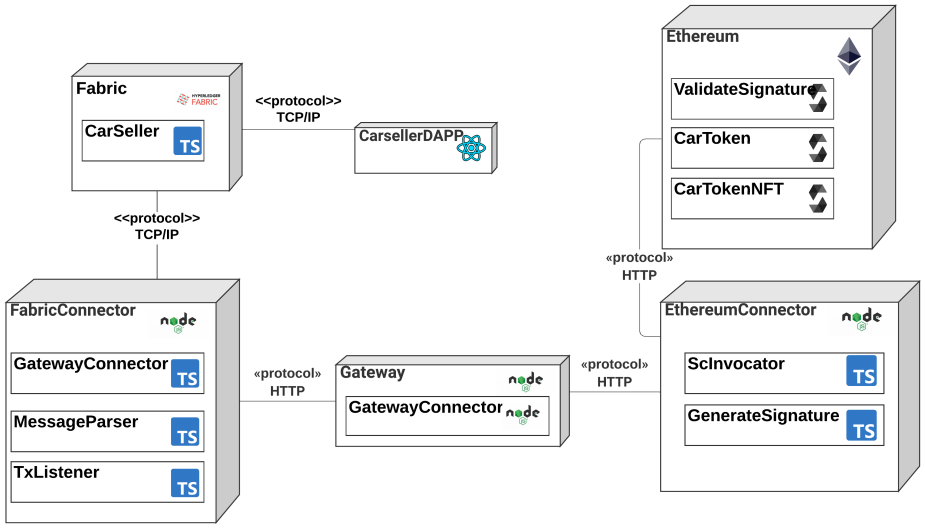


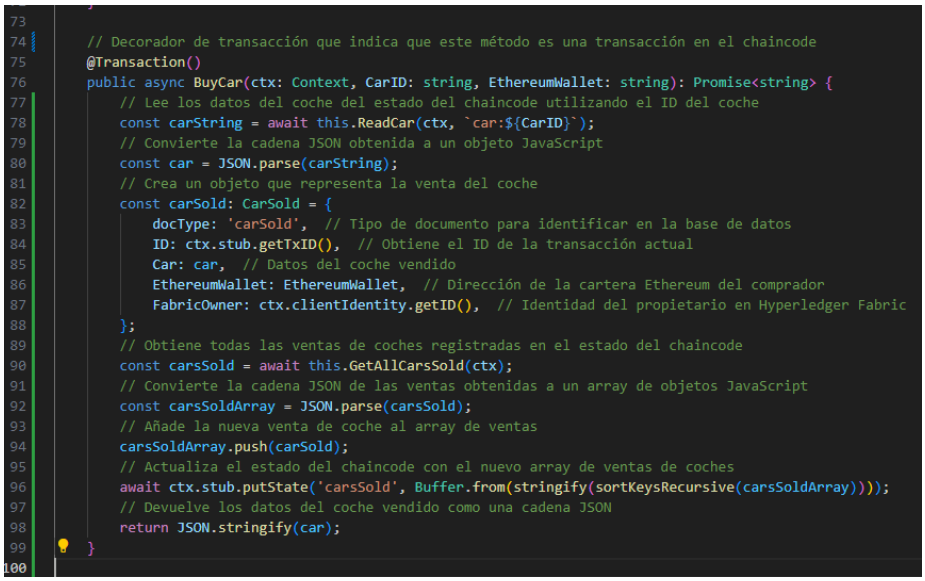
Figura 7.1: Diagrama Tecnologías utilizadas

7.2. Implementación del escenario

En esta sección se analizan los detalles más relevantes de la implementación de la solución. En la sección 7.2.1 y 7.2.4 se muestra parte de la implementación de los Smart Contract de Fabric y Ethereum respectivamente. De similar manera, en la sección 7.2.2 y 7.2.3 se muestra parte de los aspectos más importantes de las implementaciones de los conectores de Fabric y Ethereum respectivamente.

7.2.1. Implementación de Smart Contract de Fabric

En la figura 7.2 se muestra el método *BuyCar* del contrato inteligente *carSell* en Hyperledger Fabric. Este método facilita la venta de automóviles en la plataforma Fabric y asocia al comprador del vehículo con una billetera digital de Ethereum.



```
73
74      // Decorador de transacción que indica que este método es una transacción en el chaincode
75      @Transaction()
76      public async BuyCar(ctx: Context, CarID: string, EthereumWallet: string): Promise<string> {
77          // Lee los datos del coche del estado del chaincode utilizando el ID del coche
78          const carString = await this.ReadCar(ctx, `car:${CarID}`);
79          // Convierte la cadena JSON obtenida a un objeto JavaScript
80          const car = JSON.parse(carString);
81          // Crea un objeto que representa la venta del coche
82          const carSold: CarSold = {
83              docType: 'carSold',  // Tipo de documento para identificar en la base de datos
84              ID: ctx.stub.getTxID(),  // Obtiene el ID de la transacción actual
85              Car: car,  // Datos del coche vendido
86              EthereumWallet: EthereumWallet,  // Dirección de la cartera Ethereum del comprador
87              FabricOwner: ctx.clientIdentity.getID(),  // Identidad del propietario en Hyperledger Fabric
88          };
89          // Obtiene todas las ventas de coches registradas en el estado del chaincode
90          const carsSold = await this.GetAllCarsSold(ctx);
91          // Convierte la cadena JSON de las ventas obtenidas a un array de objetos JavaScript
92          const carsSoldArray = JSON.parse(carsSold);
93          // Añade la nueva venta de coche al array de ventas
94          carsSoldArray.push(carSold);
95          // Actualiza el estado del chaincode con el nuevo array de ventas de coches
96          await ctx.stub.putState('carsSold', Buffer.from(stringify(sortKeysRecursive(carsSoldArray))));
97          // Devuelve los datos del coche vendido como una cadena JSON
98          return JSON.stringify(car);
99      }
100
```

Figura 7.2: Implementación de método *BuyCar* del Smart Contract *carSell* de Fabric

Para el segundo escenario, que genera un NFT en Ethereum, se utiliza la operación *CreateCarNFT* del mismo contrato inteligente. La implementación de esta operación, mostrada en la figura 7.3, es idéntica a la del método *BuyCar*. Esta similitud se debe a que se busca mantener la misma funcionalidad de venta de un automóvil, pero orientada a la generación de un *NFT* en lugar de un *CarToken*.

```
95      @Transaction()
96      public async CreateCarNFT(ctx: Context, CarID: string, EthereumWallet: string): Promise<string> {
97          const carString = await this.ReadCar(ctx, `car:${CarID}`);
98          const car = JSON.parse(carString);
99
100         const carSold: CarSold = {
101             docType: 'carSold',
102             ID: ctx.stub.getTxID(),
103             Car: car,
104             EthereumWallet: EthereumWallet,
105             FabricOwner: ctx.clientIdentity.getID(),
106         };
107         const carsSold = await this.GetAllCarsSold(ctx);
108         const carsSoldArray = JSON.parse(carsSold);
109         carsSoldArray.push(carSold);
110         await ctx.stub.putState('carsSold', Buffer.from(stringify(sortKeysRecursive(carsSoldArray))));
111         // we insert data in alphabetic order using 'json-stringify-deterministic' and 'sort-keys-recursive'
112         return JSON.stringify(car);
113     }
114
```

Figura 7.3: Implementación de método CreateCarNFT del Smart Contract carSell de Fabric

7.2.2. Implementación del conector Fabric

El conector de Fabric es un componente que utiliza la SDK de Fabric para realizar lecturas de bloques y transacciones. Dado que los bloques de Fabric están en formato binario, estas lecturas deben ser decodificadas. En la figura 7.4, se muestra este proceso de decodificación. La decodificación convierte la información binaria en clases que contienen detalles sobre el bloque, permitiendo así extraer campos de especial interés necesarios para las validaciones descritas en la sección 5.2.

```
// Función que parsea un bloque de Hyperledger Fabric
export function parseBlock(block: common.Block): Block {
    const validationCodes = getTransactionValidationCodes(block);
    const header = assertDefined(block.getHeader(), 'Missing block header');
    // Crea un objeto 'data' que contiene métodos para acceder a diferentes partes del bloque
    const data = {
        // Método para obtener el número del bloque como un BigInt
        getNumber: () => BigInt(header.getNumber()),
        // Método para obtener los datos del bloque
        getData: () => block.getData(),
        // Método para obtener el encabezado del bloque como un objeto
        getHeader: () => header.toObject(),
        // Método para obtener las transacciones del bloque, cacheando el resultado para eficiencia
        getTransactions: cache(() =>
            // Deserializa y obtiene los payloads del bloque
            getPayloads(block)
                // Parsea cada payload
                .map((payload, i) => parsePayload(payload, validationCodes[i]))
                // Filtra los payloads para quedarse solo con las transacciones de endosadores
                .filter((payload) => payload.isEndorserTransaction())
                // Mapea cada payload filtrado a una transacción con su informacion deserializada
                .map(newTransaction)
        ),
        // Método para obtener el bloque original en formato proto
        toProto: () => block,
    };
    // Devuelve el objeto 'data' que representa el bloque parseado
    return data;
}
```

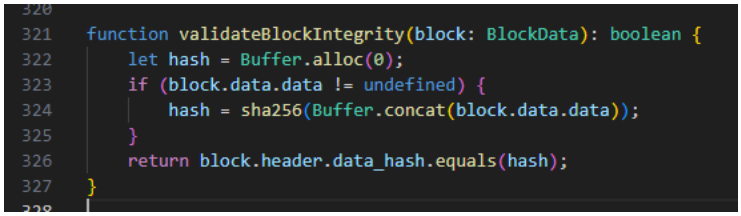
Figura 7.4: Función *parseBlock* que descodifica el bloque en estructuras de datos manipulables

En la figura 7.5 se presenta el código asociado a la validación del bloque y sus transacciones. En dicha figura se observa la invocación a la validación de la integridad del bloque, introducida en la sección 5.2.1. Asimismo, se observa la validación de la firma del creador de la transacción, descrita en 5.2.2. También, se observa el filtrado de las transacciones válidas conforme a lo descrito en la sección 5.2.2. Finalmente, en la misma imagen se observa el uso de los llamados checkpoints que funciona como un registro de transacciones procesadas. Esta funcionalidad guarda el número de bloque y el número de transacción correspondiente a la procesada en un archivo. De este modo, cuando se inicia el Ethereum Connector se retoma desde la última transacción procesada, asegurando no procesar las transacciones más de una vez.

En la figura 7.6 y 7.7 se observan la implementación de la validación de la integridad del bloque y la firma del creador de la transacción respectivamente.

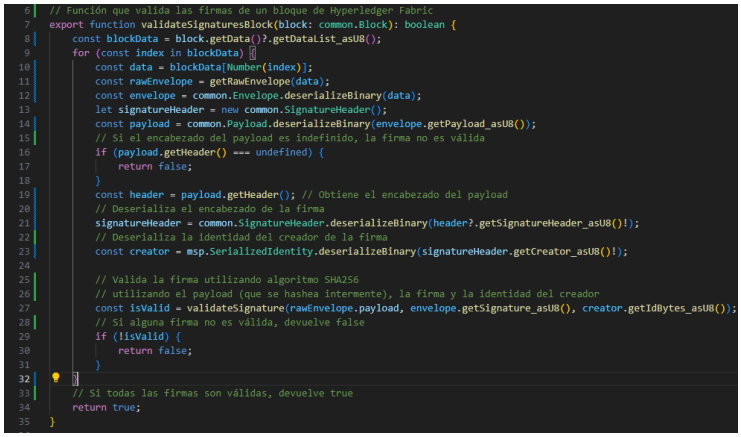
```
async validate(): Promise<void> {
    const blockNumber = this.#block.getNumber();
    console.log(`\nReceived block ${blockNumber}`);
    // Deserializacion del bloque
    const block = parseBlockData(this.#block.toProto());
    // Se valida la integridad del bloque
    const isValidBlock = validateBlockIntegrity(block);
    console.log(`Block ${blockNumber} integrity is  ${isValidBlock ? 'valid' : 'invalid'}`);
    // Se valida la firma del creador de la transaccion
    const isValidSignatures = validateSignaturesBlock(this.#block.toProto());
    console.log(`Block ${blockNumber} signature is  ${isValidSignatures ? 'valid' : 'invalid'}`);
    // Se filtran las transacciones validas
    const validTransactions = this.#getNewTransactions().filter((transaction) => transaction.isValid());
    for (const transaction of validTransactions) {
        const transactionProcessor = new TransactionProcessor({
            blockNumber,
            store: this.#store,
            transaction,
        });
        await transactionProcessor.process();
        const transactionId = transaction.getChannelHeader().getTxId();
        // Se agrega la transaccion al checkpointer, esto permite no procesar las transacciones
        // mas de una vez al iniciar el Ethereum Connector
        await this.#checkpointer.checkpointTransaction(blockNumber, transactionId);
    }
    await this.#checkpointer.checkpointBlock(this.#block.getNumber());
}
```

Figura 7.5: Función validate que realiza las validaciones sobre el bloque y sus respectivas transacciones



```
function validateBlockIntegrity(block: BlockData): boolean {
    let hash = Buffer.alloc(0);
    if (block.data.data != undefined) {
        hash = sha256(Buffer.concat(block.data.data));
    }
    return block.header.data_hash.equals(hash);
}
```

Figura 7.6: Implementación de la validación de la integridad de un bloque



```
// Función que valida las firmas de un bloque de Hyperledger Fabric
export function validateSignaturesBlock(block: common.Block): boolean {
    const blockData = block.getData()?.getDataList_asU8();
    for (const index in blockData) {
        const data = blockData[Number(index)];
        const rawEnvelope = getRawEnvelope(data);
        const envelope = common.Envelope.deserializeBinary(data);
        let signatureHeader = new common.SignatureHeader();
        const payload = common.Payload.deserializeBinary(envelope.getPayload_asU8());
        // Si el encabezado del payload es indefinido, la firma no es válida
        if (payload.getHeader() === undefined) {
            return false;
        }
        const header = payload.getHeader(); // Obtiene el encabezado del payload
        // Deserializa el encabezado de la firma
        signatureHeader = common.SignatureHeader.deserializeBinary(header?.getSignatureHeader_asU8()!);
        // Deserializa la identidad del creador de la firma
        const creator = msp.SerializedIdentity.deserializeBinary(signatureHeader.getCreator_asU8()!);

        // Valida la firma utilizando algoritmo SHA256
        // utilizando el payload (que se hashea intermente), la firma y la identidad del creador
        const isValid = validateSignature(rawEnvelope.payload, envelope.getSignature_asU8(), creator.getIdBytes_asU8());
        // Si alguna firma no es válida, devuelve false
        if (!isValid) {
            return false;
        }
    }
    // Si todas las firmas son válidas, devuelve true
    return true;
}
```

Figura 7.7: Implementación de la validación de la firma del creador de la transacción

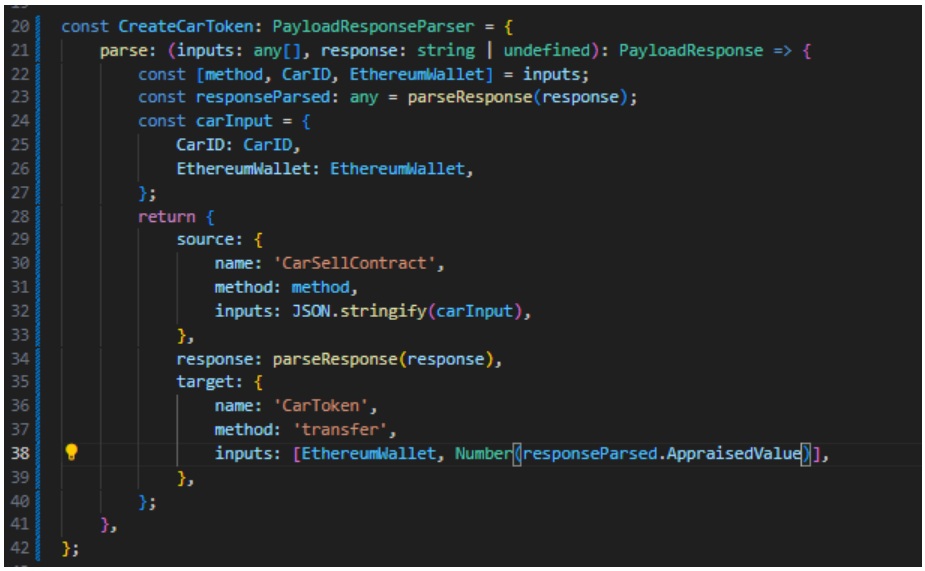
Una vez que la transacción de Fabric es validada, se realiza él parseo de la transacción para mapearla a la operación correspondiente en el Smart Contract en Ethereum. En la figura 7.8 se muestra un ejemplo de como se deserializan y descodifican las transacciones extraídas de un bloque. En dicho proceso se obtiene el nombre, método que se invocó inicialmente en Fabric y sus argumentos. Asimismo, se muestra cómo se emplean dichos argumentos, junto con el payload del bloque, para realizar el mapeo. Este mapeo establece una correspondencia uno a uno entre el nombre de la función del Smart Contract invocada en Fabric (`chaincodeArguments[0]`) y una implementación específica de la interfaz *PayloadResponseParsers* que se observa en la figura 7.9. Una de las implementaciones de esta interfaz, se observa en la figura 7.10. En esta implementación, se puede observar el mapeo de la información que se enviará al router. El campo `source` contiene el nombre del Smart Contract invocado en Fabric, junto con el método y sus parámetros de entrada. El campo `response` presenta el resultado de la ejecución en Fabric, mientras que `target` indica el Smart Contract que debe ser invocado en Ethereum, así como los parámetros necesarios.

```
export function generateRelayPayload(actions: peer.TransactionAction[]): PayloadResponse[] {
    const transactionsToRelay: PayloadResponse[] = []; // Conjunto de transacciones a enviar a Ethereum
    actions.forEach((action) => {
        // Inicio lectura y deserialización de toda información relevante extraída de la transacción. De esta forma se
        // obtiene que metodo del Smart Contract en Fabric se ejecutó y sus argumentos (p.j se obtiene la Ethereum wallet)
        const chaincodeActionPayload = peer.ChaincodeActionPayload.deserializeBinary(action.getPayload_asU8());
        const chaincodeProposalPayload = peer.ChaincodeProposalPayload.deserializeBinary(
            chaincodeActionPayload.getChaincodeProposalPayload_asU8());
        const chaincodeInvocationSpec = peer.ChaincodeInvocationSpec.deserializeBinary(
            chaincodeProposalPayload.getInput_asU8());
        const chaincodeArguments = decodeBase64Array(
            chaincodeInvocationSpec.getChaincodeSpec()?.getInput()?.getArgsList_asB64()
        );
        const actionProposalResponsePayload = chaincodeActionPayload?.getAction()?.getProposalResponsePayload_asU8();
        if (actionProposalResponsePayload !== undefined) {
            const proposalResponsePayload =
                peer.ProposalResponsePayload.deserializeBinary(actionProposalResponsePayload);
            const chaincodeAction = peer.ChaincodeAction.deserializeBinary(proposalResponsePayload.getExtension_asU8());
            // Fin lectura y deserializaciones.
            // En especial chaincodeArguments contiene el nombre del metodo invocado en Fabric (chaincodeArguments[0])
            // Y el resto de argumentos utilizados para la invocación.
            // chaincodeAction.getResponse()?.getPayload_asB64() contiene todo el payload de la transacción para poder ser
            // utilizado por el parser para extraer lo relevante para la realizacion de la transacción en Ethereum
            if (PayloadResponseParsers[chaincodeArguments[0]]) {
                const data = PayloadResponseParsers[chaincodeArguments[0]].parse(
                    chaincodeArguments,
                    chaincodeAction.getResponse()?.getPayload_asB64()
                );
                transactionsToRelay.push(data);
            } else {console.log('No parser found for ', chaincodeArguments[0]);}
        }});
    return transactionsToRelay;
```

Figura 7.8: Función *generateRelayPayload* responsable de mapear transacción de Fabric a una de Ethereum

```
interface PayloadResponseParser {
    parse: (inputs: any[], response: string | undefined) => PayloadResponse;
}
```

Figura 7.9: Interfaz PayloadResponseParsers que tendrá una implementación por cada diferente mapeo que se requiera



```
const CreateCarToken: PayloadResponseParser = {
    parse: (inputs: any[], response: string | undefined): PayloadResponse => {
        const [method, CarID, EthereumWallet] = inputs;
        const responseParsed: any = parseResponse(response);
        const carInput = {
            CarID: CarID,
            EthereumWallet: EthereumWallet,
        };
        return {
            source: {
                name: 'CarSellContract',
                method: method,
                inputs: JSON.stringify(carInput),
            },
            response: parseResponse(response),
            target: {
                name: 'CarToken',
                method: 'transfer',
                inputs: [EthereumWallet, Number(responseParsed.AppraisedValue)],
            },
        };
    },
};
```

Figura 7.10: Implementación de la interfaz PayloadResponseParsers utilizada para el mapeo del escenario de generación de un CarToken por la compra de un auto en Fabric utilizando la operación BuyCar

La figura 7.10 es un parser que hace el mapeo de la operación *BuyCar* en Fabric a la operación *transfer* del Smart Contract *CarToken* en Ethereum. Esta última operación transfiere tokens a la dirección de la billetera digital de Ethereum (`EthereumWallet`) la cantidad de tokens correspondiente (`AppraisedValue`).

La otra implementación de la interfaz *PayloadResponseParser* es la que se muestra en la figura 7.11. A diferencia de la anterior, esta implementación es únicamente cuando se realiza la operación *CreateCarNFT* en Fabric. Como consecuencia se mapea a la operación *safeMint* del Smart Contract *CarTokenNFT* con la billetera digital correspondiente (`EthereumWallet`)

```
const CreateCarNFT: PayloadResponseParser = {
    parse: (inputs: any[], response: string | undefined): PayloadResponse => {
        const [method, CarID, EthereumWallet] = inputs;
        // const responseParsed: any = parseResponse(response);
        const carInput = {
            CarID: CarID,
            EthereumWallet: EthereumWallet,
        };
        return {
            source: {
                name: 'CarSellContract',
                method: method,
                inputs: JSON.stringify(carInput),
            },
            response: parseResponse(response),
            target: {
                name: 'CarTokenNFT',
                method: 'safeMint',
                inputs: [EthereumWallet],
            },
        };
    },
};
```

Figura 7.11: Implementación de la interfaz PayloadResponseParsers utilizada para el mapeo del escenario de generación de un CarTokenNFT por la compra de un auto en Fabric utilizando la operación CreateCarNFT

Finalmente, para permitir estos mapeos se utiliza una estructura de mapa. Esto se puede observar en la figura 7.12 la cual sigue el patrón de diseño de estrategia.

```
export const PayloadResponseParsers: Record<string, PayloadResponseParser> = {
    BuyCar: CreateCarToken,
    CreateCarNFT: CreateCarNFT,
};
```

Figura 7.12: Estructura de datos de mapa utilizado para mapear el nombre de una operación realizada en Fabric con una implementación de *PayloadResponseParsers*

7.2.3. Conector Ethereum

El Ethereum Connector recibe todas las transacciones que hayan sido redirigidas por el router que originalmente hayan sido leídas y enviadas por el Fabric Connector. El método que se observa en la figura 7.13 es el encargado de procesar estas transacciones. Como se observa en la imagen, para generar los objetos necesarios para la invocación al Smart Contract se utilizan los *ABI*[1] y las direcciones de los Smart Contract donde se encuentran desplegado. Asimismo, se convierten los datos de entrada (inputs) al formato necesario para dicha invocación. Por todo esto, se es necesario contar con los *ABI* de cada Smart Contract que se desee integrar dentro del proyecto. Estos *ABI* son alojados dentro de la carpeta *ABI*.

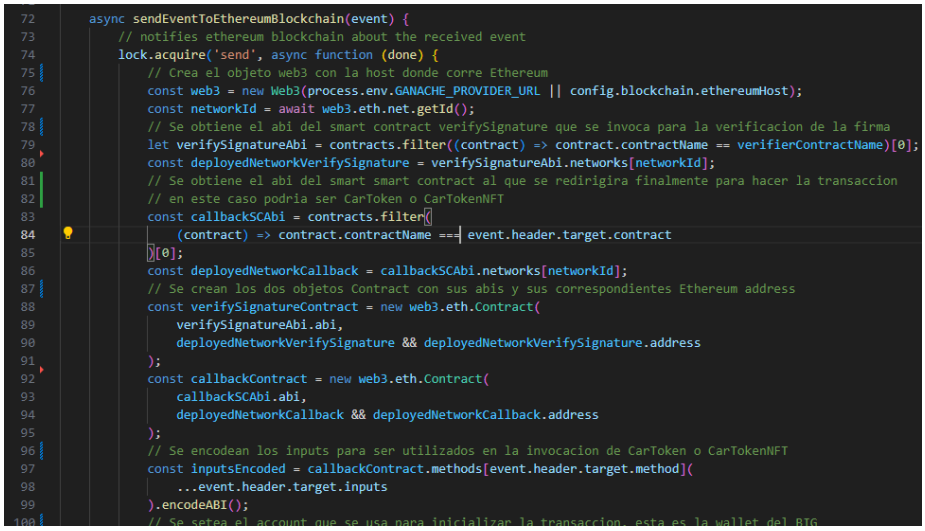


Figura 7.13: Primer fragmento de método que procesa las transacciones en el conector de Ethereum

En la figura 7.14 se observa un segundo fragmento de código del mismo método donde se puede ver algunas características importantes. Entre ellas está la utilización de una billetera digital por parte del conector, dado que este invoca el Smart Contract *VerifySignature*. Otra característica importante se observa en la línea 115, donde dicho método genera la firma sobre toda la información relevante. En la figura 7.15 se muestra el proceso de esta generación de firma utilizando una clave privada con ruta `keys/key.priv`. Esta firma concretamente se genera en formato de dirección de Ethereum[2]. Es decir, que comienza con `0x` y tienen un largo de 20 bytes.

[1] ABI es una especificacion de cómo los datos y las funciones de un contrato inteligente en Ethereum son accesibles y ejecutables

[2] Un ejemplo de direccion de ethereum seria: 0x4b6f5a3dfc911e992c3d8f38c6bb9d1563b5e9a5

```
99              ).encodeABI();
100             // Se setea el account que se usa para inicializar la transaccion, esta es la wallet del BIG
101             const account = web3.eth.accounts.privateKeyToAccount(config.blockchain.bigWalletAddress);
102             web3.eth.defaultAccount = account.address;
103             const callbackData = {
104                 name: event.header.target.contract,
105                 method: event.header.target.method,
106                 inputs: inputsEncoded,
107             };
108             const dataStr = JSON.stringify(event.dataStr);
109             // Se genera el objeto con toda la informacion para firmar.
110             // En la firma se incluye la direccion del smart contract final que se invoca (CarToken o CarTokenNFT)
111             const dataToSign = {
112                 dataStr: dataStr,
113                 ...callbackData,
114                 contractAddress: deployedNetworkCallback.address };
115             // Se genera la firma de toda la informacion de interes para ser validada posteriormente
116             const signature = signMessage(dataToSign);
117             let data_to_send = {
118                 dataStr: dataStr,
119                 signature: signature,
120                 data: callbackData };
121             // Se invoca el Smart Contract VerifySignature con la información de interes mas la firma
122             const result = await verifySignatureContract.methods['callSmartContract'](
123                 deployedNetworkCallback.address,
124                 data_to_send).send({ from: account.address, gas: 3000000 });
125             done();
126         });
```

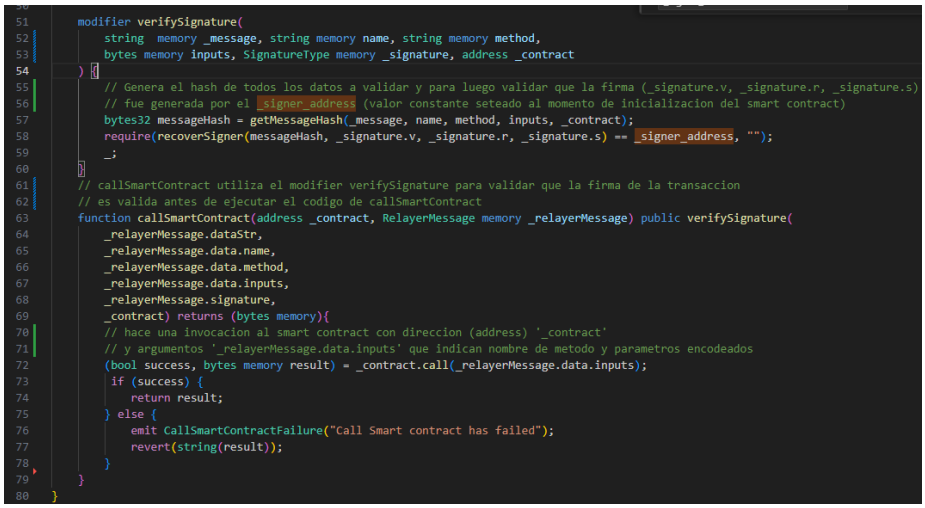
Figura 7.14: Segundo fragmento de método que procesa las transacciones en el conector de Ethereum

```
9   function signMessage(data) {
10      const privateKeyBuffer = Buffer.from(
11          fs.readFileSync(path.resolve(__dirname, 'keys', 'key.priv'), 'utf8'),
12          'hex'
13      );
14      const curve = new elliptic.ec('secp256k1');
15      const privateKey = curve.keyFromPrivate(privateKeyBuffer);
16      const publicKey = curve.keyFromPrivate(privateKeyBuffer).getPublic();
17      const publicKeyHex = '0x' + publicKey.encode('hex', false);
18      console.log('Clave pública:', publicKeyHex);
19      const msgHash = ethers.solidityPackedKeccak256(['string', 'string', 'string','bytes', 'address'],
20          [data.dataStr, data.name, data.method, data.inputs.toString('hex'), data.contractAddress]);
21      const hexStringWithoutPrefix = msgHash.slice(2);
22      const hexBuffer = Buffer.from(hexStringWithoutPrefix, 'hex');
23      const signature = privateKey.sign(hexBuffer);
24      signature.v = signature.recoveryParam + 27;
25      const signatureRes = {
26          r: ethereumjsUtil.bufferToHex(signature.r),
27          s: ethereumjsUtil.bufferToHex(signature.s),
28          v: signature.v,
29      };
30      return signatureRes;
31  }
32
```

Figura 7.15: Fragmento de código que genera una firma con formato de dirección de Ethereum utilizando una clave privada

7.2.4. Implementación de Smart Contract en Ethereum

En la figura 7.16 se observa el Smart Contract *VerifySignature* tal que su función es validar las firmas y redirigir la transacción al Smart Contract que corresponda.



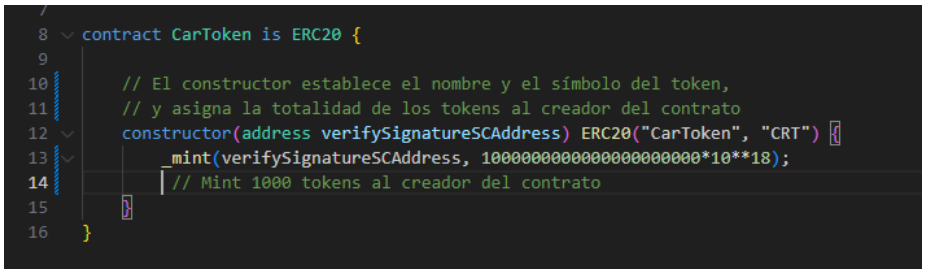
```
    modifier verifySignature(
        string  memory _message, string memory name, string memory method,
        bytes memory inputs, SignatureType memory _signature, address _contract
    ) {
        // Genera el hash de todos los datos a validar y para luego validar que la firma (_signature.v, _signature.r, _signature.s)
        // fue generada por el _signer_address (valor constante seteado al momento de inicializacion del smart contract)
        bytes32 messageHash = getMessageHash(_message, name, method, inputs, _contract);
        require(recoverSigner(messageHash, _signature.v, _signature.r, _signature.s) == _signer_address, "");
        _;
    }
    // callSmartContract utiliza el modifier verifySignature para validar que la firma de la transaccion
    // es valida antes de ejecutar el codigo de callSmartContract
    function callSmartContract(address _contract, RelayerMessage memory _relayerMessage) public verifySignature(
        _relayerMessage.dataStr,
        _relayerMessage.data.name,
        _relayerMessage.data.method,
        _relayerMessage.data.inputs,
        _relayerMessage.signature,
        _contract) returns (bytes memory){
        // hace una invocacion al smart contract con direccion (address) '_contract'
        // y argumentos '_relayerMessage.data.inputs' que indican nombre de metodo y parametros encodeados
        (bool success, bytes memory result) = _contract.call(_relayerMessage.data.inputs);
         if (success) {
            return result;
        } else {
            emit CallSmartContractFailure("Call Smart contract has failed");
            revert(string(result));
        }
    }
}
```

Figura 7.16: Primer fragmento de Smart Contract validador de firmas

El método principal que es invocado por el Ethereum Connector es *callSmartContract*. El objetivo de dicho método es redirigir las invocaciones a los Smart Contract *CarToken* o *CarTokenNFT* dependiendo del caso. Esto solo se hará en el caso de que se cumpla el prerrequisito de que la información fue firmada correctamente por el Ethereum Connector y se valide con la clave publica que el Smart Contract tiene guardada. Esta validación es realizada por el por una operacion tipo *modifier*[3] de Ethereum llamada *verifySignature*, donde a partir del hash de toda la información enviada por el Relayer, la correspondiente firma del Ethereum Connector y su clave pública se valida que efectivamente la clave pública que se tiene corresponde con la firma y la información.

En la figura 7.17 se observa el Smart Contract *CarToken* el que puede ser invocado, mientras que en la figura 7.18 se observa el otro Smart Contract llamado *CarTokenNFT*. En ambos casos el Smart Contract será invocado únicamente si la firma es validada en el Smart Contract *VerifySignature*.

[3]Modifier en solidity es un tipo especial de función que permite cambiar el comportamiento de otras funciones. Se suele utilizar para validar que ciertas condiciones se cumplen previo a que una función se ejecute



```
contract CarToken is ERC20 {

    // El constructor establece el nombre y el símbolo del token,
    // y asigna la totalidad de los tokens al creador del contrato
    constructor(address verifySignatureSCAddress) ERC20("CarToken", "CRT") {
        _mint(verifySignatureSCAddress, 100000000000000000000*10**18);
        // Mint 1000 tokens al creador del contrato
    }
}
```

Figura 7.17: Smart Contract CarToken

El Smart Contract *CarToken* tiene por finalidad generar Tokens para el usuario comprador. Para hacer esto, al inicializar el Smart Contract se le pasa al constructor de dicho Smart Contract la dirección del *VerifySignature*. De esta manera, el *VerifySignature* tendrá una cantidad limitada que podrá transferir los Tokens a los compradores. Por otro lado, el Smart Contract *CarTokenNFT* utiliza una estrategia más directa de minar los NFT a la dirección de la billetera digital del comprador final en vez de realizar una transferencia. La razón de la diferencia entre ambas implementaciones es que el Smart Contract *CarToken* fue el primero en ser implementado y no se contaba con experiencias previas similares. Una vez obtenida una primer experiencia, permitió implementar *CarTokenNFT* utilizando un mecanismo más elegante y directo para el manejo de creación y transferencias de los Tokens.

```
contract CarTokenNFT is ERC721, Ownable {
    uint256 private _nextTokenId;

    constructor() ERC721("CarTokenNFT", "CNFT") Ownable() {
        _nextTokenId = 0;
    }

    function safeMint(address to) public {
        uint256 tokenId = _nextTokenId++;
        _safeMint(to, tokenId);
    }
}
```

Figura 7.18: Smart Contract CarTokenNFT

7.3. Decisiones de implementación y limitaciones

En esta sección se detallan las decisiones de implementación adoptadas y se analizan sus respectivas limitaciones. Se profundiza en el motivo detrás de cada una de estas decisiones y se contrastan con posibles alternativas.

7.3.1. Mapeo de una transacción desde Fabric a Ethereum

Inicialmente, la solución se desarrolló como un monolito y posteriormente se adaptó e integró en la infraestructura del BIG. La división de responsabilidades que se llevó a cabo para migrar a una infraestructura de microservicios resultó en que el mapeo quedara bajo la responsabilidad del Fabric Connector. Esta decisión puede presentar limitaciones al momento de integrar otras blockchains, ya que el mapeo podría requerir una lógica diferente según la blockchain de destino. En otras palabras, el mapeo que actualmente ocurre en Fabric no es reutilizable para otras blockchains distintas de Ethereum.

Otra opción para la separación de responsabilidades habría sido implementar esta lógica en el Ethereum Connector. Sin embargo, esta propuesta también presenta limitaciones. Si existieran otras blockchains destino además de Ethereum, el Ethereum Connector tendría que contener toda la lógica para cada blockchain origen, lo cual no es conveniente.

Una solución para resolver ambas limitaciones sería definir e implementar un modelo canónico de datos tanto en el conector de lectura (origen) como en el conector de escritura (destino). Este modelo tendría un único esquema de datos, permitiendo enviar y recibir un único formato de mensaje para todas las transacciones. Finalmente, el router solo tendría que encargarse de reenviar las transacciones a los conectores correspondientes.

7.3.2. Reestructuración de formato de Mensajería dentro del Gateway

El BIG plantea un formato único para la mensajería que se genera en el Conector de Fabric, pasa por el router y llega finalmente al Conector de Ethereum. Este formato necesitó ser cambiado para soportar el mapeo que ocurre en el Conector de Fabric. Este cambio se vio motivado porque la solución actual no hace uso de eventos, mientras que el BIG es una arquitectura diseñada originalmente para la comunicación entre blockchains haciendo uso de ellos. El formato definitivo se muestra en la figura 7.19. En concreto, se cambia el antiguo campo `operation`, utilizado para el nombre del Smart Contract que se invoca, por `method` y se envían los `inputs` tanto en `source` como en `target`, mientras que en la solución anterior se enviaban en un campo fuera del encabezado llamado `data`. De esta manera, el método incluye los parámetros utilizados para la invocación.

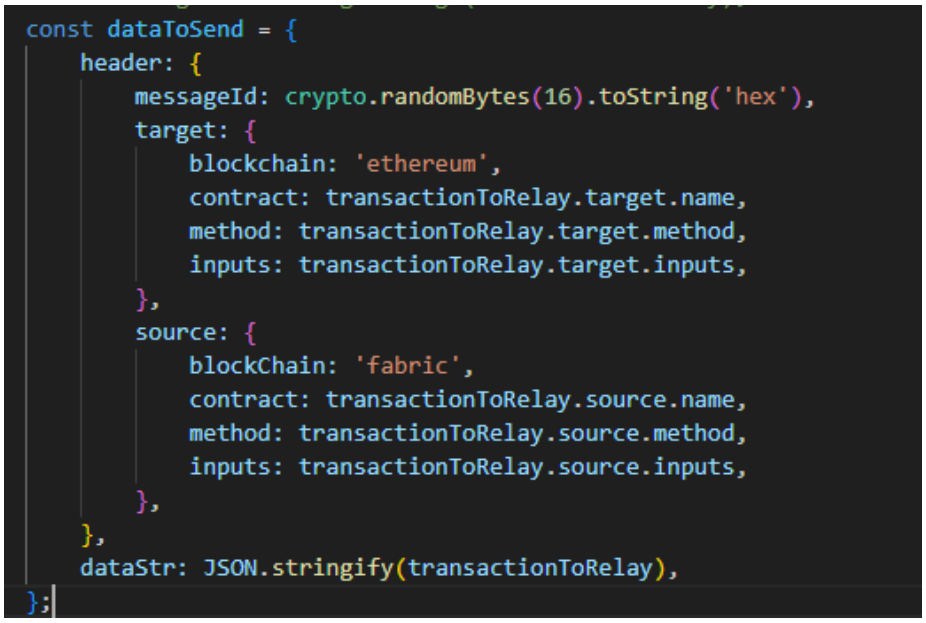


Figura 7.19: Ejemplo de formato de mensajería que envían y reciben los conectores

Capítulo 8

Evaluación de la propuesta

En este capítulo se analizarán pruebas de performance 8.2 y un análisis de costos de la solución propuesta 8.4. Dentro de las pruebas de performance veremos la topología de las pruebas 8.1, la metodología empleada 8.2.1 y diferentes comparativas de usos de la solución 8.3. En cuanto al análisis de costos, interesa conocer el costo de los despliegues de los Smart Contracts (SC) y el costo de la validación de transacciones.

8.1. Topología

En la figura 8.1 podemos ver la topología utilizada para las pruebas. Se llevaron a cabo utilizando dos hosts conectados en la misma red. En el host 1 con el uso de la herramienta K6 se ejecutaron los usuarios virtuales para invocar la API de Fabric mientras que el host 2 contenía toda la infraestructura desplegada junto con las blockchains. Todas las aplicaciones en el host 2 fueron desplegadas en contenedores Docker dentro de una misma red, excepto Ganache[1] que es una aplicación nativa. Cada prueba se monitoreó cuidadosamente para garantizar un rendimiento aceptable en cada máquina anfitriona, lo que aseguró una correcta ejecución del sistema sin cuellos de botella asociados al hardware. Este monitoreo abarcó tanto la memoria RAM como la CPU.

[1] Ganache (*Ganache*, 2024) es una herramienta de desarrollo que simula una blockchain de Ethereum en un entorno local.

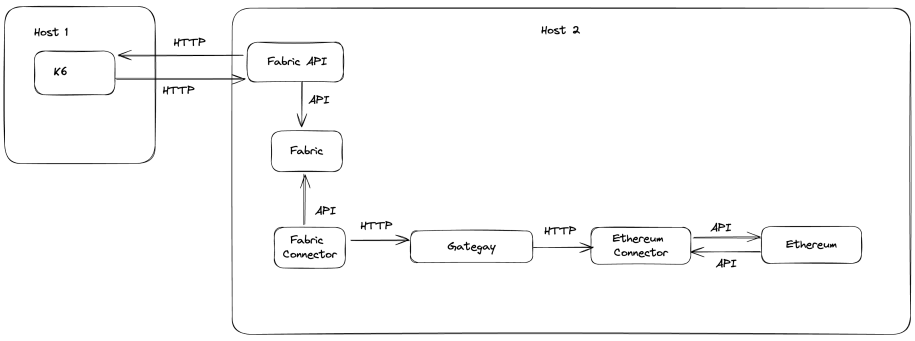


Figura 8.1: Topología de las pruebas

En lo referente al hardware utilizado, en el host 1 se utilizó una computadora Dell Latitude 5420 i7-1185G7 11th Generación y 32 GB de memoria RAM, mientras que en el host 2, se utilizó una MacBook Pro 2021 procesador Apple M1 Pro y 16 GB de memoria RAM.

Para la ejecución y simulación de usuarios virtuales se utilizó K6[2] con la configuración por default de espera entre cada invocación al endpoint. Este tiempo de espera es 1 segundo. En otras palabras, cada usuario virtual, luego de realizar una invocación al endpoint y obtener el resultado, espera 1 segundo para realizar la siguiente invocación.

En lo referente a las versiones de las aplicaciones utilizadas se encuentran Ganache *v2.7.1*, Fabric *v2.5.4* y K6 *v0.50.0*

8.2. Pruebas de Performance

8.2.1. Metodología

Cada prueba se ejecutó tres veces para garantizar la consistencia de los resultados y reducir el margen de error, tomando la mediana de estas ejecuciones para comparar los resultados finales. Este enfoque se adoptó para evitar la influencia de distorsiones al ejecutar una prueba y obtener una muestra distorsionada. Al ejecutar tres pruebas y obtener la mediana se elimina esta distorsión.

Se llevaron a cabo diversas pruebas de rendimiento sobre la solución, que incluyeron los siguientes escenarios:

1. 10 usuarios simultáneos durante 1 minuto.

2. 10 usuarios simultáneos durante 3 minutos.

3. 20 usuarios simultáneos durante 1 minuto.

[2]K6(*K6*, 2024) es una herramienta de código abierto diseñada para realizar pruebas de carga y rendimiento en aplicaciones web.

4. 20 usuarios simultáneos durante 3 minutos.

5. 50 usuarios simultáneos durante 1 minuto con dos configuraciones - 10 transacciones por bloque y 20 transacciones por bloque.

La prueba 5 consiste en cambiar la configuración de Fabric de cantidad de transacciones por bloque. Por defecto, este admite hasta 10 transacciones por bloque.

8.2.2. Medición de tiempos

En la figura 8.2.2 se presentan distintos tiempos de respuesta, los cuales serán útiles para evaluar la solución. A continuación se presentan cada uno de ellos:

1. **Roundtrip:** Este tiempo representa el intervalo entre la invocación desde K6 al Fabric API y la recepción de la respuesta correspondiente.

2. **Tiempo de espera de bloque:** Se refiere al tiempo que el Fabric Connector tarda en detectar la disponibilidad de un nuevo bloque para su procesamiento.

3. **Tiempo entre conectores:** Tiempo transcurrido desde la lectura del bloque en el Fabric Connector hasta que se invoca la transacción en Ethereum. Este tiempo indica el período necesario para realizar el paso de información desde el Fabric Connector hasta finalizar la invocación de la transacción en la blockchain de Ethereum por parte del Ethereum Connector.

4. **Tiempo de firma en Ethereum Connector:** Tiempo de procesamiento desde que se recibe el mensaje en el Ethereum Connector hasta que se realiza la invocación al smart contract. Este tiempo incluye la generación de la firma para ser validada en el Smart Contract de Ethereum, pero no incluye la validación de firma, dado que es un mecanismo asincrónico dentro de la blockchain de Ethereum.

5. **Tiempo de procesamiento de transacción:** Tiempo de procesamiento desde que se envía la solicitud en K6 hasta que se obtiene la respuesta de la invocación al Smart Contract de Ethereum.

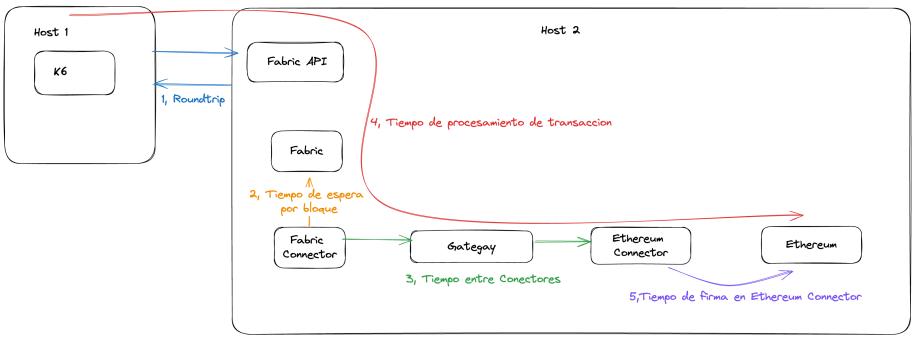


Figura 8.2: Topología con mediciones

La medición que se tomará mayormente como referencia será Tiempo de procesamiento de transacción para conocer el comportamiento de la solución ante una carga importante de procesamiento. Tomar esta medida como referencia permite conocer el tiempo total de procesamiento de una transacción. El resto de las mediciones servirán a modo de análisis para encontrar los cuellos de botella de la solución y/o arquitectura. En cuanto al tiempo de espera de bloque, indica cuanto tiempo el Fabric connector está ocioso en espera de un nuevo bloque a procesar.

8.3. Ejecución de pruebas

En esta sección se describirá el comportamiento de la solución, se compararán las pruebas más interesantes y se sacarán conclusiones acerca de la solución. Para todas las pruebas se monitoreó el uso de memoria y procesador. En las pruebas más exhaustivas, el procesador alcanzó un máximo del 60 %, mientras que el uso de memoria se mantuvo en un 70 %. Con esto aseguramos cierto grado de confiabilidad en las pruebas, dado que no fueron influenciadas por limitaciones de hardware.

Cabe mencionar que los tiempos “Tiempo entre conectores” y “Tiempo de firma en Ethereum connector” no fueron significativos para todos los escenarios, por lo que no serán tomados en cuenta en las comparativas.

8.3.1. 10 usuarios entre 1 minuto y 3 minutos

En las figuras 8.3 y 8.4 se presentan los resultados de las ejecuciones 1 y 2 para los tiempos de espera por bloque y tiempo de procesamiento de transacción. En la figura 8.3 se observa un buen desempeño en ambas ejecuciones, con pocas diferencias significativas. Se observa un pico al inicio que luego se estabiliza haciendo que ambas ejecuciones sean muy similares. En tanto la figura 8.4 se puede notar un comportamiento muy similar donde ambas pruebas terminan

convergiendo a tiempos muy similares. Dado que en ambas comparativas se observa un comportamiento similar de ambas ejecuciones más allá del tiempo, se concluye que la duración de las pruebas no es un factor determinante. Por esto diremos que la solución es estable a lo largo del tiempo.

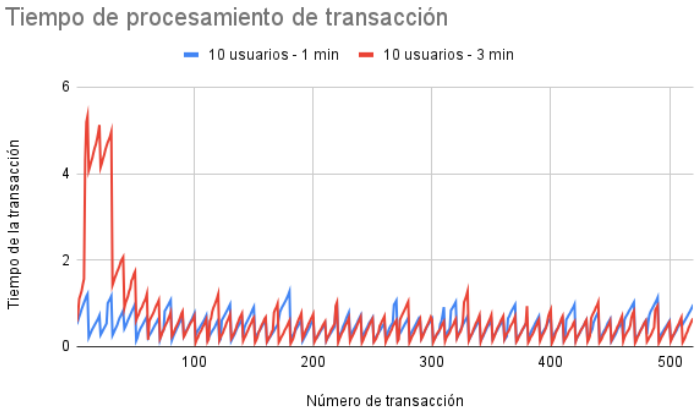


Figura 8.3: Comparativa de tiempos de procesamiento de transacción para 10 usuarios por 1 minuto y 3 minutos.

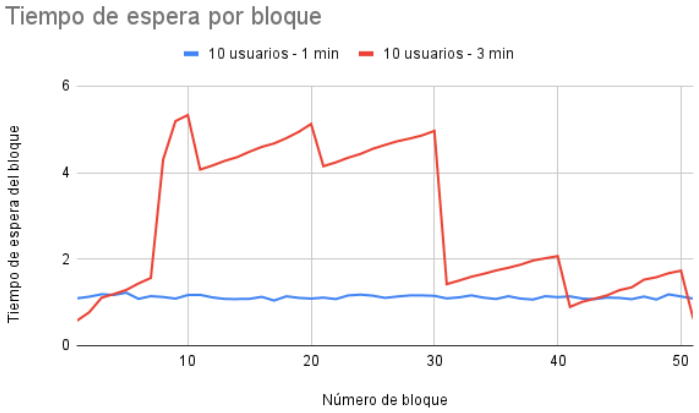


Figura 8.4: Comparativa de tiempos de espera por bloque para 10 usuarios por 1 minuto y 3 minutos.

8.3.2. 20 usuarios entre 1 minuto y 3 minutos

En las figuras 8.5 y 8.6 se presentan los resultados de las ejecuciones 3 y 4 para los tiempos de espera por bloque y tiempo de procesamiento de transacción.

En la figura 8.5 se observa que los tiempos de procesamiento de transacción son linealmente creciente para ambos casos (1 y 3 minutos). Además, observando la figura 8.6 se notan oscilaciones en los tiempos de espera de bloque en intervalos de 0 a hasta 1,50 segundos sin grandes diferencias. Esto implica que existe tiempo ocioso en el Ethereum Connector y en toda la solución en general.

Dado que el tiempo de procesamiento de transacción es linealmente creciente y a más tiempo, más cantidad de transacciones, decimos que la solución no es escalable. Esto significa que cuanto más tiempo el sistema esté sometido a una carga de 20 usuarios, más tardara en terminar con el procesamiento total de todas las transacciones. Por otro lado, al observar que existe tiempo ocioso en la solución, concluimos que existe un cuello de botella presente en la blockchain de Fabric.

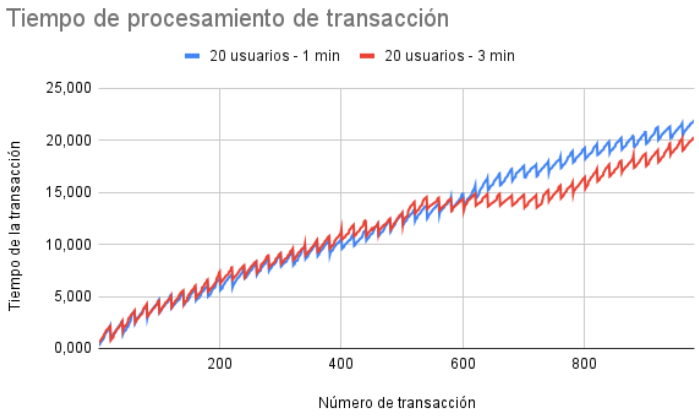


Figura 8.5: Comparativa de tiempos de transacción para 20 usuarios por 1 minuto y 3 minutos.

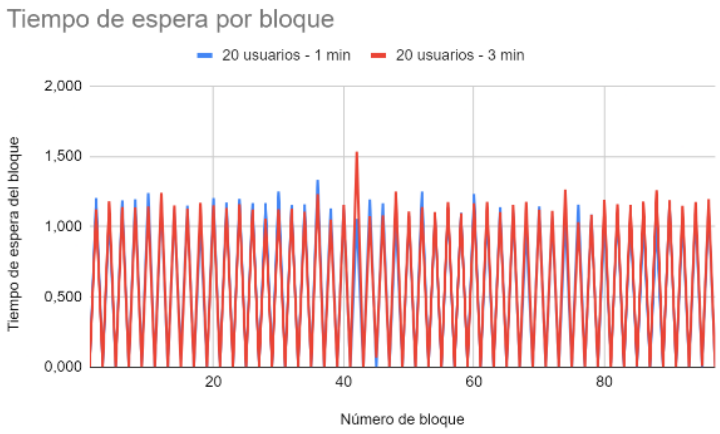


Figura 8.6: Comparativa de tiempos de espera por bloque para 20 usuarios por 1 minuto y 3 minutos.

8.3.3. Análisis general de las ejecuciones de 1 minuto

En las figuras 8.9 y 8.8 se presentan los resultados de las ejecuciones 1, 4 y 5 para los tiempos de espera por bloque y tiempo de procesamiento de transacción, utilizando en todos los casos la configuración por default de cantidad de bloques.

En la Figura 8.7, se observa que a medida que aumenta la cantidad de usuarios, también lo hace el tiempo de procesamiento por transacción, lo que indica una degradación en el rendimiento de la solución. Asimismo, en la Figura 8.8, se evidencia que el escenario con mayor tiempo de espera por un bloque de Fabric fue durante la ejecución con 10 usuarios, lo que refiere al tiempo en que la solución permaneció ociosa. Sin embargo, este escenario fue el que obtuvo el mejor tiempo de procesamiento. Esta discrepancia se debe a que Fabric primero acepta transacciones y luego las incluye en bloques para su almacenamiento, siendo el tiempo entre la aceptación y la disponibilidad de las transacciones en un bloque el factor principal que influye en las mediciones.

Al observar la Figura 8.8, se nota que los tiempos de espera de bloques para 20 y 50 usuarios son similares. No obstante, al analizar la Figura 8.7, se observa que el tiempo de procesamiento de transacciones para 50 usuarios es considerablemente peor que el de 20 usuarios, lo que se atribuye al funcionamiento de Fabric.

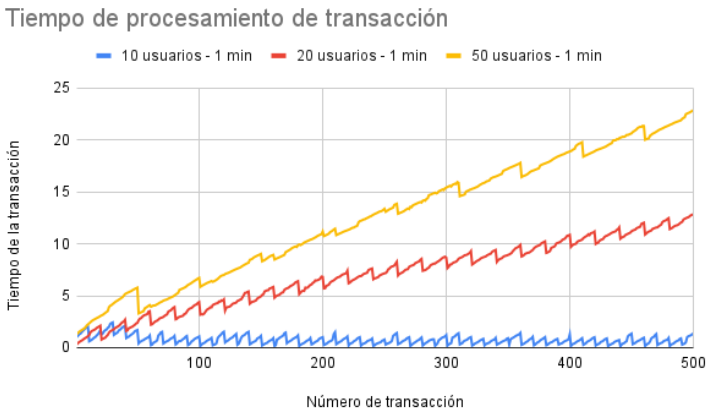


Figura 8.7: Comparativa de tiempos de transacción para 10, 20 y 50 usuarios por 1 minuto.

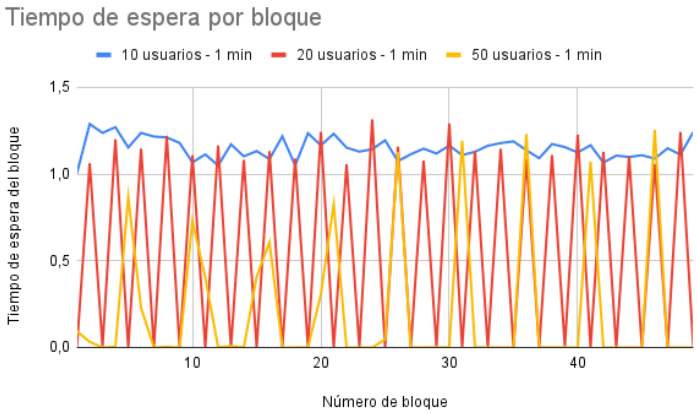


Figura 8.8: Comparativa de tiempos de bloque para 10, 20 y 50 usuarios por 1 minuto.

8.3.4. 50 usuarios 1 minuto con 10 transacciones por bloque y 20 transacciones por bloque.

En la Figura 8.10 se observará el tiempo de transacción de 50 usuarios, cambiando la configuración de cantidad de transacciones por bloque en Fabric. Dado que en las anteriores ejecuciones se detectó que el cuello de botella estaba presente en Fabric, se decidió hacerle cambios con el objetivo de mejorar el tiempo de procesamiento de transacciones.

Como primera observación vemos cambios en los Roundtrip donde para 10 transacciones por bloque este fue en promedio de 0,276 segundos, mientras que para 20 transacciones por bloque fue de 0,694 segundos. Esto implica que los usuarios virtuales realicen menos llamadas al Fabric API para el caso de 20 transacciones por bloque y por ende menor cantidad de transacciones a procesar.

Comparando en la figura 8.9 con 10 transacciones se observa un mejor desempeñó. Por otro lado, en la figura 8.10 con 10 transacciones se nota un menor tiempo ocioso de la aplicación. Por lo tanto, concluimos que la configuración por defecto (10 transacciones por bloque) tiene en general un mejor desempeñó que con 20 transacciones por bloque.

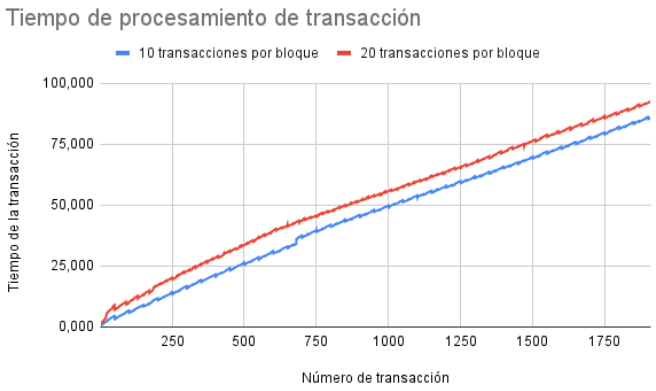


Figura 8.9: Comparativa de tiempos de transacción para 50 usuarios por 1 minuto con 10 transacciones por bloque y 20 transacciones por bloque.

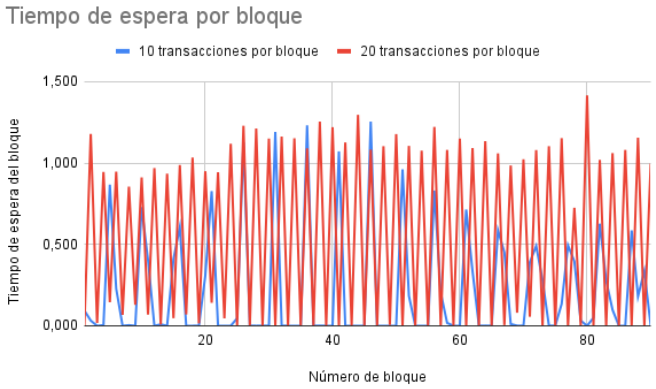


Figura 8.10: Comparativa de tiempos de bloque para 50 usuarios por 1 minuto con 10 transacciones por bloque y 20 transacciones por bloque.

8.3.5. Conclusiones

Los resultados de las pruebas de performance permiten sacar las siguientes conclusiones:

- Cuando se tienen 10 usuarios, la aplicación se comporta de manera estable en cuanto a tiempos de respuesta.
- El principal cuello de botella es Fabric. Esto se debe a que Fabric acepta transacciones y tarda en habilitarlas para su procesamiento, lo que deja al conector de Fabric inactivo mientras espera transacciones para procesar.
- Aumentar el número de transacciones por bloque de 10 a 20, no mejora los tiempos de procesamiento por bloque, debido a que Fabric igualmente demora en habilitar el bloque para ser procesado.
- Aunque los tiempos de demora de procesamiento de transacción son crecientes a medida que aumentan los usuarios, la solución en ningún momento presento algún falló en el procesamiento de alguna transacción. Por lo tanto, para las pruebas realizadas brindan cierto grado de confiabilidad que las transacciones serán completadas con éxito, pero con potenciales demoras.

8.4. Análisis de Costos

En esta sección se analizará los costos que conlleva el despliegue de la solución y el costo de validación de las transacciones. Se identifican dos áreas principales de gastos: aquellos relacionados con el despliegue de Smart Contracts (ver Tabla 8.4) y los costos de gas asociados a cada ejecución de transacciones (ver Tabla 8.4). Se consideraron tres métricas de costos distintas:

1. Costo centrado únicamente en la validación de transacciones(validación de firma).
2. Costo utilizando un contrato inteligente de tipo ERC20 denominado CarToken.
3. Costo empleando un contrato inteligente de tipo ERC721 destinado a la gestión de NFTs, denominado CarTokenNFT.

Dado que el costo del gas es variable, se realizaron 500 transacciones para determinar el valor promedio del gas, considerando que el gas utilizado debería ser constante. Para calcular los costos, se utiliza la fórmula 8.1, donde el gas representa el costo monetario de procesamiento para realizar la transacción, y el costo de gas es el valor en WEI para realizar la transacción, siendo el WEI un costo fijo en Ethers[3], en este caso 10^{-18}.

[3]El valor de Ether utilizado fue obtenido el día 26 de marzo 2024, cotizando a un valor de 3506,96 dolares

$$COSTO = gas \times costo\_de\_gas \text{ WEI} \tag{8.1}$$

SC	Gas	Valor Gas	Costo ETH	Costo Dólares
ValidateSignature	770691	2500000008	0,001926727506165528	10.2
CarToken	1170476	2500000008	0,002926190009363808	6.75
CarTokenNFT	2428948	2500000008	0,006072370019431584	21.2

Tabla 8.1: Costos de deploy de SC

SC	Gas	Valor Gas	Costo ETH	Costo Dólares
ValidateSignature	36865	2500000008	0,001926727506165528	0,32
CarToken	17410	2500000008	0,00004352500014	0.15
CarTokenNFT	73889	2500000008	0,0001847225006	0.62

Tabla 8.2: Costo por Transacción

La tabla 8.4 muestra que el costo asociado a la validación de transacciones es económicamente viable, lo que sugiere que este sistema podría aplicarse no solo en transacciones de alto valor, como las ventas de autos, sino también en operaciones que involucren artículos de menor costo.

Capítulo 9

Gestión del Proyecto

En este capítulo se detalla cómo se gestionó el proyecto. En la sección 9.1, se describe la organización del proyecto tanto con el tutor como con el equipo. En la sección 9.2, se compara la duración estimada con la duración real del proyecto.

9.1. Organización del Proyecto

Al inicio del proyecto, el equipo elaboró una planificación inicial que especificaba el tiempo asignado a cada fase del proyecto, como se detalla en la Tabla 9.1. Esta planificación incluyó un cronograma para cada actividad, estableciendo plazos específicos para su finalización. La planificación fue diseñada para proporcionar una guía clara y estructurada que permitiría al equipo avanzar de manera organizada y eficiente en el desarrollo del proyecto.

Durante el proyecto, se registró el tiempo real dedicado a cada actividad, lo cual se presenta en la Tabla 9.2. Esta comparación entre la planificación inicial y el tiempo real empleado fue fundamental para evaluar la precisión de las estimaciones y ajustar las estrategias según fuera necesario.

El proyecto se organizó en sprints de dos semanas. Al finalizar cada sprint, se acordó con el tutor realizar reuniones para presentar los avances alcanzados durante ese período. Estos entregables incluían documentos o diapositivas que resumían el progreso en los temas asignados. Las reuniones quincenales no solo permitieron monitorear el avance del proyecto, sino que también proporcionaron una oportunidad para recibir retroalimentación valiosa del tutor, abordar cualquier obstáculo encontrado y ajustar el enfoque según fuera necesario.

Además, el equipo se comprometió a trabajar al menos tres veces por semana para desarrollar estos entregables y garantizar el cumplimiento de los objetivos establecidos.

En resumen, la organización del proyecto se basó en una planificación detallada, reuniones periódicas con el tutor y un compromiso firme del equipo para trabajar de manera constante y colaborativa. Esta estructura organizativa

permitió abordar de manera efectiva los desafíos del proyecto y asegurar que se alcanzaran los objetivos propuestos.

Actividades	**Mar**	**Abr**	**May**	**Jun**	**Jul**	**Ago**	**Sep**	**Oct**	**Nov**
Plan de Trabajo	X								
Introducción a Temática	X	X							
Definición de Requerimientos de sistema		X							
Definición de casos de uso			X						
Relevamiento de soluciones existentes		X	X						
Análisis y diseño de solución				X					
Implementación de POC					X	X			
Implementación de Proyecto final						X	X	X	
Implementación de Escenario							X	X	X
Evaluación de la solución final								X	
Test de performance								X	
Documentación							X	X	X

Tabla 9.1: Planificación de Actividades Inicial

Actividades	**Mar**	**Abr**	**May**	**Jun**	**Jul**	**Ago**	**Sep**	**Oct**	**Nov**	**Dic**	**Ene**	**Feb**	**Mar**	**Abr**	**May**	**Jun**	**Jul**	**Ago**
Plan de Trabajo	X																	
Introducción a Temática	X	X																
Definición de Requerimientos de sistema		X																
Definición de casos de uso			X															
Relevamiento de soluciones existentes		X	X	X		X												
Análisis y diseño de solución							X	X										
Implementación de POC									X		X	X						
Implementación de Proyecto final												X	X					
Implementación de Escenario												X						
Evaluación de la solución final													X					
Test de performance												X	X					
Documentación													X	X	X	X	X	X

Tabla 9.2: Planificación de Actividades Final

9.2. Comparación Planificaciones

La figura 9.1 muestra la comparación entre la planificación inicial y la planificación actual del proyecto. La falta de experiencia del equipo en blockchain y en planificación de proyectos llevó a una planificación inicialmente optimista. La tarea de *Relevamiento de soluciones existentes* resultó ser particularmente desafiante. La evaluación de diversas soluciones y la decisión sobre cuál utilizar requirieron más tiempo del previsto, lo que retrasó la etapa de diseño de la solución, duplicando el tiempo originalmente estimado.

Otra desviación significativa fue la documentación. Aunque se había planificado un período de tres meses para su finalización, la documentación tomó más tiempo del esperado debido a la necesidad de ajustes y refinamientos adicionales. Estas desviaciones reflejan la necesidad de adaptar la planificación a la realidad del proyecto y ajustar las expectativas cuando se enfrentan a desafíos imprevistos. Otros desvíos fueron la implementación del POC y los test de performance, donde ambos que llevaron 1 Sprint más de lo planificado. Cabe destacar que el equipo no trabajó durante dos meses debido a vacaciones y exámenes, lo cual también contribuyó a la extensión del proyecto.

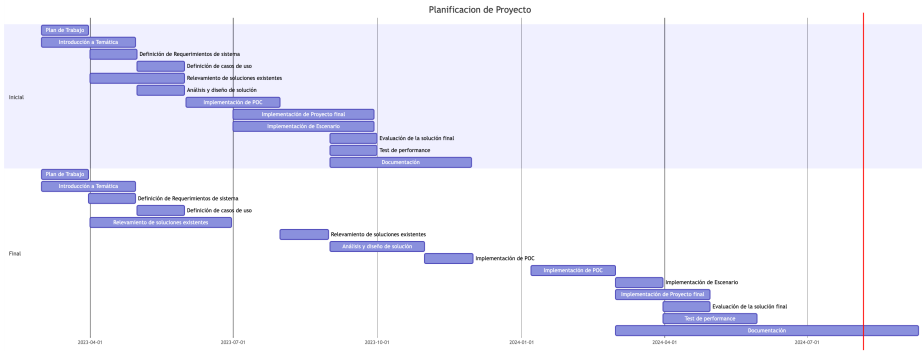


Figura 9.1: Planificación Proyecto

Capítulo 10

Conclusiones y Trabajo Futuro

En este capítulo se presentan las principales conclusiones extraídas del proyecto, así como las propuestas de trabajo futuro que pueden guiar investigaciones y desarrollos posteriores. En la sección 10.1, se exponen los resultados obtenidos, evaluando el cumplimiento de los objetivos planteados y analizando los logros alcanzados en términos de validación de transacciones e interoperabilidad entre Hyperledger Fabric y Ethereum. Por otro lado, en la sección 10.2, se discuten las posibles mejoras y extensiones que podrían implementarse en el sistema desarrollado, con el objetivo de incrementar su robustez, seguridad y adoptabilidad.

10.1. Conclusiones del proyecto

El proyecto logró cumplir con éxito los objetivos planteados. Se desarrolló una solución que permite la validación de transacciones *cross-blockchain* desde Hyperledger Fabric hacia Ethereum, basada en el Gateway desarrollado en un proyecto de grado anterior (Mathías Castro, Emiliano González y Sebastian Pandolfi, 2023). Esta solución se encarga de realizar las lecturas y validaciones de los bloques, y transacciones que son persistidos en Hyperledger Fabric. Posteriormente, esta información es firmada y enviarla a Ethereum, donde un Smart Contract se encarga de validar que la firma generada sea válida. La validación de la solución se realizó a través de un escenario de uso, pruebas de rendimiento y un análisis de costos. El escenario demostró la viabilidad técnica y práctica de la solución, permitiendo validar transacciones *cross-blockchain* en la compra de automóviles. Cada una de estas compras en Hyperledger Fabric, genera como recompensa un token o NFT en Ethereum dependiendo del caso. Las pruebas de rendimiento confirmaron la estabilidad del sistema, ya que no se registraron caídas al someter al sistema a grandes volúmenes de transacciones y usuarios simultáneos. Asimismo, se determinó que la solución no representa el cuello de

botella en la interoperabilidad entre Hyperledger Fabric y Ethereum. El principal cuello de botella fue identificado en la blockchain Hyperledger Fabric, dado que la solución permanecía ociosa hasta que Fabric persistía los nuevos bloques. Finalmente, el análisis de costos concluyó que el costo asociado a la validación de transacciones es económicamente viable, lo que sugiere que este sistema podría aplicarse no solo a transacciones de alto valor, como las compra de automóviles, sino también a operaciones que involucren artículos de menor costo.

Por otro lado, en cuanto a las conclusiones más generales del proyecto, se observó que la mayoría de las soluciones existentes están orientadas a la interoperabilidad de blockchains teniendo por objetivo reducir el consumo de *Gas* o escalar la capacidad de transacciones. Sin embargo, las soluciones para interoperar y validar transacciones entre blockchains heterogéneas son escasas y, en general, carecen de robustez y madurez. Como prueba de esto, muchas de las soluciones son constantemente actualizadas, fusionadas con otras propuestas existentes o, en algunos casos, discontinuadas, provocando incertidumbre en su adopción y uso a largo plazo. Asimismo, la falta de documentación clara en las soluciones estudiadas complica su adopción. En este contexto, se optó por implementar una solución basada en ideas de otras propuestas, con el fin de desarrollar un proyecto funcional que cumpliera con los requisitos planteados. En cuanto a la interoperabilidad de blockchains, observamos que sigue siendo un campo con escasez de estándares y soluciones consolidadas. Esto presenta una gran oportunidad para explorar, proponer e implementar una solución de referencia.

Finalmente, en cuanto al estado del proyecto y la solución, consideramos que para que la solución logre consolidarse y sea atractiva a los potenciales usuarios, se requieren mejoras técnicas. Asimismo, para lograr y facilitar su adopción por parte de los usuarios y organizaciones es necesario contar con una documentación integral y amigable al lector. Este fue un factor muy carente en las soluciones investigadas y consideramos como uno de los aspectos más importantes a la hora de considerar la adopción de una tecnología o solución.

10.2. Trabajo Futuro

En esta sección se describirán las principales mejoras detectadas de la solución diseñada e implementada.

10.2.1. Añadir transacciones validadas de Ethereum a Fabric

Si bien la arquitectura basada en routers y conectores diseñada es extensible y permite la interoperabilidad entre diferentes *blockchains*, en su estado actual la solución solo soporta transacciones desde Hyperledger Fabric hacia Ethereum. Una mejora significativa sería la implementación de la capacidad de realizar transacciones *cross-blockchain* en sentido inverso, es decir, desde Ethereum hacia Hyperledger Fabric. Para lograr esto, sería necesario desarrollar dos nuevos

conectores: uno encargado de la lectura de transacciones en Ethereum y otro que permita su persistencia e invocación de contratos inteligentes correspondientes en Hyperledger Fabric.

10.2.2. Incorporación de Múltiples Validadores

Actualmente, la solución implementada se basa en un único relayer validador, lo que introduce un punto de falla único y una dependencia de confianza considerable. Una mejora clave consiste en la incorporación de múltiples validadores, lo cual incrementaría la seguridad del sistema al distribuir la confianza entre varios nodos. Estos validadores, operando en conjunto, tendrían la responsabilidad de alcanzar un consenso para validar una transacción, lo que fortalecería la robustez y confiabilidad de la solución propuesta.

10.2.3. Descentralización del Gateway

Aunque la solución actual fue implementada con un enfoque centralizado, esta fue diseñada con miras a una futura descentralización, en consonancia con los principios fundamentales de las *blockchains*. Para avanzar en esta dirección, sería necesario realizar modificaciones en el contrato inteligente *VerifySignature* en Ethereum (ver figura 7.16). Dicho contrato podría incluir un *pool* de claves públicas correspondientes a todas las instancias de Gateway en funcionamiento. Cada vez que se reciba una transacción desde uno de estos Gateways, se registraría y se incrementaría un contador. Una vez que dicho contador alcance un umbral predefinido, se ejecutaría la transacción, implementando así un mecanismo de consenso entre los Gateways.

10.2.4. Mejoras en Términos de Seguridad

La seguridad es un aspecto crítico en las *blockchains*, y la solución desarrollada presenta áreas que requieren mejoras significativas en este ámbito. Un área de especial atención es el uso de la función `.call()` en el contrato inteligente *VerifySignature* en Ethereum (ver figura 7.16). En la práctica, se recomienda evitar el uso de este método, ya que puede ejecutar código malicioso, lo que representa un riesgo significativo para la seguridad del sistema (*Recommendations for Smart Contract Security in Solidity*, s.f.).

10.2.5. Implementación de un Modelo Canónico de Comunicación entre Conectores

Como se mencionó en la sección 7.3, una de las limitaciones de la implementación actual es la ausencia de un modelo canónico de comunicación entre conectores. La adopción de un modelo estándar de comunicación mejoraría sustancialmente la adoptabilidad de la solución por parte de los usuarios, permitiendo además que los conectores sean intercambiables y reutilizables como *plugins*.

Referencias

Abebe, E., Behl, D., Govindarajan, C., Hu, Y., Karunamoorthy, D., Novotny, P., ... Vecchiola, C. (2019). Enabling enterprise blockchain interoperability with trusted data transfer (industry track). En *Proceedings of the 20th international middleware conference industrial track* (pp. 29–35).

Ameigenda, A., De Barros, B., y Martinez, S. (2023). Gestión de la privacidad al interoperar blockchain. ".

AWS. (2024). *¿qué es blockchain? — aws.* https://aws.amazon.com/what-is/blockchain/?aws-products-all.sort-by=item.additionalFields.productNameLowercase&aws-products-all.sort-order=asc. (Accessed: 2024-07-03)

Ballamudi, V. (2016, 12). Blockchain as a type of distributed ledger technology. *Asian Journal of Humanity, Art and Literature*, *3*, 127-136. doi: 10.18034/ajhal.v3i2.528

Barghuthi, N., Ncube, C., y Said, H. (2019). *State of art of the effectiveness in adopting blockchain technology - uae survey study.* UAE Survey Study.

Belchior, R., Vasconcelos, A., Guerreiro, S., y Correia, M. (2021, oct). A survey on blockchain interoperability: Past, present, and future trends. *ACM Comput. Surv.*, *54*(8). Descargado de https://doi.org/10.1145/3471140 doi: 10.1145/3471140

Bruno Bradach, Juan Nogueira. (2021). *Interoperabilidad entre plataformas de blockchain.* https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/31368. (Accessed: 2024-04-17)

Buterin, V. (s.f.). *What is ethereum?* https://ethereum.org/en/what-is-ethereum. (Accessed: 2024-07-24)

Facebook, I. (2024). *React - a javascript library for building user interfaces.* Descargado 2024-06-12, de https://reactjs.org/

Foundation, O. (s.f.). *Fast, unopinionated, minimalist web framework for node.js.* https://nodejs.org/. (Accessed: 2024-05-16)

Ganache. (2024). https://archive.trufflesuite.com/ganache. (Last accessed: August 29, 2024)

Hyperledger fabric documentation — introduction. (2024). https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html. (Accessed: 2024-07-29)

Hyperledger fabric documentation — ledger. (2024). https://hyperledger

-fabric.readthedocs.io/en/latest/ledger/ledger.html#blocks. (Accessed: 2024-07-09)

Hyperledger fabric documentation — peers and application. (2024). https://hyperledger-fabric.readthedocs.io/en/latest/peers/peers.html#peers-and-application. (Accessed: 2024-07-09)

Hyperledger fabric documentation — transaction flow. (2024). https://hyperledger-fabric.readthedocs.io/en/latest/txflow.html. (Accessed: 2024-07-29)

Hyperledger fabric documentation — yui. (2024). https://github.com/hyperledger-labs/yui-docs. (Accessed: 2024-06-17)

Ibc — inter-blockchain communication protocol. (2024). https://ibc.cosmos.network/v8/. (Accessed: 2024-08-29)

IBM. (2024). *¿qué es blockchain? — ibm.* https://www.ibm.com/es-es/topics/blockchain#:~:text=%C2%BFQu%C3%A9%20es%20el%20blockchain%3F,activos%20en%20una%20red%20empresarial. (Accessed: 2024-07-03)

A javascript runtime built on chrome's v8 javascript engine. (s.f.). https://nodejs.org/. (Accessed: 2024-05-16)

K6. (2024). https://k6.io/. (Last accessed: August 29, 2024)

Khan, S. N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., y Bani-Hani, A. (2021). Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, *14*, 2901–2925.

Llambias, G., González, L., y Ruggia, R. (2022). Blockchain interoperability: a feature-based classification framework and challenges ahead. *CLEI electronic journal*, *25*(3), 4–1.

Mathías Castro, Emiliano González y Sebastian Pandolfi. (2023). *Interoperabilidad entre plataformas de blockchain.* https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/36720. (Accessed: 2024-04-17)

Metamask. (s.f.). *The crypto wallet for defi, web3 dapps and nfts.* https://metamask.io/. (Accessed: 2024-04-29)

Microsoft. (s.f.). *Extends javascript by adding types to the language.* https://www.typescriptlang.org/. (Accessed: 2024-05-16)

Mohanty, D., Anand, D., Aljahdali, H. M., y Villar, S. G. (2022). Blockchain interoperability: Towards a sustainable payment system. *Sustainability*, *14*(2), 913.

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system.* https://bitcoin.org/bitcoin.pdf. (Last accessed: August 8, 2024)

Nft — ethereum documentation. (2024). https://ethereum.org/es/developers/docs/standards/tokens/erc-721/#introduction. (Last accessed: September 07, 2024)

Optimism foundation. (2024). https://www.optimism.io/. (Accessed: 2024-06-17)

Parrondo, L. (2018). Tecnología blockchain, una nueva era para la empresa. *published in February*.

Pillai, B., Biswas, K., Hóu, Z., y Muthukkumarasamy, V. (2022). Cross-

blockchain technology: integration framework and security assumptions. *IEEE access*, *10*, 41239–41259.

Recommendations for smart contract security in solidity. (s.f.). https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/recommendations/#dont-make-control-flow-assumptions-after-external-calls. (Accessed: 2024-08-01)

Solidity. (s.f.). *A statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum.* https://soliditylang.org/. (Accessed: 2024-05-09)

Tasca, P., y Tessone, C. J. (2017). Taxonomy of blockchain technologies. principles of identification and classification. *arXiv preprint arXiv:1708.04872*.

TokenBridge. (2024). *Token bridge documentation.* https://docs.tokenbridge.net/. (Accessed: 2024-06-17)

Token erc20 — ethereum documentation. (2024). https://ethereum.org/es/developers/docs/standards/tokens/erc-20/. (Last accessed: September 07, 2024)

Wang, G. (2021). "sok: Exploring blockchains interoperability". *Cryptology ePrint Archive*.

Wegner, P. (1996, mar). Interoperability. *ACM Comput. Surv.*, *28*(1), 285–287. Descargado de https://doi.org/10.1145/234313.234424 doi: 10.1145/234313.234424

What is proof-of-stake (pos)? (s.f.). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos. (Accessed: 2024-07-24)

What is proof-of-work (pow)? (s.f.). https://ethereum.org/en/developers/docs/consensus-mechanisms/pow. (Accessed: 2024-07-24)

Anexo A

Glosario

1. **ABI(Application Binary Interface)**: Especifica cómo los datos y las funciones de un contrato inteligente en Ethereum son accesibles y ejecutables. Incluye definiciones de funciones, eventos y tipos de datos, permitiendo la interacción entre contratos inteligentes y otras aplicaciones sin conocer el código fuente subyacente.

   API(Application Programming Interface): Es un conjunto de definiciones y protocolos que permiten a diferentes aplicaciones comunicarse entre sí. Es una interfaz que define cómo deben interactuar los componentes de software, facilitando la integración de funcionalidades entre distintos sistemas

2. **Asset:** Las Assets en criptomonedas son un tipo de token que funciona para dar una base sustentable como respaldo de los "activos" que posee un usuario en el mundo real dentro del mercado global.

3. **Blockchain:** Una cadena de bloques es una base de datos descentralizada y distribuida que almacena registros de transacciones en bloques enlazados y cifrados. Cada bloque contiene un conjunto de transacciones y un hash del bloque anterior, lo que garantiza la integridad y la seguridad de la cadena. La blockchain es la tecnología subyacente detrás de las criptomonedas como Bitcoin y Ethereum.

4. **Bridge:** Un *bridge* es un protocolo que facilita la transferencia de datos, tokens u otros activos entre diferentes redes de blockchain. Su principal función es permitir la interoperabilidad entre blockchains, posibilitando que activos o datos de una blockchain puedan ser utilizados en otra. Esto se logra típicamente mediante mecanismos que bloquean el activo original en la cadena de origen y crean un equivalente en la cadena de destino. Los *bridges* son fundamentales para un ecosistema de blockchain interconectado, permitiendo que las distintas plataformas trabajen juntas de manera más eficiente.

5. **ERC20:** Es un estándar técnico utilizado para crear tokens en la blockchain de Ethereum. Estos tokens siguen una serie de reglas y funciones específicas que les permiten ser intercambiados entre direcciones de Ethereum de manera estandarizada. Los tokens ERC-20 son utilizados principalmente en contratos inteligentes, aplicaciones descentralizadas (DApps) y ofertas iniciales de monedas (ICO).

6. **ERC721:** Es un estándar técnico utilizado para crear tokens no fungibles (NFTs) en la blockchain de Ethereum. A diferencia de los tokens ERC-20, que son intercambiables entre sí, los tokens ERC-721 son únicos e indivisibles, lo que les permite representar activos digitales únicos como obras de arte, bienes virtuales, coleccionables, entre otros. Este estándar define una serie de funciones que permiten la transferencia y el intercambio de tokens no fungibles.

7. **Ether (ETH):** La criptomoneda nativa de la plataforma Ethereum, utilizada tanto como medio de intercambio como para pagar por las operaciones y servicios realizados en la red Ethereum. El Ether es esencial para el funcionamiento de la plataforma, ya que es la moneda con la que se paga el "gas" necesario para ejecutar contratos inteligentes y otras transacciones en la blockchain.

8. **EVM-Compatible:** Se refiere a las blockchains que son compatibles con la Ethereum Virtual Machine (EVM). Estas blockchains pueden ejecutar contratos inteligentes escritos en lenguajes de programación compatibles con Ethereum, como Solidity, y permiten la interoperabilidad con la red principal de Ethereum y otras redes que también soportan la EVM.

9. **Framework:** Un framework o marco de trabajo es un conjunto de directrices, reglas o teorías que sirven de base o punto de partida para un proyecto, sistema u organización. Proporciona una estructura, apoyo y organización para actividades, ideas y decisiones.

10. **Gas:** En Ethereum, el gas se refiere a una unidad de medida que cuantifica la cantidad de trabajo computacional requerido para ejecutar operaciones, como transacciones o contratos inteligentes, en la red. Cada operación en Ethereum consume una cantidad específica de gas, y los usuarios deben pagar una tarifa en ether (ETH) basada en el gas utilizado. Esta tarifa incentiva a los validadores a procesar transacciones y ayuda a prevenir el abuso de los recursos de la red al requerir un costo para la ejecución de operaciones.

11. **Ledger:** Un libro contable es un registro digital o físico de las transacciones asociadas a un sistema financiero. Las redes de cadena de bloques son un tipo de sistema de libro contable descentralizado diseñado para almacenar datos de forma segura.

12. **Modifier:** Un modifier en Ethereum es un tipo especial de función que permite cambiar el comportamiento de otras funciones. En muchos casos

es utilizada para validar que ciertas condiciones se cumplen previo a que una función se ejecute.

13. **NFT (Non-Fungible Token):** Un token no fungible es un tipo de criptomoneda que representa un activo único e indivisible en una blockchain. A diferencia de las criptomonedas tradicionales como Bitcoin o Ethereum, que son intercambiables entre sí y tienen el mismo valor, los NFTs son únicos y no se pueden intercambiar directamente por otro activo de igual valor.

14. **Off-Chain:** Se refiere a las transacciones o procesos que se realizan fuera de la cadena de bloques. Estas operaciones no se registran inmediatamente en la blockchain, lo que permite mayor rapidez y menores costos de transacción. Una vez completadas, pueden ser eventualmente registradas en la blockchain para mantener la integridad y transparencia de los datos.

15. **Permissionless:** Se refiere a un sistema o red en la que cualquier usuario puede participar sin necesidad de una autorización previa. En el contexto de blockchain, significa que cualquier persona puede unirse a la red, validar transacciones y contribuir a la cadena de bloques sin requerir permisos especiales.

16. **Plugin:** Un *plugin* es un pequeño programa complementario que amplía las funciones de aplicaciones web y programas de escritorio. La instalación del plugin implica la ampliación del software con nuevas funciones sin necesidad de retocar el código de todo el programa.

17. **Smart Contract:** Un Smart Contract o también llamado contrato inteligente es un programa informático autoejecutable que ejecuta automáticamente y verifica contratos o acuerdos digitales en una blockchain cuando se cumplen ciertas condiciones predefinidas. Estos contratos están escritos en lenguajes de programación específicos y están diseñados para ser inmutables y transparentes.

18. **Wallet:** Una billetera digital (o wallet, en inglés) en el contexto de las criptomonedas es un software o dispositivo que permite a los usuarios almacenar, enviar y recibir criptomonedas, como Bitcoin o Ethereum.

Anexo B

Anexo Marco Teorico

B.1. Hyperledger Fabric

En esta sección se presentan las principales características y funcionalidades de la blockchain Hyperledger Fabric. En la sección B.1.1, se ofrece una breve introducción a esta tecnología y sus usos. La sección B.1.2 describe la estructura de los bloques en Fabric y sus componentes. A continuación, en la sección B.1.4, se detalla la estructura de una transacción y sus componentes. Finalmente, en la sección B.1.3, se explica a alto nivel el mecanismo de consenso utilizado por Fabric para generar los bloques y las transacciones que finalmente son escritas en la blockchain. Este contexto general permite comprender las etapas por las que pasa una transacción hasta ser validada.

B.1.1. Introducción a Hyperledger Fabric

Entre las aplicaciones públicas más conocidas de blockchains permission less se encuentran Bitcoin y Ethereum. Por otro lado, Hyperledger Fabric es una tecnología de libro mayor distribuido con permisos, de código abierto y orientada a entornos empresariales. Está diseñada para diversas industrias y admite contratos inteligentes en lenguajes de uso general. A diferencia de las redes públicas, Hyperledger Fabric es permissioned. También admite protocolos de consenso conectables, lo que permite la personalización basada en casos de uso y modelos de confianza específicos. No requiere una criptomoneda nativa, lo que reduce el riesgo y los costes operativos. (*Hyperledger Fabric documentation — Introduction*, 2024)

B.1.2. Bloques de Hyperledger Fabric

En la figura B.1 se observa los elementos que componen a un bloque en Hyperledger Fabric. En esta se visualizan el block header (el cual contiene un número de bloque autoincremental, el hash del conjunto de transacciones del

bloque actual, y el hash del block header anterior). Luego se detalla la información del bloque (llamado block data) que almacena el conjunto de transacciones realizadas (En la imagen se visualiza con el encabezado "D2"). Finalmente, se observa en la figura la metadata (también llamado block metadata) con el encabezado M2. Dentro de esta metadata se encuentra información como la firma del creador del bloque (utilizada por los nodos para validar el bloque) así como también una secuencia de booleanos para indicar si la transacción es válida o inválida. Cabe mencionar que la metadata no es utilizada para el cálculo del block hash.

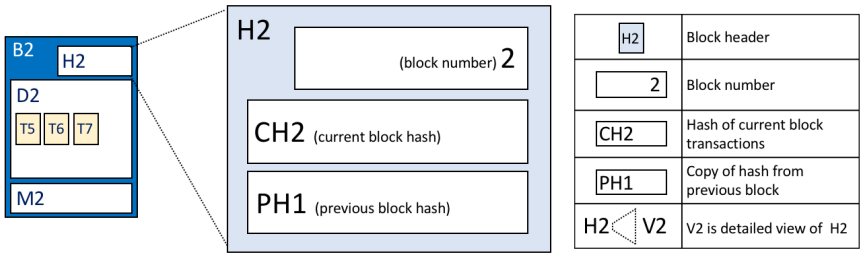


Figura B.1: Ejemplo de los elementos contenidos en un bloque. (*Hyperledger Fabric documentation — Ledger*, 2024)

B.1.3. Mecanismo de consenso en Hyperledger Fabric

En esta sección se describe el proceso que transcurre desde que se invoca un Smart Contract hasta que se valida la transacción y se cambia el estado de la blockchain.

Para comprender este proceso, es necesario introducir previamente dos elementos clave en la infraestructura de Hyperledger Fabric. En primer lugar, los *peers* son nodos responsables de almacenar componentes como la configuración de Fabric, el estado de la blockchain, los contratos inteligentes y el *fabric gateway* —una API para comunicarse con la blockchain y enviar transacciones— entre otros, tienen un rol similar a los mineros o validadores en el concepto de blockchain. Los *peers* aprueban las transacciones y, en caso de validación, las ejecutan en el nodo. En segundo lugar, se encuentran los nodos denominados *orderers*. Estos nodos se encargan de mantener la consistencia de la blockchain, ya que un *peer* por sí solo no puede actualizarla. Es necesario contar con la aprobación de otros *peers* de la red. Los *orderers* y los *peers* trabajan conjuntamente para alcanzar un consenso en las actualizaciones de la blockchain (*Hyperledger Fabric documentation — Peers and Application*, 2024).

B.1.4. Transacciones en Hyperledger Fabric

En la figura B.2 se observa la información almacenada por cada transacción dentro del bloque. Esta información se va recolectando y completando conforme transcurre la ejecución de la transacción y el mecanismo de consenso. Este mecanismo se describe en la sección B.1.3

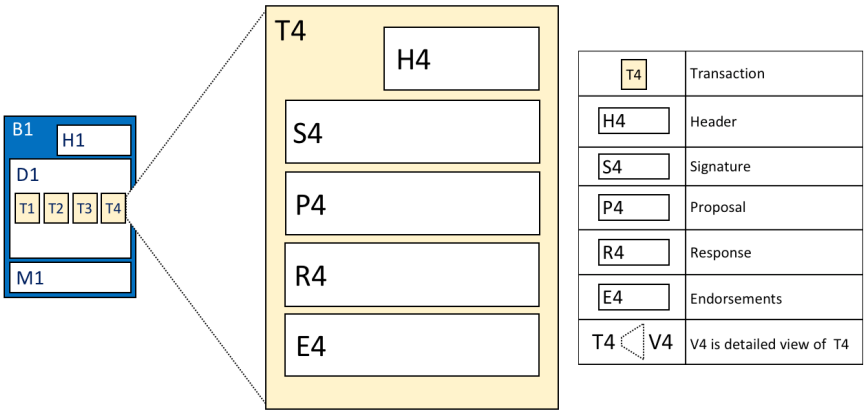


Figura B.2: Ejemplo de los elementos contenidos dentro de una transacción. (*Hyperledger Fabric documentation — Ledger*, 2024)

En la imagen se observan en cada transacción las siguientes secciones:

1. **Encabezado (Header) (H4):** Esta sección almacena información relevante acerca de la transacción.

2. **Firma (S4):** En esta sección se registra la firma del creador de la transacción (la aplicación cliente). Esta firma permite verificar la integridad de la información de la transacción y asegurar que no ha sido manipulada.

3. **Propuesta (P4):** Aquí se almacena toda la información necesaria para la ejecución del Smart Contract, incluyendo los parámetros de entrada codificados y el estado de la blockchain en el cual se ejecutará el Smart Contract.

4. **Respuesta de Propuesta (R4):** En esta sección se guarda el estado de la blockchain después de la ejecución del Smart Contract. Si la transacción es validada, el estado de la blockchain se actualizará de acuerdo con este nuevo estado.

5. **Aprobaciones (E4):** Esta sección contiene todas las firmas de las peers que han validado la transacción.

En el contexto de Hyperledger Fabric, una transacción validada y ejecutada exitosamente alterará el estado de la blockchain, actualizándolo con el nuevo estado indicado en el campo "Respuesta de Propuesta (R4)". Para que una transacción sea validada, debe recibir suficientes aprobaciones de las organizaciones. Esta validación debe cumplir con la política de aprobaciones definida para cada contrato inteligente, que especifica los requisitos necesarios para que la transacción sea considerada válida.

Protocolo de consenso

En la siguiente lista se detallan las etapas que ocurren cuando se realiza una transacción en Hyperledger Fabric (*Hyperledger Fabric documentation — Transaction Flow*, 2024), las cuales se ilustran en la Figura B.3.

1. **Generación de la propuesta:** En esta etapa, el cliente genera y firma la propuesta de la transacción (correspondiente a la lista de información del bloque 2 y 1, respectivamente).

2. **Ejecución de la transacción:** La transacción se ejecuta en el peer seleccionado por el Fabric Gateway Service la cual es una API que se encarga de enviar transacciones en la red, utilizando la información de la propuesta generada previamente para ejecutar el Smart Contract. Esto produce una respuesta de la propuesta (correspondiente a la lista de información del bloque 3). El peer que realiza la ejecución añade su firma de aprobación (correspondiente a la lista de información del bloque 4).

3. **Aprobación de la transacción:** El Fabric Gateway Service, según lo requerido por la política de aprobación, ejecuta la transacción en los peers adicionales necesarios, repitiendo el proceso de "Ejecución de la transacción". Las respuestas de las propuestas obtenidas de estas ejecuciones se incluyen como información dentro de la transacción (correspondiente a la lista de información del bloque 4). Una vez obtenidas todas las aprobaciones requeridas, toda esta información se envía al cliente para una firma adicional.

4. **Firma del cliente:** El cliente firma toda la información recopilada hasta esta etapa y la envía de vuelta al Gateway Service, que luego la reenvía al nodo Orderer.

5. **Ordenar transacciones:** El nodo Orderer valida la firma del cliente y ordena las transacciones recibidas para ser escritas en un bloque. Este bloque se distribuye a los demás peers para su validación y registro en la blockchain.

6. **Escritura en la blockchain de transacciones válidas:** Finalmente, cada peer verifica que el cliente que generó la propuesta coincida con el que realizó la firma en la etapa (correspondiente a la lista de información del bloque 4). Además, se confirma que se han obtenido todas las firmas

necesarias y que se cumple con la política de aprobación. Los peers marcan cada transacción como válida o inválida y actualizan el estado de la blockchain con el resultado de las transacciones válidas.

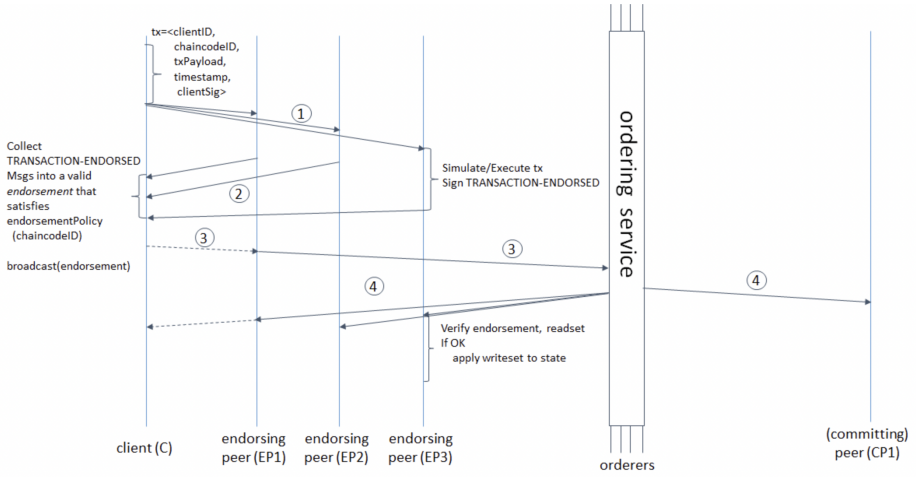


Figura B.3: Flujo de Transacciones

B.2. Ethereum

En esta sección se detallará Ethereum, una de las blockchains utilizadas para el proyecto. En la subsección B.2.1 se hará un breve recuento histórico de la misma. Además se detallará el mecanismo de consenso de Ethereum en la subsección B.2.2, tanto Proof-of-Work (PoW) como Proof-of-Stake (PoS), y se explicará cómo se validan las transacciones.

B.2.1. Introducción

Ethereum(Buterin, s.f.) es una plataforma descentralizada que permite la ejecución de contratos inteligentes y aplicaciones descentralizadas (dApps) a través de su blockchain pública. Fue propuesta en 2013 por Vitalik Buterin y lanzada en 2015. Ethereum extiende el concepto de Bitcoin al ofrecer una máquina virtual completa de Turing, conocida como Ethereum Virtual Machine (EVM), que puede ejecutar scripts y contratos inteligentes.

B.2.2. Sistema de Consenso

El sistema de consenso en Ethereum es fundamental para garantizar la integridad, seguridad y cohesión de la red. Este sistema es responsable de validar y acordar el estado de las transacciones y contratos inteligentes en la blockchain.

Prueba de Trabajo (Proof of Work, PoW)

Ethereum inicialmente utilizó Prueba de Trabajo (PoW) (*What is Proof-of-Work (PoW)?*, s.f.), similar al que emplea Bitcoin. En PoW, los mineros compiten para resolver problemas matemáticos complejos. La resolución de estos problemas requiere un gran poder computacional y tiempo de procesamiento.

El proceso de PoW implica los siguientes pasos:

- **Minería:** Los mineros compiten para resolver un problema criptográfico asociado con el bloque de transacciones. El primer minero en resolver el problema difunde su el bloque a la red.
- **Validación:** Otros nodos en la red verifican la solución propuesta por el minero. Si la solución es correcta, el bloque se añade a la blockchain, y el minero es recompensado con una cantidad de Ether (ETH) y las tarifas de transacción asociadas.
- **Seguridad:** PoW asegura que la red sea resistente a ataques al requerir una gran cantidad de poder computacional para alterar el historial de la blockchain. Para generar bloques maliciosos que, aunque válidos, comprometan la red, un atacante necesitaría controlar más del 51 % de la capacidad de minería total. Esta concentración de poder de cálculo no solo es extremadamente costosa en términos de recursos y energía, sino que el gasto energético podría superar las posibles ganancias obtenidas mediante el ataque. Este alto costo de ejecución actúa como un disuasivo efectivo contra intentos de ataque, asegurando así la seguridad de la red.

Prueba de Participación (Proof of Stake, PoS)

A finales de 2022, Ethereum realizó la transición de Proof-of-Work (PoW) a un nuevo mecanismo de consenso conocido como Proof-of-Stake (PoS)(*What is Proof-of-Stake (PoS)?*, s.f.). Este cambio busca abordar las limitaciones de PoW mejorando la escalabilidad y reduciendo el consumo energético.

En Proof-of-Stake, los validadores depositan una cantidad de ETH en un Smart Contract, lo que demuestra su compromiso y responsabilidad con la red. A cambio, los validadores tienen la tarea de verificar que los nuevos bloques propagados en la red sean válidos y, ocasionalmente, crear y propagar ellos mismos nuevos bloques. Si un validador actúa de manera deshonesta, como por ejemplo, proponer múltiples bloques cuando solo debería enviar uno, parte o la totalidad de su ETH apostada puede ser destruida como parte de una penalizacion.A continuación, se detalla el proceso de ejecución de una transacción en Ethereum bajo el mecanismo Proof-of-Stake:

1. **Creación de la transacción:**
    - Un usuario crea y firma una transacción con su clave privada utilizando una billetera digital o una biblioteca como ether.js, web3js o web3py.

- El usuario especifica la cantidad de gas que está dispuesto a pagar como propina a un validador para que incluya la transacción en un bloque.

2. **Validación inicial:**
    - La transacción se envía a un cliente de ejecución de Ethereum, que verifica su validez asegurándose de que el remitente tenga suficiente ETH y que la transacción esté correctamente firmada.
    - Si es válida, la transacción se agrega a la mempool local (lista de transacciones pendientes) y se difunde a otros nodos en la red.
3. **Propuesta de bloque:**
    - Un nodo validador es seleccionado pseudoaleatoriamente para ser el responsable de construir y distribuir el siguiente bloque a ser agregado a la blockchain y actualizar su estado global.
    - El nodo está compuesto por un cliente de ejecución, un cliente de consenso y un cliente validador.
    - El cliente de ejecución agrupa las transacciones de la mempool local en una "carga útil de ejecución" y las ejecuta localmente para generar un cambio de estado.
    - La carga útil de ejecución se envía al cliente de consenso, que la incluye en un bloque llamado "bloque beacon". Este bloque contendrá dicha información, junto con información sobre recompensas y penalizaciones.
4. **Validación de otros nodos:**
    - Los demás nodos reciben el nuevo bloque beacon en la red de consenso.
    - Los clientes de ejecución de estos nodos re-ejecutan las transacciones localmente para asegurarse de que el cambio de estado propuesto sea válido.
    - El cliente validador atestigua que el bloque es válido y lo añade a la base de datos local si cumple con las reglas de la cadena.
5. **Finalización de la transacción:**
    - La transacción se considera "finalizada" si se convierte en parte de una cadena con un "enlace de supermayoría" entre dos *checkpoints*.
    - Los *checkpoints* son bloques especificos que marcan el inicio de cada época y sirven como puntos de referencia al estado de la blockchain.
    - Una época es un conjunto de bloques que existen entre cada checkpoints, siendo la cantidad de estos bloques siempre la misma.

- Un subconjunto de validadores activos de la red atestiguan la validez de cada bloque dentro de los checkpoints.
- Todos los validadores activos habran tenido la oportunidad de participar en la atestiguación de algún bloque dentro de la epoca.
- Para lograr un "enlace de supermayoría" es necesario la aprobación del 66 % de validadores en los dos checkpoints contiguos. Esto significa que una vez que el 66 % de los validadores han dado fe de la correctitud de ambos checkpoints, los bloques entre ellos, incluidas todas las transacciones, se consideran finalizados y no pueden revertirse.

Este proceso garantiza la seguridad y la integridad de la blockchain de Ethereum mediante el uso eficiente del Proof-of-Stake.

PoS ofrece varias ventajas sobre PoW, incluyendo un menor consumo energético y una mayor capacidad de procesamiento de transacciones, lo que contribuye a una mayor escalabilidad de la red.

Anexo C

Anexo Gestión de proyecto

C.1. Herramientas

Para la adecuada gestión y desarrollo del proyecto, se emplearon diversas herramientas que facilitaron la colaboración, la organización y el seguimiento del progreso. Estas herramientas, en conjunto, proporcionaron una base sólida para la gestión eficiente del proyecto, facilitando la colaboración y asegurando que todos los aspectos del desarrollo y la documentación fueran abordados de manera organizada y profesional. A continuación, se presenta una breve descripción de cada una:

C.1.1. Gitlab

GitLab se utilizó principalmente para la gestión de versiones del código fuente del proyecto. Esta plataforma permitió a los miembros del equipo colaborar de manera eficiente, realizar seguimiento de los cambios en el código y mantener un historial detallado de las modificaciones realizadas

C.1.2. Google Drive

Google Drive es un servicio de almacenamiento en la nube que permite guardar, compartir y colaborar en archivos y documentos. En el proyecto, Google Drive fue esencial para el almacenamiento y la gestión de documentos, facilitando el acceso y la edición colaborativa en tiempo real. Esto permitió que todo el equipo tuviera acceso a la información actualizada y pudiera trabajar de manera sincronizada.

C.1.3. Google Meet

Google Meet es una herramienta de videoconferencia que permite realizar reuniones virtuales. Se utilizó para mantener la comunicación regular entre los miembros del equipo y con el tutor del proyecto. Las reuniones quincenales

permitieron revisar el progreso, discutir problemas y planificar las siguientes etapas del proyecto de manera eficiente.

C.1.4. Overleaf

Overleaf se utilizó para la redacción y formateo de la documentación técnica del proyecto. Esta herramienta fue esencial para la creación de documentos profesionales y bien estructurados, tales como la tesis y los informes de progreso. Overleaf facilitó la colaboración en línea, permitiendo a los miembros del equipo editar y revisar documentos simultáneamente, mejorando la calidad y la claridad de la documentación producida.

C.1.5. Notion

Notion es una herramienta de gestión de proyectos y toma de notas que permite organizar información, tareas y proyectos en un solo lugar. Se utilizó para la planificación y seguimiento de tareas, facilitando la organización del trabajo y la colaboración entre los miembros del equipo. Notion permitió crear y compartir tableros, listas de tareas y notas, mejorando la visibilidad del estado del proyecto y ayudando en la coordinación de las actividades.

C.1.6. LucidApp

LucidApp se utilizó para la creación de diagramas del proyecto. Esta herramienta permitió al equipo diseñar diagramas de diseño, diagrama de componentes y otros gráficos que ayudaron a visualizar las diferentes etapas del proyecto. LucidApp facilitó la comunicación de ideas complejas y mejoró la comprensión entre los miembros del equipo y los interesados.

Anexo D

Materiales complementarios

D.1. Pruebas de performance

En este link se encuentran las gráficas sobre performance
https://docs.google.com/spreadsheets/d/1Z2OQJpxRKGAQziGQZYjFQpNn2iDpva6QvbDp2JsKPYw/edit?usp=sharing

D.2. Soluciones existentes

		Tabla compartativa entre Weaver, Optimism, Token Bridge y YUI		
	Weaver	Optimism	Token Bridge	YUI
Mecanismo de validacion de transacciones	Se apoya en contratos inteligentes y su seguridad está respaldada por los mecanismos de consenso de cada blockchain. Es en general un mecanismo seguro	El mecanismo de validación de transacciones se tiene una fuerte dependencia del mecanismo de consenso de Ethereum, respaldado por la generación de pruebas criptográficas verificables. Aunque los validadores operan fuera de la cadena (off-chain), en caso de disputas, un contrato inteligente es el árbitro que determina el estado de la cadena, lo cual lo hace seguro	En el proceso de validación de transacciones, se recurre a validadores externos (off-chain) que, en cada una de las blockchains involucradas en la interoperabilidad, autorizan las transacciones mediante la observación de los eventos en ambas cadenas interconectadas. Sin embargo, es crucial depositar confianza en estos validadores, lo que lo hace menos seguro.	La validación de transacciones en YUI se fundamenta en light-clients, que son nodos con una menor cantidad de información proveniente de la blockchain opuesta. Estos nodos tienen la responsabilidad de verificar la ejecución exitosa de las operaciones en el lado contrario. El grado de seguridad que proporcionan se adapta a las circunstancias específicas de cada caso, pero en líneas generales, se considera seguro.
Escablabilidad	Para integrar una nueva blockchain se es necesario el desarrollo de un relayer especifico para esta siguiendo cierto estandar para permitir la comunicacion de este relayer con el resto. Arquitectura punto a punto que llevan potencialmente a problemas de escalabilidad	No puede escalar a otra blockchain que no sea ethereum, dado que optimism esta diseñado como una capa 2 de ethereum	Su arquitectura esta definida principalmente para una solucion punto a punto por lo cual agregar otra blockchain a interoperar no seria trivial.	Para integrar una nueva blockchain es necesario agregar un nuevo modulo a cada una de las restantes blockchain. Es decir que para lograr una interoperabilidad entre N blochains entre si, se necesitan N-1 modulos en cada blockchain. Integracion punto a punto que llevan potencialmente a problemas de escalabilidad
Documentacion	Cuenta con ejemplos en su pagina Web sin embargo la documentacion es escasa	Cuenta con ejemplos y relativamente buena documentacion. Sin embargo al ser una solucion comercial se hace difícil encontrar documentos tecnicos de como funcionan las validaciones de transacciones por detras.	Cuenta con ejemplos y relativamente buena documentacion	Cuenta con ejemplos y documentacion escasa. Sin embargo sigue el protocolo IBC el cual esta muy bien documentado y detallado
Arquitectura admite de Interoperar	Esta enfocado a permissioned blockchains. Si bien su arquitectura tendria soporte para blockchains publicas por el momento sus implementaciones son solo sobre tecnologias como Fabic y Corda. Su arquitectura admitiria interoperabilidad con otras blockchains	Es una L2 de Ethereum por lo que es una solucion especifica que solo funciona con dicha blockchain.	Podria ser utlizado en cualquier blockchain EVM compatible. No sigue ni propone un estandar o arquitecura demasiado definido.	Por su Arquitectura basada en IBC, la cual es cual propone una arquitectura en capas con modulos especificos para cada blockchain, soporta toda blockchain capaz de implementar el protocolo IBC.
Adopcion y comunidad	Cuenta con el respaldo de IBM, actualmente cuenta con productos en el mercado financiero, sin embargo no cuenta con una gran comunidad	Solucion comercial ampliamente usada y con una comunidad grande.	No hay gran adopcion ni comunidad. Las implementaciones que existieron de este ya no se encuentran disponibles	Cuenta con el apoyo de datachain una empresa japonesa, y sobre YUI han desarrollado productos para empresas como Toyota. Sin embargo no cuenta con una gran comunidad
Compatibilidad	Compatible con Fabric y Corda. Actualmente se esta trabajando en agregar compatibilidad con Besu	Compatible con Ethereum	Compatible con blockchains que sean EVM	Compatible con Fabric, Corda y Besu.
Activo o Inactivo	Activo	Activo	Inactivo	Activo
Trusted o Trustless	Es trustless, ya que no necesita confiar en niguna entidad externa debido a que las transacciones se validan mediante el mecanismo de consenso de cada blockchian	Es Trustless	En cierta forma es trusted, ya que tenes que confiar en los validadores (off-chain) de que son honestos	Es trustless, ya que no necesita confiar en niguna entidad externa debido a que las transacciones se validan mediante el mecanismo de consenso de cada blockchian
Intercambio de Assets	SI	NO	SI	SI
Tranferencia de Assets	SI	SI	SI	SI

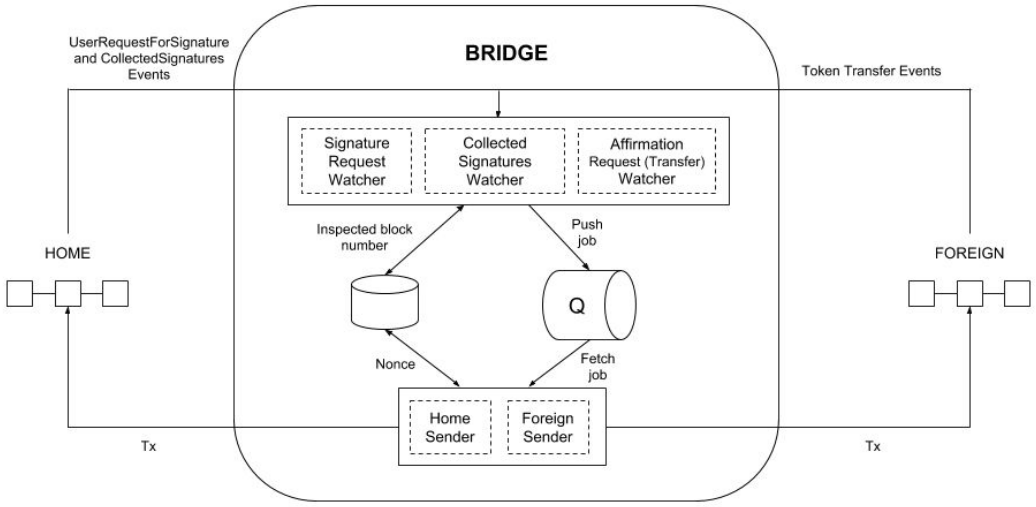
Data sharing	SI	NO	SI	SI
Compatible con BIG	NO, dado que weaver para la interoperabilidad entre dos blockchains hace uso de dos relayers (una por cada blockchain)	NO	SI	SI
Reutilizacion de codigo fuente	Si, ya que provee implementacion de ciertos grados para la implementacion de los relayer y la interoperabilidad con fabric. No hay proyectos aun para la interoperabilidad con Ethereum.	NO	Si, ya que provee templates de implementaciones para el relayer, smart contracts, etc	Si, hay implementaciones y proyectos iniciados tanto para fabric como para blockchain EVM compatibles.
Tipo de Interoperabilidad	Blockchain a blockchain? Creemos que puede llegar a ser Aplication to Blockchain, dado que viendo el flujo de datasharing, la aplicacion es la que termina actualizando el estado de la blockchian A.	Blockchain a blockchain	Blockchain a blockchain	Blockchain a blockchain
DLT/ Blockchain	DLT	Blockchain	Blockchain	DLT - Blockchain

Token bridge

Principales Componentes

- Oráculo: Encargado en escuchar eventos en las diferentes blockchain
- Smart-contracts: Diferentes smart contract para blockear, liberar, minar, o quemar assets. Además smart contracts para tener un seguimiento de la cantidad de validaciones obtenidas.

Oráculo


UserRequestForSignature and CollectedSignatures Events
BRIDGE
Token Transfer Events
Signature Request Watcher
Collected Signatures Watcher
Affirmation Request (Transfer) Watcher
Inspected block number
Push job
HOME
FOREIGN
Q
Nonce
Fetch job
Home Sender
Foreign Sender
Tx
Tx

ERC 20

.

Watchers: Tres componentes que escuchan diferentes eventos lanzados por smart contracts

- Signature Request Watcher: Escucha eventos de tipo “UserRequestForSignature” ejecutados en la network base
- Collected Signatures Watcher: Escucha eventos de tipo “CollectedSignatures” ejecutados en la network base
- Affirmation Request Watcher: Escucha eventos de tipo “Transfer ” emitidos por el token contract

Estos watchers crean eventos y la publican en colas de mensaje tipo rabbitMQ almacenado la información de la transacción junto al hash de transacción. Estos wacher son task que corren cada cierto tiempo y mantienen el registro del último bloque procesado. Además almacenan en una RedisDB estos números de bloques procesados junto a un nonce.

Los senders son los encargados de leer de estas colas de mensaje, extraer toda la información, agregarle el nonce correspondiente para posterior envío a la network de destino

UserRequestForSignature

Emitida por: “HomeBridgeErcToNative.nativeTransfer()” el cual el metodo se encarga de validar que el valor puede ser transferido teniendo en cuenta los costos de fee y validaciones de negocio como cantidad de transacciones diarias, montos u otros.

1- Verifica que el valor de la transacción (msg.value) sea mayor que cero

2- Verifica si el valor de la transacción está dentro de los límites permitidos mediante la función withinLimit(msg.value)

3- Obtiene la cantidad total de tokens mintados por el contrato blockReward llamando a la función mintedTotallyByBridge(address(this)).

4- Obtiene la cantidad total de tokens quemados llamando a la función totalBurntCoins().

5- Verifica que el valor de la transacción no exceda la diferencia entre los tokens mintados y los tokens quemados (msg.value <= totalMinted.sub(totalBurnt))

6- Registra el gasto diario total llamando a la función addTotalSpentPerDay(getCurrentDay(), msg.value)

7- Inicializa la variable valueToTransfer con el valor de la transacción (msg.value).

8- Inicializa la variable valueToBurn con el valor de la transacción (msg.value).

9- En caso de que corresponda, se calcula la comisión mediante la función calculateFee(valueToTransfer, false, feeManager, HOME_FEE)

10- Resta la comisión del valor a transferir mediante la declaración valueToTransfer = valueToTransfer.sub(fee)

11- Calcula la cantidad de tokens a quemar mediante la función getAmountToBurn(valueToBurn)

12- Actualiza el total de tokens quemados mediante la función setTotalBurntCoins(totalBurnt.add(valueToBurn)).

13- Transfiere los tokens quemados a la dirección cero (address(0)) mediante la declaración address(0).transfer(valueToBurn)

14- Emite el evento UserRequestForSignature

```
contract HomeBridgeErcToNative is
    EternalStorage,
    BasicHomeBridge,
    HomeOverdrawManagement,
    RewardableHomeBridgeErcToNative,
    BlockRewardBridge
{
    bytes32 internal constant TOTAL_BURNT_COINS = 0x17f187b2e5d1f8770602b32c1159b85c9600859277fae1eaa99

    function() public payable {
        require(msg.data.length == 0);
        nativeTransfer(msg.sender);
    }

    function nativeTransfer(address _receiver) internal {
        require(msg.value > 0);
        require(withinLimit(msg.value));
        IBlockReward blockReward = blockRewardContract();
        uint256 totalMinted = blockReward.mintedTotallyByBridge(address(this));
        uint256 totalBurnt = totalBurntCoins();
        require(msg.value <= totalMinted.sub(totalBurnt));
        addTotalSpentPerDay(getCurrentDay(), msg.value);
        uint256 valueToTransfer = msg.value;
        address feeManager = feeManagerContract();
        uint256 valueToBurn = msg.value;
        if (feeManager != address(0)) {
            uint256 fee = calculateFee(valueToTransfer, false, feeManager, HOME_FEE);
            valueToTransfer = valueToTransfer.sub(fee);
            valueToBurn = getAmountToBurn(valueToBurn);
        }
        setTotalBurntCoins(totalBurnt.add(valueToBurn));
        address(0).transfer(valueToBurn);
        emit UserRequestForSignature(_receiver, valueToTransfer);
    }
```

CollectedSignatures

Emitido por BasicHomeBridge.submitSignature(bytes signature, bytes message) external onlyValidator

el cual es un smart contract que será invocado por los validators para agregar su firma

1- Verificación de la validez del mensaje: El contrato utiliza la función "isMessageValid" de la estructura o contrato llamado "Message" para verificar que el mensaje sea válido.

2- El contrato verifica que "msg.sender" (el remitente de la transacción (validator)) sea el propietario de la firma "signature" para el mensaje dado

3- Generación de hash: Se generan dos hashes utilizando la función "keccak256". El primer hash, "hashMsg", es el hash del mensaje codificado. El segundo hash, "hashSender", es el hash de la concatenación de "msg.sender" y "hashMsg".

4- Se verifica si el número de mensajes firmados ("signed") es mayor que 1. Si es así, se verifica que no exista un mensaje firmado previamente por el remitente

5- Si no se trata de una firma duplicada, el contrato almacena el mensaje, también marca el hash "hashSender" como firmado utilizando la función "setMessagesSigned"

6- Se incrementa el número de mensajes firmados

7- El contrato emite un evento "SignedForUserRequest" para notificar que el mensaje ha sido firmado por el remitente.

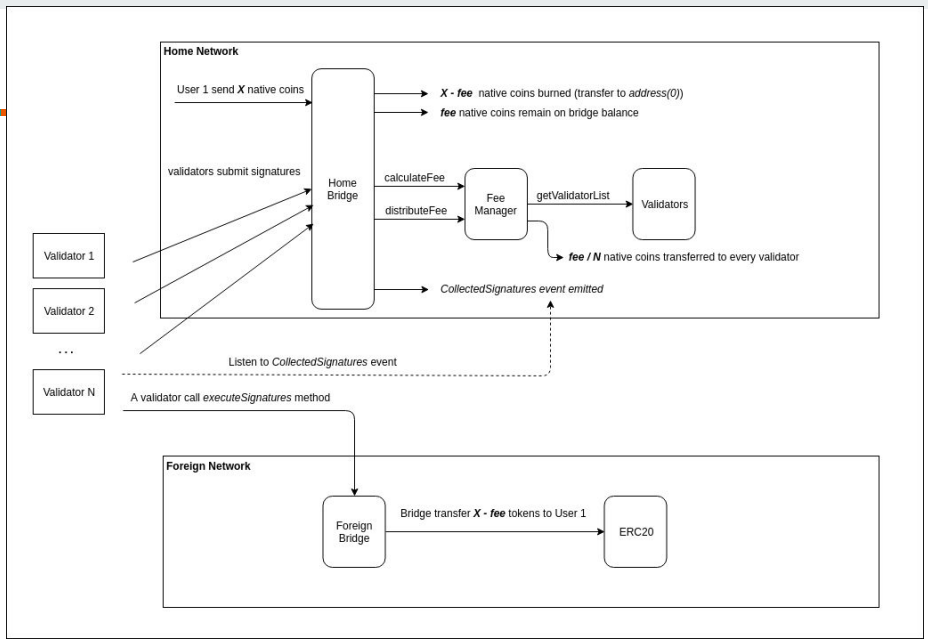
8- El contrato verifica si el número de mensajes firmados ("signed") es mayor o igual al umbral de firmas requerido utilizando la función "requiredSignatures". Si se cumple esta condición, se realiza lo siguiente:

a. Se marca el mensaje como procesado utilizando la función "markAsProcessed".

b. Se emite un evento "CollectedSignatures" para notificar que se han recopilado las firmas requeridas para el mensaje.

c. Se invoca la función "onSignaturesCollected"

```
function submitSignature(bytes signature, bytes message) external onlyValidator {
    // ensure that `signature` is really `message` signed by `msg.sender`
    require(Message.isMessageValid(message));
    require(msg.sender == Message.recoverAddressFromSignedMessage(signature, message, false));
    bytes32 hashMsg = keccak256(abi.encodePacked(message));
    bytes32 hashSender = keccak256(abi.encodePacked(msg.sender, hashMsg));

    uint256 signed = numMessagesSigned(hashMsg);
    require(!isAlreadyProcessed(signed));
    // the check above assumes that the case when the value could be overflew will not happen in the addition operation below
    signed = signed + 1;
    if (signed > 1) {
        // Duplicated signatures
        require(!messagesSigned(hashSender));
    } else {
        setMessages(hashMsg, message);
    }
    setMessagesSigned(hashSender, true);

    bytes32 signIdx = keccak256(abi.encodePacked(hashMsg, (signed - 1)));
    setSignatures(signIdx, signature);

    setNumMessagesSigned(hashMsg, signed);

    emit SignedForUserRequest(msg.sender, hashMsg);

    uint256 reqSigs = requiredSignatures();
    if (signed >= reqSigs) {
        setNumMessagesSigned(hashMsg, markAsProcessed(signed));
        emit CollectedSignatures(msg.sender, hashMsg, reqSigs);

        onSignaturesCollected(message);
    }
}
```


Home Network
User 1 send X native coins
X - fee native coins burned (transfer to address(0))
fee native coins remain on bridge balance
validators submit signatures
Home Bridge
calculateFee
distributeFee
Fee Manager
getValidatorList
Validators
fee / N native coins transferred to every validator
CollectedSignatures event emitted
Validator 1
Validator 2
...
Validator N
Listen to CollectedSignatures event
A validator call executeSignatures method
Foreign Network
Foreign Bridge
Bridge transfer X - fee tokens to User 1
ERC20

ExecuteSignatures

1- Llama a la función estática hasEnoughValidSignatures del contrato Message para validar las firmas proporcionadas en el parámetro signatures. Esta función verifica si las firmas son suficientes y válidas para autorizar la ejecución del mensaje.

2- Verifica que la dirección del contrato (contractAddress) sea la misma que la dirección del contrato actual (address(this)). Esto asegura que el mensaje se está enviando al contrato correcto

3- Verifica si el mensaje ya ha sido relayingMessages(transmitido) anteriormente llamando a la función relayedMessages del contrato actual. Si el mensaje ya ha sido transmitido, la transacción se revertirá. De lo contrario, continúa con el siguiente paso.

4- Llama a la función onExecuteMessage

5- Emite el evento RelayedMessage para indicar que el mensaje ha sido correctamente retransmitido.

```
contract BasicForeignBridge is EternalStorage, Validatable, BasicBridge, BasicTokenBridge, MessageRelay {
    /// triggered when relay of deposit from HomeBridge is complete
    event RelayedMessage(address recipient, uint256 value, bytes32 transactionHash);
    event UserRequestForAffirmation(address recipient, uint256 value);

    /**
    * @dev Validates provided signatures and relays a given message
    * @param message bytes to be relayed
    * @param signatures bytes blob with signatures to be validated
    */
    function executeSignatures(bytes message, bytes signatures) external {
        Message.hasEnoughValidSignatures(message, signatures, validatorContract(), false);

        address recipient;
        uint256 amount;
        bytes32 txHash;
        address contractAddress;
        (recipient, amount, txHash, contractAddress) = Message.parseMessage(message);
        if (withinExecutionLimit(amount)) {
            require(contractAddress == address(this));
            require(!relayedMessages(txHash));
            setRelayedMessages(txHash, true);
            require(onExecuteMessage(recipient, amount, txHash));
            emit RelayedMessage(recipient, amount, txHash);
        } else {
            onFailedMessage(recipient, amount, txHash);
        }
    }
```

Arbitrary-Message

Arbitrary-Message Mode

- Es un metodo soportado por Token Bridge
- Consiste es compartir datos en entre ambas blockchains, los datos podrian interpretarse como invocaciones a smart contracts
- Es importante ver 4 Smart Contracts, los Cuales son MessageDelivery, MessageProccesor, BasicHomeAMB, ForeignHomeAMB

MessageDelivery

- Es un contrato que hereda de los contratos "**BasicAMB**" y "**MessageProcessor**". Proporciona funcionalidades para el envío y procesamiento de mensajes en una red de puentes (bridge) utilizando el estándar AMB (Arbitrary Message Bridge).
- Consiste es compartir datos en entre ambas blockchains, los datos podrian interpretarse como invocaciones a smart contracts
- Es importante ver 4 Smart Contracts, los Cuales son MessageDelivery, MessageProccesor, BasicHomeAMB, ForeignHomeAMB

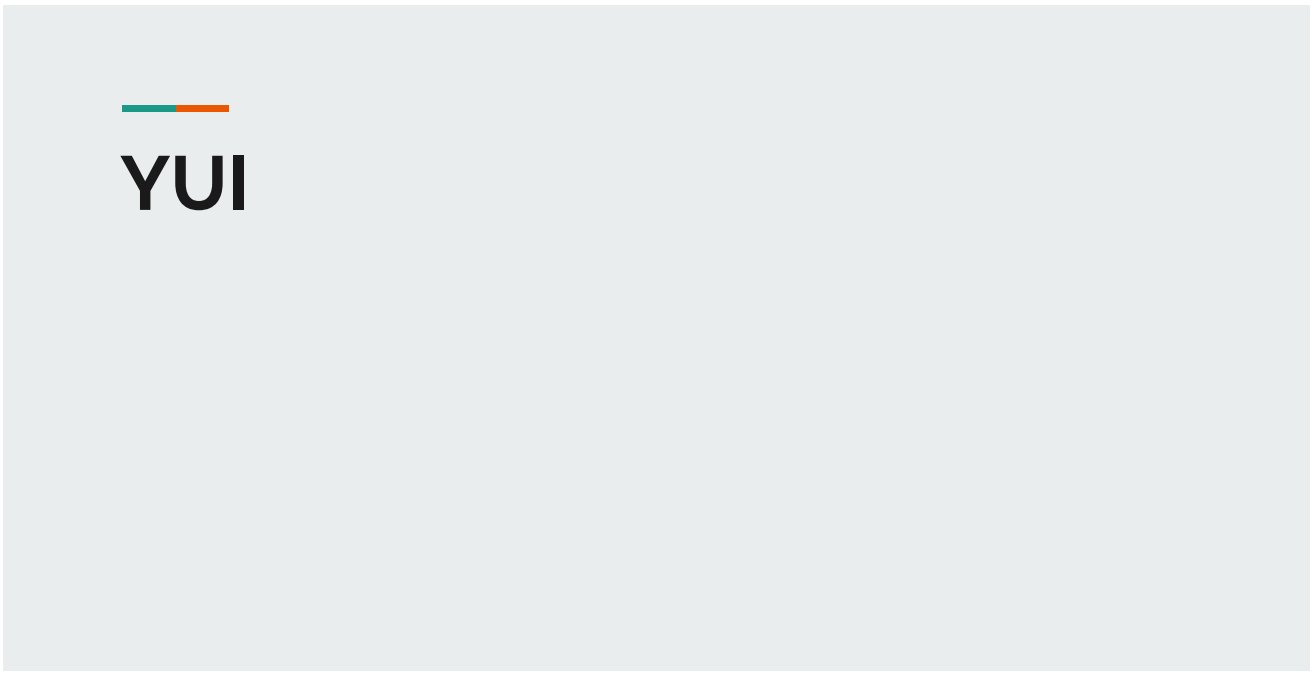
Referencias

https://github.com/omni/tokenbridge-contracts/blob/master/REWARD_MANAGEMENT.md

https://github.com/omni/tokenbridge-contracts/tree/master

https://github.com/omni/tokenbridge

https://github.com/omni/tokenbridge-contracts/blob/master/README.md


YUI

YUI

YUI es un proyecto basado en el IBC protocol propuesto por cosmos. La idea de este protocolo es tener un estándar de interoperabilidad. Para lograr sigue diferentes principios como los de minimalidad, desarrollo en capas, etc.

YUI se basa en el protocolo de Comunicación Inter Blockchain (IBC) del proyecto Cosmos, extendido para soportar diversos proyectos de Hyperledger como Fabric.

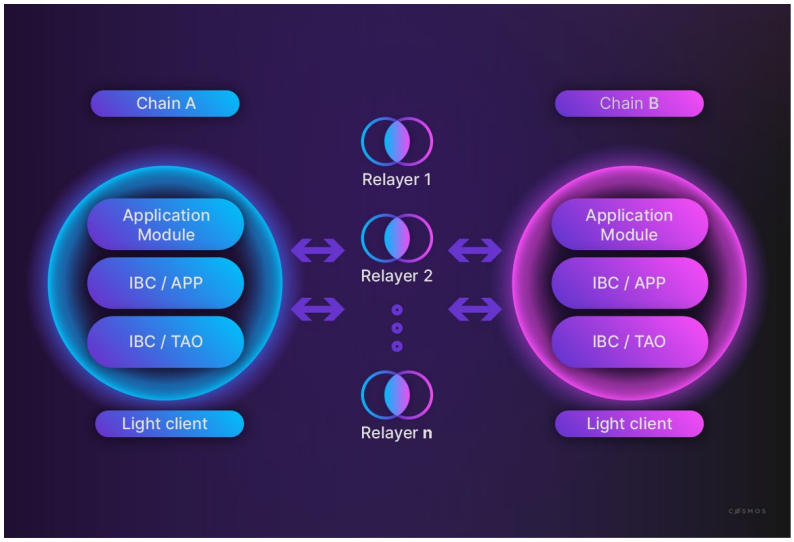
YUI al confiar en la verificación del estado de la chain de la otra chain, se puede lograr una comunicación cross-chain sin preocuparse por lo que hagan actores por fuera de la cadena.

Capas IBC

IBC esta compuesto por 5 componentes esenciales, estos son: light clients, relayers, connections, and channels y la application layer.

- Los **light clients** son nodos livianos de la blockchain, los cuales almacenan los headers de los bloque con los cuales se encargan de validar las transacciones de la otra blockchain.
- Los **Relayers** son componentes off-chain que son permissionless (pueden ser ejecutados por cualquiera) y su función es forwardear los mensajes de una blockchain a otra
- Las **connections** son responsables de conectar los light clients a las diferentes blockchains y son específicos para cada chain
- **Channels** son conductos de comunicación que tienen la finalidad de transferir los paquetes entre módulos.
- La **application layer** utiliza la capa de transporte el enviar y recibir mensajes. Es aqui donde los desarrolladores deben interpretar los datos que son enviados y recibidos entre las diferentes blockchain

IBC


Chain A
Application Module
IBC / APP
IBC / TAO
Light client
Relayer 1
Relayer 2
Relayer n
Chain B
Application Module
IBC / APP
IBC / TAO
Light client

La clave para la validación de transacciones

Los relayers al ser permissionless y off-chain se puede tener múltiples instancias sin importar la cantidad. Esto es indiferente ya que las validaciones son realizadas por los light-client. La forma general de funcionamiento de estos componentes es la siguiente.

- Cada blockchain tiene un light-client utilizado para la validación de las transacciones de la blockchain contraparte. Por ejemplo, esto hace que posible que previo a la realización de una acción en la blockchain B, poder corroborar que un evento X haya ocurrido y haya sido validado en la blockchain A. Ej: una transferencia de un activo.
- Son light porque lo que hacen es mantener menor cantidad de headers de los bloques por cuestiones de consumo de ancho de banda, tiempo, memoria, etc.

Referencias

https://tutorials.cosmos.network/academy/3-ibc/1-what-is-ibc.html

https://informal.systems/2022/05/09/building-with-interchain-security

https://www.hyperledger.org/blog/2021/06/09/meet-yui-one-the-new-hyperledger-labs-projects-taking-on-cross-chain-and-off-chain-operatihttps://tutorials.cosmos.network/academy/3-ibc/5-light-client-dev.htmlons

https://github.com/hyperledger-labs/yui-ibc-solidity

https://github.com/hyperledger-labs/yui-relayer

https://medium.com/tendermint/everything-you-need-to-know-about-the-tendermint-light-client-f80d03856f98

https://labs.hyperledger.org/yui-docs/yui-ibc-solidity/minitoken/overview

https://medium.com/the-interchain-foundation/eli5-what-is-ibc-def44d7b5b4c

https://www.hyperledger.org/blog/2021/06/09/meet-yui-one-the-new-hyperledger-labs-projects-taking-on-cross-chain-and-off-chain-operations

https://github.com/hyperledger-labs/yui-ibc-solidity/blob/main/docs/architecture.md#light-client

https://github.com/tendermint/tendermint/blob/v0.34.x/spec/core/data_structures.md -- Que validaciones se hacen utilizando tendermint light client

Interoperabilidad en blockchain

Optimism y Etherium

Optimism

- Es una plataforma de escalabilidad de blockchain que se basa en el protocolo Ethereum
- Solución que se define como L2 que utiliza Etherium (L1) para mantener garantías, seguridad, disponibilidad y descentralización que provee Etherium.
- Su objetivo es reducir los costos de transacción y mejorar la velocidad de las transacciones en la red Ethereum.
- Para lograr esto, utiliza una solución de escalabilidad llamada Optimistic Rollup, que es una solución de capa dos para Ethereum.

Optimism Rollups

- Utiliza la idea de "optimismo" para mejorar la eficiencia de las transacciones.
- Asume que todas las transacciones son válidas, pero pueden ser validadas en caso de ser necesario
- Una vez que se ha completado la transacción en la cadena secundaria, se crea un resumen de la transacción (conocido como "rollup"), que se envía a la cadena principal de Ethereum, dividiendo los gastos de la transacción aquellos que sean parte del conjunto de transacciones.
- Utiliza sanciones económicas en caso de que se validen transacciones fraudulentas
- El estado de la blockchain de Optimism es almacenado en la cadena principal de etherium como “calldata”

Optimism Bridge

- El bridge de Optimism es una herramienta que permite a los usuarios mover tokens entre la cadena principal de Ethereum y la cadena secundaria de Optimism.
- El bridge funciona mediante el bloqueo de tokens en la cadena principal de Ethereum y la emisión de tokens equivalentes en la cadena secundaria optimista.
- Los tokens bloqueados en la cadena principal se mantienen seguros en un contrato inteligente y solo se liberan una vez que se han emitido los tokens equivalentes en la cadena secundaria.
- Cuando los usuarios desean mover sus tokens de vuelta a la cadena principal, Los tokens de la cadena secundaria se queman y los se liberan los tokens retenidos por el smart contract de la cadena principal

Optimism Bridge

- La forma que esta comunicación ocurre entre ambos es mediante el uso de dos smart contracts “bridges” tanto en Optimism como en Ethereum.
- La comunicación entre Optimism y Ethereum se realiza usando un “Relayer” que escucha sucesos en ambas blockchain y actúa en consecuencia utilizando métodos tradicionales http.
- Una transacción efectuada en L1 implica que se guarde un mensaje en el smart contract que será “escuchado” por el “relayer” y este posteriormente notificará a L2 sobre dicho mensaje.
- En L1 y L2 el smart contract deberá implementar una misma interfaz (***CrossDomainEnabled***) en donde se debe implementar cómo se efectúa la comunicación entre ambas layers.
- Este relayer es llamado “sequencer” y por el momento es trusted y centralizado.
- Trusted fue deshabilitado luego de la actualización de la Optimistic Virtual Machine 2.0
- Por el momento es centralizado, pero se tienen planes de descentralizar en dos etapas. Primero rotando diferentes sequencers pero uno solo corriendo. Posteriormente lograr paralelización con varios sequencers a la vez.

Sequencer

Como dijimos anteriormente, el sequencer es el “relayer” que genera los batches de transacciones en optimsm y las persiste en la canonical chain (smart contract) la cual tiene las transacciones no validadas.

Por otro lado persiste en la state commitment chain (smart contract) baches de estas transacciones, donde cada batch tiene 7 dias para poder ser disputadas por validators .

Por este motivo se dice que el sequencer trusted y centralizado, ya que unicamente corre una sola instancia y ademas existen riesgos de “censura” u otro.

Deposits and Withdraws

Los depósitos de Ethereum a Optimism sin grandes complicaciones ya que no se hacen validaciones, y se realizan luego en L2.

En cuanto a los retiros, se utilizan principalmente dos Smart Contracts, uno en Optimism y otro en Etherium, L2ToL1MessagePasser y OptimismPortal respectivamente.

En optimismPortal se hacen varias validaciones antes de finalizar el retiro de tokens/ethers en ethetreum desde optimism, como por ejemplo que efectivamente se haya querido efectuar un retiro desde L2ToL1MessagePasser o que tenga al menos algun proposal, etc.

En especial estos parametros son creados por el op-node (corriendo en modo validator) que pullea informacion de smart contracts en ethereum y optimism para generar todos los datos que deben ser contrastados en OptimsmPortal.ProveWithdrawalParameters ()

```
interface OptimismPortal {

    event WithdrawalFinalized(bytes32 indexed withdrawalHash, bool success);


    function l2Sender() returns(address) external;

    function proveWithdrawalTransaction(
        Types.WithdrawalTransaction memory _tx,
        uint256 _l2OutputIndex,
        Types.OutputRootProof calldata _outputRootProof,
        bytes[] calldata _withdrawalProof
    ) external;

    function finalizeWithdrawalTransaction(
        Types.WithdrawalTransaction memory _tx
    ) external;
}
```

Referencias

- https://ethereum.org/es/layer-2/
- https://betterprogramming.pub/optimism-smart-contract-breakdown-18f87a7b1823
- https://ethereum.org/en/developers/tutorials/optimism-std-bridge-annotated-code/
- https://ethereum.org/es/developers/docs/scaling/optimistic-rollups/
- https://community.optimism.io/docs/protocol/
- https://github.com/ethereum-optimism/optimism/blob/develop/specs/withdrawals.md
- https://github.com/ethereum-optimism/optimism/blob/develop/op-node/withdrawals/utils.go
- https://community.optimism.io/docs/protocol/protocol-2.0/#
-