UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TRABAJO DE FIN DE GRADO

Grado en Ingeniería Electrónica Industrial y Automática

MODELO DE PROGRAMACIÓN EN SOLANA BLOCKCHAIN

Memoria y Anexos

Autor: Carlos Martin León

Director: Javier Farreres de la Morena

Convocatoria: Septiembre 2023

Resum

Des de l'aparició del primer bloc de gènesi de Bitcoin, la tecnologia Blockchain ha experimentat un creixement exponencial. Inicialment dissenyada per transaccions peer-to-peer, la seva trajectòria l'ha orientat a ser percebuda com una màquina global d'estats.

Aquest treball se centra en Solana Blockchain, una cadena de blocs pública avantguardista, destacant la seva visió per esdevenir la màquina global d'estats més adaptable i eficaç d'avui dia. S'aprofundirà en el seu model de programació per a Contractes Intel·ligents i la gestió de dades. Per contextualitzar, es revisarà l'evolució de la tecnologia Blockchain, abordant els reptes que ha superat. Després, es detallarà l'estructura de Solana, posant especial atenció als seus conceptes, aplicacions i eines open-source, que potencien la creació de Contractes Intel·ligents i amplien les opcions de personalització.

Resumen

Desde el surgimiento del primer bloque de génesis de Bitcoin, la tecnología Blockchain ha experimentado un desarrollo ininterrumpido. Inicialmente concebida para transacciones peer-to-peer, su evolución la ha llevado a ser vista como una máquina global de estados.

Este trabajo se centra en Solana Blockchain, una innovadora cadena de bloques pública, destacando su diseño orientado a ser la máquina global de estados más escalable y eficaz del momento. Se profundizará en su modelo de programación para Contratos Inteligentes y gestión de datos en cuentas. Para contextualizar, se repasará la evolución de la tecnología Blockchain, abordando los desafíos que ha enfrentado. Posteriormente, se desglosará la arquitectura de Solana, poniendo especial énfasis en sus conceptos, aplicaciones y herramientas open-source, las cuales potencian la creación de Contratos Inteligentes y amplían las opciones de personalización.

Abstract

Since the emergence of Bitcoin's first genesis block, Blockchain technology has undergone relentless progression. Initially conceived for peer-to-peer transactions, its trajectory has steered it towards being recognized as a global state machine.

This study delves into Solana Blockchain, a cutting-edge public Blockchain, highlighting its ambition to be today's most adaptable and effective global state machine. We'll explore its unique programming model tailored for Smart Contracts. To provide context, we'll trace the evolution of Blockchain technology, addressing the challenges it has encountered. Subsequently, we'll dissect Solana's architecture, accentuating its concepts, applications, and open-source tools that empower the creation of Smart Contracts and broaden customization options.

Glosario

Dentro del mundo de la Blockchain, la terminología puede ser compleja.

A continuación, un glosario para clarificar estos términos esenciales:

- **Blockhash:**En Solana, un blockhash es una referencia a un bloque específico en la cadena y se utiliza como una forma de "marca temporal" para las transacciones.
- **Bump:** Byte adicional para asegurar que el PDA no se encuentre en la Curva Elíptica ed2559.
- **CeFi (Centralized Finance):** Servicios financieros tradicionales ofrecidos por instituciones centralizadas, como bancos y corredores.
- **Command-Line-Interface(CLI):** Es una interfaz de usuario que permite interactuar con sistemas informáticos utilizando líneas de texto.
- **CPI (Cross-Program Invocation):** Técnica en Solana que permite a un programa invocar a otro programa, facilitando la interacción y cooperación entre diferentes programas en la red.
- **DAO (Decentralized Autonomous Organization):** Organización que opera basada en reglas predefinidas codificadas como contratos inteligentes en una blockchain.
- **DeFi (Decentralized Finance):** Sistema financiero que opera sin intermediarios, utilizando blockchains y contratos inteligentes. Busca democratizar el acceso a servicios financieros y eliminar barreras.
- **DEX (Decentralized Exchange):** Plataforma de intercambio que opera sin un intermediario central, permitiendo a los usuarios comerciar criptomonedas directamente entre sí.

- **DePIN (Decentralized Physical Infrastructure Network):** Red de Infraestructuras Físicas Descentralizadas. Representa una estructura que opera sin un punto central de control, permitiendo una distribución más equitativa de los recursos y servicios.
- **Gas:** En Ethereum, el gas es un mecanismo de tarificación y seguridad. Cada acción en la Máquina Virtual de Ethereum (EVM) tiene un costo en gas. Este sistema previene ataques de denegación de servicio al requerir que los usuarios especifiquen un límite de gas por transacción. Además, los usuarios determinan cuánto están dispuestos a pagar en ether (ETH) por unidad de gas para la ejecución de sus transacciones.

- **Mintear:** Proceso de creación de nuevos tokens dentro de una blockchain. En el contexto de tokens, especialmente en el caso de tokens no fungibles (NFTs), "mintear" se refiere a la acción de generar y registrar un nuevo token en la cadena de bloques.

- **Gulf Stream:** Protocolo de Solana que permite a los validadores ejecutar transacciones en paralelo antes de que se confirme el orden de los bloques, lo que mejora la velocidad y eficiencia de la confirmación de transacciones en la red.

- **Hash:** Función criptográfica que convierte una entrada (o 'mensaje') en una cadena de caracteres de longitud fija, que suele ser única para cada entrada distinta. Es fundamental en blockchain para garantizar integridad y autenticidad de los datos.

- **Hashcash:** Sistema de prueba de trabajo (PoW) utilizado en Bitcoin. Los mineros deben realizar cálculos computacionales para encontrar un valor hash que cumpla con ciertos criterios, permitiendo así la adición de un nuevo bloque a la blockchain.

- **Hashrate:** Medida de la potencia de procesamiento de la red blockchain, indicando cuántos cálculos hash puede realizar una máquina en un segundo. Es un indicador clave de la seguridad de una red de blockchain.

- **L1 (Layer-1):** Blockchain o protocolo de base que funciona como la capa principal. Ejemplos notables incluyen Bitcoin y Ethereum.

- **L2 (Layer-2):** Soluciones construidas sobre L1 para mejorar la escalabilidad y eficiencia, sin comprometer la seguridad. Ejemplos incluyen Optimism, Base y Lightning Network.

- **Lamports:** Unidad mínima posible en Solana blockchain. 1 Lamport = 10^{-9} SOL

- **Ledger:** Registro digital inmutable que contiene todas las transacciones que han tenido lugar en una red blockchain desde su inicio.

- **Liquid Staking:** Variante del staking tradicional que permite a los usuarios obtener un activo líquido en representación de su participación bloqueada, mejorando la liquidez del activo subyacente.

- **Mempool:** Conjunto de transacciones no confirmadas en una red blockchain, esperando ser incluidas en un bloque por los mineros.

- **NFT (Non-Fungible Token):** Token criptográfico único que representa un objeto o activo digital específico. A diferencia de las criptomonedas tradicionales, cada NFT es distintivo y no puede ser intercambiado en igualdad de condiciones.

- **Nodo:**Un nodo es una entidad en la red que mantiene una copia del ledger y, dependiendo de su tipo, puede validar y retransmitir transacciones o añadir nuevos bloques al ledger. En Solana, hay diferentes tipos de nodos, como validadores y nodos de servicio.

- **PDA (Program Derived Address):** En Solana, es una dirección única generada a partir de un programa principal utilizando seeds y un bump, actuando como una cuenta especial.

- **Peer-to-peer (P2P):** Modelo de red descentralizada donde los participantes (peers) interactúan directamente entre sí, eliminando la necesidad de intermediarios. Este modelo promueve la autonomía y la distribución equitativa de recursos.

- **Proof of History (PoH):**Mecanismo que permite a una blockchain crear un registro histórico que prueba que un evento ocurrió en un momento específico, sin requerir una prueba de trabajo tradicional.

- **Proof of Stake (PoS):** Mecanismo de consenso donde los participantes (validadores) proponen y validan bloques en función de la cantidad de monedas que tienen en garantía.

- **Proof of Work (PoW):** Mecanismo de consenso en el que los participantes (mineros) resuelven problemas matemáticos complejos para validar transacciones y crear nuevos bloques en la cadena.

- **Sealevel:** En Solana Blockchain, Sealevel es un motor de ejecución paralela que permite el procesamiento simultáneo de transacciones y contratos inteligentes, mejorando la eficiencia y velocidad de la red.

- **Seeds:** Valores o números generados por software que se derivan de uno o más valores iniciales, utilizados comúnmente en procesos criptográficos.

- **Smart Contracts:** Contratos autoejecutables con las reglas del acuerdo entre dos partes directamente escritas en líneas de código. Estos programas se ejecutan en la tecnología Blockchain, garantizando la inmutabilidad y transparencia de las operaciones.

- **SOL:** Criptomoneda nativa de la blockchain de Solana, utilizada para realizar transacciones y pagar comisiones en la red.

- **Solana Programs:** Denominación específica utilizada para referirse a los Contratos Inteligentes (Smart Contracts) dentro de la red blockchain de Solana.

- **Slashing:** Proceso en el que una parte de las criptomonedas depositadas por un validador en una red blockchain se toma como penalización por comportamientos maliciosos o no conformes.

- **Staking:** Proceso de participación activa en mecanismos de prueba de participación (PoS) en las blockchains, bloqueando criptomonedas en una cartera para respaldar las operaciones de una red y, a cambio, recibir recompensas.

- **TradFi (Traditional Finance):** Sistema financiero tradicional que incluye bancos, corredores y otras instituciones financieras centralizadas.

- **Turbine:** Protocolo de transmisión de bloques de Solana que divide los datos de las transacciones y los codifica para transmitirlos de manera eficiente a todos los nodos de la red, permitiendo una rápida propagación de los datos.

- **Validador:** En Solana, un validador es un nodo que participa en la confirmación y validación de transacciones y bloques en la red. Los validadores ayudan a mantener la seguridad y la integridad de la red al alcanzar un consenso sobre el estado de la blockchain.

Índice

RESUM 1

RESUMEN 2

ABSTRACT 3

GLOSARIO 5

1. INTRODUCCIÓN 13

1.1. Objetivos del trabajo 13

1.2. Alcance del trabajo 14

2. LA TECNOLOGÍA BLOCKCHAIN 15

2.1. Conceptos Fundamentales de la Blockchain 15

2.1.1. ¿Qué es una Blockchain? 15

2.1.2. Tipos de Blockchain: Públicas, privadas, consorcio e hibridas 16

2.2. Bitcoin: La primera implementación 17

2.2.1. Transacciones 17

2.2.2. Proof of Work (PoW) 18

2.2.3. Críticas al Proof of Work (PoW) 19

2.3. Ethereum y la tecnología de los Contratos Inteligentes 21

2.3.1. Introducción a los Contratos Inteligentes 21

2.3.2. Ethereum como plataforma de contratos inteligentes 22

2.3.3. Solidity 23

2.3.4. EVM: Máquina Virtual de Ethereum 23

2.3.5. El concepto de Gas: costo y eficiencia en la red 24

2.3.6. Límites de Ethereum e introducción de las Soluciones de Capa 2(L2s) 25

3. SOLANA BLOCKCHAIN 27

3.1. Introducción a Solana Blockchain 28

3.2. La red de Solana 31

3.2.1. Diseño general de la Red 32

3.2.2. Proof of History (PoH) 34

3.2.3. Modelo de consenso de la Red 38

3.3. El modelo de programación en Solana 46

3.3.1. Cuentas 46

3.3.2. Programas 49

3.3.3. Instrucciones y Transacciones 51
3.3.4. Programa de Datos personalizado mediante PDAs 55
3.3.5. Invocación entre Programas(CPI) 59
3.3.6. Estados de Compresión 61
3.3.7. Anchor Framework 63
3.3.8. Abstracción de Serialización de Anchor vs Solana Nativo 65
3.3.9. Ejemplos usando los múltiples conceptos de Solana 66
3.3.10. Seguridad en Contratos Inteligentes en Solana 73
3.4. Tokens 75
3.4.1. SPL Token Standard 76
3.4.2. Creación de un Token Fungible usando CLI 77
3.4.3. Creación de un Token No Fungible usando CLI 78
3.4.4. Token-2022 79

CONCLUSIÓN 83

BIBLIOGRAFÍA 85

ANEXO A: CONTRATOS INTELIGENTES DE LOS EJEMPLOS 89
A1. Producir mensaje en el Ledger 89
A2. Contador Básico 89
A3. Crear una Lista Blanca mediante PDAs y CPIs 90

ANEXO B: TESTING DE LOS CONTRATOS INTELIGENTES 95
B1. Producir mensaje en el Ledger 95
B2. Contador Básico 95
B3. Crear una Lista Blanca mediante PDAs y CPIs 96

ANEXO C: COMMAND LINE INTERFACE(CLI) 101

1. Introducción

En la era digital actual, las innovaciones tecnológicas no solo buscan adaptarse, sino también trascender las barreras existentes. La tecnología Blockchain emerge como un testimonio de esta aspiración, redefiniendo continuamente su alcance y potencial en un paisaje en constante evolución. A medida que esta tecnología ha avanzado, ha surgido la necesidad de plataformas más escalables, eficientes y adaptadas a una variedad de aplicaciones. Es en este contexto que Solana Blockchain se destaca, siendo una solución diferente a las propuestas por Ethereum y la mayoría de L1s, centrándose en la escalabilidad y eficiencia de la gestión de estados y la implementación de Contratos Inteligentes en la propia L1. A través de este trabajo, profundizaremos en la distintiva naturaleza de Solana, su impacto en el ámbito de las cadenas de bloques y la particularidad de su enfoque en la programación de Contratos Inteligentes.

1.1. Objetivos del trabajo

El propósito central de este trabajo es proporcionar una comprensión profunda y detallada de Solana Blockchain y su modelo de programación. Los objetivos específicos son:

- Comprender los fundamentos y principios básicos de la tecnología Blockchain y cómo se diferencia Solana de otras plataformas.
- Analizar la evolución de Blockchain desde su concepción hasta las soluciones actuales, destacando la innovación que Solana aporta al panorama.
- Investigar a fondo la arquitectura, el modelo de consenso y las características únicas de la red de Solana Blockchain.
- Profundizar en el modelo de programación característico, y sus frameworks y herramientas open-source comunes en la elaboración de Contratos Inteligentes e interacciones con la Blockchain.
- Realizar ejemplos prácticos con el modelo de programación abordando los diferentes conceptos explicados.

1.2. Alcance del trabajo

En cuanto al alcance del proyecto, se comenzará con una introducción a la tecnología Blockchain en general, seguida de una exploración detallada del funcionamiento de Solana Blockchain y su modelo de programación y datos. Se hará énfasis en las características que hacen de Solana una solución única en el espacio de Blockchain.

2. La Tecnología Blockchain

Esta sección se adentrará en los cimientos de la tecnología Blockchain, desentrañando sus conceptos fundamentales y trazando su evolución desde la creación de Bitcoin hasta el paso evolutivo con Ethereum y sus contratos inteligentes. No solo entenderemos qué es una Blockchain, sino también cómo sus características y principios han sido la base para las innovaciones que han seguido, y cómo plataformas como Ethereum y Bitcoin han creado las bases de las Blockchains modernas.

2.1. Conceptos Fundamentales de la Blockchain

2.1.1. ¿Qué es una Blockchain?

La *Blockchain*, traducida como "cadena de bloques", es una estructura de datos compuesta por bloques individuales de datos, que están intrínsecamente conectados mediante algoritmos criptográficos. Esta conexión garantiza un registro inmutable y transparente de las transacciones, reduciendo la dependencia de entidades centralizadas o intermediarios.

La idea de una cadena de bloques se originó en 1991, gracias a Stuart Haber y W. Scott Stornetta[1], quienes buscaban crear un sistema en el que las marcas temporales de los documentos no pudieran ser alteradas. Sin embargo, fue en 2008 cuando la Blockchain ganó prominencia con la aparición de *Bitcoin*, una propuesta de Satoshi Nakamoto[2]. Nakamoto presentó un sistema basado en Blockchain que permitía transacciones electrónicas directas entre dos partes sin la intervención de un tercero(*peer-to-peer*).
Esto fue un cambió en el paradigma en los sistemas de pagos, en los cuales siempre había sido necesario el permiso de bancos o gobiernos para hacer una transacción. Hoy en día, es mayormente conocido como un sistema *permissionless*[3].
Aunque el término "Blockchain" no se utilizó explícitamente en el Whitepaper de Bitcoin, la esencia de esta tecnología se encuentra intrínsecamente en él. Nakamoto esbozó la idea central detrás de la Blockchain como un concepto de un servidor de marcas temporales. En este diseño, el servidor toma un hash de un conjunto de elementos y lo difunde de manera

extensa. Estas marcas temporales validan que los datos existían en ese preciso momento para ser parte del hash. De manera crucial, cada marca temporal se construye sobre la anterior, creando una cadena enlazada. Con cada adición, la cadena se fortalece y solidifica, garantizando la integridad y la inmutabilidad de los datos. La siguiente figura muestra como vendría dada la formación de bloques mediante la concatenación de Hashes:

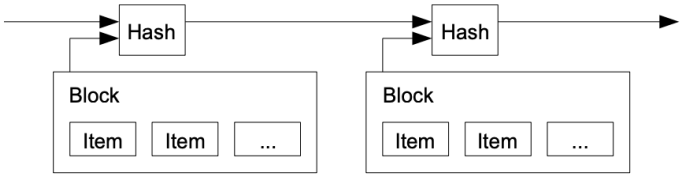

Ilustración 1. Formación de bloques mediante concatenación de hashes. Fuente: Bitcoin Whitepaper

2.1.2. Tipos de Blockchain: Públicas, privadas, consorcio e hibridas

La versatilidad de la tecnología Blockchain ha llevado a su adaptación en diversas formas, cada una diseñada para satisfacer requisitos específicos y escenarios de uso. A continuación, se detallan los principales tipos junto con ejemplos representativos:

- **Blockchain Públicas:** Estas redes son abiertas y accesibles para cualquier usuario que desee unirse. No requieren permisos previos (sin considerar capacidades en hardware o Staking mínimo).
    - *Ejemplo:* **Bitcoin** es el ejemplo más prominente de una Blockchain pública. Cualquiera puede unirse a la red, validar transacciones y participar en el proceso de minería[2].
- **Blockchain Privadas:** Operan en un entorno cerrado, generalmente dentro de una organización.
    - *Ejemplo:* **Hyperledger Fabric**[4] es una plataforma de Blockchain privada que permite a las empresas construir soluciones personalizadas. Es utilizado por empresas como IBM y Walmart para rastrear la cadena de suministro y garantizar la autenticidad de los productos.

- **Blockchain de Consorcio:** Estas redes son gestionadas por un grupo selecto de organizaciones o entidades.
    - *Ejemplo:* **R3's Corda**[5] es una plataforma de Blockchain de consorcio diseñada para el sector financiero. Varios bancos y entidades financieras colaboran en esta red para mejorar la eficiencia en transacciones y contratos.
- **Blockchain Híbridas:** Combinan elementos de Blockchains públicas y privadas.
    - *Ejemplo:* **Dragonchain**[6] es una plataforma que permite a las empresas mantener datos privados en una Blockchain privada, mientras que también pueden elegir publicar ciertos datos en una Blockchain pública para mayor transparencia.

Al considerar la adopción de una solución basada en Blockchain, es crucial entender las características y limitaciones de cada tipo. La elección adecuada dependerá del equilibrio deseado entre transparencia, eficiencia, privacidad y control.

2.2. Bitcoin: La primera implementación

Bitcoin, la primera criptomoneda basada en la tecnología Blockchain, introdujo una nueva forma de realizar transacciones electrónicas sin la necesidad de una entidad centralizada. A continuación, se detallan los conceptos clave detrás de su diseño y funcionamiento.

2.2.1. Transacciones

En el paradigma de Bitcoin, una transacción electrónica se visualiza como una serie de firmas digitales. Imagina una moneda electrónica como un documento que pasa de mano en mano. Cada vez que alguien transfiere esta "moneda", firma digitalmente un resumen criptográfico de la transacción anterior junto con la clave pública del próximo propietario, añadiendo esta información al "documento" o moneda[2].

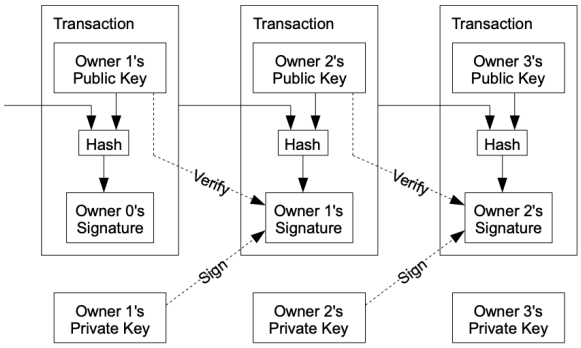

Ilustración 2. Transacciones electrónicas como una serie de firmas digitales.

Fuente: Bitcoin Whitepaper.

El receptor, o beneficiario, puede verificar estas firmas para confirmar la cadena de propiedad de la moneda. Sin embargo, surge un problema: ¿cómo puede el beneficiario estar seguro de que el propietario anterior no gastó esa misma moneda en otra transacción.

Una solución común en sistemas tradicionales es tener una autoridad central, como un banco o una casa de moneda, que verifique cada transacción para evitar el doble gasto[2]. Pero esto centraliza el sistema y lo hace dependiente de esa entidad. Bitcoin, en cambio, busca una solución descentralizada.

Para garantizar que una moneda no se gaste dos veces, es esencial tener un registro completo de todas las transacciones. En el modelo de Bitcoin, todas las transacciones se anuncian públicamente, permitiendo a todos los participantes estar al tanto de cada transacción realizada[2]. De esta manera, los participantes pueden acordar un único historial de transacciones, asegurando que cada moneda se gaste solo una vez. La clave es que la mayoría de los nodos (participantes) en la red deben estar de acuerdo en el orden de las transacciones[2].

2.2.2. Proof of Work (PoW)

El sistema de "Prueba de Trabajo" (Proof of Work) es esencial para el funcionamiento de Bitcoin. Su propósito es validar y registrar transacciones en la Blockchain de manera descentralizada. Basado en el concepto de Hashcash[7], el PoW requiere que los nodos, o

"mineros", resuelvan problemas matemáticos complejos para agregar un nuevo bloque a la cadena. Esta solución matemática, o "hash", debe cumplir ciertos criterios, como comenzar con un número específico de ceros. El esfuerzo computacional necesario para encontrar este hash es lo que garantiza la seguridad de la red.

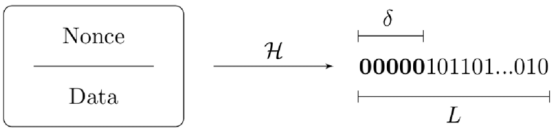

Ilustración 3. Esquema del concepto de Hashcash. Fuente: *A Survey on PoW-based Consensus*[8].

Integrando estos elementos, Nakamoto propuso el siguiente diseño:

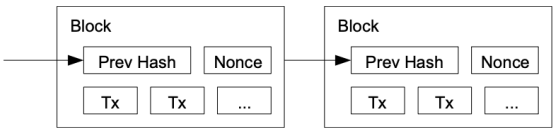

Ilustración 4. Cadena de Bloques en PoW integrando el concepto de Hashcash. Fuente: Bitcoin Whitepaper.

El PoW no solo previene ataques y comportamientos maliciosos, sino que también determina la representación en la toma de decisiones de la mayoría. En lugar de un sistema basado en "una dirección IP, un voto", el PoW opera bajo el principio de "un CPU, un voto". La cadena más larga, que representa la mayoría del poder computacional, es considerada la versión válida y correcta de la Blockchain.

2.2.3. Críticas al Proof of Work (PoW)

El sistema de Prueba de Trabajo (PoW) ha sido un pilar en el mundo de las criptomonedas, especialmente con Bitcoin liderando el camino. Sin embargo, a pesar de su importancia y eficacia en garantizar la seguridad y la integridad de las transacciones, el PoW ha sido objeto de numerosas críticas:

1. **Elevado Gasto Energético**: Una preocupación predominante es el consumo masivo de energía asociado con la minería PoW. Los mineros, en su búsqueda de resolver

problemas criptográficos, utilizan hardware potente, lo que se traduce en un consumo eléctrico que rivaliza con el de naciones enteras[9].

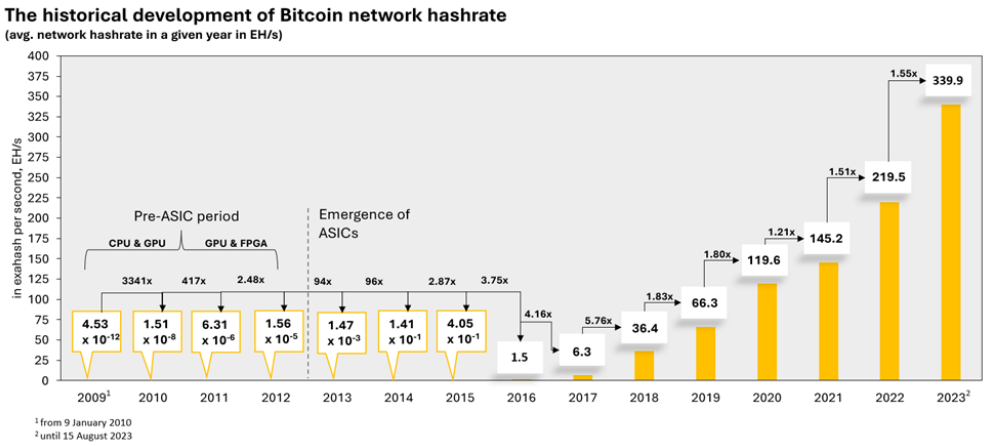

Ilustración 4. Desarrollo histórico del hashrate de la red de Bitcoin. Fuente: CoinMetrics.

2. **Tendencia hacia la Centralización**: A pesar de que la descentralización es un principio fundamental de las criptomonedas, el PoW ha visto una acumulación de poder en manos de unos pocos mineros o "piscinas" de minería. Estas entidades con grandes recursos pueden influir desproporcionadamente en la red[10].

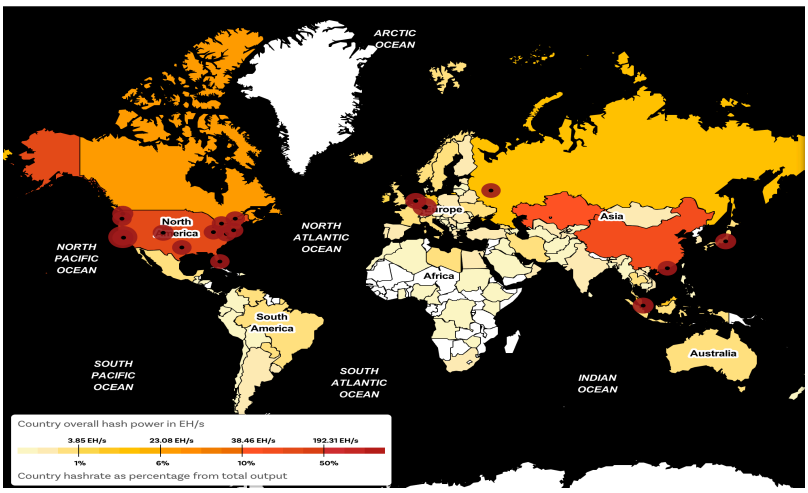

Ilustración 5. Distribución global del poder computacional en la red de Bitcoin. Fuente: ChainBulletin.

3. **Riesgo de Ataques del 51%**: En teoría, si un actor logra controlar más del 50% del poder de cómputo de la red, podría manipular o revertir transacciones, conocido

como un ataque del 51%. Aunque es poco probable en redes grandes debido al gran costo, las monedas más nuevas o menos populares son más susceptibles.

4. **Sostenibilidad a Largo Plazo**: Con la disminución de las recompensas mineras y el límite máximo de monedas que se pueden minar, surge la pregunta de qué motivará a los mineros en el futuro. Si los incentivos disminuyen, la seguridad de la red podría verse comprometida.
5. **Consecuencias Ambientales**: La minería PoW no solo consume energía, sino que también tiene un impacto ambiental debido a la producción y desecho de hardware y el uso de energías no renovables.

Estas críticas han impulsado la búsqueda de alternativas al PoW, como fue la Prueba de Participación (Proof of Stake, PoS), que promete ser más eficiente y sostenible. Ethereum, realizó la transición de PoW a PoS por motivos energéticos entre otros como veremos en la siguiente sección.

2.3. Ethereum y la tecnología de los Contratos Inteligentes

Mientras que Bitcoin presentó al mundo una nueva forma de moneda digital descentralizada, Ethereum llevó la visión de la tecnología Blockchain a un nivel completamente nuevo. No se trata simplemente de una plataforma para transacciones financieras, sino de un intento de construir una tecnología generalizada sobre la cual se pueden construir un concepto de máquinas de estado basadas en transacciones. Ethereum proporciona una infraestructura que va más allá de la simple transferencia de valor; ofrece una plataforma en la que los desarrolladores pueden construir software en un paradigma de computación hasta ahora inexplorado en el ámbito principal: un marco de computación basado en mensajería de objetos confiables[11].

2.3.1. Introducción a los Contratos Inteligentes

Los contratos inteligentes representan una transformación revolucionaria de los acuerdos entre individuales que conocemos. En lugar de depender de la confianza y la interpretación de un documento físico, un contrato inteligente es un conjunto de código que reside en

Blockchain y que opera automáticamente cuando se cumplen ciertas condiciones preestablecidas.

Los contratos inteligentes eliminan la necesidad de intermediarios. Su funcionamiento se basa en la lógica del código que haya en este, y toda interacción con este deberá someterse bajo el consenso de la red Blockchain en la que se encuentran, asegurando transparencia e inmutabilidad.

Piensa en un contrato inteligente como un sistema automatizado: introduces un dato, y si ese dato cumple con las condiciones predefinidas, el sistema realiza una acción específica. Todo esto ocurre sin intervención humana y sin lugar a duda o interpretaciones.

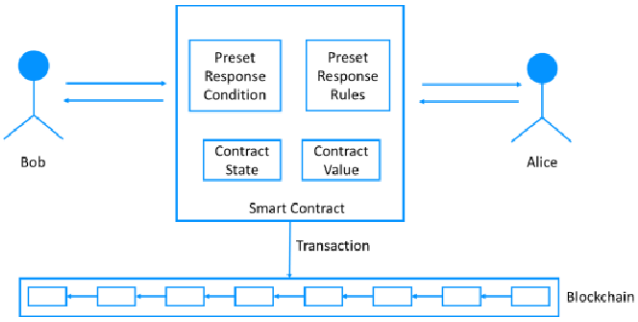

Ilustración 6. Esquema de un Contrato Inteligente. Fuente: A Medical Data Sharing Platform Based On Permissioned Blockchains[12].

Ethereum, en su esencia, ha sido pionero en proporcionar una plataforma para la creación y gestión de estos contratos, introduciendo un nuevo paradigma en la forma en que concebimos y manejamos los acuerdos y transacciones. A medida que avanzamos en esta sección, exploraremos más a fondo cómo Ethereum ha facilitado el auge de los contratos inteligentes.

2.3.2. Ethereum como plataforma de contratos inteligentes

Ethereum es la Blockchain más común para la creación y gestión de contratos inteligentes. A diferencia de otras Blockchains que se centraron en las transacciones (como Bitcoin), Ethereum se diseñó con la flexibilidad en mente. Esta flexibilidad ha llevado a una gran innovación en el espacio Blockchain, con miles de DApps construidas en Ethereum que

abarcan desde simples sistemas de votación descentralizados hasta sistemas financieros construidos en Blockchain (DeFi).

2.3.3. Solidity

Solidity se ha establecido como el lenguaje de programación predilecto para esculpir contratos inteligentes en Ethereum. A través de Solidity, los desarrolladores pueden plasmar lógicas complejas, reglas y funciones que se traducen en contratos robustos. Una vez codificado, este contrato se traduce a un formato que la EVM puede entender y ejecutar. En el siguiente apartado se verá en más detalle que sucede con este código Solidity.

2.3.4. EVM: Máquina Virtual de Ethereum

La Máquina Virtual de Ethereum (EVM) actúa como el corazón computacional de Ethereum, permitiendo la ejecución de contratos inteligentes en la red. Cuando un contrato es compilado, la EVM transforma el código en opcodes, que son instrucciones básicas que la máquina puede ejecutar. En el siguiente esquema se puede observar su funcionamiento:

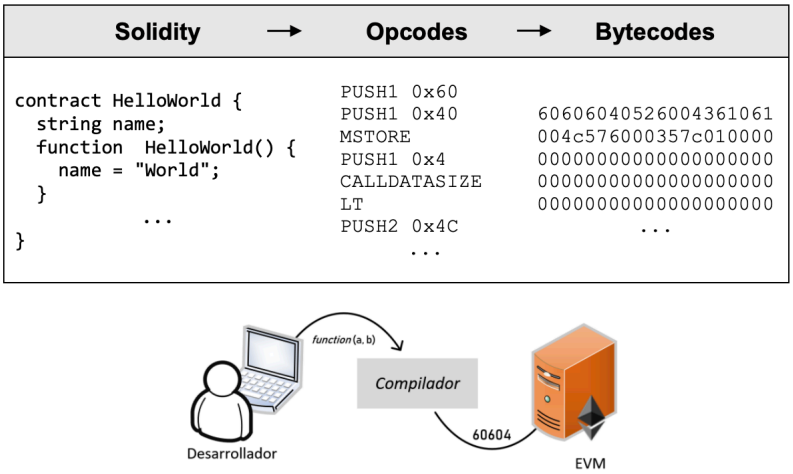

Ilustración 7. Esquema del funcionamiento de la EVM. Fuente: Explorando la Blockchain de Ethereum y el desarrollo de smart contracts.

Estos opcodes se almacenan eficientemente como bytecodes. Cada opcode tiene un costo asociado en gas, lo que implica que los desarrolladores deben ser conscientes de la optimización para minimizar los costos al interactuar con contratos.

Stack	Name	Gas	Initial Stack
00	STOP	0	
01	ADD	3	a, b
02	MUL	5	a, b
03	SUB	3	a, b
04	DIV	5	a, b
05	SDIV	5	a, b
06	MOD	5	a, b
07	SMOD	5	a, b

Ilustración 8. Siete Primeros elementos de la tabla de opcodes de Ethereum.

Fuente: Ethereum.org

Es a través de Solidity que los desarrolladores codifican la lógica y las funciones que luego son transformadas por la EVM en opcodes.

2.3.5. El concepto de Gas: costo y eficiencia en la red

En la Blockchain de Ethereum, el gas no solo es un mecanismo de tarificación, sino también una solución de seguridad y eficiencia. Dado que los nodos que ejecutan la Máquina Virtual de Ethereum (EVM) no pueden predecir de antemano los recursos necesarios para validar una transacción, existe un riesgo inherente de ataques de denegación de servicio. Para mitigar este riesgo, se introduce el concepto de gas.

Cada operación o interacción en la EVM tiene un costo asociado en unidades de gas. Cuando un usuario inicia una transacción, debe especificar un límite de gas, que es la cantidad máxima de gas que está dispuesto a consumir para esa operación. Además, el usuario define un precio por unidad de gas en ether(ETH), reflejando cuánto está dispuesto a pagar en ether por la ejecución de la transacción.

Si una operación consume más gas del establecido como límite, la transacción se revierte, pero el gas gastado no se reembolsa, garantizando así que los usuarios sean cautelosos y precisos al establecer sus límites y precios de gas.

2.3.6. Límites de Ethereum e introducción de las Soluciones de Capa 2(L2s)

A pesar de la revolución que Ethereum ha traído al mundo de la Blockchain, no está exento de desafíos. Como toda tecnología emergente, Ethereum se enfrenta a limitaciones que han llevado a la comunidad a tener múltiples obstáculos en la adopción.

Limitaciones de Ethereum:

1. **Escalabilidad:** Una de las críticas más frecuentes a Ethereum es su capacidad limitada para procesar transacciones simultáneamente. A medida que la red se ha popularizado, la congestión ha llevado a tiempos de espera más largos y tarifas más altas. Es por ello que otras Blockchain orientadas a escalabilidad no tienen Mempool, es decir, pueden proceder mediante paralelización y por ello no deben realizar largas colas.
2. **Costo:** Las tarifas de gas, aunque sirven como un mecanismo para regular la actividad en la red, son fácilmente disparadas durante períodos de alta demanda, haciendo que las transacciones y la interacción con la Blockchain sean muy costosas.

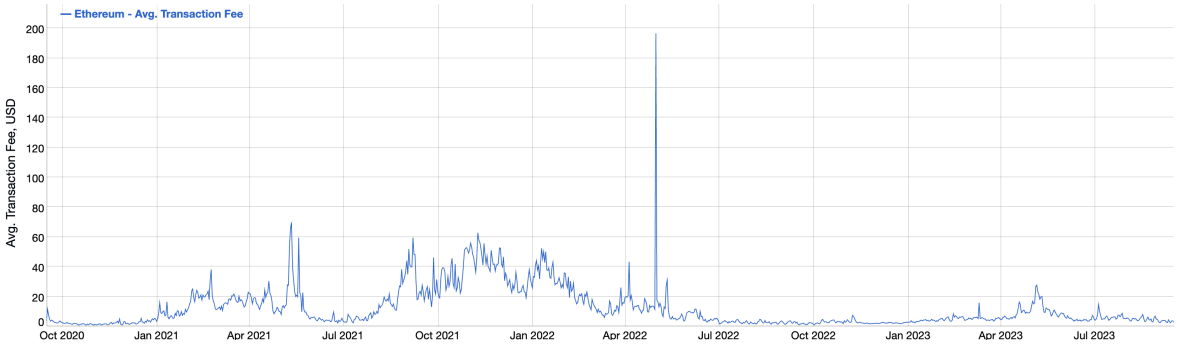

Ilustración 9. Coste promedio diario del gas en Ethereum en USD. Fuente: BitInfoCharts.

Tal y como se puede observar en la *Ilustración 9*, durante 2021 y 2022 hubo un muy alto coste de transacción en la red, llegando a picos de 200 USD simplemente por realizar una

transacción. A partir de mediados de 2022 el coste de gas disminuyo, pero esto fue debido a la bajada de tráfico e interés en la red, no por ninguna implementación de escalabilidad.

3. **Latencia:** La confirmación de transacciones en Ethereum puede tardar varios minutos, lo que no es ideal para aplicaciones que requieren respuestas en tiempo real.
4. **Consumo de Energía(resuelta):** La prueba de trabajo (PoW) fue ampliamente en fue ampliamente criticada por la comunidad por su uso extensivo de energía. Es por ello, que en Septiembre de 2022 se realizó la transición a Ethereum 2.0, pasando de PoW a PoS.

3. Solana Blockchain

A medida que avanzamos en nuestra exploración de la tecnología blockchain, hemos sido testigos de su transformación desde una plataforma de transacciones peer-to-peer hasta convertirse en una "máquina global de estados", donde cualquier individuo tiene la capacidad de registrar y almacenar información de manera descentralizada. Ethereum, la plataforma tradicional de este cambio, experimentó un auge en su adopción. Sin embargo, esta popularidad vino con su propio conjunto de desafíos. La escalabilidad de Ethereum se vio comprometida, resultando en tarifas de gas exorbitantes y tiempos de transacción lentos, especialmente durante picos de alta demanda. Para muchos usuarios, esto hizo que Ethereum fuera poco práctico para transacciones diarias.

Este dilema no es exclusivo de Ethereum. En la esfera Blockchain, este desafío se conoce comúnmente como "El Trilema de la Blockchain". Es una especie de equilibrio precario entre tres pilares fundamentales: escalabilidad, seguridad y descentralización. La creencia predominante es que, en su estado actual, es difícil, si no imposible, lograr un equilibrio perfecto entre estos tres aspectos[14]. Mientras una Blockchain puede ser rápida (escalable) y segura, puede no ser completamente descentralizada, y viceversa.

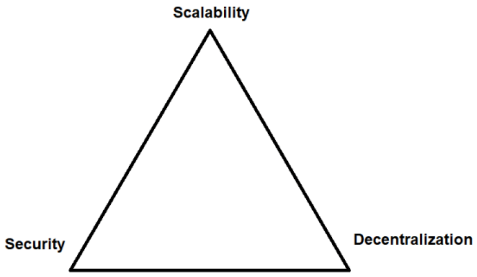

Ilustración 10. Blockchain Trilemma. Fuente: Improving Usability of Blockchain-Based Decentralized Applications.

Es en este contexto que surge Solana Blockchain, una solución innovadora a los desafíos persistentes del trilemma. Solana se presenta como una solución revolucionaria, diseñada para ser una máquina de estados global accesible para todos, sin que los usuarios tengan

que lidiar con las fluctuantes y a veces prohibitivas tarifas de gas. Su propuesta es ofrecer una plataforma que combine velocidad, seguridad y descentralización sin comprometer ninguno de estos pilares.

Solana se ha centrado en abordar las limitaciones inherentes a las arquitecturas Blockchain tradicionales, utilizando tecnologías y enfoques innovadores para garantizar transacciones rápidas y costos mínimos. Su visión es clara: democratizar el acceso a la tecnología Blockchain, permitiendo que tanto desarrolladores como usuarios finales experimenten con aplicaciones descentralizadas sin las barreras económicas y tiempos de demora que han caracterizado a otras redes. En los siguientes apartados, nos adentraremos a fondo en cómo funciona su tecnología y el diseño de su modelo de programación además de explorar muchos conceptos interesantes que caracterizan a esta red.

3.1. Introducción a Solana Blockchain

Solana se ha presentado y denominado como la solución que resuelve el dilema de la escalabilidad sin sacrificar ni seguridad ni descentralización tal y como declara Anatoly Yakovenko[15].

Las ventajas técnicas que presenta Solana Blockchain respecto al resto se pueden simplificar principalmente en tres: alto rendimiento de transacciones por segundo, rápido tiempo de confirmación y bajas tarifas por transacción. Todo ello mientras se mantiene consenso en una red de más de 2500 validadores.

En el reciente análisis llevado a cabo por VISA[16], Solana no solo destacó, sino que eclipsó a sus rivales en cuanto a rendimiento. Esta Blockchain se está posicionando rápidamente como una de las más avanzadas y eficaces en la actualidad. Con un tiempo de confirmación y generación de bloques de apenas 400 microsegundos, Solana no solo es veloz, sino que su velocidad supera en 5 veces al competidor más cercano y es unas impresionantes 450 veces más rápida que Ethereum.

Blockchain	Blocks until confirmation*	Approximate time
Solana (USDC)	1	~0.4 seconds
Avalance (USDC, EUROC)	1	~2 seconds
Flow (USDC)	1	~2.5 seconds
Algorand (USDC)	1	~5 seconds
Stellar (USDC)	1	~5 seconds
TRON (USDC)	19	~1 minute
Arbitrum (USDC)	300	~3 minutes
Ethereum (USDC, EUROC, ETH)	12	~3 minutes
Polygon PoS (Bridged USDC)	372	~20 minutes

Ilustración 11. Tiempos de confirmación de bloques. Fuente: VISA

En el mencionado estudio, se destaca que Solana posee una capacidad teórica de más de 65,000 transacciones por segundo. Aunque en la práctica la demanda suele ser menor, actualmente se procesan alrededor de 4,400 transacciones por segundo. De estas, es notable que el 90% se dedica a la comunicación entre validadores, mientras que solo el 10% corresponde a transacciones genuinas realizadas por usuarios. En la siguiente ilustración podemos observar el comportamiento en tiempo real de la red:

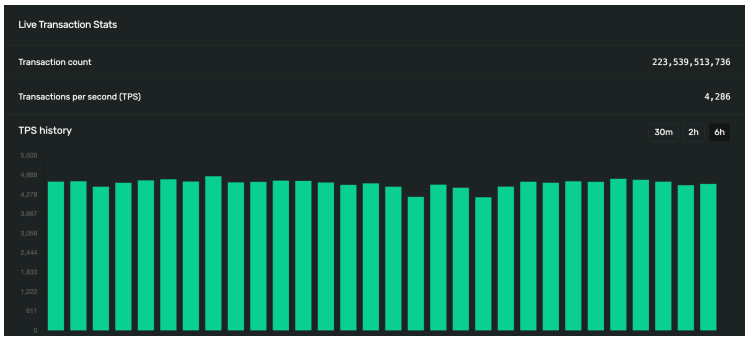

Ilustración 12. Transacciones por segundo en tiempo real. Fuente: Solana Explorer.

En el ámbito de las tarifas por transacción, Solana se erige como un pionero al ofrecer un costo promedio por transacción que es notablemente inferior al de muchas otras Blockchains. Esta eficiencia económica es una de las características distintivas de Solana, especialmente cuando se compara con gigantes del sector como Bitcoin y Ethereum. La Ilustración 13 ilustra claramente esta diferencia: mientras que las tarifas de Bitcoin y Ethereum tienden a fluctuar, influenciadas por diversos factores del mercado y la congestión de la red, Solana ha logrado mantener una estructura de tarifas estable y

consistentemente baja. Esta estabilidad no solo beneficia a los usuarios al reducir los costos de transacción, sino que también refleja la eficiencia y optimización subyacente de la red Solana.

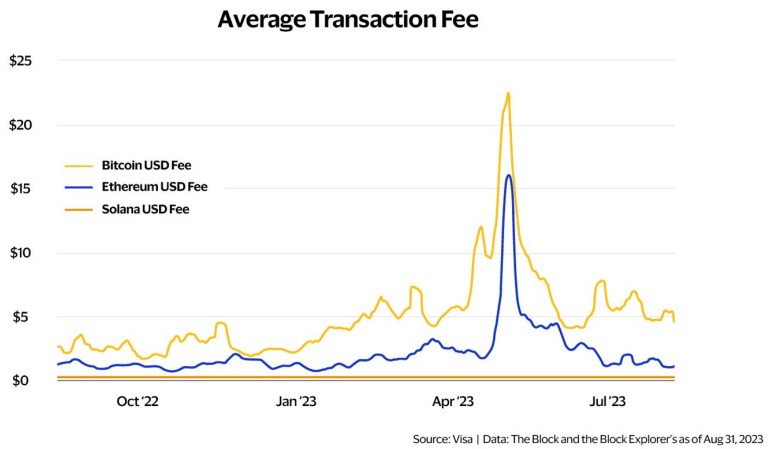

Ilustración 13. Comparativa del costo promedio de transacción en USD entre Bitcoin, Ethereum y Solana desde el tercer trimestre de 2022 hasta el tercer trimestre de 2023. Fuente: VISA

Además, el diseño intrínseco de Solana ha sido meticulosamente concebido para abordar uno de los problemas más persistentes en las Blockchains: la congestión. A diferencia de otras redes donde un aumento en la demanda puede ralentizar toda la plataforma, Solana ha implementado un sistema donde las congestiones son inherentemente locales. Esto significa que si un programa específico experimenta un pico de demanda, la congestión se limita solo a las cuentas relacionadas con ese programa, sin afectar el rendimiento general de la red. Es como si en una ciudad, un evento popular causara tráfico en una calle específica, pero las autopistas y otras rutas principales permanecieran fluidas. Esta característica de aislamiento no solo mejora la eficiencia y velocidad de Solana, sino que también garantiza una experiencia de usuario más estable y predecible, independientemente de las fluctuaciones en la demanda de aplicaciones individuales.

En la *ilustración 14* podemos ver en más detalle el esquema de este sistema:

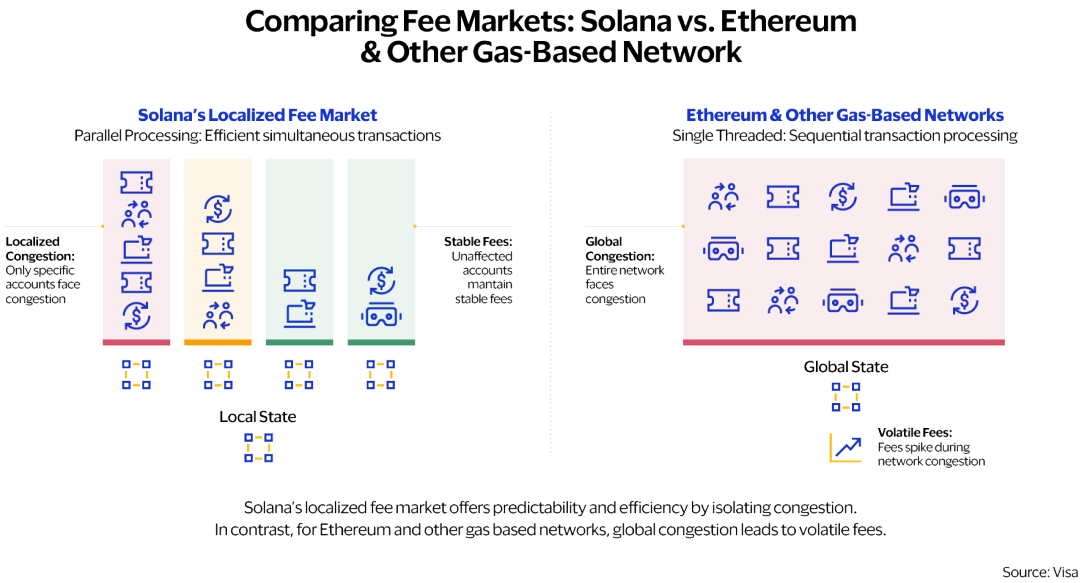

Ilustración 14. Comparativa del costo de tarifa de mercado. Fuente: VISA

En los siguientes apartados de esta sección estudiaremos más a fondo como ha sido posible obtener estas ventajas técnicas que caracterizan y popularizan a Solana Blockchain mediante múltiplos factores.

3.2. La red de Solana

En este apartado, nos adentraremos en la arquitectura y el funcionamiento interno de la red de Solana. Comenzaremos con una visión general del diseño de la red, que sienta las bases para entender las innovaciones técnicas que la distinguen de otras Blockchains. A continuación, exploraremos el concepto de *Proof of History* (PoH), una de las características más distintivas de Solana que permite una mayor eficiencia y rapidez en las transacciones.

Nos sumergiremos también en el modelo de consenso de la red, abordando temas como la sincronización, la rotación de líderes para la creación de nuevos bloques, y la generación y dirección de bifurcaciones. Examinaremos "Turbine", el protocolo de Solana para la propagación de bloques de alto rendimiento, y discutiremos el estado actual de los bloques dentro de la red.

Finalmente, abordaremos la seguridad en el modelo de consenso a través del *Proof of Stake* (PoS), así como la delegación del *Stake* y las recompensas asociadas. Este apartado tiene como objetivo proporcionar una comprensión completa de la red de Solana, desde su diseño hasta su modelo de consenso, pasando por sus mecanismos de seguridad y eficiencia.

Con esta introducción, esperamos preparar el terreno para una discusión más detallada de cada uno de estos aspectos cruciales que hacen de Solana una Blockchain única en su clase.

3.2.1. Diseño general de la Red

En la mayoría de los sistemas que emplean el modelo de consenso *Proof of Stake* (PoS), la selección de un nuevo líder para la creación de bloques es un procedimiento estándar. Sin embargo, Solana introduce una variante significativa en este proceso mediante un mecanismo innovador conocido como *Gulf Stream*[17]. A diferencia de Ethereum y otras Blockchains tradicionales, que utilizan un *Mempool* para almacenar transacciones pendientes, Solana elimina esta etapa intermedia.

La clave de esta eficiencia radica en la implementación del *Proof of History* (PoH), que permite a Solana tener un sistema determinista para predecir quién será el próximo líder en la creación de bloques. Esto significa que cuando un líder es seleccionado, ya tiene en su posesión todas las transacciones que necesitan ser incluidas en el próximo bloque. No hay necesidad de esperar a que las transacciones se acumulen en un *Mempool*; en cambio, se transmiten directamente al líder designado.

Este enfoque no solo acelera el proceso de creación de bloques, sino que también mejora la eficiencia general de la red al eliminar la necesidad de gestionar y acceder a un *Mempool* separado. En resumen, el mecanismo de *Gulf Stream*, en combinación con el *Proof of History*, permite a Solana ofrecer un rendimiento y una eficiencia sin precedentes en la creación de bloques.

Una vez el líder ha creado el nuevo bloque con las transacciones, entra en juego un mecanismo de propagación de alto rendimiento conocido como *Turbine*. Este sistema está diseñado para distribuir eficientemente la información del bloque recién creado a toda la red de validadores.

Turbine optimiza la difusión de bloques al dividir la información en fragmentos más pequeños y luego distribuirlos en paralelo a través de múltiples nodos[15]. Este enfoque reduce significativamente el tiempo necesario para que todos los validadores reciban y procesen el nuevo bloque, mejorando así la velocidad y la eficiencia de la red en su conjunto.

En resumen, *Turbine* actúa como un acelerador que facilita una rápida y eficiente distribución de bloques, permitiendo que Solana mantenga su alto rendimiento y baja latencia, incluso a medida que la red crece en tamaño y complejidad.

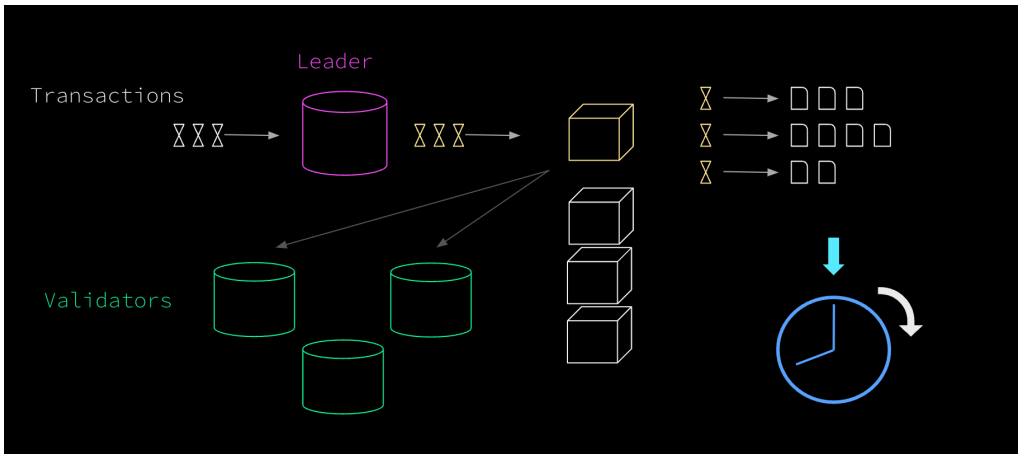

Ilustración 15. Esquema del diseño de la red. Fuente: Solana 101 - Solana Foundation, Taiwan Hacker House.

En relación a la ejecución de las transacciones contenidas en cada bloque, Solana emplea un sistema innovador de paralelización llamado Sealevel. Este es el primer entorno de tiempo de ejecución (Run Time) que ha logrado paralelizar de manera efectiva tanto transacciones como contratos inteligentes, optimizando así la eficiencia computacional de la red.

La sinergia entre *Gulf Stream, Turbine y Sealevel* es lo que permite a Solana alcanzar niveles de escalabilidad sin precedentes. Gulf Stream asegura que las transacciones sean transmitidas de manera eficiente al líder de la creación de bloques, Turbine facilita una distribución rápida y eficaz de la información del bloque a todos los validadores, y Sealevel optimiza la ejecución de transacciones al permitir su procesamiento en paralelo.

3.2.2. Proof of History (PoH)

Una de las cuestiones más desafiantes en sistemas distribuidos es el consenso sobre el tiempo. Tradicionalmente, las redes descentralizadas han resuelto este problema mediante soluciones de tiempo centralizadas y confiables. Sin embargo, en sistemas adversarios como la Blockchain, esto se vuelve aún más complicado. Solana aborda este desafío con su innovador algoritmo de *Proof of History* (PoH), que actúa como un reloj descentralizado para la red.

3.2.2.1. ¿Qué es Proof of History?

El Proof of History es una función de retardo verificable de alta frecuencia (VDF). Utiliza una función hash secuencial resistente a preimágenes que se ejecuta continuamente sobre sí misma, registrando periódicamente el recuento y la salida actual. Esto garantiza que ha pasado un tiempo real entre cada contador generado, y que el orden registrado de cada contador es el mismo que en tiempo real. Es relevante destacar que Anatoly Yakovenko, conceptualizó este enfoque de retardo verificable incluso antes de que se formalizara en el estudio sobre "*Verifiable Delay Functions*" publicado por la Universidad de Stanford y la Universidad de Nueva York[18].

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	hash1
2	sha256(hash1)	hash2
3	sha256(hash2)	hash3

Ilustración 16. Secuencia de PoH. Fuente: Solana Whitepaper.

Aquí, hashN representa la salida hash real en cada iteración.

No es necesario publicar todas las salidas hash; se pueden publicar subconjuntos en intervalos específicos para verificar el tiempo transcurrido.

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	hash1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

Ilustración 17. Secuencia de PoH con Intervalos. Fuente: Solana Whitepaper.

Si la función hash elegida es resistente a colisiones, este conjunto de hashes solo puede calcularse en secuencia por un solo hilo de la computadora. Esto se debe a que no hay forma de predecir qué valor tendrá el hash en el índice 300 sin ejecutar realmente el algoritmo desde el valor inicial 300 veces. Por lo tanto, podemos inferir que ha pasado un tiempo real entre el índice 1 y el índice 300.

En el ejemplo, se produjo el hash 62f51643c1 en el contador 510144806912 y el hash c43d862d88 en el contador 510146904064. Dadas las propiedades previamente discutidas del algoritmo PoH, podemos confiar en que ha pasado un tiempo real entre estos dos contadores[19].

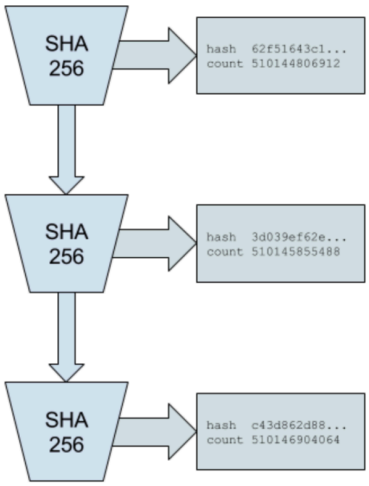

Ilustración 18. Esquema de secuencia de PoH con Intervalos. Fuente: Solana Whitepaper.

3.2.2.2. Marcas de tiempo en Proof of History

Los datos pueden insertarse en la secuencia anexando el hash de los datos al estado generado anteriormente. Esto hace que todas las salidas futuras cambien de manera impredecible y, siempre que la función hash sea resistente a preimágenes y colisiones, es imposible crear una entrada que genere un hash deseado en el futuro.

En el ejemplo mostrado en la *Ilustración 19*, observamos cómo la introducción de un nuevo valor de entrada en la secuencia genera un efecto en cadena que altera los resultados futuros, haciendo que estos sean impredecibles. Esto se debe a que un simple cambio en un valor de hash cambia completamente el resultado final.

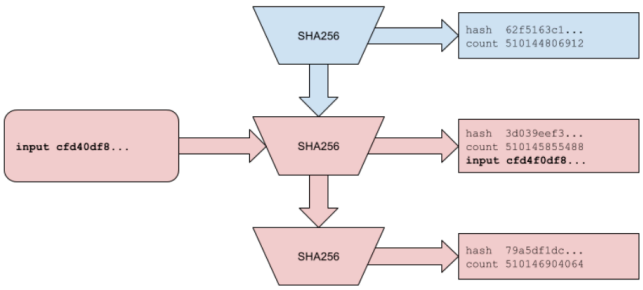

Ilustración 19. Introducción de un input en un momento específico de una secuencia de PoH. Fuente: Solana Whitepaper.

3.2.2.3. Verificación Paralela

Aunque la secuencia registrada solo puede generarse en un solo núcleo de CPU, la salida se puede verificar en paralelo. Cada segmento de secuencia grabado se puede verificar de principio a fin en núcleos separados en un tiempo significativamente menor al que se tardó en generarlo.

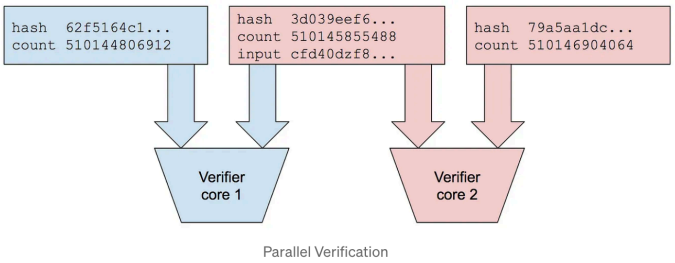

Ilustración 20. Verificación paralela en PoH. Fuente: Solana Whitepaper.

3.2.2.4. Escalado Horizontal

Es posible sincronizar múltiples generadores de Proof of History mezclando el estado de la secuencia de cada generador entre sí. Esto permite un escalado horizontal de la generación de Proof of History sin fragmentación.

3.2.2.5. Consistencia y Seguridad

Para hacer que la secuencia generada sea resistente a ataques, cada evento generado por el cliente debe contener el último hash observado que el cliente considera válido. Esto limita la capacidad de un atacante para generar una secuencia oculta o invertir el orden de los eventos.

Supongamos que un usuario quiere enviar un mensaje o realizar una transacción en la red. Este usuario incluirá en su mensaje un hash específico de la secuencia de Proof of History, que en este caso es "0xdeadc0de". Este hash se considera como el último estado válido del sistema y se incluye en el mensaje firmado digitalmente por el usuario.

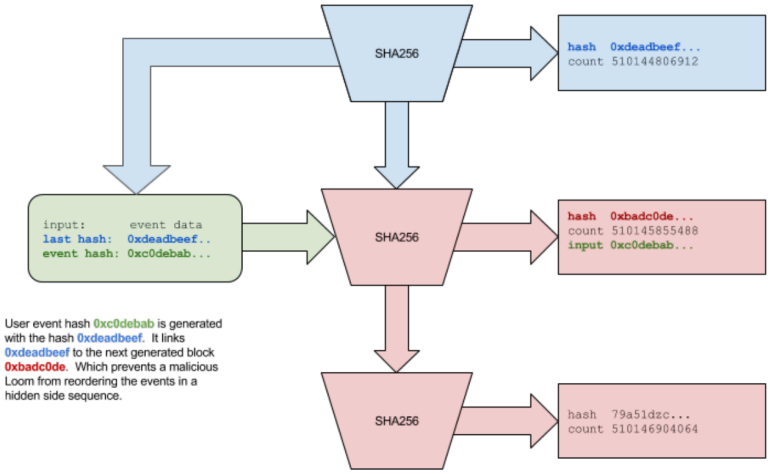

Ilustración 21. Input con una referencia anterior en PoH. Fuente: Solana Whitepaper.

Ahora bien, este mensaje firmado con el hash "0xdeadc0de" se reinserta en la secuencia de Proof of History. Esto crea un efecto similar a tomar una foto con el periódico New York Times en el fondo y, al día siguiente, ver esa misma foto publicada en el New York Times.

Sabemos que el contenido de esa foto existía antes y después de un día específico, validando así el tiempo real transcurrido.

En términos técnicos, si el hash "0xdeadc0de" fue generado después del recuento 510144806912 en la secuencia de Proof of History, y el mensaje del usuario se reinserta y se registra en un recuento posterior, digamos 510146904064, podemos confiar en que ha pasado tiempo real entre esos dos recuentos.
Este mecanismo de "referencia cruzada" actúa como un sello de tiempo criptográfico irrefutable que valida la autenticidad y la secuencia temporal de cada evento o transacción en la red. Es un método anti-fraude robusto que garantiza que el sistema sea resistente a manipulaciones maliciosas, añadiendo una capa adicional de seguridad y consistencia.

3.2.3. Modelo de consenso de la Red

En esta sección, exploraremos los elementos clave que hacen que nuestro modelo de consenso sea seguro, eficiente y escalable. Desde la sincronización de nodos hasta la generación de nuevos bloques y la seguridad a través del Proof of Stake, cada subtema ofrece una pieza del rompecabezas que es nuestro sistema de consenso. A continuación, desglosaremos estos componentes para entender mejor cómo funcionan juntos para mantener la red en marcha.

3.2.3.1. Sincronización

La sincronización en Solana se beneficia enormemente del Proof of History, permitiendo una transmisión de datos más rápida y fiable entre los nodos. A diferencia de las Blockchains tradicionales que dependen de tiempos de bloque más largos, Solana divide los bloques en "entradas" más pequeñas que se transmiten en tiempo real a los validadores. Este enfoque permite a los validadores procesar transacciones de manera optimista antes de votar sobre su validez, reduciendo así el tiempo de latencia y mejorando la eficiencia del sistema[20].

3.2.3.2. Rotación de lideres para la creación de nuevos bloques

En Solana, la responsabilidad de crear nuevos bloques se rota entre los validadores para minimizar el riesgo de censura y asegurar una red más robusta. Este proceso se conoce como rotación de líderes. Cada validador sabe quién es el líder actual y quién será el próximo, gracias a un algoritmo común y un calendario de eventos predefinido llamado "programación de líderes"(*leader Schedule*). Este programa se recalcula periódicamente y se basa en el estado del ledger y la cantidad de stake en cada validador.

La rotación se realiza en etapas(*epochs*), que son periodos de tiempo durante los cuales un conjunto específico de líderes está activo. Al final de cada época, se genera un nuevo programa de líderes basado en el staking y el rendimiento de los validadores. Este enfoque asegura que todos los validadores tengan una oportunidad justa de liderar la creación de bloques, al tiempo que se adapta a cualquier cambio en la composición de la red.
Es importante destacar que la rotación de líderes está diseñada para funcionar de manera eficiente incluso en escenarios de partición de red o ataques maliciosos, ajustando dinámicamente la programación de lideres según sea necesario.

3.2.3.3. Generación y dirección de bifurcaciones

La generación y gestión de bifurcaciones es un aspecto crucial para mantener la integridad y eficiencia de la red. En lugar de esperar un consenso completo en cada bloque, se permite que múltiples bifurcaciones coexistan temporalmente, lo que da a los validadores la flexibilidad de votar en la bifurcación que consideren más viable, generalmente a través del algoritmo de consenso Tower BFT.[21].
Desde el punto de vista del validador, el flujo de tiempo en la secuencia de Proof of History (PoH) es esencial para la generación y gestión de estas bifurcaciones.

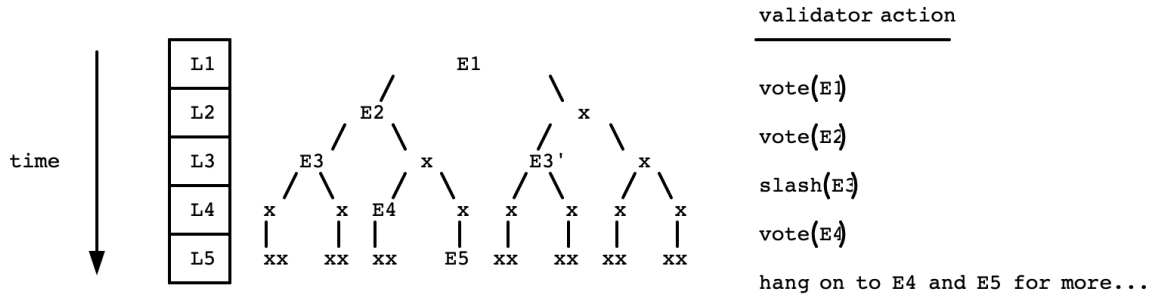

Ilustración 22. Punto de vista de un validador en la generación de bifurcaciones(“fork”). Fuente: Solana Docs por Solana Foundation.

Los líderes, identificados como L1, L2, etc., tienen slots de tiempo asignados para añadir nuevas entradas al ledger. Los "ticks" actúan como marcadores de tiempo, indicando períodos sin cambios de estado.

Si se detecta una entrada duplicada en el mismo slot en diferentes bifurcaciones, se activa una condición "slashing", lo que podría llevar a penalizaciones para el líder responsable. En tal caso, el validador tiene la libertad de descartar una bifurcación y comprometerse con la otra, minimizando así el riesgo y la incertidumbre.

La red está diseñada para tolerar la pérdida de conexión con cualquier líder. En tales casos, se puede sintetizar lo que el líder habría generado, asegurando la continuidad y la robustez del sistema. Todas las bifurcaciones posibles se almacenan en una estructura de árbol llamada "blockstore". Es tarea del validador evaluar estas bifurcaciones y decidir cuál seguir.

Con el tiempo, mantener múltiples bifurcaciones puede ser computacionalmente costoso. Para mitigar esto, se emplean técnicas mediante Tower BFT de "pruning" y "squashing" que optimizan la estructura de datos al eliminar bifurcaciones redundantes y compactar la información[22].

En resumen, la arquitectura está optimizada para manejar bifurcaciones de manera eficaz, permitiendo una toma de decisiones de consenso rápida y una alta tolerancia a fallos, todo ello con un uso eficiente de los recursos del sistema.

3.2.3.4. Turbine: Alto rendimiento en la propagación de bloques

El desafío fundamental en la escalabilidad de blockchain radica en el ancho de banda. A medida que aumenta el número de nodos en una red blockchain, el tiempo necesario para propagar datos a todos los nodos tiende a incrementar, lo que se conoce como el dilema de la escalabilidad[15].

Solana, con su protocolo Turbine, propone una solución ingeniosa a este problema. En lugar de depender de transmisiones punto a punto, donde un nodo (o líder) transmite datos a cada uno de los otros nodos individualmente, Turbine fragmenta los datos en paquetes más pequeños y los distribuye en cascada a través de grupos de nodos denominados "neighborhood". Imagina un sistema donde un mensaje se pasa no solo a un receptor directo, sino a un grupo, y ese grupo a su vez lo pasa a otros grupos, amplificando la velocidad de transmisión.

Supongamos una red de 20,000 validadores. El líder deberá dividir el bloque de 128MB en paquetes de 64KB, produciendo un total de 2,000 de ellos en los cuales cada uno será enviado a un validador diferente.

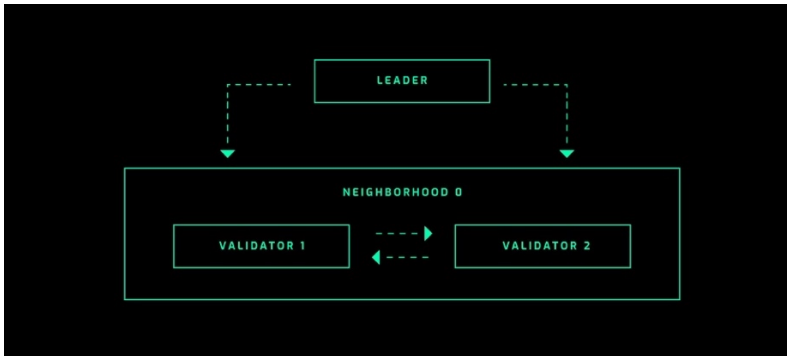

Ilustración 23. Turbine Engine. Fuente: Turbine – Anatoly Yakovenko.

Dentro de la red, cada validador tiene la responsabilidad de retransmitir el paquete a su respectivo grupo o neighborhood. Esta estructura permite visualizar la red como un árbol compuesto por múltiples neighborhoods, facilitando la expansión de la red para albergar a más de 1,000 validadores.

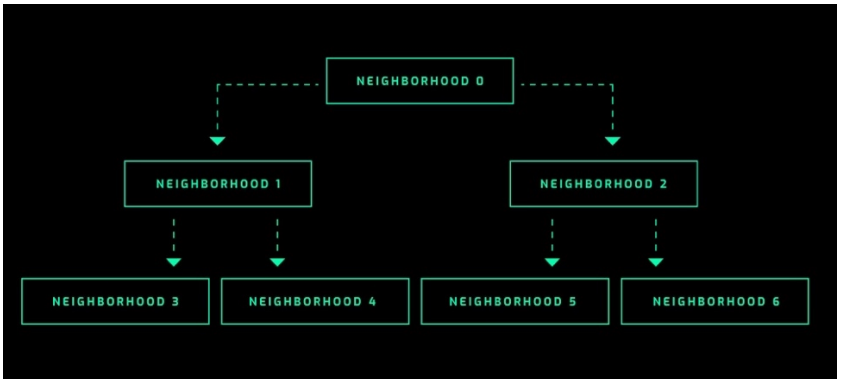

Ilustración 24. Turbine Engine. Distribución a vecindarios(neighborhoods). Fuente: Turbine – Anatoly Yakovenko.

Adicionalmente, cada neighborhood tiene la tarea de distribuir una porción de sus datos a los grupos subordinados en la estructura jerárquica del árbol.

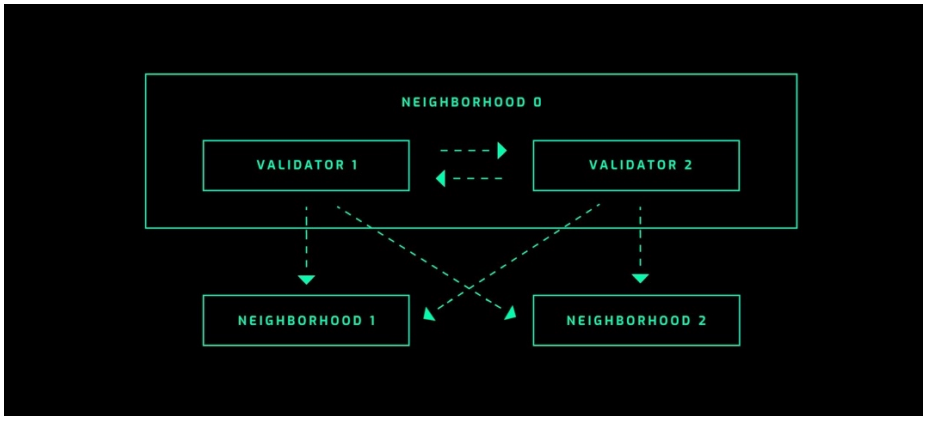

Ilustración 24. Turbine Engine. Retransmisión de paquetes a vecindarios(neighborhoods) subordinados en la jerarquía del árbol. Fuente: Turbine – Anatoly Yakovenko.

La seguridad representa un desafío significativo con esta técnica. Por ejemplo, nodos hostiles podrían optar por no retransmitir datos o difundir información incorrecta. Para contrarrestar estos nodos, el líder emplea códigos de borrado Reed-Solomon. Estos códigos permiten a cada validador reconstruir el bloque completo, incluso si no han recibido todos los paquetes.

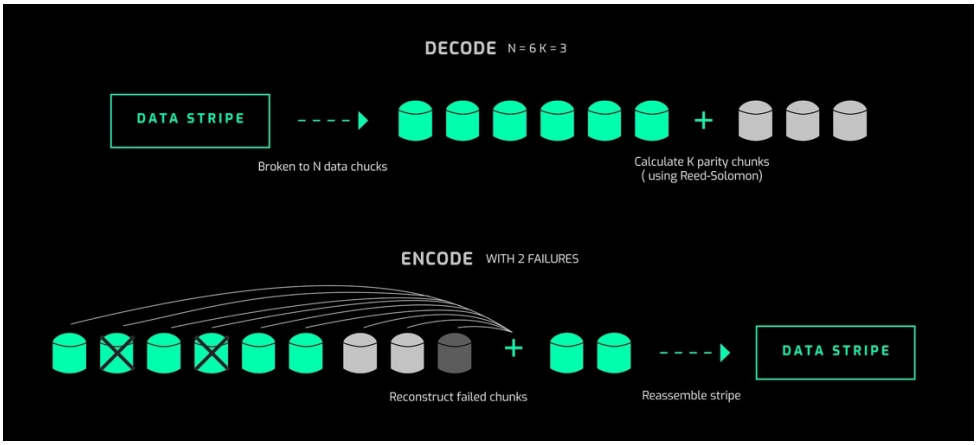

Ilustración 25. Turbine Engine. Reed-Solomon Códigos de borrado (Erasure Codes). Fuente: Turbine – Anatoly Yakovenko.

Si el líder envía el 33% de los paquetes del bloque como códigos de borrado, la red tiene la capacidad de descartar hasta un 33% de los paquetes sin comprometer la integridad del bloque. Además, los líderes tienen la flexibilidad de ajustar este porcentaje en función de las condiciones de la red, basándose en las tasas de pérdida de paquetes observadas en bloques anteriores.

Es esencial reconocer que no todos los validadores tienen la misma relevancia. Los más significativos son aquellos con mayor participación en la red. Por lo tanto, la propagación se prioriza en función de esta participación. Un algoritmo ponderado por participación estructura el árbol de manera que los validadores con mayor participación se encuentren en neighborhoods más cercanos al líder. Cada validador calcula de manera autónoma este árbol. Aunque los códigos de borrado pueden corregir fallos, existe la posibilidad de que nodos hostiles se ubiquen estratégicamente en el árbol para provocar fallos que superen su participación combinada, en especial si se suman a ataques de denegación de servicio.

3.2.3.5. Estatus del bloque

El métrico de compromiso(*commitment)* proporciona a los clientes una medida estándar de la confirmación de la red para un bloque en particular. Los clientes pueden utilizar esta información para derivar sus propias métricas de compromiso[23].

Existen tres estados específicos de compromiso para un bloque:

1. **Procesado (Processed)**

2. **Confirmado (Confirmed)**

3. **Finalizado (Finalized)**

A continuación, se presenta una tabla que detalla las propiedades asociadas con cada estado de compromiso:

Property	Processed	Confirmed	Finalized
Received block	X	X	X
Block on majority fork	X	X	X
Block contains target tx	X	X	X
66%+ stake voted on block	-	X	X
31+ confirmed blocks built atop block	-	-	X

Ilustración 26. Estatus de Bloques. Fuente: Solana Docs por Solana Foundation.

- **Procesado**: Indica que el bloque ha sido recibido por la red.

- **Confirmado**: Significa que el bloque no solo ha sido procesado, sino que también ha recibido votos de más del 66% de la participación total.

- **Finalizado**: Es el estado de compromiso más alto, indicando que el bloque ha sido confirmado y, además, se han construido más de 31 bloques confirmados sobre él.

Estos estados ofrecen a los clientes una visión clara y estructurada del nivel de compromiso y seguridad de un bloque en la red. Estos parámetros tienen gran importancia para los desarrolladores a la hora de entender el comportamiento de los estados.

3.2.3.6. Delegación del Stake y Recompensas

La infraestructura de Solana ha implementado un sistema robusto para la delegación de staking y la asignación de recompensas. A continuación, se detalla cómo los actores, desde

validadores hasta stakers, interactúan en este marco y cómo se distribuyen las recompensas por su aporte.[24]

Dinámica del Staking y Recompensas

Los usuarios que hacen staking son incentivados por su contribución en la validación del ledger. Para ello, delegan su stake a nodos validadores. Estos validadores, a su vez, se encargan de reproducir el ledger y votar mediante una cuenta de votos específica a la que los stakers pueden delegar su cantidad de staking. El sistema utiliza estos votos ponderados por stake para seleccionar opciones cuando surgen bifurcaciones. Es esencial que tanto los validadores como los stakers tengan incentivos alineados: los validadores necesitan compensación por su hardware y los stakers buscan recompensas. Delegar en validadores fraudulentos puede resultar en “slashing”, lo que significa una pérdida total de la cantidad delegada.

Distribución de Recompensas

El proceso de recompensas se divide en dos programas en la Blockchain. El programa de votos garantiza que los stakes puedan ser penalizados si es necesario. El programa de Stake, por otro lado, gestiona el pool de recompensas y permite la delegación pasiva. Es responsable de otorgar recompensas al staker y al votante cuando se demuestra que el stake delegado ha participado en la validación del ledger.

Participación y Recompensas

Las recompensas provienen de una porción de la inflación de la red. La cantidad de *lamports* disponibles para otorgar recompensas en una época es fija y se distribuye equitativamente entre todos los nodos que hacen staking según su peso relativo de stake y su participación.

Retiro y Dinámica del Staking

El sistema de Solana ha sido diseñado para equilibrar la participación y la recompensa. Los stakes no se activan instantáneamente; pasan por un período de "calentamiento"(*warm up*

period). Durante este tiempo, una parte del stake se activa gradualmente. Una vez que un stake se desactiva, parte de él sigue siendo "efectivo" y se está "desactivando". A medida que el stake se enfría (*cooldown period*), sigue generando recompensas y está expuesto a penalizaciones, pero también se vuelve disponible para retiro.

3.3. El modelo de programación en Solana

Solana, como una de las redes Blockchain más avanzadas y escalables, presenta un modelo de programación único que permite a los desarrolladores construir aplicaciones descentralizadas y contratos inteligentes con un alto grado de flexibilidad y eficiencia. Este modelo se distingue por su capacidad para manejar transacciones a gran escala sin comprometer la seguridad o la descentralización mediante un sistema de cuentas que son ejecutadas en paralelo llamado Sealevel o Solana Virtual Environment(SVM). En este apartado, exploraremos los componentes esenciales del modelo de programación de Solana, desde las cuentas y programas hasta las técnicas avanzadas como las Cuentas derivadas del Programa (PDAs) y la Invocación entre Programas. Además, se abordarán herramientas como la Interfaz de Línea de Comando y el Anchor Framework, que facilitan y optimizan el desarrollo en la plataforma.
A través de ejemplos prácticos, se ilustrará cómo estos conceptos se entrelazan para crear soluciones robustas y escalables en la red Solana.

3.3.1. Cuentas

Las cuentas en Solana son un componente esencial para el almacenamiento de estados, actuando como elementos fundamentales en la estructuración de datos dentro de esta red Blockchain[25].

Conceptos Clave

- Las cuentas se utilizan principalmente para almacenar datos.
- Cada cuenta posee una dirección única(256-bit), que suele ser una clave pública.

- Las cuentas tienen un tamaño máximo de 10MB, mientras que las cuentas PDA (Dirección Derivada del Programa) tienen un límite de 10KB.

- Las cuentas PDA pueden firmar en nombre de un programa de manera autónoma a través de Contratos Inteligentes.

- El tamaño de las cuentas se establece en el momento de su creación, pero puede ser ajustado mediante relocalización del tamaño de la cuenta(*realloc*).

- El almacenamiento de datos en las cuentas se paga mediante un sistema de renta(*rent*).
- Por defecto, el propietario de una cuenta es el Programa del Sistema (*System Program*), que viene dado por la dirección 11111111111111111111111111111111.

Modelo de Cuenta

En Solana, existen tres tipos principales de cuentas:

1. **Cuentas de Datos:** Estas almacenan información y se dividen en:

    - Cuentas propiedad del sistema.

    - Cuentas PDA (Dirección Derivada del Programa).

2. **Cuentas de Programas:** Albergan programas ejecutables. A diferencia de los contratos inteligentes en Ethereum, que combinan código ejecutable y no ejecutable, Solana separa estas funciones. Los programas ejecutables poseen y crean otras cuentas que almacenan el estado.

3. **Cuentas Nativas:** Indican programas nativos en Solana, como el Sistema, Stake y Vote.

Las cuentas en Solana pueden ser comparadas con un sistema de archivos de computadora, donde cada cuenta actúa como un "cubo de almacenamiento". Estos cubos pueden

contener desde tokens, como SOL, hasta variables de estado de un programa y programas completos[26].

Rentas en Solana

Mantener datos en las cuentas tiene un costo en SOL, conocido como renta. Esta renta es una medida preventiva para evitar que actores maliciosos sobrecarguen la red. Si una cuenta mantiene un saldo mínimo equivalente a 2 años de pagos de renta, queda exenta de este pago. La renta se cobra en dos momentos: cuando una cuenta es referenciada por una transacción y al final de cada etapa. Si una cuenta no tiene fondos suficientes para cubrir la renta, se elimina de la cadena.

Es esencial entender que cada cuenta en Solana paga una tarifa de alquiler para usar la memoria en la Blockchain. Esta estructura de renta es una medida para garantizar que la red no sea sobrecargada y para ofrecer incentivos económicos, especialmente a los validadores que suministran hardware y poder computacional para verificar todas las transacciones.

Componentes de una Cuenta

Cada cuenta en Solana tiene varios campos que almacenan información específica:

```
{
   key: number,              // The address of the account
   lamports: number,         // Lamports currently held
   data: Uint8Array,         // Data stored in the account
   is_executable: boolean,   // Is this data a program?
   owner: PublicKey,         // The program with write access
}
```

Ilustración 27. Componentes de una cuenta. Fuente: Solana 101 – Taiwán Hacker House.

Siendo Lamports la unidad mínima de SOL, equivalente a 10^{-9} SOL.

En resumen, las cuentas en Solana ofrecen una estructura flexible y eficiente para el almacenamiento de datos y la ejecución de programas, facilitando el desarrollo de aplicaciones descentralizadas y contratos inteligentes en la plataforma.

3.3.2. Programas

Solana brinda a los desarrolladores la capacidad de escribir y desplegar Solana programs en su Blockchain. Estos programas, equivalentes a los contratos inteligentes en otros protocolos, son esenciales para actividades en la cadena, desde Infraestructuras Físicas Descentralizadas (DePIN) y Finanzas Descentralizadas (DeFi) hasta aplicaciones de redes sociales y juegos[27].

Características Principales de los Programas en Solana:

- **Interactividad:** Los programas en Solana están diseñados para procesar instrucciones tanto de usuarios individuales como de otros programas, facilitando una interacción dinámica y multifacética.

- **Diseño Stateless:** A pesar de su naturaleza sin estado, los programas interactúan con datos que se almacenan en cuentas específicas, las cuales se transmiten mediante instrucciones. Esta arquitectura garantiza una gestión eficiente de los datos. Esto permite que los programas sean ejecutados en paralelo.

- **Almacenamiento Seguro:** Los programas se almacenan en cuentas específicamente designadas como ejecutables, lo que asegura la integridad y la seguridad de las operaciones dentro de la red.

- **Flexibilidad en la Programación:** Aunque los lenguajes Rust y C++ son predominantemente utilizados, Solana ofrece la posibilidad de emplear cualquier lenguaje que se dirija al backend BPF de LLVM, ampliando el espectro de desarrollo a lenguajes como Python y Typescript, entre otros.

Estructura y Funcionamiento: La arquitectura de Solana se caracteriza por separar de manera efectiva el código de los datos. Esta distinción permite que un único programa pueda operar en diversas cuentas sin la necesidad de despliegues adicionales, optimizando así los recursos y la eficiencia operativa. Esto también quiere decir que un programa puede llamar a otro programa, tal y como veremos más adelante.

Biblioteca de Programas de Solana (SPL): Solana integra una serie de programas esenciales que establecen la base para las interacciones en la cadena. Estos se categorizan en Programas Nativos, responsables de la funcionalidad básica, y Programas SPL, que respaldan una amplia gama de actividades en la cadena.

Desarrollo y Despliegue: El proceso de desarrollo en Solana permite la utilización de diversos lenguajes, aunque Rust y C++ son los más comunes. Herramientas como Anchor han facilitado el desarrollo en Rust, proporcionando un marco estructurado.

Comúnmente podemos esperar la siguiente estructura de proyectos complejos:

File	Description
lib.rs	Registering modules
entrypoint.rs	Entrypoint to the program
instruction.rs	Program API, (de)serializing instruction data
processor.rs	Program logic
state.rs	Program objects, (de)serializing state
error.rs	Program-specific errors

Ilustración 28. Estructura común de Solana Programs. Fuente: Solana Cookbook.

Antes de su implementación final, los programas se someten a pruebas en entornos controlados como Localhost, Devnet y Testnet, los cuales no tienen costo alguno para los desarrolladores. Posteriormente, se despliega a Mainnet, la cual es considerada la red principal de Solana y donde los assets digitales poseen valor real.

Cluster Environment	RPC Connection URL
Mainnet-beta	https://api.mainnet-beta.solana.com
Testnet	https://api.testnet.solana.com
Devnet	https://api.devnet.solana.com
Localhost	Default port: 8899 (e.g. http://localhost:8899, http://192.168.1.88:8899)

Ilustración 29. Múltiples Redes para desarrollar Programas. Fuente: Solana Cookbook.

3.3.3. Instrucciones y Transacciones

Instrucciones en Solana

Una instrucción en Solana puede considerarse como la unidad atómica de lógica de ejecución. Es, en esencia, un mandato que busca modificar el estado global de la Blockchain de Solana. Estas instrucciones activan programas específicos que, al ser ejecutados, interactúan con el entorno de ejecución de Solana para implementar los cambios deseados en el estado.

```
{
    program_id: number,       // The program this instruction is for
    keys: Array<{             // Accounts involved in the instruction
      key: PublicKey,
      is_mutable: boolean,
      is_signer: boolean,
    }>,
    data: Uint8Array,
}
```

Ilustración 29. Anatomía de una instrucción. Fuente: Solana 101 Taiwan Hacker House.

Aquí está una descripción más detallada de sus componentes:

- **Índice de ID de Programa (Program_id):** Es la dirección única del programa que será invocado por esta instrucción. Piense en esto como la dirección de un software específico en la blockchain que sabe cómo procesar y actuar sobre la instrucción dada.

- **Campos de Claves (Keys):** Este es un array que enumera todas las cuentas necesarias para la instrucción. Cada cuenta en esta lista tiene propiedades específicas:

    - **Mutable (is_mutable):** Indica si la cuenta puede ser modificada o no. Esta propiedad es crucial porque permite al tiempo de ejecución de Solana tomar decisiones sobre el procesamiento en paralelo. Si una cuenta es mutable, significa que podría ser alterada durante la ejecución de la instrucción, lo que podría afectar cómo y cuándo se procesa la instrucción.

    - **Signer:** Determina si la cuenta es una firmante de la transacción. Una cuenta firmante es esencialmente una cuenta que ha autenticado o "firmado" la transacción, proporcionando una capa adicional de seguridad y autenticidad.

- **Campo de Datos (Data):** Este campo contiene el código serializado que instruye al programa sobre qué acción específica debe realizar. Además de la acción, este campo también puede contener argumentos o datos adicionales que el programa necesita para ejecutar correctamente la instrucción.

En resumen, una instrucción en Solana es como un paquete de información que dice: "Quiero que el programa en esta dirección (Program_id) actúe sobre estas cuentas (Keys) de esta manera específica (Data)". La estructura y las propiedades de las instrucciones garantizan que se ejecuten de manera segura y eficiente en la blockchain de Solana.

Transacciones en Solana

En la blockchain de Solana, la ejecución de un programa comienza con la presentación de una transacción al clúster. Cada transacción consta de una o varias instrucciones. El tiempo de ejecución procesará cada una de las instrucciones contenidas en la transacción, en orden y de forma atómica. Si alguna parte de una instrucción falla, entonces toda la transacción fallará. Esto es necesario para evitar situaciones en las cuales se ha intentado

realizar un intercambio o acción, y solo unas condiciones se han cumplido, haciendo que uno de los bandos salga perjudicado.

Resumen de una Transacción

Las transacciones en Solana son el vehículo mediante el cual los clientes interactúan con el sistema. Al enviar una transacción al clúster, los clientes buscan actualizar el tiempo de ejecución, como podría ser el caso de debitando una cuenta. Una transacción en Solana se compone de:

1. Una o más instrucciones.
2. Un conjunto de cuentas para consulta o modificación.
3. Una o más firmas que autentican la transacción.

Si alguna instrucción dentro de la transacción falla, toda la transacción es considerada inválida. Esta atomicidad garantiza la integridad de las operaciones y evita escenarios donde una acción parcialmente completada pueda resultar en desventajas para alguna de las partes involucradas.

Para transmitir una transacción a la blockchain de Solana, se utiliza un cliente RPC, que sirve como intermediario, enviando la transacción a la red.

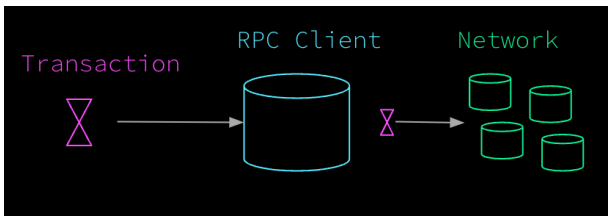

Ilustración 30. Esquema de una transacción. Fuente: Solana 101 - Taiwan Hacker House.

Las firmas verifican que tenemos la autoridad para leer o escribir datos en las cuentas que enumeramos.

Anatomía de una Transacción

Una transacción en Solana es una estructura compleja que permite la interacción y modificación del estado de cuentas en la Blockchain.

```
{
 message: {
   instructions: Array<Instruction>,      // List of instructions
   recent_blockhash: number,          // For de-duplication
   fee_payer: PublicKey,              // Pays “gas” fee
   ... }
 signers: Array<Uint8Array>,      // Signed versions of this transaction
}
```

Ilustración 30. Anatomía de una transacción. Fuente: Solana 101 - Taiwan Hacker House.

A continuación, se presenta un desglose detallado de sus componentes:

- **Mensaje:** Es el núcleo de la transacción y contiene la información esencial que define qué se quiere lograr con la transacción.
    - **Instrucciones:** Dentro del mensaje, encontramos una lista de instrucciones. Cada instrucción es una operación específica que se desea ejecutar. Las transacciones en Solana pueden contener múltiples instrucciones, y el tiempo de ejecución de Solana garantiza que siempre se ejecuten en el orden en que aparecen. Además, Las transacciones en Solana son atómicas. Esto significa que, si alguna de las instrucciones contenidas en la transacción falla, toda la transacción se considera fallida y no se realizará ningún cambio en el estado de la Blockchain.

    - **Blockhash Reciente:** Para garantizar la integridad y evitar transacciones duplicadas, se debe proporcionar un blockhash reciente. Los validadores utilizan este blockhash para identificar y descartar transacciones repetidas o antiguas.

    - **Cuenta que paga por la transacción:** Dentro de la transacción, se designa una cuenta específica que será responsable de pagar las tarifas asociadas

con la transacción. Esta cuenta debe ser una de las firmantes de la transacción.

- **Firmas (Signers):** Es un array que contiene la transacción completa cifrada con la clave privada de una cuenta. Esta acción de cifrar algo con una clave privada, para que pueda ser descifrada con la clave pública correspondiente, se conoce en criptografía como "firmar". La presencia de una firma valida la autenticidad y la autorización de la transacción. En el contexto de Solana, la cuenta que paga las tarifas de la transacción debe ser una de las que firma la transacción.

En resumen, una transacción en Solana es una estructura bien definida que encapsula las operaciones deseadas, las cuentas involucradas y las firmas necesarias para autenticar y autorizar la operación. La naturaleza atómica y ordenada de las transacciones garantiza la integridad y la coherencia del estado en la blockchain.

3.3.4. Programa de Datos personalizado mediante PDAs

Las Direcciones Derivadas de Programa (PDA, por sus siglas en inglés) representan un pilar fundamental en la arquitectura de Solana. Estas direcciones son esenciales para cuentas diseñadas para ser controladas por un programa específico en Solana. A diferencia de las aplicaciones web tradicionales que pueden leer y escribir en sus bases de datos a voluntad, en Solana, se deben utilizar las reglas de propiedad de cuentas para aprovechar las cuentas propiedad de nuestro programa y almacenar los datos del programa.

En otras palabras, un programa de Solana debe escribir en cuentas que posee para almacenar datos en la blockchain de manera similar a una base de datos tal y como se puede observar en la siguiente figura:

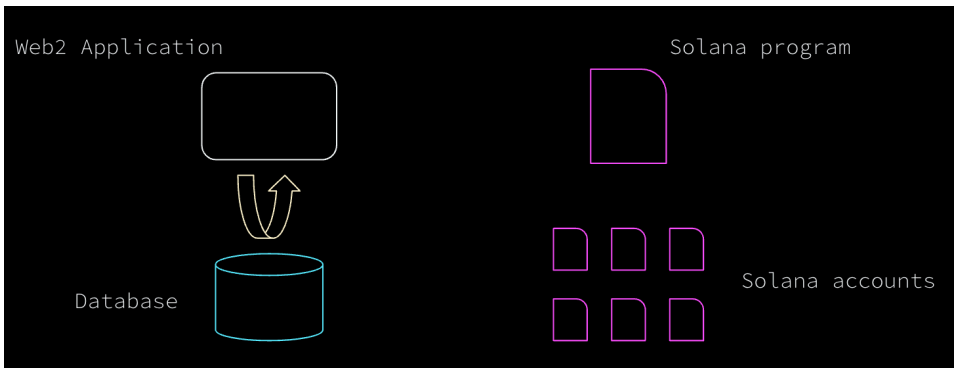

Ilustración 31. Administración de datos de un Solana program. Fuente: Solana 101 - Taiwan Hacker House.

3.3.4.1. Cuentas derivadas del Programa (PDAs)

Las PDAs son cadenas de 32 bytes que parecen claves públicas, pero no tienen claves privadas correspondientes. Son esenciales para desarrollar programas en Solana, ya que permiten a los programas firmar cuentas garantizando que ningún usuario externo también pueda generar una firma válida para la misma cuenta. Además de firmar cuentas, ciertos programas también pueden modificar cuentas mantenidas en sus PDAs[28].

Para comprender el concepto detrás de las PDAs, puede ser útil considerar que las PDAs no se crean técnicamente, sino que se encuentran. Se generan a partir de una combinación de semillas(*seeds*) y un id de programa. Esta combinación de semillas e id de programa se ejecuta a través de una función hash sha256 para ver si generan o no una clave pública que se encuentra en la curva elíptica ed25519.

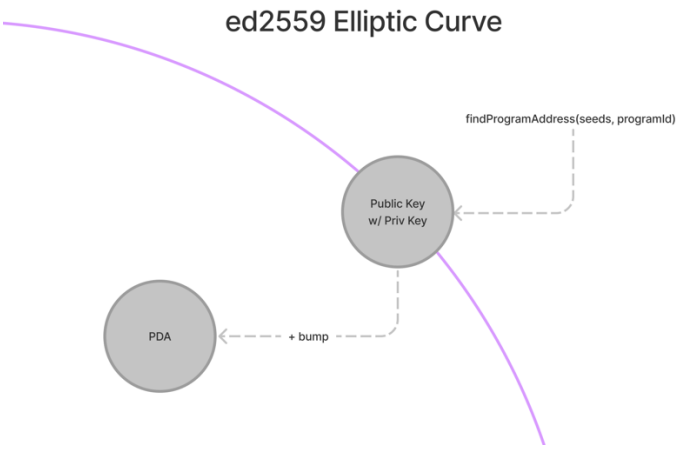

Ilustración 32. Curva Elíptica ed2559 para encontrar un PDA sin llave privada. Fuente: Solana Cookbook.

Al ejecutar nuestro id de programa y semillas a través de una función hash, hay una probabilidad del ~50% de que terminemos con una clave pública válida que sí se encuentra en la curva elíptica. En este caso, simplemente agregamos algo para ajustar nuestra entrada un poco y volvemos a intentarlo. El término técnico para este factor de ajuste es un "bump". En Solana, comenzamos con bump = 255 y simplemente iteramos hacia abajo a

través de bump = 254, bump = 253, etc., hasta que obtenemos una dirección que no está en la curva elíptica. Esto puede parecer rudimentario, pero una vez encontrado, nos da una forma determinística de derivar la misma PDA una y otra vez.

Cuando se encuentra un PDA, la función findProgramAddress devuelve tanto la dirección resultante como el valor bump que se empleó para desplazar dicha dirección fuera de la curva elíptica ed25519. Equipado con este valor bump, un programa tiene la capacidad de firmar cualquier instrucción que haga uso de su PDA. Para realizar esta firma, los programas pasan la instrucción, la lista de cuentas, y las semillas junto con el bump que se usaron para obtener la PDA a invoke_signed. La distinción clave entre un PDA y una cuenta estándar radica en que el PDA puede generar una firma utilizando las semillas y el bump, mientras que en una cuenta tradicional es esencial firmar la transacción para obtener una firma, como se ilustra en la figura siguiente.

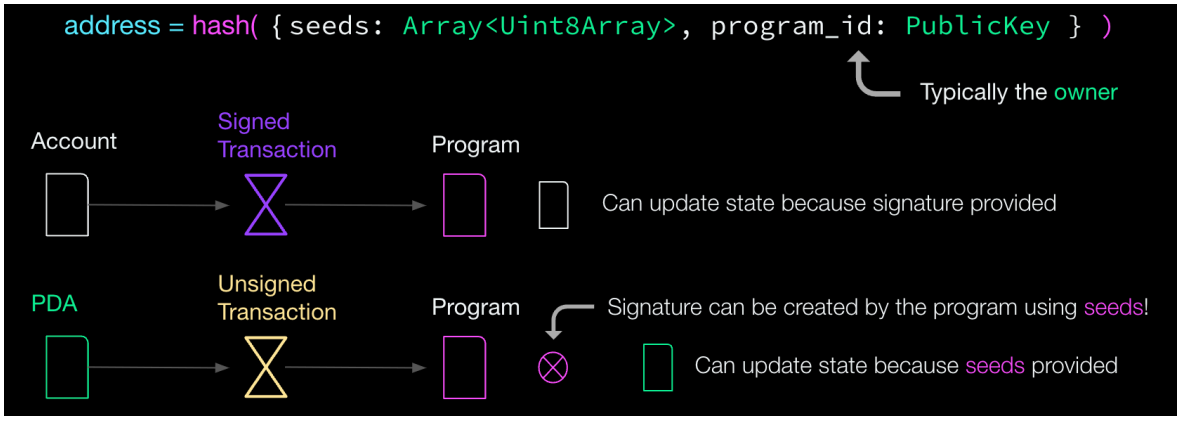

Ilustración 33. Comparación de firma entre cuenta común y PDA. Fuente: Solana 101 - Taiwan Hacker House.

3.3.4.2. Ejemplos visuales de PDAs

Para comprender a fondo la implementación y utilidad de las Direcciones Derivadas del Programa (PDAs) en Solana, es esencial analizar su aplicación práctica en escenarios específicos. A continuación, se presentan dos ejemplos detallados que ilustran la versatilidad y eficiencia de las PDAs en la gestión de datos en la cadena de bloques de Solana.

Análisis de Información de Perfil de Usuario:

Consideremos, como punto de partida, una estructura de datos que almacena información del perfil de un usuario, donde cada registro está intrínsecamente asociado a la dirección de la billetera digital del usuario. En un entorno tradicional, esta asociación se realiza directamente. Sin embargo, en el ecosistema de Solana, es posible optimizar este proceso mediante el uso de cuentas PDA. Estas cuentas, al ser derivadas utilizando patrones específicos basados en semillas, pueden contener datos como el nombre de usuario y el correo electrónico. La principal ventaja radica en que, para consultar la cuenta de perfil de un usuario, sólo es necesario tener acceso a la dirección de su billetera.

Ilustración 34. Ejemplo de un análisis de información de usuario. Fuente: Solana 101 - Taiwan Hacker House.

Este enfoque es particularmente relevante en el contexto de web3, donde las aplicaciones descentralizadas (dApps) suelen requerir que los usuarios autoricen el acceso a sus billeteras, facilitando así la exposición de su clave pública.

Registro de Actividad de Publicaciones de un Perfil:

Profundizando en la gestión de datos, analicemos una estructura que registra el número de publicaciones asociadas a un perfil específico.

En este escenario, las semillas para derivar las PDAs se obtienen no sólo de la dirección de la billetera del usuario, sino también de su dirección de perfil. Es crucial destacar que esta última puede derivarse exclusivamente de la dirección de billetera del usuario, evidenciando la capacidad de las PDAs para establecer relaciones jerárquicas y multidimensionales de datos.

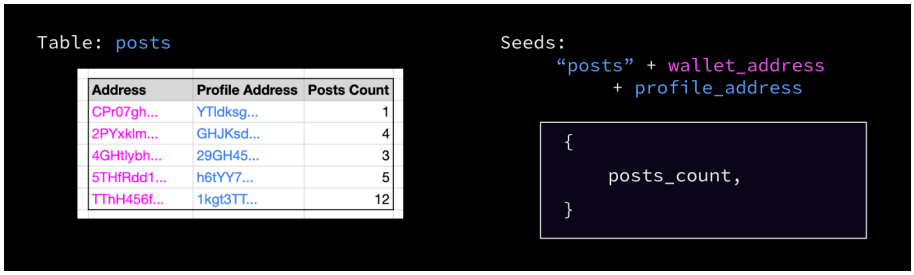

Ilustración 35. Ejemplo de un registro de actividad de publicaciones de un perfil. Fuente: Solana 101 - Taiwan Hacker House.

La metodología detrás de estos patrones de semillas refleja la capacidad de Solana para relacionar datos de manera coherente y estructurada entre diferentes tipos de cuentas, promoviendo una gestión de datos más eficiente y segura en su cadena de bloques.

3.3.5. Invocación entre Programas(CPI)

En el paisaje de las cadenas de bloques, la capacidad de interactuar entre programas distintos es esencial para la construcción de aplicaciones descentralizadas complejas y robustas. Solana, con su arquitectura innovadora, introduce el concepto de Cross Program Invocation (CPI), que permite una interacción fluida y segura entre diferentes programas dentro de su ecosistema.

Características y Beneficios del CPI

- **Composición Modular:** El CPI permite la composición de programas en Solana, facilitando la creación de aplicaciones complejas a partir de componentes más simples y reutilizables.

- **Atomicidad Garantizada:** Una característica distintiva de las transacciones en Solana es su atomicidad. Independientemente de la cantidad de instrucciones o invocaciones anidadas dentro de una transacción, si alguna de ellas falla, toda la

transacción se revierte, garantizando la coherencia y la integridad del sistema en el CPI.

- **Seguridad en la Delegación:** A través del CPI, una instrucción puede delegar responsabilidades a otras, ya sea dentro del mismo programa o en programas diferentes. Esta delegación se realiza de manera segura, con las firmas de las cuentas involucradas pasando automáticamente a las instrucciones anidadas.

Mecanismos de Seguridad en las Invocaciones

La seguridad es primordial en cualquier operación relacionada con la cadena de bloques. En el contexto del CPI:

- Las cuentas que son firmadas en una llamada de programa retienen su estado de firma en todas las CPIs subsiguientes que las incluyen.

- Al emplear invoke_signed, el sistema de Solana realiza una verificación meticulosa, utilizando las semillas proporcionadas y el ID del programa para confirmar la legitimidad de la invocación.

Desafíos y Consideraciones de Diseño

- Uso de Direcciones derivadas del programa (PDAs): Los PDAs son fundamentales en el contexto de CPI. Estas direcciones, derivadas de programas específicos, permiten invocaciones seguras entre programas sin la necesidad de claves privadas tradicionales usando las semillas y el bump.
- **Limitaciones de Profundidad:** Actualmente, Solana impone un límite de profundidad de 4 para las CPIs. Esto exige que los diseñadores y desarrolladores adopten enfoques estratégicos, especialmente cuando se prevén múltiples invocaciones anidadas.

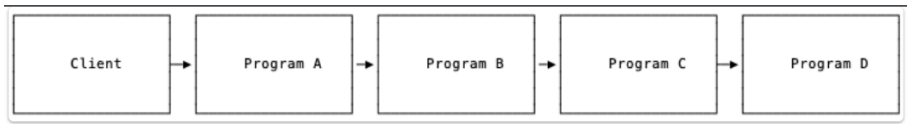

Ilustración 36. Esquema del límite de profundidad en 4. Fuente: Solana Foundation.

- **Restricciones Computacionales:** Aunque no hay un límite estricto en la cantidad de programas que pueden ser invocados de manera no secuencial, es esencial tener en cuenta las limitaciones computacionales y los costos asociados. Una práctica común para evitar los límites de profundidad en 4 es realizar CPI a múltiples programas en paralelo, aunque hay que tener en cuenta de no exceder estas restricciones computacionales en la transacción.

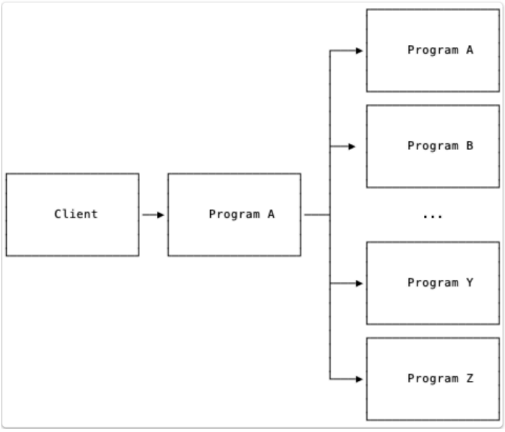

Ilustración 37. Esquema del límite de profundidad en 4. Fuente: Solana Foundation.

3.3.6. Estados de Compresión

La compresión de estados en Solana es una técnica avanzada que permite representar grandes cantidades de datos fuera de la memoria de la cadena de bloques mediante una "huella digital" o hash. Esta huella se almacena en la cadena de bloques, garantizando que los datos originales no hayan sido alterados, y aprovechando la robustez y seguridad del ledger de Solana.[31]

Concepto Central de la Compresión de Estados

La idea es simple: en lugar de almacenar grandes cantidades de datos directamente en la memoria de la cadena, se crea un hash representativo de esos datos y sólo se almacena ese hash. Esto no sólo ahorra espacio, sino que también garantiza la integridad de los datos

originales. Se substituye espacio de la memoria por espacio del ledger, el cual tiene un costo muy menor.

Merkle Trees y su variante concurrente

Un componente clave en este proceso es el merkle tree, una estructura que combina criptográficamente datos en nodos hasta llegar a un único hash raíz. Solana introduce una variante llamada "concurrent merkle tree", que mantiene un registro de cambios en la cadena, permitiendo actualizaciones rápidas y simultáneas en el árbol sin comprometer su validez.

Dimensionando un Merkle Tree Concurrente

El tamaño y costo de un árbol en Solana se determinan por tres parámetros:

1. **Profundidad máxima(depth):** Define el número máximo de niveles en el árbol.
2. **Tamaño máximo del búfer(buffer):** Indica cuántos cambios pueden ocurrir en el árbol simultáneamente.
3. **Profundidad del dosel (canopy depth):** Especifica cuantos nodos como prueba (*proof nodes*) del merkle tree se almacenan directamente en la cadena.

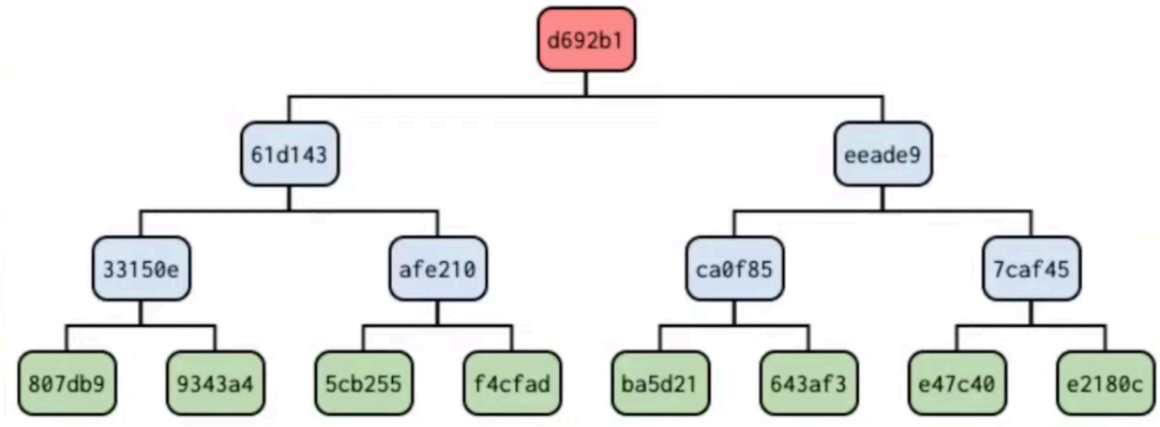

Ilustración 38. Merkle Tree con profundidad de 3, buffer de 2^3 y 8 hojas. Fuente: State Compression and Compressed NFTs, Solana Foundation.

Costo y Eficiencia

Crear un árbol en Solana tiene un costo asociado, determinado por los parámetros mencionados. Sin embargo, la ventaja es clara: al almacenar sólo el hash raíz en lugar de todos los datos, se reduce significativamente el espacio y costo de almacenamiento en la cadena.

En conclusión, la compresión de estados en Solana es una innovación que permite un almacenamiento eficiente y seguro, optimizando el uso del ledger y garantizando la integridad de los datos.

3.3.7. Anchor Framework

Anchor es una framework diseñada para acelerar el proceso de desarrollo de contratos inteligentes en Solana, abarcando desde el diseño hasta la interacción con el cliente. Su principal objetivo es simplificar y optimizar la creación y gestión de estos contratos.

Configuración y Estructura de Anchor

El archivo ***Anchor.toml*** es esencial en cualquier proyecto de Anchor. Actúa como un archivo de configuración que define parámetros cruciales del proyecto, tales como la red de despliegue y los programas gestionados. Este archivo se genera automáticamente al instanciar un nuevo proyecto y se encuentra en la raíz del directorio del proyecto.

Solana-Smart-Contract > Anchor.toml

```
[features]
seeds = false
[programs.devnet]
whitelist = "5FACFiZRqbkpLxi8SmU7vDehZ5LuhAz2vNu7uB2hJTa4"
counter = "GavbQDg7Yeq7FyNZRWpnG5UcHUCXdFqyb9oQiNvcLtkV"
[registry]
url = "https://api.apr.dev"

[provider]
cluster = "devnet"
wallet = "~/.config/solana/id.json"

[scripts]
test = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/*.ts"
whitelist = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/whitelist.ts"
counter = "yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests/**/counter.ts"
```

Ilustración 39. Estructura de Anchor.toml. Fuente: Elaboración Propia.

Componentes Clave de Anchor

1. **Funciones:** Representan bloques lógicos con un propósito específico. Una vez invocadas con éxito, modifican el estado de una o más cuentas en la cadena.

2. **Contextos:** Son conjuntos de cuentas y acciones asociadas que se pasan a una función determinada. Facilitan la interacción y gestión de las cuentas dentro de las funciones.
3. **Cuentas:** Definen la estructura y el formato de las cuentas en la cadena una vez que son deserializadas, actuando como plantillas para el almacenamiento y gestión de datos.

Herramientas de Prueba y Validación en Anchor

Anchor proporciona un conjunto robusto de herramientas para:

- Administrar direcciones de cuentas.

- Ejecutar y gestionar scripts.

- Registrar y monitorear eventos en tiempo real.
- Realizar comprobaciones y validaciones de estados.

Ventajas y Limitaciones de Anchor

Ventajas:

- Permite un desarrollo más ágil, evitando la complejidad de la serialización.

- La extracción de cuentas basada en nombres reduce el riesgo de errores humanos.

- Optimización del código al reducir la redundancia y el código repetitivo.

- Abstracción de la deserialización, simplificando la interacción con contratos.

Limitaciones:

- Los errores generados pueden ser ambiguos y difíciles de interpretar.

- La deserialización de cuentas puede presentar desafíos en ciertos escenarios.
- La nomenclatura entre Rust y TypeScript puede causar confusiones.

3.3.8. Abstracción de Serialización de Anchor vs Solana Nativo

La serialización es esencial en la transmisión eficiente de datos en contratos inteligentes. Mientras Solana ofrece herramientas nativas para este proceso, el framework Anchor simplifica aún más la tarea. En esta sección, contrastaremos la serialización en Anchor con el enfoque nativo de Solana, resaltando sus ventajas y desafíos.

Abstracción de Serialización para Llamadas a Funciones

Sin Anchor: En el enfoque nativo de Solana, cada bit tiene un propósito específico, y es responsabilidad del desarrollador escribir el código repetitivo que modificará el patrón de mensaje en bytes según la funcionalidad del programa que el cliente está invocando. Por ejemplo, para ensamblar datos de instrucción en el lado del cliente para la función *create_pda*, se requiere una estructura específica de bytes. Cualquier reordenamiento de estos bytes podría implementar la misma funcionalidad, pero la flexibilidad requerida es alta y puede ser propensa a errores.

Con Anchor: Anchor simplifica este proceso permitiendo llamadas a funciones en la cadena "directamente" desde el cliente. No es necesario un módulo de descifrado específico ni que el cliente escriba manualmente los datos de instrucción. Aunque la serialización sigue ocurriendo, se realiza de manera transparente a través de macros.

Abstracción de Serialización para Cuentas

Sin Anchor: En el lado del cliente, describir una plantilla de cuenta en TypeScript usando Borsh puede ser tedioso. Además, la deserialización y serialización de cuentas en el lado del programa requiere una gestión cuidadosa del orden y la estructura de las cuentas. Esto puede volverse complicado y propenso a errores, especialmente cuando se manejan múltiples estructuras o colecciones crecibles dentro de una cuenta.

Con Anchor: Anchor proporciona una abstracción más alta, facilitando la descripción y gestión de cuentas. El cliente conoce la estructura de función y la estructura de cuenta a partir del Lenguaje de Descripción de Interfaz (IDL) creado durante la compilación. Esto simplifica la interacción con las cuentas y reduce la posibilidad de errores. Además, las macros en Anchor ocultan interacciones tediosas relacionadas con la creación, validación y escritura en una cuenta en la cadena.

3.3.9. Ejemplos usando los múltiples conceptos de Solana

En esta sección, exploraremos ejemplos prácticos que ilustran la aplicación de conceptos clave en el desarrollo de contratos inteligentes en la blockchain de Solana, utilizando el framework Anchor.

3.3.9.1. Entorno de Trabajo

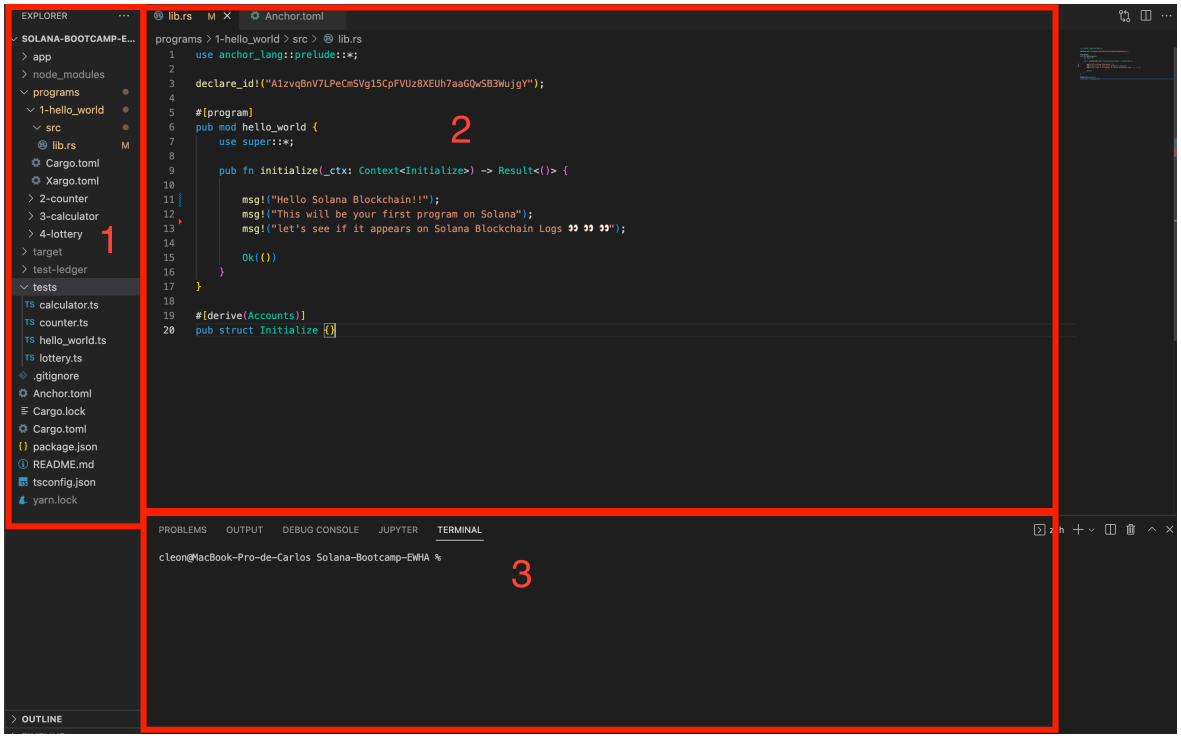

Ilustración 40. Entorno de Trabajo. Fuente: Elaboración Propia.

El desarrollo de contratos inteligentes en Solana con el uso del framework Anchor se caracteriza por tener un entorno estructurado y organizado. A continuación, se describen los componentes principales de este entorno:

1. **Estructura de Archivos**: Define la organización y jerarquía de archivos en el proyecto, siendo la base donde se almacenan y gestionan todos los recursos y códigos del proyecto.

    - **Programs**: Carpeta que contiene los programas esenciales. Estos se desarrollan en Rust, aprovechando las capacidades de Anchor y la sintaxis especial de Solana.

    - **Tests**: Espacio dedicado al lado del cliente, donde se diseñan y ejecutan pruebas usando principalmente Typescript y herramientas específicas para validar los contratos inteligentes.

    - **Anchor.toml**: Archivo de configuración vital para la gestión del proyecto. Establece parámetros como la red de despliegue, comandos de test, nombres de programas y sus direcciones, entre otros.

    - **Otros Componentes**: Además de los mencionados anteriormente, existen otros archivos y carpetas que desempeñan roles específicos en el entorno de desarrollo, como configuraciones, IDLs, librerías, y documentación, entre otros.

2. **Interfaz de Código**: Proporciona una vista detallada y organizada del código fuente, permitiendo al desarrollador entender y modificar las funcionalidades y características del contrato.
3. **Terminal y CLI**: Es la interfaz de línea de comandos donde se introducen y ejecutan las instrucciones para el desarrollo, test y despliegue de los contratos. Es una herramienta esencial para interactuar directamente con el entorno de desarrollo y la blockchain.

3.3.9.2. Interacción Contrato Inteligente – Cliente

El desarrollo en Blockchain requiere una comprensión fundamental del lenguaje de programación utilizado en los contratos inteligentes, así como del lenguaje empleado en el

lado del cliente, que suele ser Typescript o Javascript. La interacción entre el contrato inteligente y el cliente es esencial en la implementación de estos contratos. Esto se debe a que la validación de la funcionalidad correcta del contrato se realiza principalmente a través de interacciones desde el cliente, no directamente en el contrato en sí. Si surge la necesidad de modificar o mejorar un contrato inteligente, es imperativo desplegar una nueva versión en la Blockchain para reflejar estos cambios.

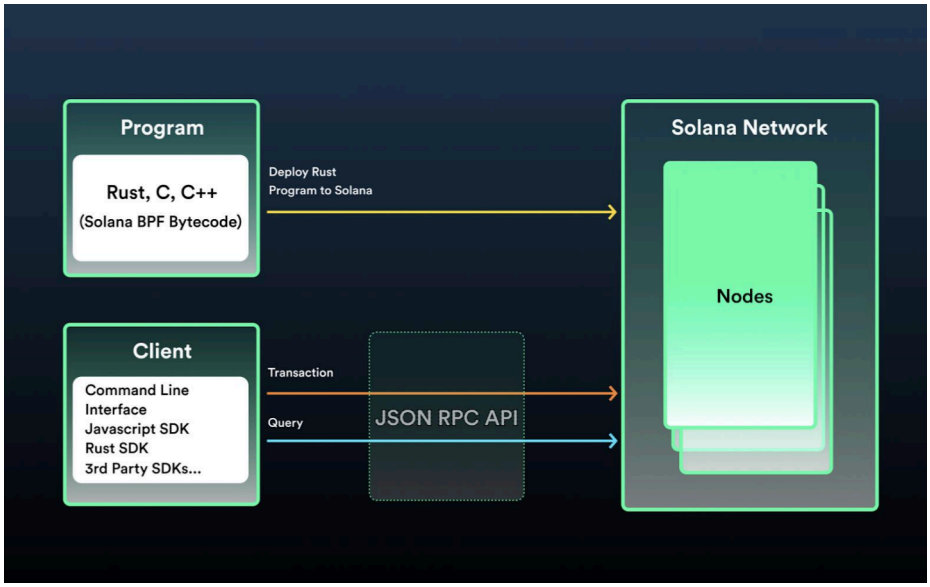

Ilustración 41. Esquema de Programa - Cliente. Fuente: Solana Foundation.

3.3.9.3. Producir Mensaje en el Ledger

En esta sección, presentaremos un ejemplo sencillo que demuestra cómo registrar un mensaje en el ledger de Solana. El objetivo principal de este programa es que, cada vez que se invoque el contrato, se genere un registro (log) con el mensaje: “Hello Solana Blockchain!”. **(Código y Testing en Anexo A1 y B1).**

- **Arquitectura del Funcionamiento**

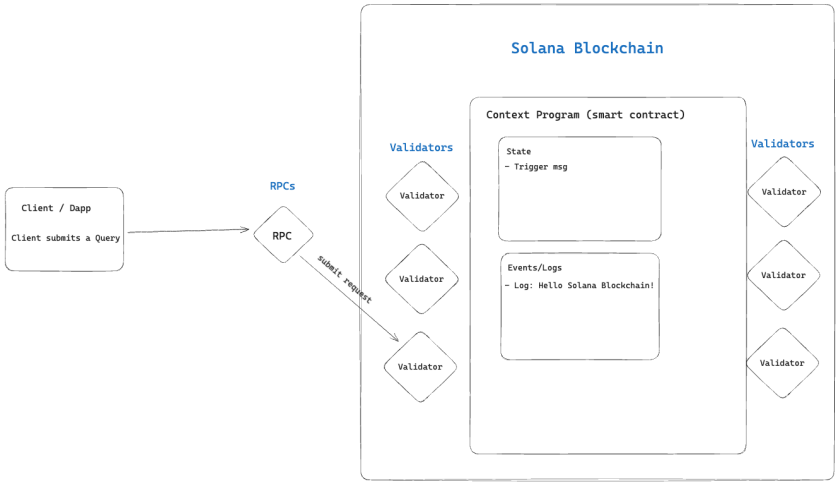

Ilustración 42. Arquitectura del programa - Cliente. Fuente: Elaboración propia.

- **Logs del Programa**

```
Transaction executed in slot 143:
  Signature: hg7tmMA7fnB5aXuwano91TX8ommWjB5pGMUX2hLydHj7aq1idR1qJ1rGCTab4fFs9Ad1u9j5xiYc9cD3Mi5pq5Z
  Status: Ok
  Log Messages:
    Program A1zvqBnV7LPeCmSVg15CpFVUz8XEUh7aaGQwSB3WujgY invoke [1]
    Program log: Instruction: Initialize
    Program log: Hello Solana Blockchain!!
    Program A1zvqBnV7LPeCmSVg15CpFVUz8XEUh7aaGQwSB3WujgY consumed 402 of 200000 compute units
    Program A1zvqBnV7LPeCmSVg15CpFVUz8XEUh7aaGQwSB3WujgY success
```

Ilustración 43. Registros del Programa . Fuente: Elaboración propia.

3.3.9.4. Contador Básico

En este apartado, introduciremos un ejemplo práctico que ilustra la capacidad de almacenar y actualizar datos en una cuenta Program Derived Address (PDA) dentro de nuestro programa en Solana. El núcleo de este ejercicio es un contador simple. Al iniciar, estableceremos un PDA denominado "Counter" con un valor inicial de 0. Posteriormente,

cada vez que se invoque la función de actualización, el contador incrementará su valor en una unidad +1. **(Código y Testing en Anexo A2 y B2).**

- **Arquitectura del Funcionamiento**

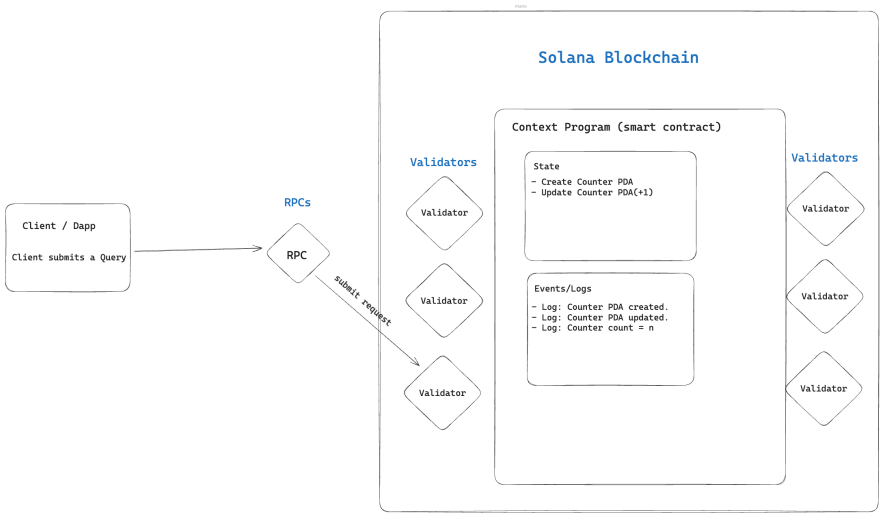

Ilustración 44. Arquitectura del programa - Cliente. Fuente: Elaboración propia.

- **Logs del Programa**

```
Transaction executed in slot 3:
  Signature: 2GiMLEEaosEYNAotWj1LurcpPJXMUm8BNmVMZHLoRwx4CJ5rRyiUFCp8SNPmWaUKd4ANuiXhVbt8ckYmd5qDg6oP
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: CreateCounter
    Program 11111111111111111111111111111111 invoke [2]
    Program 11111111111111111111111111111111 success
    Program log: Creating a Counter!!
    Program log: Current count is 0
    Program log: The Admin PubKey is: GmY3TDnBnrLgvYH1Amn3wzqmegCQkq6ubUGDCkKSSWsM
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 24584 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
Transaction executed in slot 4:
  Signature: 5Z69qBGZHYNKuBnZug6ryxDsaLUycJQvFxA1JTAH29EQWMqAMi2CzhQYDuEdWRY3Ju2qVn5zuzX7gYYVLgLFTtpb
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: UpdateCounter
    Program log: Adding 1 to the counter!!
    Program log: Current count is 1
    Program log: 999 remaining to reach 1000
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 3857 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
```

Ilustración 45. Registros del Programa. Fuente: Elaboración propia.

3.3.9.5. Crear una Lista Blanca mediante PDAs y CPIs

En este apartado, abordaremos un ejemplo de nivel intermedio que destaca la habilidad de mantener una Lista Blanca de direcciones de llaves públicas de manera anónima, aprovechando las capacidades de las Program Derived Addresses (PDAs) y las Cross-Program Invocations (CPIs). El proceso comienza con la creación de una cuenta PDA específica para un contador, que estará vinculada directamente a la Lista Blanca. Cada vez que se añada una nueva dirección a la lista, el contador incrementará su valor en una unidad. Simultáneamente, la dirección se registrará en la Lista Blanca. De manera inversa, al eliminar una dirección, el contador decrementará en uno. Esta estructura garantiza una gestión eficiente y segura de las direcciones en la Lista Blanca, permitiendo un seguimiento preciso de las entradas y salidas. **(Código y Testing en Anexo A3 y B3).**

Estructura del Programa:

- **Arquitectura del Programa**

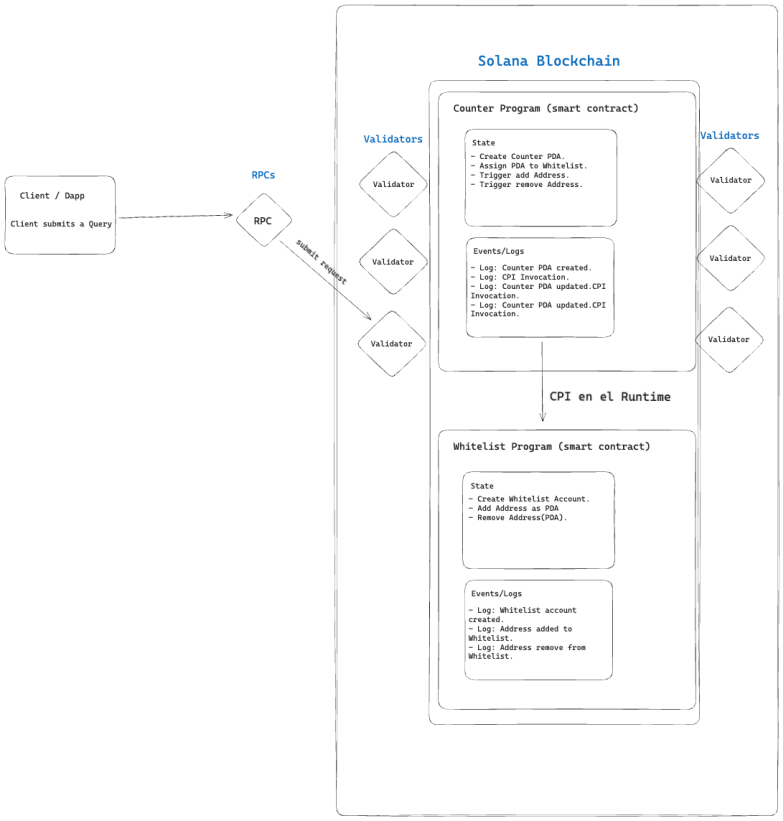

Ilustración 46. Arquitectura del programa - Cliente. Fuente: Elaboración propia.

- **Logs del Programa**

```
Transaction executed in slot 3:
  Signature: 5kMgcjP14TEUfCTGX5WdnrttzbeEPECwuH6u7zvximMoVAvDB7Y7tZHoNinwebifYYJwsp3Tbj7xKcb4VfWydbRD
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: InitCounter
    Program 11111111111111111111111111111111 invoke [2]
    Program 11111111111111111111111111111111 success
    Program log: Initializing counter account...
    Program log: Authority set to ...D15jKaEUvRCTfCKhDHqjL85XZJtVP38AWJAxUpDVxSsk
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 21094 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
Transaction executed in slot 4:
  Signature: 4ZzzZha4SHUntA4TThQvP7NQgPsgZQc1rGKMuKxoT4ikwbM84YD74HPLXtgLvjTQrvbwSmomtpQjqLBLUBNTHLmf
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: PointToWhitelist
    Program log: Pointing Counter to a New Whitelist...{}
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS invoke [2]
    Program log: Instruction: NewWhitelist
    Program 11111111111111111111111111111111 invoke [3]
    Program 11111111111111111111111111111111 success
    Program log: New Whitelist created!
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS consumed 6811 of 190944 compute units
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS success
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 16875 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
Transaction executed in slot 5:
  Signature: r8F5Bgf8k5HnKjRzj9DcfBKGqL5a3eavi4czZD7qrmKi6BUDpWPYnWM9vJ4D9C3EweLJxwFySi2VE2GKw5KNVjp
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: AddOrRemove
    Program log: Adding process started...
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS invoke [2]
    Program log: Instruction: AddAddress
    Program 11111111111111111111111111111111 invoke [3]
    Program 11111111111111111111111111111111 success
    Program log: Adding address...
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS consumed 9717 of 189069 compute units
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS success
    Program log: +1 to the counter...
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 21837 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
```

```
Transaction executed in slot 14:
  Signature: umTQDwBCruLj9XNNJyogd4wbKFXREqRsFVihSnAVkBzzfs18x4W9Px9LopFhHb42Zo9iZUdERRi54zP6oZkLxMb
  Status: Ok
  Log Messages:
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW invoke [1]
    Program log: Instruction: AddOrRemove
    Program log: Removing process started...
    Program log: Checking that the address actually exist in the whitelist
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS invoke [2]
    Program log: Instruction: CheckAddress
    Program log: Checking if address: D7G9GG9WceRJjxWQdCzYwuxqmFPmtfjaAMBGrMURqdWr is in the whitelist
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS consumed 18829 of 190157 compute units
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS success
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS invoke [2]
    Program log: Instruction: RemoveAddress
    Program log: Removing address...
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS consumed 8730 of 167216 compute units
    Program 8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS success
    Program log: -1 to the counter...
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW consumed 42701 of 200000 compute units
    Program FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW success
```

Ilustración 46. Registros del Programa . Fuente: Elaboración propia.

3.3.10. Seguridad en Contratos Inteligentes en Solana

La seguridad es un aspecto primordial en el desarrollo de contratos inteligentes en Solana y un asunto de alta importancia, ya que en 2022 el ecosistema de Solana sufrió en total $500M por exploits. Dada la naturaleza de la blockchain, es esencial garantizar que los contratos sean resistentes a posibles ataques y vulnerabilidades. A continuación, se detallan algunas de las consideraciones y prácticas recomendadas específicas de Solana:

Ataques Específicos en Solana: Dado que los contratos inteligentes de Solana son sin estado, la lógica y los datos necesarios para su ejecución y modificación posterior deben ser proporcionados desde el cliente. Estos datos se presentan en dos formas: datos de instrucción especificados desde el cliente y direcciones de las cuentas en cadena que contienen el estado. Para prevenir ataques que involucran la sustitución de cuentas, las cuentas recién iniciadas se ponen a cero y solo el programa tiene la autoridad para modificarlas.

Alimentar Cuentas para Influenciar la Lógica[32]: Para que la lógica de un programa funcione como se espera, el estado se alimenta a través de sus cuentas. Una cuenta que contiene estado puede ser propiedad del programa que se está invocando, pero no necesariamente tiene que serlo. Si es propiedad del programa, las posibilidades de que sea malicioso son mucho menores, ya que el programa es responsable de inicializar y cambiar su estado.

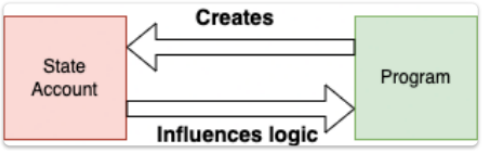

Ilustración 47. Lógica de un Programa – Cuenta con Estado. Fuente: Elaboración propia.

Sin embargo, una cuenta puede ser creada por un programa diferente específicamente para recrear la estructura de una cuenta similar propiedad del programa, pero con algunas sustituciones cruciales. Esta cuenta alimentará al programa para lograr algo que el desarrollador no tenía la intención de que ocurriera. Para prevenir este ataque, cualquier cuenta que se suponga que es propiedad y creada por el programa debe ser verificada por propiedad del programa.

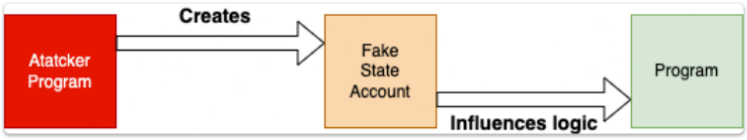

Ilustración 48. Manipulación de la Lógica mediante alimentación a través de una cuenta de estado falsa. Fuente: Elaboraropia.

Fingir ser la Autoridad: Cada transacción incluye no solo cuentas y datos de instrucción, sino también firmas de las partes que han visto la transacción y están de acuerdo con la invocación. Un atacante puede pretender ser esa autoridad alimentando una cuenta propiedad del programa, aunque no sea la autoridad declarada. Para prevenir este ataque, es necesario verificar si la cuenta proporcionada es propiedad del programa, si se adjuntó el privilegio correcto y si la cuenta prevista ha firmado la transacción.

Program Derived Addresses (PDAs) y Validación de Bump Seeds[33]: Las PDAs se utilizan en prácticamente todos los contratos inteligentes de Solana. Sin embargo, es crucial comprender que las PDAs pueden ser falsificadas si sus bump seeds no se validan. Dado que las PDAs de un programa Solana pueden tener las mismas semillas con múltiples bumps válidos, esto tiene una implicación de seguridad significativa.

```
=============VULNERABLE: BumpSeedNotValidated!============
Found a potential vulnerability at line 65, column 8 in src/instructions/withdraw.rs
The account's bump seed is not validated and may be vulnerable to seed canonicalization attacks:
 59|    #[account(mut,
 60|                seeds = [
 61|                    b"deposits".as_ref(),
 62|                    reserve.key().as_ref(),
 63|                ],
 64|                bump = bump)]
>65|    pub deposit_account: AccountInfo<'info>,
```

Ilustración 49. Vulnerabilidad Crítica en la cual el PDA puede ser falsificado al no validar el bump. Fuente: sec3.

Para evitar vulnerabilidades se recomienda la validación de los bump seeds. Para resolver esa vulnerabilidad solo hace falta realizar una pequeña modificación en la función tal y como se ve a continuación:

```
let vault_seeds = &[b"vault", payer.key.as_ref(), &[vault_bump_seed]];
let expected_vault_pda = Pubkey::create_program_address(vault_seeds, program_id)?;

assert_eq!(vault_pda.key, &expected_vault_pda);
```

Ilustración 50. Validación del bump para garantizar la seguridad del PDA. Fuente: sec3.

3.4. Tokens

Dentro del ámbito de la tecnología blockchain, los tokens se erigen como representaciones digitales de activos con propiedades y aplicaciones específicas. Estos se clasifican esencialmente en dos categorías fundamentales:

Tokens Fungibles: Estos tokens se caracterizan por su uniformidad e intercambiabilidad. Cada unidad de un token fungible conserva un valor idéntico a cualquier otra unidad perteneciente a la misma clase. En términos prácticos, se pueden comparar con divisas tradicionales o instrumentos financieros estandarizados, donde cada unidad, independientemente de su procedencia, mantiene una paridad de valor con sus contrapartes.

Tokens No Fungibles (NFTs): A diferencia de los fungibles, los NFTs ostentan una singularidad intrínseca. No pueden ser intercambiados en una relación directa 1:1 con otro NFT debido a sus atributos y valores únicos. Estos tokens son analogables a piezas de arte o activos coleccionables en el mundo físico, donde cada elemento posee características distintivas y, por ende, un valor y significado individualizados.

Ambas categorías de tokens desempeñan roles cruciales en el ecosistema blockchain, facilitando una amplia gama de aplicaciones y soluciones en el ámbito financiero, Gaming, artístico y más allá. Es imperativo comprender sus diferencias y aplicaciones para una adecuada implementación y análisis en el contexto de la tecnología blockchain.

3.4.1. SPL Token Standard

El ecosistema de Solana, al igual que Ethereum, ha establecido un estándar para la creación de nuevos tokens en su red, conocido como el SPL Token Standard. Sin embargo, la operatividad y estructura de estos estándares en Solana presentan diferencias significativas en comparación con otros sistemas.

Una de las principales distinciones radica en que Solana utiliza programas nativos, desplegados directamente en la cadena, para gestionar la creación de nuevos tokens[35]. Estos programas no solo garantizan el cumplimiento del SPL Token Standard, sino que también brindan a los usuarios la capacidad de generar tokens sin la necesidad de escribir o desplegar su propio programa o contrato inteligente.
Dentro de Solana, un token se representa mediante una cuenta denominada "Mint". Esta cuenta alberga la información esencial sobre el token en específico, incluyendo detalles técnicos de este como los decimales, si es fungible o no, ... Es importante destacar que existen otras cuentas asociadas al Mint que facilitan agregar más propiedades y funcionalidades al token en particular.

Por ejemplo:

- **Cuenta Asociada de Token (*Associated Token Account)***: Se utiliza para rastrear el saldo de un monedero de cualquier cuenta de Mint. Esta cuenta tiene una relación directa tanto con el Mint como con el monedero del usuario.

- **Cuentas de Metadatos(*Metadata Account*)**: Para cualquier metadato asociado a una cuenta de Mint, como imágenes, links, títulos, entre otros, se utilizan una o más cuentas de metadatos. Estas cuentas son generadas por el Programa de Metadatos(Metadata Program). Aunque este programa no es un programa nativo de Solana, juega un papel crucial en el estándar de metadatos.

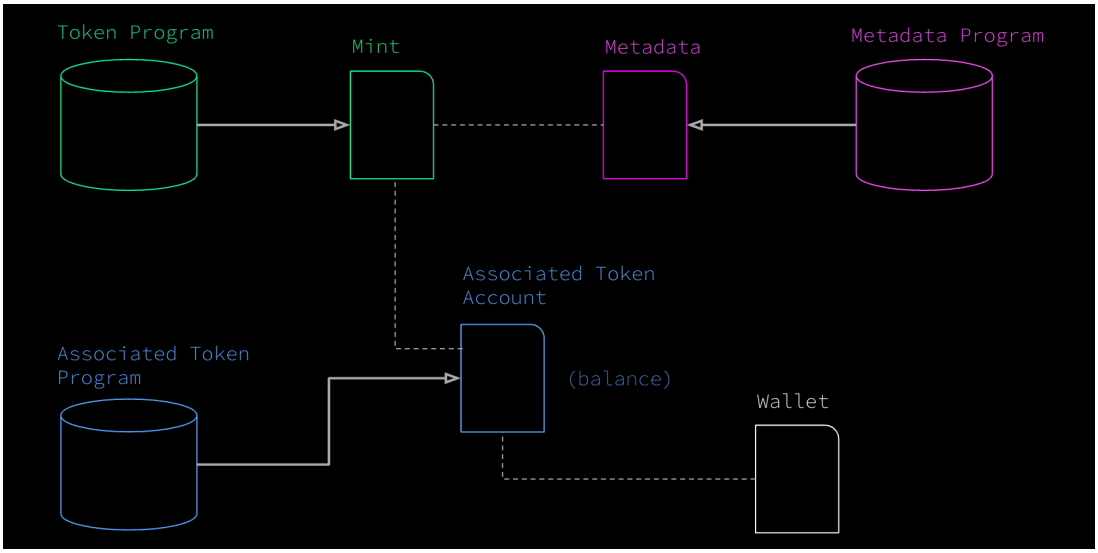

Ilustración 51. Funcionamiento del SPL Token Standard. Fuente: Solana 101, Taiwan Hacker House.

3.4.2. Creación de un Token Fungible usando CLI

La cadena de bloques de Solana ofrece una metodología eficiente y directa para la generación de tokens. Existen multiples maneras de crear tokens; mediante contratos inteligentes, mediante uso de funciones del cliente o simplemente mediante CLI. En estos apartados veremos a través de la Interfaz de Línea de Comandos (CLI) de Solana, debido a que la creación y gestión de tokens se hace de manera simple. En el contexto de este apartado, abordaremos el proceso de generación de un token fungible con un suministro total de 21 millones de unidades[34].

La creación de un token en la red de Solana se caracteriza por su eficiencia y simplicidad. A continuación, se detalla el procedimiento estructurado en pasos secuenciales:

1. Establecimiento de una Nueva Cuenta: Inicie el proceso creando una cuenta dedicada, que servirá como base para las operaciones subsiguientes.

2. Inicialización de la Cuenta como Mint: Una vez creada, es esencial configurar esta cuenta específicamente como una cuenta de Mint. Esta acción la habilita para la emisión y gestión de tokens.

3. Constitución de una Cuenta de Token Asociado: Posteriormente, es necesario establecer una Cuenta de Token Asociado. Esta cuenta actúa como un depósito, permitiendo almacenar y gestionar los tokens que se crearán.

4. Emisión de Tokens a la Cuenta Asociada: Finalmente, se procede a mintear o emitir los tokens deseados, dirigiéndolos directamente a la Cuenta de Token Asociado previamente creada.

Desde una perspectiva técnica, estos pasos se materializan a través de comandos específicos en la Interfaz de Línea de Comandos (CLI) de Solana. A continuación, se presenta una representación de cómo se visualizarían estos comandos en la CLI:

```
$ spl-token create-token
Creating token GDzzCjg6Yqc5Aoxqo48wEoJP5juDWMWLDtVHZwaa39fj

Address:  GDzzCjg6Yqc5Aoxqo48wEoJP5juDWMWLDtVHZwaa39fj
Decimals:  9

Signature: 33L73ZxZbjxWHuGQM6ZRBNvvNY2Drmwo3eefN5oBNX2ppsaKY6PaNLFyCmpyDkYqWXLUtaoiXYVuKrxQ1MBbNMj5

$ spl-token create-account GDzzCjg6Yqc5Aoxqo48wEoJP5juDWMWLDtVHZwaa39fj
Creating account AYApvj1ynqvYHN6j66Cu53dpT3enfBigxWCkRfn6Hw4F

Signature: 4yrC54bipn5z4m2Sfd9gUZpAKtRL3vYeeV64djPutW9jn7eRvsBa3vUtcUgXd4dE53udCw8UgTqHFicggtm4EWuw

$ spl-token mint GDzzCjg6Yqc5Aoxqo48wEoJP5juDWMWLDtVHZwaa39fj 21000000
Minting 21000000 tokens
  Token: GDzzCjg6Yqc5Aoxqo48wEoJP5juDWMWLDtVHZwaa39fj
  Recipient: AYApvj1ynqvYHN6j66Cu53dpT3enfBigxWCkRfn6Hw4F

Signature: 4cRSv3kxqM5dQg3WisiKxsNQxD4oCMtbgrPdeJ9jzLnDkgj8bZRq9mXtApfGwAi7JLXR3UoKAG52n5rSGbSZjFyt
```

Ilustración 52. Creación de un Token Fungible usando Command-Line-Interface(CLI). Fuente: Elaboración propia.

3.4.3. Creación de un Token No Fungible usando CLI

La creación de Tokens No Fungibles (NFTs) en Solana se diferencia de los tokens fungibles en su singularidad y representación de activos únicos. A continuación, se presenta un procedimiento detallado para la creación de un NFT utilizando la Interfaz de Línea de Comandos (CLI) de Solana:

1. Los NFTs se caracterizan por su unicidad, y para garantizar esto en Solana, se crea un token con 0 decimales.
2. Crear una cuenta asociada de Token donde se albergará el NFT.

3. Se procede a emitir una única unidad del NFT hacia la cuenta asociada previamente configurada.
4. Para preservar la unicidad del NFT, se desactiva la posibilidad de emitir más unidades de este token específico en el futuro.

Desde una perspectiva técnica, estos pasos se materializan a través de comandos específicos en la Interfaz de Línea de Comandos (CLI) de Solana. A continuación, se presenta una representación de cómo se visualizarían estos comandos en la CLI:

```
$ spl-token create-token --decimals 0
Loading SPL Token CLI...
Success.

Creating token GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb

Address:  GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb
Decimals:  0

Signature: 43bN5iDzpFtPX5B3VQxnu16rmkyFStf7VxsZvBZwbGyDUPtaiWMBoFL8wTNzwvvdRWCrWpJtwyxPNP2Kdxq4S8Q5

$ spl-token create-account GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb
Creating account 3SjDAe8ToGMz5bLkCkT3Q6s8BYr8wJNj89xNZe1tGm4R

Signature: 42rsrLAWCi1tWQxnFP7iUT5adzWwARVkjG8juT6RUSFdmzvghk97RTXQnCEGB1kNjJbvfNMioG9EiYXNaaumA9R9

$ spl-token mint GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb 1 3SjDAe8ToGMz5bLkCkT3Q6s8BYr8wJNj89xNZe1tGm4R
Minting 1 tokens
  Token: GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb
  Recipient: 3SjDAe8ToGMz5bLkCkT3Q6s8BYr8wJNj89xNZe1tGm4R

Signature: dVPVUibWxm1yzi8bUtRAj58htnn2xDSdM5CV5HgbKqGddQD1hNMw9h9fZ1KuQKCP8DVj8yHVuj985GEavxzUDZu

$ spl-token authorize GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb mint --disable
Updating GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb
  Current mint authority: 3ZKGuxfRG4rVAd2vwewFg7n63tfAbwKtjsDthUtnjydy
  New mint authority: disabled

Signature: HufvQKm5hazPgPAVL4LTFQn3C8V6PSUVxXLyASNNSiKBLoW5aVxT3niivGoVRLAja36JXLq55A1gpKnKn9geSY6

$ spl-token account-info GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb

Address: 3SjDAe8ToGMz5bLkCkT3Q6s8BYr8wJNj89xNZe1tGm4R
Balance: 1
Mint: GX2LCrWyDM5R8pMmufTdfUDabWwP43QLDoNPG3Zyy9Qb
Owner: 3ZKGuxfRG4rVAd2vwewFg7n63tfAbwKtjsDthUtnjydy
State: Initialized
Delegation: (not set)
Close authority: (not set)
```

Ilustración 53. Creación de un Token No Fungible usando Command-Line-Interface(CLI). Fuente: Elaboración propia.

3.4.4. Token-2022

El programa Token-2022 en Solana representa un avance significativo en la gestión de tokens, ofreciendo una implementación común para tokens fungibles y no fungibles. Aunque aún está bajo auditoría, ya ha sido desplegado en todos los clusters para propósitos de prueba y desarrollo[36].

Este programa se concibió como una respuesta a las limitaciones del programa Token original de Solana. A medida que la plataforma atraía a más desarrolladores con ideas innovadoras, surgía la necesidad de un marco más adaptable. Token-2022 responde a este desafío introduciendo "extensiones", que son características adicionales que amplían las capacidades del token.

Las extensiones clave del programa Token-2022 incluyen[37]:

- **Transferencias Confidenciales**: Aumentan la privacidad al permitir transacciones encriptadas.
- **Tokens con Intereses**: Permiten que los tokens acumulen valor con el tiempo.
- **Tokens No Transferibles**: Son tokens que, una vez emitidos, no pueden ser transferidos.
- **Requerimiento de Memo en Transferencias**: Obliga a adjuntar mensajes cortos en las transacciones.
- **Transferencias con Comisiones**: Introduce comisiones en las transferencias de tokens.
- **Cierre de Emisión**: Permite finalizar la capacidad de emitir más tokens de un tipo específico.
- **Tokens con Delegación Permanente**: Otorgan a un delegado autoridad sobre los tokens.
- **Estado Predeterminado de la Cuenta**: Establece restricciones predeterminadas en las cuentas de tokens.

Aunque plataformas como Ethereum y Binance Smart Chain han introducido características innovadoras en el espacio DeFi, Solana, con el programa Token-2022, se posiciona a la vanguardia, superando a sus competidores en términos de adaptabilidad, velocidad y

eficiencia. Esta superioridad técnica de Solana, combinada con las capacidades avanzadas del Token-2022, la coloca como la plataforma líder en la evolución de la tokenización en blockchain.

Conclusión

A lo largo de este trabajo, hemos navegado por el vasto océano de la tecnología blockchain, desde sus inicios con Bitcoin hasta las innovaciones más recientes con Solana. La evolución de esta tecnología ha sido testigo de numerosos avances y desafíos, y Solana emerge como una solución prometedora que aborda muchos de los problemas inherentes a las blockchains anteriores.

Solana, con su enfoque en la escalabilidad y eficiencia, no solo compite con gigantes como Ethereum y Bitcoin, sino que en muchos aspectos, los supera. Su capacidad para manejar un gran volumen de transacciones por segundo, gracias a conceptos avanzados como Turbine y Gulf Stream, la posiciona como una de las blockchains más prometedoras del momento.

El Proof of History (PoH) de Solana, un concepto revolucionario, ha demostrado ser esencial para mantener un registro temporal en un sistema descentralizado. Esta innovación, junto con el modelo de consenso de la red, garantiza una propagación de datos rápida y eficiente.

El modelo de programación de Solana, aunque complejo, ofrece ventajas sin precedentes en términos de velocidad y eficiencia. Las cuentas en Solana, en particular, representan una de las características más distintivas y ventajosas, permitiendo la paralelización y ejecución instantánea de transacciones y contratos inteligentes.

El sistema de tokens de Solana, especialmente con la introducción de Token-2022, se destaca no solo por su sofisticación, sino también por su capacidad de personalización. Esta flexibilidad es indicativa del enfoque de Solana hacia la adaptabilidad y la innovación.

Al reflexionar sobre los múltiples ejemplos presentados, es evidente que, aunque la programación en Solana puede presentar una gran curva de aprendizaje debido a su singularidad, las recompensas en términos de seguridad y eficiencia son inmensas.

Mirando hacia el futuro, Solana tiene un horizonte brillante. A medida que continúa innovando y expandiendo sus capacidades, es probable que juegue un papel fundamental en la configuración del futuro de la tecnología blockchain. Su compromiso con la mejora continua, la escalabilidad y la eficiencia la posiciona como una fuerza líder en el ecosistema Blockchain.

En conclusión, este estudio ha proporcionado una visión profunda y detallada de Solana y su potencial para revolucionar el espacio blockchain. A medida que avanzamos hacia un futuro digital, es esencial mantenerse al tanto de estas innovaciones y comprender su impacto en el ecosistema tecnológico más amplio.

Bibliografía

[1] Haber, S., & Stornetta, W. S. (1991). How to time-stamp a digital document. J.Cryptology. [DOI](https://doi.org/10.1007/BF00196791)

[2] Nakamoto, S. (2013). Bitcoin: A Peer-to-Peer Electronic Cash System. [bitcoin.org](https://bitcoin.org/bitcoin.pdf)

[3] Voorhees, E. Conferencia Permissionless II. Presentación inicial: Why are we here? [Twitter](https://twitter.com/Permissionless/status/1702054516458156126)

[4] Hyperledger. (2016). Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. [hyperledger.org](https://www.hyperledger.org/projects/fabric)

[5] Hearn, M. & Brown, R.G. Corda: A distributed ledger. [r3.com](https://r3.com/blog/corda-technical-whitepaper/)

[6] Roets, J. Dragonchain Architecture. [dragonchain.com](https://dragonchain.com/Dragonchain-Architecture.pdf)

[7] Back, A. (2002). Hashcash - A Denial of Service Counter-Measure. [hashcash.org](http://www.hashcash.org/papers/hashcash.pdf)

[8] Meneghetti, A., Sala, M., & Taufer, D. (2020). A Survey on PoW-based Consensus. Annals of Emerging Technologies in Computing. [ResearchGate](https://www.researchgate.net/publication/338330266_A_Survey_on_PoW-based_Consensus)

[9] de Vries, A. (2020). Bitcoin's energy consumption is underestimated: A market dynamics approach. Energy Research & Social Science, 70. [DOI](https://doi.org/10.1016/j.erss.2020.101721)

[10] Pagnotta, E.S. (2021). Decentralizing Money: Bitcoin Prices and Blockchain Security. The Review of Financial Studies, 35(2). [DOI](https://doi.org/10.1093/rfs/hhaa149)

[11] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum Yellow Paper. [gavwood.com](https://gavwood.com/paper.pdf)

[12] Wang, R., Tsai, W., He, J., Liu, C., Li, Q., & Deng, E. (2018). A Medical Data Sharing Platform Based On Permissioned Blockchains. [ResearchGate](https://www.researchgate.net/publication/330895691_A_Medical_Data_Sharing_Platform_Based_On_Permissioned_Blockchains)

[13] Buterin, V. (2013). Ethereum White Paper. [Ethereum.org](https://ethereum.org/en/whitepaper/)

[14] Schweiger, P. (2021). Improving Usability of Blockchain-Based Decentralized Applications. [ResearchGate](https://www.researchgate.net/publication/352284073_Improving_Usability_of_Blockchain-Based_Decentralized_Applications)

[15] Yakovenko, A. (2019). Turbine — Solana's Block Propagation Protocol Solves the Scalability Trilemma. [Medium](https://medium.com/solana-labs/turbine-solanas-block-propagation-protocol-solves-the-scalability-trilemma-2ddba46a51db)

[16] Bedawala, M. & Wijeyekoon, A. VISA: A deep dive in Solana, a high performance blockchain network. [Visa](https://usa.visa.com/solutions/crypto/deep-dive-on-solana.html)

[17] Yakovenko, A. (2019). Gulf Stream: Solana's Mempool-less Transaction Forwarding Protocol. [Medium](https://medium.com/solana-labs/gulf-stream-solanas-mempool-less-transaction-forwarding-protocol-d342e72186ad)

[18] Boneh, D., Bonneau, J., Bünz, B., & Fisch, B. (2019). Verifiable Delay Functions. [iacr.org](https://eprint.iacr.org/2018/601.pdf)

[19] Yakovenko, A. (2018). Solana Whitepaper. Solana: A new architecture for a high performance blockchain v0.8.13. [solana.com](https://solana.com/solana-whitepaper.pdf)

[20] Solana Foundation. (2023). Solana Documentation: Synchronization. [solana.com](https://docs.solana.com/cluster/synchronization)

[21] Solana Foundation. (2023). Solana Documentation: Fork Generation. [solana.com](https://docs.solana.com/cluster/fork-generation)

[22] Solana Foundation. (2023). Solana Documentation: Managing Forks. [solana.com](https://docs.solana.com/cluster/fork-generation)

[23] Solana Foundation. (2023). Solana Documentation: Commitment. [solana.com](https://docs.solana.com/cluster/commitments)

[24] Solana Foundation. (2023). Solana Documentation: Stake Delegation and Rewards. [solana.com](https://docs.solana.com/cluster/stake-delegation-and-rewards)

[25] Solana Foundation. (2023). Solana Cookbook. Core Concepts: Account Model. [solanacookbook.com](https://solanacookbook.com/core-concepts/accounts)

[26] Alchemy. (2023). What is the Solana Account Model? [alchemy.com](https://www.alchemy.com/overviews/solana-account-model)

[27] Solana Foundation. (2023). Solana Cookbook. Core Concepts: Programs. [solanacookbook.com](https://solanacookbook.com/core-concepts/programs)

[28] Solana Foundation. (2023). Solana Cookbook. Core Concepts: Program Derivated Address(PDA). [solanacookbook.com](https://solanacookbook.com/core-concepts/pdas.html#generating-pdas)

[29] Wikipedia. Curve25519. [Wikipedia](https://en.wikipedia.org/wiki/Curve25519)

[30] Solana Foundation. (2023). Solana Documentation. Core Concepts: Calling between programs. [solana.com](https://docs.solana.com/developing/programming-model/calling-between-programs)

[31] Solana Foundation. (2023). Solana Documentation. State Compression. [solana.com](https://docs.solana.com/learn/state-compression)

[32] Extropy.IO. Solana's Wormhole Hack Post-Mortem Analysis. [Medium](https://extropy-io.medium.com/solanas-wormhole-hack-post-mortem-analysis-3b68b9e88e13)

[33] sec3. (2022). Security of Solana Smart Contracts. [sec3.dev](https://www.sec3.dev/blog/pda-bump-seeds)

[34] Solana Foundation. (2022). EWHACHAIN Bootcamp, South Korea. [GitHub](https://github.com/cleon30/Solana-Bootcam)

[35] Solana Foundation. (2023). Solana Documentation. Tokens. [solana.com](https://spl.solana.com/token)

[36] Solana Foundation(2023). Solana Documentation. Token-2022. https://spl.solana.com/token-2022

[37] Helius(2023). What is Token 2022?. https://www.helius.dev/blog/what-is-token-2022

Anexo A: Contratos Inteligentes de los Ejemplos

A1. Producir mensaje en el Ledger

- **Contrato Inteligente : Hello_world**

```rust
use anchor_lang::prelude::*;

declare_id!("A1zvqBnV7LPeCmSVg15CpFVUz8XEUh7aaGQwSB3WujgY");

#[program]
pub mod hello_world {
    use super::*;

    pub fn initialize(_ctx: Context<Initialize>) -> Result<()> {

        msg!("Hello Solana Blockchain!!");

        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize {}
```

Fuente: https://github.com/cleon30/Solana-Bootcamp/blob/main/programs/1-hello_world/src/lib.rs

A2. Contador Básico

- **Contrato Inteligente: Counter - Funciones**

```rust
use anchor_lang::prelude::*;

declare_id!("9MGFR88uofDVcjHx3hoy2jmRtcnfMdQUbYRHCFFJFuXC");

#[program]

pub mod counter {
    use super::*;

    pub fn create_counter(ctx: Context<CreateCounter>) -> Result<()> {

        msg!("Creating a Counter!!");

        let counter = &mut ctx.accounts.counter;
        counter.authority = ctx.accounts.authority.key();
        counter.count = 0;

        msg!("Current count is {}", counter.count);
        msg!("The Admin PubKey is: {} ", counter.authority);

        Ok(())
    }

    pub fn update_counter(ctx: Context<UpdateCounter>) -> Result<()> {

        msg!("Adding 1 to the counter!!");

        let counter = &mut ctx.accounts.counter;
        counter.count += 1 ;

        msg!("Current count is {}", counter.count);
        msg!("{} remaining to reach 1000 ", 1000-counter.count);

        Ok(())
    }

}

// Data validators
```

- **Contrato Inteligente: Counter - Contextos**

```rust
// Data validators

#[derive(Accounts)]
pub struct CreateCounter<'info> {
    #[account(mut)]
    authority: Signer<'info>,
    #[account(init, seeds=[authority.key().as_ref()], bump, payer=authority, space=100)]
    counter: Account<'info, Counter>,
    system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct UpdateCounter<'info> {
    authority: Signer<'info>,
    #[account(mut, has_one = authority)]
    counter: Account<'info, Counter>,
}

```

- **Contrato Inteligente: Counter - Cuentas**

```rust
// Data structures
#[account]
pub struct Counter {
    authority: Pubkey,
    count: u64,
}

```

Fuente: https://github.com/cleon30/Solana-Bootcamp/blob/main/programs/2-counter/src/lib.rs

A3. Crear una Lista Blanca mediante PDAs y CPIs

- **Contrato Inteligente Whitelist - Funciones**

```rust
use anchor_lang::prelude::*;

declare_id!("8zX4BJaBzGF8QdGJ37ATzktxzDNNze9Gg4S8CKuzKCTS");

const DISCRIMINATOR_LENGTH: usize = 8;
const PUBKEY_LENGTH: usize = 32;
const UNSIGNED64_LENGTH: usize = 8;

#[program]
pub mod whitelist {
    use super::*;

    pub fn new_whitelist(ctx: Context<NewWhitelist>) -> Result<()> {

        let whitelist = &mut ctx.accounts.whitelist;
        whitelist.authority = ctx.accounts.admin.key();

        msg!("New Whitelist created!");

        Ok(())
    }
    pub fn add_address(_ctx: Context<AddAddress>, _wallet: Pubkey) -> Result<()> {

        msg!("Adding address...");

        Ok(())
    }
    pub fn remove_address(_ctx: Context<RemoveAddress>, _wallet: Pubkey) -> Result<()> {

        msg!("Removing address...");

        Ok(())
    }
    pub fn check_address(_ctx: Context<CheckAddress>, wallet: Pubkey) -> Result<()> {

        msg!("Checking if address: {} is in the whitelist", wallet);

        Ok(())
    }


}
```

- **Contrato Inteligente Whitelist - Contextos**

```
#[derive(Accounts)]
pub struct NewWhitelist<'info> {
    #[account(init,
        payer=admin,
        space=Whitelist::LEN,
    )]
    whitelist: Account<'info, Whitelist>,

    #[account(mut)]
    admin: Signer<'info>,
    system_program: Program<'info, System>,
}
#[derive(Accounts)]
#[instruction(wallet: Pubkey)]
pub struct AddAddress<'info> {
    #[account(mut, has_one = authority)]
    whitelist: Account<'info, Whitelist>,
    #[account(init,
        payer = authority,
        space = DISCRIMINATOR_LENGTH,
        seeds = [
            whitelist.key().as_ref(),
            wallet.as_ref()
            ],
        bump,
    )]
    pda_id: Account<'info, Wallet>,
    #[account(mut)]
    authority: Signer<'info>,
    system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(wallet: Pubkey)]
pub struct RemoveAddress<'info> {
    #[account(mut, has_one = authority)]
    whitelist: Account<'info, Whitelist>,
    #[account(
        mut,
        seeds=[whitelist.key().as_ref(), wallet.as_ref()],
        bump,
        close = authority,
    )]
    pda_id: Account<'info, Wallet>,
    #[account(mut)]
    authority: Signer<'info>,
    system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(wallet: Pubkey)]
pub struct CheckAddress<'info> {
    whitelist: Account<'info, Whitelist>,
    #[account(
        seeds = [
            whitelist.key().as_ref(),
            wallet.key().as_ref()
            ],
        bump,
    )]
    pda_id: Account<'info, Wallet>,
}

```

- **Contrato Inteligente Whitelist - Cuentas**

```
#[account]
pub struct Whitelist {
    pub authority: Pubkey,
}

impl Whitelist {
    const LEN: usize = DISCRIMINATOR_LENGTH + PUBKEY_LENGTH + UNSIGNED64_LENGTH;
}

#[account]
pub struct Wallet {}
```

- **Contrato Inteligente Counter – Funciones**

```rust
use anchor_lang::prelude::*;
use whitelist::cpi::accounts::{
    AddAddress, CheckAddress, NewWhitelist, RemoveAddress,
};
use whitelist::program::Whitelist;
use whitelist::{self};
declare_id!("FPsYj63RbFJS3EeN4zN1zYzBrSKfphjuiyjYjmFfnwsW");

const DISCRIMINATOR_LENGTH: usize = 8;
const PUBKEY_LENGTH: usize = 32;
const UNSIGNED64_LENGTH: usize = 8;

#[program]
pub mod counter {
    use super::*;

    pub fn init_counter(ctx: Context<InitCounter>, bump:u8) -> Result<()> {

        msg!("Initializing counter account...");

        let counter = &mut ctx.accounts.counter;
        counter.authority = ctx.accounts.authority.key();
        counter.count = 0;
        counter.bump = bump;
        counter.program = Pubkey::default();
        counter.initialized = false;

        msg!("Authority set to ...{}",ctx.accounts.authority.key());

        Ok(())
    }
    pub fn point_to_whitelist(ctx: Context<PointWhitelist>) -> Result<()> {

        let counter = &mut ctx.accounts.counter;
        assert!(counter.initialized == false);

        counter.program = ctx.accounts.whitelisting.key();
        counter.initialized = true;

        msg!("Pointing Counter to a New Whitelist...{}");

            let pointer_ctx = ctx.accounts.whitelist_ctx();
            let _ = whitelist::cpi::new_whitelist(pointer_ctx);


        Ok(())
    }

```

```rust
    pub fn add_or_remove(ctx: Context<AddOrRemove>, wallet: Pubkey, remove:bool) -> Result<()> {

        let counter = &mut ctx.accounts.counter;
        assert!(counter.initialized ==true);

        let admin = ctx.accounts.authority.key();
        let count = counter.count;
        let seeds= ["counter".as_bytes().as_ref(), admin.as_ref(), &[counter.bump]];
        let signer = &[&seeds[..]];

        match remove{

            true => {

                msg!("Removing process started...");

                    msg!("Checking that the address actually exist in the whitelist");

                    let check_ctx = ctx.accounts.check_ctx().with_signer(signer);
                    match whitelist::cpi::check_address(check_ctx, wallet.key()){

                        Ok(()) =>{
                                    let remove_ctx = ctx.accounts.remove_wallet_ctx().with_signer(signer);
                                    let _ = whitelist::cpi::remove_address(remove_ctx, wallet);

                                    msg!("-1 to the counter...");

                                        ctx.accounts.counter.count = count.checked_sub(1).unwrap();

                                    Ok(())
                                }

                        Err(e) => Err(e),

                    }
            },

            false => {

                    msg!("Adding process started...");

                        let add_ctx = ctx.accounts.adding_new_wallet_ctx().with_signer(signer);
                        let _ = whitelist::cpi::add_address(add_ctx, wallet);

                    msg!("+1 to the counter...");

                        ctx.accounts.counter.count = count.checked_add(1).unwrap();

                    Ok(())

            }

        }
    }
}
```

- **Contrato Inteligente Counter - Contextos**

```
106  // Data validators
107
108  #[derive(Accounts)]
109  pub struct InitCounter<'info> {
110
111      #[account(
112          init,
113          seeds =[
114                  "counter".as_bytes().as_ref(),
115                  authority.key().as_ref()
116                  ],
117          bump,
118          payer=authority,
119          space=Counter::LEN)
120      ]
121      counter: Account<'info, Counter>,
122      #[account(mut)]
123      authority: Signer<'info>,
124      system_program: Program<'info, System>,
125  }
126  #[derive(Accounts)]
127  pub struct PointWhitelist<'info> {
128
129      #[account(
130          mut,
131          seeds = [
132                  "counter".as_bytes().as_ref(),
133                  authority.key().as_ref()
134                  ],
135          bump,
136          has_one = authority,
137      )]
138      counter: Account<'info, Counter>,
139      pointer_to_whitelist_id: Program<'info, Whitelist>,
140      #[account(mut, signer)]
141      ///CHECK : Error checkpoint: if not they say that is not initialised
142      whitelisting: AccountInfo<'info>,
143      #[account(mut)]
144      authority: Signer<'info>,
145      system_program: Program<'info, System>,
146  }
147
148  #[derive(Accounts)]
149  pub struct AddOrRemove<'info> {
150
151      #[account(
152          mut,
153          seeds = [
154                  "counter".as_bytes().as_ref(),
155                  authority.key().as_ref()
156                  ],
157                  has_one = authority,
158          bump,
159          )]
160      counter: Account<'info, Counter>,
161      #[account(mut)]
162      ///CHECK
163      pda_id: AccountInfo<'info>,
164      #[account(mut)]
165      ///CHECK
166      whitelisting: AccountInfo<'info>,
167      update_id: Program<'info, Whitelist>,
168      #[account(mut)]
169      authority: Signer<'info>,
170      system_program: Program<'info, System>,
171  }
172
```

Contrato Inteligente Counter – Cuentas y Implementaciones

```rust
#[account]
pub struct Counter {
    authority: Pubkey,
    count: u64,
    bump:u8,
    program:Pubkey,
    initialized:bool,
}

// Implementations of Context
impl<'info> PointWhitelist<'info> {
    pub fn whitelist_ctx(&self,) -> CpiContext<'_, '_, '_, 'info, NewWhitelist<'info>> {

        let pointer_to_whitelist_id = self.pointer_to_whitelist_id.to_account_info();

        let init_account = NewWhitelist{
                                    admin: self.authority.to_account_info(),
                                    whitelist: self.whitelisting.to_account_info(),
                                    system_program: self.system_program.to_account_info(),
                                    };

        CpiContext::new(pointer_to_whitelist_id, init_account)
    }
}

impl<'info> AddOrRemove<'info> {
    pub fn remove_wallet_ctx(&self,) -> CpiContext<'_, '_, '_, 'info, RemoveAddress<'info>> {

        let update_id = self.update_id.to_account_info();

        let update_account = RemoveAddress{
                                        authority: self.authority.to_account_info(),
                                        whitelist: self.whitelisting.to_account_info(),
                                        pda_id:self.pda_id.to_account_info(),
                                        system_program: self.system_program.to_account_info(),
                                        };

        CpiContext::new(update_id, update_account)
    }
    pub fn adding_new_wallet_ctx(&self,) -> CpiContext<'_, '_, '_, 'info, AddAddress<'info>> {

        let update_id = self.update_id.to_account_info();

        let update_account = AddAddress{
                                    authority: self.authority.to_account_info(),
                                    whitelist: self.whitelisting.to_account_info(),
                                    pda_id:self.pda_id.to_account_info(),
                                    system_program: self.system_program.to_account_info(),
                                    };

        CpiContext::new(update_id, update_account)
    }
    pub fn check_ctx(&self,) -> CpiContext<'_, '_, '_, 'info, CheckAddress<'info>> {

        let update_id = self.update_id.to_account_info();

        let update_account = CheckAddress{
                                        whitelist: self.whitelisting.to_account_info(),
                                        pda_id:self.pda_id.to_account_info(),
                                    };

        CpiContext::new(update_id, update_account)
    }
}

impl Counter {
    const LEN: usize = (DISCRIMINATOR_LENGTH*2 + PUBKEY_LENGTH *2 + UNSIGNED64_LENGTH *2);
}
```

Fuente: https://github.com/cleon30/solana-labs-PDA-Whitelisting

Anexo B: Testing de los Contratos Inteligentes

B1. Producir mensaje en el Ledger

- **Test del Programa**

```
import * as anchor from "@project-serum/anchor";
import { Program } from "@project-serum/anchor";
import { HelloWorld } from "../target/types/hello_world";

describe("hello-world", () => {
  // Configure the client to use the local cluster.
  anchor.setProvider(anchor.AnchorProvider.env());

  const program = anchor.workspace.HelloWorld as Program<HelloWorld>;

  it("Is initialized!", async () => {
    // Add your test here.
    const tx = await program.methods.initialize().rpc();
    console.log("Your transaction signature", tx);
  });
});
```

Fuente: https://github.com/cleon30/Solana-Bootcamp/blob/main/tests/hello_world.ts

B2. Contador Básico

- **Test del Programa**

```

import * as anchor from "@project-serum/anchor";
import { Program } from "@project-serum/anchor";
import { Counter } from "../target/types/counter";

describe("counter", () => {
  // Configure the client to use the local cluster.
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);
  const wallet = provider.wallet;
  const systemProgram = anchor.web3.SystemProgram;
  const program = anchor.workspace.Counter as Program<Counter>;

  it("Create Counter!", async () => {
    // Keypair = account
    const [counter, _counterBump] =
      await anchor.web3.PublicKey.findProgramAddress(
        [wallet.publicKey.toBytes()],
        program.programId
      );
    console.log("Your counter address", counter.toString());
    const tx = await program.methods
      .createCounter()
      .accounts({
        authority: wallet.publicKey,
        counter: counter,
        systemProgram: systemProgram.programId,
      })
      .rpc();
    console.log("Your transaction signature", tx);
  });
```

```
33    it("Fetch a counter!", async () => {
34      // Keypair = account
35      const [counterPubkey, _] = await anchor.web3.PublicKey.findProgramAddress(
36        [wallet.publicKey.toBytes()],
37        program.programId
38      );
39      console.log("Your counter address", counterPubkey.toString());
40      const counter = await program.account.counter.fetch(counterPubkey);
41      console.log("Your counter", counter);
42    });
43
44    it("Update a counter!", async () => {
45      // Keypair = account
46      const [counterPubkey, _] = await anchor.web3.PublicKey.findProgramAddress(
47        [wallet.publicKey.toBytes()],
48        program.programId
49      );
50      console.log("Your counter address", counterPubkey.toString());
51      const counter = await program.account.counter.fetch(counterPubkey);
52      console.log("Your counter", counter);
53      const tx = await program.methods
54        .updateCounter()
55        .accounts({
56          counter: counterPubkey,
57        })
58        .rpc();
59      console.log("Your transaction signature", tx);
60      const counterUpdated = await program.account.counter.fetch(counterPubkey);
61      console.log("Your counter", counterUpdated);
62    });
63  });
```

Fuente: https://github.com/cleon30/Solana-Bootcamp/blob/main/tests/counter.ts

B3. Crear una Lista Blanca mediante PDAs y CPIs

- **Test del Programa**

```
1
2   import * as anchor from "@project-serum/anchor";
3   import { Program } from "@project-serum/anchor";
4   import { assert, expect } from "chai";
5   import { Counter } from "../target/types/counter";
6   import { Whitelist } from "../target/types/whitelist";
7   import chai from "chai";
8   import { waitForDebugger } from "inspector";
9   async function airdrop(connection, destinationWallet, amount) {
10    const airdropSignature = await connection.requestAirdrop(destinationWallet.publicKey,
11      amount * anchor.web3.LAMPORTS_PER_SOL);
12
13    const latestBlockHash = await connection.getLatestBlockhash();
14
15    const tx = await connection.confirmTransaction({
16      blockhash: latestBlockHash.blockhash,
17      lastValidBlockHeight: latestBlockHash.lastValidBlockHeight,
18      signature: airdropSignature
19    });
20  }
21    describe("counter",() => {
22
23      const provider = anchor.AnchorProvider.env();
24      anchor.setProvider(provider);
25
26      const LAMPORTS_PER_SOL = 1000000000;
27
28      const CounterProgram = anchor.workspace.Counter as Program<Counter>;
29      const WhitelistProgram = anchor.workspace.Whitelist as Program<Whitelist>;
30
31
32      let authority = anchor.web3.Keypair.generate();
33      let authority_fake = anchor.web3.Keypair.generate();
34      let whitelist = anchor.web3.Keypair.generate();
35      let counterPDA: anchor.web3.PublicKey;
36      let counterBump: number;
37      let account1 = anchor.web3.Keypair.generate();
38      let account1PDA: anchor.web3.PublicKey;
39      let account2 = anchor.web3.Keypair.generate();
40      let account2PDA: anchor.web3.PublicKey;
41      let account3 = anchor.web3.Keypair.generate();
42      let account3PDA: anchor.web3.PublicKey;
43      let number_of_accounts = 5;
44      let airdrop_number =2;
45      // let cleon_theory_time = number_of_accounts/12;
46      let balance_before = CounterProgram.provider.connection.getBalance(authority.publicKey);
47      let whitelistedAccounts: Array<anchor.web3.Keypair> = [];
48      for(let i = 0; i < number_of_accounts; ++i)
49      {
50        whitelistedAccounts.push(anchor.web3.Keypair.generate());
51      }
52
53
54      // before(async () => {
55      //     // Top up all acounts that will need lamports for account creation
56      //     await airdrop(provider.connection, authority,2);
57
58      // });
59
60      it("Creating a counter", async () => {
61          await airdrop(provider.connection, authority,2);
62          // await airdrop(provider.connection, authority,2);
63          console.clear();
64          console.log("\nAirdrop to authority complete!");
65          let initial = await CounterProgram.provider.connection.getBalance(authority.publicKey);
66          console.log("Initial SOL Balance:",`${initial / LAMPORTS_PER_SOL} SOL`);
67          [counterPDA, counterBump]= await anchor.web3.PublicKey.findProgramAddress(
68          [Buffer.from(anchor.utils.bytes.utf8.encode("counter")), authority.publicKey.toBuffer()],
69          CounterProgram.programId
70          );
71
72          await CounterProgram.methods
73            .initCounter(counterBump)
74            .accounts({
75              authority: authority.publicKey,
76              counter: counterPDA,
77              systemProgram: anchor.web3.SystemProgram.programId,
78            })
79            .signers([authority])
80            .rpc();
81          let counter = await CounterProgram.account.counter.fetch(counterPDA);
82          console.log("Your counter PublicKey: ",counterPDA.toString());
83          assert.equal(counter.count.toNumber(), 0);
84          // console.log(counterPDA, counterBump);
85
86      });
87
88      it("Point the Counter to an associated Whitelist", async () => {
89
90        await CounterProgram.methods
91          .pointToWhitelist()
92          .accounts({
93            authority: authority.publicKey,
94            counter: counterPDA,
95            whitelisting: whitelist.publicKey,
96            pointerToWhitelistId: WhitelistProgram.programId,
97            systemProgram: anchor.web3.SystemProgram.programId,
98          })
99          .signers([authority, whitelist])
100         .rpc();
101     });
102
103
```

```
103
104     it("Add multiple users", async() =>{
105       for(let i = 0; i < whitelistedAccounts.length ; ++i) {
106           let wallet = whitelistedAccounts[i];
107           let [PDA, _] = await anchor.web3.PublicKey.findProgramAddress(
108             [whitelist.publicKey.toBuffer(), wallet.publicKey.toBuffer()],
109             WhitelistProgram.programId
110           );
111       // console.log(counterPDA);
112           await CounterProgram.methods
113           .addOrRemove(wallet.publicKey, false)
114           .accounts({
115             authority: authority.publicKey,
116             counter: counterPDA,
117             pdaId: PDA,
118             whitelisting: whitelist.publicKey,
119             updateId: WhitelistProgram.programId,
120             systemProgram: anchor.web3.SystemProgram.programId,
121           })
122           .signers([authority])
123           .rpc();
124           console.log("Adding Address", wallet.publicKey.toString(),"to the whitelist");
125       }
126       // await new Promise(f => setTimeout(f, cleon_theory_time *1000));
127       let counter = await CounterProgram.account.counter.fetch(counterPDA);
128       console.log("The current count is:", counter.count.toNumber());
129     })
130
131     it("Check wallet actually is in whitelist", async () => {
132       let account2 = whitelistedAccounts[1];
133       [account2PDA,] = await anchor.web3.PublicKey.findProgramAddress(
134           [whitelist.publicKey.toBuffer(), account2.publicKey.toBuffer()],
135           WhitelistProgram.programId
136       );
137       await WhitelistProgram.methods
138       .checkAddress(account2.publicKey)
139       .accounts({
140         whitelist: whitelist.publicKey,
141         pdaId: account2PDA,
142       })
143       .signers([])
144       .rpc();
145       chai.assert(true);
146
147     });
148
149
150       it("Deletes 50% of whitelisted accounts from the whitelist ", async() =>{
151
152         let initBalance = await CounterProgram.provider.connection.getBalance(authority.publicKey);
153         for(let i = 0; i < whitelistedAccounts.length/2 ; ++i) {
154             let wallet = whitelistedAccounts[i];
155             let [PDA, _] = await anchor.web3.PublicKey.findProgramAddress(
156               [whitelist.publicKey.toBuffer(), wallet.publicKey.toBuffer()],
157               WhitelistProgram.programId
158             );
159
160
161                 await CounterProgram.methods
162                 .addOrRemove(wallet.publicKey, true)
163                 .accounts({
164                 authority: authority.publicKey,
165                 counter: counterPDA,
166                 pdaId: PDA,
167                 whitelisting: whitelist.publicKey,
168                 updateId: WhitelistProgram.programId,
169                 systemProgram: anchor.web3.SystemProgram.programId,
170                 })
171                 .signers([authority])
172                 .rpc();
173                 let counter = await CounterProgram.account.counter.fetch(counterPDA);
174                 console.log("The account:", wallet.publicKey.toString(), "has been deleted from the whitelist")//," | total count:",counter.count.toNumber());
175
176         }
177
178         let counter = await CounterProgram.account.counter.fetch(counterPDA);
179         let endBalance = await CounterProgram.provider.connection.getBalance(authority.publicKey);
180         expect(endBalance).to.be.greaterThan(initBalance);
181         console.log("The current count is ", counter.count.toNumber())
182         console.log("The end balance of authority is:",`${endBalance / LAMPORTS_PER_SOL} SOL`, "vs authority init balance:",`${initBalance / LAMPORTS_PER_SOL} SOL`);
183
184       })
185
186     it("Check if deleted wallet actually is in whitelist", async () => {
187       let account1 = whitelistedAccounts[1];
188       [account1PDA,] = await anchor.web3.PublicKey.findProgramAddress(
189           [whitelist.publicKey.toBuffer(), account1.publicKey.toBuffer()],
190           WhitelistProgram.programId
191         );
192       try{
193           await WhitelistProgram.methods
194           .checkAddress(account1.publicKey)
195           .accounts({
196               whitelist: whitelist.publicKey,
197               pdaId: account1PDA,
198           })
199           .signers([])
200           .rpc();
201       }catch(e){
202           console.log("✗ Error, this account does not belongs to whitelist");
203       }
204     })
```

```
205     it("An user tries to add people without being admin ❌", async () => {
206       let account1 = whitelistedAccounts[1];
207       [account1PDA,] = await anchor.web3.PublicKey.findProgramAddress(
208           [whitelist.publicKey.toBuffer(), account1.publicKey.toBuffer()],
209           WhitelistProgram.programId
210         );
211       let wallet = whitelistedAccounts[0];
212       let [PDA, _] = await anchor.web3.PublicKey.findProgramAddress(
213           [whitelist.publicKey.toBuffer(), wallet.publicKey.toBuffer()],
214           WhitelistProgram.programId
215       );
216       try{
217           await CounterProgram.methods
218           .addOrRemove(account1.publicKey, false)
219           .accounts({
220             authority: authority.publicKey,
221             counter: counterPDA,
222             pdaId: account1PDA,
223             whitelisting: whitelist.publicKey,
224             updateId: WhitelistProgram.programId,
225             systemProgram: anchor.web3.SystemProgram.programId,
226           })
227           .signers([wallet])
228           .rpc();
229
230       }catch(e){
231           console.log("❌ Error, this account is not the admin authority");
232       }
233     })
234     it("An user tries to remove people without being admin ❌", async () => {
235       let init_counter = await CounterProgram.account.counter.fetch(counterPDA);
236       let account1 = whitelistedAccounts[1];
237       [account1PDA,] = await anchor.web3.PublicKey.findProgramAddress(
238           [whitelist.publicKey.toBuffer(), account1.publicKey.toBuffer()],
239           WhitelistProgram.programId
240         );
241       let wallet = whitelistedAccounts[0];
242       let [PDA, _] = await anchor.web3.PublicKey.findProgramAddress(
243           [whitelist.publicKey.toBuffer(), wallet.publicKey.toBuffer()],
244           WhitelistProgram.programId
245       );
246       try{
247           await CounterProgram.methods
248           .addOrRemove(account1.publicKey, true)
249           .accounts({
250             authority: authority.publicKey,
251             counter: counterPDA,
252             pdaId: account1PDA,
253             whitelisting: whitelist.publicKey,
254             updateId: WhitelistProgram.programId,
255             systemProgram: anchor.web3.SystemProgram.programId,
256           })
257           .signers([wallet])
258           .rpc();
259
260       }catch(e){
261           console.log("❌ Error, this account is not the admin authority");
262       }
263       let final_counter = await CounterProgram.account.counter.fetch(counterPDA);
264       assert.equal(final_counter.count.toNumber(),init_counter.count.toNumber());
265     })
266     it("Retrieve all the money! 💰💰💰 ❌", async() =>{
267
268       let initBalance = await CounterProgram.provider.connection.getBalance(authority.publicKey);
269       for(let i = 0; i < whitelistedAccounts.length ; ++i) {
270           let wallet = whitelistedAccounts[i];
271           let [PDA, _] = await anchor.web3.PublicKey.findProgramAddress(
272             [whitelist.publicKey.toBuffer(), wallet.publicKey.toBuffer()],
273             WhitelistProgram.programId
274           );
275
276           try{
277               await CounterProgram.methods
278               .addOrRemove(wallet.publicKey, true)
279               .accounts({
280               authority: authority.publicKey,
281               counter: counterPDA,
282               pdaId: PDA,
283               whitelisting: whitelist.publicKey,
284               updateId: WhitelistProgram.programId,
285               systemProgram: anchor.web3.SystemProgram.programId,
286               })
287               .signers([authority])
288               .rpc();
289
290               console.log("The account:", wallet.publicKey.toString(), "has been deleted from the whitelist")//," | total count;",counter.count.toNumber());
291             } catch(e){
292               console.error("The program expected this account to be already initialized");
293             }
294       }
295       // await new Promise(f => setTimeout(f, cleon_theory_time *1000));
296       let counter = await CounterProgram.account.counter.fetch(counterPDA);
297       let endBalance = await CounterProgram.provider.connection.getBalance(authority.publicKey);
298       expect(endBalance).to.be.greaterThan(initBalance);
299       console.log("The current count is ", counter.count.toNumber())
300       console.log("The end balance of authority is:",`${endBalance / LAMPORTS_PER_SOL} SOL`, "vs authority init balance:",`${initBalance / LAMPORTS_PER_SOL} SOL`);
301     })
302
303     after(async () =>{
304       let counter = await CounterProgram.account.counter.fetch(counterPDA);
305       let init = (await balance_before / LAMPORTS_PER_SOL) + airdrop_number;
306       let endBalance = await CounterProgram.provider.connection.getBalance(authority.publicKey);
307       let sol_spend = init - endBalance/LAMPORTS_PER_SOL;
308
309       console.log("\n---------------");
310       console.log("Statistics📊📝\n");
311       console.log("Initial SOL Balance:", init);
312       console.log("Balance after the Whitelisting:", endBalance/LAMPORTS_PER_SOL)
313       console.log("Counter Total:", counter.count.toNumber());
314       console.log("Total SOL Spent:", sol_spend);
```

Fuente:https://github.com/cleon30/solana-labs-PDA-Whitelisting/blob/main/Solana-Smart-Contract/tests/counter.ts

- **Output del Test del Programa Counter**

```
Airdrop to authority complete!
Initial SOL Balance: 2 SOL
Your counter PublicKey:  H9dJ1zgTybNvr5ocmzs4CWGjQdNAx6jGHpmAMzPdaYp2
    ✔ Creating a counter📝 (742ms)
    ✔ Point the Counter to an associated Whitelist (467ms)
Adding Address DDbypng3tc4JN4PfuCaNzVLTBvqwN4iWm8oVenjASkMJ to the whitelist
Adding Address 5qFhv7oj6SPLnJNvaZDWSnJzBPN6s8pZCLaSyre9S53h to the whitelist
Adding Address 4HmeoF4cD21mY7VChQyecLCKTZBk2kxkqajf5f1eJyCs to the whitelist
Adding Address D7G9GG9WceRJjxWQdCzYwuxqmFPmtfjaAMBGrMURqdWr to the whitelist
Adding Address BzmUJ4FvNa5FetPry6qSPZDdPtJ4zchYgUvSSiMXLwL5 to the whitelist
The current count is: 5
    ✔ Add multiple users (2379ms)
    ✔ Check wallet actually is in whitelist (473ms)
The account: DDbypng3tc4JN4PfuCaNzVLTBvqwN4iWm8oVenjASkMJ has been deleted from the whitelist
The account: 5qFhv7oj6SPLnJNvaZDWSnJzBPN6s8pZCLaSyre9S53h has been deleted from the whitelist
The account: 4HmeoF4cD21mY7VChQyecLCKTZBk2kxkqajf5f1eJyCs has been deleted from the whitelist
The current count is  2
The end balance of authority is: 1.99532288 SOL vs authority init balance: 1.9924832 SOL
    ✔ Deletes 50% of whitelisted accounts from the whitelist 📝✘ (1446ms)
✘ Error, this account does not belongs to whitelist
    ✔ Check if deleted wallet actually is in whitelist🙂📝✘
✘ Error, this account is not the admin authority
    ✔ An user tries to add people without being admin 📝✘
✘ Error, this account is not the admin authority
    ✔ An user tries to remove people without being admin 📝✘
The program expected this account to be already initialized
The program expected this account to be already initialized
The program expected this account to be already initialized
The account: D7G9GG9WceRJjxWQdCzYwuxqmFPmtfjaAMBGrMURqdWr has been deleted from the whitelist
The account: BzmUJ4FvNa5FetPry6qSPZDdPtJ4zchYgUvSSiMXLwL5 has been deleted from the whitelist
The current count is  0
The end balance of authority is: 1.997216 SOL vs authority init balance: 1.99532288 SOL
    ✔ Retrieve all the money! 💰💰💰✘ (933ms)

---------------
Statistics📊📈

Initial SOL Balance: 2
Balance after the Whitelisting: 1.997216
Counter Total: 0
Total SOL Spent: 0.0027839999999998977
The counter count is 0, your remaining SOL is  0.0027839999999998977


  9 passing (6s)

✨  Done in 7.55s.
```

Anexo C: Command Line Interface(CLI)

```
solana-cli 1.16.12 (src:2168053d; feat:3949673676, client:SolanaLabs)
Blockchain, Rebuilt for Scale

USAGE:
    solana [FLAGS] [OPTIONS] <SUBCOMMAND>

FLAGS:
    -h, --help                          Prints help information
        --no-address-labels             Do not use address labels in the output
        --skip-seed-phrase-validation   Skip validation of seed phrases. Use this if your phrase does not use the BIP39
                                        official English word list
        --use-quic                      Use QUIC when sending transactions.
        --use-udp                       Use UDP when sending transactions.
    -V, --version                       Prints version information
    -v, --verbose                       Show additional information

OPTIONS:
        --commitment <COMMITMENT_LEVEL>   Return information at the selected commitment level [possible values:
                                          processed, confirmed, finalized]
    -C, --config <FILEPATH>               Configuration file to use [default:
                                          /Users/cleon/.config/solana/cli/config.yml]
    -u, --url <URL_OR_MONIKER>            URL for Solana's JSON RPC or moniker (or their first letter): [mainnet-beta,
                                          testnet, devnet, localhost]
    -k, --keypair <KEYPAIR>               Filepath or URL to a keypair
        --output <FORMAT>                 Return information in specified output format [possible values: json, json-
                                          compact]
        --ws <URL>                        WebSocket URL for the solana cluster

SUBCOMMANDS:
    account                             Show the contents of an account
    address                             Get your public key
    address-lookup-table                Address lookup table management
    airdrop                             Request SOL from a faucet
    authorize-nonce-account             Assign account authority to a new entity
    balance                             Get your balance
    block                               Get a confirmed block
    block-height                        Get current block height
    block-production                    Show information about block production
    block-time                          Get estimated production time of a block
    catchup                             Wait for a validator to catch up to the cluster
    close-vote-account                  Close a vote account and withdraw all funds remaining
    cluster-date                        Get current cluster date, computed from genesis creation time and network
                                        time
    cluster-version                     Get the version of the cluster entrypoint
    completion                          Generate completion scripts for various shells
    config                              Solana command-line tool configuration settings
    confirm                             Confirm transaction by signature
    create-address-with-seed            Generate a derived account address with a seed. For program derived
                                        addresses (PDAs), use the find-program-derived-address command instead
    create-nonce-account                Create a nonce account
    create-stake-account                Create a stake account
    create-stake-account-checked        Create a stake account, checking the withdraw authority as a signer
    create-vote-account                 Create a vote account
    deactivate-stake                    Deactivate the delegated stake from the stake account
    decode-transaction                  Decode a serialized transaction
    delegate-stake                      Delegate stake to a vote account
    epoch                               Get current epoch
    epoch-info                          Get information about the current epoch
    feature                             Runtime feature management
    fees                                Display current cluster fees (Deprecated in v1.8.0)
    find-program-derived-address        Generate a program derived account address with a seed
    first-available-block               Get the first available block in the storage
    genesis-hash                        Get the genesis hash
    gossip                              Show the current gossip network nodes
    help                                Prints this message or the help of the given subcommand(s)
    inflation                           Show inflation information
    largest-accounts                    Get addresses of largest cluster accounts
    leader-schedule                     Display leader schedule
    live-slots                          Show information about the current slot progression
    logs                                Stream transaction logs
    merge-stake                         Merges one stake account into another
    new-nonce                           Generate a new nonce, rendering the existing nonce useless
    nonce                               Get the current nonce value
    nonce-account                       Show the contents of a nonce account
    ping                                Submit transactions sequentially
    program                             Program management
    redelegate-stake                    Redelegate active stake to another vote account
    rent                                Calculate rent-exempt-minimum value for a given account data field length.
    resolve-signer                      Checks that a signer is valid, and returns its specific path; useful for
                                        signers that may be specified generally, eg. usb://ledger
    sign-offchain-message               Sign off-chain message
    slot                                Get current slot
    split-stake                         Duplicate a stake account, splitting the tokens between the two
    stake-account                       Show the contents of a stake account
    stake-authorize                     Authorize a new signing keypair for the given stake account
    stake-authorize-checked             Authorize a new signing keypair for the given stake account, checking the
                                        authority as a signer
    stake-history                       Show the stake history
    stake-minimum-delegation            Get the stake minimum delegation amount
    stake-set-lockup                    Set Lockup for the stake account
    stake-set-lockup-checked            Set Lockup for the stake account, checking the new authority as a signer
    stakes                              Show stake account information
    supply                              Get information about the cluster supply of SOL
    transaction-count                   Get current transaction count
    transaction-history                 Show historical transactions affecting the given address from newest to
                                        oldest
    transfer                            Transfer funds between system accounts
    upgrade-nonce-account               One-time idempotent upgrade of legacy nonce versions in order to bump them
                                        out of chain blockhash domain.
    validator-info                      Publish/get Validator info on Solana
    validators                          Show summary information about the current validators
    verify-offchain-signature           Verify off-chain message signature
    vote-account                        Show the contents of a vote account
    vote-authorize-voter                Authorize a new vote signing keypair for the given vote account
    vote-authorize-voter-checked        Authorize a new vote signing keypair for the given vote account, checking
                                        the new authority as a signer
    vote-authorize-withdrawer           Authorize a new withdraw signing keypair for the given vote account
    vote-authorize-withdrawer-checked   Authorize a new withdraw signing keypair for the given vote account,
                                        checking the new authority as a signer
    vote-update-commission              Update the vote account's commission
    vote-update-validator               Update the vote account's validator identity
    wait-for-max-stake                  Wait for the max stake of any one node to drop below a percentage of total.
    withdraw-from-nonce-account         Withdraw SOL from the nonce account
    withdraw-from-vote-account          Withdraw lamports from a vote account into a specified account
    withdraw-stake                      Withdraw the unstaked SOL from the stake account
```