UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Interoperabilidad entre plataformas de blockchain

Informe de Proyecto de Grado presentado por

Mathías Castro, Emiliano González y Sebastian Pandolfi

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

Supervisor
Guzmán Llambías

Montevideo, Marzo 2023

Resumen

Cada día las plataformas de blockchain están más afianzadas en el mercado de la informática y sus aplicaciones son cada vez más frecuentes. Los dominios de aplicación son diversos, pasando desde criptomonedas, sistemas de identidad digital, cadenas de suministro hasta juegos. Estos casos de usos suelen implementarse en diferentes plataformas de blockchain y hacen cada vez más necesario establecer mecanismos para que éstas puedan interoperar compartiendo datos y ejecutando comandos entre sí. Sin embargo, algunas características intrínsecas de esas plataformas cómo los mecanismo de consenso, privacidad de los datos y de la identidad de los usuarios, entre otros, presentan algunos desafíos a la hora de lograr la interoperabilidad.

Este proyecto tiene cómo objetivo lograr interoperar entre una plataforma permisionada y otra no permisionada. Las blockchain permisionadas son privadas y en ellas existe una autoridad central que gestiona el ingreso de los participantes a la red (ej. Hyperledger Fabric). Las blockchain no permisionadas son redes públicas dónde cualquier participante que cumpla las reglas puede ingresar (ej: Ethereum).

Para cumplir con los objetivos del proyecto se analizan varios mecanismos de interoperabilidad entre blockchain, profundizando en dos de ellos, un framework llamado Cactus del proyecto Hyperledger y una solución llamada BIG del tipo gateway en formato de *trusted relay*, decidiendo utilizar este último. Este gateway es una solución heredada diseñada para interoperar Hyperledger Fabric con Corda, ambas permisionadas. Se extendió su diseño e implementación agregando algunas mejoras e incorporando la posibilidad de interoperar con la red Ethereum. Cómo resultado se logró el intercambio de datos y la ejecución de comandos. Se implementó un escenario de uso representando la compra de energía eléctrica en forma de *atomic swap.* Un consorcio de estaciones de servicio, implementado en una red Hyperledger Fabric, proporciona la energía a cambio de un pago en criptomonedas de la plataforma Ethereum. Se realizaron pruebas de performance sobre el componente BIG y un análisis del gasto del gas de la plataforma Ethereum, arrojando tiempos de ejecución y costos aceptables.

Se concluye a raíz de este proyecto que la interoperabilidad entre diferentes plataformas de blockchain aún está en etapa de desarrollo. No se destacan mecanismos estandarizados para atacar el problema. Cada caso requiere un análisis de qué tipo de solución utilizar para abordar la interoperabilidad. Dependiendo de los niveles de confianza que se puede otorgar a ciertos componentes, nivel de descentralización de la solución, privacidad de los datos que se intercambien, puede que varíe el tipo de solución a implementar.

Por último, se espera que los aportes de este proyecto resulten útiles y sientan las bases para el desarrollo de nuevas plataformas de interoperabilidad entre redes públicas y privadas.

Palabras claves:
Hyperledger Fabric, Hyperledger Cactus, Ethereum, Blockchain, Interoperabilidad, Atomic Swaps.

Índice general

1. Introducción

Blockchain se ha convertido en una tecnología innovadora, que tiene el potencial de revolucionar varias industrias, luego de la publicación del artículo de Bitcoin en 2008 [1]. Ofrece una plataforma segura, transparente y a prueba de manipulaciones para un sistema orientado a transacciones, por lo que ha sido de especial interés para varias empresas y organizaciones.

Esta tecnología se puede dividir en dos categorías principales: blockchains públicas y blockchains privadas. Las públicas, como Bitcoin y Ethereum, son redes abiertas y no requieren autenticación, en las que cualquiera puede participar y realizar transacciones en la red. Son descentralizadas y las transacciones son validadas y registradas por nodos en todo el mundo. A su vez, son inmutables, una vez que se registra una transacción, la misma no puede ser modificada ni eliminada. Las privadas, como Hyperledger Fabric o Corda, son redes cerradas y autorizadas en las que solo ciertos participantes pueden operar sobre la misma. Existe una autoridad central encargada de gestionar el ingreso de los participantes a la red.

Si se analiza el concepto de interoperabilidad para el caso de sistemas tradicionales, se puede ver que la misma se logra utilizando protocolos y siguiendo estándares de la industria (por ejemplo web services soap). Por el contrario, la interoperabilidad entre blockchains presenta desafíos únicos debido a la naturaleza descentralizada y distribuida. Cada una está diseñada como una red independiente y no existe un protocolo estándar de comunicación entre ellas. Esta heterogeneidad de plataformas de blockchain, sumado a la falta de protocolos o estándares, hace que lograr la interoperabilidad resulte en un gran desafío. Especialmente el de mantener el consenso entre las blockchains participantes. Además al intercambiar información entre los dos tipos de blockchain, es fundamental que se mantenga el anonimato de las redes públicas como también la confidencialidad de los datos de las redes privadas.

La interoperabilidad entre blockchains, y específicamente entre blockchains públicas y privadas, es la motivación principal detrás de este proyecto de grado.

1.1. Objetivos

El objetivo del proyecto es lograr una solución de interoperabilidad entre plataformas de blockchain públicas y privadas. Dicha solución debe ser agnóstica de las plataformas blockchain a interoperar.

Los objetivos específicos que se plantean son los siguientes:

- Avanzar en el estudio, definición e implementación de soluciones de interoperabilidad en Blockchain.
- Analizar la solución existente desarrollada en el marco del proyecto de grado “Blockchain Interoperability Gateway” (BIG) [2].
- Estudiar la plataforma Hyperledger Cactus y compararla con BIG.
- Proveer solución de interoperabilidad entre blockchains públicas y privadas. Enfocado principalmente, en Ethereum y Hyperledger Fabric.
- Implementar transacciones atómicas para garantizar la consistencia de datos entre blockchains.
- Realizar pruebas de performance y evaluar la solución.
- Opcionalmente, plantear una propuesta de diseño para uno de los siguientes desafíos de interoperabilidad: Identidad, privacidad, consenso o autorización.

1.2. Aportes del proyecto

Los principales aportes del proyecto son:

- Relevamiento de las soluciones de interoperabilidad existentes. En particular, una comparativa detallada de Hyperledger Cactus y BIG.
- Propuesta de diseño para interoperar Ethereum con Hyperledger Fabric.
- Implementación de la propuesta de diseño.
- Para validar técnicamente la solución se implementó un escenario de prueba.
- Implementación de transacciones atómicas utilizando Atomic Swaps.
- Evaluación de la solución propuesta mediante la implementación de pruebas de carga y análisis de costos.
- Propuesta de diseño para resolver el desafío de identidad.

1.3. Organización del documento

Este documento se organiza en los capítulos que se describen a continuación. En el capítulo 2 se presentan las definiciones y conceptos necesarios para entender el documento. En el capítulo 3 se analizan los requerimientos del proyecto. En el capítulo 4 se comparan diferentes soluciones de interoperabilidad existentes. En el capítulo 5 se presenta el diseño de la propuesta de solución. En el capítulo 6 se describe la implementación de la solución diseñada en el capítulo anterior. En el capítulo 7 se evalúa la solución a nivel de carga y de uso de gas. En el capítulo 8 se presenta la gestión del proyecto. En el capítulo 9 se presentan las conclusiones como también el trabajo a futuro mencionando posibles mejoras.

2. Marco conceptual

En este capítulo se presentan los principales conceptos utilizados en el transcurso del proyecto. Se describe el proceso realizado para obtener las definiciones utilizadas y las razones que motivan a la especificación de las mismas.

En la sección 2.1 se explica que es blockchain, cómo funciona y en qué escenarios puede ser utilizado y se detalla las diferencias entre blockchains públicas y privadas. En la sección 2.2 se describen las plataformas de blockchain con las cuales se trabajó en este proyecto. En la sección 2.3 se define la interoperabilidad en blockchain, mencionando los desafíos que la misma implica. En la sección 2.4 se presentan las soluciones de interoperabilidad relevadas.

2.1. Blockchain

Repasando definiciones de blockchain en artículos podemos encontrar algunas cómo:
"… un libro mayor digital "append only", cuyos registros se distribuyen entre sus participantes. (...) Los bloques de datos son inmutables. (...) Los flujos de datos y las transacciones son rápidos, en gran parte automatizados, transparentes cuando se requiere y sellados por criptografía cuando no, altamente confiables para todos los participantes(...)" [3].

"... una blockchain es un listado de registros de datos que funciona como ledger digital descentralizado. Los datos se organizan en bloques ordenados cronológicamente y protegidos por criptografía" [4].

Basado en estas definiciones podemos decir que una blockchain es una base de datos, a la que llamamos ledger o libro mayor, distribuida en una red de participantes. A cada operación a grabar en este libro se le llama transacción y son acumuladas en bloques, estos bloques son los que forman la cadena que termina siendo el mismo libro mayor. Es por esto mismo se la denomina Cadena de bloques, o en inglés Blockchain. El libro mayor es inmutable por arquitectura, dado que sus bloques están sellados con criptografía ya que cada bloque es hasheado una vez aprobado y encadenado al resto de bloques. Esto da al libro mayor su característica de inmutabilidad. En la figura 1 vemos gráficamente la composición de la cadena de bloques.

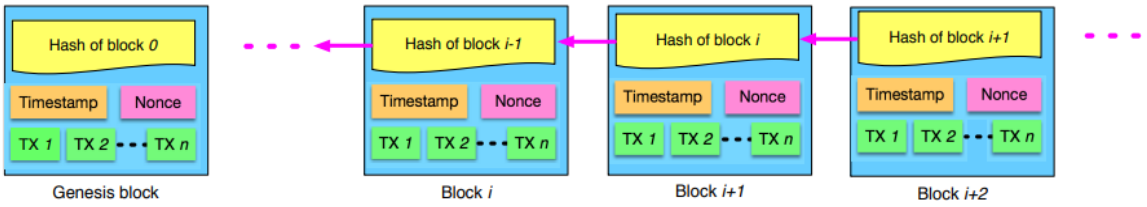

Figura 1: Cadena de bloques, fuente: [5]

Los participantes son los encargados de agregar transiciones a la red, y estos mismos acuerdan los bloques a ser aprobados para ingresar al libro mayor a través de un algoritmo de consenso previamente acordado. De esta manera, la gestión de los datos ingresados en la red queda a cargo de los participantes utilizando los mecanismos brindados por la blockchain. Además, todos los participantes tienen acceso al registro histórico de transacciones. Estas características dan a las plataformas de blockchain otro de sus principios fundamentales, que no existe un organismo central encargado de gestionar los datos, sino que estos son gobernados por los propios participantes. Por esto la particularidad de que las blockchain son descentralizadas.

2.1.1. Consenso

Dado que las blockchains son redes descentralizadas, dónde cada nodo tiene una copia de la información y gestionan los datos de la red, debe existir un mecanismo mediante el cual se acuerde con precisión qué transacciones van a ser aprobadas para el ingreso a la red. Estos mecanismos, estos son los algoritmos de consenso, mediante los cuales los nodos de la red aprueban el ingreso de los bloques a la misma. Para que un cierto bloque de transacciones pueda ingresar al libro mayor, debe ser aprobado mediante uno de estos algoritmos. Además, estos algoritmos, según su diseño proporcionan a la red mecanismos de seguridad para evitar la inserción de transacciones maliciosas. Hay diferentes algoritmos de consenso, entre los cuales se destacan Proof of work (PoW) [1], Proof of stake (PoS) [6] [7], Practical Byzantine Fault Tolerance (PBFT) [8] [5] [9].

Proof of work
Este algoritmo se basa en la resolución de un puzle computacional. La clave en este puzle es que el hash del bloque a proponer deberá tener cierto formato. Deberá comenzar con una cierta cantidad de ceros. Variando un campo del bloque, llamado "nonce" se obtienen diferentes resultados de la función de hash. El puzzle se resuelve cuando el nodo que quiera proponer un bloque encuentra un nonce que junto con el resto del bloque arroje un hash que comience con la cantidad de ceros determinada. Cabe notar qué el mecanismo para la resolución del puzle es de fuerza bruta.

El algoritmo se basa en el esfuerzo computacional que los participantes deben invertir para la búsqueda del hash del bloque. Este esfuerzo es exponencial en el número de ceros requeridos al inicio del hash. Si el mayor esfuerzo computacional reside en los nodos honestos de la red, la cadena con bloques benignos se impondrá sobre cadenas con algún bloque malicioso. Cualquier atacante con intenciones de manipular la red debe hacerse con la mitad más uno de poder computacional de la red. Estos ataques se definen cómo ataques del 50+1.

Proof of stake

El mecanismo funciona con un sistema de apuestas para el ingreso de los nodos al pool de validadores. El nodo que quiera ingresar deberá “apostar” cierta cantidad de sus criptomonedas. De esta forma, en caso de detectar actividad maliciosa de un nodo validador, se le penalizará quitándole criptomonedas de esa apuesta. Además, el nodo validador a agregar el siguiente bloque será escogido teniendo más chances si su apuesta es mayor. Las dos formas más populares de seleccionar pseudo-aleatoriamente al validador dentro del pool de validadores, son la de *hash más bajo* y la de *edad de la apuesta*. Con el método de hash más bajo el bloque se selecciona haciendo un cálculo del hash más bajo ponderado por el tamaño de la apuesta. Según este resultado se elige al nodo que agrega el nuevo bloque. Dado que tanto la apuesta de cada validador cómo el hash del bloque son públicos, con este método el validador escogido puede ser predicho por los participantes de la red. Con el método de la edad de la apuesta, el validador seleccionado para agregar el siguiente bloque, se determina teniendo en cuenta el tiempo que ha pasado desde su última elección, cuanto más tiempo haya pasado desde ésta, mayor es la probabilidad de ser elegido. Además, aquellas apuestas de mayor monto envejecen más rápido de manera de darles prioridad a estas. La apuesta mayor y más envejecida resulta ganadora en la ronda y el participante dueño de la misma es el encargado de agregar el nuevo bloque, reiniciando el ciclo de vida de dicha apuesta.

Practical Byzantine Fault Tolerance (PBFT)

Este algoritmo funciona implementando una máquina de estado para asegurar la correcta distribución de los mensajes en la red. Se seleccionan un conjunto de nodos que seguirán una modalidad de líder y seguidor, dónde uno será el líder y los restantes serán seguidores. El estado de los nodos puede cambiar, pasando un seguidor a condición de líder, por ejemplo por una falla en el nodo líder. Se debe cumplir que no existe más de un tercio de la red de nodos maliciosos para asegurar éxito. Cada ronda, a la que ese le llaman “vistas”, consta de cuatro fases que son las siguientes:

- El cliente envía el request al nodo líder
- El nodo líder envía el request recibido a todos los nodos seguidores
- Los nodos seguidores procesan el request y lo envían cómo respuesta al cliente

- El cliente espera por F + 1 respuestas iguales. Siendo F la cantidad de nodos que se aceptan que pueden ser maliciosos (hasta máximo un tercio de la red cómo se especifica anteriormente).

Se requiere que todas las respuestas en la vista sean deterministas y todas las visitas comienzan en el mismo estado.

El nodo líder se va sorteando en cada ronda con un mecanismo de round robin y los nodos seguidores pueden degradar al nodo líder en caso de contar con un mecanismo de comprobación de si éste está actuando de manera maliciosa.

2.1.2. Tipos de blockchain

La primera red blockchain usable y accesible se remonta a 2009 con la aparición de Bitcoin [1]. Esta fue una red de público acceso dónde cualquier participante que quisiera ingresar podía hacerlo sin necesidad de registrarse ante una autoridad central. Sin embargo, luego de la aparición de otras redes públicas cómo Ethereum[10], surgió la necesidad por parte de ciertos consorcios de empresas de utilizar redes blockchain descentralizadas pero en el marco de un ambiente privado. De esta manera surgieron otro tipo de blockchains a las que se le llamaron “permisionadas” [11].

Blockchains no permisionadas
En este tipo de blockchain las redes son públicas, cualquier participante puede unirse como un nodo e incorporarse a la red sin restricción alguna. Cualquier usuario registrado puede tomar parte en los protocolos de consenso y registrar transacciones en la red. A su vez, se dice que los participantes son pseudo-anónimos ya que no se conoce su identidad dentro de la red, además de que se carece prácticamente de regulaciones que haya que seguir para poder participar. Este tipo de blockchain se centra fuertemente en el concepto de descentralización, no existe ninguna entidad que gobierne la gestión de la red. Alta transparencia, inmutabilidad y libertad de uso son conceptos fuertes en este tipo de blockchains. Los ejemplos más claros de usos son las plataformas de criptomonedas cómo Bitcoin o Ethereum.

Blockchains permisionadas
En este tipo de blockchains las redes son privadas, típicamente gobernadas por una organización o un consorcio de éstas. Los usuarios son miembros de este consorcio, deben identificarse para ingresar a la red y son de confianza para la misma. Las transacciones son confidenciales para cada organización, y por lo general solo visibles para cada una de éstas. El consorcio es quien pone las reglas y las regulaciones para la gestión de las transacciones. Define por ejemplo los participantes del mecanismo de consenso y posibles nodos auditores. Esto

empobrece el concepto de descentralización. Este tipo de blockchains es ideal para organizaciones que deseen colaborar entre sí y retroalimentar sus procesos de negocio en una plataforma altamente regulada, con altos niveles de confidencialidad y de inmutabilidad de datos. Generalmente se establecen protocolos de consenso con alta tasa de eficiencia (en comparación con las blockchain públicas)[11].

2.1.3. Distributed ledger technology

Las Distributed Ledger Technology (DLT)[9] son bases de datos distribuidas de solo agregar en una red peer to peer por tanto descentralizadas. No hay una entidad que gobierne los datos, sino que son los participantes encargados de gestionarlos a través de los algoritmos de consenso.

Las estructuras de datos de una DLT pueden ser variadas, tales cómo grafos cíclicos, grafos acíclicos, listas de cadenas enlazadas o, cómo en la tecnología de blockchains, cadena de bloques inmutables.

Diferencias con Blockchain
Las redes blockchain son una implementación particular de las DLTs cuya estructura de datos forman una cadena de bloques que almacenan transacciones. La información en una blockchain se agrega siempre al final de la cadena y para enlazarlos se utiliza típicamente funciones de hash del área de criptografía[1]. Por otro lado, las redes blockchain poseen un mecanismo para la ejecución de ciertas reglas automáticamente a la hora de procesar transacciones, éste se llama Smarts Contracts [12].

2.1.4. Smarts contracts

El término smart contract se utiliza en blockchain para denominar a unos artefactos utilizados para implementar scripts con instituciones al ejecutar una transacción. Estos contratos pueden implementar algunas órdenes simples al ejecutar las transacciones o incluso puede representar todas las condiciones de un contrato típico en papel. Los contratos se implementan en piezas de código almacenadas en la blockchain. Estos son replicados en la red y heredan las características de estás: seguridad, permanencia e inmutabilidad. Cada vez que se ejecuta una transacción que cumple con los parámetros del contrato, el código se ejecutará con esos parámetros.

Los contratos inteligentes mejoran la performance de la gestión de las transacciones automatizando ciertos procedimientos. Permiten programar estos procesos, potencialmente manuales a nivel de código, eliminando el posible costo de intervención humana a la hora de chequear precondiciones y postcondiciones de los

[1] En este informe se utiliza el término DLTs cómo Blockchains indistintamente.

mismos, y abaratando costos. Además de eliminar la posibilidad de error humano. También permite agregar “inteligencia” a la red, codificando procesos algo más complejos, cómo por ejemplo los “atomics swaps” o transacciones atómicas.

Además, propiciaron el advenimiento de las llamadas “aplicaciones descentralizadas” o DApps. Aplicaciones implementadas en smarts contracts en las blockchain, que dan a estás características peculiares, siendo la principal que son una aplicación descentralizada, es decir que no está alojada en un servidor central, sino distribuida en una red blockchain.

Los smarts contracts son escritos en diferentes lenguajes de programación, por ejemplo, en las redes manejadas en este proyecto, los contratos en la red Ethereum se escriben en Solidity y los contratos en la red Fabric se escriben en javaScript, Java o GoLang [13].

2.1.5. Sharding

Es un método para fragmentar bases de datos distribuidas con el objetivo de escalar la cantidad de datos y operaciones por segundo que puede manejar.
La solución tradicional para la escalabilidad es mejorar las prestaciones del servidor ya sea mejorando su disco duro para más almacenamiento o la memoria RAM y la CPU para rapidez.

Al utilizar sharding se dividen los datos y pedidos entre múltiples servidores, pudiendo agregar más a demanda. Si bien cada servidor puede no tener tantas prestaciones, al dividir la carga en varios puntos se mejora la eficiencia y resiliencia del sistema.
La desventaja que presenta es mayor complejidad en la instalación y mantenimiento de la infraestructura [14].

2.2. Plataformas de blockchain

Las plataformas blockchain brindan un conjunto de herramientas y servicios que los desarrolladores pueden usar para crear e implementar smart contracts, tokens y otras aplicaciones descentralizadas sobre una blockchain [15].

La propia blockchain es una tecnología de registro distribuido que permite que varias partes compartan datos de forma segura y transparente sin necesidad de una autoridad central. Las plataformas Blockchain se basan en esta tecnología al proporcionar funcionalidades adicionales que permiten a los desarrolladores crear e implementar aplicaciones descentralizadas [13]. Algunos ejemplos de plataformas blockchain incluyen Ethereum [10], Binance Smart Chain [16], Solana [17] y Cardano [18].

Mientras la blockchain es la tecnología subyacente que proporciona la seguridad y la transparencia en la aplicación descentralizada, una plataforma de blockchain utiliza esta tecnología para proporcionar un entorno para crear aplicaciones descentralizadas. En otras palabras, una plataforma de blockchain es una abstracción de alto nivel que brinda a los desarrolladores las herramientas y los recursos que necesitan para crear e implementar aplicaciones basadas en blockchain sin tener que comprender la tecnología subyacente en detalle.

En el anexo I se presentan dos posibles casos de uso utilizando plataformas de blockchain. En este proyecto se trabajó con las plataformas de blockchain Hyperledger Fabric y Ethereum las cuales se describen a continuación.

2.2.1 Hyperledger Fabric

Hyperledger Fabric es una plataforma de blockchain permisionada de código abierto desarrollado por el consorcio Hyperledger de la fundación Linux. Está diseñado para ser modular, flexible y escalable, proporcionando un alto grado de privacidad y seguridad para las aplicaciones de nivel empresarial [19].[2]

Para lograr esta privacidad utiliza canales. Los canales en Hyperledger Fabric son una forma de crear subredes privadas dentro de una red más grande. Permiten que diferentes organizaciones realicen transacciones entre sí de forma privada y segura sin exponer sus transacciones al resto de la red. Cada canal tiene su propia cadena de bloques, la cual está separada de la cadena de bloques de los otros canales de la red. Las organizaciones que deseen participar en un canal deben unirse a él, y las transacciones ejecutadas en el canal son privadas y no son visibles para otras organizaciones en la red. El consenso se logra solo entre las organizaciones que forman parte del canal.

Los smart contracts en Fabric se implementan utilizando un modelo de programación llamado chaincode. La chaincode especifica las reglas y condiciones del smart contract y define cómo debe ejecutarse. Éstas permiten a los desarrolladores escribir lógica de negocio en uno de los siguientes lenguajes de programación: Go, java o javascript.

Hay varios tipos de nodos en Hyperledger Fabric, cada uno con una función específica en la red. Estos son los principales tipos de nodos en Fabric:

- Peer: los nodos peer mantienen una copia de la cadena de bloques y ejecutan las chaincode para validar y respaldar transacciones. Se comunican

[2] En el resto del documento se menciona a Hyperledger Fabric y Fabric de forma indistinta.

con otros nodos peer para validar transacciones y lograr un consenso sobre el estado de la cadena de bloques.

- Ordering: los nodos de ordering son responsables de recibir transacciones validadas de los nodos peer y crear bloques que se agregan a la cadena de bloques. Crean un orden de consenso de transacciones y las transmiten a los nodos peer para su validación con la cadena de bloques.

- Anchor: los nodos anchor son un tipo de nodo peer que se utiliza para establecer y mantener la comunicación entre organizaciones que son miembros del mismo canal. Sirven como punto de comunicación para que otros nodos de la red descubran las direcciones de los nodos que pertenecen a otras organizaciones.

- Certificate Authority: los nodos Certificate Authority (CA) son responsables de administrar los certificados digitales utilizados para asegurar la comunicación entre los nodos y los usuarios en la red. Emiten y revocan certificados además de gestionar la lista de revocación de éstos.

- Client: los nodos client se utilizan para interactuar con la red Fabric. Pueden enviar transacciones a la red, consultar la cadena de bloques y recibir notificaciones de eventos. Los nodos cliente no participan en el proceso de consenso.

Fabric permite disparar eventos que se utilizan para notificar a aplicaciones externas sobre cambios en el estado de la blockchain. Los eventos permiten que las aplicaciones externas monitoreen la red Fabric en busca de cambios específicos y tomen medidas cuando ocurran esos cambios. Las aplicaciones externas pueden suscribirse a eventos específicos enviando una solicitud a la red Fabric. La solicitud incluye los criterios para los eventos que la aplicación desea monitorear, como el tipo de evento y la ID del activo. Cuando se cumple una condición específica, la chaincode puede emitir un evento a la red Fabric. El evento puede incluir cualquier dato que la chaincode necesite proporcionar, como la identificación de un nuevo activo que se haya agregado. El evento se registra en la blockchain como una transacción especial. Esto permite que otros nodos de la red vean el evento y tomen medidas si es necesario. Cuando se registra un nuevo evento que coincide con los criterios de suscripción, la red Fabric envía una notificación a la aplicación suscrita. La notificación incluye los detalles del evento, como el ID del activo que se agregó al libro mayor.

Hyperledger Fabric puede utilizar distintos mecanismos de consenso, por ejemplo el basado en el algoritmo Practical Byzantine Fault Tolerance (PBFT), que permite un procesamiento de transacciones rápido, seguro y flexible. El mecanismo de consenso es altamente configurable y admite una variedad de modelos de

implementación, cada uno de estos enfoques tiene como objetivo diferentes requisitos de red y tolerancia a fallas [20].

2.2.2 Ethereum

Ethereum es una plataforma blockchain de código abierto y descentralizada que permite a los desarrolladores crear e implementar aplicaciones descentralizadas y smart contracts. Fue lanzado en 2015 por Vitalik Buterin, y desde entonces se ha convertido en una de las plataformas de blockchain más populares del mundo [21].

En esencia, Ethereum es similar a Bitcoin en que es una red de computadoras entre pares, conocidos como nodos, que trabajan juntos para validar transacciones y mantener una cadena de bloques de saldos de cuentas. Sin embargo, Ethereum va más allá de Bitcoin al permitir que los desarrolladores creen y ejecuten smart contracts.

Ethereum también tiene su propia criptomoneda, Ether (ETH), que se utiliza para pagar las tarifas de transacción y otros servicios en la red Ethereum. Ether es la segunda criptomoneda más grande por capitalización de mercado, después de Bitcoin [22].

En Ethereum, los nodos son la columna vertebral de la red. Son las computadoras o servidores que ejecutan el software Ethereum y participan en el proceso de consenso de la red. Los nodos se comunican entre sí a través de una red peer-to-peer. Comparten información sobre transacciones y bloques, y usan un algoritmo de consenso para acordar el estado de la cadena de bloques. Hay varios tipos de nodos en Ethereum:

- Full: Los nodos full mantienen una copia completa de la cadena de bloques de Ethereum y participan en el proceso de consenso mediante la validación de transacciones y bloques. Son responsables de hacer cumplir las reglas de la red y garantizar la integridad de la cadena de bloques.

- Light: Los nodos light, por otro lado, no mantienen una copia completa de la cadena de bloques, sino que dependen de los nodos full para proporcionarles la información necesaria. Son útiles para dispositivos con almacenamiento o capacidad de procesamiento limitados.

- Archive: Los nodos archive son similares a los nodos full, pero también almacenan todos los datos históricos en la cadena de bloques. Esto permite a los desarrolladores e investigadores acceder a estados anteriores de la red y analizar su comportamiento a lo largo del tiempo.

Los smart contracts en Ethereum deben ser desarrollados en Solidity el cual es Turing-completo, lo que significa que puede usarse para escribir cualquier programa que pueda ser computado por una máquina de Turing [23].

En Ethereum, los eventos se utilizan para notificar aplicaciones externas u otros smart contracts sobre sucesos específicos en la cadena de bloques. Los eventos son esencialmente registros emitidos por un smart contract cuando se ejecuta una función particular o cuando se cumplen ciertas condiciones. Cuando un evento es emitido por un smart contract, se registra en la cadena de bloques de Ethereum y las aplicaciones externas pueden acceder a él utilizando un cliente de Ethereum o un explorador de cadena de bloques. Esto proporciona un registro transparente y auditable de los eventos emitidos.

Si bien PoW fue el mecanismo de consenso que utilizó Ethereum desde su creación, actualmente ya ha realizado la transición a un nuevo algoritmo de consenso llamado Prueba de participación (PoS). PoS tiene como objetivo ser más eficiente energéticamente que PoW al reducir los recursos computacionales necesarios para la minería [6].

2.3. Interoperabilidad en blockchain

La interoperabilidad en blockchain se refiere a la capacidad de diferentes redes de blockchain para comunicarse, compartir datos y ejecutar transacciones de forma consistente. En otras palabras, es la capacidad de diferentes protocolos de blockchain para trabajar juntos con el fin de crear una red más grande que sea más poderosa y eficiente que cada blockchain por sí sola [24].

2.3.1 Desafíos de interoperabilidad en blockchain

Lograr la interoperabilidad en blockchain puede ser un desafío debido a varios factores técnicos y no técnicos. Algunos de los desafíos clave son [25]:

- Incompatibilidad técnica: las redes blockchain utilizan diferentes protocolos, mecanismos de consenso y estructuras de datos, lo que dificulta lograr la interoperabilidad entre ellas.

- Problemas de seguridad: la interoperabilidad entre blockchains puede generar problemas de seguridad, especialmente cuando se transfieren datos o activos valiosos entre diferentes redes. Garantizar que los datos se transfieran de forma segura y con la debida autorización requiere fuertes medidas de seguridad.

- Falta de estandarización: la falta de estandarización en las redes de blockchain, especialmente con respecto al lenguaje y la implementación de

smart contracts, puede plantear desafíos para lograr la interoperabilidad. Esto requiere coordinación y acuerdo sobre estándares para garantizar que las diferentes blockchain puedan comunicarse entre sí.

Sumado a dichos desafíos se agregan los propios que presenta la interoperabilidad entre blockchain permisionada y no permisionada [25]:

- Mecanismos de consenso: las blockchain no permisionadas se basan en mecanismos de consenso probabilístico, como prueba de trabajo (PoW) o prueba de participación (PoS) para validar transacciones y mantener la red. Las blockchain permisionadas, por otro lado, suelen utilizar un mecanismo de consenso determinista. Lograr la interoperabilidad entre estos dos tipos de blockchain requiere una forma de conciliar estos diferentes mecanismos de consenso.

- Autorización: las blockchain permisionadas a menudo tienen controles de acceso, donde solo ciertos nodos o entidades pre aprobadas pueden participar en la red. Las blockchain no permisionadas, por otro lado, están abiertas a todos los participantes, incluidos los actores potencialmente maliciosos. Esto plantea problemas de seguridad al intentar interoperar blockchain permisionadas y no permisionadas

- Privacidad de datos: en las blockchain permisionadas, los participantes suelen tener un mayor control sobre los datos almacenados en la blockchain. Por el contrario, las blockchain no permisionadas son más transparentes y cualquiera puede ver las transacciones en la red. Esto puede crear problemas de privacidad al intentar interoperar entre estos dos tipos de blockchain

- Identidad: las blockchain no permisionadas, permiten que cualquiera participe en la red y use direcciones seudónimas para representar a los usuarios. Mientras que las blockchain permisionadas están diseñadas para usarse dentro de redes cerradas con participantes conocidos y requieren algún tipo de verificación de identidad. Esta diferencia en los sistemas de gestión de identidades puede dificultar la interoperabilidad entre los dos tipos de blockchain. Ya que es necesario garantizar que las identidades de los usuarios se puedan mapear de manera confiable entre los dos sistemas sin exponer la identidad del usuario innecesariamente.

- Gobernanza: las blockchain permisionadas y no permisionadas a menudo tienen diferentes modelos de gobernanza. Las blockchain permisionadas suelen estar controladas por una autoridad central, mientras que las blockchain no permisionadas están más descentralizadas. Esto puede dificultar llegar a un consenso sobre cómo lograr la interoperabilidad entre estos dos tipos de blockchain.

2.3.2 Tipos de operaciones

En el contexto de la interoperabilidad en blockchain, la consulta de información, la ejecución de comandos y la ejecución de comandos atómicos son conceptos importantes que permiten que diferentes blockchain interactúen entre sí.
Consultar información se refiere a la capacidad de recuperar datos de una blockchain para utilizarlos en otra blockchain. Por ejemplo, si un smart contract en una blockchain necesita acceder a los datos almacenados en otra blockchain, puede hacerlo consultando esa blockchain para obtener la información relevante.
La ejecución de comandos implica la capacidad de iniciar acciones en una blockchain desde otra blockchain. Por ejemplo, un smart contract en una blockchain puede necesitar iniciar una transacción en otra blockchain para ejecutar una acción específica.
La ejecución de comandos atómicos hace referencia a ejecutar una serie de comandos como una sola transacción atómica. Esto es importante para lograr la interoperabilidad entre blockchains, ya que permite realizar múltiples acciones de manera coordinada, sin riesgo de inconsistencias o fallas en los datos. Esto garantiza que las transacciones se ejecuten de manera consistente en diferentes blockchain y que cualquier modificación de datos se confirme o revierta como una sola unidad.

2.3.3 Soluciones de interoperabilidad

Se han desarrollado varias soluciones para lograr la interoperabilidad, incluidos notary scheme, trusted relays, sidechains, blockchain de blockchains y protocolos agnósticos. A continuación se describe brevemente cada uno [26].

- Notary scheme: Un notary scheme es un mecanismo que utiliza un tercero de confianza para verificar la validez de las transacciones o mensajes que se intercambian entre diferentes redes de blockchain. El tercero, conocido como notario, firma y valida los mensajes antes de que se transmitan entre diferentes redes.

- Trusted relay: Un trusted relay es un nodo de red que actúa como puente entre diferentes redes de blockchain lo que les permite intercambiar información sin comprometer su seguridad o integridad. El relay de confianza debe ser operado por una entidad de confianza o un consorcio de entidades de confianza para garantizar que no sea vulnerable a ataques o actividades maliciosas.

- Sidechain: Una sidechain es una blockchain separada que está vinculada a una blockchain principal, lo que permite a los usuarios mover activos o datos

entre las dos redes. La sidechain puede tener su propio mecanismo y reglas de consenso, lo que permite a los desarrolladores crear aplicaciones personalizadas que interactúan con la blockchain principal.

- Blockchain de blockchains: Una blockchain de blockchains, es una red que permite que diferentes blockchain se conecten y compartan información. Cada blockchain conserva su propia gobernanza independiente, mecanismo de consenso y reglas, pero puede comunicarse y compartir datos con otras blockchain dentro de la red.

- Protocolos agnósticos: Los protocolos agnósticos están diseñados para que sea indistinta la blockchain en que se ejecute. Estos protocolos generalmente se implementan como smart contracts en diferentes blockchains, lo que les permite comunicarse e intercambiar datos sin la necesidad de un intermediario confiable.

- Hashed Time Lock Contracts (HTLCs) / Atomic Swaps[27]: Es un tipo de smart contract utilizado para la transferencia de activos entre dos actores. El detalle del protocolo se describe en el anexo II.

Los notary scheme y los trusted relay se basan en un tercero de confianza para facilitar la interoperabilidad, mientras que las sidechain, las blockchain de blockchain y los protocolos agnósticos permiten que diferentes redes se comuniquen directamente entre sí.

2.3.4 Orquestador vs coreografía

En el contexto de la interoperabilidad en blockchain, la arquitectura orquestador y coreografía son dos enfoques diferentes para coordinar e integrar diferentes redes de blockchain.
Basándonos en [28] llegamos a la siguiente definición. La arquitectura de orquestador es un enfoque centralizado de la interoperabilidad, en el que una autoridad central o coordinador es responsable de administrar y coordinar las interacciones entre diferentes blockchain. Esta autoridad central se conoce como orquestador y es responsable de crear y hacer cumplir las reglas sobre cómo las diferentes blockchain pueden interactuar entre sí. El orquestador actúa como un intermediario de confianza, verificando y validando las transacciones que se intercambian entre diferentes blockchain.

La coreografía, por otro lado, es un enfoque descentralizado de interoperabilidad, en el que diferentes blockchain interactúan entre sí directamente, sin necesidad de una autoridad central. En un modelo de interoperabilidad basado en la coreografía, cada blockchain tiene su propio conjunto de reglas y protocolos, y estas redes interactúan

entre sí en función de reglas y estándares preestablecidos. La coreografía se puede lograr mediante el uso de smart contracts que permiten que diferentes blockchain se comuniquen e intercambien datos de manera estandarizada, sin la necesidad de un intermediario centralizado.

Ambas arquitecturas tienen sus ventajas y desventajas en el contexto de la interoperabilidad en blockchain. La arquitectura de orquestador proporciona un punto de control centralizado, lo que puede resultar ventajoso para garantizar la seguridad y el cumplimiento de reglas. Sin embargo, también puede introducir un único punto de falla y puede limitar la flexibilidad y escalabilidad de la solución de interoperabilidad. La coreografía, por otro lado, proporciona un enfoque más descentralizado y flexible para la interoperabilidad, pero también puede ser más compleja y difícil de coordinar, y puede requerir más esfuerzo para establecer y mantener estándares de interoperabilidad.

2.4. Soluciones de interoperabilidad relevadas

Durante el desarrollo del proyecto, se identifican y estudian diferentes artículos, publicaciones y proyectos que apuntan a objetivos similares a los de este proyecto.

Al ser blockchain un área relativamente nueva se encuentra en plena experimentación y crecimiento. Muchos de los trabajos encontrados al respecto son muy recientes o están en pleno desarrollo.

Actualmente existen multitud de alternativas para lograr la interoperabilidad, pero en general tienen un enfoque para blockchains públicas o para privadas, sin abarcar la interoperabilidad público-privada

A continuación se hace un análisis de las distintas opciones que se consideraron.

2.4.1. Proyecto de grado BIG

Esta solución implementa un componente estilo gateway encargado de intercambiar los mensajes entre diferentes plataformas de blockchain . El gateway está compuesto de un Router y dos o más Conectores. Cómo se observa en la figura 2, el gateway recibe eventos emitidos por una red blockchain y destinado a otra blockchain y se encarga de rutear estos eventos. Vale la pena destacar que, el gateway no modifica de ninguna manera la información del evento recibido. El proyecto BIG fue diseñado para interoperar dos plataformas de blockchain permisionadas llamadas Hyperledger Fabric y Corda R3; y presenta un caso de uso utilizando estas plataformas.

La solución entra en la categoría de **trusted relays** por lo que las organizaciones que deseen interoperar las blockchains con ella deberán resignar cierto nivel de

confianza en el componente principal. Éste sigue una arquitectura de plugins, esto es porque, debido a la heterogeneidad de los eventos de cada blockchain, y la forma de la que las redes envían estos eventos y mensajes (típicamente a través de su propio SDK) es necesario la implementación de un plugin diferente para cada una de ellas. También le da la versatilidad al gateway de poder extenderse para interoperar con otra blockchain simplemente agregando otro plugin. Según se define en la sección 2.3, el gateway funciona con arquitectura de coreografía ya que únicamente se envía los mensajes a donde corresponda. Siendo los smart contracts los encargados de ejecutar la lógica de negocio.

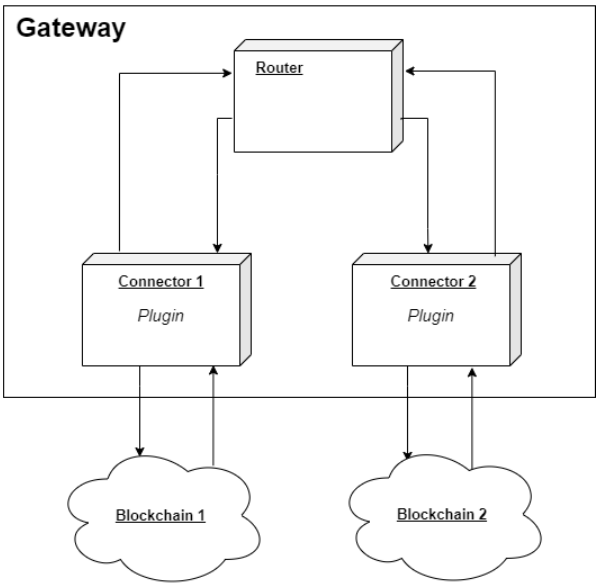

Figura 2: Arquitectura del Gateway.

A continuación se describen los componentes del Gateway.

Conector

El conector es el encargado de comunicar a la blockchain con el *Router*. Más precisamente, gestiona los eventos generados en la blockchain a ser enviados al *Router* (con destino otra blockchain) y de recibir los eventos precedentes de otras blockchains desde el *Router* para impactarlos en su red. Además, proveen al *Router* de una interfaz común para desempeñar su tarea sin tener que adaptarse a un formato de evento distinto por cada blockchain.

Router

El *Router* es el encargado de recibir y enviar los eventos desde y hacía los conectores según corresponda. Para lograrlo, el evento debe tener en su header la blockchain, smart contract y operación destino (siempre que corresponda). El *Router* es el encargado de conectar todos los conectores disponibles en el *gateway*.

Para lograr la interoperabilidad deseada la solución debe seguir los siguientes pasos:

1. La blockchain origen genera un evento a emitir a una blockchain destino
2. La red de origen emite un evento al *gateway*, más precisamente al *Conector* correspondiente a su red
3. El evento recibido por el conector es enviado por este mismo al *Router*
4. El *Router* procesa la información de qué blockchain es origen, el tipo de mensaje y cuál es la blockchain destino, para luego enviarlo al conector de la blockchain correspondiente
5. El evento es recibido por el conector correspondiente a la blockchain destino para que procese la información y la envíe a la red.

2.4.2. Hyperledger Cactus

Hyperledger Cactus es una solución de interoperabilidad en blockchain de código abierto que permite a los desarrolladores crear aplicaciones que utilicen múltiples blockchain. Fue desarrollado bajo el paraguas de Hyperledger por un grupo de empresas, incluidas Fujitsu, Accenture e Hitachi [29].[3]

Cactus funciona proporcionando un conjunto de bibliotecas, APIs y herramientas que permiten la integración de diferentes plataformas de blockchain en una sola aplicación. Esta aplicación se puede utilizar para realizar transacciones y compartir datos entre diferentes partes.

Cactus utiliza una arquitectura de plugins que permite agregar nuevas plataformas blockchain según sea necesario. Esto facilita la integración de nuevas blockchain en una aplicación existente.

Aparte de los conectores necesarios para cada blockchain, Cactus posee tres componentes principales. Ellos son validadores, verificadores y la lógica de negocio. Además de estos también existe la interfaz de ruteo encargada de comunicar estas capas. En la figura 3 podemos ver una arquitectura de los componentes mencionados.

[3] En el resto del documento se menciona a Hyperledger Cactus y Cactus de forma indistinta.

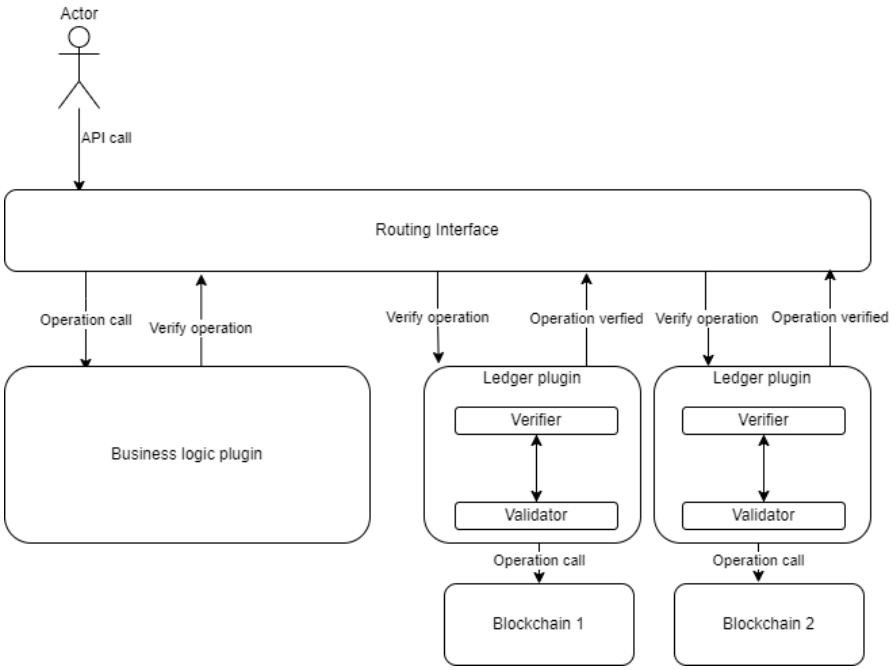

Figura 3: Arquitectura de Cactus.

Los validadores son responsables de verificar la validez de la transacción y garantizar que los cambios propuestos en la blockchain sean consistentes con las reglas definidas en el smart contract. En Hyperledger Fabric, por ejemplo, los validadores están representados por nodos peers que ejecutan el smart contract que define las reglas para la validación de transacciones.

Los verificadores, por otro lado, son responsables de verificar la validez de la propuesta de transacción y garantizar que sea consistente con el estado de la red blockchain. En Hyperledger Fabric los verificadores están representados por nodos orderers que reciben las propuestas de transacciones de los validadores y crean bloques de transacciones. Los verificadores se aseguran de que las propuestas de transacciones aprobadas sean consistentes con el estado de la blockchain.
En Cactus, los validadores y verificadores trabajan juntos para garantizar la integridad y coherencia de las transacciones en varias blockchain. Cuando se inicia una transacción, la propuesta de transacción se envía a la blockchain correspondiente para que los validadores la validen. Una vez validada la propuesta, se envía para su verificación por parte de los verificadores. Una vez que se verifica y ordena la transacción, se impacta en la blockchain y el estado de la red se actualiza en consecuencia. Mediante el uso de validadores y verificadores, Hyperledger Cactus garantiza que las transacciones se procesen de manera correcta y uniforme en todas las blockchain integradas.

La lógica de negocio en Cactus define las operaciones e interacciones entre las diferentes blockchain que se están integrando. Esto se realiza en un componente por fuera de las blockchain, con lo cual todas las transacciones entre blockchains deben pasar por la lógica de negocio de Cactus. Por ejemplo, define las reglas para

verificar la autenticidad de las solicitudes de transacciones, los requisitos para respaldar transacciones y las reglas para ordenar y enviar transacciones a las blockchain. La lógica de negocio también puede definir cómo se integran los sistemas externos y las fuentes de datos con las blockchain para permitir la ejecución de casos de uso complejos [30].

Según se define en la sección 2.3, Cactus funciona cómo un orquestador ya que la lógica de negocio es la encargada de crear y hacer cumplir las reglas sobre cómo las diferentes blockchain pueden interactuar entre sí. Cactus actúa cómo un intermediario de confianza, verificando y validando las transacciones que se intercambian entre diferentes blockchain.

2.4.3. Otras soluciones

A continuación se presentan otras alternativas relevadas.

Polkadot y Cosmos

Ambos proyectos son protocolos que proveen una interfaz para que diferentes máquinas de estado se comuniquen entre sí.

Se les conoce cómo “blockchain of blockchains”, siendo su principal ventaja no precisar de un intermediario confiable, manteniendo en la interoperabilidad, los beneficios de utilizar blockchains.

Funcionan mediante un sistema de sharding, donde existen blockchains que contienen un subconjunto de información y una blockchain central que coordina los cambios de estados de las otras blockchains. En la figura 4 se observa la arquitectura de Cosmos, donde el Hub es la blockchain central y cada zona es una blockchain independiente.

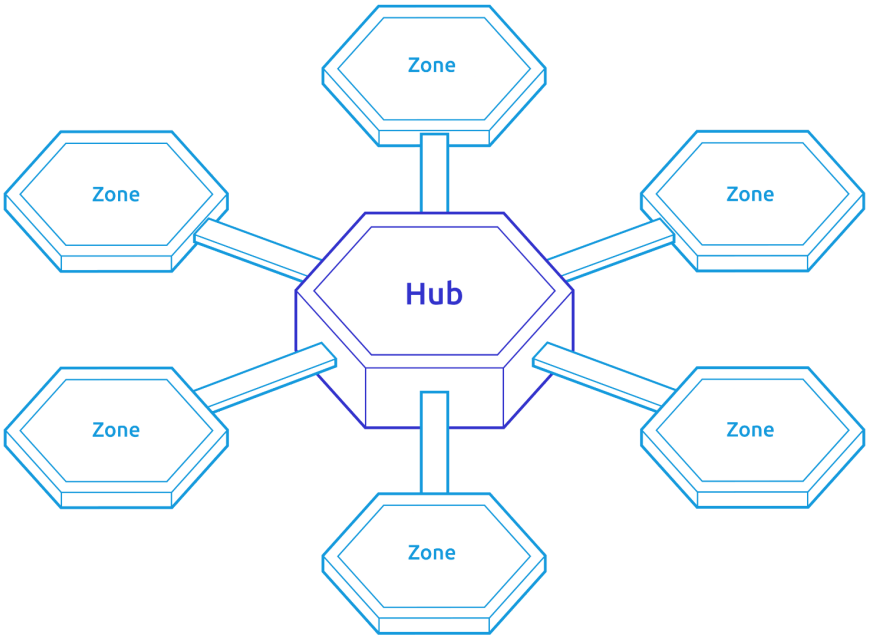

Figura 4: Arquitectura de Cosmos fuente: [31].

La principal diferencia entre ambas es que Polkadot garantiza la validación de las operaciones. Esta validación se realiza por la blockchain central creando un ambiente de confianza. Por el contrario, Cosmos es menos invasiva y plantea que cada blockchain es encargada de su seguridad. Esto permite más libertad a cada blockchain, pero implica que al interoperar se debe tener confianza en el mecanismo de validación de la blockchain destino [32].

Hyperledger YUI

YUI es, en el idioma japonés, la palabra para nudo, unir y conectar. El proyecto implementa módulos para comunicación y desarrollo cross blockchain similar a Cactus pero también dispone de un explorador que inspecciona el estado y eventos de las blockchains.

Para la comunicación se basa en el protocolo de intercomunicación de blockchains de Cosmos (IBC). Además, los módulos de desarrollo cross blockchain incluyen herramientas para la implementación de operaciones atómicas.

Según establecen, confiando en una verificación on-chain de la otra blockchain, la comunicación se puede llevar a cabo sin preocuparse por el comportamiento de agentes off-chain. Esto le permite el manejo de mensajes hacia blockchains con listas de control de acceso (ACLs) cómo son los canales en Fabric dónde no todos los nodos de la red pueden ver los mensajes brindando así solución al problema de la autenticación y privacidad.

Cómo se observa en la figura 5, es necesario instalar un módulo IBC en cada blockchain a interoperar. Dicho módulo es el encargado de consultar y enviar

paquetes a los smart contract de la aplicación. También se precisa un Relay que comunica los modulos IBC instalados en las distintas blockchains.

Ya se han implementado los módulos para los proyectos de Hyperledger Fabric y Besu, cómo también para el proyecto de R3 Corda enfocándose en blockchains permisionadas [33].

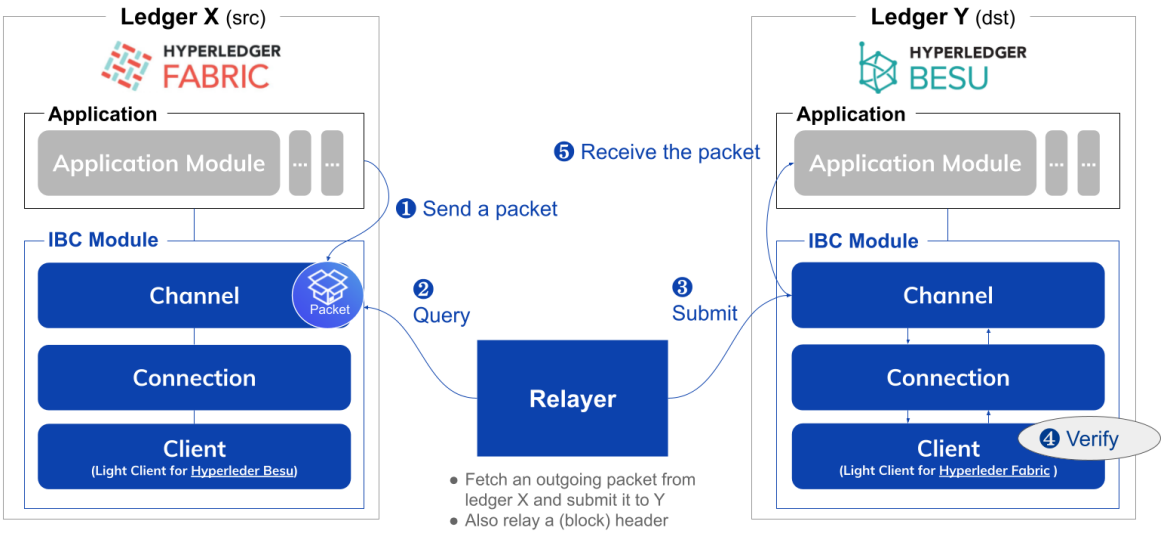

Figura 5: Arquitectura de Hyperledger YUI fuente: [34].

Hyperldger Weaver

Al igual que Cactus es un proyecto incubado por Hyperledger y respaldado por IBM. Su principal diferencia respecto a Cactus es no precisar de un gateway confiable para realizar la interoperabilidad. La interoperabilidad ocurre a través de protocolos que obtienen su confianza mediante los mecanismos de consenso de las blockchains nativas. Las reglas y requerimientos se implementan con los mecanismos de gobernanza internos. Lo cual ofrece la ventaja de minimizar la infraestructura compartida entre blockchains preservando la seguridad y descentralización de cada blockchain. Actualmente soporta Fabric y Corda únicamente [35].

Tesis de magíster de Stefano Franzoni

Stefano Franzoni implementa una Dapp que se comunica tanto con Ethereum cómo con Fabric. Su enfoque de solución es utilizar la implementación de Fabric chaincode Ethereum Virtual Machine. Esto permite correr smart contracts desarrollados en Solidity sobre Hyperledger Fabric. Por lo tanto, en la Dapp utiliza la librería web3 para comunicarse con ambas blockchains.

Para el desafío de identidad al trabajar con blockchains públicas y privadas, lo resuelve con un proxy que realiza un mapeo uno a uno entre una dirección de Ethereum y una identidad de Fabric [36].

En la figura 6 se observa cómo no existe comunicación directa entre ambas blockchain. La Dapp es la encargada de realizar los pedidos tanto a Fabric cómo a Ethereum. Para ello utiliza la librería web3 ya que los smart contract se encuentran desarrollados con solidity en ambos casos. Con la diferencia que en cuando el destino es la blockchain Fabric, el pedido pasa a través del proxy FAB3 para mapear la dirección de Ethereum a la identidad de Fabric.

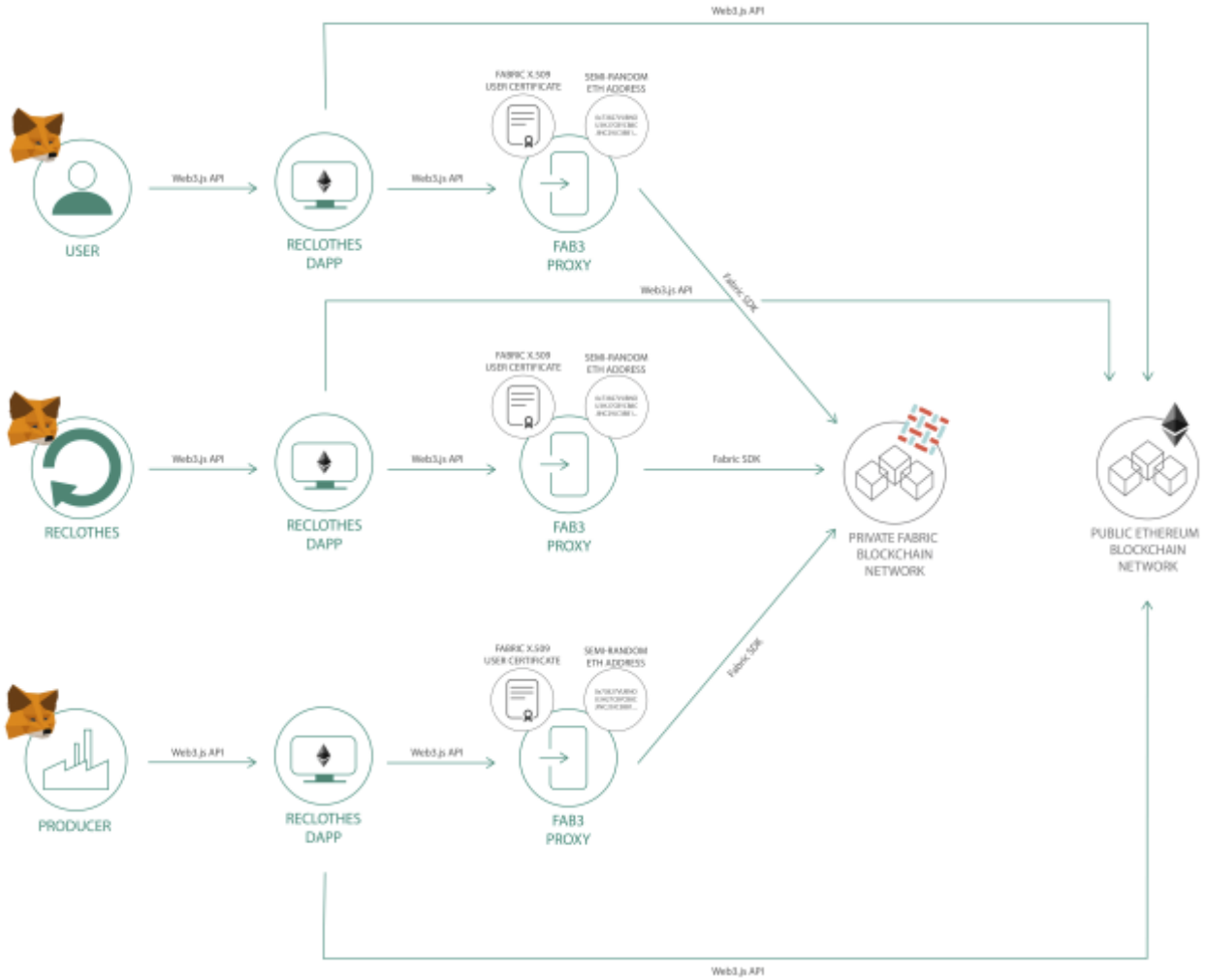

Figura 6: Arquitectura de Tesis de magíster de Stefano Franzoni [34].

3. Análisis de requerimientos

En este capítulo se realiza un análisis de la problemática planteada. Esta se basa en extender la interoperabilidad alcanzada en el marco del proyecto de grado BIG para lograr la interoperabilidad con Ethereum.
Para la comprensión de este sistema se presentan los requerimientos del proyecto en la sección 3.1, una descripción general del escenario de uso propuesto en la sección 3.2, los requerimientos del mismo en la sección 3.3 y finalmente se define el alcance final del proyecto en la sección 3.4.

3.1. Requerimientos del proyecto

El requisito principal del proyecto es continuar el trabajo realizado en el proyecto de grado BIG, incorporando Ethereum a la suite de plataformas de interoperabilidad.

Para cumplir con el requisito macro, se relevan los siguientes requisitos específicos. Primero, se deben analizar ambos frameworks, para establecer el costo/beneficio de evolucionar BIG o utilizar Cactus. Además se debe relevar el estado del arte para hallar otras posibles alternativas aparte de las previamente mencionadas. El segundo requerimiento relevado es implementar transacciones atómicas, con el fin de mantener la consistencia entre blockchains. El tercer requisito es que la solución de interoperabilidad debe ser genérica y no atada a un caso de uso particular. El último requerimiento es verificar la solución y con este fin se debe implementar y probar un caso de uso.

3.2. Descripción del escenario propuesto

Se pretende probar la factibilidad técnica de la solución mediante la implementación de un caso de uso, el cual será modelado por un sistema de suministro de energía para coches eléctricos.

Los autos eléctricos al igual que las blockchains cada vez tienen más adopción mundial. Dichos autos pueden ser recargados comprando energía en surtidores pertenecientes a estaciones de servicio. Para aprovechar las ventajas de las blockchains [38], la información relativa a la venta de energía se desea almacenar utilizando una red de estas. Al mismo tiempo para brindar más alternativas de pago a los clientes, se desea que el pago pueda ser realizado con criptomonedas. Ambas situaciones en conjunto presentan un problema ya que las blockchains públicas, donde existen las criptomonedas, no son adecuadas para almacenar información confidencial de cada estación de servicio. Por lo tanto las estaciones guardan su información sobre una blockchain privada, mientras que los clientes tienen sus fondos en una blockchain pública. Como blockchain privada se utiliza Hyperledger Fabric y como blockchain pública se opta por Ethereum.

La figura 7 presenta gráficamente el escenario, donde existen dos entidades participantes. Por un lado el consorcio de estaciones de servicio, encargadas de suministrar la energía a los clientes, donde cada estación posee un nodo en la red Fabric. Por otro lado, están los clientes quienes pueden comprar la energía utilizando criptomonedas, en este caso Ethers. Tanto los clientes cómo los surtidores poseen billeteras en Ethereum.
Todos los pedidos del cliente son realizados a través de una Web App. La Web App ejecuta un smart contract en Fabric o Ethereum que inicia la operación con la otra blockchain utilizando el framework de interoperabilidad implementado.

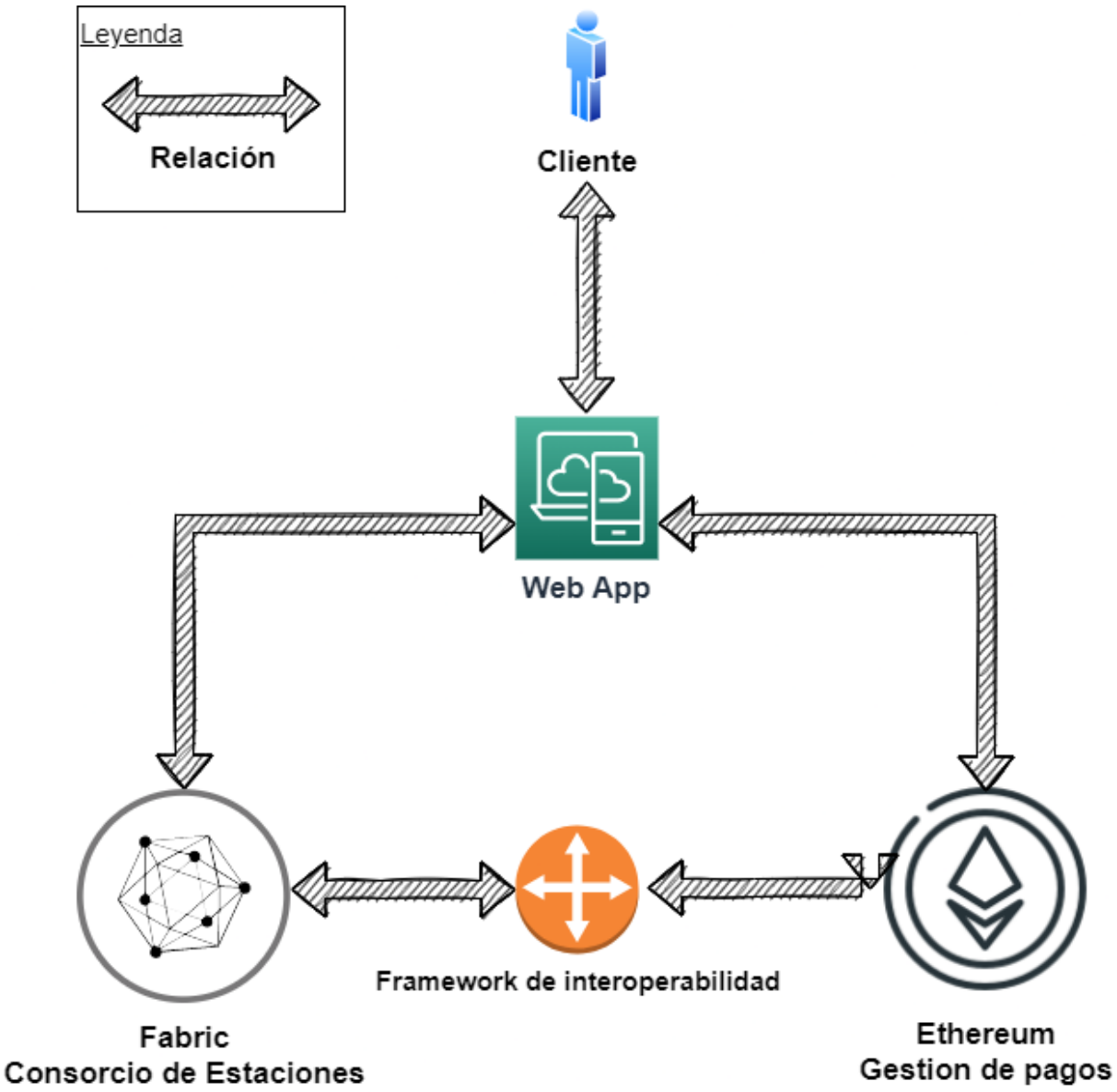

Figura 7: Escenario de uso

Cuando el cliente llega a la estación lo primero que debe hacer es consultar si el surtidor elegido tiene suficiente energía para satisfacer su pedido. Luego el cliente cuenta con dos opciones para realizar un pedido de compra. Una alternativa es mediante el inicio de la compra en Ethereum y la otra es iniciando desde Fabric. En caso de no poder liberarse la energía, debe quedar consistente la información en ambas blockchain.

3.3. Requerimientos del escenario propuesto

A continuación se presentan los requerimientos funcionales y no funcionales que debe soportar el sistema.

3.3.1. Requerimientos funcionales

En base a lo descrito en la sección 3.2 se llega a los siguientes requisitos funcionales para cada blockchain. La tabla 1 define los requisitos funcionales que se deben implementar.

ID	Título	Descripción
REQ_FUN_1	Consultar stock de energía en surtidor por parte del cliente	El cliente selecciona en la web app el surtidor y se le despliega en pantalla la cantidad de energía que cuenta en stock.
REQ_FUN_2	Compra de energía por parte del cliente iniciando desde Fabric	El cliente selecciona en la web app el surtidor, el monto y Fabric, como la blockchain, con la cual va a iniciar el pedido la web app. Se libera la energía en Fabric y luego se transfieren los Ethers en Ethereum.
REQ_FUN_3	Compra de energía por parte del cliente iniciando desde Ethereum	El cliente selecciona en la web app el surtidor, el monto y Ethereum, como la blockchain con la cual va a iniciar el pedido la web app. Se transfieren los Ethers en Ethereum y luego se libera la energía en Fabric.
REQ_FUN_4	Compra de energía de forma atómica por parte del cliente iniciando desde Fabric	El cliente selecciona en la web app el surtidor, el monto y Fabric, como la blockchain, con la cual va a iniciar el pedido la web app. Los Ethers se liberan a la billetera del surtidor cuando la energía se libera al cliente. En caso de error y no recibirla, se le devuelven los Ethers.

REQ_FUN_5	Compra de energía de forma atómica por parte del cliente iniciando desde Ethereum	El cliente selecciona en la web app el surtidor, el monto y Ethereum, como la blockchain, con la cual va a iniciar el pedido la web app. Los Ethers se liberan a la billetera del surtidor cuando la energía se libera al cliente. En caso de error y no recibirla, se le devuelven los Ethers.

Tabla 1: Requisitos funcionales

3.3.2. Requerimientos no funcionales

Además de los requisitos funcionales se relevaron los siguientes requisitos no funcionales. Dichos requisitos son deseables para la correcta seguridad, rendimiento y disponibilidad de la solución propuesta. Se pueden ver en la tabla 2.

ID	Título	Descripción
REQ_NO_FUN_1	No invasiva	La solución no modificará el código nativo de las blockchain.
REQ_NO_FUN_2	Eficiente	Tiempos cercanos a tiempo real. La solución no debe ser el cuello de botella.
REQ_NO_FUN_3	Descentralizado	No debe existir una entidad central que gobierne la solución
REQ_NO_FUN_4	Prevenir Double Spending	No debe ser posible gastar dos veces los mismos ethers.
REQ_NO_FUN_5	Sin custodia	Los fondos del cliente no deben ser custodiados por la solución
REQ_NO_FUN_6	Disponibilidad	La solución no debe depender de un único punto de falla

Tabla 2: Requisitos no funcionales

3.4. Alcance del proyecto

En esta sección se resume el alcance final de la solución luego de la negociación con el tutor. A continuación se detalla por complemento los requerimientos a realizar.

Se acordó que los requisitos funcionales REQ_FUN_3 y REQ_FUN_4 no caen dentro del alcance del proyecto. Al implementar una operación de consulta y de ejecución iniciando desde cada blockchain se considera suficiente para validar la solución.

Respecto a los requisitos no funcionales quedaron fuera del alcance los requisitos REQ_NO_FUN_3 y REQ_NO_FUN_6.

4. Análisis de soluciones existentes

En esta sección se analiza las distintas alternativas presentadas en la sección 2.4. Primero se comparan características funcionales y no funcionales de BIG y Cactus. Luego se detallan los inconvenientes encontrados en las otras soluciones relevadas. Por último se presentan las conclusiones del análisis.

4.1. BIG vs Cactus

El enfoque estuvo principalmente en estas dos alternativas. Cactus por ser la única solución relevada que actualmente permite interoperar entre blockchains públicas y privadas de manera genérica. Y BIG por ser la sugerida por el tutor. Además ambas soluciones permiten consultar información y ejecutar comandos entre blockchains. La tabla 3 presenta las características funcionales definidas en la sección 2.3.

Característica	Cactus	BIG
Consultar información	Soporta	Soporta
Ejecutar comando	Soporta	Soporta
Actividad atómica y consistente	Soporta	No soporta
Tipo de solución	Trusted relay	Trusted Relay
Identidad	No lo resuelve, pero permite utilizar soluciones externas.	No lo resuelve, solo blockchain privadas
Gestión de claves	Las claves privadas se guardan en Cactus	No lo resuelve pero no es obligatorio que sean almacenadas por BIG
Privacidad	No proporciona mecanismos	No proporciona mecanismos
Consenso	Lo resuelve parcialmente	No lo resuelve
Autorización	No proporciona mecanismos	No proporciona mecanismos

Tabla 3: Comparación características funcionales

A continuación se realiza una explicación más detallada de la tabla 3:

- Consulta y ejecución de comandos: Las operaciones son soportadas en ambas soluciones.
- Operaciones atómicas: En BIG no está implementado ningún tipo de operación atómica cross chain. Cactus provee una api para desplegar e interactuar con Hash Timed Locked Contracts[39].
- Identidad: Al comunicar una red permisionada contra una red no permisionada, Cactus no resuelve el problema directamente pero ofrece la posibilidad de integrar otro framework de la comunidad Hyperledger llamado Indy[40] para gestionar identidades. BIG no implementa una gestión de identidades pero se puede evolucionar para usar el mismo framework.
- Gestión de claves: Al momento de gestionar las claves privadas del cliente, en Cactus es necesario compartir la clave privada con el framework lo que se considera un problema grave de privacidad. Esto se debe a su arquitectura de orquestador, en la cual el cliente no se comunica directamente con las blockchains. Por lo tanto, Cactus debe conocer la clave privada para firmar las transacciones en nombre del cliente.
  En la arquitectura de BIG, el cliente inicia la operación en la web app la cual se comunica directamente con las blockchains. Por lo tanto se puede utilizar una billetera de custodia propia para la gestión de claves en la web app. Con lo cual el cliente sigue siendo dueño total de sus claves.
- Privacidad: Esta propiedad refiere a la privacidad de los datos dentro de cada blockchain y el funcionamiento de ésta en redes permisionadas en contraposición con redes no permisionadas. Ni Cactus ni BIG implementan mecanismos para la privacidad de los datos
- Consenso: Esta propiedad refiere a la verificación del consenso de la red origen previo a impactar el cambio en la red destino. Cactus monitorea las blockchain informando a la Business Logic el estado de la red, por lo que se cuenta con la información de las transacciones que suceden en la blockchain. Falta implementar la verificación en la Business Logic que decida si determinado bloque ya cuenta con el consenso de la red. BIG no implementa ni el monitoreo ni la verificación de las transacciones ocurridas en las blockchain.
- Autorización: El manejo de roles al realizar acciones sobre una blockchain no es implementado por ninguna de las dos soluciones.

La tabla 4 presenta la comparativa de las características no funcionales de cada solución.

Característica	Cactus	BIG
Comunidad	Activa, mediana, dinámica	Proyecto cerrado
Documentación	Media	Alta
Disponibilidad	Varios nodos	Único punto de falla
Complejidad	Alta	Baja
Overhead	Medio	Bajo
Reusabilidad	Medio	Alto
Seguridad	Sólo cifrado de mensajes sensibles	No implementa
Invasiva	NO	NO
Descentralización	NO	NO

Tabla 4: Comparación características no funcionales

A continuación se detalla que se entiende por cada una de las características de la tabla 4 y se profundiza en su explicación:

- Comunidad: Se entiende por comunidad la posibilidad de recibir feedback y ayuda desde el exterior del proyecto. Cactus es un proyecto activo con varios colaboradores aportando al proyecto. Al mismo tiempo cuenta con el respaldo de Hyperledger, IBM y Accenture. Pero no se pudo establecer contactos fluidos a la hora de presentar dudas. En cuanto a BIG es un proyecto cerrado, pero se tiene a los estudiantes del proyecto anterior para realizar consultas.
- Documentación: En Cactus se cuenta con el whitepaper y los readme que consideramos aceptables. Para BIG se cuenta con el informe, que a nuestro entender es bastante completo y entendible.
- Disponibilidad: Se define disponibilidad cómo la capacidad de levantar la solución con múltiples nodos. Según refleja su whitepaper, Cactus consta de varios nodos de alta disponibilidad para sus sistemas pero en la práctica no se llega a comprobar dicha característica. En BIG no hay implementado ningún mecanismo para levantar más de un nodo por componente
- Complejidad: Se define complejidad cómo la cantidad de entidades de código (clases, archivos de configuración, etc) y un estimado de sus líneas de código. Se entiende que Cactus, por sus componentes de validadores, Routing interface y Business Logic tiene mayor complejidad que BIG a la hora

de tener que impactar cambios de código.

- Overhead: Se define como overhead al camino que tienen que recorrer los mensajes para llegar de una blockchain a la siguiente. En el caso de Cactus es necesario ir primero a la business logic para luego ir a la primera blockchain, esperar que termine la ejecución, retornar a la business logic e ir a la siguiente blockchain. En el caso de BIG la comunicación inicia en la primera blockchain y pasa solo por el conector - router - conector para llegar a la siguiente, dando menos saltos.
- Reusabilidad: Se define por reusabilidad a la capacidad de utilizar la solución para otros casos de uso y otras blockchains. Por un lado ambas herramientas usan la misma arquitectura de plugins con lo cual se puede agregar conectores para otras blockchains. En cuanto a ser reutilizable para otros casos de uso, en BIG no se debe cambiar nada pero por el contrario en Cactus hay que implementar una business logic específica para cada caso de uso.
- Seguridad: Se define la seguridad cómo la posibilidad de realizar conexiones cifradas (https) entre componentes y firma de mensajes sensible. Cactus no realiza conexiones https entre los componentes de su arquitectura aunque sí cifra algunos mensajes sensibles. BIG no implementa ninguna de las dos cosas.
- Invasiva: Una solución es invasiva si es necesario modificar el funcionamiento de la blockchain para poder interoperar. Dicha propiedad es algo que respetan tanto BIG cómo Cactus.
- Descentralización: Se considera que una solución es descentralizada, si cada blockchain participa en el proceso de toma de decisiones sobre el flujo de trabajo en vez de depender de un único punto central de control. Cactus, cómo se explica en la sección 2.4.2, actúa como un middleware con arquitectura de orquestador. Esto implica que la business logic de Cactus, es la encargada de gestionar todas las interacciones. En cambio BIG utiliza una arquitectura de coreografía, delegando la responsabilidad de la lógica a las blockchains.

4.2. Otras soluciones

Se encontró que las soluciones introducidas en la sección 2.4.3 no se adaptan correctamente a los requisitos relevados en la sección 3.3. A continuación se fundamenta porque se considera que no son las tecnologías adecuadas para nuestro objetivo.

Polkadot y Cosmos

Ambas herramientas son sumamente invasivas y restrictivas en cuanto a los requerimientos de las blockchains con las que pueden trabajar. Por ejemplo, en

ninguna de las dos se permite que la blockchain utilice la máquina virtual de Ethereum. Por lo que, ni Fabric ni Ethereum se pueden utilizar sin cambios mayores en su funcionamiento.

Weaver

Se comparó su documentación, comunidad e implementación con Cactus y se encontró sumamente inmaduro. El motivo principal por el cual se decidió descartar Weaver, es porque planea unirse a Cactus en el futuro cercano.

Yui

El problema que presenta es su arquitectura de listas de control de acceso, ya que no está pensada para interoperar con blockchains públicas. Por lo que no cumple con el principal requisito del proyecto.

Tesis de magíster de Stefano Franzoni

La solución es muy específica para Fabric, por lo que no es un framework que sea escalable a otras blockchains. Sirve únicamente para la comunicación entre Ethereum y Fabric, lo cual no cumple con el requisito que la solución debe ser genérica.
Otra punto en contra es que realiza un mapeo en la capa de aplicación: la wallet de Ethereum con la identidad de Fabric. Como consecuencia, impacta en la anonimidad de usuarios en Ethereum.

4.3. Conclusiones

En cuanto a características funcionales se encontró que Cactus cuenta con mayor progreso en cuestiones de permitir transacciones atómicas, gestionar identidades y controlar el consenso. Por otro lado, BIG cuenta con la ventaja de que no es necesario que el cliente confíe sus claves privadas a la solución de interoperabilidad.

En las características no funcionales, detectamos que BIG es el que cuenta con mejores prestaciones ya que al tener arquitectura de coreografía presenta un overhead menor y una mayor descentralización. La ventaja que ofrece Cactus es la mayor disponibilidad pudiendo levantar la solución en varios nodos.

La mayor problemática para los requerimientos, que presenta Cactus, es debido a la arquitectura de orquestador que utiliza. Cada pedido inicia en la web app, es ruteado por Cactus a la blockchain correspondiente y retorna a la web app. Esto resuelve el problema de interoperabilidad a nivel de aplicación y permite que una aplicación se comunique con múltiples blockchain pero no resuelve la interoperabilidad blockchain a blockchain que es la buscada en esté proyecto.

La diferencia que notamos entre las ventajas que ofrece Cactus y las que ofrece BIG es que se puede evolucionar BIG para implementar las ventajas ofrecidas por

Cactus. En cambio para poder solucionar las desventajas de Cactus sería necesario una re-estructura de su arquitectura.

Para esto se investigaron las alternativas expuestas en el anexo III entendiendo que las únicas posibles para que Cactus cumpla los requisitos no funcionales relevados son: Extender Cactus agregando más middlewares o modificarlo para que por defecto quede escuchando eventos de las blockchain. Nuestro análisis es que esto trae más costos que beneficios debido a la complejidad de Cactus. Teniendo en cuenta lo previamente mencionado, se decidió extender BIG.

5. Diseño

En este capítulo se analizarán los detalles de diseño de la solución propuesta. En las secciones 5.1 y 5.2 se presentan la extensión de la solución BIG así cómo algunas decisiones relevantes del mismo. En la sección 5.3, se destacan algunas mejoras hechas al Gateway con respecto a la primera versión de BIG y finalmente en la sección 5.4, se presenta el diseño de las operaciones atómicas implementadas en el proyecto.

5.1. Arquitectura del Gateway

Cómo se vio en la sección 2.4.1, el Gateway está formado por el *Router* y los conectores. Éstos últimos se implementan de forma independiente y uno para cada blockchain. En este caso, cómo se extiende BIG para interoperar con la red Ethereum, se tendrá un conector para esta red. También se implementa una versión extendida del conector de Fabric del proyecto anterior para soportar el nuevo formato de los mensajes y la generalización en el mismo. En la figura 8 podemos ver la interacción de las redes con el Gateway. Las redes disparan eventos hacía el Gateway y cómo resultado de estos eventos, el Gateway dispara la ejecución de operaciones en los smarts contracts, deployados en las redes para llevar a cabo el caso de uso.

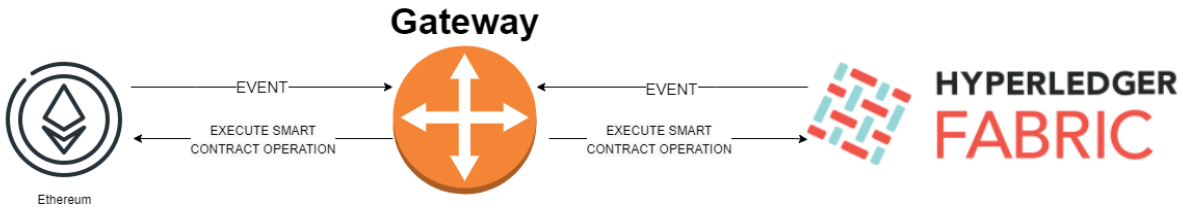

Figura 8: Interacción Gateway con redes.

En la figura 9 se puede observar un diagrama detallado de cada componente. El diagrama es similar al de la figura 2 pero con más detalles de diseño y orientado a nuestro proyecto (ya que aparecen las redes Fabric y Ethereum y los conectores correspondientes a estas redes). En verde se ven los componentes nuevos y en azul los componentes que fueron mejorados. Cabe destacar también que se debe asumir el Gateway cómo un componente de confianza.

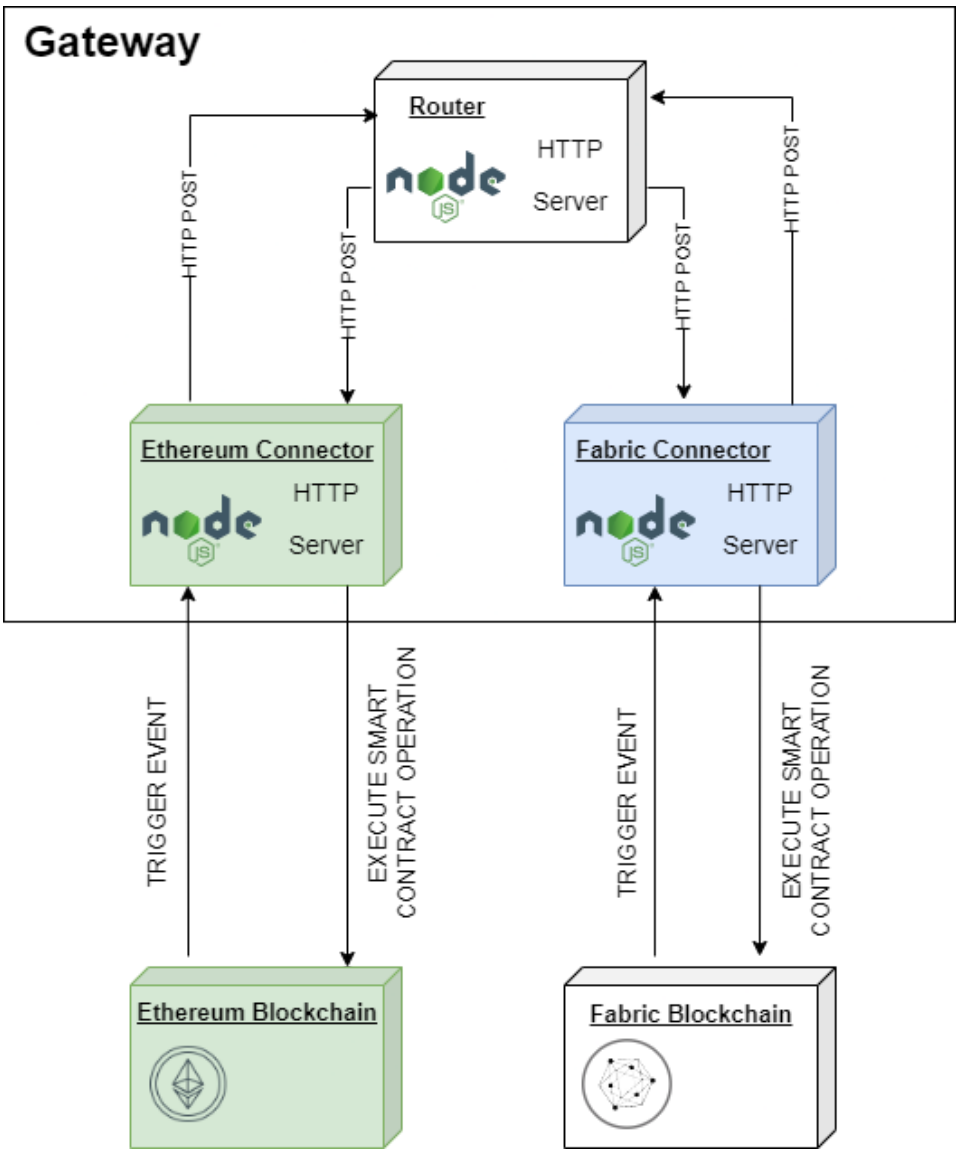

Figura 9: Arquitectura detallada del gateway

5.1.1. Componentes

A continuación se verá en detalle la funcionalidad de cada componente.

Fabric Connector
Este componente se encarga de recibir eventos desde la blockchain de Fabric para posteriormente enviarlos al *Router* consumiendo su endpoint. Por otro lado, el conector expone un endpoint HTTP dónde recibirá eventos enviados por el *Router*.

Ethereum Connector
El conector de la red Ethereum implementa las mismas funciones que el conector para la red Fabric, recibir eventos desde la blockchain Ethereum y recibir eventos desde el *Router* hacia Ethereum. Para lo último, expone un endpoint HTTP dónde recibirá los eventos, destinados a la red, desde el *Router*. Este conector cuenta con

una billetera en la red Ethereum, necesaria para poder invocar las operaciones de los smart contracts.

Router
El router es el componente encargado de tomar un evento del conector de la blockchain origen y redirigirlo hacía el conector de la blockchain destino. Para recoger el evento, el *Router* expone un endpoint HTTP de un servicio REST e interpreta el mensaje para enviar dicho evento al conector correspondiente. Dentro de la configuración de este componente se especifican todos los conectores con los que debe interactuar en su archivo de configuración, especificando su dirección y puerto.

5.1.2. Comunicaciones

La comunicación entre el *Router* y el *Conector,* se implementa mediante endpoints HTTP con servicios REST. Esto permite que los distintos conectores puedan estar desarrollados en diferentes lenguajes de programación. La comunicación entre el *Gateway* y cada blockchain depende de las plataformas utilizadas y sus SDKs. Parte del propósito de los conectores es soportar el mecanismo de comunicación específico para cada plataforma. La comunicación dentro del gateway (Conectores vs Router) se hace vía HTTP POST, consumiendo los endpoints expuestos por los componentes. La comunicación entre Fabric y el conector se realiza utilizando el SDK provisto por Hyperledger [41]. La comunicación entre Ethereum y el conector se realiza utilizando la librería Web3.js [42]. En la sección 6.1 se describe el funcionamiento de ambas conexiones.

5.2. Decisiones de diseño

En esta sección se presentan algunas decisiones destacables de diseño.

5.2.1. Forma de interacción request-response con replyTo

Se definió un formato de mensajes que permitiera una interacción similar a un sistema de mensajería request-response. Este formato se basa en los patrones de mensajería Request-Reply [43], Return Address [44] y Correlation Identifier [45].
En un sistema request-response usualmente hay un cliente y un servidor. El cliente envía un *request* al servidor, consultando por cierta información o acción, que también contiene información acerca de cómo quiere recibir la respuesta. El servidor, luego de procesar la solicitud, envía la *response* hacia el cliente. La misma contiene el resultado de la acción o la información solicitada. En este caso una blockchain va a operar como cliente y la otra como el servidor.

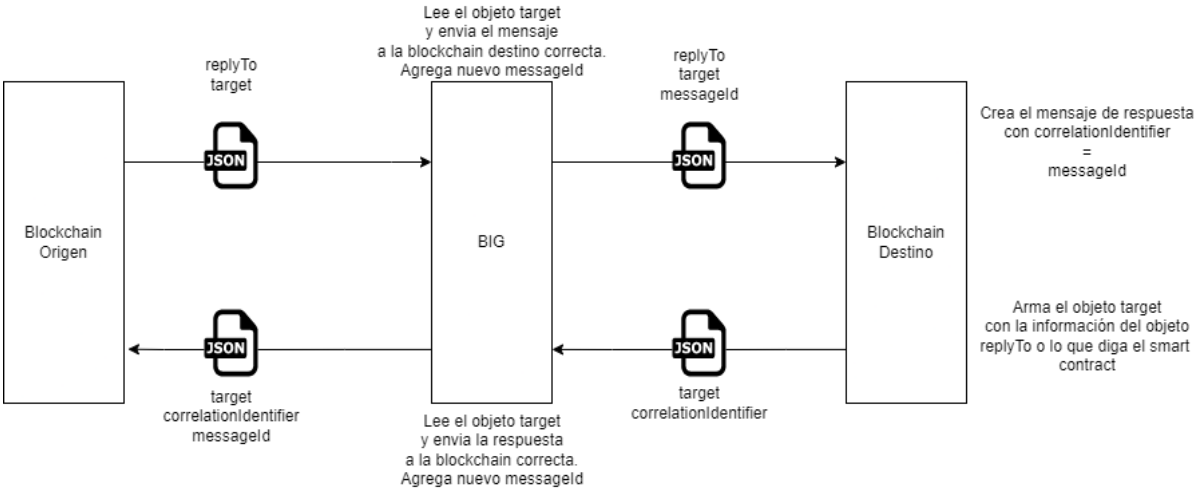

Figura 10: Esquema de interacción request-response con ReplyTo

Se definieron los campos messageId y correlationIdentifier que permiten una trazabilidad de los mensajes dentro del gateway. Además, al incorporar un objeto replyTo, es posible indicar a que blockchain, smart contract y operación enviar la respuesta, una vez procesada la solicitud. En la figura 10 se puede ver un esquema del funcionamiento.

A continuación se presenta el formato de los mensajes intercambiados entre el router y los conectores. Los eventos se conforman por dos secciones, la sección *header* y la sección *data*, el primero transporta datos de infraestructura y el segundo los datos de negocio.

En la Figura 11 se puede ver el formato completo de los mensajes. Se destacan, dentro del *header,* el objeto *target,* dónde se debe especificar la *blockchain* (que utilizará el *Router* para rutear el evento), y el *contract* junto con el *operation* (utilizada por el conector para indicar el smart contract y la operación a ejecutar).
Dentro del *header* también existe una sección opcional, *replyTo*, en la cual se puede especificar a que blockchain, contract y operation enviar la respuesta.

Figura 11: Formato del evento.

La otra parte destacable es la sección data dónde viajan los parámetros utilizados para ejecutar los métodos de los smarts contracts. El resto de los parámetros son orientadas a debug. En la tabla 5 se puede ver un detalle de cada parámetro.

Parámetro	**Descripción**
messageID	Un guid generado cómo ID del evento. Este ID se genera en el conector correspondiente con la función randomUUID de la librería Crypto
correlationIdentifier	Indica el guid (messageID) del mensaje al cual está respondiendo
timestamp	Marca de tiempo de la generación del evento. Se genera en el conector correspondiente
source.blockchain	La blockchain origen del evento
target.blockchain	La blockchain destino del evento
target.contract	El smart contract destino del evento
target.operation	La operación a ejecutar en el smart contract de la blockchain destino
replyTo.blockchain	Opcional. Indica a que blockchain se debería enviar la respuesta

replyTo.contract	Opcional. Indica a que smart contract iría la respuesta dentro de la blockchain definida en replyTo.blockchain
replyTo.operation	Opcional. Indica la operación a la cual enviar la respuesta dentro de la blockchain definida en replyTo.blockchain
data	En esta sección vienen los parámetros de la función del smart contract a ejecutar. Tienen que estar en el mismo órden que son esperados por la función.

Tabla 5: Descripción de parámetros del evento.

El formato de la versión anterior de BIG solo contenía los parámetros targetBlockchain, targetOperation y la sección de data.

5.2.2. Interacción con la red Fabric

Dado que Fabric es una red permisionada, para la interacción con ésta es necesario contar con permisos de acceso. Primeramente se requiere el usuario con permisos concedidos que se utilizará para el acceso a la red. Asimismo, se requiere el canal al que tiene acceso este usuario. Las credenciales de acceso del usuario se encuentran en su billetera, por lo que es necesario suministrar la ruta a ella. Por último se suministra un archivo json con todos los datos de conexión a la red, indicando el peer al cual se conectará el Gateway con el mencionado usuario para levantar las transacciones. Todos los datos mencionados se configuran en el archivo de configuración del Gateway para la red Fabric. Un ejemplo de este archivo se puede ver en la figura 12.

config.json 506 bytes — Open in Web IDE | Replace | Delete

```
{
    "router": {
        "endpoint": "127.0.0.1",
        "port": 3000
    },
    "blockchain": {
        "connectorHost": "127.0.0.1",
        "connectorPort": 3001,
        "fabricUser": "user1",
        "fabricChannel": "mychannel",
        "fabricContract": "estacioneco",
        "eventId": "eventToConnector",
        "walletPath": "../../fabric/fabric-blockchain/estacionEco/wallet",
        "connectionJSONPath": "../../fabric/fabric-blockchain/first-network/connection-org1.json"

    }
}
```

Figura 12: Configuración de la red Fabric en el Gateway

5.2.3. Interacción con la red Ethereum

En el caso de Ethereum no es necesaria tanta configuración, como se puede ver en la figura 13, pero si es necesaria la dirección a una billetera para uso exclusivo de BIG (bigWalletAddress).

Figura 13: Configuración de la red Ethereum en el Gateway

Para poder interactuar con los smart contracts en Ethereum, es necesario el pago de una comisión, lo que se conoce como *gas fee*. Dicha comisión se calcula en base al costo computacional que tiene que realizar la operación. Habitualmente el costo se calcula en la extensión MetaMask y el usuario hace el pago pero para esto es necesario el uso de un navegador o que exista una interfaz con la cual interactuar.
En el caso de BIG, esta interacción sucede en código por lo que no existe una interfaz, entonces es necesario que BIG pueda firmar las transacciones que se envían a la red de Ethereum. Para esto, se definió que posea una billetera propia, con la cual poder firmar y pagar las transacciones que se envíen desde Fabric hacia Ethereum. La billetera tendrá que ser cargada con saldo antes de empezar a utilizar BIG y es responsabilidad de quien publique al gateway.

5.2.4. Sistema de acciones para procesamiento de pagos

Dado que Ethereum es una red pública resulta fundamental resguardar la clave privada del usuario (necesaria para la firma de las transacciones), evitando que BIG la conozca o la guarde en todo momento. Para lograr esto fue necesario idear un sistema de pagos que permite precargar la información del mismo, lo que se llamará *Acción*. Esta información es cargada desde la red de Ethereum, utilizando la extensión MetaMask para la firma de la transacción, para que luego pueda ser confirmada en la red de Fabric.

Para esto es necesario el desarrollo de un smart contract en Ethereum, y es necesario que BIG sea el que lo despliegue en la red para así ser el dueño del

mismo y limitar el acceso. El smart contract debe tener dos operaciones: *registrar* y *liberar*. La primera se encarga de guardar la información de las acciones en memoria y también de los ethers. La segunda se encarga de liberar el pago de la acción. Los datos a guardar en memoria son el monto en ethers del pago y la dirección de la billetera a donde tienen que ir dicho monto.
El flujo del pago se inicia consumiendo la primera operación, donde se guarda la dirección de la billetera del destinatario, la cantidad de ethers de la transacción y la dirección de la billetera del cliente. Además los ethers quedan bloqueados en el smart contract. Solo puede existir una acción registrada por cada billetera de cliente. Luego, una vez que se realice la operación, se debe invocar a la segunda operación, que libera la transferencia de los ethers a la billetera del destinatario, y se culmina con el proceso de compra.

La operación *liberar* se restringe a que sea invocada exclusivamente por el dueño del mismo, léase BIG, para así evitar que los pagos sean liberados directamente desde Fabric antes de liberar la energía.

Del lado de Fabric es necesario implementar una operación que se encargue de liberar el producto y de emitir un evento hacia Ethereum que consuma la operación *liberar*.

En la figura 14 se puede ver un diagrama de secuencia del proceso mencionado.

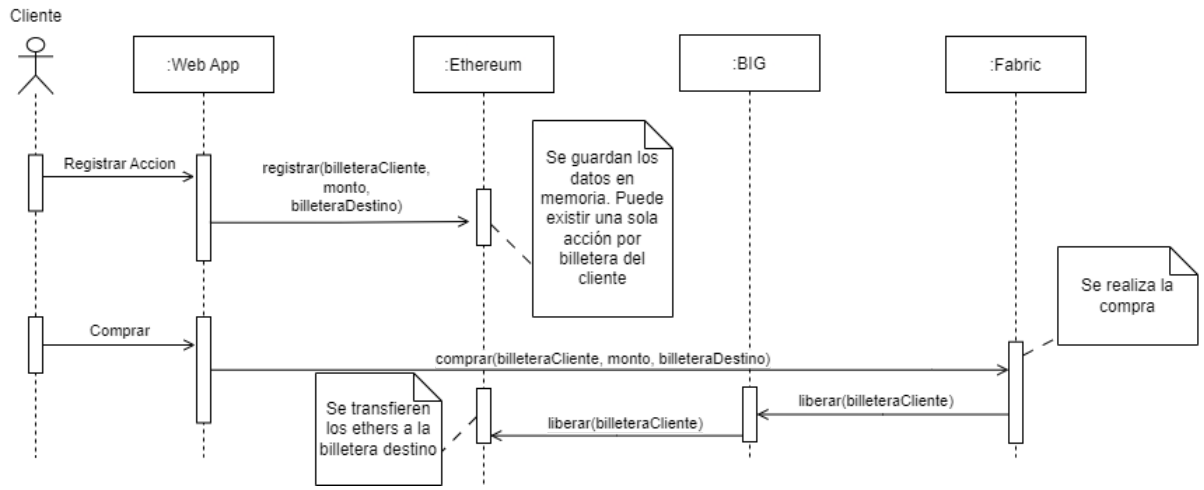

Figura 14: Registro y liberación de una acción

5.3. Mejora realizada a BIG

En esta sección se analizará la mejora realizada al componente BIG en su versión anterior.

5.3.1. Generalización de los Conectores

Si bien los conectores son piezas de código que se pueden desarrollar a medida para cada implementación, en este proyecto se buscó generar conectores genéricos e independientes del caso de uso. Para el caso de Fabric es necesario instanciar un proceso del conector por cada red que quiera interactuar con el Gateway. En el caso de Ethereum, la generalización se logra con una única instancia, que es capaz de escuchar los eventos y consumir operaciones de cualquier smart contract configurado en el conector. Para lograr esto se requiere de la carga de los archivos ABI (descriptores de los smart contracts en Ethereum). Se implementó un mecanismo que recoge todos estos archivos desde una carpeta y carga en el conector, sus smarts contracts y sus eventos, sin la necesidad de especificar cada uno en el código. Esta característica es analizada más en detalle en el capítulo 6.

5.4. Operaciones atómicas

El diseño de las operaciones atómicas en este proyecto está basado en la idea de los atomics swaps presentados en la sección 2.3.3. El intercambio atómico que se hace es el de cierta cantidad de energía a cambio de un cierto pago a realizar por el cliente.

Es necesario el desarrollo de un smart contract en Ethereum con 5 operaciones disponibles (*iniciar*, *validar*, *verificar*, *cerrar* y *reembolso*) y un smart contract en Fabric con dos operaciones (*iniciar* y *verificar*). En la figura 15 puede verse el diagrama de secuencia con las operaciones implementadas.

Como primer paso, el cliente tiene un secreto “*s*” y genera el hash “*h*” de “*s*”. Este es enviado, junto con los ethers y el time lock “*t*”, al consumir la operación *iniciar* del smart contract de Ethereum. El time lock es el tiempo máximo de espera antes de devolverle los ethers al cliente. Al consumir la operación se emite un evento hacia la operación *iniciar* de la red de Fabric y en el mismo viaja el hash “*h”* junto con otros datos de la operación. Luego de consumir la operación *iniciar* de Fabric, se envía un mensaje a Ethereum, que invoca a la operación *validar*, para que el cliente valide la transacción.

Una vez que el cliente valida la transacción, se envía el secreto “*s*” a Ethereum, invocando la operación *verificar*. Dicha operación emite un evento hacia Fabric, con el secreto “*s*”, que invoca a la función *verificar* de esta otra red. En Fabric se le aplica la misma función de hash al secreto “*s*” y tiene que ser igual al hash “*h*” recibido al comienzo. En caso afirmativo se realiza la carga de la energía y se envía un mensaje, con el secreto “*s*”, a Ethereum, que consume la operación *cerrar*. En Ethereum, al recibir dicho mensaje, se hace la misma validación (aplicar hash a “*s*” y comparar con “*h*” recibido al comienzo), y si está correcto se transfieren los ethers a la billetera del destinatario. La operación *reembolso* del smart contract de Ethereum,

se invoca en caso de que el mensaje de validar no llegue al cliente o el mismo no valide.

Los detalles de implementación del procedimiento se verán en la sección 6.3.

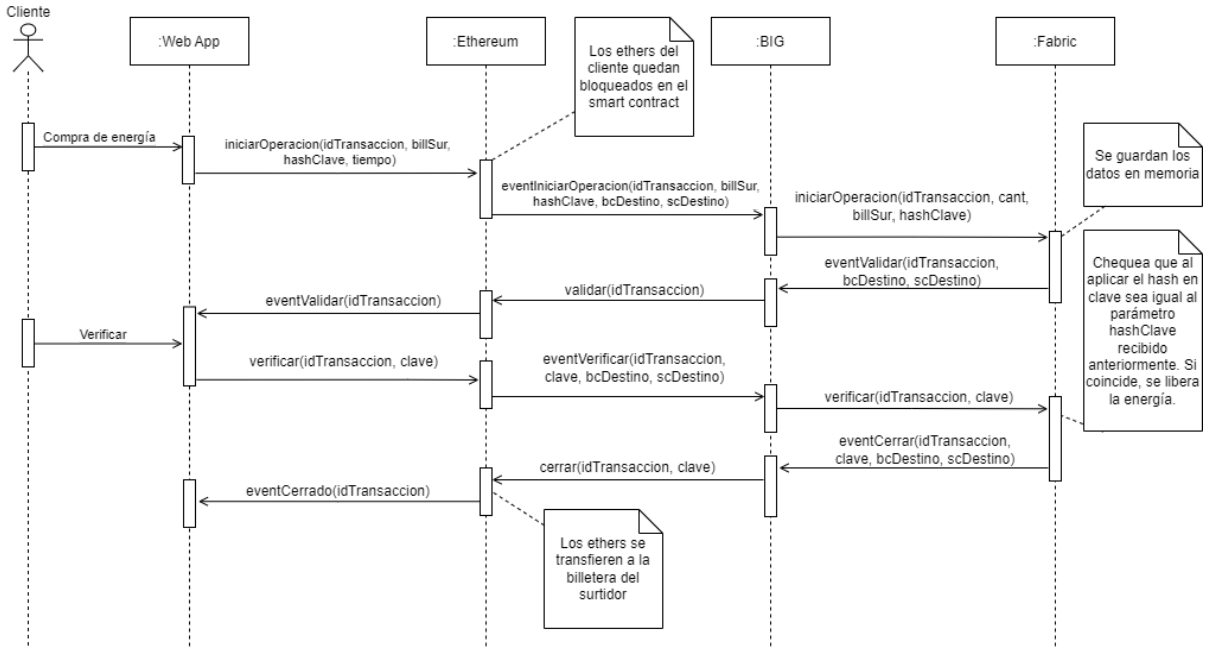

Figura 15: Diagrama de secuencia operación atómica exitosa

El time lock funciona cómo timer de la transacción, y en caso de que se venza, la transacción es anulada y se pasa al estado de “Expirado”, dónde los ether son devueltos al cliente. La web app es la encargada de chequear que haya pasado el tiempo y así solicitar el reembolso. Cabe resaltar que este método garantiza que se entregue la energía y se realice el pago. En la figura 16 se muestra el caso de que el cliente no confirma la transacción, en el cual, se termina realizando el reembolso y la energía no es transferida. Este reembolso solo se puede solicitar antes de la confirmación del cliente ya que en la misma, la transacción se cambia de estado y se bloquea esta funcionalidad. Esto garantiza que no se devuelvan los ethers al cliente mientras se libera la energía y además Fabric está obligado a liberarla sino no recibe el pago.

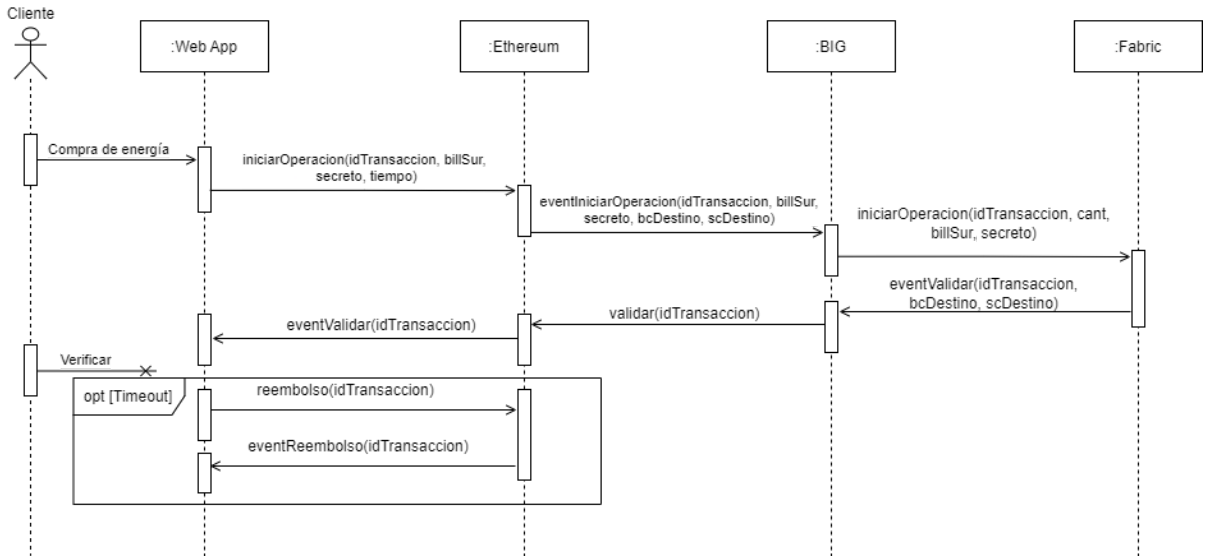

Figura 16: Diagrama de secuencia cliente no confirma

6. Implementación

En este capítulo se analizan detalles de la implementación, tanto del Gateway extendido para incorporar la red Ethereum, cómo del escenario de uso propuesto en el proyecto. En la sección 6.1 se detalla la implementación del Gateway. En la sección 6.2 se verán detalles de la implementación del escenario. En la sección 6.3 se analizan detalles de implementación de las operaciones atómicas. Para finalizar, en la sección 6.4, se presentan algunos desafíos de la implementación.

6.1. Implementación del Gateway

En esta sección se indicarán las tecnologías utilizadas en la implementación del gateway. Luego, se establecerán las precondiciones que se deben cumplir para poder utilizarlo y se mencionan las mejoras realizadas a la versión anterior del mismo.

6.1.1. Tecnologías

Los componentes del Gateway están implementados en NodeJs, tanto el Router cómo los conectores, como se puede ver en la figura 9. La comunicación interna entre ellos, se hace mediante la publicación de APIs REST por cada uno de estos. Cada componente expone sus endpoints HTTP que son consumidos por los demás.
Las APIs son publicadas usando la librería “http” de Node. La interacción entre el Conector de Ethereum y la red se realiza mediante una librería de Node llamada Web3. Ésta misma librería pero para JavaScript es la que será utilizada por la página web. La comunicación entre el Conector de Fabric y la red se realiza utilizando un SDK provisto por Hyperledger, el cual utiliza gRPC como protocolo de comunicación [46].

6.1.2. Precondiciones

El Gateway desarrollado en este proyecto requiere de algunas condiciones para funcionar. Es necesario que la plataforma de blockchain a incorporar pueda emitir eventos hacía el Gateway. También debe disponer de un mecanismo (p. ej. un SDK) por el cual se puedan invocar operaciones de los smarts contracts de la red.

Para el conector de Ethereum se deben colocar los archivos ABI (extensión json), de los smarts contracts a usar en la solución, en la carpeta ethereum-connector/ABI, ya que la automatización de la instanciación de los contratos en el conector, se toman desde allí.

El smart contract de Fabric, a usar en el caso de uso, debe extender a la clase llamada RemoteEventHandlerContract, que a su vez, extiende a la clase Contract del SDK. Esta clase personalizada contiene un método que realiza la emisión de los eventos, que son escuchados por el conector. En la figura 17 se puede observar la extensión de la clase. Por otro lado, el nombre del evento que va a ser enviado por la red al conector se debe configurar en el archivo de configuración (config.json), esto se puede ver más en detalle en el anexo IV.

```
class EstacionEcoAtomicContract extends RemoteEventHandlerContract {

  async initLedger(ctx) {
    console.info('============= START : Initialize Ledger ===========');
```

Figura 17: Extensión del manejador de eventos

6.1.3 Conector Ethereum

En esta nueva versión se agregó el conector con la red de Ethereum. El mismo fue desarrollado con la misma estructura que el conector de Fabric y se agregó una funcionalidad de generalización.

Cómo se establece en la sección 5.3.1 del apartado de diseño, se agregó al conector de Ethereum la posibilidad de cargar los archivos ABIs al código de forma genérica para no acoplar fuertemente la solución al caso de uso. Para esto, el requisito es dejar los archivos con extensión json que genera el deploy de los smarts contract en la red, en la carpeta “ABI” del conector Ethereum.

Luego, en código se recorre esta carpeta levantando los artefactos que necesita la librería Web3 para funcionar. Además, levantan todos los listeners de los eventos posibles en cada smart contract. En la figura 18, vemos la función *loadABI* que carga los ABIs y la función *listenBlockchainEvents* que recorre en busca de eventos y prepara la escucha para estos.

```
    async loadABI(){
        glob.sync( './ABI/*.json' ).forEach( function( file ) {
            var aux = require( path.resolve( file ) );
            contracts.push(aux);
          });
          await this.listenBlockchainEvents();
          await this.startRouterEndpoint();

    }

    async listenBlockchainEvents() {
        const web3 = new Web3(config.blockchain.ethereumHost)
        const networkId = await web3.eth.net.getId();
        contracts.forEach((contract) => {
          var deployedNetwork = contract.networks[networkId];
           var myContract = new web3.eth.Contract(
              contract.abi,
              deployedNetwork && deployedNetwork.address,
          );

        myContract.events.allEvents()
           .on('data', event => this.eventReceivedFromEthereum(event))
            .on('changed', changed => console.log(changed))
            .on('error', err => { throw err })
            .on('connected', str => console.log(str))
        }          );
    }
```

Figura 18: Funciones de carga generalizada de ABIs.

6.1.4. Mejoras en la implementación

A continuación se presentan las mejoras realizadas a la versión anterior del gateway. La primera consiste en separar el archivo de configuración central. La siguiente consiste en generalizar el conector de Fabric y la última consiste en la posibilidad de pasar el archivo de configuración por parámetro.

Separación de archivos de configuración

En la versión anterior se mantenía un único archivo de configuración centralizado. El mismo era leído tanto por el router como por los conectores de las distintas blockchain. Se procedió a separarlo en un archivo de configuración por componente del gateway y esto permite la ejecución de varias instancias de los distintos componentes.

Generalización del conector de Fabric

Para mejorar el desacoplamiento del conector de Fabric, en la versión anterior, se pasaron varias configuraciones desde el código al archivo de configuración. En éste se establecen todos los parámetros utilizados para uso de la red Fabric correspondiente. De esta forma, teniendo cada archivo de configuración para la red Fabric que corresponda, se puede levantar un conector para cada una de éstas. Además, es posible invocar a un smart contract configurable, algo que en la versión anterior del conector estaba hardcoded. Esta mejora junto con la de la configuración, permite el uso del conector de Fabric en cualquier escenario.

Archivo de configuración por parámetro

Por defecto los componentes del gateway (router y conectores) toman el archivo de configuración "*config.json*" que está alojado en la carpeta del mismo. Se agregó la posibilidad por código, en los componentes implementados en este proyecto, para qué también se pueda pasar dicho archivo por parámetro a la hora de levantar el componente. En caso de proporcionar un ruta con el archivo de configuración se ignora el que se encuentra alojado en la carpeta y se toma ese archivo. Para esto se debe ejecutar el comando como se muestra en la figura 19.

```
emi@DESKTOP-M298NBF:/mnt/c/Fing/BIG3/gateway/ethereum-connector$ npm run connector config.json


> ethereum_connector_blockchain_gateway@1.0.0 connector /mnt/c/Fing/BIG3/gateway/ethereum-conn
ector
> node ConectorEthereum.cjs "config.json"

Starting up ethereum connector...
```

Figura 19: Comando ejecución conector con ruta de archivo de configuración.

6.2. Implementación del escenario propuesto

En esta sección se indicarán las tecnologías utilizadas en la implementación del escenario propuesto. Luego, se comentarán las funcionalidades disponibles en la aplicación web.

6.2.1. Tecnologías del escenario

Para la aplicación web que oficia de interfaz con el usuario se utilizó HTML con scripts implementados en JavaScript que en este proyecto llamamos DApps. La Dapp de Ethereum utiliza la librería Web3 para el lenguaje JavaScript y la DApp de Fabric usa el mismo SDK provisto por Fabric para la comunicación. Dado que el SDK de Fabric no puede ser importado directamente en JavaScript, se utiliza una

API REST implementada en NodeJS, que si utiliza el SDK y se comunica con la red. La API es publicada utilizando la librería Express [47] de NodeJs y es llamada en JavaScript con Axios [48]. A su vez, la API es capaz de enviar los eventos recibidos desde Fabric, hacia la web, utilizando WebSockets. En la figura 20 se puede ver el detalle de los componentes y las comunicaciones.

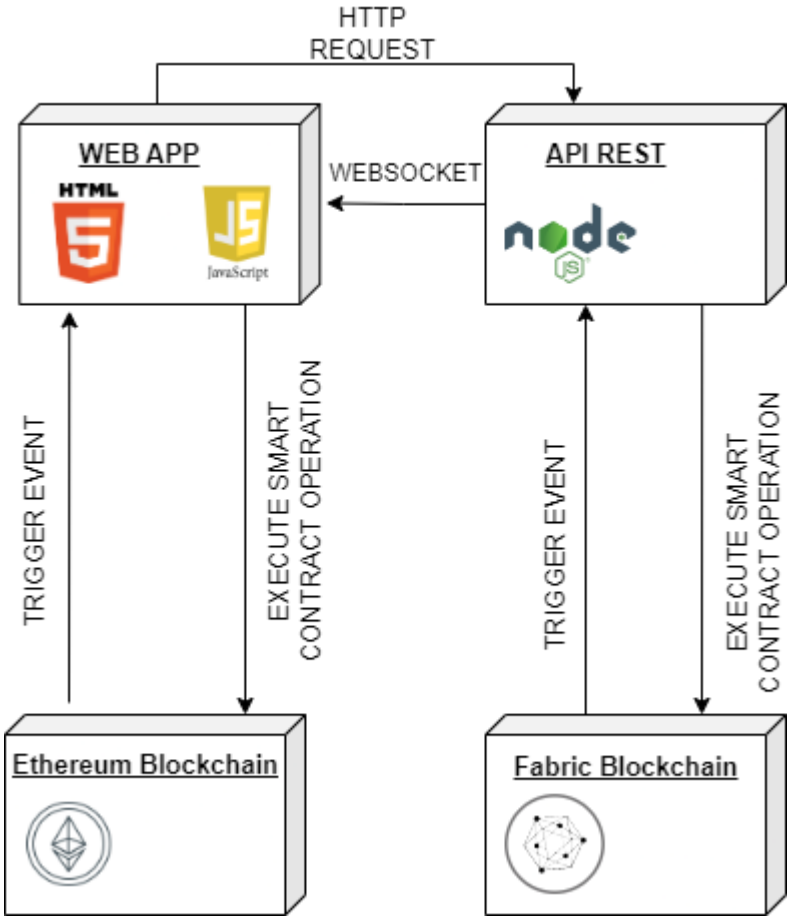

Figura 20: Componentes de la web y sus tecnologías

Los smarts contracts desarrollados para el caso de uso en la red Ethereum están en lenguaje Solidity y los de la red Fabric en NodeJS. Para el caso de uso, se utilizó la red de prueba Ganache [49] cómo red Ethereum. Para gestionar las billeteras de los clientes en la aplicación web se utilizó la extensión del navegador llamada MetaMask [50].

6.2.2. Interacciones de la aplicación web

Para la implementación del caso de uso propuesto en la sección 3.3, se desarrolló una aplicación web que se encarga de gestionar la interacción del sistema con el usuario. También se implementaron dos scripts desarrollados en lenguaje JavaScript que se utilizan para la interacción de la aplicación web con las blockchains. En la figura 21 se puede apreciar un diagrama más general de cuáles son los componentes y cómo interactúan entre ellos.

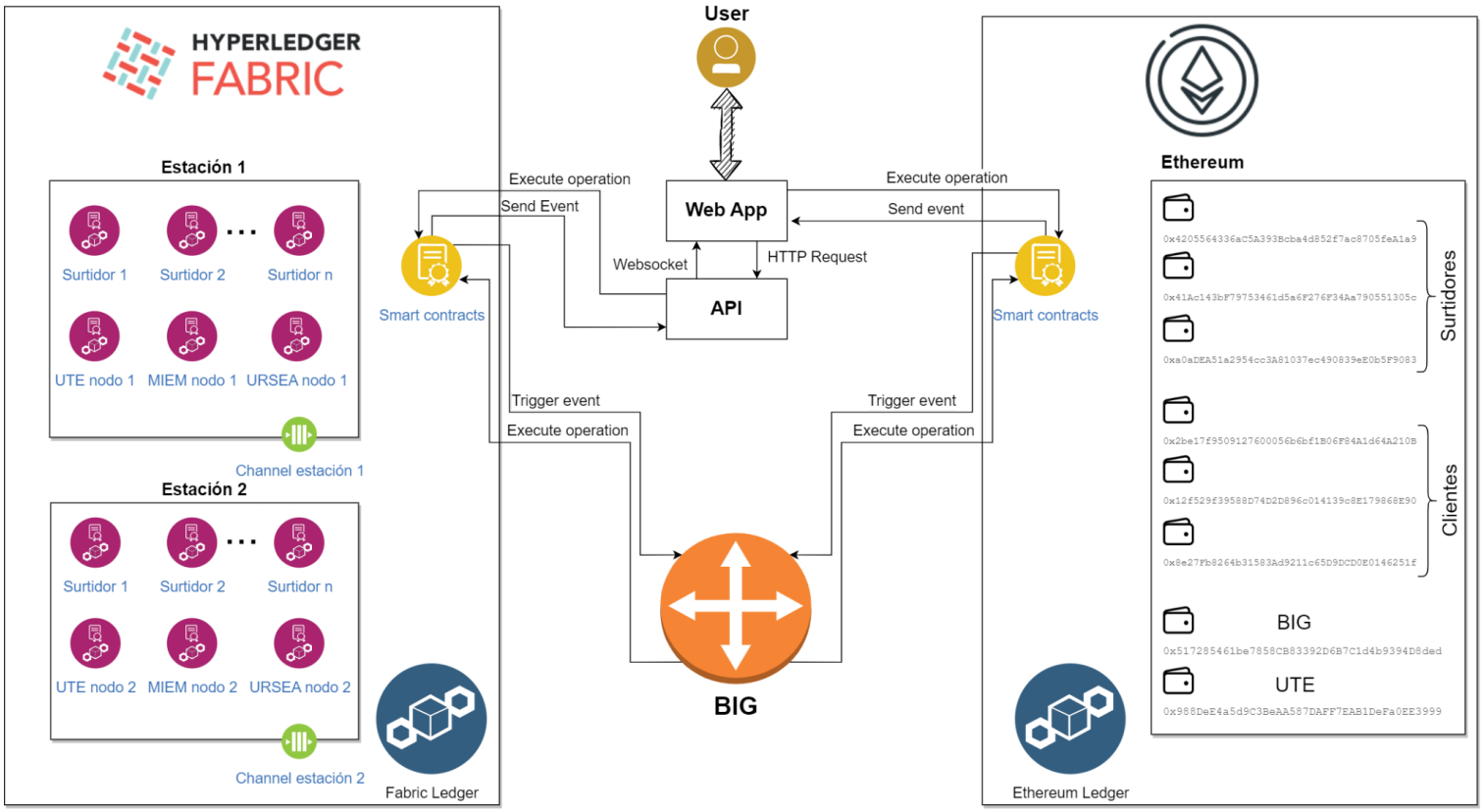

Figura 21: Diagrama de componentes e interacciones completo.

Para la interacción con Hyperledger Fabric fue necesario incluir un servidor nodeJS que expone una API REST, la cual será consumida por la web app. Además se le agregó una conexión WebSockets para que sea capaz de informar el resultado de las operaciones a la web app. Los métodos que expone la API están detallados en la tabla 6. En Ethereum se implementaron 3 smart contracts, utilizando el lenguaje Solidity, los cuales se encuentran detallados en el anexo V. Estos son invocados por la web app al iniciar los casos de uso, como también por BIG, cuando llegan mensajes desde Fabric. Para Hyperledger Fabric fue necesario la implementación de un único smart contract con varias operaciones que también se encuentran detallados en el mismo anexo. Este smart contract es invocado por la API REST y también por BIG al enviar mensajes desde Ethreum.

Ruta	Método HTTP	Descripción
/compraFabric	POST	Consume la operación *compraFabric* del smart contract de Fabric
/obtenerAccion	POST	Consume la operación *obtenerAccion* del smart contract de Fabric

Tabla 6: Métodos expuestos por la API.

6.2.3 Descripción de la aplicación web

Se creó una interfaz web implementada en html que utiliza scripts, en lenguaje JavaScript, con la lógica de negocio y la interacción con las redes. La interfaz puede verse en la figura 22.

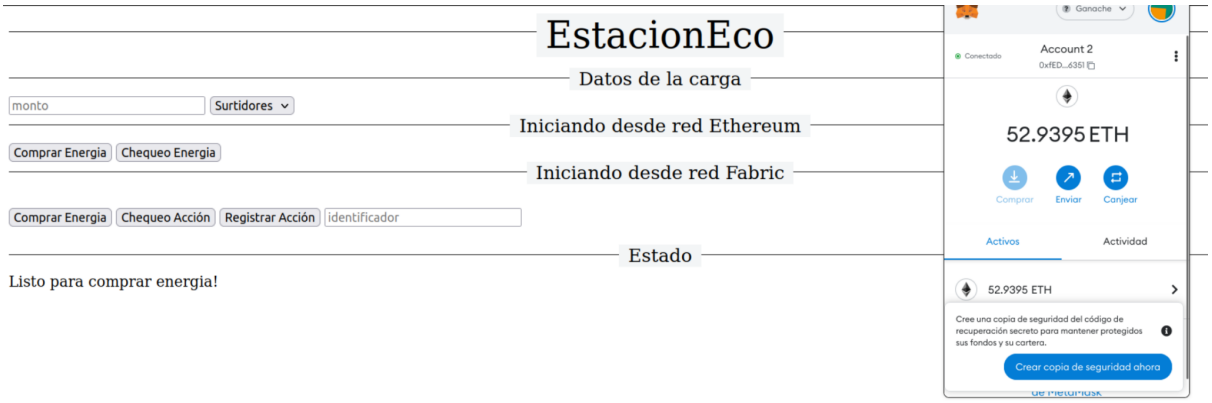

Figura 22: Interfaz web de la DApp

La interfaz se divide en una sección dónde se cargan los datos de la operación a realizar, "Datos de la carga", y dos secciones dónde se encuentran los botones y opciones a utilizar interactuando con cada red, "Iniciando desde red Ethereum" e "Iniciando desde red Fabric".
En la sección “Datos de la carga” se pueden ingresar el monto de energía a cargar y un combo para seleccionar el surtidor a utilizar. Se está utilizando la relación una unidad de energía equivale a un ether.

En la sección “Iniciando desde red Ethereum” se puede consultar la energía disponible del surtidor seleccionado en la sección anterior. Para esto se consume la operación *chequeoEnergia* del smart contract *ChequeoEnergia* de Ethereum. Al ejecutar esta operación, es necesario seleccionar la billetera del usuario usando la extensión MetaMask. Esta operación enviará un mensaje a Fabric, a través de BIG, que consumirá la operación *obtenerEnergia*. Luego, la respuesta consumirá la operación *devolverResultado (*de Ethereum), que emitirá un evento a ser capturado por la web app, donde se indicará la cantidad de energía disponible. Las interacciones pueden verse detalladas en la figura 23.

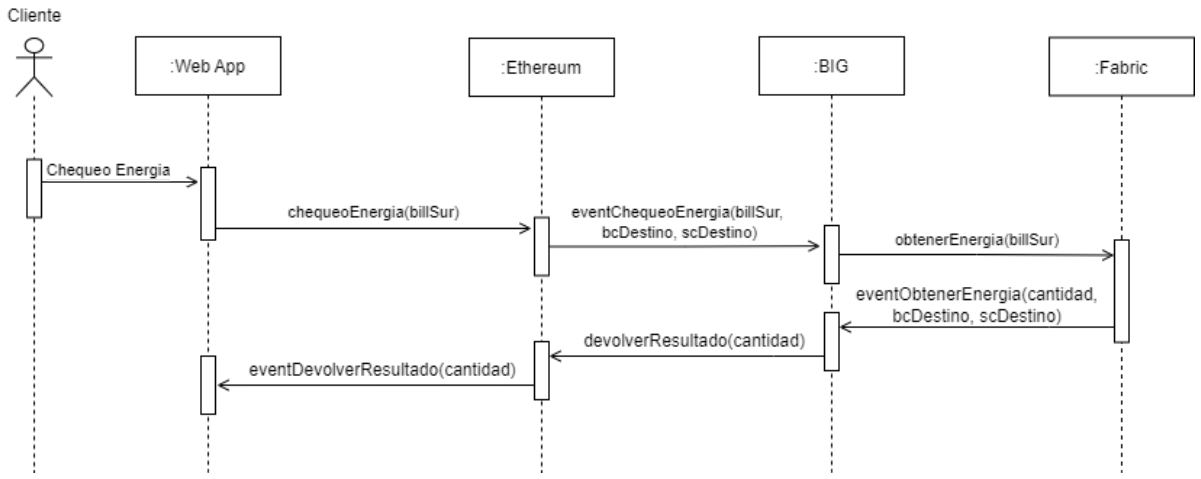

Figura 23: Chequeo de energía del surtidor

En la sección “Iniciando desde red Ethereum” también se puede ejecutar una compra atómica iniciando desde dicha blockchain, utilizando los datos de la sección anterior. El funcionamiento es similar, se inicia el flujo en la red de Ethereum, previo pago de la comisión y el monto de energía que se quiere comprar, y los mensajes viajan a través de BIG. Un detalle de las interacciones de esta funcionalidad se encuentra en la figura 15.

En la sección de “Iniciando desde red Fabric” se puede hacer una compra simple, no atómica, iniciando desde dicha red. Para esto es necesario, primero, crear una Acción utilizando el botón "Registrar Acción". La opción anterior registra una acción de carga en la red Ethereum con los datos de la sección "Datos de carga". La interacción puede verse en la figura 24.

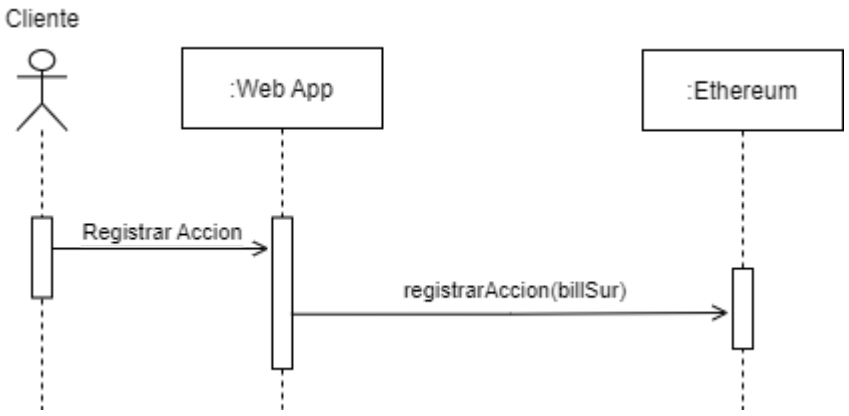

Figura 24: Registro de una acción

Con el botón "Chequeo de Acción" se puede comprobar si hay una acción de carga de energía pendiente para la billetera ingresada en el campo "*identificador*". La interacción se puede ver en detalle en la figura 25. Si existe una acción cargada, es posible confirmarla, y liberar la energía y ethers, utilizando el botón “Comprar

Energía” de esta sección. En la figura 26 se puede ver un detalle de las interacciones. A diferencia de las operaciones de la sección “Iniciando desde red Ethereum”, que directamente consumen una operación de la red, estas operaciones hacen una solicitud HTTP a la API REST. En caso de la operación “Chequeo de Acción”, se hace una solicitud POST a la ruta obtenerAccion, mientras en el caso de la operacion “Comprar Energia”, se hace un POST a la ruta compraFabric.

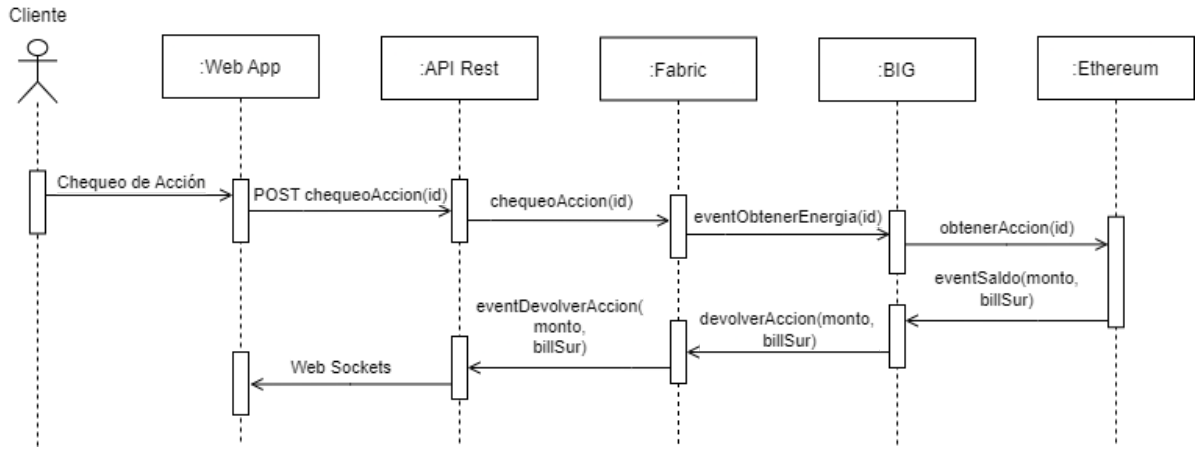

Figura 25: Chequeo de acción

Por último se registra debajo una sección informativa con el estado de aplicación y las respuestas obtenidas al utilizar las distintas funcionalidades. A la derecha de la imagen se puede ver la extensión MetaMask utilizado en el navegador.

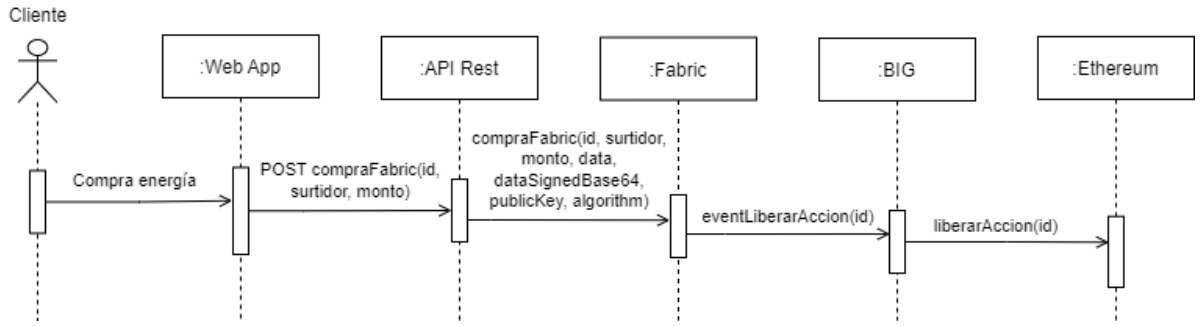

Figura 26: Comprar energía desde Fabric

En el anexo VI se pueden ver algunos detalles de la implementación de la aplicación web.

6.3. Operaciones atómicas

A continuación se verá en detalle la implementación de las operaciones atómicas mencionadas en la sección 5.4. En el anexo V se detallan los smart contracts de Ethereum y Fabric involucrados en estas operaciones.

En Ethereum, la estructura de datos de la compra se guarda en memoria y puede verse en la figura 27. En Fabric también se mantiene una estructura similar en memoria.

```
struct Compra {
    uint256 timelock;
    uint256 monto;
    address payable billCli;
    address payable billSur;
    bytes32 hash;
}
```

Figura 27: Estructura de datos de la compra

En la figura 28 se pueden ver los estados posibles por los cuales pasa la compra. Mientras que en la figura 29 se puede ver la máquina de estados asociada donde cada arista indica que operación del smart contract hace el cambio de estado.

```
enum Estados {
    INVALIDO,
    ABIERTO,
    PROCESANDO,
    CERRADO,
    EXPIRADO
}
```

Figura 28: Estados de la compra

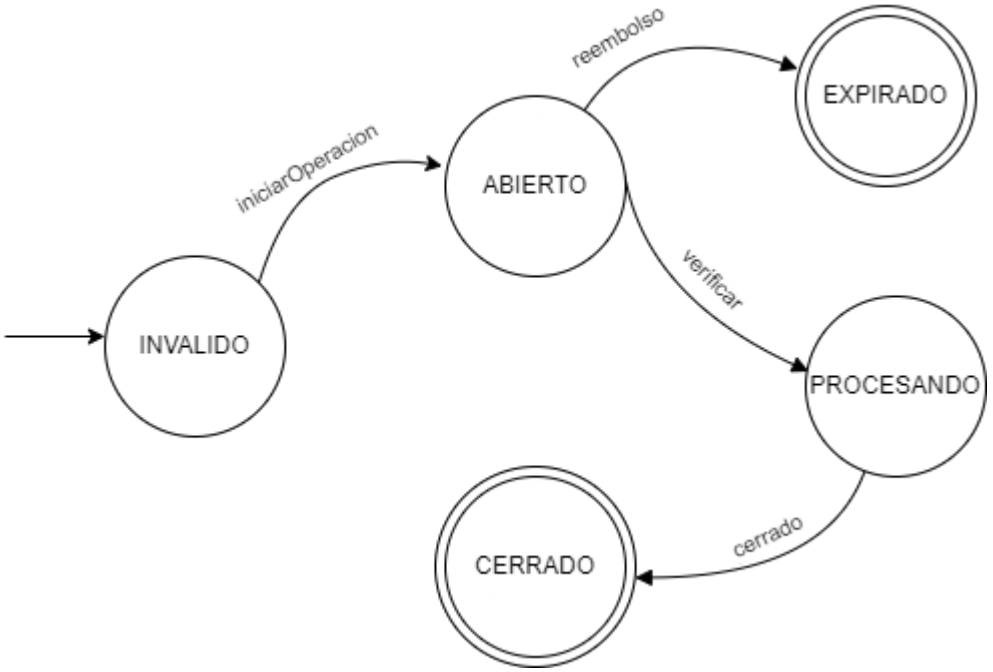

Figura 29: Máquina de estados

Otro punto general para el smart contract son los prerrequisitos de las funciones, que se definen con el atributo *modifier*. Dichos prerrequisitos se pueden observar en la figura 30.

```
modifier chequeoInvalido(bytes32 _idTransaccion) {
    require(estadoCompras[_idTransaccion] == Estados.INVALIDO);
    _;
}

modifier chequeoAbierto(bytes32 _idTransaccion) {
    require(estadoCompras[_idTransaccion] == Estados.ABIERTO);
    _;
}

 modifier chequeoProcesando(bytes32 _idTransaccion) {
    require(estadoCompras[_idTransaccion] == Estados.PROCESANDO);
    _;
}

modifier chequeoCerrado(bytes32 _idTransaccion) {
    require(estadoCompras[_idTransaccion] == Estados.CERRADO);
    _;
}

modifier chequeoExpirado(bytes32 _idTransaccion) {
    require(now >= compras[_idTransaccion].timelock);
    _;
}

modifier chequeoClaveSecreta(bytes32 _idTransaccion, bytes memory _clave) {
    require(compras[_idTransaccion].hash == keccak256(_clave));
    _;
}
```

Figura 30: Prerrequisitos

Como se vió en la figura 15, la primera operación que se invoca es *iniciarOperacion*. En la figura 31 se puede ver la implementación. La función se marca con el atributo *payable* lo que permite enviar los ethers al contrato. A su vez se indica que debe cumplir el prerrequisito *chequeoInvalido* (chequea que la transacción con el id enviado por parámetro esté en estado inválido). Esta función guarda en memoria los datos de la compra, asociados al id de transacción, y la cambia a estado abierto. Luego emite un evento hacia Fabric, a la operación “iniciarOperacion”. Esta operación se encarga de guardar en memoria el hash, la dirección de la billetera del surtidor y la cantidad de energía. Además emite un evento hacia la operación *validar* del smart contract *EnergiaAtomico* de Ethereum. El evento llega a Ethereum y se emite un evento de tipo *eventValidar* hacia la web app para que el usuario valide la transacción. La validación la hace el usuario aceptando la operación en la extensión MetaMask. Luego de esto se consume la operación *verificar*, y se pasa por parámetro el id de transacción junto con la clave original sin aplicarle el hash. La implementación de la operación se puede ver en la figura 32. Se tienen que cumplir

dos condiciones para que la verificación sea exitosa, primero que la compra asociada a ese id de transacción esté en estado abierto, y luego, que la clave que se envía por parámetro, al aplicarle el hash, sea igual al que se envió originalmente. Si se cumplen entonces se pasa la compra a estado procesando y se emite un evento hacia Fabric, operación *verificar*, con los mismos parámetros recibidos. Del lado de Fabric, también se chequea que al aplicarle el hash a la clave sea igual al hash recibido al comienzo, y en caso afirmativo, se libera la energía y se envía un evento a Ethereum. El evento va dirigido a la operación *cerrar* y también envía los mismos parámetros recibidos. La implementación de esta operación puede verse en la figura 33. Se chequea que la compra esté en estado procesando y que al aplicarle el hash a la clave sea igual al hash del comienzo. En caso de cumplirse, se transfieren los ethers a la billetera del surtidor y se marca la compra como cerrada. Luego se envía un evento, que escucha la web app, para mostrar en la interfaz que se terminó el proceso de compra.

```
function iniciarOperacion(bytes32 _idTransaccion, address payable _billSur, bytes32 _hash, uint256 _timelock)
        public chequeoInvalido(_idTransaccion) payable {

    Compra memory compra = Compra({
        timelock: _timelock,
        monto: msg.value,
        billCli: msg.sender,
        billSur: _billSur,
        hash: _hash
    });
    compras[_idTransaccion] = compra;
    estadoCompras[_idTransaccion] = Estados.ABIERTO;

    emit eventIniciarOperacion(_idTransaccion, _billSur, msg.value, _hash,"fabric","estacioneco","iniciarOperacion");
}
```

Figura 31: Función iniciarOperacion

```
function verificar(bytes32 _idTransaccion, bytes memory _clave)
    public chequeoAbierto(_idTransaccion)
           chequeoClaveSecreta(_idTransaccion, _clave)
{
    estadoCompras[_idTransaccion] = Estados.PROCESANDO;
    emit eventVerificar(_idTransaccion, _clave,"fabric","estacioneco","verificar");
}
```

Figura 32: Función verificar

```
function cerrar(bytes32 _idTransaccion, bytes memory _clave)
        public chequeoProcesando(_idTransaccion)
               chequeoClaveSecreta(_idTransaccion, _clave)
{
    Compra memory compra = compras[_idTransaccion];
    estadoCompras[_idTransaccion] = Estados.CERRADO;
    compra.billSur.transfer(compra.monto);
    emit eventCerrado(_idTransaccion, "dapp","none","cerrar");
}
```

Figura 33: Función cerrar

6.4. Desafíos de la implementación

A continuación se mencionan los desafíos que se encontraron durante la implementación.

Tecnologías utilizadas
Las tecnologías utilizadas no eran dominadas por ninguno de los integrantes. Entre ellas se destacan NodeJS para el desarrollo de los componentes del gateway, como también para el desarrollo de los smart contracts en Fabric. Por otro lado se encuentra Solidity, para el desarrollo de los smart contracts en Ethereum. En el caso de Solidity, es un lenguaje reciente por lo que los ejemplos que se pueden encontrar en la web están acotados. Esto terminó impactando en los tiempos proyectados de implementación.

Desarrollo de smart contracts
Para el desarrollo de estos smart contracts se cuenta con un conjunto limitado de herramientas, ya que al ser redes deterministas, no se pueden realizar ciertas tareas que son comunes en un entorno de desarrollo normal. Por ejemplo, no es posible generar un guid aleatorio dentro de la misma lo que imposibilita la creación del id de mensaje y esto tiene que realizarse en el gateway. Por otro lado, para el caso de Ethereum, el uso de recursos computacionales tiene costo, entonces es necesario ser eficiente en el diseño e implementación de los mismos.

7. Evaluación de la solución

En este capítulo se verán las distintas evaluaciones que se realizaron sobre la solución. Se comenzará mostrando los resultados obtenidos al realizar una prueba de carga sobre la solución, en la sección 7.1. Luego, en la sección 7.2, se analizará el gasto de gas que se realiza en cada una de las operaciones implementadas. Y para terminar, en la sección 7.3, se comentarán las fortalezas y debilidades que fueron encontradas en la solución implementada.

7.1. Pruebas de performance

Para realizar las pruebas se utilizó la herramienta Apache jMeter, la cuál permite simular a varios usuarios concurrentes, realizando operaciones sobre cierto escenario y plataforma.

Escenario

El escenario de la prueba fue implementado con un usuario que consume una función del chaincode de Hyperledger Fabric, específicamente la operación *obtenerAccion.* Esta operación envía un mensaje a la red de Ethereum, con una dirección de billetera, y en esta red se chequea si existe una acción pendiente de pago. Luego, se envía la respuesta desde Ethereum hacía Fabric, con los datos de la acción pendiente.

Hardware

La prueba se realizó en una PC con sistema operativo Windows 10 y con la funcionalidad WSL 2 (Windows Subsystem for Linux 2). Esta funcionalidad permite instalar un kernel de Linux para ser virtualizado por Windows, sin necesidad de generar máquinas virtuales. Las especificaciones de hardware de la PC se pueden observar en la tabla 7.

CPU	Intel Core I7-10750H 6 núcleos/12 hilos Frecuencia base 2.6Ghz, frecuencia máxima 5 Ghz
Memoria	32Gb DDR4 3200 Mhz
Disco duro	1Tb SSD NVMe

Tabla 7: Especificaciones PC.

Topología

En la misma PC corren las redes de Hyperledger Fabric y Ethereum (Ganache), como también la solución de BIG completa. Ésta incluye, conector de Ethereum, conector de Hyperledger Fabric y Router. Además es necesario ejecutar el servidor nodeJS que expone la API REST que se comunica con el SDK de Fabric como se mencionó en la sección 6.2.2.
Como se puede ver en la figura 34, Apache jMeter envía una petición hacia la API REST hosteada en el servidor nodeJS. Este termina consumiendo un smart contract de la red de Hyperledger Fabric. La ejecución del smart contract en la red dispara un mensaje hacia la red de Ethereum, pasando por BIG. La respuesta de Ethereum es enviada por el camino inverso.

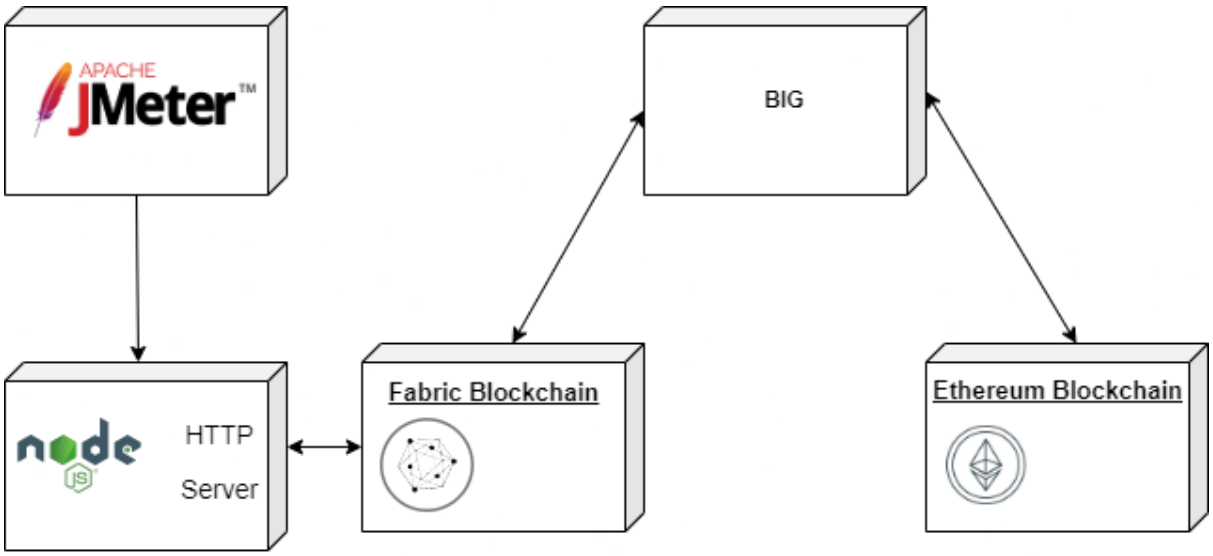

Figura 34: Diagrama de la prueba

Resultados iniciales

Los resultados iniciales no fueron satisfactorios ya que todas las transacciones fallaron al ejecutar la operación en la red de Ethereum. Para enviar un mensaje a dicha red, es necesario indicar el número de transacción de la billetera que se está utilizando. Este número es obtenido utilizando una función provista por el SDK. El problema aquí es que varias transacciones llegan al mismo tiempo, entonces, el número de transacción obtenido se repite entre transacciones, generando el error recibido.

Mejora realizada

Para solucionar el inconveniente mencionado anteriormente, se agregó un bloqueo en el código de envío a Ethereum del conector de dicha blockchain, con el cúal solo

se permite el envío secuencial de mensajes, de uno en uno. Esto permite utilizar la plataforma pero es una debilidad de la solución, como se verá en la sección 7.3.

Resultados finales

Se aplicó la mejora y se volvió a ejecutar la prueba. Inicialmente se configuraron 20 usuarios concurrentes para ejecutar el escenario mencionado anteriormente, en un tiempo de 5 minutos. Luego, se aumentó la cantidad de usuarios a 40 y se ejecutó la misma prueba con la misma duración. Los resultados detallados se pueden ver en la tabla 8.

Usuarios	Cantidad pedidos	Correctos	Porcentaje correcto	Tiempo total (mm:ss)
20	2.548	2.548	100%	08:44
40	5.139	4.073	79%	13:46

Tabla 8: Resultados de la prueba de perfomance.

El tiempo total de la ejecución indica el tiempo entre que se envió el primer mensaje de la prueba hasta que se obtuvo la respuesta del último mensaje enviado. Éste es mayor a los 5 minutos, en los cuales se envían los mensajes, por la demora que se genera en el procesamiento secuencial en el conector de Ethereum.

```
(node:4361) UnhandledPromiseRejectionWarning: Error: connection not open on send()
    at Object.ConnectionError (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/web3-core-helpers/lib/errors.js:66:23
)
    at Object.ConnectionNotOpenError (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/web3-core-helpers/lib/errors.j
s:49:21)
    at /mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/web3-providers-ws/lib/index.js:155:37
    at Map.forEach (<anonymous>)
    at WebsocketProvider._onClose (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/web3-providers-ws/lib/index.js:15
4:27)
    at W3CWebSocket._dispatchEvent [as dispatchEvent] (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/yaeti/lib/Eve
ntTarget.js:115:12)
    at W3CWebSocket.onConnectFailed (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/websocket/lib/W3CWebSocket.js:2
19:14)
    at WebSocketClient.<anonymous> (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/websocket/lib/W3CWebSocket.js:59
:25)
    at WebSocketClient.emit (events.js:400:28)
    at ClientRequest.handleRequestError (/mnt/c/Fing/BIG2.0/gateway/ethereum-connector/node_modules/websocket/lib/WebSocketClie
nt.js:227:14)
    at ClientRequest.emit (events.js:400:28)
    at Socket.socketErrorListener (_http_client.js:475:9)
    at Socket.emit (events.js:400:28)
    at emitErrorNT (internal/streams/destroy.js:106:8)
    at emitErrorCloseNT (internal/streams/destroy.js:74:3)
    at processTicksAndRejections (internal/process/task_queues.js:82:21)
```

Figura 35: Error

Como se puede observar, ya con 40 usuarios concurrentes se generan fallos de conexión con la red Ethereum. En el 21% de los mensajes se presenta un error de socket, el mismo no acepta conexiones y está cerrado, y sucede al final de la prueba. Un detalle del error puede verse en la figura 35.

En la figura 36 se puede observar la evolución del tiempo de demora en procesar cada mensaje enviado por el software de pruebas. A medida que se van enviando mensajes se empiezan a encolar cada vez más en el conector de Ethereum y esto repercute en el tiempo de espera.

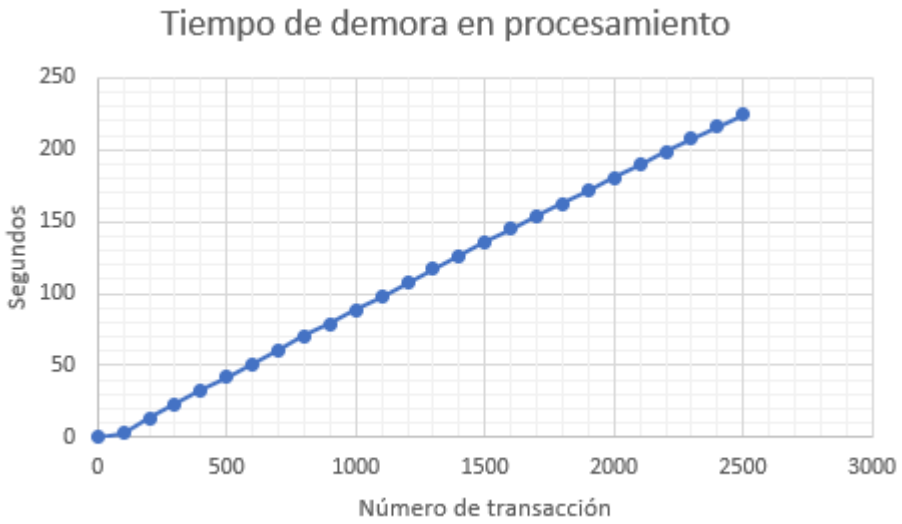

Figura 36: Gráfico evolución del tiempo de demora

La demora máxima fue de 229 segundos así que el mensaje fue enviado en cierto momento y la respuesta se obtuvo casi 4 minutos después.
El envío de mensajes entre los conectores y router no presenta demoras significativas.

7.2. Análisis del gasto de gas

Para calcular el gasto de gas se ejecutaron todas las funcionalidades presentes en la aplicación web, vista en la sección 6.2.3, y se obtuvieron los valores de gas utilizados en cada transacción.

Funcionalidad	**Gasto en gas (en miles)**	**Gasto en USD (Total)**	**Gasto en USD (BIG)**	**Gasto en USD (cliente) ***
Registrar acción	53,7	1,248	0	1,248
Chequeo surtidor	51	1,186	0,580	0,606
Chequeo acción	29,3	0,681	0,681	0
Liberar acción	23,9	0,555	0,555	0
Compra atómica	253,7	5,899	1,691	4,208

Tabla 9: Gasto de gas detallado por funcionalidad.

* El gasto en USD del cliente no incluye el gasto de la recarga

Los valores obtenidos se detallan en la tabla 9. Se tomó un gas price de 14.31 Gwei (1 Gwei = 0,000000001 ETH) y 1 ETH son 1.625 USD, cotización al día 5 de Noviembre de 2022. Se separaron los gastos en lo que se pagaría con la billetera de BIG y lo que se pagaría con la billetera del cliente que ejecuta la funcionalidad.Como se observa, el uso de compras atómicas aumenta considerablemente el gasto de gas, ya sea por utilizar estructuras que se guardan en memoria de la blockchain como también por enviar más mensajes para realizar la operación. En cambio, la compra simple que consiste en ejecutar la función Registrar Acción y Liberar Acción, tiene un costo bastante menor.

7.3. Fortalezas y debilidades

A continuación se listan las fortalezas y debilidades que tiene la solución.

Fortalezas

Adaptabilidad
Se considera adaptabilidad como la posibilidad de ejecutar los componentes en distintas configuraciones de hardware y software.
Se mantiene el punto fuerte de la versión anterior de la plataforma. Los componentes del gateway levantan servidores web lo que permite ser flexibles para determinar la infraestructura necesaria. Pueden estar todos los componentes en un único nodo de una red o completamente distribuido.

Extensibilidad
Se entiende por extensibilidad a la facilidad de integrar nuevos módulos a la solución.
Se mantiene esta fortaleza. Resultó sencillo extender el sistema y agregar el conector para la red de Ethereum. Por otro lado, para Ethereum se permite que si nuevos usuarios quisieran utilizar BIG para comunicarse desde Ethereum a Fabric (u otras redes), alcanza con que compartan los archivos ABI de los smart contract que desarrollaron en Ethereum, y el conector los va a tomar de un directorio para poder escuchar los eventos de esos contratos como también enviarles transacciones.

Compra atómica
Un nuevo punto fuerte de la plataforma es que soporta la compra atómica, utilizando Atomic Swap, y pagando con ETH.

Reusabilidad
Se entiende por reusabilidad al uso de los mismos componentes para desarrollar nuevas conexiones.
Se modificó la estructura interna para que soporte varias instancias de un conector y que apunten a distintas redes del mismo tipo. Por ejemplo, podemos levantar un

conector de Hyperledger Fabric para la red de los surtidores y otro conector para la red de UTE. Por el lado del conector de Ethereum, una misma instancia puede consumir cualquier smart contract que sea cargado con su archivo ABI, como también, es posible ejecutar distintas instancias del conector que apunten a redes diferentes.

Debilidades

Único punto de fallo
Se mantiene el punto débil de la versión anterior. En caso de caerse el router o los conectores, se pierde la conectividad de toda la solución.

Performance
Se entiende por performance a la cantidad de mensajes procesados.
Como se mencionó en la sección 7.1, el procesamiento en el conector de Ethereum se hace secuencialmente. Esto agrega una demora significativa en el procesamiento del mensaje.

Tolerancia a fallos
Se entiende por tolerancia a fallos a la posibilidad de seguir operando en caso de error de software o hardware.
Al no tener persistencia de los mensajes pendientes de enviar, cuando se encolan en el conector de Ethereum, existe la posibilidad de pérdida de todos los mensajes si se llegara a caer el conector.

8. Gestión del proyecto

En este capítulo se verá el proceso realizado para llevar adelante el proyecto. En la sección 8.1 se explicará la organización del mismo, y luego, en la sección 8.2, se analizará la planificación proyectada al comienzo del proyecto y se comparará con la planificación real.

8.1. Organización del proyecto

Al comienzo del mismo se establece que se realizarán reuniones, con el tutor, cada dos semanas con el objetivo de trabajar en sprints cortos e intentar minimizar las desviaciones.

El proyecto siguió un modelo de cascada y se dividió en 7 fases:

1. Estudio del estado del arte
2. Comparación entre soluciones de interoperabilidad entre blockchains
3. Decisión de solución a utilizar y prototipo
4. Diseño de la solución
5. Implementación de la solución
6. Pruebas de performance
7. Elaboración de informe

Las fases se describen a continuación:

1.Estudio del estado del arte
Durante esta fase se estudiaron los conceptos necesarios para este proyecto: cómo funciona una blockchain, las distintas blockchains que existen así como los distintos tipos (públicas/privadas) y sus diferencias.

2.Comparación entre soluciones de interoperabilidad entre blockchain
Se buscaron y analizaron las distintas soluciones de interoperabilidad presentes en el mercado como también BIG 1.0, la realizada por el proyecto de grado BIG. El foco principal estuvo puesto en BIG y en Hyperledger Cactus aunque también se analizaron otras como se menciona en la sección 2.4.3

3.Decisión de solución a utilizar y prototipo
Con el resultado de la comparación se decidió el uso de Hyperledger Cactus como plataforma a utilizar para luego cambiar a BIG. Esto se mencionó en la sección 4.3. También se realizaron algunas pruebas de concepto con ambas plataformas.

4.Diseño de la solución
Con la plataforma ya definida se procedió a diseñar la solución como se ve en detalle en el capítulo 5.

5.Implementación de la solución
Se implementa la solución diseñada en la etapa anterior como también el caso de uso concreto que se definió en la sección 3.2. El detalle de la implementación se encuentra en el capítulo 6.

6.Pruebas de performance
Al culminar con la implementación de la solución se procedió a realizar algunas pruebas de performance sobre la misma y también se evaluó el consumo de gas, en Ethereum, por parte de las funcionalidades. El detalle se encuentra en el capítulo 7.

7.Elaboración de informe
Luego de culminar las etapas anteriores, se realizó el presente informe con el fin de documentar lo realizado en todo el proyecto. El mismo se realizó de forma iterativa, recibiendo feedback de parte del tutor en cada iteración.

8.2. Planificación del proyecto

En la figura 37 se puede observar la planificación inicial de las distintas fases, trazada al comienzo del proyecto.

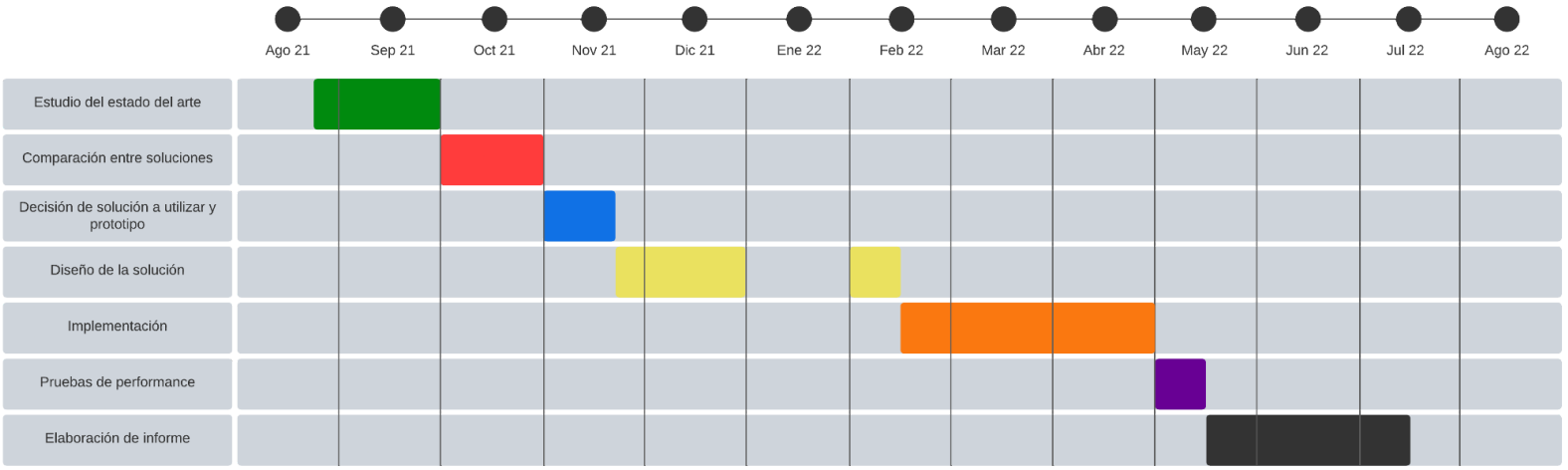

Figura 37: Diagrama Gantt con la planificación inicial

En la figura 38 se presenta la planificación final con las desviaciones que sucedieron. Se puede observar que la etapa 3 (Decisión de solución a utilizar) llevó más tiempo que el estimado previamente ya que, en un principio, se eligió utilizar Hyperledger Cactus para luego cambiar a BIG, consumiendo un tiempo considerable.
También hubo demoras significativas en la elaboración del presente informe, esto se debió en parte a licencias planificadas con anterioridad que no deberían haber afectado de haberse mantenido la planificación inicial.

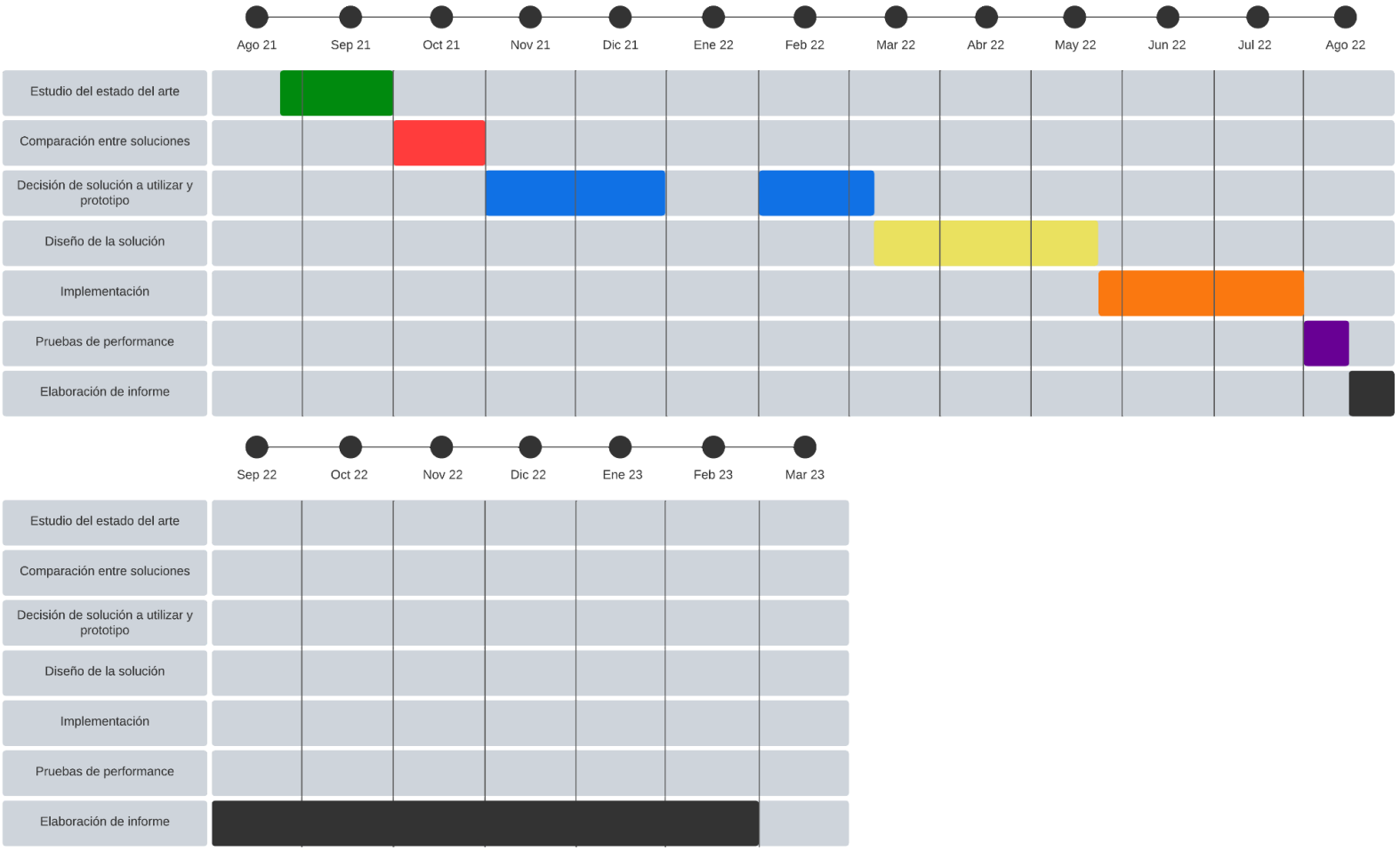

Figura 38: Diagrama Gantt con la planificación final

9. Conclusiones y trabajo a futuro

En este capítulo se presentan las conclusiones del proyecto en la sección 9.1, y los aspectos a trabajar en el futuro en la sección 9.2.

9.1. Conclusiones

El objetivo principal de este proyecto fue lograr la interoperabilidad entre blockchains públicas y privadas. Este objetivo se considera cumplido ya que la extensión realizada a BIG permite la comunicación entre ambos tipos de redes, y esto se logra de forma no invasiva. Esto es, sin necesidad de realizar modificaciones a las blockchains.

Se comenzó estudiando la situación actual respecto a la interoperabilidad entre blockchain y se analizaron varias soluciones. De todas las relevadas se decidió tomar Hyperledger Cactus y la solución BIG presentada en un proyecto de grado anterior. Estas fueron analizadas y comparadas minuciosamente, dando como resultado que BIG era la plataforma a utilizar. Cactus, a pesar que era funcional, no utilizaba el método de interoperación requerido, que es coreografía.

Se realizó un diseño e implementación de una extensión de BIG para incorporar la interoperabilidad con la red de Ethereum. Para evaluar lo implementado, se definió un escenario de uso compuesto por la recarga de vehículos eléctricos abonando con criptomonedas (Ethers en este caso). A su vez, fue implementada una solución para comprar energía utilizando operaciones atómicas, siguiendo los lineamientos de atomic swaps. Asimismo, se realizaron pruebas de carga sobre un escenario particular dentro del desarrollado, que permitieron mejorar la solución inicial, mejorando la performance. Por otro lado, se realizó un análisis de costos, calculando el gasto de gas necesario para la ejecución de cada una de las funcionalidades. Ambas pruebas arrojaron resultados favorables y realistas. Por un lado, presenta un procesamiento de mensajes adecuado, sin demoras significativas, y también un gasto de gas poco significativo.

Para culminar, se elaboró una propuesta de diseño para atacar el desafío de la identidad en la interoperabilidad la cual se detalla en el anexo VII. Su implementación quedó fuera del alcance del proyecto pero pueden considerarse las bases para futuros desarrollos.

La interoperabilidad entre blockchains es un área en constante evolución. Al comienzo del proyecto se analizó la plataforma de Hyperledger Cactus y para el final del proyecto ya había quedado obsoleto. Se integró con la plataforma Weaver y se pasó a llamar Hyperledger Cacti [29]. Vemos que aún no existe una solución en el mercado que sea modelo para lograr la interoperabilidad. Hyperledger Cactus era lo

más cercano pero utiliza un método de interoperabilidad distinto (orquestador) y aún no se encuentra en una versión para su uso en un ambiente productivo. Consideramos que actualmente se puede lograr la interoperabilidad, como se logró en este proyecto, pero siempre es necesario confiar en una tercera parte. Creemos que a largo plazo aparecerán estándares de interoperabilidad, los cuales tendrían que ser implementados por cada una de las blockchains, que permitan esta comunicación con más facilidad.

Por último, se espera que los aportes de este proyecto resulten útiles para el desarrollo de nuevas plataformas de interoperabilidad entre redes públicas y privadas.

9.2. Trabajo a futuro

En esta sección se presentan algunos puntos que quedaron fuera del alcance del proyecto y se pueden trabajar en el futuro.

Descentralización
La solución posee una única autoridad central, el dueño del componente BIG. Sería de utilidad trabajar sobre un forma de descentralizar esa responsabilidad para que el componente sea gestionado por la red en vez de por un único dueño.

Performance
Como se mencionó en la sección 7.3, la solución cuenta con un punto débil que es depender de una única billetera para operar con la red de Ethereum. Esto provoca un cuello de botella en el procesamiento afectando a los tiempos de respuesta. Una posible solución es utilizar más de una billetera para BIG y así mejorar la espera ya que se ejecutarán en paralelo.

Smart contracts desarrollados en Ethereum
Para mejorar la trazabilidad de las respuestas en BIG es necesario que las funciones de los smart contracts de Ethereum reciban el campo *messageId* como parámetro y que los eventos también emitan dicho valor. Con esto se lograría determinar a qué mensaje corresponde la respuesta que se está enviando. A su vez, las funciones tienen que recibir por parámetro los valores indicados en el objeto *replyTo* del mensaje. Actualmente no se está respetando el formato del mismo y los eventos son emitidos hacía donde indica la lógica de la función y no lo que indica el mensaje.

Análisis de otras soluciones
Al momento de finalizar el proyecto surgieron un par de soluciones de interoperabilidad entre Ethereum y Fabric, llamados Gravity Bridge [40] y Cross Framework [41], cómo también Hyperledger Cactus se integró con Weaver,

generando la solución Hyperledger Cacti [29]. Resulta interesante analizarlas y compararlas con la solución actual.

Tolerancia a fallos

Dado el procesamiento secuencial de los mensajes a enviar a Ethereum, en el conector de dicha red, resulta deseable contar con un mecanismo de persistencia de los mismos. En este versión solo se encolan en memoria, por lo que si se da una falla del conector, se perderían todos los mensajes. Además, la arquitectura trabaja con un solo nodo de BIG, por lo que un fallo de la misma afectaría todo el sistema. Sería deseable incorporar varias instancias ejecutando en paralelo, para que en caso de que una fallé, sea posible seguir operando con el resto.

10. Referencias

[1] S. Nakamoto, «Bitcoin: A Peer-to-Peer Electronic Cash System».
[2] B. Bradach y J. Nogueira, «Interoperabilidad entre plataformas de blockchain», 2021, Accedido: 25 de septiembre de 2022. [En línea]. Disponible en: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/31368
[3] PricewaterhouseCoopers, «Blockchain in advertising - Is it the answer to digital advertising's trust and transparency gap?», *PwC*. https://www.pwc.com/us/en/industries/tmt/library/blockchain-in-advertising.html (accedido 22 de febrero de 2023).
[4] «How Does Blockchain Work?», *Binance Academy*. https://academy.binance.com/en/articles/how-does-blockchain-work (accedido 22 de febrero de 2023).
[5] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, y H. Wang, «Blockchain challenges and opportunities: a survey».
[6] «Proof-of-stake (PoS) | ethereum.org». https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/ (accedido 22 de febrero de 2023).
[7] «What Is Proof of Stake (PoS)?», *Binance Academy*. https://academy.binance.com/en/articles/proof-of-stake-explained (accedido 19 de marzo de 2023).
[8] M. Castro y B. Liskov, «Practical Byzantine Fault Tolerance».
[9] X. Xu, I. Weber, y M. Staples, *Architecture for Blockchain Applications*. Cham: Springer International Publishing, 2019. doi: 10.1007/978-3-030-03035-3.
[10] «Inicio | ethereum.org». https://ethereum.org/es/ (accedido 24 de febrero de 2023).
[11] G. Llambıas, L. Gonzalez, y R. Ruggia, «Challenges on blockchain interoperability solutions».
[12] N. El Ioini y C. Pahl, *A Review of Distributed Ledger Technologies*. 2018.
[13] A. Lipton y S. Levi, «An Introduction to Smart Contracts and Their Potential and Inherent Limitations», *The Harvard Law School Forum on Corporate Governance*, 26 de mayo de 2018. https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations/ (accedido 28 de febrero de 2023).
[14] «Sharding — MongoDB Manual». https://www.mongodb.com/docs/manual/sharding/ (accedido 30 de octubre de 2022).
[15] M. Pereira, M. Toscano, y P. Villar, «Plataformas blockchain y escenarios de uso», 2019, Accedido: 24 de febrero de 2023. [En línea]. Disponible en: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/20541
[16] «BNB Chain - Build Web3 dApps on the Most Popular Blockchain», *BNB Chain*. https://www.bnbchain.org/en (accedido 24 de febrero de 2023).
[17] «Solana | Web3 Infrastructure for Everyone». https://solana.com/ (accedido 24 de febrero de 2023).
[18] «Cardano | Home». https://cardano.org/ (accedido 24 de febrero de 2023).
[19] «hyperledger_fabric_whitepaper.pdf». https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf (accedido 24 de febrero de 2023).
[20] «Hyperledger_Arch_WG_Paper_1_Consensus.pdf». https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf (accedido 24 de febrero de 2023).
[21] A. G. Gad, D. T. Mosa, L. Abualigah, y A. A. Abohany, «Emerging Trends in Blockchain Technology and Applications: A Review and Outlook», *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 34, n.º 9, pp. 6719-6742, oct. 2022, doi: 10.1016/j.jksuci.2022.03.007.
[22] J. Gómez Salazar, «Criptomonedas ¿una buena inversión?», bachelorThesis,

Universidad EAFIT, 2021. Accedido: 12 de marzo de 2023. [En línea]. Disponible en: http://repository.eafit.edu.co/handle/10784/29894
[23] «Solidity — documentación de Solidity - UNKNOWN». https://solidity-es.readthedocs.io/es/latest/ (accedido 24 de febrero de 2023).
[24] B. Kang, M. B. Amin, J. Lee, S. Chen, y X. Xu, «Towards Interconnected Blockchains: A Comprehensive Review of the Role of Interoperability among Disparate Blockchains», *ACM Comput. Surv.*, vol. 54, n.º 7, pp. 1-39, sep. 2022, doi: 10.1145/3460287.
[25] «WEF_A_Framework_for_Blockchain_Interoperability_2020.pdf». https://www3.weforum.org/docs/WEF_A_Framework_for_Blockchain_Interoperability_2020.pdf (accedido 17 de marzo de 2023).
[26] «A Survey on Blockchain Interoperability: Past, Present, and Future Trends | ACM Computing Surveys». https://dl.acm.org/doi/10.1145/3471140?sid=SCITRUS (accedido 12 de marzo de 2023).
[27] «Hashed Timelock Contract (HTLC)», *Investopedia*. https://www.investopedia.com/terms/h/hashed-timelock-contract.asp (accedido 27 de febrero de 2023).
[28] A. Nebel, «Arquitectura de microservicios para plataformas de integración», 2019, Accedido: 20 de febrero de 2023. [En línea]. Disponible en: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/20586
[29] P. Somogyvari *et al.*, «Introducing Hyperledger Cacti, a multi-faceted pluggable interoperability framework – Hyperledger Foundation». https://www.hyperledger.org/blog/2022/11/07/introducing-hyperledger-cacti-a-multi-faceted-pluggable-interoperability-framework (accedido 20 de febrero de 2023).
[30] «Hyperledger Cacti». Hyperledger, 11 de febrero de 2023. Accedido: 1 de marzo de 2023. [En línea]. Disponible en: https://github.com/hyperledger/cacti/blob/63357870004e5031c95d85663921ee15e3402339/README.md
[31] «Cosmos Network - Internet of Blockchains», *Cosmos Network*. https://cosmos.network (accedido 18 de marzo de 2023).
[32] «Polkadot and Cosmos · Polkadot Wiki». https://wiki.polkadot.network/docs/learn-comparisons-cosmos (accedido 30 de octubre de 2022).
[33] «YUI», *Hyperledger Labs*. https://labs.hyperledger.org/labs/yui.html (accedido 30 de octubre de 2022).
[34] J. Kimura y C. of Datachain, «Meet YUI, one of the new Hyperledger Labs taking on cross-chain and off-chain operations – Hyperledger Foundation». https://www.hyperledger.org/blog/2021/06/09/meet-yui-one-the-new-hyperledger-labs-projects-taking-on-cross-chain-and-off-chain-operations (accedido 18 de marzo de 2023).
[35] «Weaver DLT Interoperability», *Hyperledger Labs*. https://labs.hyperledger.org/labs/weaver-dlt-interoperability.html (accedido 30 de octubre de 2022).
[36] S. Franzoni, «Blockchain and smart contracts in the Fashion industry», laurea, Politecnico di Torino, 2020. Accedido: 30 de octubre de 2022. [En línea]. Disponible en: https://webthesis.biblio.polito.it/15336/
[37] T. di L. Magistrale, «POLITECNICO DI TORINO».
[38] «Beneficios de blockchain - IBM Blockchain | IBM». https://www.ibm.com/es-es/topics/benefits-of-blockchain (accedido 30 de octubre de 2022).
[39] «Hyperledger Cactus». Hyperledger, 1 de noviembre de 2022. Accedido: 2 de noviembre de 2022. [En línea]. Disponible en: https://github.com/hyperledger/cactus/blob/2ce098f490c0e20c7f5d00a81e8fced1ec81341c/whitepaper/whitepaper.md
[40] «Hyperledger Indy — Hyperledger Indy 1.0 documentation». https://indy.readthedocs.io/en/latest/ (accedido 2 de noviembre de 2022).
[41] «Hyperledger Fabric SDK for Node.js», *fabric-sdk-node*.

https://hyperledger.github.io/fabric-sdk-node/ (accedido 20 de octubre de 2022).
[42] «web3.js - Ethereum JavaScript API — web3.js 1.0.0 documentation». https://web3js.readthedocs.io/en/v1.8.0/ (accedido 20 de octubre de 2022).
[43] «Enterprise Integration Patterns - Request-Reply». https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html (accedido 27 de enero de 2023).
[44] «Enterprise Integration Patterns - Return Address». https://www.enterpriseintegrationpatterns.com/patterns/messaging/ReturnAddress.html (accedido 27 de enero de 2023).
[45] «Enterprise Integration Patterns - Correlation Identifier». https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html (accedido 27 de enero de 2023).
[46] «Hyperledger Fabric SDK for Node.js Tutorial: fabric-common: How to set gRPC settings». https://hyperledger.github.io/fabric-sdk-node/release-2.2/tutorial-grpc-settings.html (accedido 23 de octubre de 2022).
[47] «Express - Node.js web application framework». https://expressjs.com/ (accedido 14 de marzo de 2023).
[48] «Getting Started | Axios Docs». https://axios-http.com/docs/intro (accedido 14 de marzo de 2023).
[49] «Ganache - Truffle Suite». https://trufflesuite.com/ganache/ (accedido 14 de marzo de 2023).
[50] «The crypto wallet for Defi, Web3 Dapps and NFTs | MetaMask». https://metamask.io/ (accedido 14 de marzo de 2023).
[51] D. Simpier, «Everything You Need to Know About the Gravity Bridge Chain», *Medium*, 2 de marzo de 2022. https://blog.cosmos.network/gravity-is-an-essential-force-of-the-cosmos-aligning-all-planets-in-orbits-in-the-composable-b1ca17de18cc (accedido 20 de febrero de 2023).
[52] Datachain, «Datachain and NTT DATA Successfully Verified a Bridge Between Ethereum and Hyperledger Fabric using…», *Medium*, 1 de febrero de 2022. https://medium.com/@datachain/datachain-and-ntt-data-successfully-verified-a-bridge-between-ethereum-and-hyperledger-fabric-using-56a96480e8a4 (accedido 20 de febrero de 2023).

Anexo I. Casos de uso para plataformas de blockchain

En esté anexo se describen dos posibles casos de uso para plataformas de blockchain.

Caso de uso para Hyperledger Fabric

Un caso de uso de Fabric es la gestión de la cadena de suministro, donde varias partes necesitan realizar un seguimiento del movimiento de mercancías y garantizar su autenticidad e integridad. Por ejemplo, un consorcio de empresas está involucrado en la producción y distribución de granos de café. El consorcio incluye productores de café, tostadores, distribuidores y minoristas. Quieren usar Hyperledger Fabric para rastrear el movimiento de los granos de café desde los productores hasta los minoristas y garantizar su autenticidad y calidad.
Para utilizar Hyperledger Fabric en la gestión de su cadena de suministro deberían seguir los siguientes pasos:

- Creación de la red: El consorcio crea una red Fabric y define las reglas para la validación y aprobación de transacciones. También definen los requisitos de privacidad y seguridad de la red.

- Creación de la chaincode: se desarrolla e implementa una chaincode en la red Fabric. La chaincode define las reglas sobre cómo se pueden rastrear los granos de café y cómo se deben validar y respaldar las transacciones.

- Registro de datos de activos: cuando se cosecha un nuevo lote de granos de café, sus datos se registran en la cadena de bloques de la red Fabric. Los datos incluyen información sobre el origen de los granos de café, su calidad y su ubicación. Cada parte involucrada en la cadena de suministro puede acceder a estos datos y rastrear el movimiento de los granos de café de una parte a la siguiente.

- Validación de transacciones: Cuando los granos de café cambian de manos, se registra una nueva transacción en la red Fabric. Cada transacción es validada y respaldada por las partes involucradas en la transacción de acuerdo con las reglas definidas por la chaincode. Esto garantiza que la transacción sea válida y segura.

- Seguimiento y verificación: Cada parte puede rastrear el movimiento de los granos de café en la red Fabric y verificar su autenticidad y calidad. La transparencia de la red Fabric permite que todas las partes tengan una visión

clara de la cadena de suministro, lo que reduce el riesgo de fraude y garantiza que solo se entreguen granos de café de alta calidad a los minoristas.

- Cumplimiento normativo: los reguladores también pueden acceder a la red de Fabric y verificar que las partes involucradas en la cadena de suministro cumplan con las regulaciones, como las relacionadas con la seguridad alimentaria o los estándares ambientales. Esto ayuda a garantizar que los granos de café se produzcan y transporten de manera segura y responsable.

Caso de uso para Ethereum

Un caso de uso de Ethereum son las finanzas descentralizadas (DeFi). DeFi se refiere a un conjunto de aplicaciones financieras que se construyen sobre la cadena de bloques y funcionan sin la necesidad de intermediarios como bancos o instituciones financieras. DeFi en Ethereum tiene el potencial de brindar servicios financieros a personas desatendidas o excluidas de las instituciones financieras tradicionales. También ofrece un sistema financiero más transparente y descentralizado que es menos vulnerable a la censura o la manipulación. Por ejemplo, las plataformas de préstamos permiten a los usuarios prestar y tomar prestadas criptomonedas sin la necesidad de un intermediario centralizado. Estas plataformas operan mediante smart contracts. Así es como funcionan:

- Los prestamistas depositan criptomonedas en un smart contract: los usuarios depositan criptomonedas en un smart contract que ha sido programado para gestionar el proceso de préstamo. El smart contract mantiene la criptomoneda depositada como garantía.

- Los prestamistas ganan intereses: los prestamistas en la plataforma ganan intereses sobre su criptomoneda depositada. La plataforma establece la tasa de interés en función de la oferta y la demanda, y los titulares de tokens de gobierno de la plataforma pueden ajustarla.

- Los prestatarios pueden tomar prestadas criptomonedas: los prestatarios pueden tomar prestadas criptomonedas del smart contract al proporcionar su propia garantía. La cantidad de criptomoneda que se puede pedir prestada suele ser un porcentaje de la garantía proporcionada.

- Los prestatarios pagan intereses: los prestatarios deben pagar intereses sobre la cantidad que piden prestada, que también establece la plataforma. Si el prestatario no paga el préstamo a tiempo, su garantía se entregará al prestamista.

- Duración y términos del préstamo: la duración y los términos del préstamo también están determinados por el smart contract. Estos términos suelen estar predeterminados y no se pueden cambiar una vez que se ha emitido el préstamo.

Anexo II. Atomic Swaps

Los atomic swaps son la combinación de Hash Locks y Time Locks. Los primeros son un bloqueo que requiere que ciertos datos se hagan públicos para desbloquearse. Los segundos son un bloqueo que requiere que se alcance cierto tiempo para desbloquearse. Estos son utilizados para el intercambio atómico de criptomonedas (atomic swaps) y la forma general sigue los siguientes pasos:
Se supone que hay dos actores, Alice y Bob, que quieren intercambiar criptomonedas. Alice tiene bitcoins y Bob tiene ethers.
1) Alice crea aleatoriamente una clave secreta s y genera un hash h de ese secreto.
2) Alice publica un HTLC con un time lock t1 y sólo enviará los bitcoins a Bob si éste presenta el secreto s. En este HTLC quedan bloqueados los bitcoins a transferir. Si se cumple el tiempo t1, los bitcoins se enviarán de nuevo a Alice.
3) Alice le envía a Bob el hash h y la función de hash. Bob verifica que el HTLC esté publicado.
4) Bob publica un HTLC con time lock t2 y sólo enviará los ethers a Alice si éste presenta un secreto s tal que al aplicarle la función de hash sea igual a h. En el HTLC quedan bloqueados los ethers de Bob.
5) Alice presenta el secreto s al HTLC de Bob y se le envían los ethers.
6) Bob presenta el secreto s al HTLC de Alice y se le envían los bitcoins.

Anexo III. Propuesta cambios a Cactus

En esté anexo se proponen cambios a la arquitectura de Cactus para convertirlo a interoperabilidad basada en coreografía.

Realizar consultas http desde los smarts contracts

Se investigó cómo poder iniciar las transacciones cross chain desde alguna de las blockchains, particularmente desde sus smarts contracts vía consumo de endpoints http y se encontró que es inviable. Según investigado, no se puede, desde los smarts contracts, hacer consumos http. Esto es debido a la característica de las blockchain de ser deterministas, para poder lograr la correcta ejecución de los algoritmos de consenso. Dado que el sistema debe contar con esta característica es que se entra en conflicto con los consumos http, dado que éstos pueden arrojar resultados distintos en dos diferentes consumos rompiendo así el determinismo.

Implementar middleware para cada conector

Una opción para superar el problema de las llamadas http es implementar un middleware entre la blockchain y el conector. La idea es que el middleware esté escuchando los eventos emitidos por los smart contract (principalmente los que comienzan las ejecuciones de los casos de uso) y, al recibir alguno, se encargue de realizar un llamado Http a la lógica de negocio que expone Cactus y de esta manera proseguir con la ejecución.

Lo negativo de esta solución es que complejiza la arquitectura innecesariamente ya que agregamos más puntos de falla. Otro punto es que el middleware, básicamente, sería un conector para la blockchain con lo que estaríamos realizando componentes que ya están desarrollados.

Modificar Cactus para que funcione escuchando eventos y no levantando una interfaz REST HTTP.

Ya que el problema es que no se puede realizar llamadas http desde un smart contract, la última opción realizable, y también la más costosa, es modificar el comportamiento de Cactus y hacer que los conectores escuchen eventos inmediatamente que levanta la lógica de negocio. Esta opción, si bien es posible, se considera que es cambiar la esencia del software que fue desarrollado por Hyperledger.

Anexo IV. Configuración de los componentes del gateway

Todos los componentes del *Gateway* utilizan un archivo de configuración propio llamado *config.json*. En estos archivos se encuentran los datos del *host* y *puerto* a utilizar para dicho componente. También los datos de conexión con los otros componentes (el host y puerto de cada uno). A continuación se presentarán detalles de estos archivos y de dónde se usan sus datos en el código.

Router

En el config.json del router se encuentran los datos del host y puerto a utilizar por dicho componente para publicar su API REST. También la conexión con los otros componentes y los conectores de cada red. Estos datos se presentan en la figura 39. En la figura 40 podemos observar que los parámetros son utilizados a la hora de levantar el endpoint del router y en la figura 41 se puede ver la utilización de los parámetros para rutear los eventos a cada conector.

config.json 323 bytes

```
{
    "router": {
        "endpoint": "127.0.0.1",
        "port": 3000
    },
    "blockchains": {
        "fabric": {
            "connectorHost": "127.0.0.1",
            "connectorPort": 3001
        },
        "ethereum": {
            "connectorHost": "127.0.0.1",
            "connectorPort": 30400
        }
    }
}
```

Figura 39: Config.json del Router.

```
startConnectorsEndpoint() {
    // Starts server that exposes endpoint to be called by connectors when
    // an event needs to be sent to another blockchain.
    const app = http.createServer((request, response) => {
        this.receiveRequestData(request)
            .then(data => {
                this.eventReceivedFromConnector(data, response);
            });
    });

    app.listen(config.router.port, config.router.endpoint);
}
```

Figura 40: Start del Router.

```
routeEvent(event) {
    // Routes the received event to the corresponding blockchain
    const connectorPort = config.blockchains[event.header.target.blockchain].connectorPort;
    const options = {
        host: config.blockchains[event.header.target.blockchain].connectorHost,
        path: '',
        port: connectorPort,
        method: 'POST'
    }
```

Figura 41: Operación de envío del evento al conector correspondiente.

Conector Ethereum

En la figura 42 se observa el config.json del conector de Ethereum. Este archivo tiene establecida la información del endpoint del router al que va a enviar los eventos destinados a otros conectores. En la figura 43 se puede ver la operación que utiliza dichos parámetros. Los parámetros dentro de la estructura “blockchain” son parámetros específicos de la red, el *host* y *puerto* del endpoint dónde se recibirán los eventos, la dirección de la red Ethereum a utilizar. La dirección en esta red asignada cómo billetera para uso de este componente. En las figuras 44 y 45 se puede ver su utilización en el código.

config.json 313 bytes Open

```json
{
    "router": {
        "endpoint": "127.0.0.1",
        "port": 3000
    },
    "blockchain": {
        "ethereumHost": "ws://127.0.0.1:7545",
        "connectorPort": 30400,
        "connectorHost": "127.0.0.1",
        "bigWalletAddress": "0x357FE607324Eb13899CbE94f50cE93B0Dc5171De"

    }
}
```

Figura 42: Config.json del conector Ethereum.

```
sendEventToRouter(event) {
// sends an event to the router
    const options = {
        host: config.router.endpoint,
        path: '/',
        port: config.router.port,
        method: 'POST'
    }

    this.callendpoint(options, JSON.stringify(event))
        .then(() => {
            console.log('\nEvent sent to router properly');
```

Figura 43: Operación de envío del evento al router.

```
async startRouterEndpoint() {
// starts server that exposes endpoint that is going to be called by the router when there's an event
    const app = http.createServer((request, response) => {
        this.receiveRequestData(request)
            .then(async data => {
                await this.eventReceivedFromRouter(data, response);
            });
    });

    app.listen(config.blockchain.connectorPort, config.blockchain.connectorHost);
}
```

Figura 44: Start del conector Ethereum.

```
async sendEventToEthereumBlockchain(event) {

    // notifies ethereum blockchain about the received event
    lock.acquire("send", async function(done) {
        const web3 = new Web3(config.blockchain.ethereumHost)
        const networkId = await web3.eth.net.getId();
        contracts.forEach(async (contract) => {
            if (contract.contractName == event.header.target.contract){
                var deployedNetwork = contract.networks[networkId];
                var myContract = new web3.eth.Contract(
                    contract.abi,
                    deployedNetwork && deployedNetwork.address,
                );
                var params = [];
                for (var param in event.data)
                    params.push(event.data[param]);
                //console.log(params);
                var nonce = await web3.eth.getTransactionCount(config.blockchain.bigWalletAddress);
                var rawTransaction = {
                    nonce: web3.utils.toHex(nonce),
                    from: config.blockchain.bigWalletAddress,
                    to: deployedNetwork.address,
                    data: myContract.methods[event.header.target.operation].apply(null,params).encodeABI()
                };
```

Figura 45: Operación de envío del evento a Ethereum.

Conector Fabric

En la figura 46 se ve la configuración del conector de la red Fabric. De forma análoga al conector de Ethereum, el config del conector de Fabric tiene establecido la información del endpoint del *Router* al que va a enviar los eventos destinados a otros conectores. En la figura 47 se puede ver su utilización. Los parámetros dentro de la estructura “blockchain” son parámetros específicos de la red, el *host* y *puerto* del endpoint dónde se recibirán los eventos. Su utilización puede verse en la figura 48. Los demás parámetros son propios de la conexión contra la red Fabric explicados en la sección 5.2.2. En la figura 49 puede verse como se establece la conexión. Para levantar el listener de los eventos se requiere el eventId y el canal de la red desde dónde se emitirá el evento y el nombre del smart contract. El Channel también se requiere para enviar los eventos recibidos por el router hacía la red, junto con el *target.contract y target.operation*. Estas dos características se pueden observar en las figuras 50 y 51.

config.json 506 bytes Open in Web IDE Repla

```
{
    "router": {
        "endpoint": "127.0.0.1",
        "port": 3000
    },
    "blockchain": {
        "connectorHost": "127.0.0.1",
        "connectorPort": 3001,
        "fabricUser": "user1",
        "fabricChannel": "mychannel",
        "fabricContract": "estacioneco",
        "eventId": "eventToConnector",
        "walletPath": "../../fabric/fabric-blockchain/estacionEco/wallet",
        "connectionJSONPath": "../../fabric/fabric-blockchain/first-network/connection-org1.json"

    }
}
```

Figura 46: Config.json del conector Fabric.

```
sendEventToRouter(event) {
// sends an event to the router
    const options = {
        host: config.router.endpoint,
        path: '/',
        port: config.router.port,
        method: 'POST'
    }

    this.callendpoint(options, JSON.stringify(event))
        .then(() => {
            console.log('\nEvent sent to router properly');
        })
        .catch(err => {
            console.error('\nAn error occurred while trying to send
            console.error(err);
        });
```

Figura 47: Operación de envío del evento al router.

```
startRouterEndpoint() {
// starts server that exposes endpoint that is going to be called by the router when there
    const app = http.createServer((request, response) => {
        this.receiveRequestData(request)
            .then(data => {
                this.eventReceivedFromRouter(data, response);
            });
    });
    app.listen(config.blockchain.connectorPort, config.blockchain.connectorHost);
}
```

Figura 48: Start del conector de Fabric

```
async loadFabricGateway() {
    const ccpPath = config.blockchain.connectionJSONPath;
    const user = config.blockchain.fabricUser;
    const walletPath = config.blockchain.walletPath;
    const wallet = new FileSystemWallet(walletPath);
    const userExists = await wallet.exists(user);

    if (!userExists) {
        throw new Error(`An identity for the user "${user}" does not exist in the wallet`);
    }

    this.fabricGateway = new Gateway();
    await this.fabricGateway.connect(ccpPath, { wallet, identity: user, discovery: { enabled: true, asLocalhost: true }});
}

printLine(msg) {
```

Figura 49: Operación de conexión del conector con red Fabric.

```
async listenBlockchainEvents() {
// listen events on fabric blockchain
    const network = await this.fabricGateway.getNetwork(config.blockchain.fabricChannel);
    const contract = network.getContract(config.blockchain.fabricContract);
    await contract.addContractListener('listener', config.blockchain.eventId, this.eventReceivedFromFabric.bind(this));
}
```

Figura 50: Operación de escucha de eventos de red Fabric.

```
async sendEventToFabricBlockchain(event) {
// notifies fabric blockchain about the received event
    const network = await this.fabricGateway.getNetwork(config.blockchain.fabricChannel);
    const contract = network.getContract(event.header.target.contract);
    await contract.submitTransaction(event.header.target.operation, JSON.stringify(event));
}
```

Figura 51: Operación de envío del evento a red Fabric.

Anexo V. Smart Contracts

En este anexo se detallan los smart contracts implementados en Ethereum y en Fabric.

Ethereum

A continuación se detallan los smart contracts implementados en Ethereum junto con sus operaciones y los eventos emitidos.

Vale la pena mencionar, que los smart contracts de Ethereum no implementan completamente el diseño presentado en la sección 5.2.1. Esto se debió a dos factores. El primero es que las operaciones de Ethereum reciben una lista fija de parámetros tipados, lo que imposibilita el envío del JSON del mensaje como parámetro de las funciones. El segundo factor se refiere al costo ya que al enviar más datos hacia la función, la ejecución de la misma requiere más comisión, aumentando los costos.

Smart contract ChequeoEnergia
Se desarrolló el smart contract *ChequeoEnergia* el cual contiene dos operaciones: *chequeoEnergia* y *devolverResultado*, y se encarga de obtener la cantidad de energía disponible en cierto surtidor. La operación *chequeoEnergía* recibe la dirección de la billetera del surtidor que se quiere consultar el disponible y emite el evento *EventChequeoEnergia*. Por otro lado, la operación *devolverResultado* recibe la cantidad de energía disponible en un surtidor y emite el evento *EventDevolverResultado*.

Smart contract EnergiaAtomico
El siguiente smart contract, *EnergiaAtomico*, se encarga de la operación de compra atómica de energía iniciando desde Ethereum. Contiene cinco operaciones: *iniciarOperacion*, *validar*, *verificar*, *cerrar* y *reembolso*. La operación *iniciarOperacion* recibe un guid de transacción aleatorio, la dirección de la billetera del surtidor donde se va a cargar la energía, una clave (frase) convertida a un hash utilizando keccak256, y un número entero indicando el tiempo máximo de espera en milisegundos. Esta operación guarda en memoria los datos de la compra de energía y emite el evento *eventIniciarOperacion* con el guid de la transacción, la dirección de la billetera del surtidor, el hash y la cantidad de energía a comprar. La operación *validar* recibe el guid de la transacción y emite el evento *eventValidar* con el mismo dato recibido. La operación *verificar* recibe el guid de la transacción y la frase enviada en la operación *iniciarOperacion* pero sin realizarle el hash. Esta operación emite el evento *eventVerificar* con los mismos datos recibidos. La operación *cerrar* recibe el guid de la transacción y frase sin realizarle el hash, el mismo valor que en

la operación anterior. Se encarga de verificar que la frase corresponda con la que se guardó en memoria al iniciar la operación, marca la transacción como finalizada y transfiere los ethers a la billetera del surtidor. Además emite el evento eventCerrado con el guid de la transacción. La operación *reembolso* recibe el guid de la transacción. Se encarga de marcar la transacción como expirada y transfiere los ethers de vuelta a la billetera del cliente. Además, emite el evento *eventReembolso*, con guid de la transacción.

Smart contract EnergiaFabric

Por último se implementó el smart contract *EnergiaFabric*, el cual se encarga del control del sistema de pagos mencionado en la sección 5.2.4. Contiene tres operaciones: *registrarAccion*, *obtenerAccion* y *liberarAccion*. La operación *registrarAccion* recibe la dirección de la billetera del surtidor en el cual se va a realizar la carga y guarda en memoria la acción, que corresponde a la billetera del surtidor junto con el monto a cargar. No emite eventos. La operación *obtenerAccion* recibe la dirección de la billetera del cliente. Esta operación chequea si existe en la memoria alguna acción cargada para esa billetera y emite el evento *eventSaldo*. El evento tiene los datos el monto a cargar enviado en la operación *registrarAccion*, y la dirección de la billetera del surtidor donde se va a realizar la carga. La operación *liberarAccion* recibe la dirección de la billetera del cliente. Obtiene la acción previamente registrada para esa billetera del cliente, se encarga de marcar la acción como completada y transfiere los ethers hacia la billetera del surtidor. No emite eventos.

Hyperledger Fabric

Para esta red se desarrolló un único smart contract con varias operaciones detalladas a continuación.

Operaciones:

- obtenerEnergia(event)
- iniciarOperacion(event)
- verificar(event)
- obtenerAccion(id)
- devolverAccion(event)
- compraFabric(id, pumpWallet, amount, data, dataSignedBase64, publicKeyPem, algorithm)

La operación *obtenerEnergia* recibe la dirección de la billetera del surtidor del cual se quiere saber la energía disponible. Emite un evento con la cantidad de energía disponible.

La operación *iniciarOperacion* recibe el monto de energía a cargar, la dirección de la billetera del surtidor donde se realiza la carga, la clave (frase) convertida a un hash

utilizando keccak256, y guid de la transacción. Se encarga de guardar en memoria los datos de la compra y emite un evento con el guid de la transacción, igual al guid recibido.

La operación *verificar* recibe el guid de la transacción, y la clave (frase) sin convertirla a hash. En este método se convierte la clave al hash, utilizando keccak256, y se valida que corresponda a lo recibido en la operación *iniciarOperacion* para ese guid de transacción. En caso afirmativo, se libera la energía y se emite el evento con los mismos datos recibidos, guid de transacción y clave (frase) sin hash. La operación *obtenerAccion* recibe la dirección de la billetera del cliente, y emite un evento con el mismo dato recibido. La operación *devolverAccion* recibe los datos de la acción previamente cargada en la red de Ethereum. Estos son: cantidad de energía que se va a cargar y dirección de la billetera del surtidor donde se va a realizar la carga. Emite un evento de nombre *devolverAccion*, con los mismos datos recibidos. Por último se encuentra la operación *compraFabric*. Recibe la dirección de la billetera del cliente, la dirección de la billetera del surtidor donde se realiza la carga, la cantidad de energía a cargar, una clave (frase) utilizada para firmar, la clave firmada y convertida a Base64, utilizando un par de claves RSA aleatorio y algoritmo SHA512, la clave pública del par RSA en formato PEM, y el algoritmo usado para firmar. En este método se valida que la firma sea correcta, teniendo en cuenta la clave sin firmar, la clave firmada, la clave pública y el algoritmo utilizado. En caso de que se verifique, se libera la energía y se emite un evento con la dirección de la billetera del cliente que hace la carga.

A su vez, el smart contract contiene una lista con los surtidores disponibles, como se puede ver en la figura 52. Para cada uno de ellos se mantiene un número entero que representa la cantidad de energía disponible, en las compras de energía se resta de este número, y la dirección de la billetera correspondiente a cada surtidor en la red de Ethereum.

```
const surtidores = [
  {
    id: 'Surtidor 1',
    ethAddress: '0x6fF95BB4292A1b50310B354bc60C8aaa8dcb3E8e',
    energia: 100
  },
  {
    id: 'Surtidor 2',
    ethAddress: '0x6aF0e9BB01B1D5be9c0317f3653802B89fdfC1d9',
    energia: 200
  },
  {
    id: 'Surtidor 3',
    ethAddress: '0xa2853F221805a98ba9bbC2C71ffE2617579Fb8a2',
    energia: 300
  }
];
```

Figura 52: Lista de surtidores en Fabric

Anexo VI. Detalles de implementación de la web

El código fuente de las DApps se encuentra en la carpeta "*dapp*" del proyecto, en una carpeta para cada una de las redes involucradas, y dentro de ésta también se encuentra el index.html. La figura 53 muestra la estructura de las carpetas.

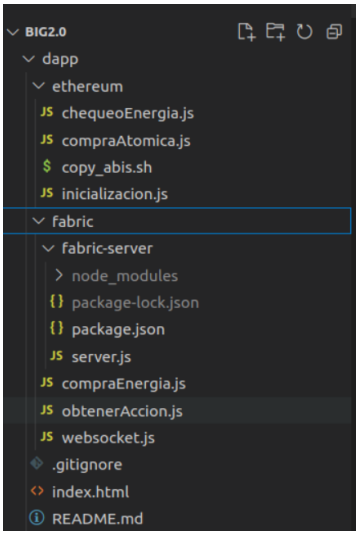

Figura 53: Estructura de carpetas

En la figura 54 se puede observar el head del html dónde lo más destacable es la parte dónde se cargan los scripts JavaScript de las respectivas DApps.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <title>EstacionEco Web</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <script src="ethereum/web3.min.js" crossorigin="anonymous"></script>
  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.js"></script>
  <script src="https://bundle.run/buffer@6.0.3"></script>

  <script defer type="module" src="ethereum/inicializacion.js"></script>

  <script defer type="text/javascript" src="ethereum/compraAtomica.js"></script>
  <script defer type="text/javascript" src="ethereum/chequeoEnergia.js"></script>
  <script defer type="text/javascript" src="fabric/compraEnergia.js"></script>
  <script defer type="text/javascript" src="fabric/obtenerAccion.js"></script>

  <script type="text/javascript" src="fabric/websocket.js"></script>

</head>
```

Figura 54: Carga scripts javascript

Cada uno de estos scripts se encargan de gestionar la interacción con el usuario de la app web y las redes, ejecutando así los diferentes flujos. El script de inicialización de Ethereum posee la función que inicializa las variables necesarias para la operativa de la librería Web3, la carga de los ABIs. En el caso de la DApp la carga no se ha generalizando. En la figura 55 puede verse la carga de los ABIs.

inicializacion.js 1.31 KiB Open in Web IDE

```
import { ContractAtomico } from "./jsonAtomico.js";
import { ContractChequeoEnergia } from "./jsonChequeo.js";
import { ContractEnergiaFabric } from "./jsonEnergiaFabric.js";

async function load() {
    if (window.ethereum) {
        window.web3 = new Web3(window.ethereum);
        window.ethereum.enable();
        updateStatus('Listo para comprar energia!');
    }
    const networkId = await web3.eth.net.getId();

    const deployedNetworkAtomico = ContractAtomico.networks[networkId];
    window.contractAtomico = await new web3.eth.Contract(
        ContractAtomico.abi,
        deployedNetworkAtomico && deployedNetworkAtomico.address,
    );

    const deployedNetworkChequeoEnergia = ContractChequeoEnergia.networks[networkId];
    window.contractChequeoEnergia = await new web3.eth.Contract(
        ContractChequeoEnergia.abi,
        deployedNetworkChequeoEnergia && deployedNetworkChequeoEnergia.address,
    );

    const deployedNetworkEnergiaFabric = ContractEnergiaFabric.networks[networkId];
    window.contractEnergiaFabric = await new web3.eth.Contract(
```

Figura 55: Carga ABIs

Los demás scripts de la DApp de Ethereum son interacciones con la librería Web3. Por ejemplo, dando inicio a la compra atómica o iniciando el proceso de chequeo de energía en un surtidor. Para las pruebas realizadas a lo largo de este proyecto, el complemento del navegador utilizado para la gestión de las billeteras de la red de Ethereum fue MetaMask, como se ve en la figura 22.

Como se mencionó en la sección 6.2.2, se necesita un servidor nodeJS con una API REST, que se conecta a Hyperledger Fabric. La función de conectar a la red Fabric es similar a la del conector, como puede verse en la figura 56.

```
async function connectToFabric() {
// initializes connection to fabric
    const wallet = new FileSystemWallet(WALLET_PATH);
    const userExists = await wallet.exists(FABRIC_USER);

    if (!userExists) {
        console.log('An identity for the user "' + FABRIC_USER + '" does not exist in the wallet');
        console.log('Run the registerUser.js application before retrying');
        return;
    }

    const gateway = new Gateway();
    await gateway.connect(CONNECTION_FILE, { wallet, identity: FABRIC_USER, discovery: { enabled: true, asLocalhost: true } });

    const network = await gateway.getNetwork(config.blockchain.fabricChannel);
    contract = network.getContract(config.blockchain.fabricContract);
    await contract.addContractListener('listener', 'devolverAccion', eventReceivedFromFabric.bind(this));
}
```

Figura 56: Conexión Hyperledger Fabric

Para el caso de obtener la acción, en la figura 57 puede verse cómo se publica el endpoint HTTP REST. Luego será llamado en el script obtenerAccion.js con el módulo axios de JavaScript, como puede observarse en la figura 58.

```
server.post('/obtenerAccion', async function (req, res) {
    try {
        console.log(req.body.id);
        await contract.submitTransaction('obtenerAccion', req.body.id);

    } catch (error) {
        res.send({
            error: error
        });
    }
});
```

Figura 57: Publicación endpoint HTTP

obtenerAccion.js 623 bytes

```
async function obtenerAccion() {
    const id = document.getElementById('identificador').value;
    const data = {
        id
    };
    axios.post('http://localhost:3010/obtenerAccion', data);
```

Figura 58: Interfaz web de la DApp

Anexo VII. Propuesta de incorporación de gestor de identidades descentralizado.

Hyperledger Indy es una plataforma de identidad descentralizada de código abierto que se basa en la tecnología de ledger distribuida (DLT). Es parte del proyecto Hyperledger. Su objetivo es proporcionar una plataforma segura, y que preserve la privacidad, para administrar identidades digitales. Permite a los usuarios crear y controlar sus propias identidades digitales, que se pueden verificar y compartir con otros de una manera que protege la privacidad del usuario y minimiza el riesgo de robo de identidad. Utiliza un enfoque único para la gestión de identidades que se basa en una arquitectura de identidad descentralizada. Esto significa que en lugar de depender de una autoridad centralizada para administrar las identidades, los usuarios controlan sus propias identidades y usan una red de confianza para verificarlas. Está diseñada para ser altamente escalable e interoperable con otras plataformas de cadena de bloques. Utiliza una arquitectura modular que permite a los desarrolladores crear aplicaciones y servicios personalizados sobre la plataforma.

Para incorporar la blockchain Hyperledger Indy, nuestra propuesta consiste en desarrollar un nuevo plugin que se encargue de la comunicación, similar a lo realizado para comunicarse con la red de Ethereum. En la figura 59 se puede apreciar la arquitectura propuesta.

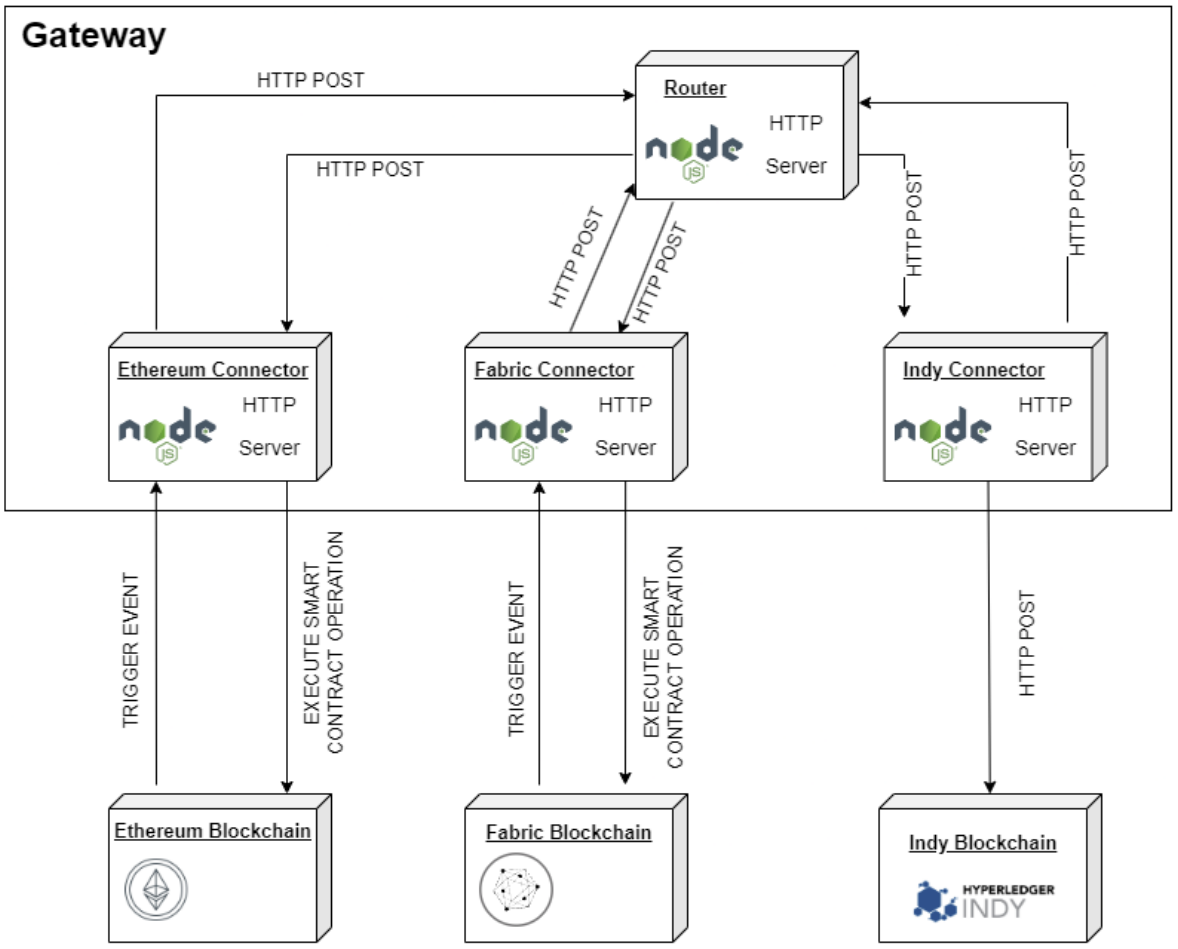

Figura 59: Propuesta de arquitectura incorporando Hyperledger Indy

Este conector obtendría las definiciones de los esquemas de credenciales presentes en la blockchain, con los cuales, se podría validar si las pruebas de identidad incluídas en los mensajes son verdaderas.
En el capítulo 5 se verá el caso de uso propuesto para la interacción entre las blockchain, en ese caso se podría agregar Hyperledger Indy y daría la posibilidad de manejar precios diferenciales para clientes que cumplan ciertas condiciones, por ejemplo, que trabajan en una empresa específica.
En este escenario, el cliente enviaría en el mensaje una prueba que indique a qué empresa pertenece y luego se validaría esa prueba yendo contra Hyperleder Indy.
Si el resultado es positivo, se cobraría un precio diferencial.