UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Diseño y desarrollo de aplicaciones basadas en patrones de interoperabilidad en blockchain

Informe de Proyecto de Grado presentado por

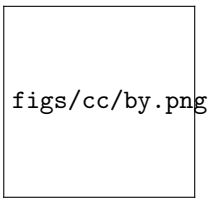
Luis Pedro Bonomi, Ignacio Olivan y Lucas Romero

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

Supervisor

Guzmán Llambías

Montevideo, 11 de abril de 2025



Diseño y desarrollo de aplicaciones basadas en patrones de interoperabilidad en blockchain por Luis Pedro Bonomi, Ignacio Olivan y Lucas Romero tiene licencia CC Atribución 4.0.

Agradecimientos

Yo, Luis Pedro, quiero agradecer principalmente a papá, mamá, José e Inés por acompañarme y apoyarme en todo momento en este lindo recorrido por la facultad. Agradezco también a todos mis amigos, los viejos y también los nuevos de la facultad, por estar conmigo en esta etapa tan importante de mi vida.

Yo, Ignacio, quiero agradecer a mis padres, hermanas y amigos, quienes me han acompañado a lo largo de toda mi carrera. Además, quiero agradecerle a mis compañeros de la facultad, muchos de los cuales se han convertido también en amigos.

Yo, Lucas, quiero agradecer a mi madre y hermana por su apoyo, así como a mis compañeros y amigos que me han acompañado durante el trayecto.

Finalmente, de parte de todos, queremos agradecer a Guzmán, por su inmensa buena onda y paciencia durante todo el proyecto, no se podría haber pedido un mejor tutor para esta iniciativa.

Resumen

El creciente uso de tecnologías blockchain ha impulsado la necesidad de lograr una interoperabilidad efectiva entre redes. Este contexto plantea nuevos desafíos para los desarrolladores y usuarios, que requieren herramientas y mecanismos para interactuar con diferentes blockchains de forma fluida, segura y eficiente. Si un desarrollador requiere de la capacidad de interoperar entre blockchains, tendrá que recurrir a utilizar algunas de las soluciones existentes de interoperabilidad, las cuales pueden resultar caras, o no amoldarse a lo que el usuario desee. Si se deseara implementar una solución propia para interoperar para evitar el uso de soluciones existentes, se cuentan con diversos patrones de interoperabilidad que asisten al desarrollo de una implementación. Sin embargo, estos patrones están definidos a un alto nivel de abstracción, por lo que no se cuenta con documentación y ejemplos concretos de como implementarlos. Esto resulta en una mayor complejidad y costo a la hora de desarrollar una solución para la interoperabilidad. Este es el problema que se aborda en este proyecto, en el cual se intenta proveer una implementación de referencia para algunos de los patrones de interoperabilidad. Entonces, se propuso el diseño e implementación de tres de los patrones: el patrón Relayer, que funciona como intermediario de comunicación entre redes; la Transferencia Permanente, que permite mover activos de manera definitiva entre blockchains; y la Transferencia Temporal, que facilita envíos temporales de activos con la posibilidad de devolución. La metodología incluyó un análisis comparativo de soluciones existentes en la industria, como Chainlink y Wormhole, seguido del diseño arquitectónico basado en contratos inteligentes y conectores personalizados. La implementación se aplicó, a modo de ejemplo de uso de los patrones, en un caso práctico centrado en la venta y transferencia de discos musicales digitales y entradas para conciertos, representados mediante tokens no fungibles (NFTs). Los principales resultados son una implementación bien documentada de cada uno de estos patrones, así como un escenario de ejemplo de uso de los mismos. En este escenario los patrones son utilizados para transferir activos entre dos blockchains no permisionadas (públicas), Ethereum y Polygon, y una blockchain permisionada (privada), Hyperledger Fabric. Se desarrollaron tres aplicaciones descentralizadas (dApps) que permiten a los usuarios interactuar con contratos inteligentes desplegados en cada red, obteniendo y transfiriendo NFTs de forma segura. También se realizó un análisis de costos de las implementaciones que demuestra la viabilidad económica del enfoque adoptado. Como conclusión, este

trabajo pone en evidencia el valor del aprendizaje adquirido en la integración de tecnologías blockchain complejas. En conjunto, esta investigación ofrece una base sólida para desarrollos futuros en el ámbito de la interoperabilidad entre blockchains, con potencial para expandirse con nuevos patrones y mejoras en la eficiencia operativa.

Palabras clave: Blockchain, Interoperabilidad, Blockchain permisionada, Blockchain no permisionada, Finalización de bloques, Patrones de Interoperabilidad

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 2
- 1.3. Aportes del proyecto . . . 2
- 1.4. Estructura del documento . . . 3

2. Marco teórico **5**
- 2.1. Blockchain . . . 5
  - 2.1.1. ¿Cuáles son las características de la tecnología Blockchain? 6
  - 2.1.2. Clasificación . . . 6
  - 2.1.3. Algoritmos de consenso . . . 7
  - 2.1.4. Finalidad de transacciones . . . 8
  - 2.1.5. Smart Contracts . . . 9
  - 2.1.6. Tokenización . . . 9
- 2.2. Ethereum . . . 10
  - 2.2.1. ¿Qué es el Gas en Ethereum? . . . 10
  - 2.2.2. Finalidad en Ethereum . . . 11
- 2.3. Polygon . . . 11
  - 2.3.1. Características principales de Polygon . . . 12
- 2.4. Hyperledger Fabric . . . 12
  - 2.4.1. Características Principales . . . 12
  - 2.4.2. Algoritmo de Consenso en Hyperledger Fabric . . . 13
  - 2.4.3. Ausencia de un Token Nativo . . . 13
- 2.5. Interoperabilidad en Blockchains . . . 13
  - 2.5.1. ¿Que es la interoperabilidad? . . . 13
  - 2.5.2. Tipos de interoperabilidad . . . 14
  - 2.5.3. Tipos de operaciones . . . 14
  - 2.5.4. Patrones de interoperabilidad . . . 16

3. Análisis de requerimientos **19**
- 3.1. Introducción . . . 19
- 3.2. Requerimientos funcionales . . . 19
- 3.3. Requerimientos no funcionales . . . 19
- 3.4. Alcance del proyecto . . . 20

3.5. Descripción del escenario propuesto . . . 20
3.6. Casos de Uso . . . 22

4. Análisis de soluciones **25**
4.1. Funcionamiento de los Bridges . . . 25
4.1.1. Wormhole . . . 25
4.1.2. Chainlink . . . 27
4.1.3. LayerZero . . . 28
4.1.4. Conclusión . . . 29
4.2. Método de Pago . . . 30
4.2.1. Celer Bridge . . . 30
4.2.2. Chainlink . . . 31
4.2.3. Allbridge . . . 32
4.2.4. Conclusión . . . 32

5. Diseño de la solución **33**
5.1. Diseño de los patrones . . . 33
5.1.1. Relayer . . . 33
5.1.2. Permanent Transfer . . . 34
5.1.3. Temporal Transfer . . . 35
5.2. Arquitectura del escenario . . . 36
5.2.1. Vista de implementación . . . 37
5.2.2. Vista de Casos de Uso . . . 39
5.2.3. Vista lógica . . . 40
5.2.4. Vista de distribución . . . 42
5.3. Principales decisiones de diseño . . . 43
5.3.1. Pago del gas . . . 43
5.3.2. Base de datos para cada conector EVM . . . 44
5.3.3. Arquitectura basada en Wormhole . . . 44
5.3.4. Emisión de eventos cómo mecanismo de comunicación . . . 44

6. Implementación de la solución **45**
6.1. Tecnologías . . . 45
6.2. Implementación del escenario propuesto . . . 46
6.2.1. Music Discs Smart Contract . . . 46
6.2.2. Concert Tickets Smart Contract . . . 52
6.2.3. Music Discs dApp . . . 54
6.2.4. Concert Tickets dApp . . . 55
6.2.5. Integration dApp . . . 55
6.3. Decisiones de implementación . . . 55
6.3.1. Separación en 3 dApps . . . 55
6.3.2. ERC721 en Hyperledger Fabric . . . 56
6.3.3. Autenticación en Hyperledger Fabric . . . 56

7. Evaluación de la solución **57**
7.1. Análisis de costos . . . 57

8. Gestión del proyecto **61**
8.1. Organización del proyecto 61
8.2. Cronograma final 62

9. Conclusiones y Trabajo Futuro **63**
9.1. Conclusiones 63
9.2. Trabajo futuro 64
9.2.1. Implementar más patrones de interoperabilidad 64
9.2.2. Calculo de tarifa y gas 64
9.2.3. Incorporar un mecanismo de reintento para operaciones fallidas 64

Referencias **67**

A. Anexo Marco Teórico **71**
A.1. Algoritmos de consenso 71
A.2. Patrones de diseño 73
A.3. Soluciones de interoperabilidad 75
A.3.1. Soluciones de interoperabilidad 76

B. Anexo Análisis de Soluciones **79**
B.1. Wormhole 79
B.1.1. Proceso de Transferencia 80
B.2. Chainlink 81
B.2.1. Componentes OffChain: 81
B.2.2. Componentes OnChain 82
B.2.3. Proceso de Transferencia 84
B.3. LayerZero 85
B.3.1. Componentes principales: 85
B.3.2. Proceso de Transferencia 86
B.4. Celer Bridge 87
B.5. AllBridge 87

C. Anexo Diseño de la solución **89**
C.1. Caso de uso 89

D. Anexo Implementación **91**
D.1. Implementación de los Patrones 91
D.1.1. Relayer 91
D.1.2. Transferencia Permanente 99
D.1.3. Transferencia Temporal 106
D.1.4. Despliegue de los contratos 118

Capítulo 1

Introducción

Este capítulo brinda una perspectiva general del documento y una introducción al trabajo desarrollado. En la sección 1.1 se da a conocer la motivación para la elaboración del proyecto. Luego, en la sección 1.2 se exponen los objetivos del proyecto. A continuación, en la sección 1.3 se expresan los aportes realizados. Por último, en la sección 1.4 se describe la estructura del documento.

1.1. Motivación

Blockchain es una tecnología que permite almacenar información de forma inmutable y segura en entornos distribuidos. Desde su aparición en el año 2008 junto a la criptomoneda Bitcoin (Nakamoto, 2008), esta tecnología ha evolucionado más allá del uso para las criptomonedas, convirtiéndose en un pilar de aplicaciones más amplias como contratos inteligentes, trazabilidad, identidades digitales y gobernanza.

En este contexto, se tiene dos tipos principales de blockchains: públicas, como Ethereum y Polygon, y privadas, como Hyperledger Fabric. Las blockchains públicas pueden ser accedidas por cualquier persona, mientras que las privadas son administradas por un conjunto de entidades cerradas. Cada una de ellas propone enfoques particulares en cuanto a descentralización, privacidad, escalabilidad y gobernanza. Sin embargo, uno de los principales desafíos que enfrenta actualmente el ecosistema blockchain es la complejidad y costos a la hora de interoperar entre estas redes. Las blockchains fueron concebidas originalmente como sistemas cerrados, lo que no permite el intercambio directo de datos y activos entre redes sin soluciones de terceros.

La necesidad de lograr interoperabilidad entre blockchains ha cobrado cada vez más relevancia, ya que la comunicación entre blockchains permitiría a los usuarios y aplicaciones operar entre diferentes redes sin inconvenientes. Sin embargo, esta interoperabilidad conlleva desafíos importantes, especialmente cuando se busca conectar blockchains con diferencias fundamentales, como lo son las públicas y las privadas. Estos desafíos van desde diferencias técnicas,

como los mecanismos de consenso y estructuras de datos, hasta cuestiones de privacidad y validación de transacciones.

Actualmente, existen diversas soluciones en el ámbito de la industria (Llambías, González, y Ruggia, s.f.), como Chainlink o Wormhole, que han abordado parcialmente este problema, sin embargo, estas soluciones pueden ser muy costosas. Si se deseara desarrollar una solución de interoperabilidad propia, esto puede resultar complejo y costoso. De cara a facilitar el desarrollo de soluciones de interoperabilidad es que se propusieron un conjunto de seis patrones (Llambías, González, y Ruggia, 2022). Sin embargo, estos patrones están pensados para arquitectos de software, estando definidos a un alto nivel. Este proyecto busca ofrecer una implementación de referencia de tres de estos patrones: Relayer, Transferencia Permanente y Transferencia Temporal. Además, a modo de proveer un ejemplo de uso de los mismos, se implementará un escenario de uso.

1.2. Objetivos

El objetivo principal de este proyecto es proveer una implementación de referencia de los patrones de interoperabilidad Relayer, Transferencia Permanente y Transferencia Temporal, identificados en el artículo (Llambías y cols., 2022). Los objetivos específicos de este proyecto son:

- Investigar las soluciones existentes de interoperabilidad entre blockchains.
- Implementar y documentar los patrones Relayer, Permanent Transfer y Temporal Transfer.
- Validar la dificultad de trasladar la solución en Ethereum a blockchains EVM compatibles, en particular, Polygon.
- Evaluar la solución mediante la implementación de un caso de uso basado en la venta de discos musicales y de entradas a conciertos.
- Evaluación de costos de las implementaciones de los patrones.
- Documentar el trabajo realizado en un informe.

1.3. Aportes del proyecto

Los aportes realizados en el proyecto son los siguientes:

- Relevamiento y análisis de soluciones existentes para interoperabilidad en Blockchain: Se realizó un estudio de las distintas implementaciones de Bridges, incluyendo Chainlink, Wormhole, Celer Bridge, AllBridge y LayerZero.
- Implementación de referencia de 3 patrones de interoperabilidad: Relayer, Permanent Transfer y Temporal Transfer

- Aplicación de patrones mediante caso de uso práctico: Se desarrolla una aplicación para poder mostrar las implementaciones desarrolladas
- Análisis de costos de las implementaciones de los patrones realizadas.
- Documentación: Se crean secciones donde se ven distintos aspectos del diseño e implementación de las soluciones, así como también se mencionan las tecnologías utilizadas.

1.4. Estructura del documento

El documento cuenta de 9 capítulos. El actual y primer capítulo es la introducción. El capítulo 2, es el Marco Teórico, en donde vemos distintos conceptos principalmente del mundo Blockchain. El capítulo 3, Análisis de requerimientos, menciona todos los requerimientos para que el proyecto sea considerado exitoso. El capítulo 4, Análisis de soluciones, expone las distintas maneras de resolver los problemas similares al actual, que servirá cómo base para nuestra propia solución. El siguiente capítulo, capítulo 5, Diseño de la solución, describe de manera detallada nuestra solución propuesta junto a la arquitectura definida, muestra distintos esquemas para facilitar su compresión y justifica las distintas decisiones tomadas. El capítulo 6, Implementación de la solución, muestra en detalle la implementación de cada problema resuelto, se comentan y justifican los métodos utilizados y se adjuntan ejemplos de código. En el capítulo 7 se evalúa el desempeño de nuestra solución en base a sus costos. En el capítulo 8, Gestión del proyecto, se mencionan las metodologías utilizadas para llevar a cabo el proyecto y también se muestra la planificación inicial, comparándola con la ejecución real. Por último, en el capítulo 9, Conclusiones y Trabajo Futuro, se resumen los hallazgos, resultados, aprendizajes y también áreas con posibles mejoras.

Capítulo 2

Marco teórico

En el siguiente capítulo se presentan los conceptos fundamentales que serán utilizados en el transcurso del documento. El capítulo se divide en 5 secciones principales. La sección 2.1 presenta conceptos que permiten comprender los fundamentos de la blockchain, características principales, diferencia entre redes permisionadas y no permisionadas, algoritmos de consenso y el concepto de *tokenización*. En la sección 2.2 se explican algunas particularidades de la red Ethereum, en la sección 2.3 se presenta la red Polygon, en la sección 2.4 la red Hyperledger Fabric y en la sección 2.5 se presenta el problema de la interoperabilidad en las blockchains, diferentes tipos, operaciones, y patrones.

2.1. Blockchain

Una blockchain es un tipo de base de datos pública que se mantiene y comparte entre muchas computadoras en una red (*Ethereum Docs*, s.f.-a). Los encargados de actualizar la red son computadoras llamadas "nodos". Su trabajo consiste en crear, verificar, almacenar y distribuir bloques con los demás miembros de la red. El propósito de esto es que todos los bloques se mantengan al día con información correcta. El nombre "Blockchain", viene de las palabras en inglés para "bloque" y "cadena" que en conjunto significa una cadena de bloques. En este caso, los bloques son un grupo de transacciones almacenadas en una base de datos, con movimientos de dinero u otro tipo de información. Se dice que es una cadena porque justamente todos los bloques están enlazados entre sí, formando una cadena ordenada de bloques. Cada bloque está vinculado a su bloque predecesor mediante hashes criptográficos que garantizan la seguridad del enlace (Llambías y cols., s.f.). El hash de un bloque se construye utilizando los datos del bloque y el hash del bloque anterior. Si una transacción del bloque n cambia, el hash del bloque n también cambiará y, como consecuencia, el hash del bloque n+1 no podrá ser verificado, ya que requeriría un valor de hash diferente para el bloque n. Esta protección permite a los usuarios detectar si la blockchain ha sido alterada. Este comportamiento es el que proporciona

la seguridad característica de blockchain, ya que los bloques manipulados serán descartados por los usuarios.

En Blockchain, cada bloque contiene un Merkle root, que es el hash raíz de un árbol de Merkle (Kuznetsov y cols., 2024). Este árbol de Merkle organiza de manera eficiente las transacciones dentro de un bloque y permite verificar la integridad de todas las transacciones de manera rápida y eficiente. El Merkle root es calculado a partir de los hashes de todas las transacciones del bloque y es incluido en el encabezado del bloque. Este mecanismo mejora la seguridad y la verificación de las transacciones, ya que cualquier cambio en una transacción modificará el Merkle root, lo que a su vez cambiará el hash del bloque y hará evidente cualquier alteración sin necesidad de recorrer el árbol.

2.1.1. ¿Cuáles son las características de la tecnología Blockchain?

Las características principales de la tecnología blockchain son la **descentralización**, la **inmutabilidad** y el **consenso**. (*AWS - What is Blockchain Technology?*, s.f.)

La **descentralización** significa que no dependen de una entidad centralizada (individuo, grupo o organización) ya que se trata de una red distribuida donde cada nodo mantiene una copia del estado actual de la base de datos. Mediante la transparencia, la red logra que las transacciones sean verificables por cualquier usuario de la red, reduciendo los niveles de confianza necesarios entre participantes. La **inmutabilidad** alude a que, una vez que se confirma una transacción, resulta imposible cambiarla, ya que para ello se precisa el consenso de la mayoría de la red. En caso de error, por ejemplo, es necesario registrar una nueva transacción para revertirla, siendo ambas transacciones visibles para la red. El **consenso** implica que solo se pueden registrar transacciones nuevas cuando la mayoría de los participantes de la red dan su consentimiento, siguiendo un conjunto de reglas ya preestablecidas.

2.1.2. Clasificación

Basándose en la capacidad para generar y agregar nuevos bloques, las redes se dividen en ≪sin permisos≫ o permisionless y ≪con permisos≫ o permissioned. A su vez, también es posible clasificarlas en públicas, privadas y de consorcio, donde las públicas suelen ser no permisionadas y las privadas y de consorcio permisionadas.

Blockchains no permisionadas

Son redes públicas y descentralizadas, como Ethereum y Polygon. En estas, cualquier persona puede acceder, participar y validar transacciones sin necesidad de autorización o verificación de identidad. (*Persmissionless Blockchains vs Permissioned Blockchains*, s.f.) No hay una autoridad central que controle

la red y cualquier usuario puede verificar el estado pasado de la blockchain en cualquier momento. Algunas de las características principales de una blockchain no permisionada son: la **transparencia**, todas las transacciones son accesibles y verificables públicamente mediante exploradores de blockchain; el **anonimato**, los usuarios pueden permanecer pseudo-anónimos al utilizar direcciones públicas, no hay necesidad de proporcionar información personal como nombres de usuario o correos electrónicos; la **resistencia a la censura**, ninguna entidad controla la red, lo que minimiza el riesgo de censura de transacciones o direcciones; y el **consenso**, cualquier persona puede participar en el consenso lo cual fortalece la seguridad, reduce la centralización y recompensa a los validadores por su trabajo. Además, pueden enviar transacciones y tener una copia de la base de datos).

Blockchains permisionadas

Son redes privadas, pensadas para empresas o similar, donde una entidad central regula el acceso y las reglas de operación. (*Persmissionless Blockchains vs Permissioned Blockchains*, s.f.) La información de esta blockchain es privada y no se puede participar sin autorización previa. Además, estas redes son jerárquicas, hay participantes con más privilegios de acceso que otros. Algunas características principales de una blockchain permisionada son: la gestión de la **privacidad**, dependiendo del caso de uso, los registros de transacciones pueden mantenerse privados y ocultos para ciertas organizaciones dentro de la red; la **autorización**, varía según la administración de la red que regula los privilegios de acceso de cada usuario; la **censura**, las autoridades y organizaciones involucradas pueden establecer regulaciones específicas según sus necesidades comerciales y censurar a ciertos usuarios o restringir tipos específicos de transacciones que no cumplan con las reglas de la red; y los algoritmos de **consenso**, dado que están orientadas a modelos empresariales, la entidad centralizada puede definir su propio mecanismo de consenso.

Blockchain de consorcio

Una blockchain de consorcio, a diferencia de blockchain privada, no es administrada por una única organización si no que por un conjunto de organizaciones. Por ejemplo, un consorcio de varias organizaciones, donde cada una de las cuales opera un nodo y para que un bloque sea válido, deben firmar la mayoría o siguiendo algunas reglas predefinidas. La lectura puede ser pública o restringida a los participantes. Estas blockchains se pueden considerar parcialmente descentralizadas. (Parrondo, 2018) Un ejemplo de este tipo de blockchain es Hyperledger Fabric.

2.1.3. Algoritmos de consenso

Los algoritmos o mecanismos de consenso son fundamentales para garantizar la integridad y la seguridad de las transacciones de manera descentralizada. No

hay un único nodo central que garantice que los registros de los distintos nodos de la blockchain sean iguales. Entonces, son los distintos nodos que forman parte del la blockchain los que se deben de poner de acuerdo (consensuar) en cuanto al estado de la blockchain, y el historial de transacciones de la misma. Alcanzar un consenso en un ambiente distribuido puede ser un desafío, y existen distintos algoritmos que intentan lograr esto, como **Proof of Work**, **Proof of Stake**, **Crash Fault Tolerance (CFT)**, **Practical Bizantine Fault Tolerance (PBFT)**, **Delegated Proof of Stake (DPOS)** (Zheng, Xie, Dai, y Wang, 2017). En esta sección se describirá solo el **PoS** ya que es el algoritmo de consenso utilizado en Ethereum y Fabric. El detalle del resto de los protocolos se encuentra en el anexo A.1.

Proof of Stake (POS)

Proof of Stake (PoS) es una alternativa de bajo consumo energético a PoW. En PoS, los mineros deben demostrar la propiedad de una cantidad de criptomonedas. Se cree que las personas con mayores cantidades de monedas tienen menos probabilidades de atacar la red.

La selección basada únicamente en el saldo de la cuenta sería injusta, ya que la persona más rica dominaría la red. Como resultado, se han propuesto varias soluciones que combinan el tamaño de la participación (stake) para determinar quién forjará el siguiente bloque. En particular, en Ethereum, se selecciona de manera aleatoria a un validador para que proponga un bloque (*Ethereum Docs*, s.f.-b). Este validador será responsable de crear un nuevo bloque y de enviarlo a otro nodo de la red. Por otra parte, se escoge de manera aleatoria a un comité de validadores, cuyos votos se utilizarán para determinar la validez del bloque propuesto. Dividir la configuración del validador en comités es importante para mantener la carga de la red manejable. Los comités dividen el conjunto de validadores para que todos los validadores activos certifiquen cada cierto período de tiempo.

Otras blockchains como por ejemplo, Blackcoin, utiliza randomización para predecir el próximo generador (*BlackCoin's Proof-of-Stake Protocol v2*, s.f.). Emplea una fórmula que busca el valor hash más bajo en combinación con el tamaño de la participación. Peercoin favorece la selección basada en la edad de las monedas (King y Nadal, 2012). En Peercoin, los conjuntos de monedas más antiguos y grandes tienen una mayor probabilidad de minar el siguiente bloque.

Comparado con PoW, PoS consume menos energía y es más eficiente. Sin embargo, dado que el costo de minería es casi nulo, esto podría aumentar el riesgo de ataques. (Zheng y cols., 2017)

2.1.4. Finalidad de transacciones

La finalidad de transacciones se refiere al punto en el cual un bloque añadido a la cadena se considera inmutable, es decir, que no puede ser eliminado o revertido. La finalidad puede ser de tipo probabilística o determinista (Anceaume, Del Pozzo, Rieutord, y Tucci-Piergiovanni, 2022). Por ejemplo, en casos donde

se generen bifurcaciones, cuanto más larga sea la cadena luego del bloque, más improbable es que exista otra cadena más larga. Dependiendo del algoritmo de consenso utilizado, cuan larga ha de ser la cadena para que la probabilidad sea nula y se considere entonces, el bloque por finalizado. Por otro lado, la finalidad determinista puede ser inmediata o eventual. Esto dependerá exclusivamente del propio algoritmo de consenso utilizado.

2.1.5. Smart Contracts

Un *smart contract* es un conjunto de reglas programadas que se ejecutan automáticamente en una blockchain cuando se cumplen ciertas condiciones (Mohanta, Panda, y Jena, 2018). A diferencia de los contratos tradicionales, los *smart contracts* no requieren intermediarios y ofrecen mayor transparencia y seguridad. Al operar sobre blockchain, ya se cuenta con descentralización, seguridad y transparencia intrínsecamente. Los *smart contracts* son una parte clave de blockchain que están transformando múltiples sectores al mejorar la eficiencia, reducir costos y eliminar la necesidad de intermediarios. Algunos de estos sectores son las finanzas, seguros, cadena de suministros y la gestión de identidad digital, así como de otros activos.

2.1.6. Tokenización

La tokenización refiere al proceso de convertir activos físicos o digitales en unidades digitales llamadas tokens, que se registran y gestionan en una blockchain. Estos tokens actúan como representaciones digitales de activos reales, permitiendo su gestión y transferencia de manera más eficiente y segura, sin necesidad de un intermediario. Estos activos pueden ser digitales como acciones, contratos o físicos como médicos (historiales de pacientes), inmobiliarios (la propiedad de un inmueble), entre otros. (Hernandez Casco, 2024)

En las próximas subsecciones se describirán dos tipos de tokens, **fungibles** y **no fungibles**.

Tokens fungibles

Son activos digitales que son intercambiables entre sí y poseen el mismo valor (*Ethereum Docs - ERC-20 Token Standard*, s.f.). Cada unidad es idéntica a otra del mismo tipo. Un ejemplo de token fungibles son Bitcoin (BTC) o Ether (ETH). En particular, en Ethereum, el ERC-20 (Ethereum Request for Comment) introduce un estándar para Tokens Fungibles.

Tokens no fungibles (NFT)

Son activos digitales únicos que no pueden ser intercambiados de manera equivalente (*Ethereum Docs - ERC-721 Non-Fungible Token Standard*, s.f.). Cada NFT posee información específica que lo hace diferente de cualquier otro. Por ejemplo, puede ser utilizado para objetos coleccionables o asientos numerados

para conciertos. Estos tokens también pueden contener un uri que apunte hacia su metadata. Al igual que con los tokens fungibles, en Ethereum se introdujo un estándar para los Tokens no Fungibles: el ERC-721.

2.2. Ethereum

Ethereum es una blockchain con la particularidad de que tiene una computadora integrada (*Ethereum Docs*, s.f.-a). El propósito de esta es poder crear y alojar aplicaciones con todas las características de una blockchain: descentralizada, sin permisos, y resistente a la censura.

En el mundo de Ethereum, existe una única computadora a la que cada nodo de la red puede acceder y realizar ejecuciones de código. Es trabajo de los nodos mantener el estado e integridad de esta computadora. Cada vez que se ejecuta un programa en esta computadora, los nodos validan, ejecutan la computación, además de propagar el cambio de estado. A estas peticiones de cómputo se les llaman transacciones.

Un *smart contract* en Ethereum es simplemente un programa que se puede cargar y ejecutar en la Ethereum Virtual Machine (EVM). Cualquier desarrollador puede crear un programa para Ethereum y subirlo a la red, por un precio. Para ejecutar estos programas se debe enviar una transacción que también se debe pagar un precio.

Para pagar por estas transacciones se utiliza Ether. Ether es la criptomoneda nativa de Ethereum que permite un incentivo económico detrás de todos los nodos que mantienen la red. Cada vez que se desea ejecutar un *smart contract* se debe pagar con Ether, en donde una parte de este se *quema* y la otra queda cómo recompensa al que verifica la transacción

Los programas que se pueden utilizar pueden ser tan complejos cómo uno quiera, por ejemplo, se pueden crear redes sociales, juegos, sistemas de votación anónimos y descentralizados, etc.

2.2.1. ¿Qué es el Gas en Ethereum?

Para ejecutar *smart contracts* y las distintas transacciones en la red, se precisan recursos computacionales y una manera de medir cuanto se usan. Para esto se usa el "gas", ya que para realizar ejecuciones en la red se utiliza recursos, se debe pagar una tarifa por estos, de lo contrario podría haber ataques spam o errores en el código en donde un programa nunca termine y quede estancada la red.

Cómo se calcula el gas a pagar

El gas a pagar se calcula como: (*Ethereum Docs - Gas*, s.f.)

$$\text{Gas utilizado} \times (\text{Tarifa base} + \text{Tarifa de prioridad})$$

- **Tarifa base**: Establecida por el protocolo, necesaria para que la transacción sea válida.
- **Tarifa de prioridad (propina)**: Incentivo opcional para que los validadores procesen tu transacción más rápido.
- Cualquier gas no utilizado en una transacción se devuelve al usuario.

2.2.2. Finalidad en Ethereum

En Ethereum, una transacción tiene **finalidad** cuando se encuentra en un bloque que no puede modificarse sin que se queme una gran cantidad de ETH (*Ethereum Docs*, s.f.-b). Se tienen los conceptos de **slots** y **epochs**. El tiempo en Ethereum se divide en estos **slots**, que duran **12 segundos**, y es cuando un validador puede proponer un bloque. Una **epoch** esta compuesta por **32 slots** (**6.4 minutos**). El **primer** bloque de cada **epoch** se denomina **checkpoint**, donde los validadores votan. Si al menos $\frac{2}{3}$ del ETH en stake respalda un **checkpoint**, este se **justifica**. Luego, si en la siguiente **epoch** otro **checkpoint** se justifica, el anterior pasa de **justificado** a **finalizado**. Esto se puede ver representado gráficamente la figura 2.1.

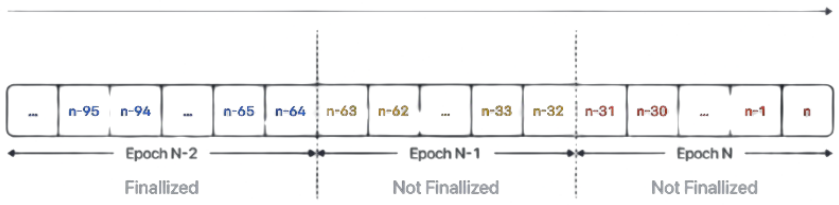


Figura 2.1: Finalidad en Ethereum POS

2.3. Polygon

Polygon es un protocolo para construir y conectar redes blockchain compatibles con Ethereum (*Polygon Doc*, s.f.). Su propósito principal es mejorar la escalabilidad de Ethereum al proporcionar soluciones de segunda capa (Layer 2) que permiten transacciones rápidas y económicas sin comprometer la seguridad ni la descentralización.

Polygon actúa como un ecosistema de cadenas laterales (sidechains) que están diseñadas para expandir la funcionalidad de Ethereum. A diferencia de una blockchain independiente, Polygon aprovecha la seguridad y la infraestructura

de Ethereum, mientras que permite transacciones más rápidas y baratas. La criptomoneda nativa de Polygon es `POL` (previamente denominada `MATIC`).

2.3.1. Características principales de Polygon

Algunas de las características principales de Polygon son: la **escalabilidad**, Polygon permite procesar transacciones en sidechains que corren en paralelo a la blockchain principal de Ethereum, permitiendo grandes volúmenes de transacciones; la **velocidad**, el diseño de la red facilita el procesamiento rápido de transacciones, lo que es fundamental para las aplicaciones que dependen de confirmaciones rápidas; **costos reducidos**, las tarifas de transacciones en Polygon suelen ser significativamente más baratas que las realizadas directamente en la blockchain de Ethereum; y una **experiencia de usuario mejorada**, al ofrecer tiempos de transacción rápidos y costos bajos, Polygon proporciona una experiencia más accesible para los usuarios de aplicaciones descentralizadas). (*Understanding the Polygon Blockchain: Benefits and Use Cases*, s.f.)

Polygon utiliza el modelo de consenso Proof of Stake (Prueba de Participacion) mencionado anteriormente, donde los validadores apuestan tokens `POL` para participar en la validación de transacciones. (*Polygon Doc*, s.f.)
Además está diseñado para ser compatible con Ethereum, lo que significa que cualquier `smart contract` de Ethereum puede implementarse en la red Polygon sin necesidad de modificaciones. Esto permite que las dApps existentes en Ethereum se beneficien de la escalabilidad de Polygon sin perder compatibilidad con la infraestructura de Ethereum. (*Polygon Doc*, s.f.)

2.4. Hyperledger Fabric

Hyperledger Fabric es una plataforma de "Tecnología de Contabilidad Distribuida" (*Distributed Ledger Technology*, DLT) de código abierto y nivel empresarial (*Fabric Docs*, s.f.). Está diseñada para la implementación de redes blockchain permisionadas, lo que significa que los participantes de la red están identificados y autorizados.

2.4.1. Características Principales

Hyperledger Fabric se distingue de otras plataformas blockchain debido a sus características innovadoras:

- **Arquitectura Modular:** Permite personalizar distintos componentes, como mecanismos de consenso, gestión de identidad y almacenamiento de datos.
- **Smart Contracts con Lenguajes Generales:** A diferencia de Ethereum, donde se usa Solidity, Fabric permite desarrollar contratos inteligentes en Java, Go y JavaScript.

- **Red permisionada:** Los participantes deben estar identificados, autenticados y autorizados antes de interactuar con la red, lo que brinda mayor seguridad al sistema.
- **Privacidad y Confidencialidad:** Implementa canales privados dentro de la red, lo que permite que solo ciertas partes accedan a información específica de una transacción.

2.4.2. Algoritmo de Consenso en Hyperledger Fabric

Hyperledger Fabric no utiliza un único algoritmo de consenso, sino que permite elegir entre diferentes protocolos según las necesidades de la red. Su enfoque modular permite implementar mecanismos de consenso personalizables, que se adapten al caso de uso. Algunos de los algoritmos soportados son **Crash Fault Tolerance (CFT)** y **Byzantine Fault Tolerance (BFT)**

2.4.3. Ausencia de un Token Nativo

A diferencia de muchas plataformas blockchain que dependen de una criptomoneda nativa para incentivar la seguridad y ejecución de contratos inteligentes, Hyperledger Fabric no requiere un token propio. Esto se debe a que, al ser una blockchain permisionada, los participantes están identificados y autorizados, y las transacciones no se validan por un sistema de incentivos económico como *PoS*, por lo que no se necesita un token para recompensar o incentivar a los validadores. (*Fabric Docs*, s.f.)

2.5. Interoperabilidad en Blockchains

En esta sección se hablará de varios conceptos de interoperabilidad en blockchain, como su definición, los distintos tipos de interoperabilidad, los tipos de operaciones que se pueden realizar, algunos de los patrones de soluciones de interoperabilidad que existen, así como algunas de las soluciones existentes.

2.5.1. ¿Que es la interoperabilidad?

Las blockchains por definición son sistemas de información cerrados que no son capaces de comunicarse con otros sistemas o blockchains. La búsqueda por la interoperabilidad entre blockchains surge entonces de la necesidad de conectar estos sistemas para poder lograr que puedan trabajar en conjunto, pudiéndose realizar operaciones conjuntas, transferencias de activos y validaciones de transacciones entre redes públicas y privadas. Según el IEEE, la interoperabilidad se puede definir como "la capacidad de dos o mas sistemas o componentes para intercambiar información y utilizar la información intercambiada" (Pillai, Biswas, Hóu, y Muthukkumarasamy, 2022). En el contexto de blockchain, esto implica la posibilidad de ejecutar transacciones entre diferentes redes sin comprometer la integridad ni la seguridad de los datos.

2.5.2. Tipos de interoperabilidad

Hay distintos tipos de interoperabilidad en blockchain, basándose en los tipos de sistemas que se desean "interoperar", así como del tipo de blockchains. En el caso de las blockchains, dependerá si estas son publicas, privadas o de consorcio. Por un lado, se tiene el caso que una aplicación (por ejemplo, una dApp) quiera interactuar con una blockchain, como se puede ver en la Figura 2.2 (A). Se tiene también el caso en el que una blockchain quiera operar con otra blockchain (Figura 2.2 (B)). Finalmente, se tiene el caso donde se desea que una blockchain opere con una aplicación, como se puede ver en la Figura 2.2 (C) (Llambías y cols., s.f.)

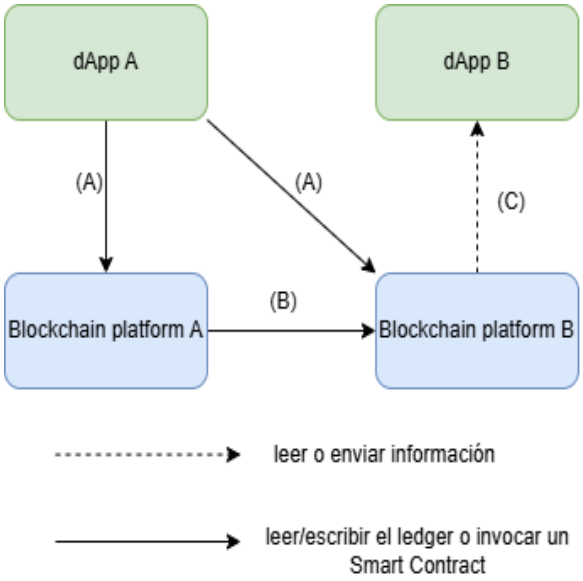


Figura 2.2: Tipos de interoperabilidad (Llambías y cols., s.f.)

2.5.3. Tipos de operaciones

Existen tres tipos de operaciones de interoperabilidad entre blockchains. Estos son *Query*, *Command*, y *Coordinated*. Consideremos entonces dos blockchains, una de origen, y otra destino.(Llambías y cols., s.f.)

Query

En esta operación, la blockchain origen le solicita información a la blockchain destino. La respuesta de esta solicitud contiene una prueba que le asegura a la blockchain origen que la información es valida y que fue corroborada por el mecanismo de consenso de la blockchain destino. La información, junto a su prueba, queda guardada en la blockchain origen.

Command

Las operaciones Command son operaciones unidireccionales desde la blockchain origen hacia la destino. En estas operaciones se quiere enviar información desde la blockchain origen a la destino, junto a la prueba correspondiente que asegure que la información fue verificada por el mecanismo de consenso de la blockchain origen. La blockchain destino valida la prueba, y puede guardar la información si lo desea. Un ejemplo de esta operación es la transferencia de *assets*.

Coordinated

Estas operaciones requieren de una coordinación entre la blockchain origen y destino para la realización de transacciones que sean atómicas y consistentes según las propiedades ACID. En estas transacciones se tiene una acción en la blockchain origen, un mensaje que se envía a la blockchain destino, y una acción en la blockchain destino luego de haber recibido el mensaje. Las acciones en ambas blockchains deben ejecutarse atómicamente, y si la acción en la blockchain destino falla en procesar el mensaje, la acción en la blockchain origen se debe de revertir. Estas acciones también deben garantizar consistencia de datos. Un ejemplo de esta acción es el intercambio de *assets*.

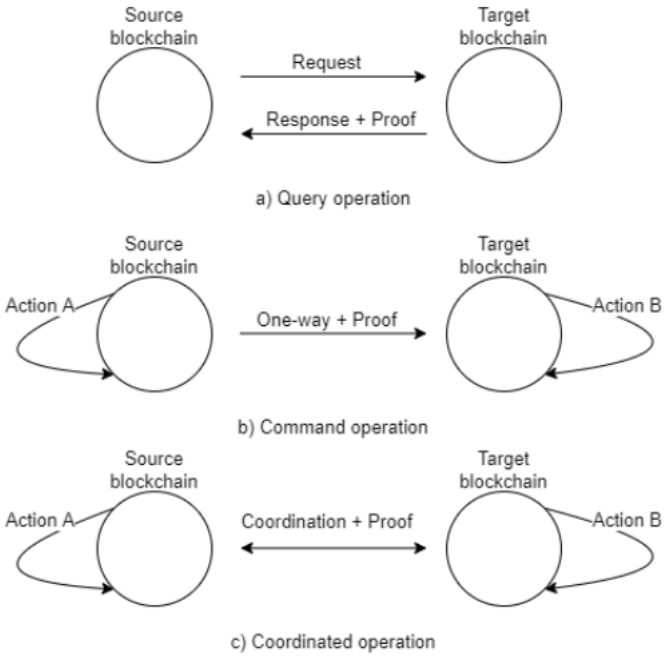


Figura 2.3: Tipos de operaciones (Llambías y cols., s.f.)

2.5.4. Patrones de interoperabilidad

Similar a los patrones de diseño en el área de desarrollo de software, es de gran interés la investigación y desarrollo de patrones de interoperabilidad, que ayuden a desarrolladores a desarrollar soluciones de interoperabilidad. Estos patrones pueden ayudar a reducir la complejidad y los costos de implementación al proporcionar estructuras ya definidas para, por ejemplo, la transferencia de *assets*, ejecución de transacciones, y validación de datos entre redes. Entre estos patrones de interoperabilidad se encuentran el *Relayer*, *API Gateway*, *Transaction Aggregator*, *Light Client Permanent Transfer*, y *Permanent Transfer* y *Temporal Transfer*, los cuales pueden ser categorizados en 4 categorías principales: infraestructura, seguridad, migración de datos y aplicaciones cross-chain (Llambías y cols., 2022). A continuación se van a describir los patrones más relevantes para el proyecto. El resto de los patrones se verán en el Anexo A.2.

Patrón Relayer

El *Relayer* es un patrón que facilita la comunicación entre dos blockchains, actuando como un *middleware* que escucha las transacciones realizadas en una blockchain origen, y las envía hacia una blockchain destino (Figura 2.4). El *relayer* es capaz de convertir el formato de la información de la transacción de la blockchain origen al formato de información de la blockchain destino. También firma la transacción en la blockchain destino de parte de la blockchain origen.

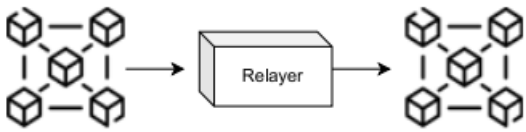


Figura 2.4: Patrón Relayer (Llambías y cols., 2022)

Patrón Permanent Transfer

El patrón *Permanent Transfer* se utiliza cuando se quiere realizar la transferencia permanente de un *asset* de una blockchain origen a una destino (Figura 2.5). Esto implica la eliminación (*burn*) del *asset* en la blockchain origen, así como la creación (*mint*) de un asset en la blockchain destino.

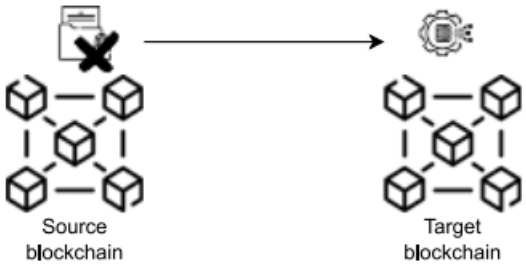


Figura 2.5: Patrón Permanent Transfer (Llambías y cols., 2022)

Patrón Temporal Transfer

El *Temporal Transfer* se utiliza cuando se quiere transferir temporalmente un *asset* de una blockchain origen a una destino, pudiendo ser devuelto a la blockchain original cuando se desee (Figura 2.6). Esto se hace mediante el bloqueo (*lock*) del *asset* en la blockchain origen, creándose (*mint*) el *asset* en la blockchain destino. Una vez que se desea devolver el *asset*, se elimina (*burn*) el *asset* en la blockchain destino, y se desbloquea (*unlock*) el *asset* en la blockchain origen.

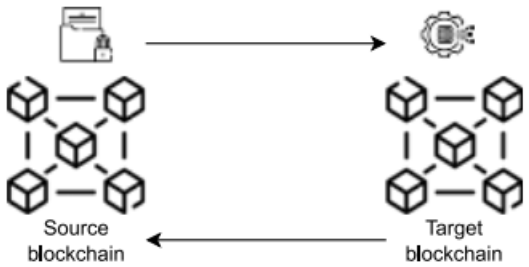


Figura 2.6: Patrón Temporal Transfer (Llambías y cols., 2022)

Capítulo 3

Análisis de requerimientos

3.1. Introducción

En este capítulo se expondrán los distintos requisitos funcionales y no funcionales obtenidos en la etapa de relevamiento de requisitos del proyecto. La sección 3.2 consta de una tabla de los requerimientos funcionales, mientras que en la sección 3.3 se describen los requerimientos no funcionales. En la sección 3.4 se define el alcance del proyecto y en la sección 3.5 se describe el escenario propuesto. Finalmente, en la sección 3.6 se listan los casos de uso de la solución.

3.2. Requerimientos funcionales

En la tabla 3.1 se presentan los requerimientos funcionales del proyecto.

3.3. Requerimientos no funcionales

A continuación, se presentan los requerimientos no funcionales del proyecto:

- Solución reutilizable: El sistema debe ser diseñado para que pueda ser reutilizado en proyectos futuros, permitiendo la adaptación a nuevas funcionalidades o blockchains sin necesidad de una refactorización significativa.
- Claridad del código: El código debe ser claro y comprensible, facilitando la lectura y entendimiento del mismo.
- Documentación: Toda la solución debe estar acompañada de una documentación completa que explique tanto el diseño como el funcionamiento del sistema. Esto debe incluir explicaciones detalladas de los procesos y patrones de interoperabilidad utilizados.
- Costos: La solución debe ser económicamente viable.

Identificador	Descripción
REQ 1	Realizar una implementación del patrón de interoperabilidad Relayer entre las blockchains Ethereum, Polygon y Fabric reutilizando la solución de proyectos de grado anteriores.
REQ 2	Realizar una implementación del patrón de interoperabilidad Transferencia Temporal entre las blockchains Ethereum, Polygon y Fabric.
REQ 2.1	Realizar una transferencia temporal de un activo entre Ethereum y Polygon de manera bidireccional.
REQ 2.2	Realizar una transferencia temporal de un activo entre Ethereum y Fabric de manera bidireccional.
REQ 2.3	Realizar una transferencia temporal de un activo entre Polygon y Fabric de manera bidireccional.
REQ 3	Realizar una implementación del patrón de interoperabilidad Transferencia Permanente entre las blockchains Ethereum, Polygon y Fabric.
REQ 3.1	Realizar una transferencia permanente de un activo entre Ethereum y Polygon de manera bidireccional.
REQ 3.2	Realizar una transferencia permanente de un activo entre Ethereum y Fabric de manera bidireccional.
REQ 3.3	Realizar una transferencia permanente de un activo entre Polygon y Fabric de manera bidireccional.
REQ 4	Implementar el escenario descrito en la sección 3.5.

Tabla 3.1: Requerimientos funcionales del proyecto

3.4. Alcance del proyecto

El alcance final del proyecto logrado abarca a todos los requisitos mencionados anteriormente. Para el escenario descrito en la sección 3.5, se consiguió implementar los patrones de interoperabilidad Relayer, Transferencia Temporal, y Transferencia Permanente, para las distintas blockchains previamente establecidas.

3.5. Descripción del escenario propuesto

La necesidad de una solución como *TransferenciaYA* surge en un ecosistema digital donde los amantes de la música compran e intercambian discos de sus artistas favoritos en formato digital, representados como NFTs (tokens no fungibles).

Por un lado, hay una empresa *DiscosOnline* que se encarga de gestionar la compra de discos musicales en las redes de Ethereum y Polygon. Y por otro, hay otra empresa *TicketChain* que gestiona la venta de entradas a eventos,

particularmente de conciertos musicales en la blockchain de Fabric.

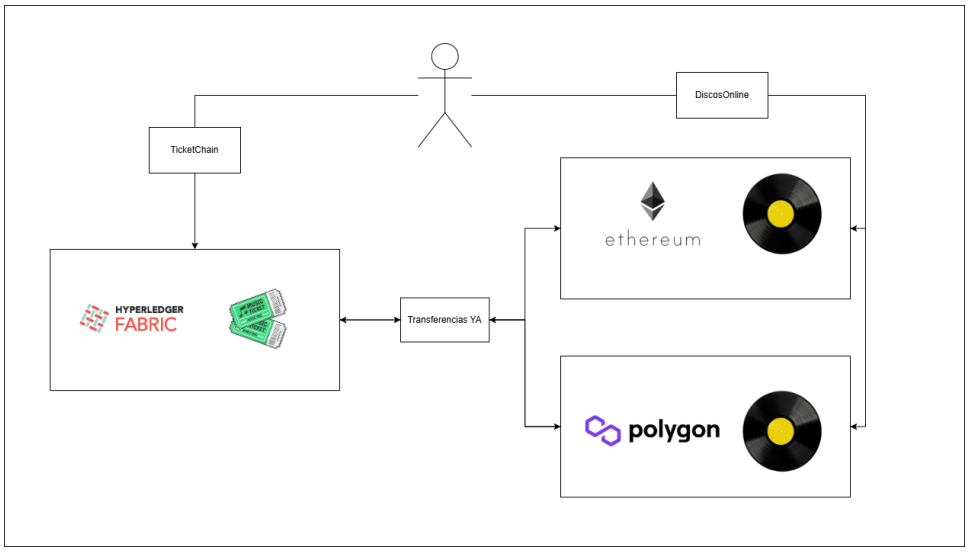


Figura 3.1: Escenario propuesto

La empresa *DiscosOnline* y *TicketChain* deciden llegar a un convenio en donde aquellos usuarios que compren discos de música, podrán aprovecharlos para poder obtener una promoción en conciertos de las bandas autoras de dichos discos, permitiéndoles acceder a un *2x1*.

Por ejemplo, un usuario que tiene discos musicales en *DiscosOnline* quiere comprar entradas para ver un concierto. Para obtener el beneficio de 2x1, el usuario debe transferir temporalmente su disco a la blockchain en la que opera *TicketChain*. Sin embargo, ambas empresas operan en blockchains diferentes y no cuentan con una manera de interoperar entre distintas blockchains. Para esto, contratan a *TransferenciaYA* que les permite integrarse ofreciendo una solución de interoperabilidad entre blockchains distintas. Cuando el usuario quiera transferir su disco, va a invocar a los *smart contract* de *TransferenciaYA* en la blockchain origen, lo cual desencadenará la invocación de los *smart contract* de *TransferenciaYA* en la blockchain destino para realizar *mint* del NFT del disco. Luego, el usuario podrá hacer uso del disco en la blockchain de *TicketChain* para reclamar el descuento. Cuando desee devolver su disco a la blockchain original, análogamente se invocará al *smart contract* de *TransferenciaYA* realizando el camino inverso. Además, *TransferenciaYA* también le ofrece a los usuarios de *DiscosOnline* la posibilidad de migrar sus discos entre las distintas blockchains soportadas por *DiscosOnline*.

En este escenario, *TransferenciaYA* actúa como el puente entre las diferentes blockchains: Ethereum y Polygon, que son públicas, y Hyperledger Fabric, una blockchain privada. Este sistema proporciona interoperabilidad, es decir, los usuarios pueden migrar y prestar sus NFTs de manera segura y transpa-

rente, sin importar en qué blockchain se encuentren, brindando una solución de interoperabilidad a las empresas *DiscosOnline* y *TicketChain*.

3.6. Casos de Uso

En esta sección se verán los Casos de Uso del escenario. A partir del escenario descrito, se pueden identificar tres casos de uso: Comprar un Disco, Migrar un Disco, y Comprar una Entrada.

Nombre	Compra de Disco
Actores	Usuario
Actividades	Comprar un Disco
Descripción	El usuario ingresa a la dApp de discos musicales y realiza una compra de un disco.
Pre-condiciones	El usuario tiene los fondos suficientes para comprar el disco.
Post-condiciones	Al usuario se le desacredita el costo del disco, y se realiza *mint* de un nuevo disco NFT en la blockchain, con el usuario como dueño del mismo, quedando vinculado el NFT a su billetera digital.

Tabla 3.2: Caso de Uso: Comprar Disco

Nombre	Migrar Disco
Actores	Usuario
Actividades	Transferir Permanentemente un Disco a otra blockchain
Descripción	El usuario ingresa a la dApp de discos musicales, e indica que desea realizar una transferencia permanente. Este es redirigido a la dApp integradora, a través de la cual realiza la transferencia.
Pre-condiciones	El usuario es dueño del disco en la blockchain origen.
Post-condiciones	El usuario es dueño del disco en la blockchain destino, tiene un NFT semánticamente equivalente en la billetera del usuario en la blockchain destino.
Post-condiciones	El usuario ya no es dueño del disco en la blockchain origen.

Tabla 3.3: Caso de Uso: Migrar un Disco

Nombre	Compra de una Entrada
Actores	Usuario
Actividades	Comprar una Entrada
Descripción	El usuario accede a la dApp de entradas de conciertos e indica que desea realizar la compra de una entrada a un concierto. En el caso que el usuario quiera acceder a la promoción de *2x1*, se lo redirige a la dApp integradora, a través de la cual puede transferir temporalmente uno de sus discos de la banda para la cual se esta comprando una entrada. Luego de que se realizó esta transferencia, y de que el usuario haya comprado las entradas, se redirige nuevamente a la dApp integradora para devolver el disco a su blockchain original.
Pre-condiciones	El usuario tiene los fondos para comprar una entrada
Post-condiciones	En caso de haberse prestado un disco para el descuento *2x1*, se realiza un *mint* de dos entradas NFT para el concierto, se les asocia a su usuario en Fabric, y se devuelve el disco a su blockchain original. De lo contrario, se realiza *mint* de una única entrada.

Tabla 3.4: Caso de Uso: Comprar una entrada

Capítulo 4

Análisis de soluciones

En esta sección se analizará el funcionamiento de algunos ejemplos de Bridges muy utilizados en la industria. A partir del artículo (Llambías y cols., 2022) se obtuvieron múltiples soluciones existentes para los distintos patrones de interoperabilidad. En particular, en esta sección se verán las soluciones de Wormhole, Chainlink, LayerZero, Celer Bridge y Allbridge. Se dividirá el análisis en dos aspectos. En la sección 4.1 se analizará cómo funciona el Bridge, su arquitectura y componentes, y en la sección 4.2 se discutirá cómo se cubren los costos de realizar las transacciones. Al final de ambas secciones se presentarán las conclusiones obtenidas. El análisis realizado en esta sección será de alto nivel, delegando un análisis más detallado al Anexo B.

4.1. Funcionamiento de los Bridges

Existen una gran variedad de soluciones de Bridges que permiten realizar transferencias temporales y/o permanentes de *assets*. En esta sección se verán tres soluciones en particular, Wormhole, Chainlink y LayerZero. Para la investigación de estas soluciones, se utilizó tanto la documentación disponible como los repositorios de código abierto de las mismas.

4.1.1. Wormhole

El proyecto Wormhole asiste a los desarrolladores a crear aplicaciones que puedan operar a través de múltiples blockchains. Provee un protocolo de pasaje de mensajes genérico que permite la transferencia de datos y *assets* entre varias redes de blockchains, sin necesidad de depender de una entidad centralizada (*Wormhole Docs*, s.f.). Mientras que Wormhole provee una solución para la transferencia **temporal** de *assets* entre blockchains, no cuenta con la posibilidad de realizar una transferencia **permanente** de *assets* ya que para la misma utiliza un mecanismo de *lock-and-mint*. En cuanto a la finalidad, dependiendo del algoritmo de consenso de la blockchain, Wormhole espera un tiempo

determinado a que el bloque se finalice, antes de validar y enviar el mensaje (*Wormhole Finality*, s.f.).
Los componentes de la arquitectura de Wormhole se pueden visualizar en la figura 4.1. Para ilustrar una transferencia, utilizaremos un escenario de ejemplo basado en préstamos de NFT entre blockchains, pudiéndose ver así como los distintos elementos que componen a Wormhole llevan esto a cabo. Primero, se tienen *smart contracts* provistos en cada una de las blockchains soportadas por Wormhole, los cuales se encargan de inicializar y recibir las transferencias entre blockchains (*Wormhole Repo - NFTBridge.sol*, s.f.). Si se está realizando el préstamo de un NFT hacia otra blockchain, el usuario o *dApp* realizará una transferencia del NFT hacia este *smart contract* de Wormhole. Esto actúa como un bloqueo del mismo, ya que el NFT pasa a ser propiedad del *smart contract* de Wormhole, quedando este inutilizable para el usuario o *dApp* en cuestión. Luego, estos *smart contracts* emiten un evento en la blockchain notificando acerca de dicha transferencia. Estos eventos son detectados por una red de "guardianes", los cuales son un grupo de validadores descentralizados que proporciona Wormhole por fuera de las blockchains. Estos guardianes monitorizan la actividad en las distintas blockchains, y una vez que identifican un mensaje de transferencia, lo validan y generan una prueba criptográfica de que la transacción fue aprobada. Luego de generarse esta prueba, un *relayer* provisto por Wormhole se encarga de enviar la misma al *smart contract* de Wormhole encargado de las transferencias en la blockchain destino. Este contrato verifica la validez de la prueba, y procede a realizar un *mint* de un *Wrapped Token*, lo cual es un NFT en la blockchain destino que representa al NFT prestado desde la blockchain origen. Finalmente, este *Wrapped Token* es transferido al destinatario.
A la hora de retornar un NFT que fue prestado, el flujo es muy similar, donde el usuario o *dApp* transfieren el *Wrapped Token* al *smart contract* de Wormhole. Al ser un *Wrapped Token*, el *smart contract* realiza un *burn* del mismo, ya que se lo está devolviendo a su usuario y blockchain original. Luego, se emite un evento, se realiza una validación a través de los guardianes y el *relayer* envía la prueba a la blockchain original del NFT. El *smart contract* de la blockchain originaria del NFT valida la prueba, y verifica que se trata de un NFT que se esta devolviendo, por lo que simplemente realiza la transferencia del NFT hacia su dueño original.

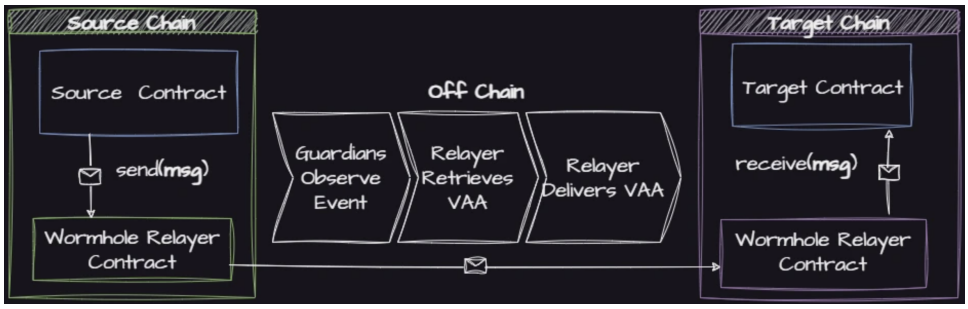


Figura 4.1: Arquitectura de Wormhole (*Wormhole Doc - Run a Custom Relayer*, s.f.)

4.1.2. Chainlink

El protocolo Chainlink CCIP (Cross-Chain Interoperability Protocol) permite a los desarrolladores construir aplicaciones que operen a través de múltiples blockchains, proporcionando un sistema seguro y descentralizado para la transferencia de datos y *assets* entre distintas redes sin depender de una entidad centralizada. A diferencia de Wormhole, Chainlink permite transferencias tanto **temporales** como **permanentes** de *assets*. Además, ofrece múltiples capas de seguridad a lo largo del flujo de una transferencia, y procura la finalidad de las transacciones antes de ser propagadas en las blockchain destino. Veamos a alto nivel el flujo de una transferencia de un token de una blockchain origen a una destino en Chainlink, así como algunos de los componentes que entran en juego.

En la Figura 4.2 (*Chainlink Doc - CCIP Architecture*, s.f.) se puede ver los distintos componentes de CCIP que permiten la transferencia de un token. Se explicará brevemente la función de cada componente, entrando más a detalle en cada uno en el Anexo B.2. Primero, se tiene un usuario en la blockchain origen que desea transferir un token a una blockchain destino. Este se comunica con el **Router**, el cual reenvía el mensaje al **OnRamp** *smart contract*. El **OnRamp** es la primera capa de seguridad de CCIP, realizando validaciones (consultando a la **Risk Management Network**), así como también encargándose del bloqueo (transferencia temporal) o *burn* (transferencia permanente o devolución de token prestado) de los tokens a transferir (utilizando el **Token Pool** *smart contract*). Luego emite un evento, el cual es monitoreado por el **Commiting DON** (**Red de Oráculos Descentralizada**). Esta red de oráculos valida la transacción, y una vez que esta fue aprobada y finalizada, el **Executing DON** la ejecuta en la blockchain destino, enviándola al **OffRamp** *smart contract*. Este valida nuevamente la transacción, consultando a la **Risk Management Network**, y si es aprobada, desbloquea (en caso de devolución) o realiza *mint* (en caso de transferencia permanente o préstamo de un *Wrapped Token*) de los tokens. Estos son enviados al **Router**, para luego ser enviados al usuario destinatario.

Como se puede ver, gracias a los componentes **OnRamp**, **OffRamp**, **Committing DON**, **Executing DON**, así como la **Risk Management Network**, Chainlink cuenta con varias capas de validación y seguridad, minimizando el riesgo de ataques o validaciones incorrectas. (*Chainlink Doc - CCIP Architecture Security*, s.f.)

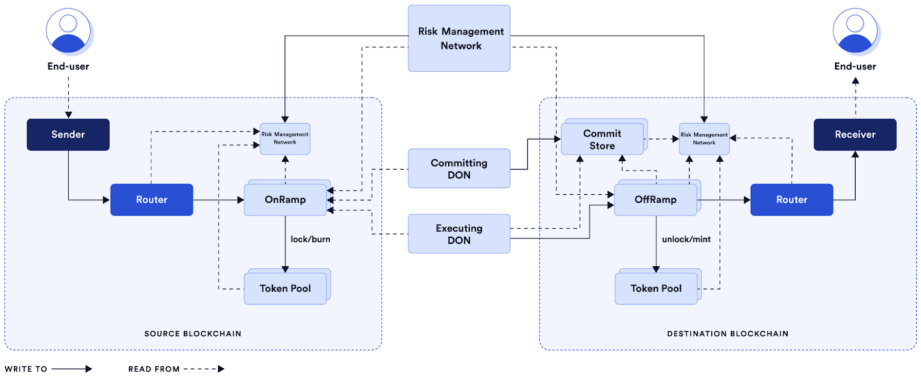


Figura 4.2: Arquitectura del protocolo CCIP de Chainlink

4.1.3. LayerZero

LayerZero es un protocolo de interoperabilidad que permite la comunicación entre blockchains similar a Wormhole y Chainlink, pero que le permite a las aplicaciones definir sus propias configuraciones de seguridad y costos. Implementa el envío de mensajes genéricos y soporta tanto la transferencia permanente como la temporal de tokens, usando *burn/mint* y *lock/unlock*. También toma en cuenta la finalización de las transacciones, ya que dependiendo de la blockchain destino y de la configuración realizada, se espera un número suficiente de confirmaciones de bloques antes de propagar la transacción (*Layer Zero Docs - EVM DVN and Executor Configuration*, s.f.). La arquitectura de LayerZero puede ser visualizada en la figura 4.3. Los componentes interactúan de la siguiente manera: cuando un usuario quiere realizar una transferencia a otra blockchain, este genera un mensaje de transferencia, el cual es manejado por un **LayerZero Endpoint**. Este encapsula la información en un formato estándar y la envía a la **Message Library**, un módulo configurable que define la estructura del mensaje y sus reglas de validación. Luego, el mensaje es transmitido a través de una combinación de dos entidades de LayerZero: **Decentralized Verifier Networks (DVNs)** y **Executor**. La DVNs es una red descentralizada responsable de verificar la correctitud y finalización de los mensajes. Luego de que la cantidad de configurada de validadores adjunte su correspondiente prueba criptográfica, los Executor son los responsables de entregar el mensaje a la

blockchain destino. Esto lo hacen interactuando con el **LayerZero Endpoint** en la blockchain destino. (*Layer Zero Docs - Protocol Overview*, s.f.)

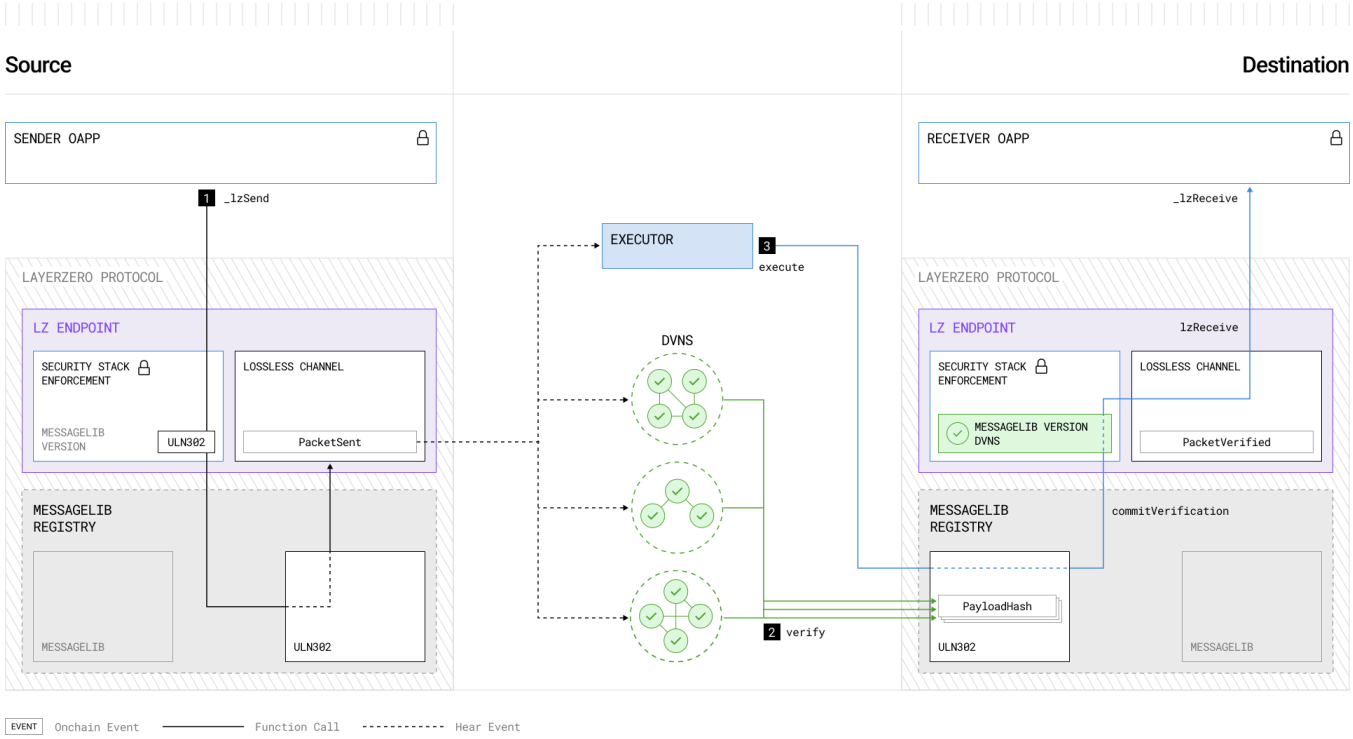


Figura 4.3: Arquitectura de LayerZero

4.1.4. Conclusión

Como se puede ver, muchas de las soluciones existentes implementan la transferencia de *assets* entre blockchains de maneras similares, contando con *smart contracts* en las blockchain origen y destino, encargados del envío y recepción de los *assets*, así como una entidad off-chain, encargada de escuchar los eventos emitidos por estos *smart contracts*, y de invocarlos en la blockchain destino.

Respecto a las soluciones analizadas, de Wormhole se destaca la velocidad de las transacciones. Esto se debe a tener una arquitectura simple, que cuenta únicamente con la red de guardianes para validar las transacciones. En contraparte, Chainlink agrega más complejidad y múltiples capas de seguridad como por ejemplo, la Risk Management Network, siendo la opción con mayor nivel de seguridad. Por último se destaca la personalización que ofrece LayerZero, ya que permite configurar sus propios parámetros de seguridad y de costos de las transacciones. La tabla 4.1 ofrece un resumen de los patrones que cada solución soporta, así como de sus ventajas y desventajas.

En conclusión, se optó por realizar una solución similar a estas arquitecturas, con mayor enfoque en la solución de Wormhole por su simplicidad, debido a que algunas de las consideraciones de seguridad que tienen las otras soluciones (como Chainlink o LayerZero) escapan el alcance de este proyecto.

	Permanent	Temporal	Finality	Ventajas	Desventajas
Wormhole	No	Si	Si	Simpleza y velocidad de transacciones	Ha tenido incidentes de seguridad [1]
Chainlink	Si	Si	Si	Nivel de seguridad alto	Costos más elevados
LayerZero	Si	Si	Si	Seguridad y costos personalizables	Mayor complejidad

Tabla 4.1: Conclusión análisis de soluciones

4.2. Método de Pago

Cuando se desarrolla una solución de interoperabilidad entre blockchains, es importante definir como se realizará el pago por las transacciones. Supongamos que un usuario quiere transferir un *asset* desde una blockchain origen a una destino utilizando un bridge. La realización del pago por la transacción en la blockchain origen es evidente, el usuario debe ser el que pague por la misma. Pero, a la hora de realizar la transacción en la blockchain destino, la cual es ejecutada por el bridge, ¿cómo se puede realizar la misma, sin que el bridge pierda dinero en el proceso?. A continuación, exploraremos cómo algunas de las soluciones existentes abordan este tema, específicamente Celer Bridge, Chainlink y Allbridge, seleccionadas por la claridad de su documentación en el ámbito de los pagos.

4.2.1. Celer Bridge

Celer Bridge es una solución de interoperabilidad entre blockchains diseñada para permitir transferencias eficientes y seguras de *assets* y datos entre múltiples blockchains, similar a las descritas anteriormente. En esta solución, el usuario que está iniciando la transacción cross-blockchain es el encargado de cubrir todos los costos de gas necesarios (tanto en la blockchain origen como en la destino).

Cuando un usuario realiza una transacción, paga la totalidad de los costos en la moneda nativa de la blockchain de origen, lo cual incluye un costo base (para cubrir el costo del *mint* de una NFT en la blockchain destino) y el costo del pasaje del mensaje por el *relayer* de Celer Bridge (*Celer cBridge Docs - NFT Bridge - NFT Bridge Fee*, s.f.). Una vez que el usuario paga este costo, Celer Bridge utiliza este dinero para realizar el pago del gas en la blockchain destino.

[1] https://www.certik.com/resources/blog/1kDYgyBcisoD2EqiBpHE5l-wormhole-bridge-exploit-incident-analysis

Para saber cuanto tiene que pagar en total el usuario, Celer Bridge calcula la tarifa del gas de la blockchain destino utilizando una tarifa base fija más una tarifa de protocolo.

4.2.2. Chainlink

De manera similar a la solución Celer Bridge, cuando un usuario desea realizar una transacción a otra blockchain en Chainlink, este debe pagar por la totalidad de los costos en la blockchain de origen.

Chainlink utiliza un modelo al cual llaman "Smart Execution", el cual apunta a remediar posibles problemas que pueden surgir en las transacciones entre blockchains, como por ejemplo la fluctuación de los precios de gas en la blockchain destino. (*Chainlink Doc - CCIP Billing*, s.f.)

Smart Execution fija el costo del gas en la blockchain origen basado en las condiciones del momento. Cuando se envía una transacción entre blockchains a través de Chainlink, se paga una única tarifa al principio de la transacción en la blockchain origen, y Chainlink garantiza la ejecución en la blockchain destino, incluso aunque los precios de gas hayan aumentado al momento en el que se ejecute la transacción.

Chainlink logra esto manteniendo un registro interno (cache) de las ultimas tarifas de gas de varias blockchains destino (similar a Celer Bridge). Cuando se inicia un mensaje o transacción entre blockchains, Chainlink revisa la tarifa de gas que tiene guardada para la blockchain destino y calcula un estimativo de cuánto gas se va a necesitar para ejecutar la transacción. Esta información es actualizada periódicamente para mantener las estimaciones precisas.

Además, a la tarifa de gas que se tiene guardada para la blockchain destino se la multiplica por un Gas Multiplier, el cual actúa como un margen de seguridad, previniendo ante incrementos inesperados en las tarifas del gas. Por ejemplo, cuando la tarifa de gas estimada es demasiado baja a la actual debido a una congestión repentina en la red, el multiplicador se asegura de que suficiente gas haya sido proporcionado para la ejecución de la transacción, evitando que falle la transacción debido a insuficiencia de gas. En caso de que sobre gas, no se realiza una devolución al usuario, por lo que es importante que las estimaciones sean precisas. Por lo tanto, el usuario solamente debe pagar una única tarifa al principio de la transacción en la blockchain origen (pudiendo ser el mismo tanto en la moneda nativa de la blockchain origen, como en LINK (moneda nativa de Chainlink)), y Chainlink se encarga de la ejecución de la transacción en la blockchain destino (realizando una conversión a una moneda nativa de la blockchain destino en caso de que sea necesario).

4.2.3. Allbridge

Allbridge es una plataforma de interoperabilidad de blockchain que permite la transferencia de tokens entre diferentes redes, similar a los bridges analizados en las secciones previas. Para el sistema de pago de gas de esta solución, se tienen algunas consideraciones.

Una Tarifa Puente se aplica a cada transferencia entre blockchains y se cobra únicamente en la transacción de "Enviar" realizada en la blockchain de origen. Esta tarifa se paga utilizando el mismo token que se está transfiriendo. El monto de la Tarifa Puente varía dependiendo de la blockchain utilizada(*Allbridge Docs - Bridge Fee*, s.f.). Para la mayoría de las blockchains, la tarifa es un porcentaje de la cantidad total siendo transferida, a excepción de, por ejemplo, Ethereum, la cual tiene una tarifa fija. No hay una Tarifa Puente para la transacción Recibir en la blockchain destino, pero los usuarios deben pagar las tarifas de gas separadas en ambas blockchains. Sin embargo, Allbridge cuenta con una funcionalidad en la cual los usuarios pueden delegar el pago de la tarifa de gas de la transacción Recibir en la blockchain destino, a cambio de una tarifa adicional al usuario en la blockchain origen. Existe más de un tipo de tarifa, basadas en monto fijo, en el valor del monto enviado y recibido, liquidez en las redes origen y destino. En el anexo B.5 se puede ver más en detalle como se implementa este sistema.

4.2.4. Conclusión

En los métodos de pago de los bridges analizados, la solución más común es que el usuario solamente debe pagar una única tarifa al principio de la transacción en la blockchain origen. El pago del gas en la blockchain destino lo realiza el propio bridge, cobrándoselo por adelantado al usuario mediante una estimación del costo y sumándole un margen de seguridad por la posible fluctuación de precio. Esta estimación puede ser dinámica, como es en el caso de Celer Bridge o Chainlink, o más sencilla, cobrando una tarifa fija como es el caso de Allbridge.

Capítulo 5

Diseño de la solución

Este capítulo presenta el diseño propuesto para implementar una solución de interoperabilidad entre Ethereum, HyperLedger Fabric y Polygon.

En primer lugar, en la sección 5.1 se describe el diseño de los patrones de interoperabilidad, en la sección 5.2 se presenta la arquitectura del escenario mediante las vistas 4+1 y por último, en la sección 5.3 se detallas las principales decisiones de diseño tomadas en el transcurso del proyecto.

5.1. Diseño de los patrones

En esta sección se describirá el diseño de los patrones de interoperabilidad **Relayer**, **Permanent Transfer** y **Temporal Transfer**.

5.1.1. Relayer

Para este patrón se diseñó un *Relayer* centralizado como un subsistema compuesto por un *Router*, así como los distintos conectores encargados de interactuar con las blockchains, habiendo así un conector dedicado a cada blockchain. Cada conector se encarga de escuchar eventos de transacciones emitidos por los *smart contracts* de sus respectivas blockchains, almacenarlos temporalmente en una base de datos propia del conector mientras se espera a su finalización, y finalmente enviarlos al Router una vez que el conector haya verificado que las transacciones se encuentren finalizadas.

El Router actúa como el componente encargado del enrutamiento. Recibe los eventos de los conectores, identifica la blockchain de destino y direcciona la información al conector correspondiente. En la Figura 5.1 se puede ver un diagrama de este subsistema para el escenario propuesto, teniéndose un conector definido para Fabric, así como un EVM Connector para cada una de las blockchains EVM (es decir, se tendrá un EVM Connector para Ethereum, y otro para Polygon). Cada EVM Connector cuenta con una base de datos, en la cual guardará las transacciones hasta que estas queden finalizadas. El conector

de Fabric no cuenta con lo mismo, ya que en Fabric todas las transacciones son automáticamente transacciones finalizadas. Estos conectores se comunican a través del Router. En la Figura 5.2 se puede ver un ejemplo de como sería el flujo para la transferencia de una transacción entre blockchains.

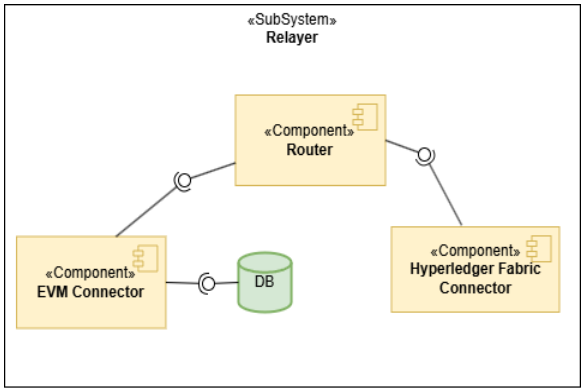


Figura 5.1: Relayer

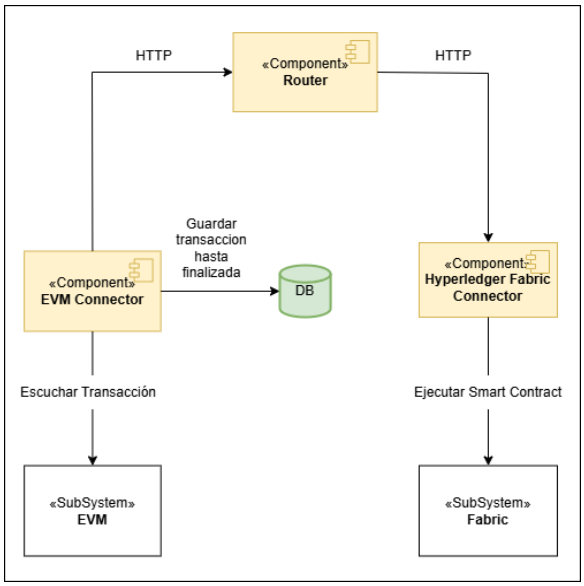


Figura 5.2: Relayer

5.1.2. Permanent Transfer

Para el patrón de transferencia permanente, se diseñó un *Permanent Transfer Smart Contract* que se encarga tanto del envío como de la recepción del

asset que se desea transferir. Se debe contar con una instancia de este contrato tanto en la blockchain origen como en la destino. Cuando se desea realizar una transferencia de un *asset* entre blockchains, el mismo debe ser transferido al *Permanent Transfer Smart Contract* en la blockchain origen. Este contrato se encargará de realizar un *burn* del *asset*, luego de lo cual emitirá un evento que contiene información de la transacción (*asset*, blockchain destino, usuario destino). El *Relayer* mencionado en la subsección anterior escuchará este evento, y lo enviará a la blockchain destino correspondiente, llamando al *Permanent Transfer Smart Contract* de dicha blockchain. Este contrato realizará el *mint* del *asset*, asignándoselo al usuario destino. En la Figura 5.3 podemos ver un ejemplo, con el *Permanent Transfer Smart Contract* instanciado en cada una de las blockchains a interoperar, así como el uso del *Relayer* para el pasaje entre blockchains.

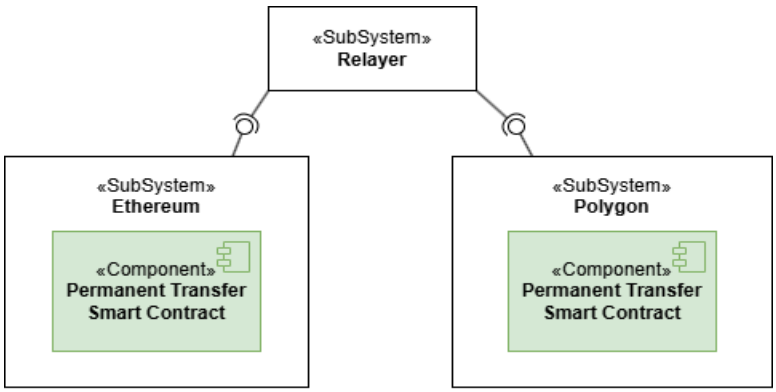


Figura 5.3: Permanent Transfer

5.1.3. Temporal Transfer

Para la transferencia temporal se tiene un diseño similar a la transferencia permanente. Se define un *Temporal Transfer Smart Contract*, el cual se debe encontrar tanto en la blockchain origen como destino. Cuando se desea realizar una transferencia temporal de un *asset* a otra blockchain, se transfiere el mismo al *Temporal Transfer Smart Contract* en la blockchain origen. Esto actúa como un *lock* del *asset*, ya que este deja de ser propiedad del usuario, y pasa a ser propiedad del *smart contract*. Luego, se emite el evento que contiene la información de la transacción. El *Relayer* escucha el evento, y lo envía a la blockchain destino, invocando al *Temporal Transfer Smart Contract* de la misma. Este contrato realizará un *mint* de un *asset* que representa al *asset* prestado de la blockchain origen (*Wrapped Token*), asignándoselo al usuario destino. Cuando se quiere realizar la devolución de este *asset*, el proceso es similar al préstamo del mismo. Se transfiere el *asset* prestado al *Temporal Transfer Smart Contract*. El contrato realiza un *burn* del mismo, y emite un evento que sera escuchado por el *Relayer*. Una vez enrutado el mensaje, el *Temporal Transfer Smart Contract* en

la blockchain original transfiere el *asset* al usuario que había realizado el préstamo. Esto actúa como un *unlock* del *asset*, ya que este vuelve a ser propiedad del usuario. En la Figura 5.4 se tiene un ejemplo muy similar al del Permanent Transfer, con el *Temporal Transfer Smart Contract* instanciado en cada una de las blockchains a interoperar, así como el uso del *Relayer* para el pasaje entre blockchains.

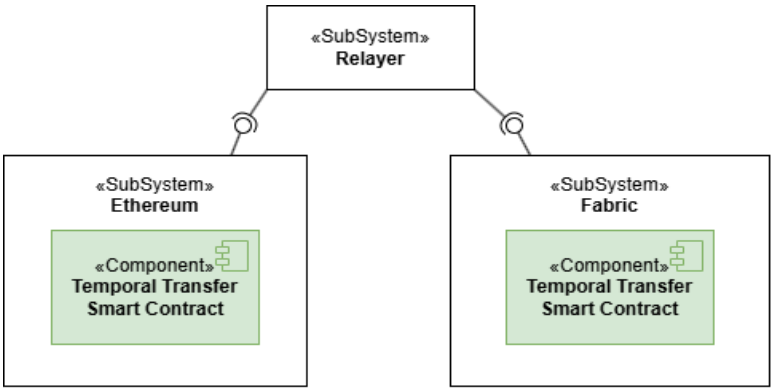


Figura 5.4: Temporal Transfer

5.2. Arquitectura del escenario

A continuación, se detalla la arquitectura del escenario mediante la representación de las vistas 4 + 1.

5.2.1. Vista de implementación

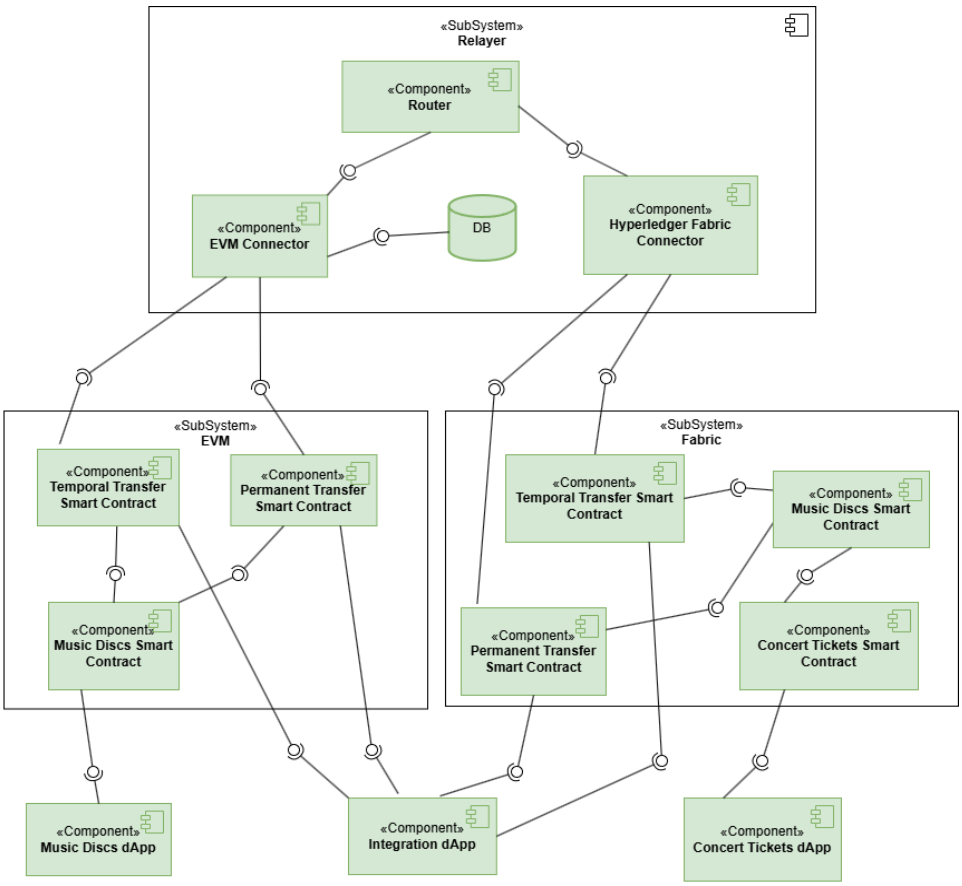


Figura 5.5: Diagrama de componentes

Descripción de Componentes

- **Music Discs Smart Contract**:
  Contiene la lógica de negocio de los discos musicales, siendo el responsable de la gestión de los mismos en las blockchains de *Ethereum*, *Polygon* y *Fabric*. Maneja la lógica de *minting*, transferencia (dentro de la misma blockchain) y *burning* de los *NFT* (discos musicales).

- **Concert Tickets Smart Contract**:
  Específico de la blockchain de Fabric, contiene la lógica de negocio de las entradas de conciertos, siendo el responsable de la gestión de los mismos. Maneja la logica de *minting*, transferencia (dentro de la misma blockchain) y *burning* de los *NFT* (entradas). Es capaz de realizar *mint* de mas de una (entrada) en el caso que corresponda (promoción *2x1*).

- **Permanent Transfer Smart Contract**:
  El *smart contract* del patrón de transferencia permanente mencionado en la sección anterior. En el escenario propuesto, se comunica con el *Music Discs Smart Contract* para las operaciones de *mint*, *burn*, y transferencia dentro de la blockchain de los *NFTs* de los discos musicales.

- **Temporal Transfer Smart Contract**:
  El *smart contract* del patrón de transferencia temporal mencionado en la sección anterior. En el escenario propuesto, se comunica con el *Music Discs Smart Contract* para las operaciones de *mint*, *burn*, y transferencia dentro de la blockchain de los *NFTs* de los discos musicales.

- **Music Discs dApp**:
  Una aplicación descentralizada que permite a los usuarios interactuar con los discos musicales. Los usuarios pueden comprar, visualizar sus discos musicales, así como migrarlos a otras blockchains. La *dApp* se comunica con el contrato de *Music Discs Smart Contract* como con la *Integration dApp* según sea necesario.

- **Concert Tickets dApp**:
  Permite la compra y visualización de entradas de concierto. Está integrada con el *Concert Tickets Smart Contract* de Fabric, y se conecta con la *Integration dApp* para manejar descuentos *2x1*.

- **Integration dApp**:
  Es la interfaz central para realizar transferencias permanentes y temporales entre blockchains. Recibe usuarios desde las otras *dApps* y coordina las interacciones con los *smart contracts* de transferencia.

- **EVM Connector**:
  Un componente *Connector* del *Relayer* mencionado en la sección anterior, para las blockchains EVM. En el escenario propuesto, escucha los eventos emitidos por el *Temporal Transfer Smart Contract* y *Permanent Transfer Smart Contract*.

- **Hyperledger Fabric Connector**:
  Un componente *Connector* del *Relayer* mencionado en la sección anterior, para la blockchain de Fabric. En el escenario propuesto, escucha los eventos emitidos por el *Temporal Transfer Smart Contract*.

- **Router**:
  El componente del *Relayer* encargado del enrutamiento, mencionado en la sección anterior.

- **DB**:
  Base de datos asociada a los conectores, mencionada en la sección anterior. Para el escenario propuesto, los *EVM Connectors* (para Ethereum y Polygon) contaran con una, para almacenar las transacciones mientras se aguarda a que estén finalizadas.

- **Relayer**:
  El subsistema que agrupa al *Router*, los conectores (*EVM Connector*, *Hyperledger Fabric Connector*) y sus DBs, mencionado en la sección anterior.

- **Ethereum**:
  Representa la instancia de la blockchain de *Ethereum* en el escenario. Dentro de este subsistema, se encuentra el *Music Discs Smart Contract* y los *Temporal Transfer Smart Contract* y *Permanent Transfer Smart Contract*.

- **Polygon**:
  Representa la instancia de la blockchain de *Polygon* en el escenario. Dentro de este subsistema, se encuentra el *Music Discs Smart Contract* y los *Temporal Transfer Smart Contract* y *Permanent Transfer Smart Contract*.

- **Fabric**:
  Representa la instancia de la blockchain de *Hyperledger Fabric* en el escenario. Dentro de este subsistema, se encuentra el *Concert Tickets Smart Contract*, *Music Discs Smart Contract* y los *Temporal Transfer Smart Contract* y *Permanent Transfer Smart Contract*.

5.2.2. Vista de Casos de Uso

En la Figura 5.6 se puede ver la interacción del usuario con los casos de uso mas relevantes para la arquitectura del escenario. La descripción de estos casos de uso fue vista en la sección 3.6

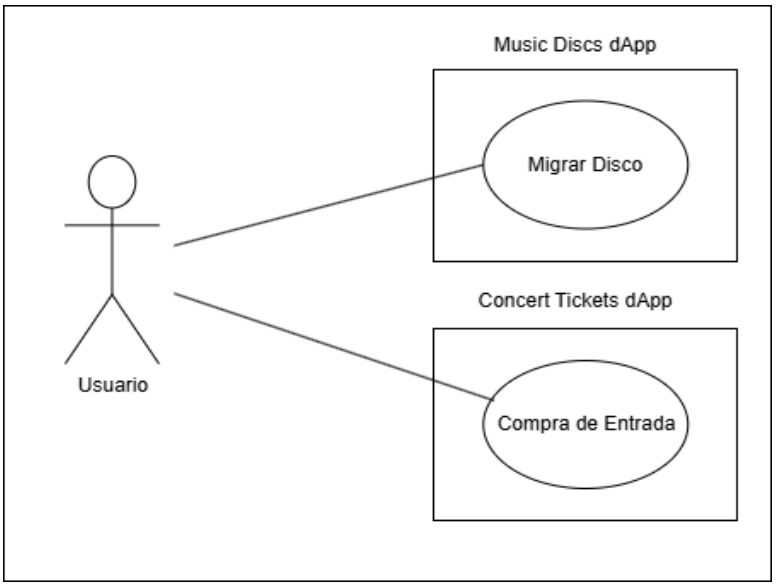


Figura 5.6: Casos de Uso

5.2.3. Vista lógica

En esta sección, se presentan los diagramas de secuencia de los casos de uso mas relevantes, definidos en la sección 3.6

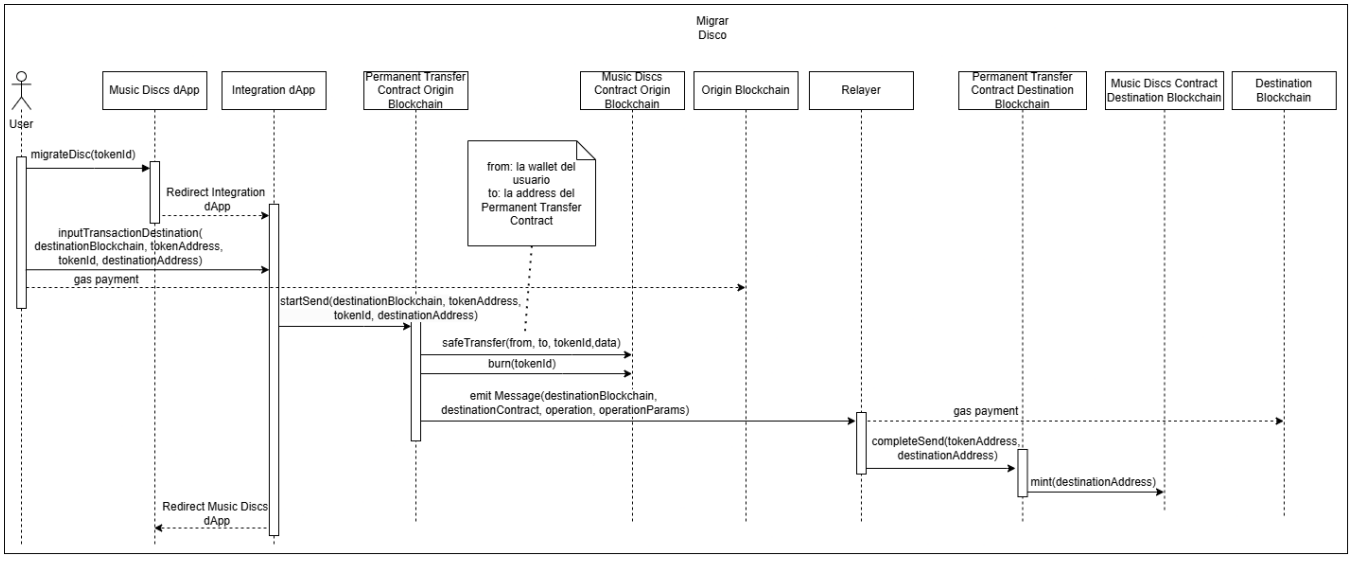


Figura 5.7: Migrar Disco

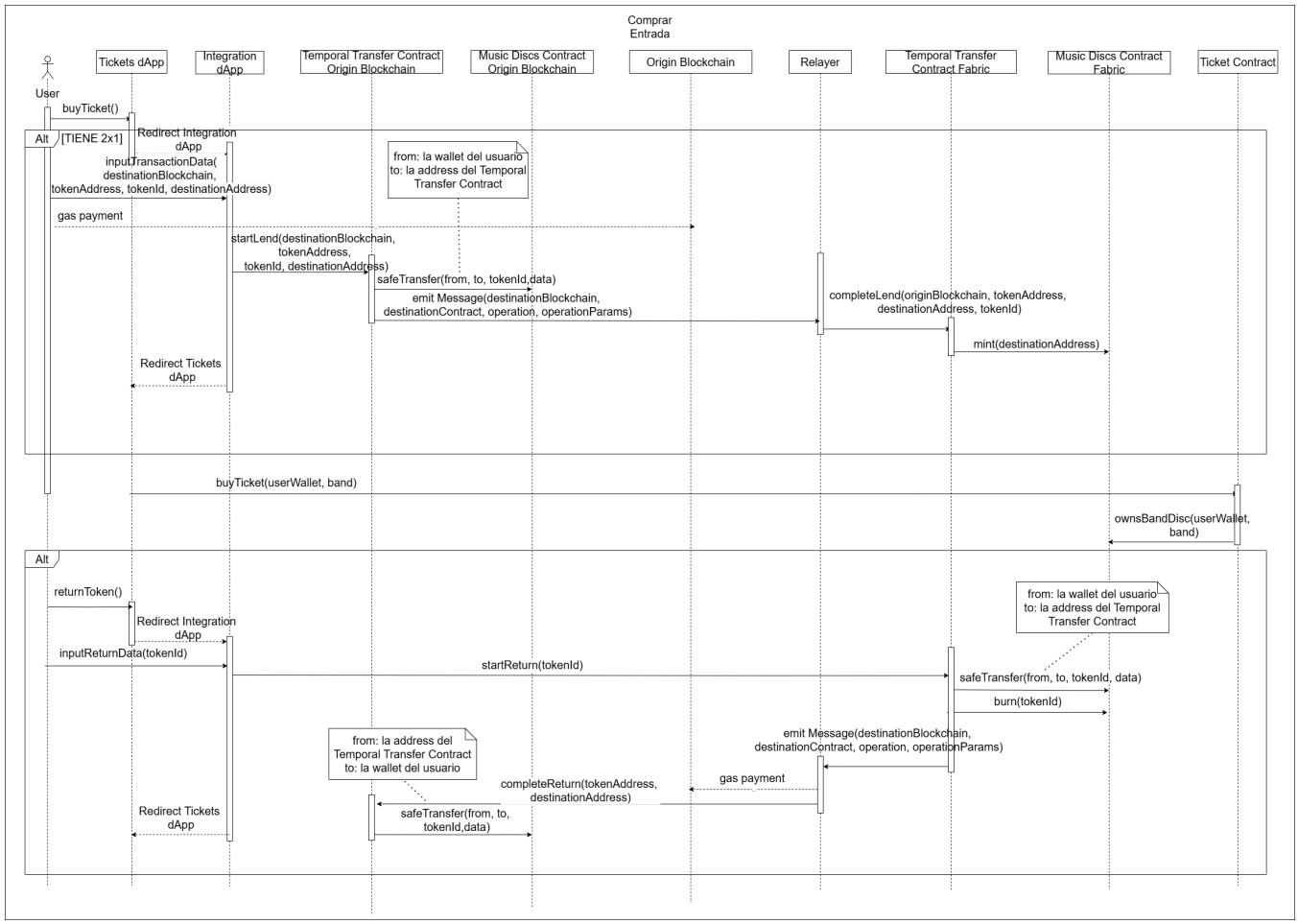


Figura 5.8: Comprar Entrada

Migración de disco (Figura 5.7)

El usuario inicia el proceso desde la *Music Discs dApp*, seleccionando el disco que desea transferir y llamando a *migrateDisc(tokenId)*. La aplicación redirige al usuario a la *Integration dApp*, donde ingresa los detalles de la transacción, como la blockchain de destino, la dirección del token, el identificador del token y la dirección de destino. A continuación, la *Integration dApp* interactúa con el *Permanent Transfer Smart Contract* invocando el método *startSend*, que transfiere el *NFT* desde la billetera del usuario al contrato y ejecuta un *burn* de la token en la blockchain de origen. El contrato emite un evento *Message*, que es procesado por el *Relayer*, encargándose de realizar el *minting* del nuevo *NFT* en la blockchain de destino para asignarlo a la cuenta correspondiente. Al finalizar este flujo, el disco queda disponible de forma permanente en la blockchain de

destino.

Compra de entradas (Figura 5.8)

El usuario inicia la compra de una entrada llamando al método *buyTicket*. Si desea acceder a la promoción de *2x1*, es redirigido a la *Integration dApp*, donde debe seleccionar que disco y desde que blockchain desea transferir temporalmente. La Integration dApp llama al método *startLend* en el *Temporal Transfer Smart Contract*, transfiriendo el *NFT* desde la billetera del usuario al contrato. Se emite un evento *Message*, que es procesado por el *Relayer* para completar el *minting* de un *NFT* equivalente en la blockchain de destino, llamando a la función *completeLend*. Una vez finalizado este proceso, el usuario vuelve a la *Concert Tickets dApp* para completar la compra de su entrada con la promoción activa. Posteriormente, se vuelve a redirigir al usuario a la *Integration dApp* para que realice la devolución del *NFT* a su blockchain de origen. La *dApp* llama al metodo *startReturn* del *Temporal Transfer Smart Contract*, transfiriendo el *NFT* prestado al contrato. El contrato realiza un *burn* del mismo, y emite un evento *Message*, que es procesado por el *Relayer*. Este llama al método *completeReturn* del *Temporal Transfer Smart Contract* de la blockchain original, devolviéndose así el *NFT* a su dueño original.

5.2.4. Vista de distribución

En la Figura 5.9 se puede observar cómo se despliegan los diferentes componentes de la solución. Los componentes del Relayer pueden ser desplegados de manera independiente, por ejemplo, cada uno en su propio servidor Linux. Los conectores deben tener visibilidad sobre el router y viceversa. Además, los conectores de EVM deben tener accesible un *filesystem* para el uso de la base de datos y acceso a internet para interactuar con la blockchain pertinente. Por otro lado, se debe tener desplegada la red de Fabric en otro servidor utilizando docker. El mismo debe ser accesible por internet para que la dApp de entradas y la dApp de integración la consuman. Los componentes correspondendientes a las redes de Ethereum y Polygon deben tener los *smart contracts* correspondientes desplegados y las dApps deben estar desplegadas en servidores o CDNs accesibles por internet.

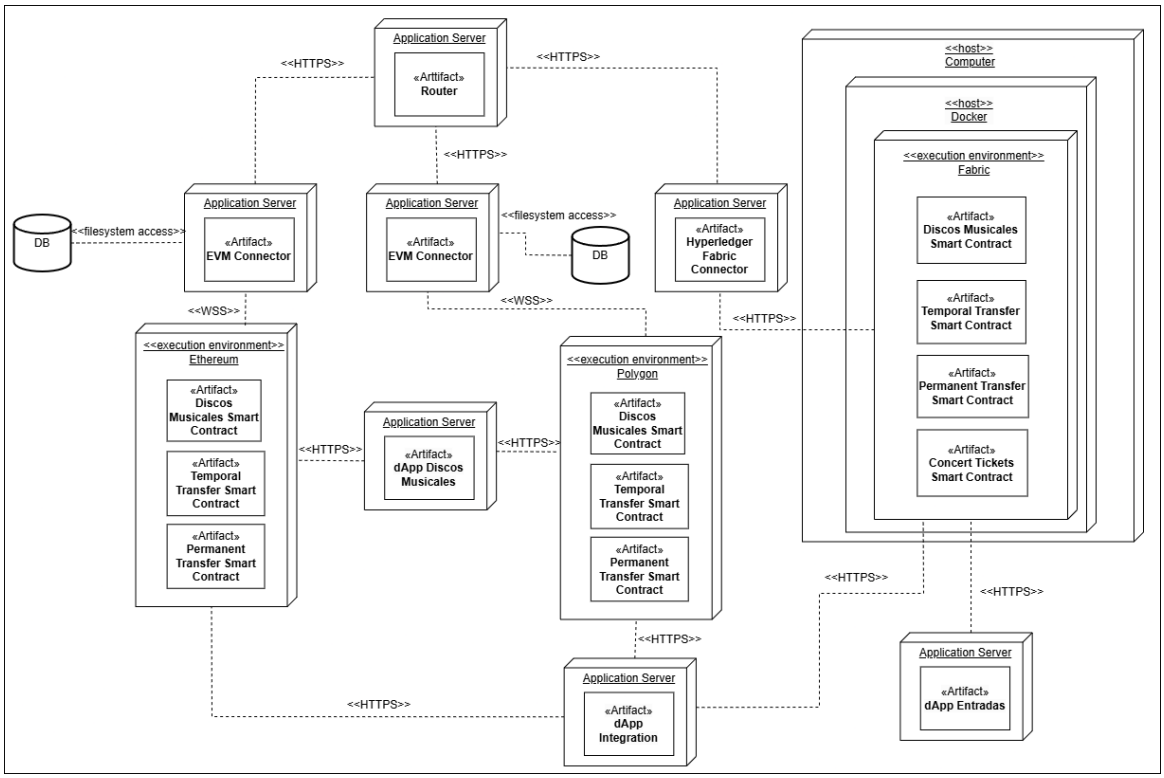


Figura 5.9: Vista de Distribución

5.3. Principales decisiones de diseño

En esta sección se exponen algunas de las decisiones que se tomaron a la hora de diseñar la solución.

5.3.1. Pago del gas

Analizando la manera en que las distintas soluciones de bridges existentes resuelven el tema del pago del gas por las transacciones en las blockchain origen y destino, se decidió descartar las soluciones mas complejas que implican un sistema de estimación dinámico de tarifas, como las de Celer Bridge o Chainlink. En cambio, se optó por una solución mas sencilla acorde al alcance del proyecto, como la de Allbridge, en la cual se tiene una tarifa fija extra que se le cobra al

usuario en la blockchain origen, para que luego el bridge se encargue del pago del gas en la blockchain destino. Entonces, en nuestra solución, se le cobrara al usuario una tarifa fija adicional de 0.005 (en la moneda nativa de la blockchain de origen).

5.3.2. Base de datos para cada conector EVM

Uno de los objetivos del proyecto es que se espere a la finalización de los bloques de transacciones antes de realizar las mismas en la blockchain destino. Para esto, se decidió que cada conector use una base de datos independiente. Cuando un conector escucha un evento emitido de una transacción, lo guarda en la base de datos. Cuando el bloque donde se encuentra la transacción está finalizado, el conector envía el mensaje al router, para luego ser recibido por el conector de la blockchain destino. Al mantener la base de datos de cada conector independiente, cada conector puede configurarla a medida. Como en Fabric los bloques siempre son finalizados, no es necesario el uso de una base de datos para el conector del mismo.

5.3.3. Arquitectura basada en Wormhole

La arquitectura propuesta toma elementos de la solución de Wormhole, como por ejemplo, la conceptualización del bridge como un *smart contract* en la blockchain origen encargado de inicializar la transferencia, emitiendo un evento que es escuchado por un relayer, y un *smart contract* equivalente en la blockchain destino encargado de recibir la transferencia. A esto se le agregó también la posibilidad de realizar transferencias permanentes, así como la posibilidad de interoperar con una blockchain permisionada (Fabric).

5.3.4. Emisión de eventos cómo mecanismo de comunicación

En base a la investigación realizada en el estado del arte, se decidió utilizar eventos cómo mecanismo de comunicación. Los eventos, tanto en las EVM cómo en Fabric son maneras de emitir un mensaje hacia el "mundo exterior" para comunicar algo que ocurrió dentro de la blockchain. En el caso de los bridges, los eventos son utilizados para comunicar que se comenzó a migrar un activo, donde luego este es escuchado y se continua con la migración hacia la otra blockchain.

Cuando se esta realizando un envío permanente, un envío temporal, o la devolución de un envío temporal, nuestros bridges emiten un mismo evento *Message*. Para mas detalle de este evento, ver Anexo **??**.

Capítulo 6

Implementación de la solución

En este capítulo se analiza y detalla los aspectos más relevantes de la implementación de los patrones y su aplicación en un escenario de uso. En la sección 6.1, se mencionan las tecnologías utilizadas en la implementación, En la sección 6.2 se muestra la implementación del escenario propuesto. Por ultimo, en la sección 6.3 se muestran algunas decisiones tomadas durante la implementación del escenario. El análisis de la implementación de los patrones *relayer*, transferencia permanente y transferencia temporal se encuentra disponible en el Anexo D debido a su extensión.

6.1. Tecnologías

Para la implementación de los patrones y del escenario, fue necesario utilizar varias tecnologías y librerías. En la Figura 6.1 se puede observar cada componente junto con la tecnología que se utilizó para el desarrollo del mismo. La estructura del repositorio del proyecto consiste de un monorepo configurado utilizando turbo (Turbo, s.f.). Las tres *dApps* fueron desarrolladas utilizando NextJS (Next.js, s.f.), Bun (Bun, s.f.) y ethers (Ethers.js, s.f.). El relayer fue desarrollado utilizando Typescript (TypeScript, s.f.) con Bun, y SQLite (SQLite, s.f.) para las bases de datos de los conectores. Para el desarrollo de los los patrones de envío temporal y permanente, en el caso de Ethereum y Polygon se utilizó Solidity y librerías de OpenZeppelin (OpenZeppelin, s.f.) (para el contrato `ERC721` de los NFTs), junto a Hardhat para realizar el despliegue de los mismos. En el caso de Hyperledger Fabric, se utilizó JavaScript y Docker para los contratos y su despliegue.

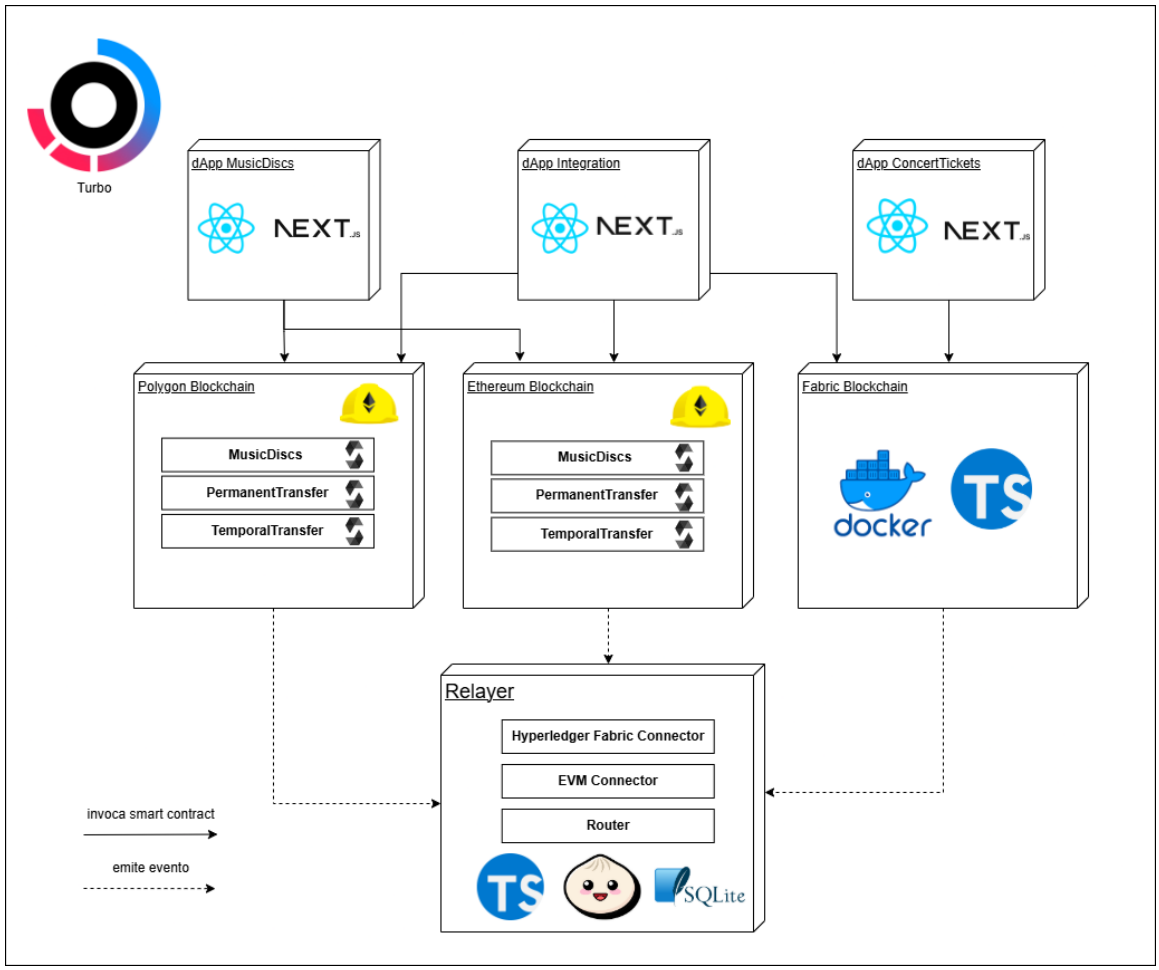


Figura 6.1: Diagrama de tecnologías utilizadas

6.2. Implementación del escenario propuesto

6.2.1. Music Discs Smart Contract

La implementación del `Music Discs Smart Contract` se basa en la especificación ERC-721 con algunas extensiones. El contrato hereda de múltiples contratos e interfaces, como `TokenImplementationInterface` (menciona-

do en la subsección D.1.2) y `Context` de OpenZeppelin. También implementa interfaces estándar como `IERC721Enumerable` para enumeración de NFTs, `IERC721Metadata` para manejo de nombre, símbolo y URI, `ERC165` para detección de interfaces soportadas, e `IERC721Errors` para definir errores estandarizados conforme a la especificación `ERC-721`. Además, hereda del `NFTState Smart Contract`, que gestiona el almacenamiento del estado de los NFTs, como sus propietarios, URIs y metadatos individuales. Este se puede ver en la Figura 6.2

```
contract NFTState {
    struct Bridges {
        address permanent;
        address temporal;
    }

    struct State {
        Bridges bridges;

        string name;
        string symbol;

        address owner;

        mapping(uint256 => address) owners;
        mapping(uint256 => address) tokenApprovals;
        mapping(address => uint256) balances;
        mapping(address => mapping(address => bool))
            operatorApprovals;
    }

    State internal _state;
}
```

Figura 6.2: NFTState (EVM)

En las Figuras 6.3, 6.4 y 6.5 se ven las funciones principales de `mint`, `burn` y `safeTransferFrom` (con sus funciones auxiliares).

```
function mint(address to, uint256 tokenId, string memory
    uri) public onlyOwner returns (uint256) {
    uint256 nextTokenId = _nextTokenId++;
    _mint(to, nextTokenId, uri);
    return nextTokenId;
}

function _mint(address to, uint256 tokenId, string memory
    uri) internal {
    require(to != address(0), "ERC721: mint to the zero
        address");
    require(!_exists(tokenId), "ERC721: token already
        minted");

    if (to == address(0)) {
        revert ERC721InvalidReceiver(address(0));
    }

    address previousOwner = _update(to, tokenId,
        address(0));

    if (previousOwner != address(0)) {
        revert ERC721InvalidSender(address(0));
    }

    _tokenURIs[tokenId] = uri;
}
```

Figura 6.3: Función `mint` de `MusicDiscs` (EVM)

```
function burn(uint256 tokenId) public onlyOwner {
    _burn(tokenId);
}

function _burn(uint256 tokenId) internal {
    address owner_ = MusicDiscs.ownerOf(tokenId);
    _approve(address(0), tokenId);
    address previousOwner = _update(address(0), tokenId,
        address(0));
    if (previousOwner == address(0)) {
        revert ERC721NonexistentToken(tokenId);
    }
    delete _state.owners[tokenId];
    emit Transfer(owner_, address(0), tokenId);
}
```

Figura 6.4: Función `burn` de `MusicDiscs` (EVM)

```
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId,
    bytes memory _data
) public override (IERC721, TokenImplementationInterface) {
    _safeTransfer(from, to, tokenId, _data);
}

function _safeTransfer(
    address from,
    address to,
    uint256 tokenId,
    bytes memory _data
) internal {
    _transfer(from, to, tokenId);
    require(_checkOnERC721Received(from, to, tokenId,
        _data), "ERC721: transfer to non ERC721Receiver
        implementer");
}

function _transfer(
    address from,
    address to,
    uint256 tokenId
) internal {
    require(MusicDiscs.ownerOf(tokenId) == from, "ERC721:
        transfer of token that is not own");
    require(to != address(0), "ERC721: transfer to the zero
        address");

    if (to == address(0)) {
        revert ERC721InvalidReceiver(address(0));
    }
    address previousOwner = _update(to, tokenId,
        address(0));
    if (previousOwner == address(0)) {
        revert ERC721NonexistentToken(tokenId);
    } else if (previousOwner != from) {
        revert ERC721IncorrectOwner(from, tokenId,
            previousOwner);
    }

    _approve(address(0), tokenId);

    emit Transfer(from, to, tokenId);
}
```

Figura 6.5: Función `safeTransferFrom` de `MusicDiscs` (EVM)

En las Figuras 6.6, 6.7 y 6.8 se tienen las mismas funciones para Fabric:

```
async Mint(
 ctx: Context,
 to: string,
 _tokenId: string,
 tokenUri: string
): Promise<string> {
    await this.CheckInitialized(ctx);
    const clientMSPID = ctx.clientIdentity.getMSPID();
    if (clientMSPID !== "Org1MSP") throw new
        Error("Unauthorized to mint");

    const counterBytes = await
        ctx.stub.getState(tokenIdCounterKey);
    let nextTokenId = 1;
    if (counterBytes && counterBytes.length > 0) {
        nextTokenId = parseInt(counterBytes.toString()) + 1;
    }
    await ctx.stub.putState(
        tokenIdCounterKey,
        Buffer.from(nextTokenId.toString())
    );

    const tokenId = nextTokenId.toString();
    const nft: NFT = { tokenId, owner: to, tokenUri,
        burned: false };
    await this._updateNFT(ctx, tokenId, nft);

    await this._updateBalance(ctx, to, 1);
    await this._addTokenToOwner(ctx, to, tokenId);

    const transferEvent = { from: "0x0", to, tokenId:
        parseInt(tokenId) };
    ctx.stub.setEvent("Transfer",
        Buffer.from(JSON.stringify(transferEvent)));

    return tokenId;
}
```

Figura 6.6: Función `Mint` de `MusicDiscs` (Fabric)

```
async Burn(ctx: Context, tokenId: string): Promise<boolean>
    {
    await this.CheckInitialized(ctx);
    const sender = this.getCurrentUserFromContext(ctx);
    const nft = await this._readNFT(ctx, tokenId);

    if (nft.owner !== sender) throw new Error("Not owner");
    if (nft.burned) throw new Error("Already burned");

    nft.burned = true;
    await this._updateNFT(ctx, tokenId, nft);

    await this._updateBalance(ctx, nft.owner, -1);
    await this._removeTokenFromOwner(ctx, nft.owner,
        tokenId);

    const transferEvent = {
      from: nft.owner,
      to: "0x0",
      tokenId: parseInt(tokenId),
    };
    ctx.stub.setEvent("Transfer",
        Buffer.from(JSON.stringify(transferEvent)));

    return true;
}
```

Figura 6.7: Función `Burn` de `MusicDiscs` (Fabric)

```
async TransferFrom(ctx: Context, from: string, to: string,
    tokenId: string) {
    await this.CheckInitialized(ctx);
    const sender = this.getCurrentUserFromContext(ctx);

    const nft = await this._readNFT(ctx, tokenId);
    if (nft.burned) {
      throw new Error("Cannot transfer burned token");
    }
    if (nft.owner !== from) {
      throw new Error("Transfer from wrong owner");
    }

    const isApproved =
      nft.approved === sender ||
      (await this.IsApprovedForAll(ctx, from, sender));
    if (from !== sender && !isApproved) {
      throw new Error("Not authorized to transfer");
    }

    nft.owner = to;
    nft.approved = "";
    await this._updateNFT(ctx, tokenId, nft);

    await this._updateBalance(ctx, from, -1);
    await this._updateBalance(ctx, to, 1);

    await this._removeTokenFromOwner(ctx, from, tokenId);
    await this._addTokenToOwner(ctx, to, tokenId);

    const transferEvent = { from, to, tokenId:
        parseInt(tokenId) };
    ctx.stub.setEvent("Transfer",
        Buffer.from(JSON.stringify(transferEvent)));

    return true;
}
```

Figura 6.8: Función `TransferFrom` de `MusicDiscs` (Fabric)

6.2.2. Concert Tickets Smart Contract

El `Concert Tickets Smart Contract` hereda de la clase `Contract` del paquete `fabric-contract-api`, lo que le permite integrarse con la infraestructura de Hyperledger y definir funciones transaccionales.

En las Figuras 6.9 y 6.10 se tienen las funciones `Mint` y `TransferFrom` del contrato. Para el escenario propuesto, no fue necesario que este cuente con una función de *burn*.

```
async Mint(ctx: Context) {
    const clientMSPID = ctx.clientIdentity.getMSPID();
    if (clientMSPID !== "Org1MSP") {
      throw new Error("client is not authorized to mint new tokens");
    }

    const tokenId = await this._getNextTokenId(ctx);
    if (isNaN(tokenId)) {
      throw new Error(
        `The tokenId ${tokenId} is invalid. tokenId must be an integer`
      );
    }

    const owner = this.getCurrentUserFromContext(ctx);

    const nft: NFT = {
      tokenId,
      owner,
    };

    const nftKey = ctx.stub.createCompositeKey(nftPrefix, [tokenId.toString()]);
    await ctx.stub.putState(nftKey, Buffer.from(JSON.stringify(nft)));

    const balanceKey = ctx.stub.createCompositeKey(balancePrefix, [
      owner,
      tokenId.toString(),
    ]);
    await ctx.stub.putState(balanceKey, Buffer.from("\u0000"));

    const transferEvent = { from: "0x0", to: owner, tokenId };
    ctx.stub.setEvent("Transfer", Buffer.from(JSON.stringify(transferEvent)));

    return nft;
}
```

Figura 6.9: `Mint`

```
async TransferFrom(ctx: Context, tokenId: string, to:
    string) {
    const from = this.getCurrentUserFromContext(ctx);
    const nft = await this._readNFT(ctx, tokenId);

    if (nft.owner !== from) {
      throw new Error(
        "The sender is not allowed to transfer the
            non-fungible token"
      );
    }

    nft.owner = to;
    const nftKey = ctx.stub.createCompositeKey(nftPrefix,
        [tokenId]);
    await ctx.stub.putState(nftKey,
        Buffer.from(JSON.stringify(nft)));

    const balanceKeyFrom =
        ctx.stub.createCompositeKey(balancePrefix, [
      from,
      tokenId,
    ]);
    await ctx.stub.deleteState(balanceKeyFrom);

    const balanceKeyTo =
        ctx.stub.createCompositeKey(balancePrefix, [
      to,
      tokenId,
    ]);
    await ctx.stub.putState(balanceKeyTo,
        Buffer.from("\u0000"));

    const tokenIdInt = parseInt(tokenId);
    const transferEvent = { from: from, to: to, tokenId:
        tokenIdInt };
    ctx.stub.setEvent("Transfer",
        Buffer.from(JSON.stringify(transferEvent)));

    return true;
}
```

Figura 6.10: `TransferFrom`

6.2.3. Music Discs dApp

Para ver los distintos discos que tiene el usuario, este primero debe conectar su billetera a la *dApp*. Para esto se integró la *dApp* con Metamask, estando el

manejo de la red y dirección de usuario a cargo del mismo. Una vez que la *dApp* recibe estos parámetros a través de la integración con Metamask, esta se encarga de obtener todos los discos que tiene el usuario, utilizando su dirección para buscarlos. Para esto último se utilizan dos de las funciones que los `ERC-721` ofrecen: `balanceOf` y `tokenOfOwnerByIndex`. `balanceOf` permite obtener el número de NFTs que tiene una dirección, `tokenOfOwnerByIndex` permite conseguir la información de un NFT a partir del índice de este. Para conseguir la información de todos los NFTs lo primero que se hace es conseguir el número de NFTs que tiene el usuario y luego llamar a la función `tokenOfOwnerByIndex` para cada uno de estos, así consiguiendo el id de cada token. Por último, los NFT del usuario que han sido transferidos de manera temporal no son mostrados, ya que al enviarlos de manera temporal se cambia el dueño de estos, habiendo sido transferidos al contrato del bridge.

6.2.4. Concert Tickets dApp

La *dApp* de Concert Tickets tiene una implementación análoga a la *dApp* de Music Discs, pero para los NFTs del `Concert Tickets Smart Contract`

6.2.5. Integration dApp

La Integration dApp sirve para ver NFTs de contratos que se han integrado con el Bridge. El usuario puede ver tanto los NFTs de los cuales es dueño, como también los que se le han prestado. Se utilizan los mismos mecanismos de las Music Discs y Concert Tickets dApp para mostrar los NFTs. Además de mostrar los NFTs, el usuario puede seleccionar un NFT y enviarlo de manera permanente o temporal, pudiendo también realizar la devolución de los que hayan sido prestados.

6.3. Decisiones de implementación

En este sección se verán algunas de las decisiones de implementación tomadas durante el desarrollo del escenario. Estas decisiones están relacionadas a algunas de las dificultades que se encontraron a la hora de desarrollar para Fabric, como la instanciación de NFTs en Fabric, así como la autenticación de los usuarios para llamar a los *smart contracts*.

6.3.1. Separación en 3 dApps

Se realizó una separación en 3 dApps, debido a que cada dapp tiene su propia responsabilidad, con su propia lógica de negocio. La dApp de los MusicDiscs tiene únicamente la lógica del manejo de los discos musicales, la dApp de Concert Tickets la de los tickets, y la dApp de integración resuelve el problema de la interoperabilidad entre ambas.

6.3.2. ERC721 en Hyperledger Fabric

Así como para las EVM se utilizo las librería de OpenZeppelin para crear contratos de NFTs, para la implementación de NFTs en Fabric, se utilizó un ejemplo de implementación de `ERC721` que ofrece la fundación de Hyperledger Fabric (*Fabric ERC-721 Token Scenario*, s.f.), agregándole además la funcionalidad para realizar *burn* de los NFTs. Para esto ultimo, simplemente se le agrega un campo booleano a cada NFT, que indica si el NFT fue quemado o no.

6.3.3. Autenticación en Hyperledger Fabric

Para poder utilizar distintos usuarios dentro de las dApps que soportan Hyperledger Fabric, se crearon y guardaron distintos usuarios con sus certificados. La custodia de estos usuarios es a través del servicio backend que se utiliza para interactuar con la blockchain. Cada uno de estos usuarios tiene su propio nombre de usuario y contraseña, con el cual se puede acceder y modificar los recursos del usuario. Una vez el usuario ingresa en su cuenta, se almacena una cookie en el navegador para mantener la sesión iniciada. El usuario puede cerrar sesión, en donde se destruye la cookie.

Capítulo 7

Evaluación de la solución

En este capítulo se analizarán los costos asociados a la solución implementada (sección 7.1).

7.1. Análisis de costos

En esta sección se analizará los costos asociados al despliegue y operación de la solución desarrollada (Relayer, Permanent y Temporal Transfer). A la hora de implementar la solución, se observa que se tienen los siguientes costos:

- Costo de infraestructura del Relayer
- Costo por despliegue de *Smart Contracts*
- Costo del gas en la ejecución de los *Smart Contracts*

Costos del Relayer

Los costos asociados a la infraestructura del Relayer son de al menos un router, un conector por cada blockchain y una base de datos por cada blockchain EVM. Para implementar el escenario propuesto entonces, serían necesario infraestructura como mínimo para el router y los 3 conectores. Si cada servidor fuera por ejemplo una instancia t3.micro de 2vcpu y 1 gb de memoria, el costo asociado a esta infraestructura en Amazon Web Services (AWS) sería de 7.59 x 4 = 30.37 USD por mes. Siendo la misma una estimación muy simplificada solo a modo de ejemplo, sin considerar todos los factores necesarios para un entorno de producción real (escalabilidad, alta disponibilidad, recuperación ante desastres, etc).

Costos del despliegue de los Smart Contract

Es un gasto que se genera una única vez, o a lo sumo, cuando se desee crear nuevas versiones. Los costos para realizar el despliegue de los contratos

de Permanent y Temporal Transfer en Ethereum y Polygon se pueden observar en la tabla 7.2. Cabe destacar que el gas consumido para Ethereum como para Polygon es el mismo, diferenciándose en los costos debido a la congestión de red y a la diferencia de valor entre ETH y POL, como puede verse en la tabla 7.1.

Blockchain	Moneda	Valor Moneda	Valor gas
Ethereum	ETH	1788.48 USD	0.4 gwei
Polygon	POL	0.1842 USD	40 gwei

Tabla 7.1: Valores al momento de realizar las pruebas

Smart Contract	Gas unit	Costo ETH/POL	Costo USD
PermanentTransfer ETH	1322443	0.0005289772 ETH	0.9445
TemporalTransfer ETH	2729958	0.0010919832 ETH	1.9519
PermanentTransfer POL	1322443	0.05289772 POL	0.0097
TemporalTransfer POL	2729958	0.10919832 POL	0.0201

Tabla 7.2: Costo despliegue contrato Permanent y Temporal Transfer

Costo del gas

Para calcular el gasto de gas se ejecutaron las funciones del Permanent y Temporal Transfer. En las tabla 7.3 y en la tabla 7.4 se pueden ver los costos de ejecutar cada función respectivamente. Al analizar las tablas, se puede ver que la solución propuesta es económicamente viable ya que por ejemplo, se obtuvieron los siguientes costos entre Polygon y Ethereum:

- Envío permanente de Ethereum a Polygon: 0.073829 USD
- Envío permanente de Polygon a Ethereum: 0.207538 USD
- Envío temporal de Ethereum a Polygon: 0.2232 USD
- Envío temporal de Polygon a Ethereum: 0.4979 USD

Smart Contract	Gas	Costo ETH/Pol	Costo USD
StartSend ETH	100233	0.0000400932 ETH	0.0717
CompleteSend ETH	289113	0.0001156452 ETH	0.2068
StartSend POL	100233	0.00400932 POL	0.000738
CompleteSend POL	289113	0.01156452 POL	0.002129

Tabla 7.3: Costos de ejecución de Permanent Transfer en Ethereum y Polygon

Smart Contract	Gas	Costo ETH/POL	Costo USD
StartLend ETH	209865	0.000083946 ETH	0.1500
CompleteLend ETH	547140	0.000218856 ETH	0.3915
StartReturn ETH	145616	0.000058246 ETH	0.1042
CompleteReturn ETH	95201	0.0000380804 ETH	0.0681
StartLend POL	209865	0.0083946 POL	0.0015
CompleteLend POL	547140	0.0218856 POL	0.0040
StartReturn POL	145616	0.00582464 POL	0.0011
CompleteReturn POL	95201	0.00380804 POL	0.0007

Tabla 7.4: Costos de ejecución de Temporal Transfer en Ethereum y Polygon

Capítulo 8

Gestión del proyecto

En el siguiente capítulo, se explica cómo se gestionó el proyecto. En la sección 8.1 se describe cómo se organizó el equipo. Luego, en la sección 8.2 se compara la planificación inicial con la ejecución real.

8.1. Organización del proyecto

Al inicio del proyecto, se realizó una planificación inicial, donde se determinaron las actividades a realizar y una estimación de la duración de las mismas. Se puede ver el cronograma en la tabla 8.1.

Se decidió dividir el trabajo en base a los patrones, iterando las etapas de análisis, diseño e implementación sobre cada patrón, de manera secuencial. Es decir, hasta no haber terminado con un patrón, no se avanzaría con el siguiente. Esto debido a la dependencia y similitud entre los patrones.

Se definió que se trabajaría mediante sprints de 2 semanas, y que luego de los mismos, se presentarían los avances al tutor. Al terminar cada una de las actividades, se realizaron entregables como documentación y diagramas, y aparte demos del proyecto, para validar y obtener retroalimentación del trabajo realizado hasta el momento.

Actividades	Febrero	Marzo	Abril	Mayo	Junio	Julio	Agosto	Setiembre	Octubre	Noviembre
Plan de trabajo	X	X								
Estudio temática	X	X								
Definición de requerimientos		X								
Definir caso de uso		X								
Análisis y diseño patrón Relayer			X							
Implementación patrón Relayer			X	X						
Análisis y diseño patrón Transferencia Permanente					X	X				
Implementación patrón Transferencia Permanente					X	X				
Análisis y diseño patrón Transferencia Temporal						X	X			
Implementación patrón Transferencia Temporal						X	X			
Implementación del escenario			X	X	X	X	X			
Evaluación de la solución							X	X		
Documentación								X	X	X

Tabla 8.1: Planificación inicial

8.2. Cronograma final

La tabla 8.2 muestra como terminó siendo el cronograma final.

Actividades	Feb	Mar	Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic	Ene	Feb	Mar	Abr
Plan de trabajo	X	X													
Estudio temática	X	X	X												
Definición de requerimientos		X													
Definir caso de uso		X													
Análisis y diseño patrón Relayer			X	X	X										
Implementación patrón Relayer			X	X	X										
Análisis y diseño patrón Transferencia Permanente					X	X	X								
Implementación patrón Transferencia Permanente					X	X	X								
Análisis y diseño patrón Transferencia Temporal						X	X	X	X						
Implementación patrón Transferencia Temporal						X	X	X	X						
Implementación del escenario									X	X	X		X	X	
Evaluación de la solución														X	
Documentación										X	X		X	X	X

Tabla 8.2: Planificación final

El proyecto duró cuatro meses más de lo esperado. Este se debe en parte a varios motivos. El primero fue que nos encontramos con la dificultad de los proyectos de Blockchain era un poco más elevada de lo que se esperó y el equipo no estaba del todo familiarizado con la tecnología. Por otro lado, la implementación del Relayer llevó más tiempo del estimado debido a que en un principio, se esperaba mayor reutilización de código por parte de proyectos de grados anteriores. Sin embargo, debido a dificultades a la hora de levantar los proyectos y a nuevas versiones en librerías, se decidió realizar una implementación de cero.

Otra dificultad que se encontró fue el desacoplar el patrón *Relayer* de los `Permanent Transfer Smart Contract` y `Temporal Transfer Smart Contract`. En un principio, con el objetivo de poder realizar una transferencia de un NFT de una blockchain a otra, se implementó el Relayer de cierta manera que resultaba imposible generalizar a los contratos escuchados por el Relayer. Mas adelante en el proyecto, se realizó una refactorización del código que permitió el desacoplamiento del Relayer con respecto a los contratos del bridge.

Una demora sustancial en el proyecto fue debido al descanso tomado por el equipo en los meses de Diciembre, Enero y Febrero. Se intentó llegar a la fecha planificada y se dedicó mucho esfuerzo al proyecto en los últimos meses del año, pero al ver que no iba a ser posible entregar en Diciembre, se decidió tomarse todos un descanso y retomar a fines de Febrero.

Finalmente, en el ultimo tramo del proyecto, se terminaron de desarrollar las distintas *dApps* del escenario. En un principio, se tenia una única *dApp* que manejaba todos los patrones, y contenía tanto los discos musicales como las entradas. Sobre el final del proyecto, se realizó la separación de las lógicas de negocio para cada una de las *dApps* del escenario.

Capítulo 9

Conclusiones y Trabajo Futuro

En este capítulo se presentan las conclusiones del trabajo realizado y se discute mejoras a futuro. La sección 9.1 presenta las conclusiones, mientras que en la sección 9.2 se discuten las posibles mejoras y extensiones que podrían implementarse en el trabajo a futuro.

9.1. Conclusiones

El proyecto logró cumplir con los objetivos planteados, implementando exitosamente los patrones de interoperabilidad "Relayer", "Transferencia Permanente" y "Transferencia Temporal", creando también un escenario de uso entre las blockchains Ethereum, Polygon e Hyperledger Fabric, donde se ofrece un ejemplo de aplicación de los mismos. Se contribuyó al campo de interoperabilidad en blockchain proveyendo herramientas para la implementación de los patrones, al haber realizado una implementación de referencia para estos. Se investigaron las soluciones existentes y se tomaron distintas ideas a partir de ellas para nuestra investigación. Se pudieron superar los distintos desafíos encontrados, cómo por ejemplo el de la curva de aprendizaje pronunciada para el ecosistema de Ethereum y Hyperledger Fabric. Se desarrollaron herramientas para la finalización de los bloques, así asegurando unicamente el procesamiento de transacciones que no vayan a ser revertidas. Se realizó una evaluación de costos de las implementaciones de los patrones, pudiéndose observar una viabilidad económica de los mismos.

Sobre la gestión del proyecto, se aprendió que no hay que subestimar el aprendizaje de una tecnología nueva, que la elección del orden en el desarrollo de los módulos de software es muy importante y que no hay que descuidar el balance en las horas de trabajo para evitar el agotamiento. Se aprendió que el cuidado del código es fundamental para un proyecto grande y grupal. Es importante ser prolijo al escribir y documentar, ya que el mismo código puede

ser abarcado por más de un integrante. Así como la dificultad de crear una solución sea genérica y no enfocada en un escenario en particular.

Finalmente, a lo largo de la realización de este proyecto, se pudieron observar algunas carencias en el área de interoperabilidad entre blockchains. Con la investigación realizada se observó la escasez de soluciones de interoperabilidad que permitan la comunicación con blockchains permisionadas. También se encontró una carencia de documentación clara de las distintas soluciones de interoperabilidad, lo que resulta en una alta curva de aprendizaje si se desea adentrarse en el desarrollo de estas. Debido a que el campo de blockchain, y de interoperabilidad en blockchain, sigue en constante evolución, se considera que hay muchas áreas de mejora. Por eso mismo, se destaca la importancia del acercamiento entre la industria y el mundo académico, para seguir desarrollando y mejorando las soluciones disponibles. A medida que la necesidad y el interés en la interoperabilidad sigan creciendo, este campo seguirá evolucionando y mejorando.

9.2. Trabajo futuro

Existen múltiples oportunidades para continuar el trabajo y enriquecer la solución propuesta. A continuación, se detallan algunas de estas líneas de trabajo futuro:

9.2.1. Implementar más patrones de interoperabilidad

Además de los patrones Relayer, Permanent Transfer y Temporal Transfer implementados, quedan pendientes Blockchain API Gateway, Light Client y Aggregator. Cada uno ofrece ventajas específicas: Gateway facilita la comunicación entre redes heterogéneas, Light Client permite interoperabilidad en nodos ligeros, y Aggregator consolida datos de múltiples fuentes. Su integración en futuros trabajos ampliaría las soluciones propuestas y permitiría más flexibilidad a la hora de interoperar.

9.2.2. Calculo de tarifa y gas

Debido a la simplificación que se realizó por el alcance de proyecto, se le cobra al usuario una tarifa fija en la moneda de la blockchain origen. Una mejora posible sería realizar la estimación del costo de manera dinámica, tomando en cuenta costos en las blockchain destino en tiempo real. Implementar mecanismos como los utilizados en Chainlink, AllBridge o CelerBridge.

9.2.3. Incorporar un mecanismo de reintento para operaciones fallidas

Actualmente, el sistema carece de un mecanismo para reintentar operaciones fallidas, lo que posibilita la pérdida de tokens dentro de la blockchain. Desarrollar

e integrar esta funcionalidad en futuras iteraciones permitiría una solución de mayor confianza.

Referencias

Allbridge docs - bridge fee. (s.f.). https://docs.allbridge.io/allbridge-overview/bridge-fee.

Allbridge repo - state. (s.f.). https://github.com/allbridge-io/allbridge-contract-docs/tree/master/state.

Anceaume, E., Del Pozzo, A., Rieutord, T., y Tucci-Piergiovanni, S. (2022). On Finality in Blockchains. En Q. Bramas, V. Gramoli, y A. Milani (Eds.), *25th international conference on principles of distributed systems (opodis 2021)* (Vol. 217, pp. 6:1–6:19). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. Descargado de https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2021.6 doi: 10.4230/LIPIcs.OPODIS.2021.6

Aračić, I. (2023). *Blockchain short story 011: Hashed time-lock contracts (htlcs).* https://medium.com/blockchain-short-stories/blockchain-short-story-011-hashed-time-lock-contracts-htlcs-5566e27ce89b.

Aws - what is blockchain technology? (s.f.). https://aws.amazon.com/what-is/blockchain.

Belchior, R., Vasconcelos, A., Guerreiro, S., y Correia, M. (2021). *A survey on blockchain interoperability: Past, present, and future trends.* https://dl.acm.org/doi/10.1145/3471140.

Blackcoin's proof-of-stake protocol v2. (s.f.). https://blackcoin.org/blackcoin-pos-protocol-v2-whitepaper.pdf.

Bun. (s.f.). Bun documentation [Manual de software informático]. Descargado de https://bun.sh/ (Recuperado el 30 de mayo de 2025)

Castro, M., y Liskov, B. (s.f.). *Practical byzantine fault tolerance.* https://pmg.csail.mit.edu/papers/osdi99.pdf.

Celer cbridge docs - nft bridge - introduction. (s.f.). https://cbridge-docs.celer.network/nft-bridge/introduction.

Celer cbridge docs - nft bridge - nft bridge fee. (s.f.). https://cbridge-docs.celer.network/nft-bridge/nft-bridge-fee.

Celer network repo - nftbridge.sol. (s.f.). https://github.com/celer-network/nft-bridge-contracts/blob/main/contracts/NFTBridge.sol.

Chainlink doc - ccip architecture. (s.f.). https://docs.chain.link/ccip/architecture.

Chainlink doc - ccip architecture security. (s.f.). https://blog.chain.link/five-levels-cross-chain-security.

Chainlink doc - ccip billing. (s.f.). https://docs.chain.link/ccip/billing.

Ethereum docs. (s.f.-a). https://ethereum.org/en/developers/docs/intro-to-ethereum/.

Ethereum docs. (s.f.-b). https://ethereum.org/es/developers/docs/consensus-mechanisms/pos/.

Ethereum docs - erc-20 token standard. (s.f.). https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

Ethereum docs - erc-721 non-fungible token standard. (s.f.). https://ethereum.org/en/developers/docs/standards/tokens/erc-721/.

Ethereum docs - gas. (s.f.). https://ethereum.org/en/developers/docs/gas/.

Ethers.js. (s.f.). Ethers.js documentation (v6) [Manual de software informático]. Descargado de https://docs.ethers.org/v6/ (Recuperado el 30 de mayo de 2025)

Fabric docs. (s.f.). https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html.

Fabric erc-721 token scenario. (s.f.). https://github.com/hyperledger/fabric-samples/blob/main/token-erc-721/README.md.

Graham, S. L., Lamport, L., Shostak, R., y Pease, M. (1982). *The byzantine generals problem.* https://dl.acm.org/doi/10.1145/357172.357176.

Hernandez Casco, F. (2024, dic.). Blockchain y tokenización: Oportunidades y retos para centroamérica. *Revista de Fomento Social*(310), 473-493. Descargado de https://revistas.uloyola.es/rfs/article/view/5526 doi: 10.32418/rfs.2024.310.5526

Horizen academy - what is a sidechain? (s.f.). https://www.horizen.io/academy/sidechains/.

King, S. (2013). Primecoin: Cryptocurrency with prime number proof-of-work. Descargado de http://primecoin.org/

King, S., y Nadal, S. (2012). Ppcoin: Peer-to-peer crypto-currency with proof-of-stake.. Descargado de https://api.semanticscholar.org/CorpusID:42319203

Kuznetsov, O., Rusnak, A., Yezhov, A., Kuznetsova, K., Kanonik, D., y Domin, O. (2024, julio). Merkle trees in blockchain: A study of collision probability and security implications. *Internet of Things*, *26*, 101193. Descargado de http://dx.doi.org/10.1016/j.iot.2024.101193 doi: 10.1016/j.iot.2024.101193

Layer zero docs - evm dvn and executor configuration. (s.f.). https://docs.layerzero.network/v2/developers/evm/configuration/dvn-executor-config.

Layer zero docs - message library. (s.f.). https://docs.layerzero.network/v2/home/protocol/message-library.

Layer zero docs - protocol overview. (s.f.). https://docs.layerzero.network/v2/home/protocol/protocol-overview.

Llambías, G., González, L., y Ruggia, R. (s.f.). *Blockchain interoperability: a feature-based classification framework and challenges ahead.*

Llambías, G., González, L., y Ruggia, R. (2022). *Blockchain interoperability patterns.*

Mohanta, B., Panda, S., y Jena, D. (2018, 10). An overview of smart contract and use cases in blockchain technology.. doi: 10.1109/ICCCNT.2018.8494045

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

Next.js. (s.f.). Next.js documentation [Manual de software informático]. Descargado de https://nextjs.org/ (Recuperado el 30 de mayo de 2025)

OpenZeppelin. (s.f.). Openzeppelin contracts 4.x documentation [Manual de software informático]. Descargado de https://docs.openzeppelin.com/contracts/4.x/ (Recuperado el 30 de mayo de 2025)

Parrondo, L. (2018). Tecnología *blockchain* una nueva era para la empresa [Presentación]. Recuperado de https://www.slideshare.net/luisparrondologo. *Revista de Contabilidad y Dirección*, *27*(6), 11–

Persmissionless blockchains vs permissioned blockchains. (s.f.). https://www.alchemy.com/overviews/permissionless-vs-permissioned-blockchains.

Pillai, B., Biswas, K., Hóu, Z., y Muthukkumarasamy, V. (2022). *Cross-blockchain technology: Integration framework and security assumptions.* https://ieeexplore.ieee.org/document/9756564.

Polygon doc. (s.f.). https://docs.polygon.technology/pos/overview/.

The raft consensus algorithm. (s.f.). https://raft.github.io/.

SQLite. (s.f.). Sqlite documentation [Manual de software informático]. Descargado de https://www.sqlite.org/ (Recuperado el 30 de mayo de 2025)

Turbo. (s.f.). Turbo documentation [Manual de software informático]. Descargado de https://turbo.build/ (Recuperado el 30 de mayo de 2025)

TypeScript. (s.f.). Typescript documentation [Manual de software informático]. Descargado de https://www.typescriptlang.org/ (Recuperado el 30 de mayo de 2025)

Understanding the polygon blockchain: Benefits and use cases. (s.f.). https://www.debutinfotech.com/blog/polygon-blockchain-benefits-and-use-cases.

Wormhole doc - run a custom relayer. (s.f.). https://wormhole.com/docs/infrastructure/relayers/run-relayer/.

Wormhole docs. (s.f.). https://wormhole.com/docs/.

Wormhole finality. (s.f.). https://wormhole.com/docs/build/reference/consistency-levels/.

Wormhole repo - nftbridge.md. (s.f.). https://github.com/wormhole-foundation/wormhole/blob/main/whitepapers/0006_nft_bridge.md.

Wormhole repo - nftbridge.sol. (s.f.). https://github.com/wormhole-foundation/wormhole/blob/main/ethereum/contracts/nft.

Yi, H. (2023). *A post-quantum blockchain notary scheme for cross-blockchain exchange.* https://www.sciencedirect.com/science/article/abs/pii/S0045790623002562.

Zheng, Z., Xie, S., Dai, X., Hong-Ning , y Wang, H. (2017, 06). An overview

of blockchain technology: Architecture, consensus, and future trends.. doi: 10.1109/BigDataCongress.2017.85

Anexo A

Anexo Marco Teórico

A.1. Algoritmos de consenso

Proof Of Work (POW)

Proof Of Work es una estrategia de consenso utilizada por ejemplo, en la red Bitcoin (Zheng y cols., 2017). En una red descentralizada, alguien debe ser seleccionado para registrar las transacciones. En PoW, si un nodo quiere publicar un bloque de transacciones, debe realizar una gran cantidad de trabajo para demostrar que no es probable que ataque la red. Generalmente, este trabajo implica realizar cálculos computacionales. Cada nodo de la red calcula un valor hash a partir del encabezado del bloque. El consenso requiere que el valor calculado sea igual o menor que un valor específico determinado. El encabezado del bloque contiene un valor llamado nonce, que se modifica hasta conseguir un valor de hash válido. Cuando un nodo alcanza el valor objetivo, transmite el bloque a los demás nodos, y estos deben confirmar mutuamente la corrección del valor hash. Si el bloque es validado, los demás mineros [1] lo agregarán a sus propias blockchains.

Uno de los problemas que pueden surgir con los algoritmos de consenso PoW, es que varios nodos encuentren un nonce válido casi simultáneamente. Como resultado, pueden surgir bifurcaciones, como se muestra en la Figura A.1. Sin embargo, es poco probable que dos ramas competidoras generen el siguiente bloque al mismo tiempo nuevamente. Por lo tanto, la cadena más larga se considerará la auténtica. Por ejemplo, supongamos que se crean dos bifurcaciones debido a los bloques validados simultáneamente, U4 y B4. Los mineros continúan minando sus bloques hasta que se encuentra una cadena más larga. La cadena formada por B4, B5 es más larga, por lo que los mineros en U4 cambiarán a la rama más larga. (Zheng y cols., 2017)

Los mineros deben realizar una gran cantidad de cálculos computacionales, pero este trabajo desperdicia demasiados recursos. Para mitigar esta pérdida, se

[1] Los nodos que calculan los valores hash se llaman mineros, y el proceso de PoW se denomina minería en Bitcoin

han diseñado algunos protocolos PoW en los que el trabajo realizado puede tener aplicaciones secundarias. Por ejemplo, Primecoin busca cadenas de números primos especiales que pueden ser utilizadas para la investigación matemática. (King, 2013)

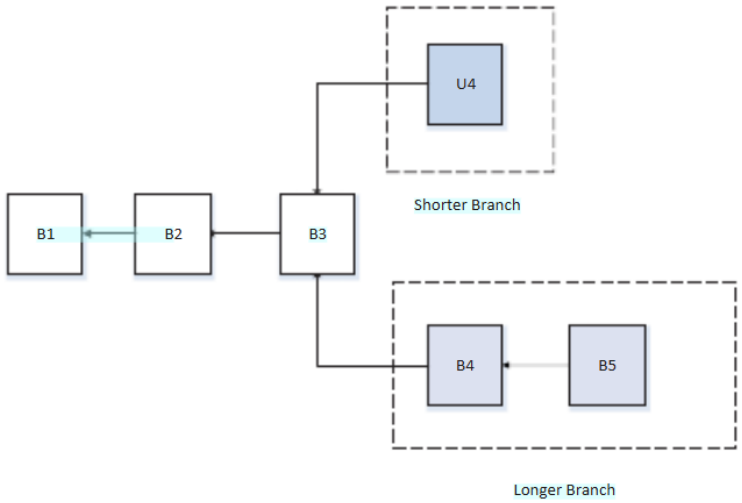


Figura A.1: Caso de bifurcaciones (Zheng y cols., 2017)

Crash Fault Tolerance (CFT)

Crash Fault Tolerance es la capacidad de un sistema distribuido para seguir funcionando incluso si algunos de sus nodos fallan al detenerse inesperadamente (crash). Un sistema con CFT puede resistir fallas en nodos siempre que estos simplemente se apaguen o dejen de responder.

Un ejemplo de algoritmo de consenso CFT es Raft, el cual es utilizado por defecto en Hyperledger Fabric. (*The Raft Consensus Algorithm*, s.f.)

Practical Bizantine Fault Tolerance

En pocas palabras, la tolerancia a fallas bizantinas (BFT) es la propiedad de un sistema que puede resistir la clase de fallas derivadas del problema del dilema de los generales bizantinos. (Graham, Lamport, Shostak, y Pease, 1982) Esto significa que un sistema BFT puede continuar funcionando incluso si algunos de los nodos fallan o actúan maliciosamente.

Hay más de una solución posible al Problema de los Generales Bizantinos y, por lo tanto, múltiples formas de construir un sistema BFT. Del mismo modo, hay varios enfoques diferentes para que una blockchain alcance la tolerancia a fallas bizantinas.

Por ejemplo, PBFT (Practical Byzantine Fault Tolerance) es un algoritmo de replicación diseñado para tolerar fallas bizantinas (Castro y Liskov, s.f.). Hyperledger Fabric utiliza PBFT como su algoritmo de consenso, ya que PBFT puede manejar hasta 1/3 de réplicas bizantinas maliciosas (Zheng y cols., 2017). En el algoritmo PBFT, se tiene que un nuevo bloque se determina en rondas. En cada ronda, se selecciona un nodo primario según ciertas reglas, y este es responsable de ordenar las transacciones. El proceso se divide en tres fases, **Pre-preparación**, **Preparación** y **Compromiso (commit)**. En cada fase, un nodo avanza a la siguiente si ha recibido votos de más de 2/3 de todos los nodos. Debido a esto, PBFT requiere que todos los nodos sean conocidos en la red. Similar a PBFT, el Stellar Consensus Protocol (SCP) también es un protocolo de acuerdo bizantino. Sin embargo, mientras que en PBFT cada nodo debe consultar a los demás nodos, en SCP los participantes pueden elegir en qué conjunto de nodos confiar.

DPOS

DPOS (Delegated Proof of Stake). La principal diferencia entre PoS y DPOS es que PoS es una democracia directa, mientras que DPOS es una democracia representativa (Zheng y cols., 2017). En DPOS, los propietarios de tokens eligen delegados que se encargan de generar y validar bloques. Al haber significativamente menos nodos validando los bloques, estos pueden ser confirmados más rápido, lo que permite una validación rápida de las transacciones. A su vez, los delegados pueden ajustar parámetros de la red, como el tamaño del bloque y los intervalos entre bloques. Por otro lado, los usuarios no tienen que preocuparse por delegados deshonestos, ya que pueden ser expulsados mediante votación. DPOS es la base del protocolo de Bitshares.

A.2. Patrones de diseño

El *API Gateway* es un *middleware* que simplifica la conectividad entre aplicaciones tradicionales y blockchains, mediante una *API* unificada. En vez de que una aplicación requiera contar con nodos en cada una de las blockchains con las que se quiere comunicar, la *API Gateway* ofrece una interfaces de alto nivel para poder comunicarse con distintas blockchains, sin la necesidad de mantener nodos en estas, ni de entender como funcionan las mismas a bajo nivel.

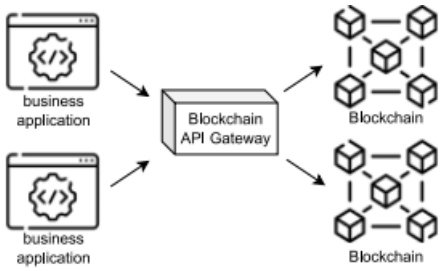


Figura A.2: Patron API Gateway (Llambías y cols., 2022)

El patron *Transaction Aggregator* es muy similar al *relayer*, con la diferencia que la entidad *aggregator* escucha a varias transacciones en una blockchain origen y las agrupa en un solo conjunto de datos, llamado *rollup*. Luego, se envía el *rollup* a una blockchain destino como una unica transaccion. Esto disminuye los costos, ya que procesar un *rollup* es mas barato que procesar cada transacción individualmente.

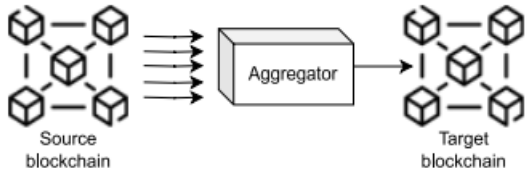


Figura A.3: Patron Transaction Aggregator (Llambías y cols., 2022)

Patrones de Seguridad

El *Light Client* consiste en un *smart contract* en la blockchain destino que almacena un estado de consenso de una blockchain origen, y verifica transacciones entrantes sin la necesidad de confiar en una entidad tercera de confianza. Estas transacciones se recibirian a traves de un *relayer*, sin embargo, el *Light Client smart contract* es el encargado de realizar la verificacion.

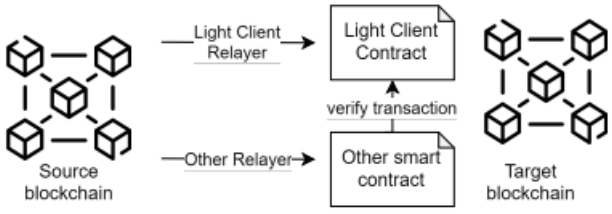


Figura A.4: Patron Light Client (Llambías y cols., 2022)

A.3. Soluciones de interoperabilidad

El *notary scheme* es una solución *third party* de confianza, compuesta por una entidad (*notary*), que monitorea varias blockchains y verifica las transacciones entre ellas. Los ejemplos mas conocidos de esta solución incluyen a Binance y Coinbase, que le permiten a sus usuarios intercambiar criptomonedas entre blockchains distintas.

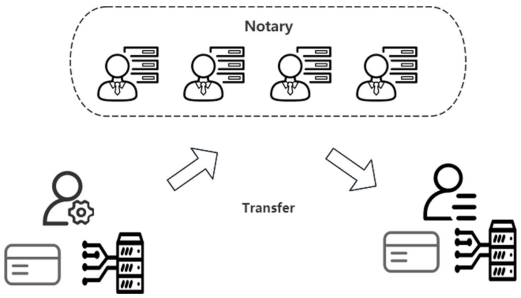


Figura A.5: Ejemplo de Notary Scheme (Yi, 2023)

Los *hash locks* son un mecanismo criptográfico utilizado para poder realizar transacciones seguras sin la necesidad de terceros. Para entender este mecanismo, tomemos dos partes como ejemplo, Alice y Bob. Alice quiere intercambiar un asset que tiene en la blockchain A, por un asset que Bob tiene en la blockchain B. Entonces, se tienen los siguientes pasos:

1. Alice crea una llave secreta generando una cadena aleatoria de caracteres y un hash de esa llave secreta.

2. Alice bloquea el *asset* en la blockchain A en un *smart contract* de *hash locking*, de tal manera que si se presenta el secreto correspondiente al hash, el *asset* será transferido a Bob.

3. Alice comparte el hash con Bob.

4. De la misma manera, Bob bloquea su *asset* en un *smart contract* en la blockchain B utilizando el mismo hash que recibió de Alice, de tal manera que si se presenta el secreto correspondiente al hash, el *asset* será transferido a Alice.

5. Alice ve que se bloqueó el *asset* de Bob, y luego Alice presenta la llave secreta para reclamar el *asset* de la blockchain B. En este momento, se le revela la llave secreta a Bob.

6. Bob utiliza la llave secreta para reclamar el *asset* de la blockchain A antes que expire el tiempo.

Además, se tiene que los *smart contracts* utilizados por Alice y Bob tienen un mecanismo de *timeout* que asegura que, si los *assets* fueron bloqueados por una cantidad de tiempo definida y no fueron intercambiados, la operación sera cancelada, y estos se desbloquearan para retornar a sus respectivos dueños. (Aračić, 2023)

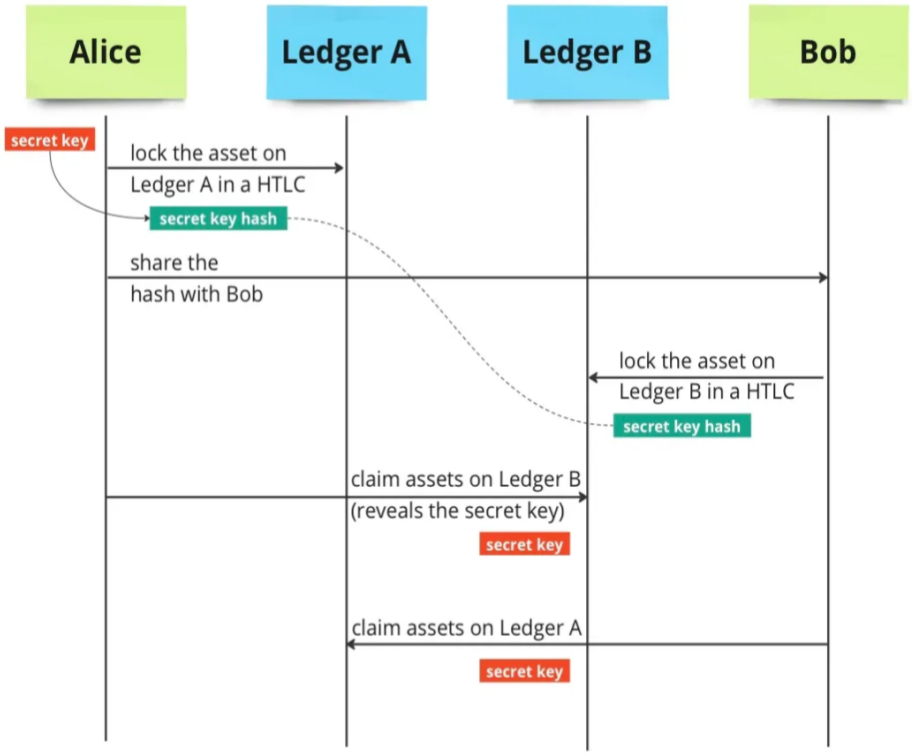


Figura A.6: Ejemplo de Hash Locking (Atomic Swap) (Aračić, 2023)

A.3.1. Soluciones de interoperabilidad

En esta sección se verán algunas de las soluciones actuales para la interoperabilidad entre blockchains relevantes al proyecto. Se verán más en el Anexo A.3.

Conectores públicos

Este tipo de solución se utiliza principalmente en blockchains publicas, en casos de uso relacionados a las criptomonedas. Dentro de esta categoría, se tienen tres subcategorías distintas, *Sidechain/Relays*, *Notary Scheme*, y los *Hash Locks* o *Atomic Swaps*. (Llambías y cols., s.f.)
En particular, una *sidechain* es una blockchain "secundaria", conectada a una

blockchain "principal" (o *mainchain*), actuando como una extensión de la misma. La comunicación entre estas blockchains puede ser tanto unidireccional (la *mainchain* envía información a la *sidechain*), o también bidireccional. Esto permite la transferencia de *assets* entre ambas blockchains, mientras que la *sidechain* sigue siendo su propia blockchain independiente. Esto puede ser útil para mejorar la escalabilidad de la *mainchain*. Un ejemplo de *sidechain* es Polygon, cuya *mainchain* es Ethereum.

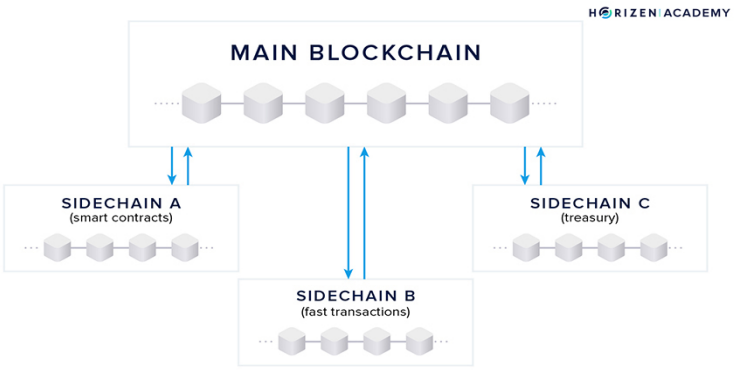


Figura A.7: Ejemplo de Sidechain (*Horizen Academy - What is a Sidechain?*, s.f.)

Blockchain de blockchains

Una blockchain de blockchains es un sistema el cual, a través de un mecanismo de consenso, organiza bloques que contienen transacciones de distintas blockchains. De aquí el nombre de "blockchain de blockchains", ya que se tiene un registro de muchas blockchains distintas. Este sistema cuenta con distintos componentes, como *relays* (para recibir transacciones de una blockchain origen), conectores y routers para rutear y enviar la transacción a la blockchain destino, un método de consenso para validar las transacciones y proveer seguridad a la red entera, y validadores en la blockchain destino, para validar las transacciones entrantes. Cosmos y Polkadot son ejemplos de estos sistemas. (Llambías y cols., s.f.)

Conectores híbridos

Los conectores híbridos hacen referencia a las soluciones de interoperabilidad que no son ni blockchain de blockchains ni conectores públicos. Esta solución apunta a la interoperabilidad entre blockchains tanto publicas como privadas. Dentro de esta categoría, se cuenta con, por ejemplo, *Trusted Relays*, *Blockchain Agnostic Protocols* y *Blockchain Migrators*. (Belchior, Vasconcelos, Guerreiro, y Correia, 2021)
Particularmente, los *trusted relays* son entidades terceras de confianza que actúan

como intermediarios para transferir transacciones entre diferentes blockchains, redirigiendo las transacciones generadas en una blockchain origen hacia una blockchain destino. Esto permite que los usuarios puedan definir sus propias lógicas de negocio, ya que cada blockchain puede tener sus propios protocolos y estructuras. Estos *relays* cuentan con una red de validadores, los cuales firman las transacciones entre blockchains y las envían a la blockchain destino. Una transacción se considera valida cuando una cantidad suficiente de validadores la valida. Se puede tener *relays* centralizados, donde hay una única entidad que valida las transacciones, como así también *relays* descentralizados, donde la validación se reparte entre varias partes.

Anexo B

Anexo Análisis de Soluciones

B.1. Wormhole

Smart Contracts principales:

(*Wormhole Repo - NFTBridge.sol*, s.f.)

TokenState

El smart contract TokenState se utiliza para guardar información del NFT, cómo por ejemplo nombre, dueño, *address* del contrato nativo y "chainId". El nativeAddress sirve para saber la dirección del contrato del token que se refiere. El "chainId" indica la blokchain de donde se originó el token. Este identificador se utiliza para diferenciar los NFTs que son nativos a la blockchain de los que son "Wrapped". En conjunto con el chainId y nativeAddress, se puede identificar el token .

TokenImplementation

Este *smart contract* es una implementación personalizada del estándar ERC-721 (NFTs) adaptada para interoperar con el *Bridge* de Wormhole. Crea y gestiona NFTs, soporta operaciones básicas como el mint, burn, transfer y approve. Utiliza el *TokenState*.

NFTBridgeState

Contiene toda la información del estado del bridge, como por ejemplo *mappings* de qué contratos son nativos y cuales son *Wrapped*, de las transferencias hechas y de las direcciones de los contratos del Bridge en las otras blockchains. Esto sirve para saber que hacer en caso de recibir una orden de transferencia.

NFTBridge

Es el contrato que permite transferir NFTs entre cadenas, creando *wrapped tokens* de ser necesario. Usa Wormhole para enviar, recibir mensajes y mantener la integridad de los mismos.

B.1.1. Proceso de Transferencia

El proceso de transferencia se divide en dos partes, el envío desde la Blockchain Origen y la recepción desde la Blockchain Destino.

Envío del token

El proceso para iniciar una transferencia de un NFT de una blockchain origen a una blockchain destino es el siguiente:

1. El usuario que desea transferir un NFT llama al *smart contract* NFTBridge.
2. NFTBridge verifica si el NFT que se quiere enviar es un *Wrapped Token* o no, y en base a esto se eligen las direcciones de Chain ID y *contract address* correspondientes.
3. El contrato NFTBridge ejecuta una transferencia del token de la cuenta del usuario al NFTBridge, funcionando como un Lock, ya que el usuario pierde la capacidad de interactuar con el NFT.
4. Si se trata de un *Wrapped Token*,se realiza un *burn*, ya que es el envío de una devolución de un token.
5. Se emite un evento de la transferencia, el cual contiene los datos del token, la blockchain, los *smart contracts*, etc. Este evento es escuchado por el relayer.

Recepción del token

El proceso para recibir el token en el *smart contract* es el siguiente:

1. El NFTBridge verifica si el token está *wrapped* o no.
2. Si no está *wrapped*, NFTBridge despliega un smart contract a partir del bytecode de TokenImplementation y se guarda la dirección de este en el mapping de tokens *wrapped* de NFTBridgeState.
3. Si `tokenChainID == contract.chainID`, significa que el token está siendo devuelto, y el contrato transfiere el token al usuario (funcionando como un unlock al devolverle el ownership al usuario). En caso contrario (el token está siendo prestado a la blockchain actual), NFTImplementation realiza *mint* de un nuevo *Wrapped Token*.

(*Wormhole Repo - NFTBridge.md*, s.f.) (*Wormhole Doc - Run a Custom Relayer*, s.f.)

B.2. Chainlink

Chainlink es un proyecto que mediante una red de nodos descentralizados, permite que los smart contracts accedan a datos del mundo real de manera segura y confiable. Su objetivo es resolver el problema de la conectividad entre blockchains y fuentes de información externas, como precios de mercado, datos climáticos o resultados de eventos.

Chainlink implementa un protocolo de interoperabilidad de blockchain llamado Chainlink Cross-Chain Interoperability Protocol (CCIP) cuya arquitectura puede visualizarse en la figura 4.2. Este protocolo permite enviar datos arbitrarios (codificados como bytes) a un smart contract de otra blockchain, permite transferir tokens a smart contracts o a una cuenta, y también permite transferir ambas en simultáneo en una única transacción (datos y tokens).

A continuación, se realizará una breve descripción de los componentes más importantes de la arquitectura. A los componentes externos a las blockchain se los denomina componentes *Offchain*, mientras que lo son parte de la misma, OnChain. Luego, se explicará como funciona el proceso de transferencia.

B.2.1. Componentes OffChain:

Se cuenta con 3 redes independientes descentralizadas que se encargan de ejecutar y verificar las transacciones. Esta separación permite mejor división de responsabilidades, mayor disponibilidad y dotar de mayor seguridad al bridge. . Se tienen al **Commiting DON**, **Executing DON**, y la **Risk Management Network**

Committing DON

El Committing DON tiene varios "jobs" (procesos agendados), donde cada job monitorea transacciones cross-chain entre una blockchain origen y destino. Cada job monitorea los eventos de un contrato OnRamp de una blockchain origen. El job espera por la finalidad del mensaje, lo cual dependerá de cada blockchain origen. Luego, agrupa transacciones y crea un Merkle root. Este Merkle root es firmado por un cuorum de nodos que son parte del Committing DON. Finalmente, el job escribe el Merkle root en el contrato Commit Store, para la blockchain destino que corresponda.

Executing DON

Como el Committing DON, el Executing DON tiene varios "jobs", cada uno de los cuales ejecuta transacciones cross-chain entre una blockchain origen y una destino. Cada job monitorea los eventos de un contrato OnRamp en la blockchain origen. El job chequea si la transacción es parte del Merkle root de la Commit Store. Luego, espera a que la Risk Management Network apruebe el mensaje. Finalmente, el job crea una Merkle proof válida, la cual es verificada por el contrato OffRamp junto al Merkle root de la Commit Store. Luego de

estos chequeos, el job llama al contrato OffRamp para completar la transacción en la blockchain destino.

Risk Management Network

La Risk Management Network es un conjunto de nodos independientes que monitorean los Merkle roots que son enviadas por el Committing DON a la Commit Store. Cada nodo compara los Merkle roots "commiteados" (es decir, que ya han sido registrados en una blockchain como prueba de la integridad de los datos transferidos) con las transacciones recibidas por el contrato On-Ramp. Luego de que la verificación es exitosa, el nodo llama al contrato de Risk Management para aprobar el Merkle root commiteado. Cuando hay suficientes votos de aprobación, el Merkle root queda disponible para ejecución. En caso de que falle la verificación, el nodo llama al contrato de Risk Management para desaprobar el sistema. Si el cuorum de desaprobación se alcanza, se pausa el contrato de Risk Management para evitar que se ejecute cualquier transacción.

B.2.2. Componentes OnChain

Estos son el **Router**, **Commit Store**, **OnRamp**, **OffRamp**, **Token Pools** y **Risk Management Network**.

Router

El router es un contrato encargado de iniciar las interacciones cross-chain. Cada una de las blockchains contiene un contrato Router, el cual se encarga de dirigir las instrucciones al componente OnRamp que corresponda. Para las blockchain destino, el router es el encargado de enviar las tokens a la cuenta destino, o enviar el mensaje al *smart contract* receptor, según corresponda el caso.

Commit Store

El Committing DON interactúa con el Commit Store, para guardar el Merkle root del mensaje finalizado en la blockchain origen. Este Merkle root debe ser aprobado por la Risk Management Network antes de que el Executing DON pueda ejecutarlos en la blockchain destino. La Commit Store se asegura de que el mensaje esté aprobado por la Risk Management Network, y existe una Commit Store por cada lane (camino unidireccional de una blockchain origen a una destino).

OnRamp

Existe un contrato OnRamp por lane que se encarga de:

- Realiza validaciones específicas de la blockchain destino (ej. sintaxis de la account address).

- Lleva registro de números de secuencia para conservar el orden de los mensajes.
- Verifica los límites de gas y de tamaño del mensaje.
- Maneja el pago de la tarifa por la transacción.
- Interactúa con el Token Pool en caso de que el mensaje contenga alguna transferencia de tokens.
- Emite un evento, el cual es escuchado por el Commiting DON.

OffRamp

Existe un contrato OffRamp por lane que se encarga de:

- Asegurarse de la autenticidad del mensaje, validando la proof provista por el Executing DON con un Merkle root aprobado de la Commit Store.
- Asegurarse que las transacciones solo se ejecuten una vez.
- Los mensajes validados son transmitidos al Router.
- Si las transacciones incluyen transferencia de tokens, el contrato OffRamp llama a la TokenPool para la transferencia.

Token Pools

Cada tipo de Token esta asociada con su propia Token Pool. Un Token Pool es una capa de abstracción por encima de ERC-20, designada para facilitar el OnRamping y OffRamping para transacciones relacionadas a transferencia de tokens. Las token pools están configuradas para realizar lock o burn de tokens en la blockchain origen, y unlock o mint de tokens en la blockchain destino. Se tiene entonces cuatro posibles opciones principales:

- Burn y Mint: Se queman las tokens de la blockchain origen, y se emite una cantidad equivalente de tokens en la blockchain destino.
- Lock y Mint: Se bloquean tokens en la blockchain origen, y se realiza *mint* de una cantidad equivalente de **Wrapped Tokens** en la blockchain destino. Estos tokens envueltos representan los tokens originales bloqueados, y pueden ser utilizados en la blockchain destino como si fueran los tokens originales. Este proceso asegura que los tokens originales estén completamente colateralizados, es decir, cada token envuelto en la blockchain de destino está respaldado por un token bloqueado en la blockchain de emisión. Los tokens envueltos pueden ser transferidos a blockchains destino que no emitan estos tokens.

- Burn y Unlock: Este mecanismo es el inverso de Lock y Mint, y se utiliza cuando se quiere devolver tokens a la blockchain de la que fueron prestados. Los tokens de la blockchain que recibió el préstamo se queman, y una cantidad equivalente de tokens se desbloquean en la blockchain destino.

- Lock y Unlock: Los tokens son bloqueados en la blockchain de origen, y una cantidad equivalente son desbloqueados en la blockchain destino.

Risk Management Network

El contrato de Risk Management Network mantiene la lista de las direcciones de los nodos de Risk Management que tienen como rol aprobar o desaprobar las transacciones *cross chain*. El contrato también contiene la lógica de consenso para la aprobación de un Merkle Root que se encuentre en la Commit Store.

B.2.3. Proceso de Transferencia

Para ejemplificar como interactúan estos componentes entre sí, a continuación, se detalla el flujo de una transferencia de origen a destino en una serie de pasos:

1. Usuario envía transacción a *smart contract* del router.

2. Router le envía información al *smart contract* del OnRamp que le corresponda (hay uno por cada "lane").

3. Interactúa con el token pool en caso de que la transacción incluya transferencia de tokens (haciendo, lock, burn o lo que corresponda) y aparte emite un evento de esto.

4. El committing DON escucha el evento y se encarga de validar la información de la transacción y de que esta transacción esté finalizada.

5. Una vez finalized, el Committing DON crea un Merkle root y lo envía a la Commit Store.

6. La Risk Management Network valida el Merkle Root.

7. El Executing DON chequea en el commit store por esta transacción y la envía al *smart contract* del OffRamp.

8. El *smart contract* del OffRamp se encarga de verificar que solo se ejecute una vez la acción y, en caso que corresponda, hacer la acción en el Token Pool. Además, envía la información al Router.

9. El Router envía el mensaje al *smart contract* del Receiver.

B.3. LayerZero

LayerZero es un protocolo de interoperabilidad diseñado para facilitar la comunicación segura entre distintas blockchains. Su arquitectura permite que las aplicaciones descentralizadas (dApps) envíen y reciban mensajes entre blockchains sin necesidad de intermediarios centralizados (*Layer Zero Docs - Protocol Overview*, s.f.). LayerZero cuenta con una libreria MessageLib, la cual consiste en *smart contracts* que se encargan de gestionar la verificación y configuración de seguridad en la transferencia de mensajes entre blockchains. Los Ultra Light Nodes (ULN) son la implementación por defecto de esta librería, aunque los desarrolladores pueden realizar sus propias configuraciones. También se tiene una red de verificadores independientes decentralizados (DVN), que validan la integridad de los mensajes entre las blockchains. Cuando se inicia una transferencia, MessageLib toma los datos del mensaje, aplica la configuración establecida, y genera un paquete estructurado que contiene la informacion. Luego, se emite un evento, el cual es escuchado por la DVN. Esta se encarga de validar el paquete, y lo transmite a la blockchain destino. En la blockchain destino, MessageLib verifica que el hash del payload del mensaje coincida con la configuración de seguridad establecida.

B.3.1. Componentes principales:

En la Figura 4.2 se tiene un diagrama de los distintos componentes de LayerZero. En particular, veamos el **LayerZero Endpoint**, **MessageLib**, **Omnichain Application (OApp)**, y el **Executor**

LayerZero Endpoint

Es un contrato inmutable que implementa interfaces estandarizadas para el envío, recepción y configuración de mensajes.

Omnichain Application (OApp)

Son *smart contracts* que se pueden desplegar para pasar mensajes genéricos entre distintas blockchains. Es un tipo de aplicación donde los desarrolladores puedan construir utilizando la tecnología de LayerZero.

MessageLib

Es una una librería de verificación que los dueños de las aplicaciones pueden configurar para usar su aplicación (*Layer Zero Docs - Message Library*, s.f.). El protocolo usa esta configuración antes de enviar y recibir mensajes. Se encarga de:

- Aplicar la configuración de la OApp para generar paquetes de mensajes.
- Emitir mensajes hacia la pila de seguridad y los ejecutores seleccionados.

- Verificar en la blockchain de destino que el "payload hash" del paquete cumpla con la configuración de seguridad antes de permitir su ejecución.

Executor

Se encarga de ejecutar las instrucciones de los mensajes verificados de la MessageLib en la blockchain destino.

B.3.2. Proceso de Transferencia

Para mostrar como interactúan estos componentes, se ejemplifica con un ejemplo de proceso de transferencia. Se va a proceder a dividir el proceso de transferencia se divide en dos partes: el envío desde la blockchain de origen y la recepción en la blockchain de destino.

Envío

El proceso para iniciar una transferencia desde la blockchain de origen es el siguiente:

1. El usuario envía una transacción a un smart contract de una OApp.
2. La OApp envía una transacción en la misma cadena, cobrando fees en el token nativo de la blockchain y en el LayerZero token.
3. La MessageLib genera un paquete con la información proporcionada por el usuario, aplicando la configuración de la OApp.
4. Se emite un evento PacketSent con los datos del paquete.
5. La red descentralizada de verificadores (DVN) escucha el evento, valida la transacción y, tras confirmar su finalidad, envía un mensaje al Executor.

Recepción

El proceso para recibir el mensaje en la blockchain de destino es el siguiente:

1. Las DVN configuradas en la pila de seguridad de la OApp verifican el payload hash del paquete en la MessageLib de destino.
2. Una vez que el mensaje cumple con el nivel de consenso requerido por la pila de seguridad, el paquete se marca como "commiteado".
3. El Executor envía el nonce del paquete verificado al `LayerZero Endpoint` de destino.
4. El Executor ejecuta las instrucciones del mensaje en la blockchain de destino.

B.4. Celer Bridge

Tarifa Base Predefinida (destTxFee)

En el contrato NFTBridge.sol, se tiene un mapping destTxFee el cual tiene guardado la tarifa para el mint de NFTs para cada una de las blockchain destino soportadas por Celer Bridge. Esta tarifa puede ser actualizada a través de la función setTxFee. La tarifa es un estimado basado en los costos promedios de gas en la blockchain destino.

Cálculo de Tamaño de Mensaje

La función totalFee calcula la tarifa total a pagar por el usuario sumando dos componentes, la tarifa de pasaje de mensaje, la cual es proporcional al tamaño del mensaje siendo transferido (incluyendo la metadata del NFT, el token ID, y el token URI), así como la tarifa base, la cual consta de el costo de ejecutar el mint en la blockchain destino (destTxFee)

Entonces, cuando un usuario inicia una transacción, ocurre lo siguiente:

1. La función `totalFee` en el contrato `NFTBridge` calcula la tarifa total combinando la tarifa fija `destTxFee` y la tarifa calculada para el pasaje del mensaje.
2. El usuario paga esta tarifa en la blockchain origen con la moneda nativa de la misma.
3. Una vez que la transacción es procesada, los relayers encargados del manejo del mensaje en la blockchain destino utilizan el `destTxFee` para cubrir los costos de gas para el mint de una NFT.
4. Esta `destTxFee` puede ser actualizada periódicamente por los operadores del bridge basándose en las fluctuaciones del costo del gas en la blockchain destino, permitiendo al bridge tener costos actualizados.

(*Celer cBridge Docs - NFT Bridge - Introduction*, s.f.) (*Celer cBridge Docs - NFT Bridge - NFT Bridge Fee*, s.f.) (*Celer Network Repo - NFTBridge.sol*, s.f.)

B.5. AllBridge

En el contrato bridge.rs, se define cómo se calculan las tarifas basadas en la cantidad transferida, el tamaño del stake [1], el tamaño del pool[2], y el multiplicador de tarifas. El contrato tiene un atributo base_fee_rate_bp, el cual representa

[1] Si el usuario ha participado en el staking del token de AllBridge (ABR), puede recibir descuentos en las tarifas

[2] Si el pool es grande, las tarifas son más bajas, ya que hay suficiente liquidez para facilitar la transacción, de lo contrario, puede aumentar la tarifa para evitar desequilibrios en la liquidez

la tasa de tarifa base en “basis points”. Esto se ajusta para cada blockchain, donde blockchains como Ethereum pueden tener tarifas fijas en lugar de porcentajes. También cuenta con un fee_multiplier, un multiplicador el cual ajusta las tarifas dependiendo del bridge y la blockchain. Se tiene la función calculate_fee, la cual toma el monto a transferir, el tamaño del stake (si aplica, por ejemplo, si el usuario tiene participación en un pool), y el tamaño del pool para calcular el monto de la tarifa que será deducido en la transacción de bloqueo (lock) en la blockchain de origen. El contrato instructions.rs maneja el proceso de la transacción de Lock, donde la función lock llama a la instrucción BridgeProgramInstruction::Lock, que aplica la tarifa calculada a la cantidad enviada en la blockchain de origen. La tarifa puente se cobra durante esta transacción.

(*Allbridge Repo - State*, s.f.)

Anexo C

Anexo Diseño de la solución

C.1. Caso de uso

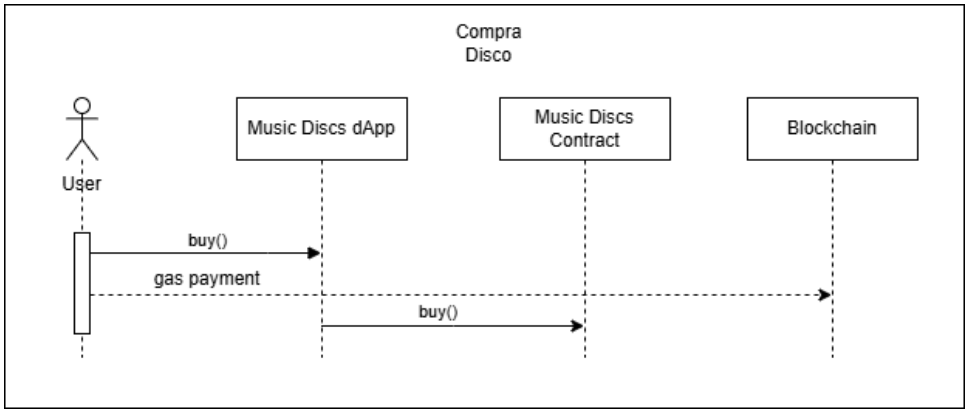


Figura C.1: Comprar Disco

Compra de disco (Figura C.1)

El usuario inicia la compra desde la *Music Discs dApp* llamando al método *buy()*. La aplicación interactúa con el *Music Discs Contract* para realizar el *minting* de un nuevo *NFT* que representa el disco adquirido. Este contrato asigna automáticamente el *NFT* a la billetera del usuario, completando el proceso de compra.

Anexo D

Anexo Implementación

D.1. Implementación de los Patrones

En esta sección se analizan los detalles de la implementación de los patrones de interoperabilidad.

D.1.1. Relayer

Para la implementación del `Relayer`, veamos primero como se implementaron los conectores del mismo, comenzando con el `EVM Connector`. En la Figura D.1 se ve la función encargada de la escucha de eventos en una blockchain EVM, donde se inicializan los *listeners* del evento `Message` emitido por los *smart contracts* de los bridges.

```
export async function listenBlockchainEvents(
  chain: Blockchain,
  prisma: PrismaClient,
  config: Config
) {

  const contracts = await getContracts(config);
  const provider = getProvider(config);

  provider.on("block", async () => {
    await processPendingFinalizedBlocks(chain, prisma,
        config);
  });

  for (const contract of Object.values(contracts)) {
    (contract as unknown as
        PermanentTransfer).on<MessageEvent.Event>(
      (contract as unknown as
          PermanentTransfer).getEvent("Message"),
      (
        destinationBlockchain,
        destinationContract,
        operation,
        operationParams,
        eventDetails
      ) =>
        eventReceived({
          contract,
          destinationBlockchain,
          destinationContract,
          operation,
          operationParams,
          chain,
          prisma,
          eventDetails,
        })
    );
  }
}
```

Figura D.1: Función `listenBlockchainEvents` (EVM Connector)

Además de los *listeners* del evento `Message` para los distintos *smart contracts*, también se inicializo un *listener* de los eventos generados cada vez que se crea un nuevo bloque en la blockchain. Dado que los estados de los bloques solo pueden modificarse con la creación de un nuevo bloque, se implementa el siguiente algoritmo:

En las blockchains EVM, para obtener cuál es el último bloque finalizado en

Algorithm 1 Algoritmo para procesar eventos de nuevos bloques

```
1: while escucha eventos de bloques nuevos do
2:     Obtener número del último bloque finalizado B_último_finalizado
3:     E ← Eventos pendientes con número de bloque B ≤ B_último_finalizado
4:     for all eventos e ∈ E do
5:         Procesar evento e
6:     end for
7: end while
```

la cadena se utiliza la función `getBlock`, que acepta cómo parámetro un filtro del estado de finalización del bloque y devuelve el bloque más reciente que tenga este estado. En nuestro caso, se llama a esta función con valor de parámetro "finalized" y esta devuelve el último bloque finalizado. En las Figuras D.2 y D.3 se tienen las funciones encargadas del procesamiento de los bloques finalizados.

```
async function processPendingFinalizedBlocks(
  chain: Blockchain,
  prisma: PrismaClient,
  config: Config
) {
  const finalizedTransactions = await
      getFinalizedtransactions(prisma, config);

  for (const transaction of finalizedTransactions) {

    await sendEventToRouter(
      {
        header: {
          sourceBlockchain: chain,
          sourceContract: transaction.destinationContract,
        },
        data: {
          destinationBlockchain:
            transaction.destinationBlockchain as Blockchain,
          destinationContract:
              transaction.destinationContract,
          operation: transaction.operation,
          operationParams: transaction.operationParams,
        },
      },
      chain
    );
    await prisma.event.update({
      where: { id: transaction.id },
      data: { processed: true },
    });
  }
}
```

Figura D.2: Función `processPendingFinalizedBlocks` (EVM Connector)

```
async function getFinalizedtransactions(prisma:
    PrismaClient, config: Config) {
  const provider = getProvider(config);
  const latestFinalizedBlock = await
      provider.getBlock("finalized");
  if (!latestFinalizedBlock) {
    return [];
  }
  return (
    await prisma.event.findMany({
      where: {
        blockNumber: { lte:
            Number(latestFinalizedBlock.number)  },
        processed: false,
      },
    })
  ).map((event) => {
    return {
      ...event,
      operationParams: JSON.parse(event.operationParams),
    };
  });
}
```

Figura D.3: Función `getFinalizedtransactions` (EVM Connector)

En la Figura D.4 se ven las funciones equivalentes encargadas de la escucha de eventos para Fabric.

```
async function listenBlockchainEvents() {
  assertIsFabricConfig(configs.FABRIC);
  const network = fabricGateway.getNetwork(
    configs.FABRIC.blockchain.fabricChannel
  );
  const relayerContracts = Object.values(
    configs.FABRIC.blockchain.contracts.relayerContracts
  );
  for (const relayerContract of relayerContracts) {
    const events = await
        network.getChaincodeEvents(relayerContract);
    listenEventsFromOneContract(events);
  }
}
async function listenEventsFromOneContract(events: any) {
  for await (const event of events) {
    const eventData = getDataFromFabricEvent(event.payload);
    sendEventToRouter(eventData);
  }
}
function getDataFromFabricEvent(payload: Uint8Array) {
  const decodedPayload = new TextDecoder().decode(payload);
  const parsedPayload = JSON.parse(decodedPayload);
  return routerEventSchema.parse(parsedPayload);
}
```

Figura D.4: Funciónes *listenBlockchainEvents*, *listenEventsFromOneContract* y *getDataFromFabricEvent* (Fabric Connector)

En el caso de Fabric, el concepto de finalizado no existe cómo tal y se puede asumir que todos los bloques que fueron emitidos con éxito son finalizados. Por lo tanto no se debió realizar ninguna implementación relacionada a la finalización de bloques.

Una vez que el bloque esta finalizado, se envía al router. En la Figura D.5 se puede ver el envió del bloque al router para las EVM.

```
export async function sendEventToRouter(event: RouterEvent,
    chain: Blockchain) {
    const url = routerConfig.endpoint + ":" +
        routerConfig.port;
    const urlWithHttp = url.startsWith("http") ? url :
        `http://${url}`;
    const response = await fetch(urlWithHttp, {
      method: "POST",
      body: JSON.stringify(event),
      headers: { "Content-Type": "application/json" },
    });
}
```

Figura D.5: Función `sendEventToRouter` (EVM Connector)

En la figura D.6 se puede ver el envío del evento al router desde Fabric.

```
function sendEventToRouter(event: any) {
  const options = {
    host: routerConfig.endpoint,
    path: "/",
    port: routerConfig.port,
    method: "POST",
  };

  callendpoint(options, JSON.stringify(event))
}

function callendpoint(options: RequestOptions, data:
    string): Promise<string> {
  return new Promise((resolve, reject) => {
    const req = request(options, (response) => {
      var respStr = "";

      response.on("data", (chunk) => {
        respStr += chunk;
      });
    });
  });
}
```

Figura D.6: Función `sendEventToRouter` (Fabric Connector)

En la Figura D.7 se puede ver cómo los conectores EVM reciben los eventos del router, y luego en la Figura D.8 proceden a realizar el llamado del *smart contract* que corresponda en la blockchain EVM.

```
async function eventReceivedFromRouter(
  req: Request,
  chain: Blockchain,
  config: Config
) {
  const event = await Bun.readableStreamToJSON(req.body);

  const eventValidation =
      routerEventSchema.safeParse(event);
  if (!eventValidation.success) {
    return new Response("Invalid event schema", { status:
        400 });
  }

  await sendEventToBlockchain(eventValidation.data, config,
      chain);

  return new Response(null, { status: 200 });
}
```

Figura D.7: Función `eventReceivedFromRouter` (EVM Connector)

```
async function sendEventToBlockchain(
  event: RouterEvent,
  config: Config,
  chain: Blockchain
) {
  const contracts = await getContracts(config);
  const contract =
    contracts[event.data.destinationContract as keyof
        typeof contracts];
  const operation = event.data.operation;
  const params = event.data.operationParams;
  const nonce = await getNonce(config);
  const response = await (contract as
      any)[operation](...params, { nonce });
}
```

Figura D.8: Función `sendEventToBlockchain` (EVM Connector)

En la Figura D.9 se ve como se reciben los eventos desde el router en el conector de Fabric, y cómo este luego invoca a la función del *smart contract* correspondiente en Fabric en la Figura D.10.

```
async function eventReceivedFromRouter(
  requestBody: string,
  response: ServerResponse
) {
  let event = JSON.parse(requestBody);
  response.writeHead(200, { "Content-Type":
      "application/json" });
  response.end();

  const eventValidation =
      routerEventSchema.safeParse(event);
  const validatedEvent = eventValidation.data;
  await sendEventToFabricBlockchain(validatedEvent);
}
```

Figura D.9: Función `eventReceivedFromRouter` (Fabric Connector)

```
async function sendEventToFabricBlockchain(event:
    RouterEvent) {
    const contract = await getContracts(
      event.data.destinationContract,
      "Admin"
    );
    await contract.submitTransaction(
      event.data.operation,
      ...event.data.operationParams.map((param) =>
          param.toString())
    );
}
```

Figura D.10: Función `sendEventToFabricBlockchain` (Fabric Connector)

D.1.2. Transferencia Permanente

En las Figura D.11 se observa un diagrama de clases del Permanent Transfer. El `Permanent Transfer Smart Contract` hereda del *smart contract* `ERC721Holder`, que permite que el contrato reciba *NFTs*. Además, tiene dependencias con `TokenImplementationInterface`, interfaz definida para el bridge que define las funciones mínimas que un contrato NFT debe tener para ser compatible con el bridge. En las Figuras D.12 y D.13 se tiene a la interfaz `TokenImplementationInterface` para EVM y Fabric.

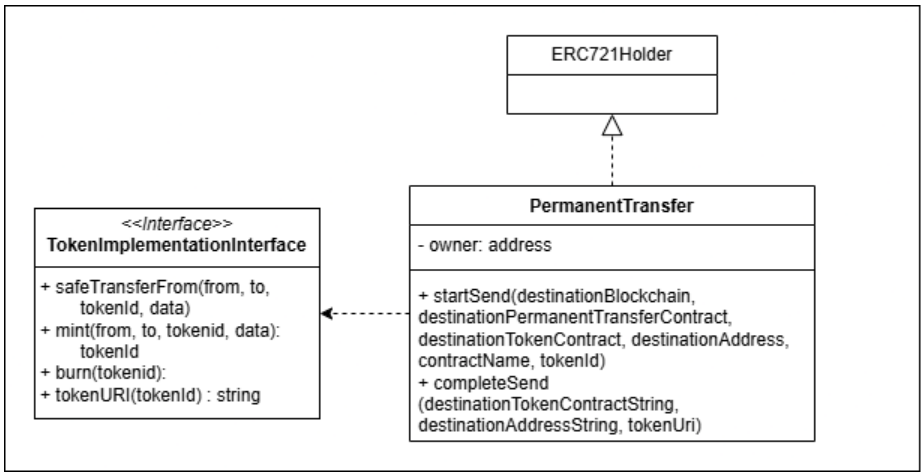


Figura D.11: Diagrama de clases del Permanent Transfer

```
interface TokenImplementationInterface {
    /**
     * @dev Mints a new token
     * @param to The address that will own the minted token
     * @param tokenId The token id to mint
     * @param uri The uri of the token
     */
    function mint(address to, uint256 tokenId, string
        memory uri) external returns (uint256);
    /**
     * @dev Safely transfers token from one address to
         another
     * @param from The current owner of the token
     * @param to The new owner
     * @param tokenId The token to transfer
     * @param data Additional data with no specified format
     */
    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes memory data
    ) external;
    /**
     * @dev Burns a specific token
     * @param tokenId The token id to burn
     */
    function burn(uint256 tokenId) external;
    /**
     * @dev Returns the URI of a specific token
     * @param tokenId The token id to get the URI for
     * @return The URI of the token
     */
    function tokenURI(uint256 tokenId) external view
        returns (string memory);
}
```

Figura D.12: `TokenImplementationInterface` (EVM)

```
export type MintParams = [
    to: string,
    tokenId: string,
    uri: string
];
export type TransferFromParams = [
    from: string,
    to: string,
    tokenId: string
];
export type BurnParams = [tokenId: string];
export type TokenURIParams = [tokenId: string];

interface TokenImplementationInterface {
  /**
   * Mints a new token
   * @param ctx The transaction context
   * @param to The address that will own the minted token
   * @param tokenId The token id to mint
   * @param uri The uri of the token
   */
  Mint(ctx: Context, ...args: MintParams): Promise<any>;
  /**
   * Safely transfers token from one address to another
   * @param ctx The transaction context
   * @param from The current owner of the token
   * @param to The new owner
   * @param tokenId The token to transfer
   * @param data Additional data with no specified format
   */
  TransferFrom(ctx: Context, ...args: TransferFromParams):
      Promise<boolean>;
  /**
   * Burns a specific token
   * @param ctx The transaction context
   * @param tokenId The token id to burn
   */
  Burn(ctx: Context, ...args: BurnParams): Promise<boolean>;
  /**
   * Returns the URI of a specific token
   * @param ctx The transaction context
   * @param tokenId The token id to get the URI for
   * @return The URI of the token
   */
  TokenURI(ctx: Context, ...args: TokenURIParams):
      Promise<string>;
}

export default TokenImplementationInterface;
```

Figura D.13: `TokenImplementationInterface` (Fabric)

Al inicializar la transferencia permanente, primero se llama a la función `startSend` (Figura D.14)

```
function startSend(
    string memory destinationBlockchain,
    string memory destinationContract,
    string memory destinationTokenContract,
    string memory destinationAddress,
    TokenImplementationInterface originTokenContract,
    uint256 tokenId
) external payable {
    require(msg.value >= 0.005 ether, "Insufficient fee
        sent");
    originTokenContract.safeTransferFrom(msg.sender,
        address(this), tokenId, "");
    string memory tokenUri =
        originTokenContract.tokenURI(tokenId);
    originTokenContract.burn(tokenId);
    emit Message(
        destinationBlockchain,
        destinationContract,
        "completeSend",
        abi.encode(
            destinationTokenContract,
            destinationAddress,
            tokenUri
        )
    );
}
```

Figura D.14: Función `startSend` (EVM)

Luego de transferido el NFT al `Permanent Transfer Smart Contract`, y que este realice *burn* del mismo, el evento `Message` emitido por la función `startSend` es escuchado por el `Relayer`. Una vez que la transacción fue enviada a la blockchain destino, el conector de la blockchain destino llamara a la función `completeSend` del `Permanent Transfer Smart Contract` de la blockchain destino. Para EVM, se tiene la implementación de la Figura D.15, donde se realiza *mint* del nuevo NFT, asignándoselo a la billetera destino.

```
    function completeSend(
        string memory destinationContractString,
        string memory destinationAddressString,
        string memory tokenUri
    ) onlyOwner external {
        address destinationAddress =
            stringToAddress(destinationAddressString);
        TokenImplementationInterface destinationContract =
            TokenImplementationInterface(
                stringToAddress(destinationContractString)
            );

        destinationContract.mint(
            destinationAddress, // recipient
            0,
            tokenUri
        );
    }
```

Figura D.15: Función `completeSend` (EVM)

Para el caso de Fabric, en las Figuras D.16 y D.17 se tienen las implementaciones de las funciones `startSend` y `completeSend`.

```
async startSend(
  ctx: Context,
  destinationBlockchain: string,
  destinationContract: string,
  destinationAddress: string,
  contract: string,
  tokenId: string
) {
  const BurnParams: BurnParams = [tokenId];

  await ctx.stub.invokeChaincode(
    contract,
    ["Burn", ...BurnParams],
    ctx.stub.getChannelID()
  );

  const TokenUriParams: TokenURIParams = [tokenId];

  const tokenUri = await ctx.stub.invokeChaincode(
    contract,
    ["TokenURI", ...TokenUriParams],
    ctx.stub.getChannelID()
  );

  if (tokenUri.status !== 200 || !tokenUri.payload) {
    throw new Error(`Failed to get token URI:
        ${tokenUri.message}`);
  }

  const eventPayload: MessagePayload = {
    destinationBlockchain,
    destinationContract,
    operation: "completeSend",
    operationParams: [
      destinationContract,
      destinationAddress,
      tokenUri.payload.toString(),
    ],
  };

  ctx.stub.setEvent("Message",
      Buffer.from(JSON.stringify(eventPayload)));
}
```

Figura D.16: Función `startSend` (Fabric)

```
async completeSend(
  ctx: Context,
  destinationAddress: string,
  tokenId: string,
  tokenUri: string,
  contract: string
): Promise<void> {
  const MintParams: MintParams = [destinationAddress,
      tokenId, tokenUri];

  await ctx.stub.invokeChaincode(
    contract,
    ["Mint", ...MintParams],
    ctx.stub.getChannelID()
  );
}
```

Figura D.17: Función `completeSend` (Fabric)

D.1.3. Transferencia Temporal

Al igual que el `Permanent Transfer Smart Contract`, en la figura D.18 se observa un diagrama de clases del Temporal Transfer. El `Temporal Transfer Smart Contract` hereda de `ERC721Holder` y `TokenImplementationInterface` como en el caso del `Permanent`. Además, debe tener el manejo de los tokens prestados, por lo cual hereda del *smart contract* `NFTBridgeState`. Para las EVM se modularizo este contrato, separando la lógica en 2 contratos adicionales: `NFTBridgeGetters` y `NFTBridgeSetters`. Estos *smart contracts* permiten centralizar toda la lógica necesaria de las transferencias temporales de NFTs. El `NFTBridgeState` provee el almacenamiento compartido que mantiene información clave como los `Wrapped Tokens`. El `NFTBridgeGetters` y `NFTBridgeSetters` proporcionan funciones para acceder y modificar el estado del bridge definido en el `NFTBridgeState`. En las Figuras D.19, D.20 y D.21 se tienen la implementación de estos *smart contracts*.

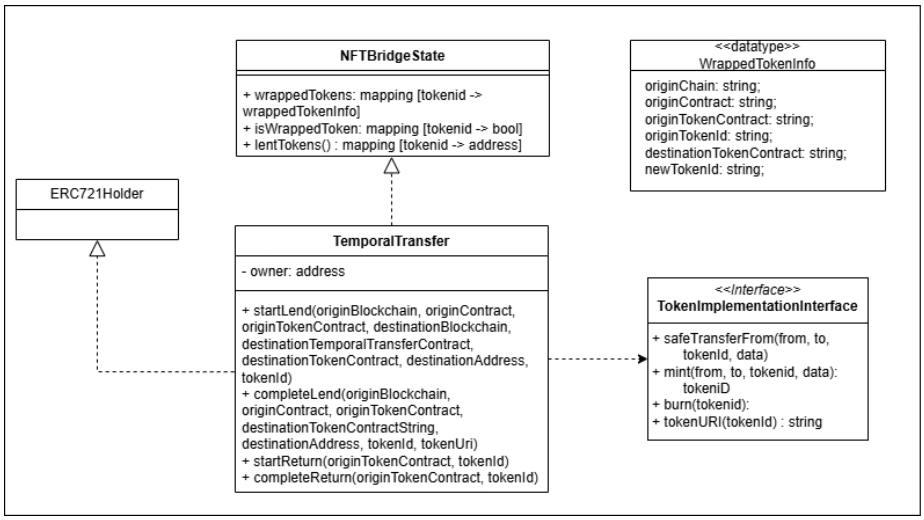


Figura D.18: Diagrama de clases del Temporal Transfer

```
contract NFTBridgeState {
    // Struct to hold information about wrapped tokens
    struct WrappedTokenInfo {
        string originChain;  // Source blockchain
        string originContract;  // Address of the bridge contract on the source blockchain
        string originTokenContract; // Address of the token on the source blockchain
        uint256 originTokenId; // ID of the token on the source blockchain
        TokenImplementationInterface destinationTokenContract;
    }

    // State variables
    TokenImplementationInterface public tokenImplementation; // Address of the NFT contract implementation
    string public chain;              // Current chain

    // Mappings to store wrapped token information
    mapping(string => mapping(uint256 => WrappedTokenInfo)) public wrappedTokens; // Maps token ID to wrapped token info
    mapping(string => mapping(uint256 => bool)) public isWrappedToken;             // Maps token ID to bool for wrapped status
    mapping(string => mapping(uint256 => address)) public lentTokens;
}
```

Figura D.19: `NFTBridgeState` (EVM)

```
import "./NFTBridgeState.sol";
import "./TokenImplementationInterface.sol";
contract NFTBridgeGetters is NFTBridgeState {

    function getChain() public view returns (string memory)
        {
        return chain;
    }

    function wrappedToken(string memory
        originTokenContract, uint256 tokenId) public view
        returns (WrappedTokenInfo memory) {
        return wrappedTokens[originTokenContract][tokenId];
    }

    function checkIsWrappedToken(string memory
        originTokenContract, uint256 tokenId) public view
        returns (bool) {
        return isWrappedToken[originTokenContract][tokenId];
    }

}
```

Figura D.20: `NFTBridgeGetters` (EVM)

```
import "./NFTBridgeState.sol";
import "./TokenImplementationInterface.sol";

contract NFTBridgeSetters is NFTBridgeState {

    function setWrappedToken(uint256 tokenId, string memory
        originChain, string memory originContract, string
        memory originTokenContract, uint256 originTokenId,
        TokenImplementationInterface
        destinationTokenContract) internal {
        wrappedTokens[originTokenContract][tokenId] =
            WrappedTokenInfo({
            originChain: originChain,
            originContract: originContract,
            originTokenContract: originTokenContract,
            originTokenId: originTokenId,
            destinationTokenContract:
                destinationTokenContract
        });
    }

    function setIsWrappedToken(string memory
        originTokenContract, uint256 tokenId, bool
        isWrapped) internal {
        isWrappedToken[originTokenContract][tokenId] =
            isWrapped;
    }

}
```

Figura D.21: `NFTBridgeSetters` (EVM)

Para Fabric, se tiene un único `NFTBridgeState Smart Contract`. En las Figuras se pueden ver las funciones equivalentes de manejo de `Wrapped Tokens` como se puede ver en la Figura D.22

```
type WrappedTokenInfo = {
  originChain: string;
  originContract: string;
  originTokenContract: string;
  originTokenId: string;
  destinationTokenContract: string;
  newTokenId: string;
};

interface NFTBridgeStateData {
  wrappedTokens: { [key: string]: WrappedTokenInfo };
  isWrappedToken: { [key: string]: boolean };
  lentTokens: { [key: string]: string };
}
```

Figura D.22: `NFTBridgeState` (Fabric)

```
async setWrappedToken(ctx: Context, key: string, token:
    WrappedTokenInfo) {
  const state = await this.getState(ctx);
  state.wrappedTokens[key] = token;
  await this.updateState(ctx, state);
}

async getWrappedToken(ctx: Context, key: string):
    Promise<WrappedTokenInfo> {
  const state = await this.getState(ctx);
  return state.wrappedTokens[key];
}

async checkIsWrappedToken(ctx: Context, key: string):
    Promise<boolean> {
  const state = await this.getState(ctx);
  return state.isWrappedToken[key] ?? false;
}
```

Figura D.23: `Set` y `Get` de `Wrapped Tokens` (Fabric)

Al inicializar la transferencia temporal, primero se llama a la función `startLend` (Figura D.24).

```
function startLend(
    string memory originBlockchain,
    string memory originContract,
    string memory originTokenContractString,
    string memory destinationBlockchain,
    string memory destinationContract,
    string memory destinationTokenContractString,
    string memory destinationAddress,
    uint256 tokenId
) external payable {
    require(msg.value >= 0.005 ether, "Insufficient fee
        sent");
    TokenImplementationInterface originTokenContract =
        TokenImplementationInterface(
    stringToAddress(originTokenContractString)
    );
    require(lentTokens[originTokenContract][tokenId] ==
        address(0), "Token is already lent");
    originTokenContract.safeTransferFrom(msg.sender,
        address(this), tokenId,"");
    string memory tokenUri =
        originTokenContract.tokenURI(tokenId);
    lentTokens[originTokenContract][tokenId] = msg.sender;
    emit Message(
        destinationBlockchain,
        destinationContract,
        "completeLend",
        abi.encode(
            originBlockchain,
            originContract,
            originTokenContractString,
            destinationTokenContractString,
            destinationAddress,
            tokenId,
            tokenUri
        )
    );
}
```

Figura D.24: `startLend` (EVM)

Una vez transferido el NFT al `Temporal Transfer Smart Contract`, este queda en posesión del contrato, lo cual actua como el *lock* del NFT. Como para el caso de la transferencia permanente, esto emite un evento `Message`, el cual es escuchado por el relayer.

Para Fabric, se tiene la implementación de `startLend` como se ve en la Figura D.25.

```
async startLend(
    ctx: Context,
    originBlockchain: string,
    originContract: string,
    originTokenContract: string,
    destinationBlockchain: string,
    destinationContract: string,
    destinationUser: string,
    tokenId: string
  ): Promise<void> {
    const currentUser = this.getCurrentUserFromContext(ctx);

    const TransferFromParams: TransferFromParams = [
      currentUser,
      destinationUser,
      tokenId,
    ];

    const tokenUri = await ctx.stub.invokeChaincode(
      originTokenContract,
      ["TokenURI", tokenId],
      ctx.stub.getChannelID()
    );

    await ctx.stub.invokeChaincode(
      originTokenContract,
      ["TransferFrom", ...TransferFromParams],
      ctx.stub.getChannelID()
    );

    await this.setLentToken(ctx, tokenId,
        originTokenContract);
    const eventPayload = {
      destinationBlockchain,
      destinationContract,
      operation: "completeLend",
      operationParams: [
        originBlockchain,
        originContract,
        destinationContract,
        destinationUser,
        tokenId,
        tokenUri,
      ],
    };

    ctx.stub.setEvent("Message",
        Buffer.from(JSON.stringify(eventPayload)));
}
```

Figura D.25: `startLend` (Fabric)

Luego, para recibir el préstamo se invoca a la función `completeLend` (Figura D.26).

```
function completeLend(
        string memory originBlockchain,
        string memory originContract,
        string memory originTokenContract,
        string memory destinationTokenContractString,
        string memory destinationAddress,
        uint256 tokenId,
        string memory tokenUri
    ) onlyOwner external {
            TokenImplementationInterface
                destinationTokenContract =
                TokenImplementationInterface(
            stringToAddress(destinationTokenContractString)
            );
            uint256 newTokenId =
                destinationTokenContract.mint(
                stringToAddress(destinationAddress), //
                    recipient
                0,
                tokenUri
            );

            setWrappedToken(
                newTokenId,
                originBlockchain,
                originContract,
                originTokenContract,
                tokenId,
                destinationTokenContract,
                newTokenId
            );

            // // Mark the token as wrapped
            setIsWrappedToken(destinationTokenContract,
                newTokenId, true);
}
```

Figura D.26: `completeLend` (EVM)

En esta función se realiza el *mint* del `Wrapped Token` que esta siendo prestado, asignándoselo a la billetera destino. Además, se guarda la información de este `Wrapped Token` en el estado del bridge (`setWrappedToken`).

En Fabric, la función `completeLend` se implementó como se puede ver en la Figura D.27.

```
async completeLend(
    ctx: Context,
    originBlockchain: string,
    originContract: string,
    originTokenContract: string,
    destinationTokenContract: string,
    destinationAddress: string,
    tokenId: string,
    tokenUri: string
  ) {
    const MintParams: MintParams = [destinationAddress,
        tokenId, tokenUri];

    const response = await ctx.stub.invokeChaincode(
      destinationTokenContract,
      ["Mint", ...MintParams],
      ctx.stub.getChannelID()
    );

    if (response.status !== 200 || !response.payload) {
      throw new Error(`Failed to mint:
          ${response.message}`);
    }

    const newTokenId = response.payload.toString();

    await this.setWrappedToken(ctx, {
      originChain: originBlockchain,
      originContract,
      originTokenContract,
      originTokenId: tokenId,
      destinationTokenContract,
      newTokenId,
    });

    return newTokenId;
}
```

Figura D.27: `completeLend` (Fabric)

Para iniciar la devolución del `asset`, se invoca a la función `startReturn` (Figura D.28).

```
function startReturn(TokenImplementationInterface
    destinationTokenContract, uint256 newTokenId) external
    payable {
       require(msg.value >= 0.005 ether, "Insufficient fee
           sent");
       WrappedTokenInfo memory info =
           wrappedToken(destinationTokenContract, newTokenId);
       destinationTokenContract.safeTransferFrom(msg.sender,
           address(this), newTokenId, "");
       destinationTokenContract.burn(newTokenId);
       emit Message(
            info.originChain,
            info.originContract,
            "completeReturn",
            abi.encode(
                 info.originTokenContract,
                 info.originTokenId
            )
       );

       delete
           wrappedTokens[destinationTokenContract][newTokenId];
       setIsWrappedToken(destinationTokenContract,
           newTokenId, false);
}
```

Figura D.28: `startReturn` (EVM)

En esta función, se realiza el *burn* del `Wrapped Token`, y se emite el evento `Message` que sera escuchado por el relayer. Para Fabric, se tiene el `startReturn` de la Figura D.29.

```
async startReturn(
    ctx: Context,
    destinationTokenContract: string,
    newTokenId: string
  ): Promise<boolean> {
    const wrappedTokenInfo = JSON.parse(
      await this.getWrappedToken(ctx,
          destinationTokenContract, newTokenId)
    ) as WrappedTokenInfo;
    await this.burnFromContract(
      ctx,
      wrappedTokenInfo.destinationTokenContract,
      newTokenId
    );

    const eventPayload = {
      destinationBlockchain: wrappedTokenInfo.originChain,
      destinationContract: wrappedTokenInfo.originContract,
      operation: "completeReturn",
      operationParams: [
        wrappedTokenInfo.originTokenContract,
        wrappedTokenInfo.originTokenId,
      ],
    };
    ctx.stub.setEvent("Message",
        Buffer.from(JSON.stringify(eventPayload)));

    await this.deleteWrappedToken(
      ctx,
      destinationTokenContract,
      wrappedTokenInfo.newTokenId
    );

    return true;
}
```

Figura D.29: `startReturn` (Fabric)

Finalmente, para recibir la devolución del `asset` se llama a la función `completeReturn` (Figura D.30). En esta función se realiza una transferencia del NFT original a su dueño original. Esto funciona como el `unlock` del NFT, al estar siendo devuelto a la propiedad del usuario.

```
function completeReturn(
        TokenImplementationInterface originTokenContract,
        uint256 tokenId
    ) onlyOwner external {
        address user =
            lentTokens[originTokenContract][tokenId];
        originTokenContract.safeTransferFrom(address(this),
            user, tokenId, "");
        delete lentTokens[originTokenContract][tokenId];
}
```

Figura D.30: `completeReturn` (EVM)

En la Figura D.31 se tiene la función `completeReturn` para Fabric.

```
async completeReturn(
    ctx: Context,
    originTokenContract: string,
    tokenId: string
  ): Promise<void> {
    const user = await this.getLentToken(ctx,
        originTokenContract, tokenId);

    const TransferFromParams: TransferFromParams =
        ["Admin", user, tokenId];

    await ctx.stub.invokeChaincode(
      originTokenContract,
      ["TransferFrom", ...TransferFromParams],
      ctx.stub.getChannelID()
    );

    await this.deleteLentToken(ctx, tokenId,
        originTokenContract);
}
```

Figura D.31: `completeReturn` (Fabric)

D.1.4. Despliegue de los contratos

Dentro de la solución de la EVM se encuentran varios contratos que hay que desplegar a la red. Estos son `MusicDiscs`, `PermanentTransfer` y `TemporalTransfer`. `MusicDiscs` debe saber cuales son los otros dos contratos para otorgarles acceso a las funciones restringidas que se mencionaron anteriorment y para esto debe saber sus direcciones. Teniendo en cuenta este requerimiento, se optó por hacer el despliegue en dos fases, en donde en la primera fase se despliegan el `PermanentTransfer` y `TemporalTransfer` y luego en la segunda, cuando ya se

sabe las direcciones de estos dos, se despliega `MusicDiscs` pasando estas direcciones cómo parámetro.

En el caso de Fabric, para poder llamar a un contrato simplemente se tiene que saber el nombre de este, por lo tanto no fue necesario definir ninguna estrategia y no hace falta más que desplegar los contratos.