



# UCA

---

Pontificia Universidad Católica Argentina

**Tesis de grado  
Ingeniería Informática**

## **Rollups de Confianza Cero (Zero Trust Rollups)**

**Ejecución delegada offchain con retribución de  
resultados onchain**

**Alumno: Matías Ratcliffe**

**DNI: 39.694.694**

**Legajo: 15-152236-7**

**Tutor: Ignacio Nicolás Parravicini**

**Año: 2024**

## Índice de contenidos

1	Introducción .....	7
2	Motivación.....	8
3	Objetivo.....	9
4	Estado del arte.....	10
4.1	Procesamiento On-Premises .....	10
4.2	Cloud Computing .....	11
4.3	Volunteer Computing .....	12
4.3.1	BOINC .....	13
4.4	Procesamiento en Blockchain .....	15
4.4.1	Estructura de una blockchain .....	16
4.4.2	Interacción con la red .....	17
4.4.3	Algoritmos de consenso .....	18
4.4.4	Ethereum .....	20
4.4.5	Rollups: una solución de escalabilidad .....	24
5	Desarrollo y Experimentación .....	31
5.1	Primera Solución: Optimistic Execution .....	31
5.1.1	Propuesta .....	31
5.1.2	Funcionamiento y características .....	32
5.1.3	Estructura de los Contratos .....	37
5.1.4	Ciclo de vida de una petición.....	42
5.1.5	Ejemplo de uso: Números primos.....	44
5.1.6	Pros y Contras.....	55
5.2	Segunda Solución: Open Chain Computing.....	56
5.2.1	Propuesta .....	56

5.2.2	Funcionamiento y Características .....	57
5.2.3	Estructura de los Contratos .....	65
5.2.4	Ciclo de vida de una petición.....	69
5.2.5	Ejemplo de uso.....	72
5.2.6	Pros y Contras.....	85
5.3	Tercera Solución: Zero Trust Rollups.....	85
5.3.1	Propuesta .....	86
5.3.2	Funcionamiento y características .....	86
5.3.3	Estructura de los Contratos .....	88
5.3.4	Ciclo de vida de una petición.....	91
5.3.5	Ejemplos de uso .....	92
5.3.6	Pros y Contras.....	105
5.4	Análisis comparativo .....	106
6	Conclusión .....	109
7	Futuras Discusiones .....	111
8	Referencias.....	112
9	Anexos .....	116

## Índice de figuras

FIGURA I: Ciclo de vida de una petición en Optimistic Execution.....	43
FIGURA II: código en solidity para la búsqueda de números primos.....	45
FIGURA III: código en python para testear costos de ejecución onchain.....	46
FIGURA IV: impresión de la ejecución del test de la Figura III.....	46
FIGURA V: código en solidity para la búsqueda de números primos de forma delegada. .....	47
FIGURA VI: código en python para testear costos de ejecución optimista delegada....	48
FIGURA VII: impresión de la ejecución del test de la Figura VI. ....	48
FIGURA VIII: lógica de un contrato cliente trivial. ....	51
FIGURA IX: impresión de la ejecución del test de la Figura VI, con el contrato de la Figura VIII.....	51
FIGURA X: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del protocolo Optimistic Execution para distintos valores de mercado de gas virtual.....	53
FIGURA XI: Resultado obtenido de correr el test de la Figura VI, para una petición con un tamaño de respuesta de 1KB.....	54
FIGURA XII: Ciclo de vida de una petición en Open Chain Computing. ....	70
FIGURA XIII: código en python del test de costos de una petición trivial en OCC.....	74
FIGURA XIV: impresión de la ejecución del test de la Figura XIII.....	74
FIGURA XV: Gráfico del costo de registro de un ejecutor por la cantidad de ejecutores ya presentes.....	77
FIGURA XVI: Gráfico del costo de creación de una petición por la cantidad de ejecutores contratados.....	78
FIGURA XVII: Gráfico del costo de publicación inicial de resultados por la cantidad de ejecutores contratados.....	79
FIGURA XVIII: Gráfico del costo de publicación final de resultados por la cantidad de ejecutores contratados.....	80
FIGURA XIX: Gráfico del costo de liberación inicial de resultados por el tamaño de los resultados liberados.....	81

FIGURA XX: Gráfico en R3 del costo liberación final de resultados por la cantidad de ejecutores contratados y el tamaño de los resultados liberados.....	82
FIGURA XXI: Gráfico del costo añadido total por petición por cantidad de ejecutores.	83
FIGURA XXII: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del protocolo Open Chain Computing para distintos valores de poder de ejecución.....	84
FIGURA XXIII: Ciclo de vida de una petición en Zero Trust Rollups.....	92
FIGURA XXIV: código en solidity para el minado onchain y la comprobación de fragmentos de una clave en TS3000 (basado en el contrato cliente). .....	96
FIGURA XXV: código en python del test de costos de una petición de 5 fragmentos de dificultad 3 en ejecución onchain. ....	96
FIGURA XXVI: impresión de la ejecución del test de la Figura XXV.....	97
FIGURA XXVII: código en python del test de costos de una petición de 5 fragmentos de dificultad 3 en el ZTR de TS3000.....	98
FIGURA XXVIII: impresión de la ejecución del test de la Figura XXVII.....	98
FIGURA XXIX: código en solidity para el minado onchain de fragmentos de una clave en TS3000 (versión simplificada). .....	100
FIGURA XXX: código en python del test de costos de una petición de 30 fragmentos de dificultad 4 en la versión simplificada onchain de TS3000. ....	100
FIGURA XXXI: impresión de la ejecución del test de la Figura XXX.....	101
FIGURA XXXII: impresión de la ejecución del test de la Figura XXVII (adaptado) para una petición de 30 fragmentos de dificultad 4 en el ZTR de TS3000.....	102
FIGURA XXXIII: Gráfico de los distintos componentes del costo del ZTR de TS3000. ....	103
FIGURA XXXIV: impresión de la ejecución del test de la Figura XXVII (adaptado) para una petición con tamaño de respuesta de 1KB en el ZTR de TS3000.....	104
FIGURA XXXV: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del ZTR de TS3000 para distintos valores de mercado de gas virtual.....	105

## Índice de tablas

TABLA I: Costo de registro en OCC para cada ejecutor (en orden de registro).....	76
TABLA II: Costo promedio de creación de una petición por cantidad de ejecutores y por tamaño de respuesta.....	78
TABLA III: Costo promedio de publicación parcial (solo aplicable cuando hay más de un ejecutor) por cantidad de ejecutores y por tamaño de respuesta.....	78
TABLA IV: Costo de publicación final (realizada por el último ejecutor, ejecutando la lógica adicional de cierre de fase) por cantidad de ejecutores y por tamaño de respuesta.....	79
TABLA V: Costo promedio de liberación parcial de resultados (solo aplicable cuando hay más de un ejecutor) por cantidad de ejecutores y por tamaño de respuesta. ....	80
TABLA VI: Costo de liberación final de resultados (realizada por el último ejecutor, ejecutando la lógica adicional de cierre de fase) por cantidad de ejecutores y por tamaño de respuesta.....	81
TABLA VII: Cuadro comparativo de las tres soluciones propuestas .....	107

## 1 Introducción

En la actualidad, en un mundo impulsado por avances tecnológicos y la omnipresencia de la informática, el cómputo se ha convertido en un pilar fundamental para una amplia gama de servicios y aplicaciones. Desde el procesamiento de grandes volúmenes de datos hasta la ejecución de operaciones especializadas para brindar servicios informáticos a empresas y consumidores, la capacidad de gestionar y procesar la información ha adquirido una relevancia sin precedentes y se ha vuelto esencial para empresas, instituciones académicas y organizaciones en general [1]. Ante este panorama, existen diversas alternativas para llevar a cabo el cómputo de datos, cada una con sus propias características y consideraciones a tener en cuenta. La elección de la plataforma adecuada a cada caso de uso resulta fundamental para aprovechar al máximo los recursos y adaptarse rápidamente a las demandas cambiantes del mercado.

Dentro de los paradigmas de procesamiento de datos más populares hoy en día [2], originalmente, el único esquema para el procesamiento de datos consistía en montar físicamente la infraestructura necesaria. Sin embargo, con el pasar de los años, continuaron apareciendo más paradigmas alternativos de computo, como por ejemplo, la ejecución de procesos en arquitecturas virtualizadas en la nube.

Durante los últimos años, apareció una alternativa descentralizada, basada en protocolos de cadena de bloques (blockchain). Las redes blockchain tienen la ventaja de ser transparentes y descentralizadas, es decir, no hay una sola entidad que tenga el control total, y que cualquiera puede participar en el ecosistema.

Por otro lado, la ejecución de procesos en blockchain tiene un costo elevado (como se demostrará en la sección de experimentación del trabajo). A lo largo del trabajo, se propondrán, implementarán y analizarán distintos mecanismos para montar plataformas de procesamiento de datos, basadas en blockchain, en las cuales la ejecución de procesos se realice por fuera de la cadena (offchain), con el objetivo de reducir los costos de uso de la red. Luego, se compararán los resultados obtenidos del análisis de cada protocolo/mecanismo propuesto para determinar ventajas y desventajas de cada uno.

## 2 Motivación

Con el advenimiento de las criptomonedas, han aparecido diferentes protocolos de procesamiento de pagos basados en la tecnología blockchain. Como se mencionó en la introducción (y se explicará más en detalle en la sección de marco teórico), algunas blockchain ya poseen la característica de ejecutar código de forma descentralizada. Concretamente, la blockchain 2.0 con mayor capitalización de mercado es Ethereum [3], y es por esto que este trabajo se centrará en dicho ecosistema. Sin embargo, su entorno de ejecución es muy limitado, ya que su objetivo principal es darle flexibilidad y programabilidad a las transacciones de la blockchain subyacente. Esto quiere decir que la plataforma no está diseñada para, ni tampoco soporta, casos de computo de grandes volúmenes de datos.

Por otro lado, en la cotidaneidad la gente vive rodeada de montones de dispositivos capaces de realizar cómputos complejos de datos (computadoras personales, celulares, dispositivos IoT, etc.) [4]. Esto significa que existe mucha capacidad de computo ociosa distribuida entre las personas. Esta capacidad, es una fuente de recursos potencial para formar una red de computo completamente descentralizada, de acceso público y gestión independiente, solo se requiere de un protocolo para coordinarla de manera eficiente, tanto en recursos como en tiempo.

### 3 Objetivo

Este trabajo es investigar las posibilidades de la tecnología blockchain, en particular de la red Ethereum, con el fin de implementar soluciones que puedan resolver problemas de cómputo general, de manera descentralizada. Concretamente, los objetivos son:

- Analizar la factibilidad técnica de formar una red de computo completamente descentralizada, de acceso público y gestión independiente, mediante la creación de mecanismos de coordinación e incentivos para el cómputo delegado (offchain), basados en Ethereum.
- Verificar si efectivamente el costo de los mecanismos propuestos resultan ser menores a los de la ejecución onchain (y en qué magnitud).
- Comparar los costos de las alternativas propuestas contra las plataformas comerciales de procesamiento delegado (cloud computing).

Para esto, se realizará un análisis cualitativo de las propuestas hechas, comparándolas entre sí y contra las soluciones ya existentes. Posteriormente, se realizarán pruebas de implementación de los protocolos, a fin de realizar un análisis cuantitativo de la complejidad y los costos asociados.

## 4 Estado del arte

Como se mencionó en la sección introductoria, existen diversas alternativas a la hora de elegir una plataforma para el procesamiento de datos. A continuación, se analizarán algunas de las metodologías más populares para la ejecución de dichos procesos.

### 4.1 Procesamiento On-Premises

Si bien en los comienzos de la informática el cómputo de datos se ofrecía como servicio (uno llevaba un programa en una tarjeta perforada a un centro de cómputo, donde se ejecutaba el código y se le entregaba el resultado al cliente, de manera análoga a la virtualización ofrecida por el paradigma cloud), debido a que este paradigma quedó obsoleto [2], no se analizará en este trabajo. Por este motivo, en el contexto de los paradigmas de cómputo presentes hoy en día, el primer esquema consistía en montar físicamente la infraestructura necesaria. La práctica de tener los equipos de procesamiento de manera física se conoce como “On-Premises”. Las principales características de este esquema son contar con un gran nivel de privacidad y control sobre los datos, ya que nadie más tiene acceso físico a los dispositivos de procesamiento y almacenamiento [5]. Un corolario que surge de esta cualidad, es que la propiedad y control de los equipos físicos permite implementar cualquier arquitectura y configuración deseada ya que, al tener el control sobre los equipos, no hay límites en cuanto al tipo de equipos que se pueden desplegar, ni a la arquitectura que se le puede dar al conjunto.

Además, el esquema “On-Premises” implica que los principales gastos se realizan al adquirir los equipos, ya que el costo de energía y mantenimiento es proporcionalmente menor. Teniendo esto en cuenta, el retorno sobre la inversión se maximiza a largo plazo, y cuando se utilice el mayor porcentaje posible de la capacidad total de los equipos, disminuyendo o evitando así la capacidad ociosa. Por otro lado, cambiar o escalar la arquitectura, tiene un alto costo debido a que esto se traduce en la adquisición de nuevos equipos y posiblemente el desecho de otros [5]. Entonces, si las necesidades de procesamiento aumentan significativamente, resulta costoso y complejo expandir la infraestructura existente para satisfacer la demanda creciente.

Otro aspecto a tener en cuenta es la necesidad de tener conocimientos técnicos sólidos para configurar y mantener la infraestructura, así como la responsabilidad de asegurar la protección y copia de seguridad de los datos, lo que puede requerir una inversión adicional en seguridad y sistemas de respaldo [5].

## 4.2 Cloud Computing

Una alternativa a mantener los equipos físicamente, es el cómputo en la nube (cloud computing). El Cloud Computing es una metodología de procesamiento de datos que consiste en la provisión de infraestructura o servicios de cómputo como un servicio en lugar de como un producto, obteniendo acceso sólo durante el tiempo en el que se contrató el servicio [5]. Dicho de otra manera, a través del portal del proveedor, se puede crear toda una arquitectura con infraestructura virtual, por la cual se paga un alquiler correspondiente al tipo de infraestructura y a las condiciones de uso (tiempo/capacidad).

Respecto del esquema “On-Premises”, el cómputo en la nube tiene la ventaja de brindar una mayor escalabilidad y flexibilidad en cuanto a la arquitectura. Esto significa que se pueden adaptar rápidamente los recursos de cómputo para adaptarlos de forma elástica ante fluctuaciones de las necesidades del sistema de cómputo, evitando indisponibilidad de servicio o la necesidad de realizar inversiones en infraestructura para adaptar el sistema a una demanda que puede ser eventual o estacional. Además, los servicios en la nube a menudo ofrecen características avanzadas, como autoescalado, alta disponibilidad y recuperación ante desastres, que pueden mejorar la eficiencia y confiabilidad de las operaciones [6]. Las consecuencias prácticas de estas características radican en la adaptabilidad presupuestaria, es decir, como la infraestructura virtual se adapta dinámicamente a la demanda, se incrementa la capacidad de computo de manera automática cuando lo sea necesario, y cuando la demanda caiga, se reduce dicha capacidad, abstrayendo al usuario de los inconvenientes de comprar nuevos equipos e instalarlos, y desinstalarlos y venderlos. Otra ventaja financiera respecto del esquema “On-Premises” es que, en la nube, toda la infraestructura de respaldo ante desastres está cubierta por el proveedor, y esa garantía

está incluida dentro del precio de contratación de prestaciones, pudiendo el usuario desentenderse de la administración y manutención de la lógica y equipos de respaldo.

Por otro lado, una de las preocupaciones asociadas con el cómputo en la nube es la pérdida ligeramente de privacidad. Después de todo, la nube no deja de ser los equipos de otra empresa, quienes tienen acceso físico constante a los mismos [7]. Al confiar en un proveedor de servicios en la nube, es necesario asegurarse de que cumpla con los estándares de seguridad y confidencialidad de la información como la ISO 27001. Otra desventaja del cómputo en la nube es la dependencia de una plataforma y proveedor específicos. Esto conlleva una especialización en el conocimiento de la plataforma del proveedor elegido, y un vínculo con sus tecnologías y soluciones particulares, lo que puede limitar la portabilidad de aplicaciones y datos en el futuro.

Es importante considerar que los costos de los servicios en la nube pueden variar y resultar más altos a largo plazo en comparación con el esquema “On-Premises”, especialmente si se requiere un uso intensivo de recursos computacionales. De esta dualidad se obtiene como corolario que contratar un proveedor en la nube es eficiente en cuanto a costo cuando se tiene una demanda de computo variada, ya que en caso de poseer los equipos físicamente, cuando no se esté trabajando al 100% se estarían desperdiando recursos. Esto no pasa en la nube, ya que solo se paga por la capacidad utilizada. Sin embargo, para una carga intensiva y constante que pueda acaparar el 100% de los recursos disponibles, tener los equipos “On-Premises” resulta más barato a largo plazo, ya que se omiten los gastos generales de contratación [5].

### **4.3 Volunteer Computing**

Una alternativa menos conocida pero igualmente interesante es el volunteer computing. Este enfoque involucra la participación de usuarios individuales que ofrecen voluntariamente la capacidad de cómputo no utilizada de sus dispositivos personales para realizar tareas computacionales de gran escala. Un ejemplo destacado de esta modalidad es la plataforma BOINC (Berkeley Open Infrastructure for Network Computing). BOINC permite a los usuarios donar su poder de cómputo para proyectos

científicos y de investigación en áreas como la medicina, la astronomía y el cambio climático [8].

Una de las principales características del volunteer computing es la capacidad de aprovechar recursos infrautilizados y distribuir el procesamiento de manera descentralizada, lo que puede acelerar significativamente el avance científico. Además, al no requerir una infraestructura dedicada, es una opción de bajo costo [8].

Sin embargo, también presenta desventajas, como depender de la disponibilidad y participación voluntaria de los usuarios, lo que puede llevar a una capacidad de cómputo irregular y menos predecible. Además, no hay privacidad de datos, debido a la naturaleza distribuida y la falta de control centralizado [8].

A pesar de las limitaciones mencionadas, el volunteer computing se presenta como una alternativa atractiva para casos de uso que no requieran altos niveles de privacidad, como por ejemplo el procesamiento distribuido de datos científicos abiertos [9].

#### **4.3.1 BOINC**

La idea detrás de BOINC es utilizar la capacidad de procesamiento ociosa de miles de computadoras dispersas por todo el mundo para abordar problemas computacionales complejos. Los proyectos de BOINC dividen sus cálculos y procesos en tareas más pequeñas y distribuyen las mismas a los voluntarios que ejecutan el software BOINC en sus dispositivos. Luego, los resultados se envían de regreso a los servidores del proyecto para su análisis [9].

BOINC es de código abierto y está disponible de forma gratuita para cualquier persona que desee unirse a uno o varios de los proyectos de computación distribuida que utilizan esta plataforma. Los voluntarios pueden elegir a cuáles proyectos desean contribuir y configurar sus preferencias en cuanto a cuándo y cuánto poder de procesamiento desean donar [9].

El proceso comienza con los usuarios que desean participar en proyectos de computación distribuida instalando el cliente BOINC en sus dispositivos personales, como computadoras de escritorio, laptops o incluso dispositivos móviles. Este cliente es una aplicación que actúa como intermediario entre el usuario y los proyectos de BOINC [9].

Luego los usuarios pueden elegir entre una variedad de proyectos que utilizan la plataforma BOINC. Cada proyecto aborda una tarea específica de investigación científica o cálculo, como la búsqueda de señales de radio de vida extraterrestre, la simulación de proteínas o la predicción del cambio climático [9].

Una vez que un usuario se une a un proyecto, el cliente BOINC se conecta al servidor del proyecto correspondiente y descarga unidades de trabajo o "tareas" en la computadora del usuario. Estas tareas suelen ser pequeñas porciones de cálculos que pueden completarse en un tiempo razonable. El cliente BOINC gestiona las tareas descargadas y las coloca en una cola de procesamiento. Para aumentar la confiabilidad, BOINC a menudo ejecuta múltiples copias de la misma tarea en diferentes dispositivos de usuarios para mitigar errores o fallos en el hardware y el software [9].

Una vez que se completa una tarea, el cliente BOINC la envía de vuelta al servidor del proyecto. Los servidores del proyecto reciben los resultados y realizan verificaciones para asegurarse de que las tareas se hayan completado correctamente. Si se detecta un error en un resultado, el servidor puede asignar la misma tarea a otro voluntario para verificar la consistencia de los resultados. Este proceso de redundancia ayuda a garantizar la precisión y la confiabilidad de los cálculos. El cliente BOINC obtiene continuamente nuevas tareas del servidor del proyecto, lo que permite a los usuarios seguir contribuyendo de manera continua [9].

BOINC posee diversos incentivos para la donación de poder de cómputo. Los usuarios pueden rastrear su contribución y progreso en el proyecto a través de un sistema de

puntuación y estadísticas. A menudo, se otorgan puntos a los voluntarios según la cantidad y la complejidad de las tareas completadas [10].

Algunos proyectos otorgan distinciones especiales a los voluntarios que han alcanzado ciertos hitos, como un número específico de tareas completadas o un alto nivel de puntos. Estas distinciones pueden incluir títulos honoríficos, medallas virtuales u otros reconocimientos [10].

Otros proyectos emiten certificados virtuales o reales a los voluntarios que han hecho contribuciones significativas. Estos certificados pueden ser una forma de reconocer el esfuerzo y la dedicación de los participantes [10].

Es importante destacar que, en general, BOINC se basa en la voluntariedad y la filantropía, y los incentivos suelen ser simbólicos. La principal motivación para los voluntarios suele ser contribuir a la investigación científica y ayudar en proyectos que consideran importantes. La comunidad de BOINC está compuesta en gran parte por personas interesadas en la ciencia y la tecnología que están dispuestas a donar su tiempo de cómputo para avanzar en el conocimiento en lugar de recibir compensación financiera directa [10].

#### **4.4 Procesamiento en Blockchain**

En el ámbito de las plataformas de ejecución de procesos, cabe mencionar a las blockchain 2.0, es decir, aquellas redes descentralizadas de nodos en la que los datos y sus reglas de modificación simulan una máquina virtual, con cada nueva transacción no siendo más que ejecución de código resultante en un cambio de estado de dicha máquina virtual [11]. Si bien existe una infinidad de protocolos con características distintas, en la mayoría de ellos, el objetivo del cómputo virtual llevado a cabo, es el de darle una mayor flexibilidad a la red subyacente e incrementar las posibilidades de uso, y no pretende imponerse como un nuevo paradigma o alternativa para el cómputo en general [12].

#### 4.4.1 Estructura de una blockchain

Para entender cómo funciona el procesamiento en blockchains hay que entender qué es una blockchain. En el contexto del cómputo descentralizado, una blockchain, o mejor dicho, un protocolo blockchain, es un conjunto de reglas que permite la existencia y manipulación de una estructura de datos conocida como cadena de bloques, de manera descentralizada. Esto significa que está compuesta por un numero de nodos independientes que voluntariamente se adhieren a la red. A diferencia de un sistema distribuido tradicional, un sistema descentralizado tiene la particularidad de que ningún nodo en específico tiene el control sobre dicha red, sino que existe un algoritmo de predeterminado para definir el estado del sistema.

En cuanto a la representación del estado de la red, una cadena de bloques no es más que una estructura de datos compuesta por bloques, valga la redundancia, que poseen un encabezado donde está toda la información pertinente a la cadena como el identificador de bloque y el timestamp de emisión, y una carga útil que no es más que información a ser usada por el protocolo pertinente. Una particularidad de esta estructura de datos es que cada bloque contiene un “digest” del bloque anterior, es decir, un pequeño “resumen” de tamaño fijo obtenido como resultado de aplicarle un algoritmo matemático a los datos del bloque anterior. Dado que este “resumen” forma parte del nuevo bloque, y que el resumen del nuevo bloque formará parte del siguiente, es fácil ver cómo el digest de cada bloque va a estar influenciado por todos los bloques anteriores. Como consecuencia de esto, es imposible modificar un bloque en la cadena, sin modificar todos los bloques siguientes [13].

Este conjunto de bloques forma una secuencia histórica de todos los cambios de estado que sufrió la blockchain, siendo el último bloque, la más reciente actualización. Las primeras blockchain consistían solamente en sistemas de pago. Es decir, que la carga útil de sus bloques representa transacciones que no son más que movimientos de valores dentro de un libro de cuentas virtual. Con la llegada de las blockchain 2.0, siendo Ethereum el principal exponente, el estado interno de la cadena paso de representar un libro de cuentas, a representar un conjunto de variables que simulan una máquina virtual.

De igual manera, los cambios en dicho estado, dejan de ser simples ajustes de balances, y pasan a regirse por un código programable por los mismos integrantes de la red. Esto significa que las transacciones representan la ejecución de dicho código, lo que abre un abanico de posibilidades técnicas mucho mayor a un simple libro de cuentas con programabilidad limitada [11]. De esta manera, cada vez que un participante interactúa con la blockchain mediante una transacción, se ejecuta el código correspondiente a la misma, traduciéndose en un potencial cambio de estado.

#### **4.4.2 Interacción con la red**

Cada vez que alguien desee interactuar con una red Blockchain, lo debe hacer a través de una interacción con un nodo RPC (del inglés Remote Procedure Call). En el contexto de las blockchain, esto significa una simple comunicación con cualquier nodo perteneciente a la red. Dicha interacción puede ser de lectura, en cuyo caso el nodo responde como si se tratara de una simple API. Si la interacción es de escritura, se le debe enviar al nodo una transacción, firmada por una dirección o “cuenta” particular. Estas “cuentas” no son más que pares de llaves públicas/privadas, que identifican únicamente a una entidad particular dentro de la red [14]. En el ámbito de la criptografía asimétrica, esto significa que dichas llaves, no son más que valores numéricos de los cuales se vale un algoritmo matemático para realizar operaciones criptográficas. La operación criptográfica que nos interesa en este contexto es la de firma digital. Básicamente, una firma digital es un método criptográfico que autentica y asegura la integridad de un bloque de información digital, asociando digitalmente la identidad de un remitente con el contenido del archivo, lo que permite verificar que no ha sido alterado y que proviene de la fuente declarada. De esta manera, diferentes usuarios pueden realizar interacciones con la red de una manera segura y verificable, manteniendo la confidencialidad de su clave privada [15].

Como la cadena está montada sobre una red descentralizada de nodos, cada transacción es propagada desde el nodo inicial que recibió el RPC, hacia el resto de los nodos en su lista de participantes. Es así como todos los nodos se van comunicando constantemente para intercambiar información sobre transacciones, y sobre la versión

más actualizada de la cadena. Los nodos también intercambian sus listas de participantes entre sí de una manera reminiscente a un algoritmo de ruteo, para que cuando un nuevo nodo independiente se une a la red, le baste con conocer a un nodo participante para empezar a intercambiar información y eventualmente conocer el estado actual de la cadena, así como la lista del resto de los nodos [16].

Dado que la cadena está compuesta por una red de nodos independientes, cada uno debe mantener una copia del estado actual de la cadena. Como no hay ninguna autoridad central para determinar cuál es la última versión de la cadena, para que la red funcione, los nodos deben tener un mecanismo de consenso, para acordar cuál es la versión oficial de la misma.

#### **4.4.3 Algoritmos de consenso**

Un algoritmo de consenso es un conjunto de reglas y procedimientos que le permite a una red descentralizada llegar a un acuerdo sobre el estado actual del sistema. En el contexto de las blockchain, estos algoritmos son esenciales para que los participantes, que son nodos independientes, puedan llegar a un consenso sobre la integridad de la información sin depender de una autoridad central. Estos algoritmos facilitan la coordinación entre nodos de la red, asegurando que todos estén de acuerdo sobre la versión única y válida del historial de transacciones.

El algoritmo de consenso más popular en blockchain es Proof of Work (PoW) [17], o prueba de trabajo. Este consiste en que para que cada bloque sea válido, ciertos nodos especiales llamados “mineros”, recopilan transacciones propagadas por la red y las empaquetan en la carga útil de un nuevo bloque. Luego, para que dicho bloque sea válido y pueda ser añadido a la red, los mineros compiten para resolver un problema criptográfico. Este problema consiste en encontrar un número que haga que el “digest” del bloque tenga una forma específica, cómo comenzar con una cierta cantidad de ceros. La particularidad de este problema es que es muy costoso de resolver, pero muy fácil de comprobar. De esta manera, la regla de consenso consiste en tomar como oficial a la cadena que más bloques tenga, es decir, aquella en la que más veces se haya resuelto

el problema criptográfico [18]. Por consiguiente, para que un conjunto de nodos fraudulentos pueda controlar la red, debería poseer como mínimo el 51% del poder de cómputo de todos los nodos mineros. Como el cómputo involucrado en este algoritmo es intensivo, ya que la resolución es por fuerza bruta, entre más grande la red, menos vulnerable a un ataque de este estilo.

Por otra parte, un algoritmo de consenso que fue ganando popularidad en los últimos años, es el de Proof of Stake (PoS) [17], o prueba de participación. De hecho, Ethereum, previo a la oficialización de la versión 2.0, utilizaba PoW. Desde la actualización de Ethereum 2.0, el algoritmo de consenso utilizado por la red es PoS. En este algoritmo, los nodos validadores adquieren el derecho a validar bloques conforme a la cantidad de Ether que tengan bloqueados. Cuanto más tengan bloqueado, mayor será su participación y posibilidades de ser elegidos para validar un bloque, ya que en lugar de competir en una carrera de “resolución de rompecabezas” como en PoW, los validadores se seleccionan aleatoriamente para proponer y validar bloques. Para garantizar que los validadores actúen de manera honesta, el sistema PoS generalmente incluye mecanismos de castigo. Si un validador actúa de manera fraudulenta o perjudica la red, puede perder parte o la totalidad de su garantía [18]. Es interesante notar que, dado que PoS requiere un nivel de cómputo ínfimo a diferencia de PoW, ya que está basado en un incentivo económico, y no en un cálculo criptográfico, el pasaje en Ethereum de PoW a PoS dejó un montón de dispositivos con poder de cómputo ocioso [19] (de ahí surge la idea de diseñar un protocolo para aprovechar no solo esa capacidad excedente, sino todo el computo ocioso en general).

Independientemente del algoritmo utilizado, el resultado final es que a mediano/largo plazo, la totalidad de los nodos tiene un consenso de cuál es la versión oficial de la cadena. Un corolario de la necesidad de que los nodos participantes en una red lleguen a un consenso acerca de cuál es la versión oficial de la cadena, es la idea de procesamiento determinístico. Esto significa que cualquier procesamiento de datos sobre una cadena, no debe depender en absoluto de variables aleatorias, sino que, ante la misma condición inicial, cualquier ejecución de dicho procesamiento, debe llevar a un

estado resultante idéntico independientemente del nodo en donde se haya ejecutado. Esta cualidad determinista de las blockchain, es una condición necesaria para que todos los nodos puedan estar en consenso, manteniendo una cadena con el mismo estado.

Habiéndose delineado conceptualmente los algoritmos de consenso más populares entre los protocolos blockchain con mayor capitalización de mercado, se procederá a profundizar sobre la plataforma Ethereum, cuya máquina virtual es conocida como EVM (Ethereum Virtual Machine). Como se mencionó en las secciones anteriores, este trabajo está basado en el protocolo Ethereum, ya que su implementación oficial es la red con mayor ecosistema y su moneda fue una de las pocas que mantuvo un valor significativo después de la caída del 2021 [3].

#### **4.4.4 Ethereum**

Ethereum, como todas las blockchain 2.0, da vida a una máquina virtual, llamada EVM en este caso, montada sobre una red descentralizada de nodos independientes. Estos nodos no son más que computadoras corriendo el protocolo, que comparten información sobre el estado actual de la EVM. Hay que recordar que las reglas de comportamiento de una máquina virtual en una blockchain 2.0 son que tiene un estado virtual, y que los cambios en dicho estado se producen mediante transacciones que, como se mencionó en capítulos anteriores, no son más que ejecuciones de código sobre la misma red. Dichas transacciones se agrupan en bloques que son añadidos cíclicamente a la cadena, siendo el último bloque la más reciente actualización a la red [11].

En cuanto al contenido de las transacciones, en Ethereum, existen esencialmente 3 tipos mediante los cuales una dirección particular puede interactuar con la red. La primera y más simple de todas, es el envío de fondos. La criptomoneda nativa de Ethereum es el ether, y la forma más básica de interactuar con la red es una simple transacción en la cual la dirección “A” le envía una cierta cantidad de fondos (ether) a la cuenta “B” [12].

El segundo tipo de transacciones son la interacción con contratos inteligentes. Un contrato inteligente es un programa informático que se ejecuta en la blockchain. Estos

contratos son autónomos y se ejecutan de manera automática sin la necesidad de una autoridad centralizada. En el contexto de la programación orientada a objetos, cada contrato es análogo a una clase, y sus instancias en la red, son análogos a objetos. Al igual que los objetos, las instancias de contratos pueden tener un estado interno (que es parte del estado global de la EVM), así como también funciones que le dan comportamiento. Algunas de estas funciones tienen la característica de ser públicas o externas, lo que significa que pueden ser accedidas por otros contratos, o directamente por una “cuenta particular” [12]. Y este es efectivamente el segundo tipo de transacciones, en el cual una entidad externa decide interactuar con un contrato, enviando a la red una transacción con la dirección del contrato (identidad única de la instancia), la función con la que desea interactuar, y los parámetros que le correspondan a dicha función. También dependiendo de la función en sí, esta puede aceptar fondos junto con el llamado. Toda transacción que llame a una función posee un contexto que brinda datos como el número de bloque, el “digest” o “hash” de la transacción, el valor del ether enviado, y la dirección desde la cual viene el llamado, entre otros [12].

Por último, el tercer tipo de transacción consiste en la instanciación de contratos. Esto se trata simplemente de tomar un contrato programado en “Solidity” (el lenguaje en el que se programan los contratos en Ethereum), compilarlo al lenguaje máquina de la EVM, y enviarlo a la red para ser instanciado, para que posteriormente se pueda interactuar con el mismo [12]. Solidity, es un lenguaje orientado a objetos, compilable a un lenguaje de máquina virtual, que es el que finalmente se termina por ejecutar en la EVM. Algunos conceptos y definiciones particulares de Solidity, son:

- Estructura [struct]: estructura de datos que permite definir un conjunto de variables con diferentes tipos que pueden ser agrupadas bajo un solo nombre para representar un tipo de dato personalizado.
- Modificadores de funciones [modifier]: función especial que se utiliza para agregar lógica adicional a otras funciones, permitiendo la validación de condiciones antes de ejecutar el código de la función principal.
- Función pública [public]: función que es accesible desde tanto dentro como fuera del contrato.
- Función pagable [payable]: función que puede recibir fondos.

- Función pura [pure]: función con lógica pura, cuyo comportamiento depende exclusivamente de los parámetros de entrada, es decir, que no permite ni la lectura ni modificación del estado interno de la EVM.
- Función de lectura [view]: una función que no permite la modificación del estado interno de la EVM, pero sí la lectura del mismo.
- Evento [event]: declaración que permite que un contrato emita información a la cadena de bloques, permitiendo a las aplicaciones externas escuchar y reaccionar.
- Contrato abstracto [abstract]: tipo de contrato que no puede ser instanciado por sí mismo y que generalmente contiene métodos sin implementación, destinados a ser sobreescritos por contratos derivados para proporcionar implementaciones específicas.
- Mapping: un tipo de dato que funciona como un arreglo asociativo.

Mediante la programación y el despliegue de una serie de contratos interconectados, un usuario puede crear una aplicación con lógica robusta para un fin en particular. El comportamiento de dicha aplicación va a estar regido por el o los contratos subyacentes. La particularidad que tienen dichas aplicaciones es que, al vivir en una red descentralizada, ellas también son descentralizadas y su comportamiento depende exclusivamente de la lógica subyacente de los contratos que la componen y, por ende, no son controladas por nadie. Es por eso que a las aplicaciones que viven en las blockchain 2.0 se las conoce como dApps (del inglés decentralized applications) [20].

#### **4.4.4.1 Reglas de ejecución en Ethereum**

Una cosa importante a entender en cuanto a la interacción con la red Ethereum respecta, es que, si bien la lógica de las interacciones está definida por los contratos inteligentes desarrollados por la comunidad, el protocolo de interacción está definido por la misma red. Para que la red tenga viabilidad, debe haber un incentivo económico para los ejecutores de dicho código. La ejecución de una transacción conlleva una comisión a pagar por los servicios de cómputo prestados por el nodo ejecutor. Dada la variedad en complejidad de los diversos contratos, no tendría sentido que dicha comisión sea fija, por lo que la suma a pagarse es proporcional a la cantidad de código ejecutado por dicha transacción. Otra cuestión a tener en cuenta, es que a diferencia de las blockchain 1.0, los contratos pueden poseer lógica irrestricta al igual que un programa de computadoras

convencional [11]. Esto significa que existe la posibilidad de que haya un bucle infinito dentro del código. Para evitar que un nodo se cuelgue eternamente, y para que el ejecutor reciba una retribución proporcional al trabajo computacional que hizo, se creó el concepto de “Gas”. Ethereum estipula que cada ejecución de una instrucción dentro de la máquina virtual, tiene un costo expresado en unidades llamadas “Gas”, y que la comisión de cada transacción está representada por una cantidad de Gas proveída por la cuenta creadora de dicha transacción. De esta manera, un nodo sólo ejecuta instrucciones hasta que la cantidad pagada de Gas se lo permita, impidiendo así que haya un bucle infinito, ya que para que este sea el caso, la cuenta creadora de la transacción debería haber pagado una cantidad infinita de gas. En la realidad, la determinación del precio y cantidad de Gas, así como una comisión “extra” para incentivar a los nodos procesadores de transacciones, es más compleja y excede el alcance de este trabajo [21]. En resumen, cada unidad de este gas tiene un precio que se determina algorítmicamente por el nivel de uso que tiene la red. A mayor demanda, mayor el precio del gas y, por ende, más caras son las transacciones.

Sin ánimos de adentrarse fuertemente en los aspectos económicos de la plataforma, ya que este es un trabajo enfocado en la parte técnica de la red, cabe mencionar que, para que el ether tenga un valor firme a lo largo del tiempo, debe haber un equilibrio entre la oferta y demanda de esta moneda. En esta danza de mercado juegan las fuentes y sumideros del activo. La fuente de creación de nuevo ether, es la creación de nuevos bloques. Cada conjunto de transacciones es agrupado en un bloque y se procesa y publica en la red. Como recompensa para el nodo que procesó, validó y publicó dicho bloque, la red acuña una cantidad de ether que es destinada a la dirección particular del administrador del nodo. Esta acuñación de moneda aumenta la oferta de dicho activo y, por ende, de no haber un sumidero de ether, el carácter monetario de este activo sería eternamente inflacionario. Inicialmente el precio del gas se determinaba puramente por oferta y demanda, y todo el ether destinado al Gas de una transacción, era transferido al nodo que validaba el bloque. Luego de la actualización del protocolo EIP-1559, se determinó que el precio del gas se determina algorítmicamente de acuerdo a la congestión de la red, y que parte del gas proveído se “quemaría”, formando así un

equilibrio dinámico del activo circulante [21]. Los detalles de la acuñación y quema de ether exceden el alcance del trabajo.

En cuanto a los costos de ejecución promedio de una instrucción, debido a que la EVM no fue diseñada para realizar tareas de cómputo general, (sino para ejecutar una serie simple de pasos, que sea suficiente para darle complejidad y extensibilidad a las transacciones que forman parte de la red [12]), el precio de ejecución de cada instrucción hace que realizar cualquier tarea computacionalmente significativa sea prohibitivamente caro [21]. Incluso, en muchos casos, el conjunto de instrucciones a ejecutarse por parte los contratos inteligentes, para casos de uso vinculados a movimientos de dinero más complejos, suele ser demasiado caro, por lo que naturalmente surgieron alternativas de ejecución para mejorar la escalabilidad.

#### **4.4.5 Rollups: una solución de escalabilidad**

Un rollup es una solución de escalabilidad (para cualquier blockchain 2.0 que lo implemente), que busca mejorar el rendimiento de las cadenas de bloques al realizar operaciones fuera de la chain, para quitarle estrés a la red, pero retribuyendo los resultados a la chain, para mantener la seguridad e integridad que esta brinda. El objetivo principal de un rollup es reducir la carga de procesamiento y el consumo de recursos en la cadena de bloques subyacente, permitiendo así un mayor rendimiento y eficiencia [22].

Como todo lo que sucede dentro de la chain es determinístico, para que un rollup sea efectivo, debe haber un sistema para garantizar la integridad de la ejecución de los procesos offchain.

Al utilizar un rollup, se puede lograr una mayor capacidad de procesamiento de transacciones, tiempos de confirmación más rápidos y menores costos de transacción en comparación con la cadena de bloques principal. Además, los rollups pueden admitir la ejecución de contratos inteligentes y aplicaciones descentralizadas (DApps), lo que amplía aún más sus posibles casos de uso [22].

#### 4.4.5.1 Optimistic Rollups

Un tipo de rollup son los optimistic rollups. Un optimistic rollup es una solución de escalabilidad para blockchain que busca mejorar el rendimiento y la eficiencia al procesar transacciones de manera "optimista" en una capa secundaria (es decir, por fuera de la cadena). A diferencia de otros enfoques, los optimistic rollups confían inicialmente en la validez de las transacciones sin realizar una verificación exhaustiva en la cadena principal [23].

En un optimistic rollup, las transacciones se agrupan y procesan en la capa secundaria de manera rápida y eficiente, sin la necesidad de una validación criptográfica exhaustiva. Luego, se genera una "rollup block" que contiene un resumen de las transacciones y una prueba criptográfica que demuestra que el procesamiento en la capa secundaria ha sido correcto [23].

Aunque las transacciones se consideran válidas en la capa secundaria, existe la posibilidad de que se produzcan transacciones no válidas o fraudulentas. Para abordar esto, los optimistic rollups incluyen un mecanismo de "disputa" en el que cualquier parte interesada puede presentar una objeción y cuestionar la validez de las transacciones incluidas en un bloque. Cuando se presenta esta situación, se inicia un período de "ventana de disputa" durante el cual se permite la presentación de pruebas y argumentos sobre la validez de las transacciones [23].

Durante la ventana de disputa, los usuarios pueden presentar pruebas que demuestren la invalidez de una transacción. Si se demuestra que una transacción es inválida, se revierte en la capa secundaria y ejecutan los procesamientos, esta vez directamente en la cadena principal, para llegar a los resultados correctos de manera determinista. Por otro lado, si no se presentan disputas dentro del período establecido, las transacciones se consideran válidas y se confirman en la cadena principal [23].

La ventaja clave de los optimistic rollups radica en su capacidad para procesar un gran número de transacciones fuera de la cadena principal, lo que conduce a una mayor

escalabilidad y menores costos de transacción. Sin embargo, existe un breve período de tiempo durante la ventana de disputa en el que las transacciones podrían revertirse, lo que implica un riesgo potencial de reorganización. Por lo tanto, la efectividad y la seguridad de los optimistic rollups dependen de un equilibrio adecuado entre la eficiencia de los ejecutores y la capacidad de resolución de disputas [22].

#### **4.4.5.2 Zero Knowledge Rollups**

La cuestión central en todas las soluciones de tipo Rollup giran en torno al “cómo” se resuelve el tema de la confianza y la comprobación de los resultados de ejecuciones hechas offchain. Entre menos confianza requiera una solución, menos margen de error hay y, por ende, más robusta es [22]. Al adentrarse en el campo de las comprobaciones computables, es inevitable toparse con el concepto de pruebas criptográficas. Una prueba criptográfica es un método matemático que verifica la autenticidad o veracidad de una pieza de información, de forma segura y confiable utilizando criptografía. En las secciones anteriores de este trabajo, se mencionó que la intención general de un rollup es la de generar un mecanismo de verificación de ejecución delegada, para permitir la comprobación de validez de los procesos realizados offchain. En la sección de Optimistic Rollups se habló de como ese mecanismo utiliza la ejecución determinista onchain como respaldo de integridad para los procesos realizados offchain. Sin embargo, existen tipos de rollups cuyo mecanismo de integridad no depende de una ejecución total onchain, sino que se valen de pruebas criptográficas para realizar las comprobaciones necesarias de manera determinista. La alternativa más popular a los Optimistic Rollups, son los Zero Knowledge Rollups. Estos, se basan en una serie de pruebas criptográficas llamadas pruebas de conocimiento cero, para validar los procesos ejecutados offchain.

Una prueba de conocimiento cero (zero-knowledge proof, en inglés) es un concepto de criptografía y matemáticas que se utiliza para demostrar la veracidad de una afirmación sin revelar ninguna información adicional sobre la misma. En esencia, es una forma de probar que algo es verdadero sin revelar cómo se llegó a esa conclusión [24].

En una prueba de conocimiento cero, una parte (el "verificador") desea obtener una prueba de que otra parte (el "demostrador") posee cierta información o conoce una solución a un problema, sin que se revele la información o la solución en sí. El objetivo es que el verificador pueda estar convencido de la veracidad de la afirmación sin aprender nada más que eso [24].

Para lograr esto, el demostrador realiza una serie de pasos para generar una prueba, sobre la cual el verificador realiza comprobaciones sin obtener información adicional más que la veracidad de la misma. Mediante este proceso, el demostrador demuestra que tiene conocimiento de algo, revelando únicamente la validez de esa información, pero sin revelar el contenido real del conocimiento [24].

La propiedad fundamental de una prueba de conocimiento cero es que no proporciona ninguna pista o información adicional que permita a un tercero reproducir la prueba y fingir tener el conocimiento sin poseerlo realmente. Además, se espera que la prueba sea convincente para el verificador, es decir, que el verificador pueda estar seguro de que el demostrador posee el conocimiento sin necesidad de revelar la información en sí.

Una forma imaginativa de ilustrar cómo funcionan las pruebas de conocimiento cero sin revelar información confidencial es el siguiente juego de pensamiento:

Imaginemos una cueva con dos entradas/salidas. En el medio de esta cueva, hay una puerta que puede ser desbloqueada mediante un código secreto ingresado en un teclado. Si alguien quiere demostrarle a otra persona que conoce el código de desbloqueo sin revelar el mismo, todo lo que debe mostrar es la capacidad de ingresar por un extremo de la cueva, abrir la puerta y salir por el otro extremo. Si esta persona ha tenido éxito en esta demostración, entonces la otra persona sabe sin lugar a dudas que la primera ha sido capaz de desbloquear la puerta, sin que se haya revelado el código de desbloqueo [25].

Una acepción más purista de las pruebas de conocimiento cero, las define de tal manera que la misma sólo debe tener validez para el verificador, y nadie más. Es decir, lo único que se demuestra es que el demostrador posee determinado conocimiento ante el verificador, pero el verificador no podría a su vez proclamar ante un tercero que el demostrador en verdad posee ese conocimiento [24]. En otras palabras, la prueba solo es válida para el verificador.

En el ejemplo de la cueva, una “prueba de conocimiento cero” purista se daría de la siguiente manera: el verificador cierra los ojos y el demostrador entra por alguna de las dos entradas al azar. Luego, el verificador abre los ojos y grita “A” o “B”, refiriéndose a una de las entradas, y el demostrador saldrá por ella. En primera instancia el verificador podría pensar que el demostrador tiene un 50% de probabilidades de haber adivinado y de no tener la clave. Por consiguiente, se repite el experimento tantas veces como el verificador quiera. Eventualmente, las posibilidades de que el demostrador haya acertado siempre en vez de tener la clave, se hacen prácticamente nulas [25]. Este ejemplo se trata de una prueba interactiva, ya que se requiere una serie de acciones sincronas entre demostrador y verificador para que la prueba tenga lugar.

En este caso, la demostración solo le sirve al verificador ya que, incluso si este filma el proceso, alguien podría sospechar que el verificador y demostrador habían acordado previamente el orden en el que el primero le pediría al segundo que aparezca, por lo que dicha filmación, no sería evidencia irrefutable de que el demostrador posee la clave.

En el campo de la informática, podríamos ejemplificar un caso purista de prueba de conocimiento cero de la siguiente manera. Sea el demostrador un agente que proclama poseer la clave privada vinculada con cierta clave pública, y sea el verificador un agente interesado en saber si esta aseveración es cierta. En este caso, el verificador no tendría más que generar una cadena secreta y cifrarla con la clave pública. Luego darle el código cifrado al demostrador, quien la descifraría, y así demostraría que posee la clave privada ya que esa es la única manera en la cual podría obtener la cadena secreta inicial del verificador. Adicionalmente, el verificador no podría convencer a un tercero de que el

demostrador en efecto tiene la clave privada, ya que dicho tercero fácilmente podría acusar al verificador y demostrador de haber coordinado la misma cadena secreta de antemano.

Esta tecnología tiene mucho potencial para blockchain, ya que de lograrse formar una prueba de conocimiento cero sobre que cierto conjunto de transacciones ocurrió por fuera de la cadena, sus resultados podrían ser confiablemente oficializados y retribuidos a la cadena principal, sin la necesidad de que todo ese procesamiento de transacciones haya ocurrido por dentro de la red principal, escalando tanto el tiempo de ejecución, como el costo en gas de los resultados retribuidos.

Un zk-rollup (Zero-Knowledge Rollup) es una solución de escalabilidad para blockchain que combina la capacidad de procesamiento de transacciones fuera de la cadena (off-chain) con la seguridad y la transparencia de la cadena de bloques (on-chain) mediante el uso de Pruebas de Conocimiento Cero [25].

En un zk-rollup, las transacciones individuales se agrupan y se procesan en una capa secundaria (off-chain), y luego se genera una prueba criptográfica que resume y verifica todas las transacciones agrupadas. Esta prueba se publica en la cadena de bloques principal (on-chain), que actúa como un árbitro y garantiza la seguridad e integridad de las transacciones sin necesidad de verificar cada una individualmente [25].

El zk-rollup permite un procesamiento de transacciones mucho más eficiente, ya que reduce drásticamente la carga de trabajo en la cadena de bloques principal al agrupar múltiples transacciones en una única prueba criptográfica. Esto resulta en tiempos de confirmación más rápidos y tarifas de transacción más bajas en comparación con las soluciones tradicionales basadas en la cadena principal [26].

Concretamente, hay dos tipos de zk-rollups: zk-SNARK y zk-STARK. Un zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) es un tipo de prueba

criptográfica que permite demostrar el conocimiento de cierta información sin revelarla explícitamente [26].

El término "zero-knowledge" se refiere al hecho de que el conocimiento demostrado en una prueba zk-SNARK no revela ninguna información adicional más allá de la validez de la afirmación. Es decir, la prueba demuestra la validez de ciertos datos sin revelar los mismos [25].

La característica "succinct" indica que las pruebas zk-SNARK son altamente compactas, lo que significa que son muy pequeñas en comparación con la cantidad de información que verifican. Esto las hace eficientes en términos de procesamiento y almacenamiento [25].

Por último, "non-interactive" significa que la prueba zk-SNARK puede ser generada de antemano y verificada sin requerir interacciones adicionales entre quien genera la prueba y quien la verifica [26].

Las pruebas zk-SNARK dependen de una configuración inicial confiable entre un demostrador y un verificador, por lo que se requiere un conjunto de parámetros públicos para construir las pruebas. Estos parámetros están codificados en el protocolo y son uno de los factores necesarios para probar que una transacción fue válida [26].

Por otro lado, están los zk-STARKs (Zero-Knowledge Scalable Transparent Argument of Knowledge): Los zk-STARKs son otra variante de zk-rollup que utiliza zk-STARK como la prueba criptográfica subyacente. A diferencia de los zk-SNARKs, los zk-STARKs son completamente transparentes y no requieren parámetros de confianza, lo que los hace más adecuados para casos de uso en los que la transparencia y la seguridad (frente a posibles ataques en la generación de parámetros) son importantes. Sin embargo, los zk-STARKs tienden a ser más grandes en términos de tamaño de prueba y verificación, lo que puede afectar la eficiencia y el rendimiento en comparación con los zk-SNARKs [26].

## 5 Desarrollo y Experimentación

Para comenzar con la sección de desarrollo y experimentación de este trabajo, es útil recordar cual es la problemática que le brinda motivación. Cuando uno escucha hablar escalamiento en ethereum, por lo general se está haciendo referencia a escalamiento transaccional, es decir, proponer soluciones para aumentar la cantidad de transacciones por segundo que puede soportar la red. El escalamiento propuesto en este trabajo, en cambio, tiene que ver con el costo de ejecución de procesos computacionales significativos, ejecutándolos por fuera de la blockchain, con el fin de abaratar el procesamiento orientado al cómputo general. La intención es explorar la posibilidad de un escalamiento orientado al cómputo de datos generalizado, basado en blockchain, mediante tres soluciones/propuestas.

### 5.1 Primera Solución: Optimistic Execution

Como se mencionó en secciones anteriores del trabajo, la plataforma ethereum se encuentra constantemente en la lucha por el escalamiento, para poder hacer que el uso de la red sea más barato y accesible. Como cada instrucción ejecutada dentro de la EVM cuesta gas, la ejecución de lógicas complejas resulta inevitablemente en un costo muy elevado (comparado con otras formas de cómputo delegado, como los servicios de nube).

De esta manera nace la idea de ejecutar código offchain, para que no se consuma gas de procesamiento, y que solo estén los gastos vinculados a la publicación de los resultados obtenidos. Ahora bien, no es tan sencillo como simplemente ejecutar el código de manera desconectada y cargar los resultados a la red, ya que hay cuestiones importantes a tener en cuenta como la creación de incentivos para la ejecución del código y mecanismos la verificación de la integridad de los resultados cargados.

#### 5.1.1 Propuesta

A primera vista es fácil darse cuenta de que, por la definición de la problemática, todas las soluciones propuestas en este trabajo se tratan de plataformas para crear Rollups,

es decir, protocolos de cómputo de código off-chain, con retribución de resultados a la red principal. La diferencia entre cada propuesta yace en el mecanismo empleado para lograr dicha integración de manera segura.

En cuanto a la primera solución “Optimistic Execution” se trata de una plataforma de creación de rollups optimistas, en la que cada cliente publique las tareas a ser procesadas junto a una oferta de remuneración, y que los ejecutores puedan elegir resolverlas, publicar los resultados, y reclamar la recompensa. Para adquirir el derecho de ejecución, un ejecutor debe depositar un “seguro de desafío”, es decir, una cantidad de ether definida por el cliente. La integridad es resguardada por un sistema de control por oposición de intereses, donde cualquier ejecutor puede desafiar los resultados de una ejecución anterior, y en caso de demostrarse que el resultado publicado es incorrecto, el desafiante obtiene tanto la recompensa como el seguro de desafío, generando un sistema de incentivos que busque asegurar la integridad de comportamiento de todos los ejecutores y, por ende, de los resultados publicados.

Ya existen plataformas de escalabilidad para Ethereum, como Optimism y Arbitrum, con el objetivo de mejorar la eficiencia de procesamiento de transacciones en la red Ethereum mediante la implementación de rollups optimistas. Estas buscan acelerar la velocidad de las transacciones y aliviar la congestión de la red, lo que beneficia a los desarrolladores de aplicaciones descentralizadas (DApps) y a los usuarios al ofrecer una experiencia más rápida y económica en la red Ethereum [27].

La diferencia principal entre la primera propuesta y estas plataformas ya existentes es que el foco de “Optimistic Execution” es extender la red Ethereum para cubrir casos de computo de uso general y no simplemente escalar el ecosistema de dApps.

### **5.1.2 Funcionamiento y características**

La idea central es que el cliente publique peticiones con sus requerimientos y oferta de pago. Cada ejecutor que quiera competir por cobrar, tiene que ejecutar el código y

publicar el resultado obtenido. Durante el desarrollo fue necesario diseñar los sistemas de incentivos y aseguramiento de la integridad.

En cuanto a los incentivos, es decir, algo que incite a los ejecutores a efectivamente correr el código y publicar los resultados, se optó por ofrecer una compensación económica en la moneda nativa de la red ethereum, el ether.

Luego está el problema de la integridad. ¿Cómo se pueden plantear los incentivos y las reglas de la plataforma para garantizar que los ejecutores actúen de forma honesta? ¿Cómo se puede asegurar que los resultados sean correctos? Para resolver esta cuestión, se le da la posibilidad a otros ejecutores de desafiar resultados publicados, ejecutando el código esta vez onchain, y así obteniendo un resultado determinista. Si un ejecutor desafía a otro, y se descubre que este último publicó un resultado incorrecto, la recompensa se la lleva el desafiante. Este sistema trae varias cuestiones adicionales a resolver.

En principio, es necesario que el código sea determinista, es decir, que siempre dé el mismo resultado ante las mismas condiciones de entrada. No puede haber factores externos como llamadas a APIs o lectura de información de ningún contrato, ya que los valores de respuesta de estas operaciones pueden variar dependiendo del momento y contexto en el que se realicen. Tampoco puede haber ningún ingreso de datos dinámico no contemplado en las condiciones de entrada iniciales. Por último, tampoco se puede usar el timestamp, número de bloque, o cualquier otro dato que varíe con la transacción. Para garantizar esto, se creó un contrato abstracto “BaseClient”, con el fin de estandarizar la interacción con el contrato de administración y distribución de peticiones “ExecutionBroker”. En esta plantilla de contrato cliente, se definen funciones inmutables mediante las cuales se interactúa con el contrato broker, y plantillas de funciones cuya lógica deberá ser definida por cada cliente, dentro de los límites establecidos por dicha plantilla.

Entre las plantillas extensibles se encuentra la función que poseerá toda la lógica del cliente, definiendo el formato de entrada y salida, y con el modificador “pure”, que garantiza el comportamiento determinista. También se encuentra una función de procesamiento de resultados, en caso de que el cliente desee que se ejecute alguna lógica adicional al finalizarse una petición (como lo puede ser la creación automática de otra petición). Por último, el cliente debe implementar una función que retorne una descripción de la estructura de las variables de entrada a la función de lógica, para facilitar la decodificación por parte de los ejecutores.

Entre las funciones inmutables están la de creación y cancelación de una petición. Estas deben tener un comportamiento fijo para evitar que se creen peticiones con un formato erróneo, y permitirle al contrato broker a tener una referencia al contrato cliente. También se encuentran definidas de manera fija las funciones para desafiar un resultado, y recolectar el pago por una petición resuelta. Dado que tanto el reclamo de un pago, como el desafío de un resultado siempre cierran el ciclo de vida de una petición, esto significa que, en caso de haber lógica de procesamiento de resultados por parte del cliente, se ejecutaría en estos casos. El motivo por el cuál las interacciones que dan lugar al cierre de una petición se den desde el contrato cliente y no directamente desde el contrato broker, es debido a la existencia de una vulnerabilidad de suplantación de identidad. Por diseño del lenguaje Solidity, cuando un contrato llama a una función de otro contrato, se reemplaza el valor de “msg.sender” (la dirección de quien invocó a la función actual), por la dirección del contrato llamante. Esto se puede evitar haciendo que el llamado sea de un tipo especial llamado “delegate call”, en cuyo caso el valor de “msg.sender” permanece inmutado. Muchos mecanismos de seguridad consisten en agregarle un “modifier” personalizado a ciertas funciones, de manera que solo puedan ser llamados por una dirección en particular. Por este motivo, si la lógica de desafío y recolección de recompensa está en su totalidad en el contrato broker, en especial la parte de llamado a la lógica de procesamiento de resultados, el contrato broker haría la llamada a dicha función de procesamiento, reemplazando el valor de “msg.sender” por su propia dirección, y un cliente malicioso podría después efectuar un “delegate call” para impersonar a el contrato broker. Esto se resuelve haciendo que ambas interacciones que

le dan cierre a una petición, y que por ende desembocan en el llamado de la función de procesamiento de resultados, se efectúen a través del contrato del cliente, con lógica definida de manera inmutable en el contrato “Base Client”. De esta manera, un contrato cliente malicioso solo se podrá impersonar a sí mismo. Este problema no sucede con el llamado a la función de lógica del cliente, ya que esta está restringida como “pure” y, por ende, solo puede realizar cálculos endógenos, y no impersonar a nadie.

Otra cuestión a tener en cuenta debido a la introducción de lógica de procesado de resultados, es el tema del gas. Si la función de procesado es muy compleja, podría consumir demasiado gas haciendo que cerrar una petición no sea rentable. Por este motivo, la llamada a dicha función de procesado se hace con una restricción de gas, mediante la cual ese “sub-llamado” solo podrá consumir una cantidad fija definida por la petición misma. Si la función se pasa de esa cantidad, falla el procesado, pero no la transacción general, dándole cierre a la petición de igual manera.

Volviendo a las problemáticas que nacen de la necesidad de garantizar la integridad de los cálculos hechos por fuera de la cadena (offchain), la segunda cuestión a tener en cuenta es que cualquier nodo podría “robar” los resultados anunciados por otro ejecutor, y publicarlos antes. Esto se resuelve dividiendo la ejecución en dos pasos. Primero el ejecutor acepta una petición, reservándola temporalmente, y luego publica los resultados.

La tercera problemática surge de que, si un ejecutor cobra inmediatamente después de publicar un resultado, entonces podría publicar datos al azar sin realizar el cómputo requerido por la petición. En este caso, no importa si hay un desafío posterior, porque el ejecutor inicial ya habrá cobrado su recompensa. La solución consiste en poner un retardo, a elegir por el cliente, que dictamine el tiempo mínimo que tiene que pasar para dar lugar a que otros nodos puedan desafiar el resultado publicado, antes de que el ejecutor inicial pueda cobrar ya que, una vez realizado el cobro, se da por cerrada o solidificada la petición.

La cuarta cuestión radica en que ejecutar el código onchain, puede y suele ser muy caro, al punto de que al desafiante no le sería rentable efectuar un desafío. Esto se soluciona pidiendo que el ejecutor inicial ponga una prima lo suficientemente grande cuando acepta la petición, para poder reintegrarle el gas gastado a un desafiante en caso de ser necesario. De no haber desafío, se le devuelve la prima junto con la recompensa cuando recolecta el pago, después de que haya transcurrido el tiempo mínimo establecido inicialmente por el cliente en la petición. Una nota interesante radica en que la medición de tiempo en una blockchain, se realiza mediante la lectura de la variable “block.timestamp”. Dicho valor es confiable dentro del margen de tiempo promedio de creación de bloques. Es decir, concretamente en Ethereum, para plazos mayores a 12 segundos (tiempo promedio de bloque), se puede utilizar este valor.

La quinta cuestión está relacionada con que si la petición conlleva un volumen de computo muy grande, hay poco incentivo para que la gente corrobore los resultados para un eventual desafío. Esto se resuelve haciendo que los ejecutores se puedan quitarse la aceptación de una petición entre sí, en caso de que el ejecutor aceptante inicial tarde más de unos minutos en publicar los resultados. Sin esta posibilidad, un ejecutor podría aceptar la petición y después ponerse a realizar el procesamiento, con la tranquilidad de que nadie va a competir porque ya tiene la petición bloqueada indefinidamente. Implementando un sistema de vencimiento de aceptación, primero se calcula el resultado, compitiendo con el resto, y después en una sucesión rápida, se acepta la petición y se publican los resultados obtenidos. Esto genera la posibilidad de que haya otros ejecutores que también hayan hecho el cálculo, pero hayan sido más lentos al intentar aceptar y publicar, y por ende tengan el resultado a mano para corroborar que sea el mismo que el publicado. El periodo de gracia de una aceptación debe ser lo suficientemente corto como para impedir que la gente bloquee la petición antes de empezar el cálculo, y lo suficientemente largo como para que sea inviable que un nodo malicioso robe y republique los resultados de otro ejecutor como suyos. También, para que los esfuerzos de los otros ejecutores no hayan sido en vano, y a modo de incentivo a que varios ejecutores intenten calcular el resultado a la vez, hay una cantidad definida de confirmadores permitidos, es decir, que un ejecutor puede confirmar que el resultado

de otro ejecutor es correcto, dejando una fracción del seguro, pero postulándose para ganar una fracción de la comisión pagada por el cliente.

Por último, en cuanto al cobro de la recompensa, una vez pasado el tiempo de garantía establecido por el cliente, el ejecutor principal hace una llamada a la función “claimPayment”, a través del contrato cliente por los motivos explicados anteriormente. Algo a tener en cuenta es que en solidity, una transferencia también es un llamado a una función y, por consiguiente, si el recipiente de la transferencia es un contrato, esto puede desencadenar una serie de instrucciones que podrían, bajo las circunstancias correctas, terminar llamando nuevamente a la función que inició dicha transferencia en primer lugar, creándose así un bucle. Si no se tiene en cuenta esta mecánica a la hora de programar lógica de transferencias, esto puede generar una vulnerabilidad llamada “reentrancy vulnerability” [28], mediante la cual un contrato atacante que simplemente desea retirar fondos de otro contrato, llama a dicha función, y cuando recibe la transferencia, ejecuta nuevamente la petición de retiro de fondos. Si el contrato víctima, por ejemplo, no actualizó los balances antes de realizar la transferencia, la nueva entrada a la función vería el balance original, y se volvería a realizar la transferencia, logrando así que el contrato atacante retire más fondos de los que poseía. Concretamente, en el caso de este protocolo, esta vulnerabilidad se mitiga cerrando la petición, cambiando el valor de una bandera, antes de realizar cualquier pago. Esto, complementado con un chequeo que solo ejecute el cierre si la petición no ha sido cerrada aún, eliminan por completo esta vulnerabilidad.

### 5.1.3 Estructura de los Contratos

Esta solución tiene un total de 1 contrato desplegable y 1 contrato extensible. El contrato principal y único desplegable directamente es el de **ExecutionBroker**. Este contrato tiene la finalidad de almacenar y administrar la información relacionada a las peticiones y requerimientos de los clientes, a la vez de regular el intercambio de información y ether entre clientes y ejecutores. El contrato abstracto y extensible es el de **BaseClient**. Este contrato ofrece una plantilla de implementación para los casos de uso particulares de los

clientes, y dicha plantilla es la que funciona como interfaz para que dicho contrato se comunique con el broker.

La estructura del contrato **ExecutionBroker** es la siguiente:

**Estructuras definidas:**

- Request: representa una petición de un cliente determinado en el sistema. Almacena información como el ID de la petición, la información de entrada para la ejecución, el pago, el gas máximo de procesamiento de resultados, el seguro de desafío requerido, el tiempo mínimo de espera para poder cobrar una recompensa, y el estado de la petición en cuanto a su cancelación, aceptación por parte de los ejecutores, la presentación de resultados de los mismos, y las confirmaciones de ejecutores pares.
- Acceptance: representa el compromiso entre un ejecutor y una petición. Posee información como la dirección del ejecutor y el timestamp de aceptación.
- Submission: representa una entrega de resultados de parte del ejecutor aceptador. Tiene información como la dirección del ejecutor (nótese que debe coincidir con la dirección del aceptador dentro de una misma petición, salvo en caso de desafío), el timestamp de entrega, el resultado de la ejecución, y el estado de la entrega en cuanto a su solidificación.

**Variables internas del contrato:**

- uint ACCEPTANCE\_GRACE\_PERIOD: representa el periodo de tiempo máximo (expresado en segundos) en el cual la aceptación de una petición no puede ser reemplazada por otro ejecutor.
- uint8 CONFIRMERS\_FEE\_PERCENTAGE: representa tanto el porcentaje del pago a cobrar por los ejecutores confirmantes de un resultado, como el porcentaje del seguro de desafío a ser dejado por dichos confirmantes.
- uint8 AMOUNT\_OF\_CONFIRMERS: representa la cantidad máxima de confirmantes que puede tener una petición.
- Request[] requests: representa la colección de peticiones de los diversos clientes.

**Eventos:**

- requestCreated: se dispara cuando se crea una petición. Contiene el ID de la petición, el pago, el gas máximo de procesamiento del resultado, y la cifra del seguro de desafío.
- requestCancelled: se dispara cuando se cancela una petición. Contiene el ID de la petición y una bandera indicando si se le reintegró el dinero al cliente exitosamente.

- requestAccepted: se dispara cuando un ejecutor acepta una petición. Contiene el ID de la petición y la dirección del ejecutor.
- acceptanceCancelled: se dispara cuando un ejecutor cancela una aceptación. Contiene el ID de la petición y una bandera indicando si se le reintegró el dinero del seguro al ejecutor exitosamente.
- resultSubmitted: se dispara cuando un ejecutor publica los resultados de una petición. Contiene el ID de la petición y la dirección del ejecutor.
- resultConfirmed: se dispara cuando un ejecutor confirma el resultado de otro ejecutor. Contiene el ID de la petición.
- requestSolidified: se dispara cuando el resultado de una petición se solidifica, es decir, que ya se le transfirió el pago a el ejecutor y a los confirmantes y, por ende, dicho resultado ya no puede ser desafiado. Contiene el ID de la petición.
- challengeProcessed: se dispara cuando se lleva a cabo el desafío de un resultado publicado. Contiene el ID de la petición y el resultado de la ejecución.
- challengePayment: se dispara cuando se efectúa el pago como consecuencia de un desafío, ya sea en favor del desafiante, o del ejecutor original. Contiene el ID de la petición, la dirección del destinatario del pago, y una bandera indicando si la transferencia fue exitosa.

### **Funciones públicas:**

- submitRequest [Payable]: Crea una petición, la agrega a la colección de peticiones, y dispara el evento “requestCreated”. Recibe la información necesaria para crear una petición y los fondos para los pagos correspondientes. Solo puede ser llamada por una instancia de un contrato cliente (que herede de BaseClient). Retorna el ID de la petición creada.
- cancelRequest: Cancela una petición, le reintegra el dinero al cliente, y dispara el evento “requestCancelled”. Recibe el ID de la petición a cancelar. Solo puede ser llamada por el contrato cliente que creó la petición, y solamente si esta petición está activa y no se encuentra aceptada por un ejecutor.
- publicizeRequest: vuelve a emitir el evento “requestCreated” para una petición determinada. Recibe el ID de la request a publicitar. Puede ser llamada por cualquiera.
- acceptRequest [Payable]: Registra la aceptación de un ejecutor sobre una petición y dispara el evento “requestAccepted”. Recibe el ID de la petición a ser aceptada y los fondos del seguro de desafío. Solo puede ser llamada sobre peticiones que no hayan sido canceladas, que no tengan ninguna entrega, y que no tengan una aceptación más reciente que el valor establecido por ACCEPTANCE\_GRACE\_PERIOD.
- cancelAcceptance: Cancela la aceptación de una petición devuelve el seguro de desafío al ejecutor. Emite el evento “acceptanceCancelled”. Recibe el ID de la

petición a cancelar. Solo puede ser llamada por el ejecutor aceptante, y solo si no se publicó ningún resultado aún.

- submitResult: registra un resultado en una petición y dispara el evento “resultSubmitted”. Recibe el ID de la petición, y el resultado codificado en bytes. Solo puede ser llamada por el ejecutor aceptante, y si no hay ningún resultado publicado aún.
- confirmResult [Payable]: añade al llamante de la función a la lista de confirmadores del resultado de una petición. Recibe el ID de la petición. Solo puede ser llamado ante una petición que tenga un resultado no solidificado, que no haya llegado al número máximo de confirmaciones, y cuyo resultado no haya sido publicado por el llamante. También se debe enviar un pago proporcional al seguro de desafío total.
- challengeSubmission: realiza la ejecución de la lógica del cliente de manera determinista (onchain) y compara el resultado con lo publicado por el ejecutor. Si el resultado coincide, se le paga al ejecutor inicial y a sus confirmantes, y se devuelven los seguros. En caso contrario, se le paga al desafiante y este, también se lleva los seguros como compensación por la gran cantidad de gas gastado. En ambos casos, solidifica el resultado de la petición. Emite los eventos “challengeProcessed” y “challengePayment”. Recibe el ID de la petición. Solo puede ser llamada sobre una petición con un resultado publicado sin solidificar, y solo a través del contrato cliente. Retorna verdadero si el desafío fue exitoso, y falso si el ejecutor original había publicado el resultado correcto.
- claimPayment: Solidifica el resultado de la petición, y le envía el pago y los seguros al ejecutor y los confirmantes. Emite el evento “requestSolidified”. Recibe el ID de la petición. Solo puede ser llamada sobre una petición con un resultado publicado, por el ejecutor que publicó ese resultado a través del contrato cliente, y solo cuando haya pasado el tiempo de garantía mínimo establecido por la petición. Retorna una bandera indicando si la transferencia de fondos al ejecutor fue exitosa.

La estructura del contrato **BaseClient** es la siguiente:

**Estructuras definidas:**

- ClientInput: posee un arreglo dinámico de bytes representando los datos de entrada para la lógica a ejecutarse en cada petición, y un número indicando cuál de las funciones internas de lógica se debe ejecutar. Este número le permite al cliente darle más versatilidad a su contrato, pudiendo hacer varias peticiones distintas con el mismo, en lugar de necesitar un contrato distinto para cada tipo de cálculo.

**Variables internas del contrato:**

- ExecutionBroker brokerContract: es una referencia al contrato broker, utilizada para poder interactuar con el mismo. Dicha referencia es configurada en el momento de creación del contrato cliente.
- mapping(uint => bool) activeRequestIDs: un registro key-value donde se almacena información sobre qué índices de petición pertenecen a peticiones activas del cliente.

**Eventos:**

- requestSubmitted: se dispara cuando se crea efectivamente una nueva petición. Contiene el ID de la petición.
- resultProcessed: se dispara luego de haberse procesado el resultado de una petición, antes de cerrarse. Contiene el ID de la petición y una bandera indicando si el procesamiento fue exitoso.

**Modificadores:**

- onlyClient: las funciones marcadas con este modificador, sólo pueden ser llamadas internamente y por la misma instancia.

**Funciones extensibles (virtuales):**

- clientLogic [pure]: función determinista que contiene la lógica a ser procesada por los nodos ejecutores. Recibe una estructura del tipo “ClientInput”. Retorna un arreglo de bytes representando el resultado obtenido por el ejecutor.
- processResult: función a ser llamada en el momento de cerrar una petición, encargada de ejecutar la lógica final con el resultado entregado. Se puede utilizar para crear una petición nueva. Recibe un arreglo de bytes con los datos codificados del resultado de la petición. Solo puede ser llamada por dentro del mismo contrato.
- getInputStructure [pure]: recibe el ID de la función sobre la cual se quiere conocer la estructura de los datos de entrada. Retorna un string representando el formato de la estructura de los datos de entrada de una función.

**Funciones inmutables:**

- submitRequest [Payable]: llama a la función “submitRequest” del contrato broker, y dispara el evento “requestSubmitted”. Recibe la información necesaria para crear una petición y puede recibir los fondos para los pagos correspondientes o tomarlos implícitamente del balance del contrato. Solo puede ser llamada por el dueño del contrato cliente. Retorna el ID de la petición creada.

- cancelRequest: llama a la función “cancelRequest” del contrato broker. Recibe el ID de la petición a cancelar. Solo puede ser llamada por el dueño del contrato cliente, es decir, quien lo instanció originalmente.
- challengeSubmission: llama a la función “challengeSubmission” del contrato broker, luego ejecuta la lógica de procesamiento de resultados, y finalmente emite el evento “resultProcessed”. Recibe el ID de la petición. Solo puede ser llamada sobre una petición activa perteneciente al contrato cliente particular, y solo si este tiene un resultado publicado sin solidificar. Retorna verdadero si el desafío fue exitoso, y falso si el ejecutor original había publicado el resultado correcto.
- claimPayment: llama a la función “claimPayment” del contrato broker, luego ejecuta la lógica de procesamiento de resultados, y finalmente emite el evento “resultProcessed”. Recibe el ID de la petición. Solo puede ser llamada sobre una petición activa perteneciente al contrato cliente particular, y solo si este tiene un resultado publicado sin solidificar. También solo puede ser llamada por el ejecutor que publicó el resultado actual, y únicamente habiendo pasado el plazo mínimo de garantía establecido por el cliente en la petición. Retorna una bandera indicando si la transferencia de fondos al ejecutor fue exitosa.

#### 5.1.4 Ciclo de vida de una petición

Como se muestra en la Figura **FIGURA I** a continuación, el ciclo de vida de una petición comienza cuando un cliente la crea a través del Contrato Cliente (paso 1, flecha blanca). Esta llamada desemboca en una llamada interna a la función homónima de creación de petición en el Contrato Broker (paso 1, flecha blanca). Como resultado, se crea una petición con todos los datos y requerimientos del cliente, y se emite un evento que le deja saber a los ejecutores que una nueva petición fue creada.

Posteriormente, los ejecutores comienzan a realizar el procesamiento requerido por la petición, y cuando alguno llegue al resultado final, procederá a aceptar dicha petición (a través del contrato broker), poniendo la cantidad requerida de ether como seguro de desafío y reservando de esta manera temporalmente el derecho a publicar un resultado (paso 2, flecha blanca). Luego de que se confirme su reserva, el ejecutor publica los resultados calculados (paso 3, flecha blanca), a través también del contrato broker. A partir de este momento, otros ejecutores que también hayan realizado el computo requerido, pero que no hayan llegado a aceptar la petición primeros, pueden valerse del resultado calculado para confirmar (a través del contrato broker) la publicación oficial en caso de que coincida (paso 4, flecha blanca) dejando suma proporcional al

seguro de desafío, o arriesgarse a desafiar (a través del contrato cliente) al ejecutor original (paso 5.b, flecha roja). Esto último desemboca en un llamado a la función de desafío homónima dentro del contrato broker (paso 5.b, flecha roja). En este caso, si el desafío es llevado a cabo de manera correcta, se utiliza el seguro del primer ejecutor para reintegrarle el gas al ejecutor desafiante, y se le añade la paga de la petición. Si el desafío se lleva a cabo de manera incorrecta, es decir, si el ejecutor original había publicado un resultado correcto, simplemente se le reintegra el seguro al ejecutor original, junto con la paga de la petición. Independientemente del resultado del desafío, se llama a la función de procesado de resultado dentro del contrato cliente (paso 5.b, flecha roja), y se cierra la petición, solidificando el resultado.

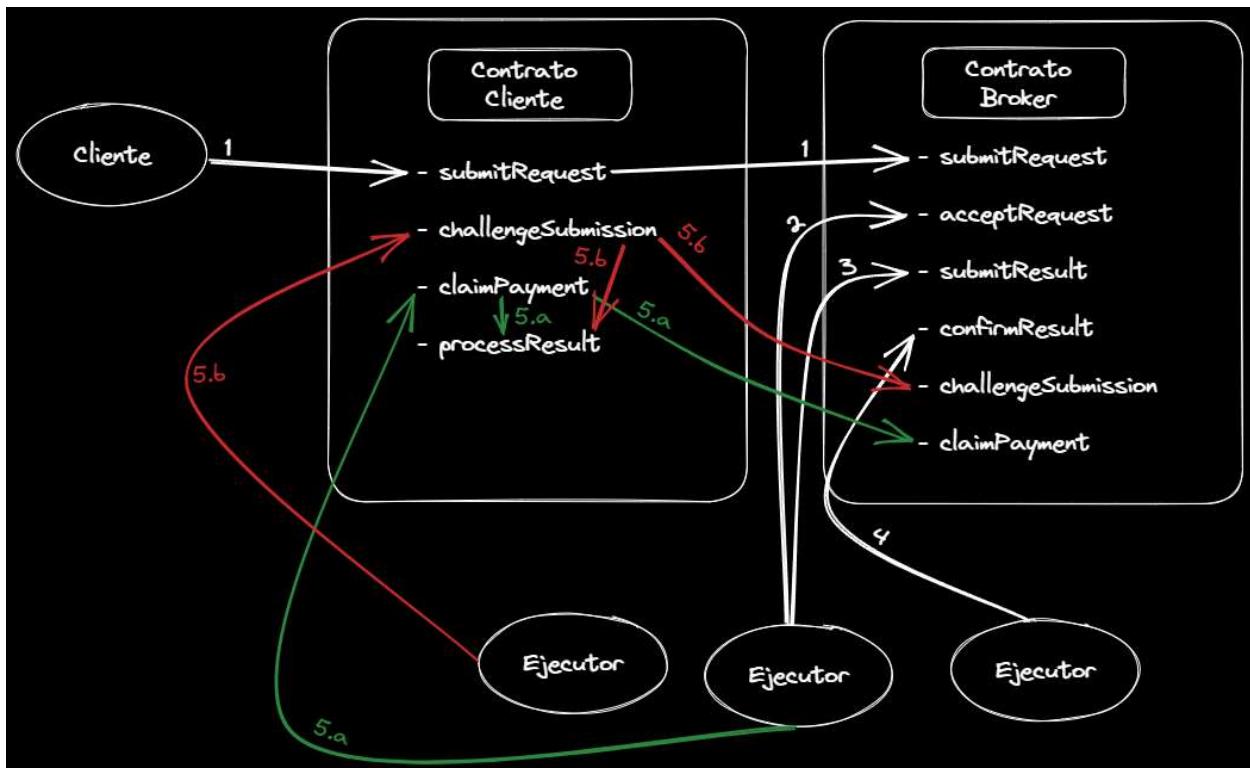


FIGURA I: Ciclo de vida de una petición en Optimistic Execution.

Por otro lado, si nadie desafía el resultado publicado, y una vez transcurrido el tiempo de espera mínimo establecido por el cliente en la petición, el ejecutor original puede llamar a la función de recolección de pago (paso 5.a, flecha verde). Esto desemboca en el llamado a la función homónima del contrato broker (paso 5.a, flecha verde), que le reintegra el seguro de desafío al ejecutor original junto con la paga por la petición.

También, en este paso se les reintegra el seguro parcial a los ejecutores confirmantes, junto con la paga proporcional. Por último, se llama a la función de procesado de resultado dentro del contrato cliente (paso 5.a, flecha verde), y se cierra la petición, solidificando el resultado.

### **5.1.5 Ejemplo de uso: Números primos**

Para realizar pruebas sobre el protocolo y poder hacer mediciones de los costos de cada operación, se utilizó el framework web3 “Brownie” para el lenguaje de programación python 3. Los contratos fueron desplegados en una red Ethereum local utilizando la herramienta Ganache. El código de desarrollo de los contratos y pruebas, del cual se tomaron los fragmentos mostrados en las diferentes figuras a lo largo de la sección de desarrollo y experimentación de este trabajo, se encuentra en el repositorio de GitHub [29].

Una implementación práctica para demostrar la utilidad de este protocolo es un contrato para la búsqueda de números primos. En cada petición, este contrato brinda un número inicial, a elegirse por el cliente, y busca todos los números primos a partir de este número, dentro de un rango definido también por el cliente. El valor del resultado de la ejecución, consiste en un arreglo codificado de todos los números primos hallados en ese rango, además del último número analizado. La razón para retornar siempre el último número analizado es para que la lógica de procesamiento de resultados tenga un punto de referencia respecto de cuál debería ser el próximo valor a analizar. Luego, la función de procesamiento de datos toma dicho arreglo, y almacena automáticamente en el contrato todos los números menos el último, al cual se le realiza una prueba de primalidad para decidir si se almacena o no. Posteriormente, se crea una nueva petición, en la cual el número inicial es el número siguiente al último número analizado.

Se puede observar cómo, para rangos de números muy grandes, el procesamiento delegado es enormemente más barato, ya que, si el rango de búsqueda es de 1000 números, se procesa el análisis de primalidad de esos 1000 números offchain, mientras que onchain solo se procesa el almacenamiento de los números encontrados.

```
function findPrimes(uint256 limit) public returns (uint[] memory) {
    uint i;
    for (i = PRIMES[PRIMES.length - 1] + 1; i < limit; i += 2) {
        if (_isPrime(i)) {
            PRIMES.push(i);
        }
    }
    return PRIMES;
}

function _isPrime(uint256 number) private pure returns (bool) {
    if (number < 2 || (number % 2 == 0 && number > 2)) {
        return false;
    }
    for (uint i = 3; i < number/2; i += 2) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

FIGURA II: código en solidity para la búsqueda de números primos.

Concretamente, para encontrar todos los números primos menores que 1000, en el caso de intentar resolverlo con un solo llamado onchain, se requiere de 298.638 unidades de gas para el despliegue inicial del contrato, y otras 8.929.056 unidades para el cálculo de los números, llevándonos a un total de 9.227.694 unidades (como se observa en las Figuras **FIGURA III**, **FIGURA III** y **FIGURA IV**).

```
def test_find_first_primes_with_gas_onchain(self):
    self.reference_gas_price = 58160642908 # 58 GWei
    account = accounts[0]
    contract = OnChainPrimeFinder.deploy({"from": account, "gas_price": self.reference_gas_price})
    tx = contract.findPrimes(1000, {"from": account, "gas_price": self.reference_gas_price})
    print(f"Reference gas price: {self.reference_gas_price}")
    print(f"Deployment gas used: {contract.tx.gas_used}")
    print(f"Deployment cost: {contract.tx.gas_used * self.reference_gas_price}")
    print(f"Calculation gas used: {tx.gas_used}")
    print(f"Calculation cost: {tx.gas_used * self.reference_gas_price}")
```

FIGURA III: código en python para testear costos de ejecución onchain.

```
----- Captured stdout call -----
Reference gas price: 58160642908
Deployment gas used: 298638
Deployment cost: 17368978076759304
Calculation gas used: 8929056
Calculation cost: 519319637521534848
===== short test summary info =====
```

FIGURA IV: impresión de la ejecución del test de la Figura FIGURA III.

A la fecha (20 de noviembre 2023), el precio promedio del gas en ethereum es de 58160642908 Wei, es decir 0,00000058160642908 Ether. A su vez, cada Ether posee actualmente un valor de 2,086.46 USD [3]. Esto significa que onchain, el gasto total para realizar el cálculo estipulado al día de hoy es de 1.119,80 USD.

Por otra parte, si se intenta realizar el mismo calculo a través del protocolo Optimistic Execution, el flujo de resolución se da de la siguiente manera. Primero el cliente debe desplegar su contrato con la lógica a ser ejecutada. En este ejemplo en concreto, el costo inicial de despliegue consume 1.726.978 de gas. Luego, se debe crear una petición, proceso el cual cuesta otras 263.365 unidades (como se observa en las Figuras **FIGURA V**, **FIGURA VI** y **FIGURA VII** a continuación). En sumatoria son 1.990.343 unidades, y a los precios tomados como referencia en la sección anterior, el cliente tendría un gasto inicial de 241,50 USD, sin tener en cuenta la paga para los ejecutores, cuyo valor se determinará por oferta y demanda. Lo seguro es que, teniendo la alternativa Onchain a 1.119,80 USD, no tendría sentido que la paga para el ejecutor delegado supere la diferencia entre el costo total del método onchain y el gasto inicial de la petición, es decir 878,30 USD.

```
function clientLogic(clientInput calldata input) external override pure returns (bytes memory) {
    bytes memory output = "";
    if (input.functionToRun == 1) { output = getPrime(input.data); }
    else if (input.functionToRun == 2) {output = isPrime(input.data); }
    return output;
}

struct OneInput {uint256 startPoint; uint256 batchSize;}
function getPrime(bytes memory data) private pure returns (bytes memory) {
    OneInput memory input = abi.decode(data, (OneInput));
    return _getPrime(input.startPoint, input.batchSize);
}
function _getPrime(uint256 _startPoint, uint256 _batchSize) private pure returns (bytes memory) {
    uint256 i;
    bytes memory result;
    for (i = _startPoint | 1; i < _startPoint + _batchSize; i += 2) {
        if (_isPrime(i)) {
            result = abi.encodePacked(result, i);
        }
    }
    result = abi.encodePacked(result, i);
    return result;
}

struct TwoInput {uint256 number;}
function isPrime(bytes memory data) private pure returns (bytes memory) {
    TwoInput memory input = abi.decode(data, (TwoInput));
    return abi.encode(_isPrime(input.number));
}
function _isPrime(uint256 _number) private pure returns (bool) {
    if (_number < 2 || (_number % 2 == 0 && _number > 2)) {
        return false;
    }
    for (uint i = 3; i < _number/2; i += 2) {
        if (_number % i == 0) {
            return false;
        }
    }
    return true;
}
```

FIGURA V: código en solidity para la búsqueda de números primos de forma delegada.

```

def test_find_first_primes_with_gas_offchain(self):
    broker = BrokerFactory.create(account=Accounts.getFromIndex(0))
    requestorAccount = Accounts.getFromIndex(1)
    initialFunds = requestorAccount.balance()
    clientContract = ClientFactory.create(broker, owner=requestorAccount, gas_price=self.reference_gas_price)
    requestor = Requestor(clientContract)
    deploymentGas = (initialFunds - requestorAccount.balance()) // self.reference_gas_price
    request = requestor.createRequest(functionToRun=1, dataArray=[3, 1000], payment=0, postProcessingGas=0,
                                      requestedInsurance=2e14, gas_price=self.reference_gas_price, getTransaction=True)
    reqID = request.return_value
    executorAccount = Accounts.getFromIndex(0)
    executor = Executor(executorAccount, broker, populateBuffers=True)
    acceptanceTransaction = executor._acceptRequest(reqID, gas_price=self.reference_gas_price)
    result = executor._computeResult(reqID)
    submissionTransaction = executor._submitResult(reqID, result, gas_price=self.reference_gas_price)
    confirmationInsurance = broker.requests(reqID).dict()["challengeInsurance"] // broker.CONFIRMERS_FEE_PERCENTAGE()
    confirmationTransaction1 = broker.confirmResult(reqID, {"from": Accounts.getFromIndex(2), "value": confirmationInsurance})
    confirmationTransaction2 = broker.confirmResult(reqID, {"from": Accounts.getFromIndex(3), "value": confirmationInsurance})
    paymentTransaction = executor._claimPayment(reqID, gas_price=self.reference_gas_price)
    print(f"Deployment gas: {deploymentGas}")
    print(f"Request creation gas: {request.gas_used}")
    print("-----")
    print(f"Acceptance gas: {acceptanceTransaction.gas_used}")
    print(f"Submission gas: {submissionTransaction.gas_used}")
    print(f"First confirmation gas: {confirmationTransaction1.gas_used}")
    print(f"Second confirmation gas: {confirmationTransaction2.gas_used}")
    print(f"Payment gas: {paymentTransaction.gas_used}")
    print("-----")
    print(clientContract.getPrimes())

```

FIGURA VI: código en python para testear costos de ejecución optimista delegada.

```

----- Captured stdout call -----
Deployment gas: 1726978
Request creation gas: 263365
-----
Acceptance gas: 74256
Submission gas: 3523943
First confirmation gas: 62982
Second confirmation gas: 69149
Payment gas: 328723
-----
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113
, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251
, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397
, 401, 409, 419, 421, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557
, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701
, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863
, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]
===== short test summary info =====

```

FIGURA VII: impresión de la ejecución del test de la Figura FIGURA VI.

Continuando con el flujo de la petición, por parte del ejecutor, este tiene algunos costos implícitos relacionados a la interacción con el contrato broker a la hora de resolverla. En primer lugar, debe aceptarla, lo que lo habilita a publicar un resultado para la misma posteriormente. Para este caso, el gas de aceptación es de 74.256. Una vez aceptada la petición, sigue la publicación de resultados, que en este ejemplo tiene un costo de 3.523.943 de gas. El costo de este paso, depende del tamaño del resultado a publicarse. En este caso se guardan 168 números de 256 bits cada uno (5.376 Bytes en total). Adicionalmente, luego de haberse publicado un resultado, puede haber un numero de

confirmaciones por parte de ejecutores que realizaron el cálculo, pero no fueron lo suficientemente rápidos como para aceptar la petición y publicarlos. En el ejemplo, hay dos confirmaciones, costando 62.982 y 69.149 unidades respectivamente. Finalmente, habiendo esperado el tiempo de garantía establecido por el cliente para esa petición, el ejecutor puede efectuar recolección de la paga, proceso con un gasto de 328.723 unidades. En sumatoria, el costo total en gas de los procesos de ejecución (suponiendo que no hubo ningún desafío), es de 4.059.053 unidades de gas (Figura **FIGURA VII**). Conservando los precios de referencia, esto lleva a una suma de 492,50 USD. Esto representa el pago mínimo que un ejecutor estará dispuesto a aceptar (sin tener en cuenta factores como el gasto energético, ni el riesgo asumido al bloquear una petición), para que una ejecución le sea rentable. Siendo este el piso de la paga para los ejecutores, y los 878,30 USD (calculados en la sección anterior) el techo de dicha paga, se puede calcular que, en el mejor de los casos para el cliente, solo tendría que pagar el costo base del protocolo, más el piso de 492,50 USD, sumando 734 USD en total. Además, en el caso promedio (entre el techo y piso de la paga de los ejecutores), para este cálculo en particular, el cliente estaría pagando 685.40 USD a los ejecutores, que sumados al costo base, se llega a los 926.90 USD.

En conclusión, para este cálculo en específico, en caso promedio para el cliente, estaría pagando un 83% del monto requerido por la alternativa onchain. En el mejor de los casos para el cliente, paga solo un 66%.

Cabe destacar que en cálculos más grandes (por ejemplo, calcular los primeros 100.000 números primos), la diferencia porcentual entre la ejecución onchain y offchain crece muchísimo, debido a que los gastos más importantes en el protocolo provienen de la interacción con el contrato broker, están compuestos por una parte de cálculo de distribución de tareas fija, y otra que solo escala con el tamaño de la respuesta a publicar (ya que entre mayor sea el volumen de datos a registrar en la cadena, mayor cantidad de operaciones de escritura habrán) [12]. Sin embargo, debido a las limitaciones de las EVM y concretamente, a los simuladores de EVM que se utilizaron para los cálculos en este trabajo, la comparativa onchain es más difícil y hasta a veces imposible de simular.

Por otro lado, si se porta este protocolo hacia otra blockchain compatible con EVM, se obtienen precios de gas mucho menores [30], manteniendo la diferencia porcentual entre ejecución onchain y offchain, pero llegando a un valor en USD mucho menor. Por este motivo, para los siguientes análisis, se tomará un costo de gas simbólico de 1 ya que, a la hora de comparar la eficiencia de cada método, solo importa la variación del gas usado, y no el precio del mismo.

Para calcular los gastos generales dentro del protocolo Optimistic Execution, independientemente de la complejidad tamaño de ejecución de una petición, se creó una petición simbólica en la que el resultado es trivial. De esta manera, al medir el gas gastado solo estamos teniendo en cuenta dichos gastos generales, sin importar el tamaño de la respuesta, ya que el costo de publicación incrementa de manera directamente proporcional a dicho tamaño. Es importante la medición de los gastos generales del protocolo para tener una estadística de comparación que es totalmente independiente de la implementación específica de cada cliente.

Según las mediciones hechas sobre la ejecución de una petición trivial, los gastos de creación de una petición por parte del cliente son de aproximadamente 223.000 unidades de gas (como se observa en las Figuras **FIGURA VIII** y **FIGURA IX** a continuación). La diferencia de 57.000 unidades con el caso de números primos se debe a que en este caso los datos de entrada de la petición son nulos, y por ende no hay costo de creación de la estructura “ClientInput” en las reiteradas veces que aparece como parámetro de una función. Como todas las peticiones reales necesitaran indefectiblemente de una estructura de datos de entrada, para el análisis comparativo se tomó como gasto base de creación, la cifra obtenida en la Figura **FIGURA VII**, es decir, unas 260.000 unidades de gas.

```
function clientLogic(ClientInput calldata input) external override pure returns (bytes memory) {
    bytes memory output = "";
    if (input.functionToRun == 1) { output = trivialFunction(input.data); }
    return output;
}

struct OneInput {uint256 trivialData;}
function trivialFunction(bytes memory data) private pure returns (bytes memory) {
    return data;
}
```

FIGURA VIII: lógica de un contrato cliente trivial.

```
----- Captured stdout call -----
Deployment gas: 1257875
Request creation gas: 223728

-----
Acceptance gas: 74256
Submission gas: 76581
First confirmation gas: 62982
Second confirmation gas: 69149
Payment gas: 136041

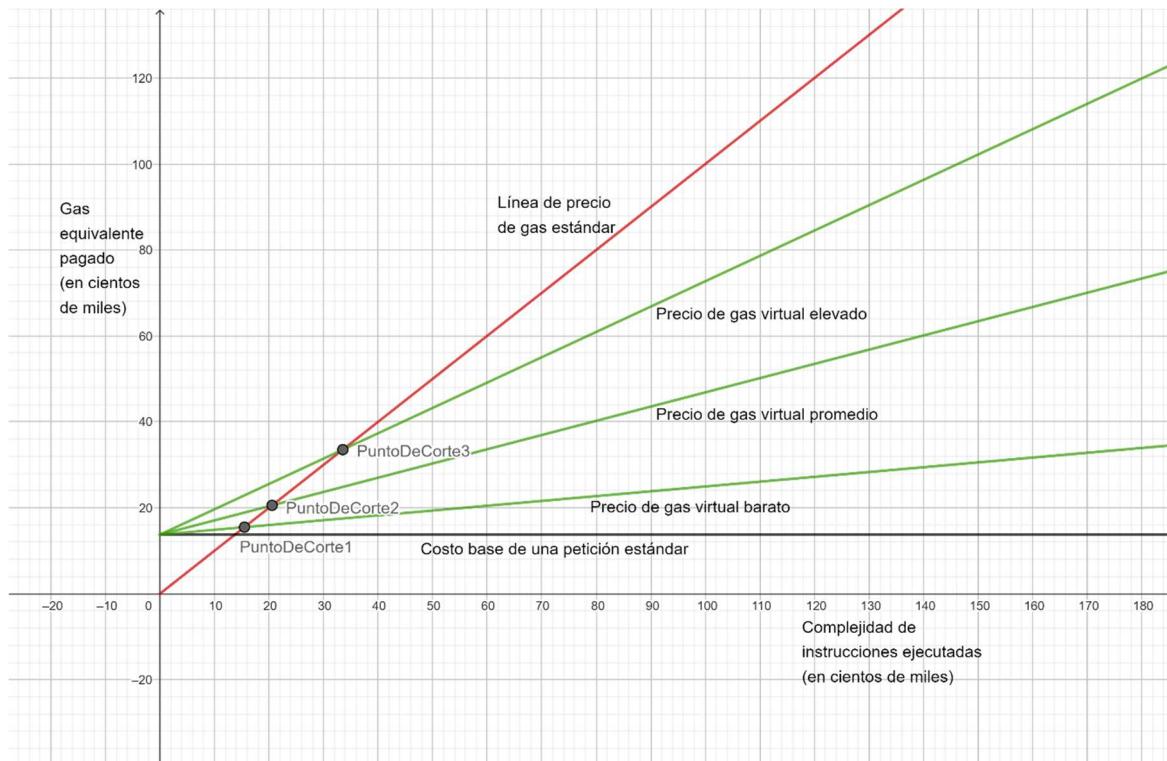
===== short test summary info =====
```

FIGURA IX: impresión de la ejecución del test de la Figura FIGURA VI, con el contrato de la Figura FIGURA VIII.

Por parte del ejecutor, el costo de aceptación permanece inalterado en 74.000, el costo de publicación de resultados, que es el que mayor diferencia tiene al no tener que guardar un gran volumen de datos, se ve reducido a solo 76.000 (3.424.000 unidades menos, es decir, una reducción del 98%). Por otro lado, los costos de confirmación permanecieron iguales, en 63.000 y 69.000 unidades respectivamente, mientras que el costo de recolección de la paga se redujo a menos de la mitad, pasando a ser 136.000 (al no tener que procesar un resultado complejo). Esta última reducción podría indicar que un aspecto negativo del protocolo es la necesidad de codificar y procesar el resultado por segunda vez a la hora de cerrar una petición. Sin embargo, la lógica de procesamiento de resultados depende del cliente, y no del protocolo, por lo que puede ser omitida. En sumatoria, hay un gasto total de 418.000 unidades de gas por parte del ejecutor (Figura FIGURA IX).

En definitiva, sumando los gastos de gas de los llamados a funciones por parte tanto del cliente (260.000 gas) como el ejecutor (418.000 gas), hay un total de 678.000 unidades de gas gastadas en promedio por petición independientemente del código a ejecutarse. En cuanto al costo añadido del código extra necesario para integrar la lógica del cliente al broker, para un contrato con lógica vacía, es de aproximadamente 1.257.000 unidades de gas (Figura **FIGURA IX**). Nótese que el costo de un contrato sin lógica, sumado al costo de despliegue de la lógica pura onchain (Figura **FIGURA IV**), es menor que el costo del contrato cliente de Optimistic Execution por una cantidad de aproximadamente 171.000 unidades. Esto se debe a la lógica de procesamiento de resultados. La misma, no depende de la complejidad de la petición, sino que depende de la cantidad de lógica extra que el cliente desee añadir para procesar los resultados calculados. Sin embargo, el hecho de que el despliegue de un contrato se deba realizar solo una vez, independientemente de la cantidad de peticiones a realizarse sobre ese contrato, significa que a largo plazo, para la ejecución de procesos repetitivos (que requieran muchas peticiones sobre la misma lógica, solamente variando los datos de entrada), el costo de despliegue se vuelva despreciable, haciendo que para el análisis de comparación de costos, en el largo plazo, solo importen las 678.000 unidades de gas totales por llamados a funciones para crear e interactuar con una petición.

Para hacer un análisis comparativo entre la ejecución de código directa onchain, y la utilización del protocolo Optimistic Execution, y para visualizar la relación complejidad/precio, se utilizaron dos ejes cartesianos, como se muestra en la Figura **FIGURA X** a continuación.



*FIGURA X: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del protocolo Optimistic Execution para distintos valores de mercado de gas virtual.*

Uno de esos ejes representa la medición del gas equivalente a los procesos ejecutados, como estimación de la complejidad de los mismos. El otro eje representa la cantidad de gas abonada por el cliente. Lógicamente, el caso onchain está representado por la **recta roja X=Y**, ya que cada instrucción ejecutada onchain, se paga en su totalidad, pero no tiene costos adicionales ya que no hay protocolo de mediación de peticiones (ExecutionBroker). En cuanto a la utilización del protocolo Optimistic Execution, cada petición conlleva gastos de tres tipos: despliegue del contrato cliente, gasto base de la lógica de gestión de peticiones, y la compensación a los ejecutores por las tareas realizadas. Dado que el mismo contrato cliente se puede reutilizar infinitas veces, se ignorará su costo de despliegue para este análisis. Por otro lado, el gasto base no depende de la lógica a ejecutarse, solo del procesamiento de cambios de estado en el ciclo de vida de la petición. Para el gráfico de la Figura **FIGURA X**, se calcularon los costos asociados a una petición estándar con dos confirmaciones y un tamaño de respuesta de 1KB. Los resultados obtenidos se muestran en la Figura **FIGURA XI** a continuación.

```
Deployment gas: 1726978
Request creation gas: 263353
-----
Acceptance gas: 74256
Submission gas: 735287
First confirmation gas: 62982
Second confirmation gas: 69149
Payment gas: 172540
```

*FIGURA XI: Resultado obtenido de correr el test de la Figura FIGURA VI, para una petición con un tamaño de respuesta de 1KB.*

Como se observa, ignorando el despliegue, el costo base está compuesto por 263.000 unidades de creación que, sumado al gas de aceptación, la publicación, las dos confirmaciones, y el reclamo de pago, llegan a un total de 1.376.000 unidades de gas. Entonces, en cuanto al caso estándar, independientemente de la complejidad de la lógica a ejecutarse, hay un costo base de 1.376.000 unidades (**recta negra**, Figura **FIGURA X**).

Finalmente, está el precio de ejecución de los ejecutores. Esto se refiere a la remuneración que un ejecutor recibe por cada instrucción ejecutada, más allá de los costos de gas de las interacciones con el contrato broker (que ya fueron contemplados por la **recta negra**). Este precio, que va a estar definido por la oferta y la demanda de ejecución delegada, representa un “precio de gas virtual” que los ejecutores cobrarán por cada instrucción ejecutada de forma delegada. Es importante notar que la paga de una petición no representa únicamente este “gas virtual”, sino que engloba también los reembolsos por los gastos fijos que incurrirán los ejecutores. La separación de estos dos conceptos se hizo únicamente a modo de análisis.

Sea cual sea la recta que lo defina, para que tenga sentido que el cliente utilice el protocolo, el punto en la recta que corresponda a la complejidad de una petición debe estar ubicado por debajo de la **recta roja** ya que, si no lo está, esto significa que ejecutar dicho proceso directamente onchain, es más barato. De esta regla se sacan dos

conclusiones. En primer lugar, debido a que la paga por parte del cliente tiene un mínimo representado por los gastos generales del protocolo (**recta negra**), para que en algún momento la suma entre gastos generales y gastos de gas virtual sea más barato que la alternativa onchain, la pendiente de la recta (en la Figura **FIGURA X** son las **rectas verdes**, representando diferentes precios posibles), debe ser menor a la pendiente de la **recta roja**. Esto significa que el precio del gas virtual debe ser siempre más barato que el gas estándar de Ethereum, para que la utilización de este protocolo tenga sentido. En segundo lugar, la utilización de Optimistic Execution, solo será rentable en los puntos posteriores al punto de corte entre la **recta roja** y la correspondiente **recta verde**. Estos puntos, representan, para cada precio de “gas virtual”, la complejidad de petición mínima para la cual es rentable la utilización del protocolo. Nótese que entre más barato sea dicho gas virtual, antes se dará dicho punto de corte, es decir, menor será la complejidad necesaria para la rentabilidad. Para un caso promedio, el punto de corte se da alrededor de los **2.000.000** de unidades de gas.

En definitiva, el límite inferior de complejidad, que defina la viabilidad del protocolo, va a estar dado por el punto de corte antes mencionado. El límite superior, por otra parte, son las 30.000.000 unidades de gas (máximo de gas permitido por bloque en ethereum [21]). Dicho límite se debe a que, si la lógica del cliente llegara a ser más compleja que 30.000.000 de unidades, no se podría correr ningún desafío, y el protocolo perdería su mecanismo de garantía de integridad.

### 5.1.6 Pros y Contras

#### *Pros:*

- Este modelo amplía la capacidad de procesamiento de la red al delegar ejecución de código, así como disminuye el costo en gas a pagarse por la ejecución de procesos más complejos.
- La definición de los precios de ejecución se da automáticamente de manera natural por un proceso de oferta y demanda.
- La mecánica de desafíos permite tener un elevado grado de confiabilidad sobre los resultados de cada ejecución, que escala directamente con el número de ejecutores disponibles.

- Si el cliente tiene la necesidad de ejecutar un proceso repetitivo, pero con valores de entrada cambiantes, se puede reutilizar el contrato cliente, reduciéndose enormemente el costo por petición.

**Contras:**

- Entre más grande y compleja sea la resolución de la petición, mayor deberá ser la prima de aceptación con respecto al pago, lo que aumenta el riesgo y disminuye el retorno porcentual de los ejecutores, y por lo tanto podría resultar en una falta de oferta de los mismos.
- Los ejecutores deben tener algún mecanismo de estimación de complejidad sobre una petición, para calcular si les conviene resolverla por la remuneración ofrecida.
- Ethereum tiene un límite de gas por bloque por lo que, en caso de efectuarse un desafío, la cantidad total de instrucciones a ejecutarse estaría limitada por este techo definido por la EVM [21].
- Cada cliente debe desplegar su propio código en la red, y para problemas de complejidad no trivial, ese despliegue inicial puede ser muy caro.
- El tamaño máximo de un contrato en Ethereum es de 24.577 bytes, lo que representa una restricción a la implementación de la lógica de cada cliente [12].
- Los resultados no son infalibles, ya que existe la posibilidad de que un resultado erróneo se solidifique por falta de ejecutores desafiantes.

## 5.2 Segunda Solución: Open Chain Computing

Debido a los límites máximos de gas establecidos en la red Ethereum, sumado a el enorme costo de desplegar contratos grandes, y a la existencia de un límite máximo de tamaño de un contrato, surgió la idea de prescindir totalmente de la necesidad de almacenar la lógica del cliente dentro de la cadena. Esto trae ciertos interrogantes como el hecho de que, si el código no está por dentro de la cadena, como se logra que todos los ejecutores obtengan el código de manera universal y determinista, o cómo se logra garantizar la integridad de los resultados, si no hay ninguna forma de computarlos onchain.

### 5.2.1 Propuesta

La idea central es tener un agente onchain que genere las tareas a ejecutar, y que estas tareas se asignen a otros agentes para que las ejecuten. De esta manera, tanto la

creación (con referencia al código almacenado), asignación, publicación y procesamiento de resultados ocurriría onchain, mientras que el almacenamiento del código de la petición, y su ejecución, se resolvería por fuera.

Respecto a la búsqueda de integridad de resultados, analizando diversos paradigmas de ejecución descentralizada o distribuida, concretamente el caso de BOINC, surgió la idea de buscar garantizar dicha integridad, dentro de cierto grado de confiabilidad, buscando redundancia de ejecución, y contrastando las respuestas de diversos ejecutores.

La diferencia con BOINC, además de tratarse de una solución web3, es que todas las interacciones de asignación y retribución ocurren on-chain, haciendo que la solución sea transparente y descentralizada, en contraposición con BOINC donde el agente maestro es un sistema privado y todas las comunicaciones entre agentes ocurren también por privado.

Otra diferencia es el sistema de incentivos. Como se explicó anteriormente, en BOINC, todos los incentivos son simbólicos, por lo que hay una fuerte dependencia con la voluntad de los “donantes” de poder de cómputo. Aprovechando que Ethereum es una plataforma con una criptomonedra nativa, se puede utilizarla para incentivar a los ejecutores.

### **5.2.2 Funcionamiento y Características**

La idea principal es prescindir de la lógica de desafíos, eliminando cuestiones como la necesidad de una prima de aceptación y la necesidad de que el código se encuentre onchain, lo que resuelve el inconveniente generado por los límites de gas por bloque y de tamaño de un contrato.

Como se hizo alusión anteriormente, el hecho de que el código no esté onchain, implica el desafío de lograr que todos los ejecutores puedan obtener el código de cada petición de una manera universal y determinista (inmutable, igual siempre y para todos). La forma de resolver esto es publicando el código en un sistema de archivos descentralizado como

IPFS. Básicamente se publica el código en la red, y este se propaga a través de los nodos, dando redundancia y persistencia, como si de un torrent se tratara. Luego, en la blockchain solo se guarda una referencia a el código en IPFS y de esta manera cualquier ejecutor puede leerlo con una simple consulta.

El otro desafío, surge de la ausencia de corroboración determinista (onchain), por lo que se requiere de otro mecanismo para garantizar la integridad de los resultados publicados por los ejecutores. Al igual que en el sistema de BOINC, se busca que haya redundancia de ejecución, para poder contrastar los resultados de diversos ejecutores, logrando un mayor grado de confiabilidad. En pocas palabras, el broker le distribuye la misma petición de forma redundante a un número de ejecutores, y cuando estos publican los resultados, se toma como correcto aquel en el cual la mayoría de los ejecutores coincide. Este sistema supone una serie de problemáticas y desafíos a la hora de garantizar el funcionamiento del protocolo aun ante la presencia de agentes corruptos.

El primer problema a resolver nace de la posibilidad de que un agente malicioso controle una mayoría de los agentes asignados a una petición concreta. Si esto sucede, dicho agente podría imponer cualquier resultado y el protocolo lo tomaría como válido por ser mayoritario. Una forma de evitar esto es pasando de un esquema en el cual los ejecutores pueden resolver una petición voluntariamente, a uno en donde los ejecutores son entidades pre-registradas, y que cada petición seleccione ejecutores al azar, basándose en una semilla aleatoria introducida por el cliente, quien tiene todos los incentivos para que el proceso de selección sea realmente azaroso, y por ende prescindir de un sistema de generación de aleatoriedad complejo. Entre mayor sea la cantidad de ejecutores registrados, y entre mayor sea la cantidad de ejecutores asignados por petición, menor será la probabilidad de que un solo agente controle la mayoría de ejecutores de una petición.

El segundo problema tiene que ver con el cálculo de la compensación para los ejecutores. En el modelo anterior los precios se definían por oferta y demanda. Sin embargo, el hecho de que este protocolo asigne a los ejecutores de manera directa,

implica que el pago tiene que ser calculado algorítmicamente, y no elegido por los participantes. La única influencia que pueden tener tanto los ejecutores como los clientes es la oferta y demanda de presencia. Es decir, el hecho de que haya más o menos ejecutores disponibles en un momento dado. Entre más ejecutores haya disponibles tiene sentido que el precio a pagar por unidad de ejecución sea menor, y viceversa. El número de ejecutores disponibles puede bajar si hay muchos clientes realizando peticiones, y por ende reservando a más ejecutores. Dicho número, también puede bajar si el precio de ejecución se encuentra lo suficientemente bajo como para que muchos ejecutores elijan pausar su participación en la red, hasta que el precio vuelva a subir a un nivel satisfactorio. En este caso, esas son las fuerzas de mercado que intervienen en la determinación del precio a pagar/cobrar. Sin embargo, el número exacto debe ser calculado por el protocolo de todas formas.

No existe una fórmula universal para definir un precio justo. Cualquier algoritmo que se escoja, tendrá un carácter heurístico. Los factores a tener en cuenta a la hora de definir el precio son la cantidad de ejecutores disponibles en el momento de crear una petición, el valor del gas en ese momento (porque recordemos que los ejecutores gastan gas al realizar las operaciones de publicación), un precio anterior de referencia para hacer que la evolución sea gradual, y el tamaño y complejidad de las instrucciones a ser ejecutadas en una petición. Es fácil ver como hay tres factores que están relacionados al contexto de la plataforma (cantidad de ejecutores disponibles, precio del gas, y precio de referencia anterior), mientras que el último factor (el tamaño y complejidad de la petición) depende exclusivamente de los requerimientos concretos del cliente en ese momento. Teniendo eso en cuenta, se puede crear el concepto de “poder de ejecución”, que cuantifica el tamaño y complejidad de un set de instrucciones a ser ejecutadas. Este concepto es prácticamente gemelo al concepto de Gas. La diferencia es que no está pensado necesariamente para las mismas instrucciones, ni tampoco posee los mismos valores por instrucción.

Una vez definido el concepto de poder de ejecución, es necesario definir el mecanismo de uso del mismo. El protocolo requiere de un entorno de ejecución con tres

características principales. En primer lugar, debe ser determinista, para garantizar que un procedimiento siempre responda de la misma manera ante la misma entrada de datos. La razón, es para garantizar que todos los ejecutores asignados a una petición en particular, lleguen indefectiblemente al mismo resultado, asumiendo que no se corrompió el proceso en ningún momento. Esto no significa que el entorno no pueda tener un estado interno que intervenga en el cálculo del resultado. De ser necesario un estado interno, simplemente se debe considerar al mismo como otro dato de entrada más.

En segundo lugar, el entorno debe tener la capacidad de llevar una cuenta precisa del volumen de instrucciones ejecutado. Esto es para poder controlar y limitar el procesamiento contra la cantidad de poder de ejecución proveída. La razón es debido a que, al no haber límite en el tamaño del código a ejecutarse, se requiere un mecanismo para prevenir que un ejecutor quede atascado en la ejecución de una petición, y que haya una proporcionalidad entre el volumen de instrucciones ejecutado y la paga que recibe el ejecutor. La idea es que la ejecución se frene inmediatamente cuando la cantidad de instrucciones ejecutadas, ponderada por la complejidad de las mismas, sea igual a la cantidad de poder de ejecución proveída.

En tercer lugar, cuando una ejecución llegue a su fin, ya sea por el agotamiento del poder de ejecución suplido, o por la llegada efectiva a un punto final de la ejecución, se debe presentar el resultado de esa finalización de una manera estándar. No basta simplemente con un mero valor de retorno. Se debe definir una estructura que represente el estado completo del programa en el momento de corte de la ejecución. Esta estructura debe contener información como el estado interno de todas las variables, el stack de llamadas, el punto de salida (es decir la última instrucción ejecutada), además de una bandera que indique si la ejecución terminó naturalmente, o por agotamiento de poder de ejecución. A esta característica se la decidió llamar “serialización del estado del entorno”. Esto es necesario para que, en caso de que el poder de ejecución suplido inicialmente no sea suficiente para completar todo el procedimiento, el cliente pueda realizar una nueva petición con dicha estructura como valor de entrada, para poder retomar la ejecución en el mismo punto. Un corolario es que la estructura de entrada de

toda petición debe estar definida por dicha estructura representante del estado del programa. La implementación de un lenguaje de máquina virtual que posea todas estas características excede el alcance de este trabajo, y por ende el protocolo permite que quede abierto a la elección de los usuarios del mismo. Es por eso que la codificación de la referencia del código, y el estado de entrada en una petición, es de tipo “string”, para no imponer ninguna limitación. Una sugerencia es que haya dos tipos de estructuras, a identificarse por una pequeña cabecera. La primera representa la estructura de serialización del entorno, y la segunda, únicamente el resultado final, para así aprovechar al máximo el espacio. Otra sugerencia es que ambos parámetros sean un JSON donde se contenga la información útil, como la referencia y el estado respectivamente, además de metadatos que identifiquen qué lenguaje, máquina virtual, y versión deben ser usados para cada caso. Se podría extender el protocolo para limitar que solo se permita una sola o una serie de opciones, para evitar problemas de incompatibilidad.

Una vez determinada la utilidad y funcionamiento del poder de ejecución dentro del entorno elegido, solo resta la definición dinámica del precio por unidad del mismo. Y es aquí donde entran en juego los otros tres factores mencionados anteriormente: la cantidad de ejecutores disponible, el precio del gas al momento de crear una petición, y el precio anterior de referencia (que puede iniciar en cualquier valor elegido por el creador del protocolo). Debido a la naturaleza heurística de la definición de este valor, se omitirá la profundización sobre formulas específicas, delegando esa tarea a cualquier persona que quiera implementar el protocolo en una cadena existente, mencionando únicamente que, sea cual sea el valor formado por la combinación de los factores mencionados, debería estar dividido por una constante de proporcionalidad para lograr que efectivamente el valor final sea proporcionalmente menor a el precio del gas en la red contenedora.

Luego de definir el mecanismo de compensación, y volviendo a el análisis sobre la garantía de la integridad de los resultados publicados y de los incentivos de los agentes que participan en el protocolo, un tercer problema es que tiene que haber algún incentivo para que los ejecutores efectivamente realicen el cálculo y no publiquen algo al azar. La

solución a este problema es sencilla. Si el resultado publicado de un ejecutor no coincide con el publicado por la mayoría de ejecutores, no se le entrega ninguna remuneración a dicho ejecutor. Además, el hecho de que los ejecutores estén registrados, facilita el mantenimiento de un sistema de estadísticas y reputación de los mismos. La organización de la entidad “ejecutor” consiste en mantenerlo en tres colecciones separadas. Una donde se encuentren los ejecutores ocupados o bloqueados por una petición (cada ejecutor puede estar asignado únicamente a una petición a la vez). La segunda donde se encuentren los ejecutores inactivos, es decir, que no pueden ser seleccionados para una petición. Y la última, donde se encuentren los ejecutores activos, es decir, todos aquellos que sí puedan ser seleccionados para una petición. Dentro de esta última colección, se encuentran otras sub colecciones que almacenan jerárquicamente a los ejecutores de acuerdo a la relación entre veces que han resuelto una petición correctamente, y veces que han entregado un resultado minoritario, o que no han entregado nada. De acuerdo a la categoría a la que pertenezcan aumentan o disminuyen las chances de ser elegidos para resolver una petición.

Para complementar este sistema, se le pide a cada ejecutor que cuando se registra, coloque una prima como “stake”, de la cual serán deducidos los gastos de gas en caso de haber cometido un error. Cada ejecutor puede pasar del estado activo al estado inactivo de dos maneras. O desactivándolo manualmente, en cuyo caso se le puede reintegrar el “stake” bloqueado inicialmente, o como resultado de una ejecución mala, que puede consistir en la tardanza excesiva en entregar un resultado o en simplemente entregar un resultado minoritario. En este caso, el ejecutor sería “castigado” automáticamente por la red moviéndolo a la colección de ejecutores inactivos, y utilizando parte de su stake para reintegrar los fondos utilizados como gas (junto con el porcentaje de la paga que le hubiese correspondido) al cliente. En cualquier caso, un nodo inactivo debe reactivar la ejecución manualmente, introduciendo los fondos que sean necesarios para llegar al nivel de “stake” requerido por el protocolo.

El cuarto problema yace en que para que la redundancia de ejecución sea efectiva, todos los ejecutores deben realizar los cálculos de manera independiente. Si el sistema

simplemente asigna a ejecutores a una petición, y luego estos publican los resultados obtenidos, un nodo malicioso podría simplemente esperar a que otro nodo publique su resultado, y copiarlo para replicar el mismo. Esto degrada la integridad del sistema, además de permitir que el nodo que realiza la copia cobre una prima por un trabajo que no realizó. La solución a este problema consiste en dividir la publicación de resultados en dos fases.

Primero está la parte de publicación de “hashes”. Esta fase consiste en que los ejecutores generen una estructura de datos compuesta por el resultado codificado (idealmente igual en todos los ejecutores para una petición dada), seguido por la dirección que identifica únicamente a ese ejecutor. Luego se calcula el digest o hash de esa estructura, y se publica dicho hash. De esta manera, ningún ejecutor malicioso podrá copiar el hash de otro ejecutor, ya que la dirección usada para obtener ese hash solo puede corresponder a un ejecutor.

Una vez que el último ejecutor publique su hash específico, comienza la segunda fase de la publicación. En esta fase, los ejecutores deberán publicar el contenido del resultado codificado, y al mismo se le agregará la dirección del ejecutor onchain y se calculará el hash. Si el hash obtenido onchain coincide con el hash publicado en la fase uno, se toma como válido el resultado y se “libera”. Caso contrario, se anula el resultado de ese ejecutor. Una vez que el último ejecutor liberó su resultado, se realiza la comparación de todos los resultados válidos publicados, se distribuye la paga entre los nodos con un resultado mayoritario (idealmente todos si ninguno comete un error), se castigan los nodos con un resultado minoritario, y se le reintegran los fondos necesarios a el cliente, en caso de que haya habido uno o más ejecutores erróneos (que no cobran). ¿Significa esto que el último ejecutor en publicar los resultados deberá pagar más gas que el resto debido al adicional de operaciones de cierre? Tanto para la primera como para la segunda fase la respuesta es afirmativa, lo que genera un leve incentivo para entregar los resultados antes que el resto de nodos. En cualquier caso, estos gastos adicionales deben ser tenidos en cuenta en el algoritmo de definición de precio por unidad de ejecución.

Una quinta cuestión a tener en cuenta es la definición de los plazos en los cuales los ejecutores deben entregar los resultados, así como también un mecanismo para el cliente para poder forzar la continuación de una ejecución estancada eternamente por la presencia de un ejecutor inactivo. Debido a esta posibilidad, las peticiones se pueden estancar en cualquiera de las dos fases de publicación.

En la primera fase, se define un tiempo máximo que tienen los ejecutores para entregar los resultados. Una vez pasado dicho límite, el cliente tiene la opción de realizar dos cosas. En primer lugar, el cliente puede pedir la rotación de ejecutores, en cuyo caso todos los ejecutores cuyo plazo de entrega se haya vencido, serán castigados como se explicó con anterioridad, y serán reemplazados por nuevos ejecutores. Esta función, puede ser llamada por el cliente cuantas veces quiera, siempre que se cumpla que ninguno de los ejecutores al que aún le falta entregar el hash de sus resultados, esté dentro del plazo permitido. Si hay una menor cantidad de ejecutores disponibles que de ejecutores morosos, se reemplazarán tantos ejecutores como sea posible, pero el cliente deberá esperar a que se liberen más ejecutores, o a que el resto de ejecutores morosos entreguen sus hashes. Una segunda opción del cliente, es truncar a los ejecutores. Esto consiste en pasar forzosamente a la segunda fase removiendo por completo, sin reemplazar, a los ejecutores morosos. Para que este pueda ser el caso, al menos un ejecutor debió haber entregado su hash. Al igual que con la rotación de ejecutores, los morosos serán castigados, y parte de su stake será utilizado para reintegrarle el gas de ejecución al cliente.

En cuanto a la posibilidad de estancamiento en la segunda fase de publicación, el cliente puede forzar la liberación de resultados. En este caso, se debe cumplir que haya al menos un resultado válido publicado, y que haya pasado el periodo de gracia (una constante definida) de liberación de resultados. Las consecuencias prácticas de este método son iguales a truncar los ejecutores morosos, solo que en la segunda fase. El cliente elige acelerar el cierre de su petición a cambio de perder redundancia de resultados y, por ende, reduciendo la confiabilidad de los mismos.

Existe un caso límite, en el cual absolutamente todos los ejecutores tengan una discrepancia entre el hash publicado en la fase uno, y el resultado publicado en la fase dos. Cuando el último ejecutor discrepante libere, erróneamente, su resultado, el protocolo no podrá elegir ninguno como válido, ya que ninguno de los ejecutores siguió el protocolo correctamente. En este caso, el contrato “recicla” la petición, castigando a todos los ejecutores, e intentando volver a crear una petición igual a nombre del cliente. En caso de que no haya la cantidad suficiente de ejecutores activos para ser asignados a la nueva petición, simplemente se le reintegra el dinero al cliente.

### 5.2.3 Estructura de los Contratos

Esta solución tiene un total de 1 contrato desplegable, el del “ExecutionBroker”. Al igual que en el caso anterior, este contrato está encargado de almacenar y administrar las peticiones, así como las transacciones necesarias para la creación y resolución de las mismas. Además, este contrato almacena una colección con todos los ejecutores registrados, con su estado actual y sus estadísticas de resolución (lo que los ubica en una escala de categorías jerárquicas).

La estructura del contrato **ExecutionBroker** es la siguiente:

#### **Estructuras definidas:**

- Executor: representa a un ejecutor registrado en la plataforma. Contiene la dirección de dicho ejecutor, la cantidad de stake, estadísticas de resolución, e información sobre asignaciones actuales en caso de haberlas.
- ExecutorsCollection: contiene las colecciones correspondientes a cada uno de los estados posibles de un ejecutor, así como las sub-colecciones jerárquicas de los ejecutores activos.
- Request: representa una petición. Contiene información como el ID de la petición, la dirección del cliente que la creó, el precio pagado por unidad de poder de ejecución en el momento de su creación, la cantidad de poder de ejecución pagado por el cliente, el estado inicial del procesamiento, la referencia a la ubicación del código a ejecutar, el resultado de la petición (comienza estando vacío), y banderas acerca del estado actual de la misma.
- TaskAssignment: representa el vínculo entre un ejecutor y una petición. Contiene la dirección del ejecutor, la marca de tiempo del momento de la asignación,

banderas que representan el estado de la asignación, y los hashes de los datos de resultado con y sin la dirección del ejecutor incluida, así como el resultado en sí. Estos últimos tres valores comienzan en cero.

- Result: representa el resultado de una petición. Contiene el resultado puro codificado, y la dirección del ejecutor que la publicó. En el caso de que sea el resultado final elegido por el algoritmo, el valor de la dirección será la del contrato broker.

#### ***Variables internas del contrato:***

- uint EXECUTION\_TIME\_FRAME\_SECONDS: representa el tiempo máximo que un ejecutor se puede tomar para completar una fase.
- uint BASE\_STAKE\_AMOUNT: representa la cantidad mínima de fondos que un ejecutor debe tener bloqueados en el contrato (stake), para poder estar activo y ser asignado a una petición. Esta cantidad debe ser mayor al costo en gas de las funciones de rotación o truncación, contemplando la cantidad máxima de ejecutores, y considerando un precio de gas particularmente alto.
- uint MAXIMUM\_EXECUTION\_POWER: representa la cantidad máxima de poder de ejecución que un cliente puede pagar para una petición en concreto.
- uint MAXIMUM\_EXECUTORS\_PER\_REQUEST: representa la cantidad máxima de ejecutores redundantes que pueden ser asignados a una misma petición.
- Request[] requests: colección de peticiones.
- mapping(uint => TaskAssignment[]) taskAssignmentsMap: un arreglo asociativo que vincula el ID de una petición con una colección de asignaciones, que son las estructuras vinculantes entre ejecutores y peticiones.
- ExecutorsCollection executorsCollection: representa el contenedor de todas las sub-colecciones de ejecutores en sus diversos estados.
- uint referenceExecutionPowerPriceCache: variable donde se cachea el último precio de referencia cobrado por unidad de poder de ejecución. Se utiliza para calcular el nuevo precio.

#### ***Eventos:***

- requestCreated: se dispara cuando se crea una petición. Contiene el ID de la petición y la dirección del cliente que la creó.
- resultSubmitted: se dispara cuando un ejecutor publica el hash firmado de un resultado. Contiene el ID de la petición, y la dirección del ejecutor publicante.
- resultLiberated: se dispara cuando un ejecutor libera los resultados calculados en la segunda fase de publicación. Contiene el ID de la petición, y la dirección del ejecutor.

- requestSubmissionsLocked: se dispara cuando el último ejecutor publica el hash de su resultado calculado firmado. Contiene el ID de la petición.
- requestClosed: se dispara cuando se cierra una petición, es decir, cuando se obtienen todos los resultados liberados posibles y se elige el mayoritario como absoluto. Contiene el ID de la petición, y la cantidad de coincidencias que tuvo el resultado mayoritario elegido.
- requestRecycled: se dispara en el caso borde en el que ninguno de los resultados liberados se condiga con el hash previamente publicado por el ejecutor correspondiente, y por ende no haya resultados válidos, forzando a que el protocolo reabra la petición desde cero. Contiene el ID de la petición.
- executorLocked: se dispara cuando un ejecutor activo es asignado y bloqueado por una petición. Contiene la dirección del ejecutor y el ID de la petición a la cual fue asignado.
- executorUnlocked: se dispara cuando un ejecutor vuelve a ser elegible para ser asignado a una petición. Puede ser en el caso de que haya estado inactivo y haya sido reactivado por el agente dueño, o puede también darse cuando se cierre satisfactoriamente una petición, retornando los ejecutores que no cometieron ninguna falla a la colección de los activos. Contiene la dirección del ejecutor.
- executorPunished: se dispara cuando un ejecutor es castigado, ya sea por inactividad, o por publicación de datos incorrectos. Contiene la dirección del ejecutor.

### **Funciones públicas:**

- registerExecutor [payable]: crea un nuevo ejecutor, vinculándolo con la dirección que llamó a la función, y lo añade a la colección de ejecutores activos, en la categoría neutral, ya que no posee estadísticas de ejecución al ser nuevo. Solo puede ser llamada por una dirección que no haya sido registrada antes, y requiere que el valor enviado en la transacción sea mayor o igual al stake mínimo definido por el protocolo para ejecutores activos.
- pauseExecutor: mueve un ejecutor activo a la colección de ejecutores inactivos. Recibe una bandera indicando si el ejecutor desea que se le reintegre su stake. Solo puede ser llamada por ejecutores que se encuentren activos. Retorna una bandera indicando si el reintegro del stake fue exitoso.
- activateExecutor [payable]: mueve a un ejecutor desde la colección de inactivos hacia la de activos, y lo coloca en la categoría correspondiente de acuerdo a sus estadísticas de ejecución. Requiere que la suma del stake actual y el valor enviado en la transacción sea mayor o igual al stake mínimo definido por el protocolo para ejecutores activos. Solo puede ser llamado por ejecutores inactivos.
- submitRequest [payable]: crea una petición y calcula la cantidad de poder de ejecución disponible en base a los fondos enviados en la transacción, la cantidad

de ejecutores pedidos, y el precio por unidad calculado en el contexto local. Luego le asigna de manera aleatoria (utilizando el seed proveído por el cliente) la cantidad pedida de ejecutores (basándose en la categoría mínima pretendida), y bloquea a dichos ejecutores para que no puedan volver a ser asignados hasta terminar con esta petición. Recibe el estado inicial codificado, la referencia a la ubicación del código, la cantidad de ejecutores pretendida, la categoría pretendida de dichos ejecutores, y una semilla de aleatoriedad. Emite el evento “requestCreated”. Requiere que la cantidad de ejecutores elegidos sea impar y que no supere la cantidad máxima permitida, ni la cantidad máxima disponible para la categoría pretendida por el cliente. También requiere que no se supere la cantidad máxima de poder de ejecución contratado. Retorna el ID de la petición creada.

- `rotateExecutors`: reemplaza a todos los ejecutores morosos en una petición, castigándolos, y utilizando parte del stake de los mismos para reintegrarle el gas al cliente. Recibe el ID de la petición, una semilla de aleatoriedad para la selección de nuevos ejecutores, y la categoría pretendida para los mismos. Si la cantidad de ejecutores morosos a rotar es mayor a la cantidad de ejecutores disponibles, solo rota los que puede. Solo puede ser llamada por el cliente que creó la petición, y sólo si existen ejecutores que aún no publicaron su hash. Retorna una bandera indicando si el reintegro del gas al cliente fue exitoso.
- `truncateExecutors`: se comporta de la misma manera que la función “`rotateExecutors`”, con la diferencia de que solo recibe el ID de la petición, y simplemente elimina los ejecutores morosos en lugar de reemplazarlos. Requiere que haya al menos un ejecutor que haya publicado su hash, y que no haya ejecutores que se encuentren aun dentro de su periodo de gracia para publicar su hash. Retorna una bandera indicando si el reintegro del gas al cliente fue exitoso.
- `submitSignedResultHash`: publica el hash de los resultados firmados por un ejecutor, actualiza el estado de la asignación del mismo con la petición en cuestión y emite el evento “`resultSubmitted`”. Si se trata del último ejecutor en publicar su hash, además se marcan las publicaciones como bloqueadas, pasando la petición a la segunda fase de publicación de resultados, emitiendo el evento “`requestSubmissionsLocked`” en el proceso. Recibe el ID de la petición, y el hash de los resultados firmados. Solo puede ser llamada por un ejecutor asignado a la petición en cuestión, siempre y cuando este no haya publicado ningún hash aún.
- `liberateResult`: publica el resultado codificado, y se calcula el hash firmado por el ejecutor llamante equivalente a ese resultado, para luego comparar dicho hash con el publicado por el ejecutor en la primera fase. Si el hash coincide, se actualiza el estado de la asignación ejecutor & petición, cargando el resultado codificado y el hash sin firmar (para luego poder comparar si es igual a los hashes sin firmar del resto de ejecutores). Si los hashes no coinciden, no se actualiza el estado de

la asignación, lo que implica que, al cerrarse la petición, se detecte como incompleto resultando en un castigo para el ejecutor. En cualquier caso, marca la asignación como liberada y en caso de tratarse del último ejecutor en liberar su resultado, se hace un chequeo para determinar si al menos uno de los ejecutores publicó un resultado que se condiga con el hash publicado en la primera fase. Si este es el caso, se llama a la función “`_closeRequest`” para darle cierre a la petición. Caso contrario, se recicla la petición, es decir, se intenta crear una nueva petición desde cero, manteniendo los parámetros de la original, emitiendo el evento “`requestRecycled`”. En caso de no haber más ejecutores disponibles, se le reintegran los fondos al cliente. Solo puede ser llamada por un ejecutor asignado a la petición en cuestión, siempre y cuando esta se encuentre abierta y en la segunda fase de publicación, y solo si el ejecutor no ha liberado su resultado aún. Retorna una bandera indicando si el hash calculado a partir del resultado coincide con el publicado en la fase uno.

- `forceResultLiberation`: elimina de una petición a todos los ejecutores que se hayan demorado en liberar los resultados en la segunda fase de publicación de los mismos, y luego llama a la función “`_closeRequest`”. De esta manera, el cliente logra forzar el cierre acelerado de una petición, reduciendo la confiabilidad de los resultados obtenidos al tener menos redundancia. Recibe el ID de la petición. Sólo puede ser llamada por el cliente de una petición abierta que se encuentre en la segunda fase de publicación, y sólo si al menos uno de los ejecutores publicó sus resultados de manera correcta.

#### ***Funciones privadas:***

- `_closeRequest`: elige un resultado basándose en aquel que tenga la mayor cantidad de coincidencias entre los redundantes publicados. Si hay un empate, se queda con el primero. Luego, agrupa a los ejecutores en una colección de ganadores, si su resultado publicado coincidió con el mayoritario, o perdedores, en caso contrario para que sean castigados. Luego se le paga a cada uno de los ejecutores ganadores, mientras que la paga que le hubiera correspondido a los perdedores, junto con parte de su stake para compensar el gas gastado, se le reintegra al cliente. Emite el evento “`requestClosed`”. Recibe el ID de la petición, y la cantidad inicial de gas en la transacción, para poder posteriormente calcular cuánto stake quitarle a cada ejecutor castigado, en caso de haberlos, y reintegrarle el gas al cliente.

#### **5.2.4 Ciclo de vida de una petición**

Como se muestra en la Figura **FIGURA XII** a continuación, para que la solución funcione, un número de ejecutores deben registrarse de manera directa dentro del

contrato, para poder ser asignados a peticiones de forma dinámica. El ciclo de vida de una petición comienza cuando un cliente la crea (paso 1, flecha blanca), con todos los datos y requerimientos necesarios. Esto, asigna al azar a el número de ejecutores requeridos, emitiendo un evento que les deja saber que tienen una petición pendiente.

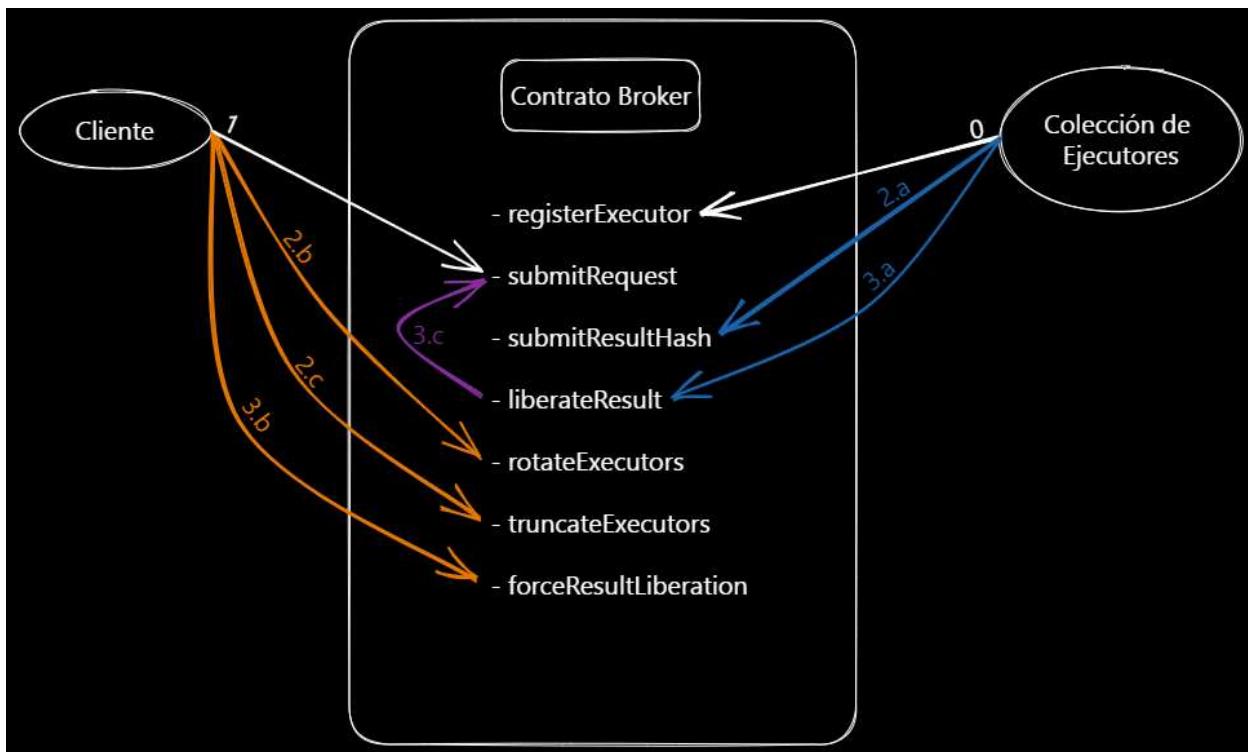


FIGURA XII: Ciclo de vida de una petición en Open Chain Computing.

Posteriormente, los ejecutores comienzan a realizar el procesamiento requerido por la petición, y a medida que van obteniendo el resultado, realizan un hash firmado de los mismos, y lo publican (paso 2.a, flecha azul). Una vez que el último ejecutor publica su hash firmado, la petición pasa a su segunda fase de publicación, lo que habilita que los ejecutores comiencen a liberar los resultados (es decir, a publicar la estructura a partir de la cual obtuvieron el hash firmado que publicaron en la primera fase).

Mientras aún haya ejecutores asignados que se estén demorando en publicar el hash firmado, el cliente puede llamar a la función de rotación de ejecutores (paso 2.b, flecha naranja), cambiando y castigando a todos los ejecutores que se hayan excedido del tiempo máximo de cómputo. Este paso lo puede realizar cuantas veces quiera. De igual

manera, si el cliente considera que el numero de ejecutores que ya publico su hash firmado es suficiente, puede optar por truncarlos (paso 2.c, flecha naranja), es decir, castigar a los ejecutores morosos, pero sin remplazarlos, sacrificando confianza en la integridad del resultado, a cambio de velocidad de resolución. Este paso, lo puede realizar solo una vez, ya que fuerza a la petición a pasar a la segunda fase de publicación de resultados (la liberación de los mismos).

Una vez en la segunda fase de la petición, los ejecutores proceden a publicar sus resultados mediante el llamado a la función de liberación de los mismos (paso 3.a, flecha azul). El contrato realiza la comprobación contra el hash firmado publicado por cada ejecutor en la primera fase. Si la firma no coincide, el ejecutor es castigado. Caso contrario, se guarda el resultado en la petición. Una vez que el último ejecutor libera su resultado, se comparan todos los resultados válidos, y se toma como oficial aquel que se repita la mayor cantidad de veces. Si hay un empate, se toma el primero en haber sido publicado. Por último, si resulta que todos los ejecutores liberaron un resultado que no coincide con su respectivo hash firmado, se recicla la petición, castigando a todos los ejecutores, y reasignando nuevos. En caso de no haber ejecutores disponibles, se le reintegran los fondos al cliente y se emite un evento especial indicando lo sucedido.

De manera similar a la función de truncado de ejecutores en la primera fase, en la segunda el cliente puede optar por forzar la liberación de resultados (paso 3.b, flecha naranja), quedándose solamente con los resultados ya publicados y castigando a aquellos ejecutores que no hayan liberado su resultado aún. De esta manera, el cliente puede acelerar el cierre de una petición, sacrificando confiabilidad en el resultado final. Para que el cliente pueda llamar a esta función, debe haber al menos un resultado válido liberado, y tiene que haber transcurrido el tiempo mínimo de liberación de resultados establecido por el protocolo.

### 5.2.5 Ejemplo de uso

Debido a su complejidad, la implementación de un entorno de ejecución determinista con monitoreo de volumen de instrucciones procesadas y con estado global serializable excede el alcance de este trabajo, y deberá ser desarrollado en otro proyecto de investigación en el futuro. Por este motivo, no se implementó ningún ejemplo en concreto para este protocolo. Sin embargo, esto no impide realizar un análisis numérico ya que, al poseer el código del protocolo (también en GitHub [29]), se pueden calcular los gastos generales producto de la distribución de peticiones. Es importante notar que, en este caso, al no depender del entorno de ejecución de solidity, el volumen y complejidad de la petición solo se vería limitado por las características de implementación de dicho entorno, el cual puede ser elegido por el cliente. Por ende, teniendo en cuenta el mejor de los casos, el volumen y complejidad de las peticiones podrían ser perfectamente ilimitados. El único límite establecido por el protocolo, como consecuencia de estar montado sobre una EVM, es en cuanto a el tamaño del bloque de datos de serialización del estado del entorno de ejecución, que es lo único que ingresa a la cadena de bloques en cada petición/publicación. Cabe aclarar que, en este caso, también se utilizó Brownie, Python y Ganache para realizar las mediciones pertinentes.

Para calcular los costos generales de una petición utilizando el protocolo Open Chain Computing, se creó una petición simple de resolución trivial, es decir, carente de instrucciones y de entorno de ejecución, con 3 ejecutores y un tamaño de respuesta de 1024 bytes. En este protocolo el cliente no debe desplegar ningún contrato por lo que no hay ningún gasto previo a la creación de la petición. En primer lugar, los ejecutores se deben registrar en el broker, mediante un proceso que cuesta en promedio 125.000 de gas. Este proceso de registración se debe realizar solamente una vez, por lo que no será tenido en cuenta para el cálculo de costos generales. Adicionalmente, al registrarse, los ejecutores deben depositar una cierta cantidad de ether (a definir por la implementación del protocolo), que servirá como incentivo para que los mismos actúen correctamente. Cuando un ejecutor lo deseé, este puede retirarse de la plataforma, recuperando la suma depositada. La creación de una petición genérica estándar con tres ejecutores tiene un costo promedio de 760.000 (Figura **FIGURA XIV**). Una vez creada y distribuida la

petición, los 3 ejecutores (sin ejecutar nada debido a que se trata de un test de ejecución trivial) simplemente firman el hash de un resultado trivial (1024 bytes aleatorios), y publican dicha firma. Los dos primeros ejecutores incurren un costo de aproximadamente 90.000 de gas al hacerlo, mientras que el ultimo, al tener que ejecutar el proceso de cambio de fase de la petición, tiene un costo mayor en alrededor de 112.000 unidades (Figura **FIGURA XIV**). Es importante aclarar que estos números, si bien no dependen fuertemente de la complejidad de la petición, si dependen de la cantidad de ejecutores, en especial los procesos ejecutados por el ultimo en entregar/liberar sus resultados, ya que debe actualizar el estado de la petición, lo que implica iterar sobre todos los ejecutores de la misma. Luego del cambio de fase, se procede a liberar los resultados. Los dos primeros ejecutores incurren un costo aproximado de 782.000 de gas al hacerlo, mientras que el ultimo, al tener que ejecutar el proceso de cambio de fase de la petición, tiene un costo mayor (por los mismos motivos que en el cierre de la primera fase) en 1.870.000 unidades (como se observa en las Figuras **FIGURA XIII** y **FIGURA XIV** a continuación).

```

def test_liberate_last_result_all_correct(self):
    printInfo = True
    resultSize = 1024
    amountOfExecutors = 3
    executionStateSampleValue = "9" * resultSize
    broker = BrokerFactory.create(account=Accounts.getFromIndex(0))
    registrationCosts = []
    for i in range(amountOfExecutors):
        registrationCosts.append(Accounts.getFromIndex(i + 2).balance())
    executors = []
    for i in range(amountOfExecutors):
        executors.append(Executor(broker, Accounts.getFromIndex(i + 2), True, gas_price=1))
    for i in range(amountOfExecutors):
        registrationCosts[i] = registrationCosts[i] - Accounts.getFromIndex(i + 2).balance()
    requestor = Requestor(broker, Accounts.getFromIndex(1))
    request = requestor.createRequest(executionStateSampleValue, "code reference", amountOfExecutors=i+1, executionPower=1000)
    requestCreationCost = request.gas_used
    reqID = request.return_value
    results = []
    submissionTXs = []
    firstSubmissionsAvgCost = 0
    for i in range(amountOfExecutors - 1):
        results.append(executors[i].calculateFinalState(reqID, state_value=executionStateSampleValue)) # Hardcoded trivial result
        submissionTXs.append(executors[i].submitSignedHash(reqID, results[i]))
        firstSubmissionsAvgCost += submissionTXs[-1].gas_used / (amountOfExecutors - 1)
    results.append(executors[-1].calculateFinalState(reqID, state_value=executionStateSampleValue)) # Hardcoded trivial result
    submissionTXs.append(executors[-1].submitSignedHash(reqID, results[-1]))
    assert dict(broker.requests(reqID))["submissionsLocked"] == True
    liberationTXs = []
    firstLiberationsAvgCost = 0
    for i in range(amountOfExecutors - 1):
        liberationTXs.append(executors[i].liberateResult(reqID))
        firstLiberationsAvgCost += liberationTXs[-1].gas_used / (amountOfExecutors - 1)
    assert dict(broker.requests(reqID))["closed"] == False
    assert dict(broker.requests(reqID))["result"] == ('', '0x000000000000000000000000000000000000000000000000000000000000000')
    liberationTXs.append(executors[-1].liberateResult(reqID))
    results[0].signingAddress = broker.address
    assert dict(broker.requests(reqID))["closed"] == True
    assert dict(broker.requests(reqID))["result"] == results[0].toTuple()
    if printInfo:
        for i in range(amountOfExecutors):
            print(f"Executor {i + 1} registration cost: {registrationCosts[i]}")
        print("-----")
        print(f"Request creation cost: {requestCreationCost}")
        print("-----")
        for i in range(amountOfExecutors):
            print(f"Hash submission cost {i + 1}: {submissionTXs[i].gas_used}")
        print("-----")
        for i in range(amountOfExecutors):
            print(f"Result liberation cost {i + 1}: {liberationTXs[i].gas_used}")

```

FIGURA XIII: código en python del test de costos de una petición trivial en OCC.

```

----- Captured stdout call -----
Executor 1 registration cost: 142241
Executor 2 registration cost: 115064
Executor 3 registration cost: 117887
-----
Request creation cost: 763306
-----
Hash submission cost 1: 89966
Hash submission cost 2: 89946
Hash submission cost 3: 112755
-----
Result liberation cost 1: 782571
Result liberation cost 2: 782583
Result liberation cost 3: 1870552
===== short test summary info =====

```

FIGURA XIV: impresión de la ejecución del test de la Figura FIGURA XIII.

En este caso, el gasto directo del cliente es de aproximadamente 760.000 unidades de gas. Por otro lado, los costos de los tres ejecutores suman aproximadamente 3.727.000

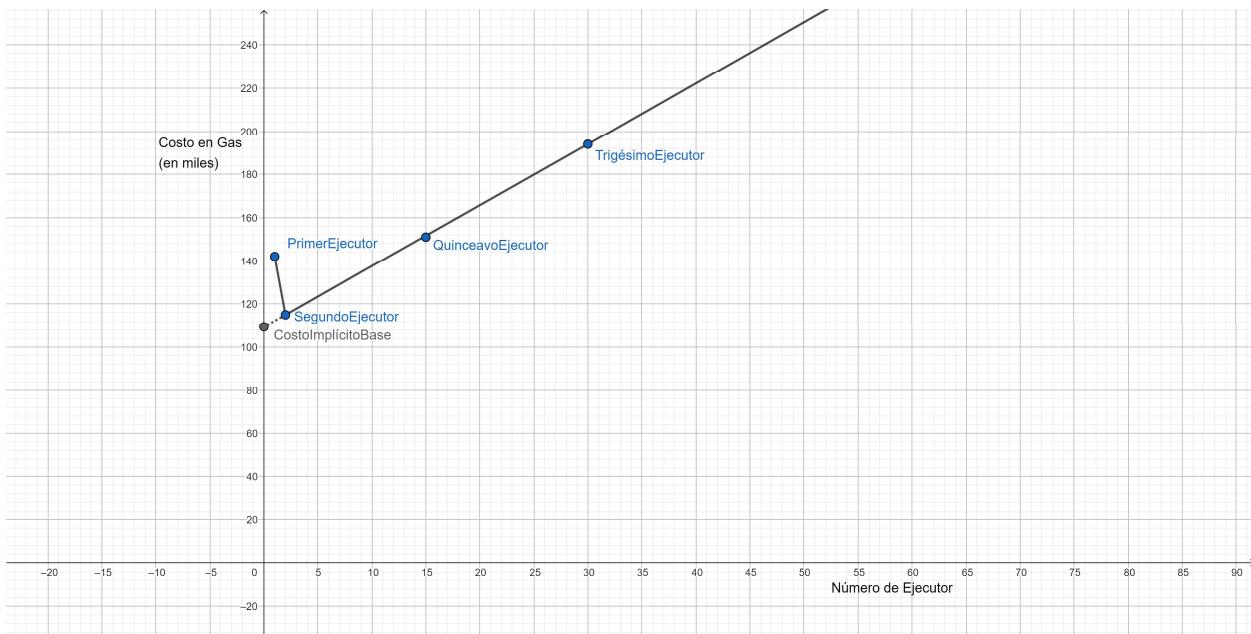
unidades de gas. Estos costos, al igual que los de petición, aumentan considerablemente con la complejización de la estructura de los datos de respuesta. Otra cosa importante a tener en cuenta es que los costos base se multiplican por la cantidad de ejecutores contratados y, además, los precios a pagar por la petición, y por las funciones ejecutadas por el último ejecutor, también aumentan al incrementar la cantidad de ejecutores, ya que estas funciones iteran sobre la colección de asignados.

Debido a que este protocolo introduce variables que afectan el costo de las distintas interacciones con el contrato broker, se realizaron una serie de mediciones para estimar el impacto de cada variable (cantidad de ejecutores y tamaño del estado respuesta) en el costo de las interacciones. En primer lugar, y de manera previa a la creación de cualquier petición, los ejecutores candidatos se deben registrar. Este proceso implica recorrer una colección de ejecutores ya registrados e insertar al nuevo candidato en el primer lugar libre. Lógicamente, entre más extensa sea la colección de ejecutores ya presentes, la complejidad de introducción de uno nuevo será ligeramente mayor. Como muestran la Tabla **TABLA I** y la Figura **FIGURA XV** a continuación, el costo de registro de cada ejecutor esta dado por un costo base de procesamiento, más un incremento lineal dado por la cantidad de ejecutores ya presentes. Cabe aclarar que el primer ejecutor en registrarse tendrá un costo base mayor debido a los procesos de iniciación de la colección.

Número de Ejecutor	Costo en Gas
1	142.241
2	115.064
3	117.887
4	120.710
5	123.533
6	126.356
7	129.179
8	132.002
9	134.825

10	137.648
11	140.471
12	143.294
13	146.117
14	148.940
15	151.763
16	154.586
17	157.409
18	160.232
19	163.055
20	165.878
21	168.701
22	171.524
23	174.347
24	177.170
25	179.993
26	182.816
27	185.639
28	188.462
29	191.285
30	194.108

TABLA I: Costo de registro en OCC para cada ejecutor (en orden de registro).

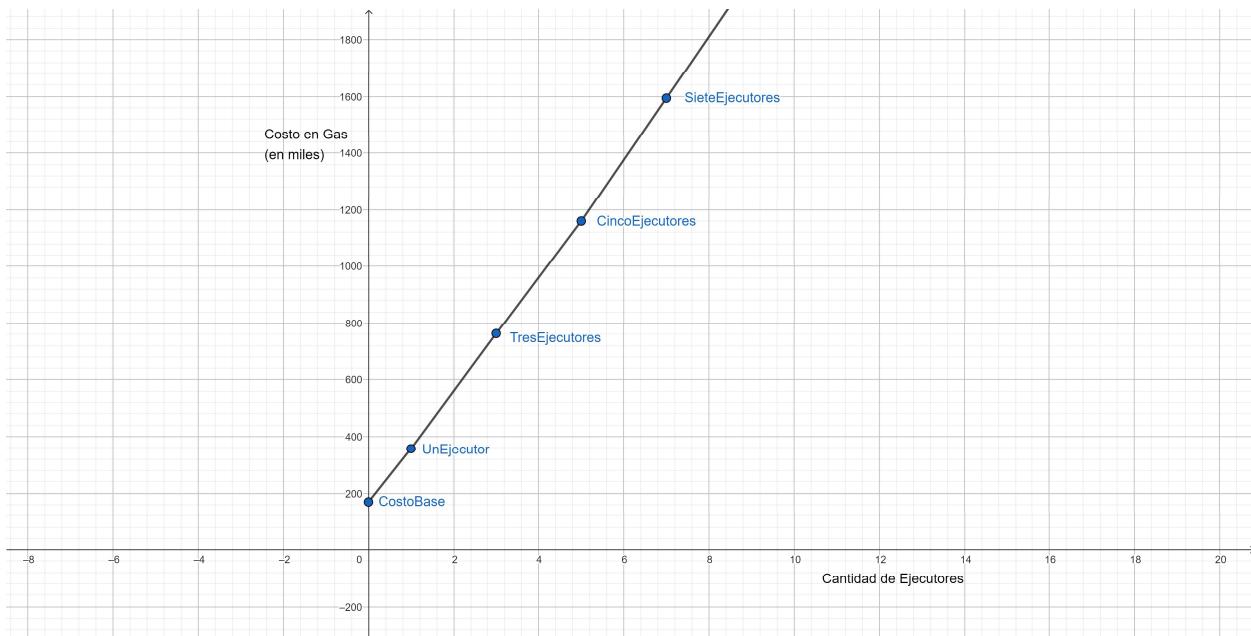


*FIGURA XV: Gráfico del costo de registro de un ejecutor por la cantidad de ejecutores ya presentes.*

En cuanto a los costos asociados a las peticiones, para estimar el efecto de la cantidad de ejecutores y el tamaño de la respuesta en el mismo, se realizó una serie de mediciones para cada combinación de casos de 1, 3, 5 y 7 ejecutores, con respuestas de 256, 512, 1024 y 2048 bytes. Para la creación de una petición, la publicación inicial de un resultado, y su publicación final, el tamaño de la respuesta no afecta en lo absoluto al costo. Y esto es lógico, ya que en estos pasos no se presenta ningún resultado de manera directa y, por ende, no es necesario su procesamiento. En cuanto a la creación de una petición (como muestran la Tabla **TABLA II** y la Figura **FIGURA XVI** a continuación), el costo aumenta de manera lineal, con una pendiente muy elevada, respecto de la cantidad de ejecutores contratados. Esto se debe a que, por cada ejecutor asignado, hace falta procesar toda la lógica de asignación aleatoria.

	1 ejecutor	3 ejecutores	5 ejecutores	7 ejecutores
256 bytes	357.027	763.184	1.160.583	1.594.062
512 bytes	357.027	763.184	1.156.757	1.593.330
1024 bytes	357.039	763.306	1.162.813	1.593.830
2048 bytes	357.039	765.048	1.162.569	1.590.102

*TABLA II: Costo promedio de creación de una petición por cantidad de ejecutores y por tamaño de respuesta.*



*FIGURA XVI: Gráfico del costo de creación de una petición por la cantidad de ejecutores contratados.*

Para los casos de publicación de hashes parcial y final (como se ve en las Tablas **TABLA III** y **TABLA IV**, y Figuras FIGURA XVII y FIGURA XVIII respectivamente a continuación), el costo también aumenta de manera lineal respecto a la cantidad de ejecutores asignados, pero con una pendiente más leve, ya que solo se itera levemente por el estado de entrega de todos los ejecutores asignados para saber si se requiere pasar a la segunda fase de entrega de resultados.

	3 ejecutores	5 ejecutores	7 ejecutores
256 bytes	89.956	95.569	101.178
512 bytes	89.962	95.569	101.180
1024 bytes	89.956	95.569	101.180
2048 bytes	89.962	95.572	101.182

*TABLA III: Costo promedio de publicación parcial (solo aplicable cuando hay más de un ejecutor) por cantidad de ejecutores y por tamaño de respuesta.*

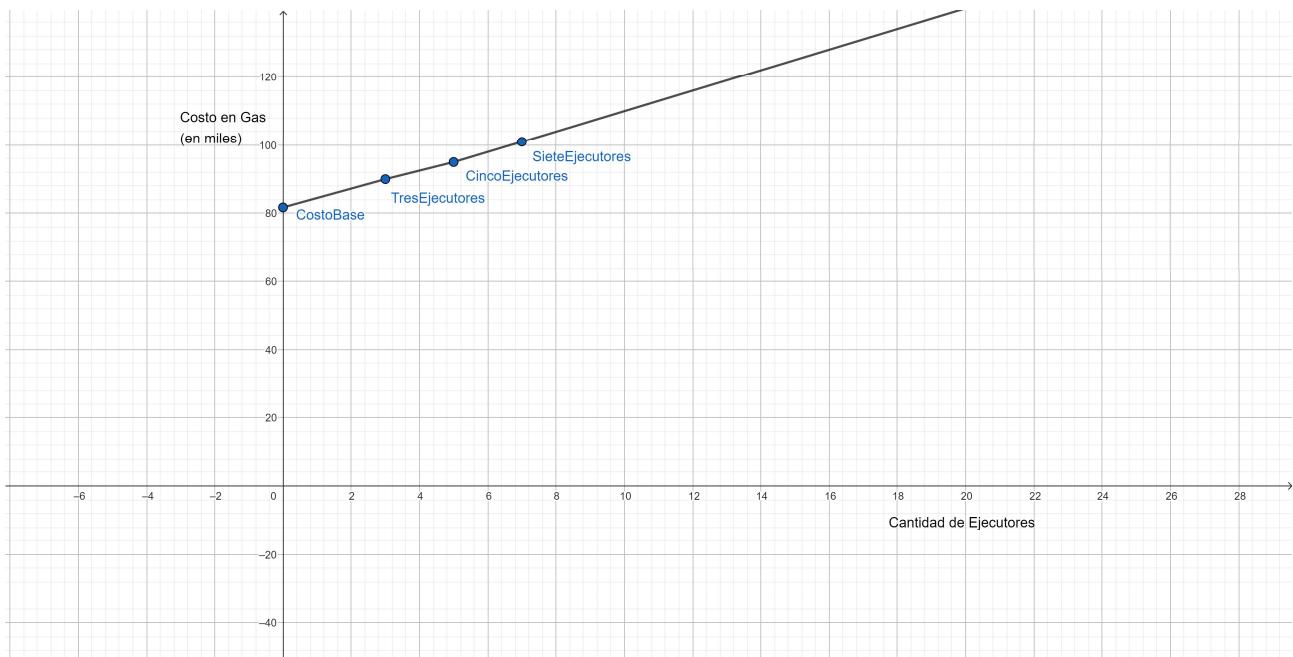


FIGURA XVII: Gráfico del costo de publicación inicial de resultados por la cantidad de ejecutores contratados.

	1 ejecutor	3 ejecutores	5 ejecutores	7 ejecutores
256 bytes	107.141	112.755	118.357	123.959
512 bytes	107.153	112.743	118.357	123.959
1024 bytes	107.153	112.755	118.357	123.947
2048 bytes	107.153	112.755	118.357	123.959

TABLA IV: Costo de publicación final (realizada por el último ejecutor, ejecutando la lógica adicional de cierre de fase) por cantidad de ejecutores y por tamaño de respuesta.

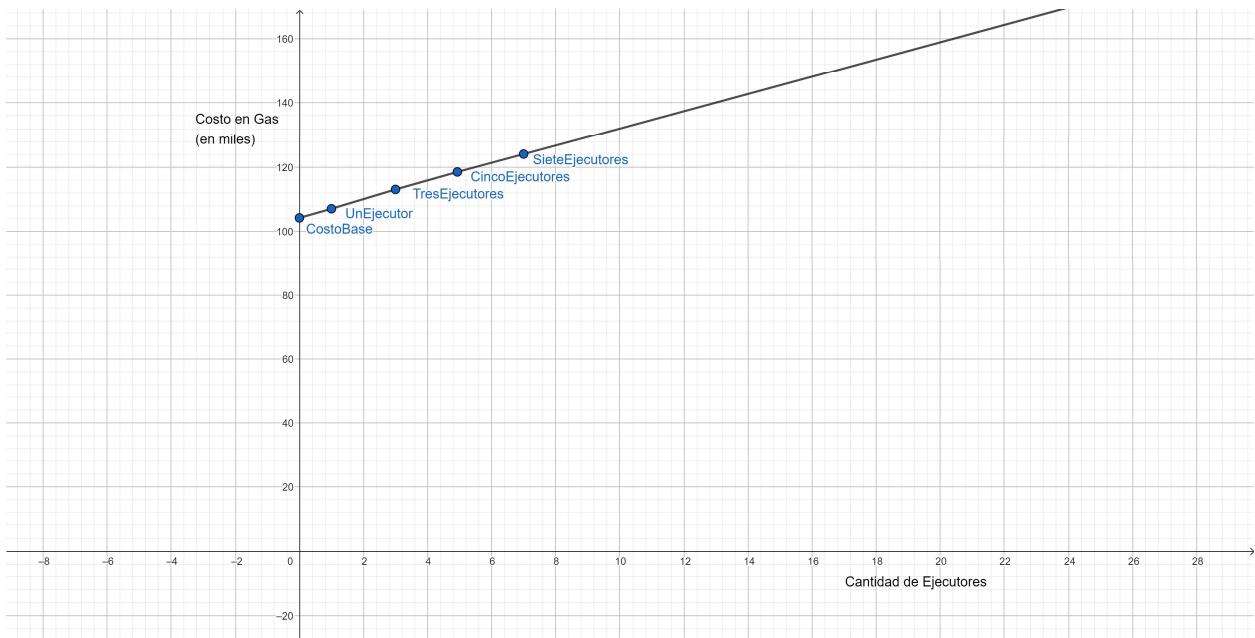


FIGURA XVIII: Gráfico del costo de publicación final de resultados por la cantidad de ejecutores contratados.

En cuanto al proceso de liberación, como sí involucra la publicación de un resultado directo, y su procesamiento, se ve que el costo aumenta de manera lineal, con una pendiente pronunciada, respecto del tamaño de la estructura a procesar. Para las liberaciones iniciales, la cantidad de ejecutores no tiene efecto alguno en el costo (como se ve en la Tabla TABLA V y la Figura FIGURA XIX a continuación), ya que solo se itera sobre los mismos en el cierre de la petición, llevado a cabo por la liberación final.

	3 ejecutores	5 ejecutores	7 ejecutores
256 bytes	286.727	288.166	286.252
512 bytes	452.965	457.276	455.841
1024 bytes	782.577	782.580	784.017
2048 bytes	1.449.007	1.449.728	1.449.968

TABLA V: Costo promedio de liberación parcial de resultados (solo aplicable cuando hay más de un ejecutor) por cantidad de ejecutores y por tamaño de respuesta.

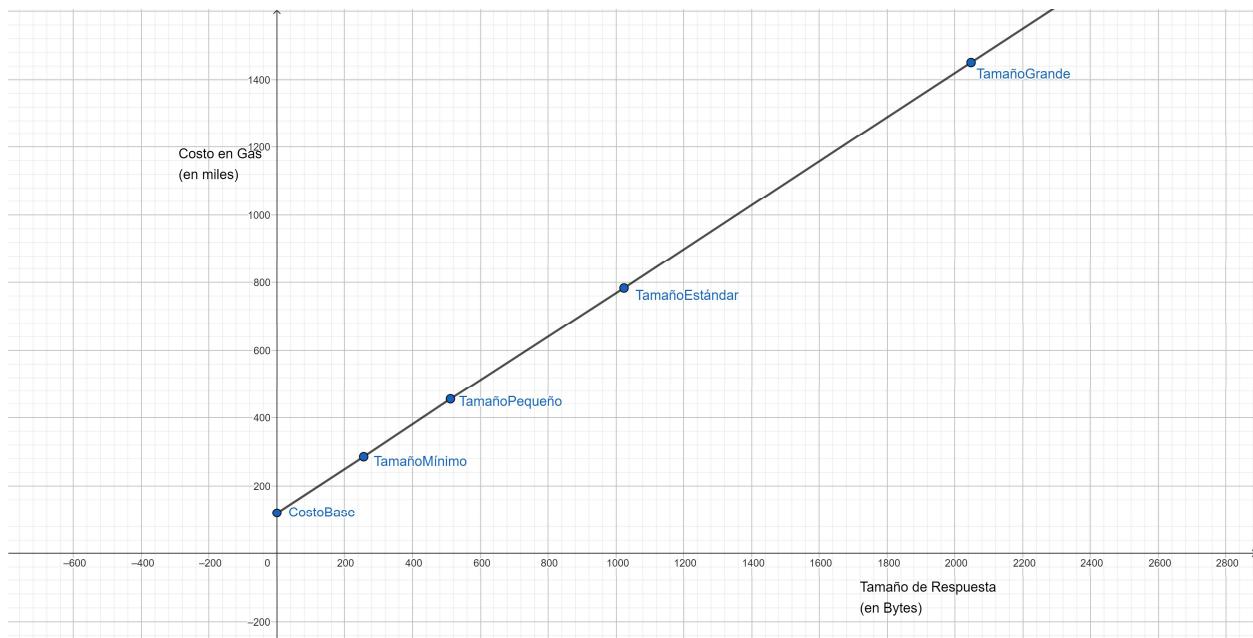


FIGURA XIX: Gráfico del costo de liberación inicial de resultados por el tamaño de los resultados liberados.

En esta última interacción, se ve claramente como tanto la cantidad de ejecutores asignados, como el tamaño de la respuesta a procesar afectan también de manera lineal, y ambas con un coeficiente pronunciado.

Para medir el impacto de ambas variables en este proceso, se graficó un plano en R3 (como se ve en la Tabla TABLA VI y la Figura FIGURA XX a continuación). Es lógico que este sea el proceso más costoso y con mayor dependencia respecto de las dos variables, ya que involucra el de cierre de la petición.

	1 ejecutor	3 ejecutores	5 ejecutores	7 ejecutores
256 bytes	666.293	870.650	1.086.290	1.313.223
512 bytes	999.589	1.203.949	1.419.590	1.646.526
1024 bytes	1.666.188	1.870.552	2.086.198	2.313.138
2048 bytes	2.999.413	3.203.786	3.419.441	3.646.390

TABLA VI: Costo de liberación final de resultados (realizada por el último ejecutor, ejecutando la lógica adicional de cierre de fase) por cantidad de ejecutores y por tamaño de respuesta.

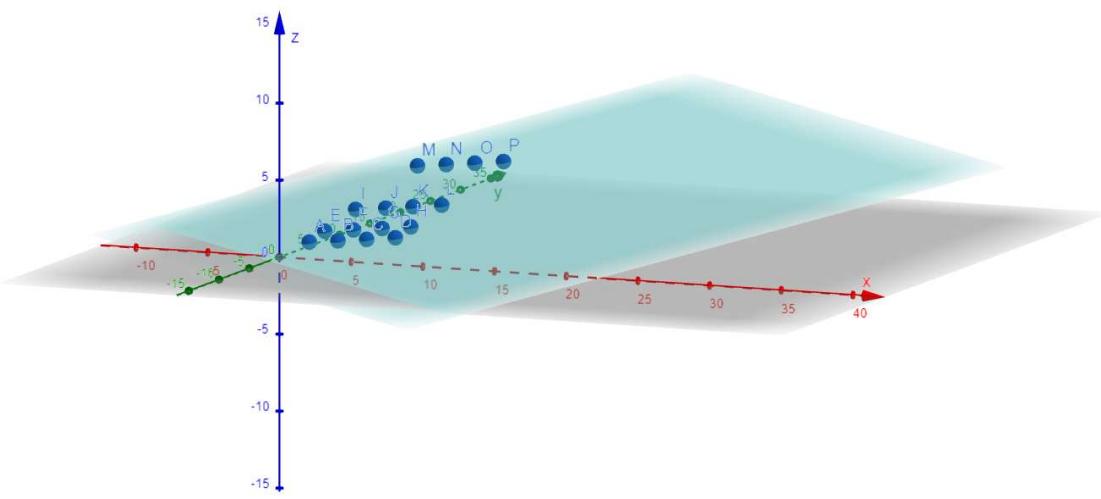


FIGURA XX: Gráfico en R3 del costo liberación final de resultados por la cantidad de ejecutores contratados y el tamaño de los resultados liberados.

Por último, si se suman todos los costos en gas de una petición resuelta sin inconvenientes, y multiplicando los gastos repetidos por la cantidad de ejecutores, aun así, se obtiene una relación lineal como se observa en la Figura FIGURA XXI a continuación.

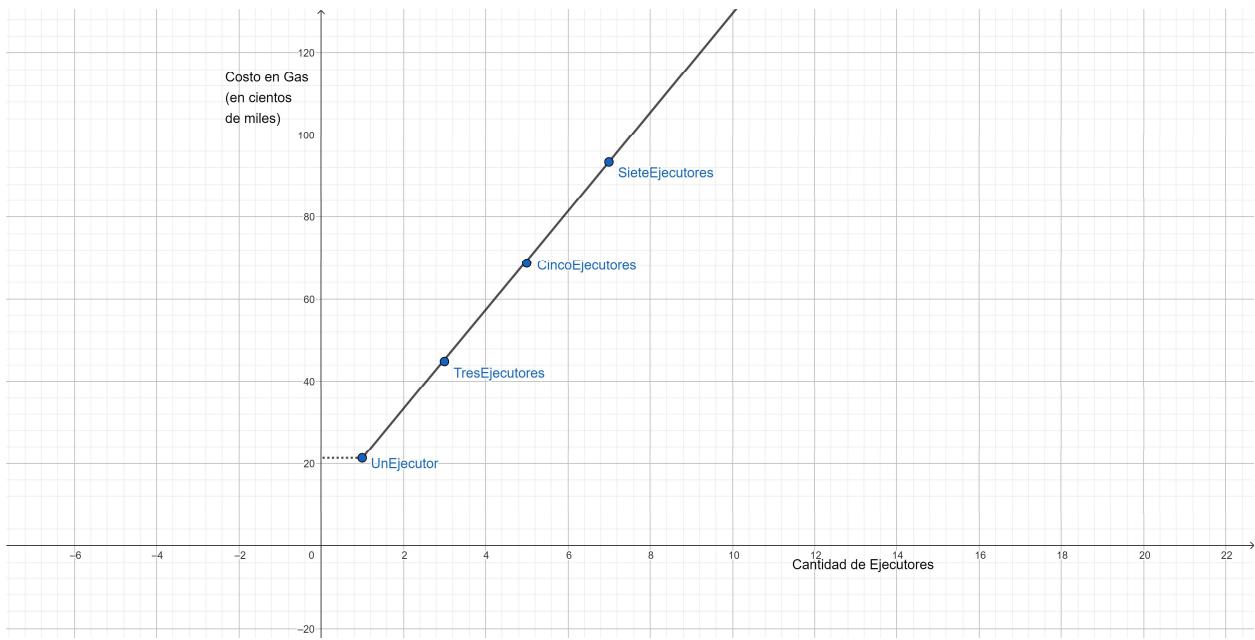
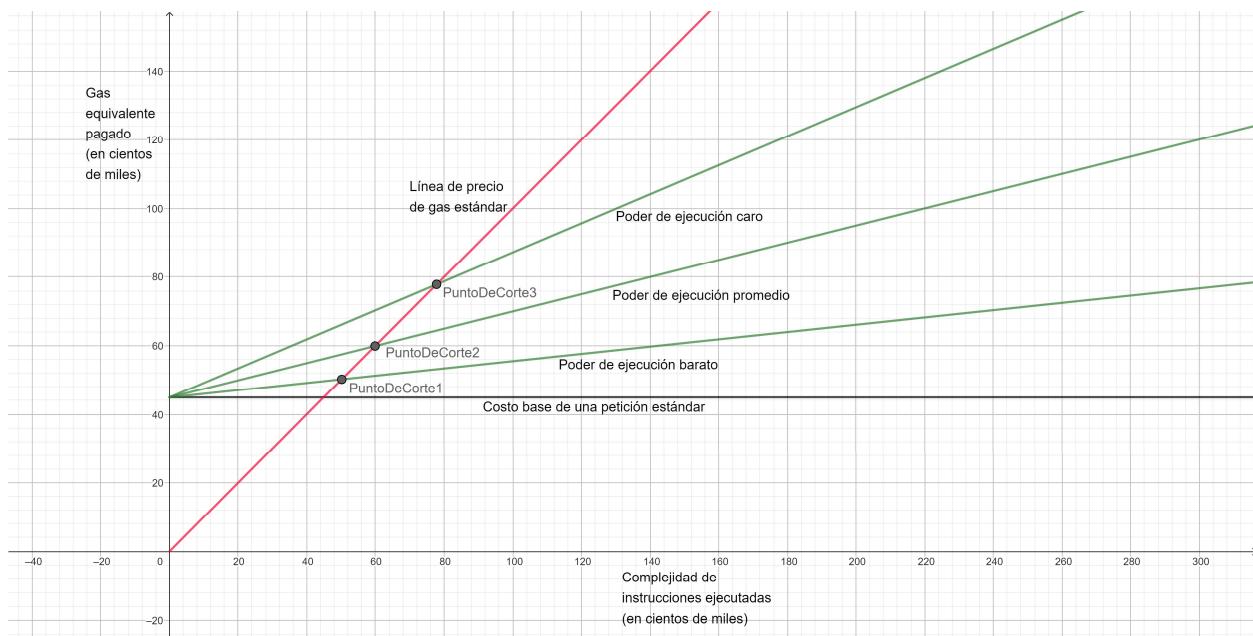


FIGURA XXI: Gráfico del costo añadido total por petición por cantidad de ejecutores.

Para hacer el análisis de rentabilidad, se tomó un caso estándar de 3 ejecutores y un tamaño de respuesta de 1024 bytes. En definitiva, hay un costo mínimo de petición que escala con la cantidad de ejecutores (Figura FIGURA XXI). Para el caso estándar, este costo base es de 4.487.000 de gas (**recta negra** de la Figura FIGURA XXII a continuación).



*FIGURA XXII: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del protocolo Open Chain Computing para distintos valores de poder de ejecución.*

Luego está el costo extra a pagar como retribución para cada ejecutor, que depende de la cantidad de instrucciones ejecutadas, y del precio del poder de ejecución. En conclusión, para que sea rentable la utilización del protocolo Open Chain Computing, el costo a pagar debe ser menor al equivalente para una ejecución de tal complejidad de modo onchain (**recta roja** de la Figura FIGURA XXII). Al igual que en el caso de Optimistic Execution, entre más caro sea el costo del poder de ejecución (gas virtual en OE), mayor deberá ser la cantidad de instrucciones a ejecutar para superar el punto de corte de conveniencia (definido por la intersección entre la **recta roja** de precio de gas estándar y las **rectas verdes** de precio de poder de ejecución), respecto de la ejecución onchain. Para un caso promedio, el punto de corte se da alrededor de los **6.000.000** de unidades de gas.

Cabe aclarar que este protocolo soporta una complejidad infinita, mientras que el límite máximo de complejidad onchain (y a su vez de Optimistic Execution al requerir una lógica de desafíos onchain), es de 30.000.000 de gas como se mencionó en la especificación del protocolo anterior.

### 5.2.6 Pros y Contras

#### Pros:

- Amplía la capacidad de procesamiento de la red para procesos de tamaño y complejidad elevados.
- El protocolo no posee ninguna limitación en cuanto al tamaño del código a ejecutarse.
- No requiere que el cliente despliegue su propio contrato, resultando en un ahorro de gas.
- Al no depender necesariamente de solidity, el entorno de ejecución posee más flexibilidad (dentro de lo permitido por un lenguaje determinista).
- No hace falta un pre-cálculo estimativo de parte de los ejecutores ya que estos son asignados automáticamente.

#### Contras:

- Los resultados obtenidos no son infalibles, y se requiere de más ejecutores para mejorar la confiabilidad de los mismos.
- El hecho de que se asignen varios ejecutores a una petición implica un pago proporcionalmente mayor de parte del cliente.
- Se complejiza el sistema de resolución, partiéndose la publicación de resultados en dos fases.
- El precio del poder de ejecución requiere una definición algorítmica y la definición de su valor es más compleja que si se resolviera simplemente por oferta y demanda.
- El costo en gas de complejizar los datos de resultado, aumenta multiplicativamente con la cantidad de ejecutores involucrados.

### 5.3 Tercera Solución: Zero Trust Rollups

Hasta ahora las dos soluciones propuestas tienen algo en común, hay una compleja lógica de tratamiento de resultados, ya sea con desafíos, o redundancia, y en ambos casos no hay garantía absoluta de que los resultados finales sean correctos. La manera más sencilla de implementar un rollup es con un protocolo que automáticamente verifique la integridad de un resultado al instante de publicarse, sin necesidad de una lógica de desafíos, pero que logre de alguna manera ejecutar esta verificación sin tener que ejecutar la lógica en cuestión. Si se restringe el alcance del rollup a problemas de resolución compleja pero fácil verificación se puede lograr esto mismo.

### 5.3.1 Propuesta

La idea es prescindir de la complejidad añadida por la verificación de resultados a base de incentivos y estadística, y reemplazarla por una lógica sencilla y determinista. Para que tenga sentido el rollup, la lógica de verificación debe ser mucho más simple que la lógica de resolución. En matemáticas y programación, hay una clase de problemas llamados problemas NP (Non-deterministic Polynomial time). Esta clase de problemas se caracteriza por la propiedad de que, dada una posible solución, es posible verificar en tiempo polinomial si la misma es correcta. Sin embargo, encontrar la solución óptima o determinar si una existe en sí misma no puede resolverse en tiempo polinomial con algoritmos deterministas conocidos. Teniendo este concepto en mente, si se restringe el protocolo a este tipo de problemas, es muy sencillo implementar una función simple para la comprobación de resultados de cada problema, dejando que el código de resolución complejo (en tiempo no-polynomial), exista y corra por afuera de la cadena (offchain).

En cuanto a la implementación de los contratos, una solución de estas características se parece mucho a la primera, ya que hay comprobación de resultados onchain, pero esta vez, sin un sistema de desafíos, sino que dicha comprobación es inmediata e inevitable, haciendo que los resultados sean 100% íntegros.

Es fácil ver la similitud entre este nuevo protocolo y un ZK Rollup. Ambos son no-optimistas, ambos tienen demostraciones inmediatas y sencillas, y ambos son no interactivos. Los ZTR, sin embargo, difieren de los ZK Rollups en que la comprobación de integridad no se hace necesariamente a través de un mecanismo de conocimiento cero. En este caso, los resultados sí son revelados ante el algoritmo de comprobación de validez.

### 5.3.2 Funcionamiento y características

En cuanto al funcionamiento del sistema es muy similar a la primera propuesta. La diferencia yace en que no hay lógica de desafío, sino que cada vez se publica un resultado, se ejecuta la lógica de comprobación implementada por el contrato cliente. Al

igual que con el primer caso, para evitar ataques de copia de resultados, se dividió la lógica de publicación en dos fases, una de aceptación y otra de publicación efectiva.

En este caso, el periodo de gracia de aceptación de una petición es mucho más grande, porque no hace falta que haya competencia entre nodos, ya que la comprobación es inmediata, determinista e infalible. El ejecutor aceptante deja una prima al aceptar la petición, la cual se retornará íntegramente en caso de una resolución correcta, y de la cual se descontará un porcentaje que será enviado al cliente en caso de una resolución incorrecta. Este porcentaje también se le deducirá a un ejecutor en caso de que cancele su aceptación, o de que, una vez pasado el periodo de gracia de dicha aceptación sin que se haya publicado ningún resultado, otro ejecutor le quite la aceptación. En caso de que se publique una respuesta incorrecta, la petición volverá a su estado inicial, y se emitirá el evento “requestReOpened” para que otros ejecutores puedan aceptarla. Cabe destacar que, bajo este modelo, no es necesaria la lógica de reclamo de pago, ya que el mismo se efectúa de manera automática en caso de que se haya publicado una solución correcta.

Como en este caso la ejecución de la lógica de chequeo es inevitable, al crear una petición, es necesario que el cliente aclare cuánto gas consume la comprobación de resultados, para que los nodos puedan sustraer ese valor de la paga total y de esta manera comprobar la rentabilidad de la petición.

Otra sutil diferencia con el protocolo Optimistic Execution, es en el contrato abstracto del cliente. Debido a que la función para comprobar un resultado toma tanto la entrada como la salida para poder realizar la comprobación y los chequeos pertinentes, además de una función de lectura que retorne la estructura de los datos de entrada, debe haber una función de lectura que retorne la estructura de los datos de la respuesta.

La última diferencia yace en que, en este protocolo, la única función que acaba por ejecutar lógica de procesamiento de resultados, y por ende vulnerable a un ataque de tipo “delegate call”, es la función “submitResult”. Es por eso que, al igual que

“claimPayment” y “challengeSubmission” en el protocolo “Optimistic Execution”, necesita ser llamada a través del contrato cliente, y no de manera directa.

Cabe destacar una cualidad única de los ZTR, que es que permiten la ejecución de procesos criptográficamente asimétricos, es decir, aquellos procesos en los que se requiere que parte de la información involucrada en el cálculo permanezca oculta. Este tipo de procesos no se puede realizar de manera directa en la blockchain, ya que todo lo que sucede allí es totalmente público. Un ejemplo de este tipo de procesos es la comprobación determinística de firmas, es decir, se puede delegar la obtención de ciertos datos a un tercero, con la garantía absoluta de que fue este tercero quien proveyó dichos datos, gracias a la capacidad de firmarlos digitalmente. Luego, en la blockchain, mediante un ZTR, se puede comprobar que dichos datos provienen indefectiblemente de una fuente autentica, eliminando la necesidad de confiar en quien los publica. Desde este punto de vista, se podría considerar a los zk-Rollups, un caso particular de los ZTR, ya que cumplen con las características de confianza cero y simplicidad de comprobación.

### 5.3.3 Estructura de los Contratos

Esta solución es una adaptación de la propuesta “Optimistic Execution” y, por ende, en esta sección, los elementos que permanezcan inalterados solo se nombrarán, mientras que los elementos nuevos o modificados serán explicados con más detalle. Al igual que en la solución original, se cuenta con dos contratos, uno desplegable que administra la creación de peticiones, así como la interacción con las mismas (**ExecutionBroker**), y un contrato abstracto a modo de plantilla de implementación para los casos de uso particulares de los clientes (**BaseClient**).

La estructura del contrato **ExecutionBroker** es la siguiente:

#### **Estructuras definidas:**

- Request: igual que en el caso original, con las diferencias que no hay seguro de desafío, y el resultado está expresado directamente en forma de bytes, y no con una estructura separada, ya que no se necesita mantener un registro, debido a que, si un resultado pasa el chequeo, se publica inmediatamente.
- Acceptance: igual que en el caso original.

**Variables internas del contrato:**

- uint ACCEPTANCE\_GRACE\_PERIOD: igual que en el caso original.
- uint ACCEPTANCE\_STAKE: al no haber seguro de desafío, se fija una cantidad estándar de stake que deben poner los ejecutores al aceptar una petición (a modo de incentivo para que no bloquen una petición indefinidamente), y del cual se sustraerá un porcentaje en caso de que el resultado publicado no pase el chequeo.
- Request[] requests: igual que en el caso original.

**Eventos:**

- requestCreated: como en el caso original, aunque sólo contiene el ID de la petición, el pago ofrecido por el cliente, y el gas total de procesamiento.
- requestCancelled: igual que en el caso original.
- requestAccepted: igual que en el caso original.
- acceptanceCancelled: igual que en el caso original.
- requestCompleted: se dispara cuando un ejecutor publica los resultados de una petición, y los mismos pasan el chequeo del cliente. Contiene el ID de la petición y una bandera indicando si el pago se realizó con éxito.
- requestReOpened: se dispara cuando un ejecutor publica los resultados de una petición, y los mismos no pasan el chequeo del cliente. Contiene todos los datos del evento “requestCreated”, y una bandera indicando si el reintegro parcial al ejecutor fue exitoso.

**Funciones públicas:**

- submitRequest [Payable]: igual que en el caso original.
- cancelRequest: igual que en el caso original.
- publicizeRequest: igual que en el caso original.
- acceptRequest [Payable]: como en el caso original, pero espera recibir un pago igual al valor de ACCEPTANCE\_STAKE en lugar del seguro de desafío.
- cancelAcceptance: igual que en el caso original.
- submitResult: comprueba si el resultado entregado por el ejecutor aceptante pasa el chequeo del contrato cliente. En caso de hacerlo, se le entrega el pago al ejecutor, y se emite el evento “requestCompleted”. En caso contrario, se le resta un porcentaje del stake al ejecutor (que se le envía al cliente), se le reintegra el resto del stake al ejecutor, y resetea el estado de la petición emitiendo el evento “requestReOpened”. Recibe el ID de la petición, y el resultado codificado en bytes. Solo puede ser llamada por el ejecutor aceptante, y si no hay ningún resultado publicado aún. También solo puede ser llamada a través del contrato cliente, para

mitigar la vulnerabilidad de “delegate call”. Retorna una bandera indicando si los resultados publicados pasaron el chequeo del cliente.

La estructura del contrato **BaseClient** es la siguiente:

**Variables internas del contrato:**

- ExecutionBroker brokerContract: igual que en el caso original.
- mapping(uint => bool) activeRequestIDs: igual que en el caso original.

**Eventos:**

- requestSubmitted: igual que en el caso original.
- resultProcessed: igual que en el caso original.

**Modificadores:**

- onlyClient: igual que en el caso original.

**Funciones extensibles (virtuales):**

- checkResult [view]: recibe dos conjuntos de bytes, representando las codificaciones para la entrada y salida del proceso calculado por el ejecutor, y corre la lógica implementada por el cliente, retornando verdadero o falso dependiendo de si el resultado publicado por el ejecutor pasó los chequeos o no. Es de tipo “view” y no “pure”, para darle más flexibilidad al cliente a la hora de definir su lógica de chequeo. Debido a que se prescinde de un sistema de varios ejecutores para la garantía de integridad, no es necesario que la función sea 100% determinista.
- processResult: igual que en el caso original.
- getInputStructure [pure]: igual que en el caso original.
- getResultStructure [pure]: igual que “getInputStructure”, pero refiere a la estructura del resultado.

**Funciones inmutables:**

- submitRequest [Payable]: igual que en el caso original.
- cancelRequest: igual que en el caso original.
- submitResult: llama a la función “submitResult” del contrato broker, y si esta se ejecuta exitosamente, llama a la implementación de la función “processResult” hecha por el cliente, limitando la cantidad de gas al valor fijado para el procesamiento de resultados en el momento de creación de la petición. Si el procesamiento se ejecuta correctamente, emite el evento “resultProcessed”. Solo

puede ser llamada sobre una petición activa perteneciente al contrato cliente en cuestión, y solo por el ejecutor que haya previamente aceptado la petición. Retorna una bandera indicando si la publicación de resultados se ejecutó correctamente.

#### 5.3.4 Ciclo de vida de una petición

Como se muestra en la Figura **FIGURA XXIII** a continuación, y de manera muy similar a Optimistic Execution, el ciclo de vida de una petición comienza cuando un cliente la crea a través del Contrato Cliente (paso 1, flecha blanca). Esto desemboca en una llamada interna a la función homónima de creación de petición en el Contrato Broker (paso 1, flecha blanca). Como resultado, se crea una petición con todos los datos y requerimientos del cliente, y se emite un evento que le deja saber a los ejecutores que una nueva petición fue creada. Posteriormente, los ejecutores comienzan a realizar el procesamiento requerido por la petición, y cuando alguno llegue al resultado final, procederá a aceptar dicha petición (a través del contrato broker), poniendo la cantidad requerida de ether como seguro de desafío y reservando de esta manera temporalmente el derecho a publicar un resultado (paso 2, flecha blanca).

Luego de que se confirme su reserva, el ejecutor publica los resultados calculados (paso 3, flecha blanca), esta vez, a través del contrato cliente (a diferencia de Optimistic Execution). Esto desemboca en un llamado a la función homónima en el contrato broker. La diferencia con Optimistic Execution es que un ZTR, al ser infalible, se comprueba automáticamente el resultado una vez publicado. Esto significa que la función llama a la lógica de comprobación de resultados del cliente, y, en caso de haber un error, simplemente detener la ejecución. En caso contrario, se guarda el resultado, y se lo procesa a la vez que se le paga al ejecutor, cerrando efectivamente la petición.

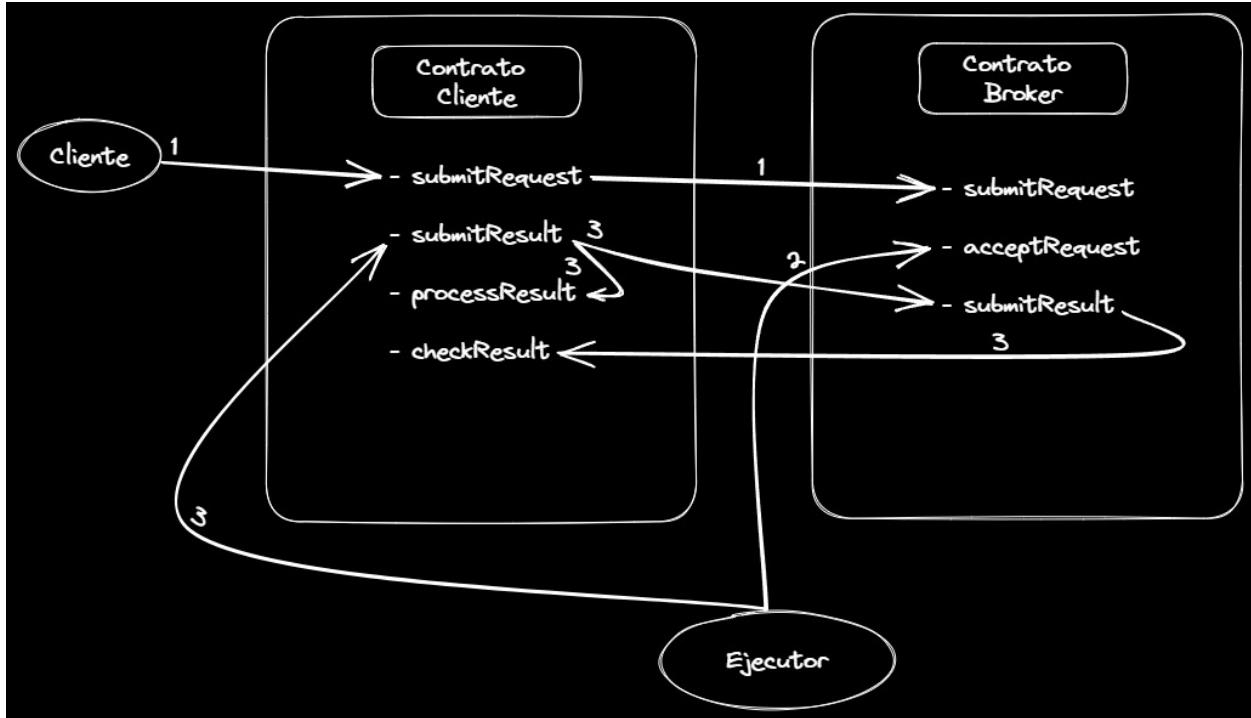


FIGURA XXIII: Ciclo de vida de una petición en Zero Trust Rollups.

### 5.3.5 Ejemplos de uso

Al igual que con las dos propuestas anteriores, para simular una cadena y probar las implementaciones prácticas de este protocolo, se utilizó Brownie, Python y Ganache. Para demostrar la utilidad de los Zero Trust Rollups, se plantearán dos ejemplos. Uno para mostrar las cualidades extra que ofrece este protocolo (la ejecución de procesos criptográficamente asimétricos), y otro para mostrar las cualidades de escalabilidad (ahorro de costos). El código de esta solución y de ambos ejemplos está en GitHub [29].

#### API Registration

El caso más sencillo de imaginarse es la comprobación de una firma digital. Una aplicación práctica, es un sistema de registro de APIs que permita vincular una API RESTful con una dirección, para poder corroborar la autenticidad de las respuestas onchain. Este ejemplo es uno de los casos de implementación de un proceso criptográficamente asimétrico, únicamente posible mediante un ZTR.

La implementación del sistema de API Registration consiste en dos contratos y dos scripts en, en este caso, python. El contrato “APIProvider” funciona simplemente a modo de registro de APIs, mapeando en un contrato centralizado un string representante de una API, con la dirección firmante de dicha API. Su script homónimo en python posee tres clases. APIProvider se encarga de la lectura y escritura del contrato homónimo. MockAPI simplemente posee una clave privada y retorna una estructura de datos firmada por la misma. Por último, APIOracle posee la lógica para interactuar con el contrato broker para interaccionar con una petición, y posee también lógica para interactuar con una MockAPI que posea la dirección correspondiente a la API de la petición, registrada en el contrato APIProvider.

Por otro lado, el contrato “APIConsumer” es una implementación de BaseContract, que tiene la lógica necesaria para verificar si una estructura de datos está efectivamente firmada por una dirección determinada. Este contrato posee una referencia al contrato broker, para la gestión de peticiones, y otra referencia al contrato APIProvider, para poder efectuar el chequeo de firmas. Asimismo, su script homónimo posee una clase encargada de codificar y publicar una nueva petición al contrato broker.

La ventaja en este ejemplo está más orientada a la transparencia que al escalamiento de la red, demostrando la versatilidad del protocolo. Mediante este mecanismo se puede obtener de manera infalible, teniendo cero confianza en los oráculos, que una estructura de datos proviene efectivamente de una API en particular. Para que este sistema funcione, sin embargo, los desarrolladores de las APIs en cuestión, deben integrarse al contrato APIProvider.

Este ejemplo en concreto es imposible compararlo con una alternativa onchain ya que requiere forzosamente que la firma se haga offchain. Por ende, se planteó el ejemplo como demostración de que los ZTR no solo ofrecen una solución de escalamiento, sino que también proveen nuevas posibilidades imposibles en la ejecución onchain.

El segundo ejemplo, implementa un principio muy similar a “proof of work”, haciendo que los ejecutores busquen por fuerza bruta una clave numérica con la que al procesar su “hash”, se obtenga un resultado específico. De esta manera se puede implementar un sistema para el descifrado parcial y paulatino de una gran clave, particionada en tantos fragmentos como el cliente lo desee, siendo cada fragmento la clave numérica a calcular por los ejecutores en cada petición. Si además de la lógica de chequeo del hash (que parte del concepto de fuerza bruta), se implementa el sistema de manera tal que el hash a buscarse en una petición, sea igual al hash de la respuesta anterior concatenado con un nuevo código numérico, podemos garantizar por inviabilidad computacional, que nadie va a poder resolver una petición, sin que se haya resuelto la anterior. Esto es debido a que, si el tamaño de la clave numérica de cada fragmento es significativamente menor a el tamaño del hash, encontrar un número que satisfaga que el hash de la concatenación de un hash desconocido (en caso de que se desconozca el código de la petición anterior) con él mismo, implica encontrar por fuerza bruta un número del tamaño del hash.

Adicionalmente, se puede añadir otro chequeo que no permita que una petición sea resuelta, si la anterior no fue resuelta aún, desincentivando el “minado” desordenado de los fragmentos. Esta es otra ventaja de que la función de chequeo sea de tipo vista y no pura.

Por último, si nos interesa que los fragmentos sean “minados” en un plazo de tiempo predecible, se puede añadir un chequeo más, que asegure que haya pasado por lo menos una definida cantidad de tiempo desde el minado del fragmento anterior antes de poder minarse el fragmento actual, y si se estima que la cantidad de tiempo que toma minar un fragmento es ligeramente inferior el tiempo mínimo de espera por fragmento, se puede tener una buena garantía de que la totalidad de los fragmentos va a ser minada en una fecha predecible. Es importante que el tiempo de minado de un fragmento sea similar al tiempo de espera entre cada uno, porque si el tiempo minado es muy corto, alguien incentivado exclusivamente por el conocimiento de la clave total, podría minar

todos los bloques en privado y hacerse con la clave final (representada por la concatenación de todas las claves numéricas).

El principal uso de este sistema, es el de tener una buena garantía de que una pieza de información va a salir a la luz en un plazo predecible de tiempo. Solo basta con tomar cualquier bloque de información, generar una clave particionable en N fragmentos, utilizar dicha clave para encriptar el bloque, y luego publicar tanto el bloque encriptado como las N peticiones correspondientes a los fragmentos, para tener una buena garantía de que el bloque va a poder ser desencriptado cuando haya pasado como mínimo una cantidad predecible de tiempo. Nótese que entre más largo sea el plazo pretendido, menor será la precisión de la estimación, debido a que es difícil predecir a largo plazo el poder de cómputo disponible para el minado de fragmentos dentro de la plataforma.

Para darse una idea de la diferencia entre el costo en gas de utilizar un ZTR y un sistema completamente onchain, se particionó una clave generada aleatoriamente en solo 5 fragmentos, con una dificultad (cantidad de cifras del código de cada fragmento) de 3. Luego se utilizó esa clave para encriptar un texto (ver Anexos ANEXO I, ANEXO II, ANEXO III, ANEXO IV y ANEXO V), y se generaron los bloques TS3000 (uno por cada fragmento) para ser minados. En caso de querer realizar toda la ejecución onchain, se obtienen los siguientes números: el despliegue inicial del contrato es de 1.901.379 unidades de gas, mientras que el costo de minado de todos los fragmentos es de 6.732.257 unidades (Figuras FIGURA XXIV, FIGURA XXV y FIGURA XXVI a continuación).

```

function processResult(bytes calldata resultData) public override onlyClient {
    Result memory result = abi.decode(resultData, (Result));
    keyFragments[result.fragmentIndex].passcode = result.passcode;
    if (result.fragmentIndex == keyFragments.length - 1) {
        finalKey = keccak256(abi.encode(result.passcode));
        emit keyFullyMined();
    } else {
        keyFragments[result.fragmentIndex + 1].localHash = keccak256(abi.encode(result.passcode));
        if (postProcessingEnabled) {
            Input memory input = Input({
                fragmentIndex: result.fragmentIndex + 1,
                globalHash: keyFragments[result.fragmentIndex + 1].globalHash,
                localHash: keyFragments[result.fragmentIndex + 1].localHash,
                minTimestamp: result.timestampRestriction + minTimeFramePerFragment
            });
            _submitRequest(rewardPerFragment, abi.encode(input), postProcessingGas);
        }
    }
}

function resolveOnChain(bytes calldata inputData) public override view returns (bytes memory) {
    Result memory result;
    Input memory input = abi.decode(inputData, (Input));
    result.fragmentIndex = input.fragmentIndex;
    result.timestampRestriction = input.minTimestamp;
    for (uint passcode = 0; passcode < 10**fragmentDifficulty; passcode++) {
        if (keccak256(abi.encode(passcode, input.localHash)) == input.globalHash) {
            result.passcode = passcode;
            break;
        }
    }
    return abi.encode(result);
}

```

FIGURA XXIV: código en solidity para el minado onchain y la comprobación de fragmentos de una clave en TS3000 (basado en el contrato cliente).

```

def test_ts3000(self):
    fileName = "text_file"
    broker = BrokerFactory.getInstance()
    requestorAccount = Accounts.getFromIndex(0)
    initialRequestorBalance = requestorAccount.balance()
    requestor = TS3000Requestor(broker, requestorAccount, fileName, numberOfKeyFragments=5,
                                difficulty=3, paymentPerFragment=0, gas_price=1)
    requestDeploymentCost = initialRequestorBalance - requestorAccount.balance()
    reqID = requestor.getInitialRequestID()

    transactions = []
    while True:
        tx = broker.resolveRequestOnChain(reqID, {"from": Accounts.getFromIndex(1)})
        transactions.append(tx)
        if int(requestor.client.finalKey().hex(), 16) != 0:
            break
        reqID = tx.events["requestCreated"]["requestID"]
    finalKey = bytes(requestor.client.finalKey())

    print(f"Request deployment cost: {requestDeploymentCost}")
    print("-----")
    totalMiningCost = 0
    for i in range(len(transactions)):
        fragment = requestor.client.keyFragments(i).dict()
        miningCost = transactions[i].gas_used
        totalMiningCost += miningCost
        print(f"Fragment {i} mining cost: {miningCost}")
        print(f"\tglobalHash: {fragment['globalHash']}")
        print(f"\tlocalHash: {fragment['localHash']}")
        print(f"\tpasscode: {fragment['passcode']}")
    print("-----")
    print(f"totalMiningCost: {totalMiningCost}")
    print(f"finalKey: {requestor.client.finalKey()}")
    print("===== short test summary info =====")

```

FIGURA XXV: código en python del test de costos de una petición de 5 fragmentos de dificultad 3 en ejecución onchain.

```
----- Captured stdout call -----
Request deployment cost: 1901379

Fragment 0 mining cost: 1351827
    globalHash: 0xc5110b2c754466210719d22062af5c6732f577b513a8bd96327176899c29b929
    localHash: 0x8077777ae4769de06cbfd1c0b8b1f653b51ec156d91a9aca16a4102f19e03d9e
    passcode: 644
Fragment 1 mining cost: 1826857
    globalHash: 0x1b61aa23e1a0199cd0f18191251aa63d0ec9446e8213e204a5e3fd0ff3b761de
    localHash: 0xbc051cc3908f28992d3828b5e3a61dbaa81b2c34576a11d210e08d506c7564ad
    passcode: 972
Fragment 2 mining cost: 1789192
    globalHash: 0x818f0f6e885880bc49bb7a557a4791b4ef9207ea3b05ec1f2ae8ff036cf361cd
    localHash: 0xdee8ba79eae04290fb0f7ffd03c9950f2f50a43bb11588c5be8429d9359bb6e0
    passcode: 945
Fragment 3 mining cost: 1126636
    globalHash: 0x5a544f1da00119b262a783df1d248b18e1378d68afe776c2be12e2edde497998
    localHash: 0x27e4e2fe475a2fa91b453993a7d8a424752955f38b258c56ae95d030b4eac111
    passcode: 467
Fragment 4 mining cost: 637745
    globalHash: 0x5ab5f257d61b6776f9ffc793f4aff581ba69c634a8fd28aa82c4071da78b7d7a
    localHash: 0xf958fb397749aa20a86803c89952cf5b3c03183cc04a47ce81e2c499438466ee
    passcode: 279
-----
totalMiningCost: 6732257
finalKey: 0xa6c825f473ff3ad15e15e48cc3e5c35d6b1e6b3ae7623a44ed3bcf3e1c4d4054ac
===== short test summary info =====
```

*FIGURA XXVI: impresión de la ejecución del test de la Figura FIGURA XXV.*

En sumatoria, para el minado de la clave de este bloque de información, se llega a un total de 8.633.636 unidades de gas. Para tener un poco de perspectiva, el gasto total de gas en una simple transferencia de ether es de 21.000 (cifra que sale de la sumatoria del costo de operaciones de comprobación de la firma digital de una transacción) [12].

Por otro lado, la versión ZTR, al no incluir el código de resolución, sino que solamente el de corroboración de resultados, el costo de despliegue del contrato cliente es ligeramente más barato, costando 1.814.574 de gas. Además, el costo de gas total que proviene de todas las interacciones de los ejecutores con el contrato broker para cada fragmento, a ser pagado por los ejecutores, es en este caso de tan solo 2.253.208 de gas (Figuras FIGURA XXVII y FIGURA XXVIII a continuación).

```

def test_ts3000(self):
    fileName = "text_file"
    broker = BrokerFactory.getInstance()
    requestorAccount = Accounts.getFromIndex(0)
    initialRequestorBalance = requestorAccount.balance()
    requestor = TS3000Requestor(broker, requestorAccount, fileName, numberOfKeyFragments=5,
                                difficulty=3, paymentPerFragment=0, gas_price=1)
    requestDeploymentCost = initialRequestorBalance - requestorAccount.balance()
    reqID = requestor.getInitialRequestID()
    miner = TS3000Miner(broker, Accounts.getFromIndex(1))

    accTXs = []
    submitTXs = []
    while True:
        accTXs.append(broker.acceptRequest(reqID, {"from": miner.account, "value": BrokerFactory.ACCEPTANCE_STAKE}))
        fragment = miner.mineFragment(reqID, 5)
        tx = miner.submitFragmentResult(reqID, fragment)
        submitTXs.append(tx)
        if int(requestor.client.finalKey().hex(), 16) != 0:
            break
        reqID = tx.events["requestCreated"]["requestID"]
        finalKey = bytes(requestor.client.finalKey())

    print(f"Request deployment cost: {requestDeploymentCost}")
    print("-----")
    totalMiningCost = 0
    for i in range(len(accTXs)):
        fragment = requestor.client.keyFragments(i).dict()
        miningCost = accTXs[i].gas_used + submitTXs[i].gas_used
        totalMiningCost += miningCost
        print(f"Fragment {i} mining cost: {miningCost}")
        print(f"\tglobalHash: {fragment['globalHash']}")
        print(f"\tlocalHash: {fragment['localHash']}")
        print(f"\tpasscode: {fragment['passcode']}")
    print("-----")
    print(f"totalMiningCost: {totalMiningCost}")
    print(f"finalKey: {requestor.client.finalKey()}")
    print("===== short test summary info =====")

```

*FIGURA XXVII: código en python del test de costos de una petición de 5 fragmentos de dificultad 3 en el ZTR de TS3000.*

```

----- Captured stdout call -----
Request deployment cost: 1814574

Fragment 0 Acceptance Gas: 73370 Submission Gas: 407986
    globalHash: 0x5b60c2c264aafe3f53d61f5f4246a13be4d67774134617148abd4e6d27689dcf
    localHash: 0x330713cfcb9ba83322f5ea10b76918d1069f3fcf7d7fd317f5f7fb9bb3fa4237
    passcode: 825
Fragment 1 Acceptance Gas: 73382 Submission Gas: 427210
    globalHash: 0x01618ec074c778824ea6f8089cc5ffef7f580c64c3338f6c3522334095398f7b
    localHash: 0x9f8f6b777320728647db1e5a2fb0e207151a23199c7ab6a5b1f835f5cd4c7b32
    passcode: 596
Fragment 2 Acceptance Gas: 73382 Submission Gas: 427210
    globalHash: 0xb1f7b9be3c32652d1fa4a6c0e1372bde41d3cd68d328cb316806d8c2a6cb5a2a
    localHash: 0x496c16b9a83d6378b861d2fde2546741db381f90a71c8aae6bab4811f6446a85
    passcode: 392
Fragment 3 Acceptance Gas: 73382 Submission Gas: 427210
    globalHash: 0x87a3be7f848a71a55a6ac7509a4aa8687347f352349631af379274bfd78c8d29
    localHash: 0xebeb61f9a7cf51dfe2f7af2492fdcd2702cacee857677c504c9282a939b0fe659
    passcode: 837
Fragment 4 Acceptance Gas: 73382 Submission Gas: 196694
    globalHash: 0xc76f504d60ded47866072639a1cb1b741929ffff2dd7b24a8e37e9d24997df2a4
    localHash: 0x3774c1f30e2118f59a05a7402b3f826b4765d85da75a7905339613f69d4791d5
    passcode: 143

totalMiningCost: 2253208
finalKey: 0x337f7913db22d91ef425f82102bc8075ef67e23a2be359965ea316e78e1eff3f
===== short test summary info =====

```

*FIGURA XXVIII: impresión de la ejecución del test de la Figura FIGURA XXVII.*

Todo esto suma un total de 4.067.782 unidades. Como se observa en la Figura FIGURA XXVIII, el costo de aceptación de los fragmentos es constante, pero el de publicación de resultados es ligeramente más barato en el primer fragmento (debido a una simplificación de operaciones para almacenar datos en el primer elemento de un arreglo), y substancialmente más barato en el último fragmento (costando solo la mitad ya que se carece de la necesidad de creación de una nueva petición al procesar los resultados del mismo).

Tomando los precios de gas y Ether utilizados como referencia en el análisis y exemplificación de Optimistic Execution, se llega a que para el caso onchain, el cliente pagaría un total de 1.047,70 USD. Por otra parte, utilizando ZTR, el costo inicial en gas del cliente sería de solo 220,20 USD, dejando un margen de 827,50 USD para la negociación del pago a ejecutores. Estos últimos, de manera similar al primer protocolo, tienen un piso de 273,40 USD.

Con estos números, se puede calcular que, en el caso promedio (los pagos mínimos de creación de petición y el gas de ejecución, sumados a la mitad de la diferencia entre el límite inferior y el límite superior), para esta ejecución en particular, el cliente estaría pagando 770,70 USD, es decir, un 74% de lo que pagaría de ejecutar todo el procedimiento onchain. Además, en el mejor de los casos, el cliente pagaría solo 493,60 USD (solo los pagos mínimos de creación de petición y de gas de ejecución), resultando en un costo de tan solo el 47% del costo del caso onchain.

Al igual que con el caso de Optimistic Execution, la diferencia entre ambas alternativas crece con la complejidad del problema a resolver, pero a la hora de realizar una simulación para hacer los cálculos con un ejemplo concreto, las limitaciones de los programas simuladores nos impiden demostrar el verdadero potencial de este protocolo. De igual manera, si se implementa el protocolo en una red compatible con EVM, pero con un costo de gas mucho menor, se mantendrían las diferencias porcentuales, pero se reduciría el costo a nivel USD. Para demostrar el crecimiento de la brecha de costos entre un ZTR y un equivalente onchain, se creó una versión altamente simplificada del

contrato TS3000, sin usar un broker, e incluso sin tener un mecanismo de vinculación de fragmentos con hashes locales y globales (como muestran las Figuras FIGURA XXIX, FIGURA XXX y FIGURA XXXI a continuación).

```

constructor(bytes32[] memory _targetHashes, uint8 _fragmentDifficulty) {
    currentIndex = 0;
    fragmentDifficulty = _fragmentDifficulty;
    for (uint8 i = 0; i < _targetHashes.length; i++) {
        targetHashes.push(_targetHashes[i]);
    }
}

function resolveOnChain() public {
    bytes32 targetHash = targetHashes[currentIndex];
    for (uint32 passcode = 0; passcode < 10**fragmentDifficulty; passcode++) {
        if (keccak256(abi.encode(passcode)) == targetHash) {
            passcodes.push(passcode);
            currentIndex++;
            return;
        }
    }
    revert("No number of the set difficulty match the target hash for this round!");
}

```

FIGURA XXIX: código en solidity para el minado onchain de fragmentos de una clave en TS3000 (versión simplificada).

```

def test_onchain_ts3000(self):
    amount = 30
    difficulty = 4
    passcodes, hashes = self.get_test_data(amount, difficulty)
    requestorAccount = Accounts.getFromIndex(0)
    initialRequestorBalance = requestorAccount.balance()
    contract = OnchainTS3000.deploy(hashes, difficulty, {"from": requestorAccount, "gas_price": 1})
    requestDeploymentCost = initialRequestorBalance - requestorAccount.balance()

    transactions = []
    calculatedPasscodes = []
    for i in range(amount):
        transactions.append(contract.resolveOnChain({"from": requestorAccount}))
        calculatedPasscodes.append(contract.passcodes(i))

    print(f"Request deployment cost: {requestDeploymentCost},")
    print("-----")
    totalMiningCost = 0
    for i, tx in enumerate(transactions):
        print(f"Fragment {i} mining cost: {tx.gas_used},")
        totalMiningCost += tx.gas_used
    print("-----")
    print(f"totalMiningCost: {totalMiningCost},")
    print("-----")
    print(f"Original Passcodes: {passcodes}")
    print(f"Calculated Passcodes: {calculatedPasscodes}")

```

FIGURA XXX: código en python del test de costos de una petición de 30 fragmentos de dificultad 4 en la versión simplificada onchain de TS3000.

```
-- Captured stdout call --
Request deployment cost: 995,422
-----
Fragment 0 mining cost: 8,441,353
Fragment 1 mining cost: 4,057,013
Fragment 2 mining cost: 7,421,777
Fragment 3 mining cost: 12,940,124
Fragment 4 mining cost: 3,729,906
Fragment 5 mining cost: 12,980,857
Fragment 6 mining cost: 8,804,241
Fragment 7 mining cost: 1,552,941
Fragment 8 mining cost: 2,414,977
Fragment 9 mining cost: 2,962,334
Fragment 10 mining cost: 9,886,067
Fragment 11 mining cost: 12,236,802
Fragment 12 mining cost: 1,253,453
Fragment 13 mining cost: 4,478,076
Fragment 14 mining cost: 6,106,153
Fragment 15 mining cost: 8,300,441
Fragment 16 mining cost: 3,487,545
Fragment 17 mining cost: 13,693,822
Fragment 18 mining cost: 1,999,796
Fragment 19 mining cost: 4,715,018
Fragment 20 mining cost: 1,387,923
Fragment 21 mining cost: 11,788,337
Fragment 22 mining cost: 6,990,440
Fragment 23 mining cost: 8,735,648
Fragment 24 mining cost: 7,338,583
Fragment 25 mining cost: 4,000,967
Fragment 26 mining cost: 12,530,250
Fragment 27 mining cost: 2,510,844
Fragment 28 mining cost: 4,566,882
Fragment 29 mining cost: 11,521,121
-----
totalMiningCost: 202,833,691
-----
Original Passcodes: [5668, 2761, 5015, 8602, 2539, 8628, 5926, 1048, 1631, 2016, 6633, 8152, 841, 3046, 4140, 5595, 2364, 9082, 1356, 3206, 934, 7864, 4729, 5881, 4950, 2723, 8340, 1707, 3106, 7692]
Calculated Passcodes: [5668, 2761, 5015, 8602, 2539, 8628, 5926, 1048, 1631, 2016, 6633, 8152, 841, 3046, 4140, 5595, 2364, 9082, 1356, 3206, 934, 7864, 4729, 5881, 4950, 2723, 8340, 1707, 3106, 7692]
===== short test summary info =====
```

*FIGURA XXXI: impresión de la ejecución del test de la Figura FIGURA XXX.*

Para el minado de una serie de 30 fragmentos de dificultad 4, el contrato simplificado tiene un costo de despliegue de apenas 995.000 de gas. Sin embargo, la suma total del minado de los 30 fragmentos tiene un costo que supera las 200.000.000 unidades (Figura FIGURA XXXI). En comparación, el ZTR, si bien tiene un costo de despliegue de 2.400.000, es decir, más del doble, para el minado de esa misma cantidad de fragmentos, de la misma complejidad, el costo total es menor a 15.000.000 (Figura FIGURA XXXII a continuación).

```
----- Captured stdout call -----  
Request deployment cost: 2,417,677  
  
Fragment 0 Acceptance Gas: 73,370 Submission Gas: 407,986  
Fragment 1 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 2 Acceptance Gas: 73,382 Submission Gas: 427,198  
Fragment 3 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 4 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 5 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 6 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 7 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 8 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 9 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 10 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 11 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 12 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 13 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 14 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 15 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 16 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 17 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 18 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 19 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 20 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 21 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 22 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 23 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 24 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 25 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 26 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 27 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 28 Acceptance Gas: 73,382 Submission Gas: 427,210  
Fragment 29 Acceptance Gas: 73,382 Submission Gas: 196,706  
  
totalMiningCost: 14,768,008  
finalKey: 0x930fa885ae38d7754c7ba89941c109236f7205b31e2776cc3958def1cb4b6ff7  
===== short test summary info =====
```

FIGURA XXXII: impresión de la ejecución del test de la Figura FIGURA XXVII (adaptado) para una petición de 30 fragmentos de dificultad 4 en el ZTR de TS3000.

Esto deja en evidencia que, incluso cuando busquemos hacer una comparación desfavorable reduciendo el alcance de la versión onchain, con el fin de abaratarlo lo más posible, si bien el despliegue inicial tiene un costo de menos de la mitad, para procedimientos de complejidad elevada, como este, el costo total en ZTR representa menos de un 9%.

A modo de establecimiento de un piso mínimo de costos, se calculó que, en definitiva, la viabilidad económica de este protocolo se da para todos los problemas aplicables (aquellos en los que la corroboración de las soluciones sea más simple que el cálculo de

las mismas), cuyo costo de realización de manera onchain supere la diferencia de gastos generales (suma entre **curva azul** y **recta naranja**), sumados a el valor de mercado del “gas virtual” (**recta verde**) (Figura FIGURA XXXIII a continuación). La forma de la curva del costo de procesamiento (**curva azul**), depende de la naturaleza del problema, pero es siempre de orden menor a la complejidad de la lógica de obtención de resultados (para problemas NP).

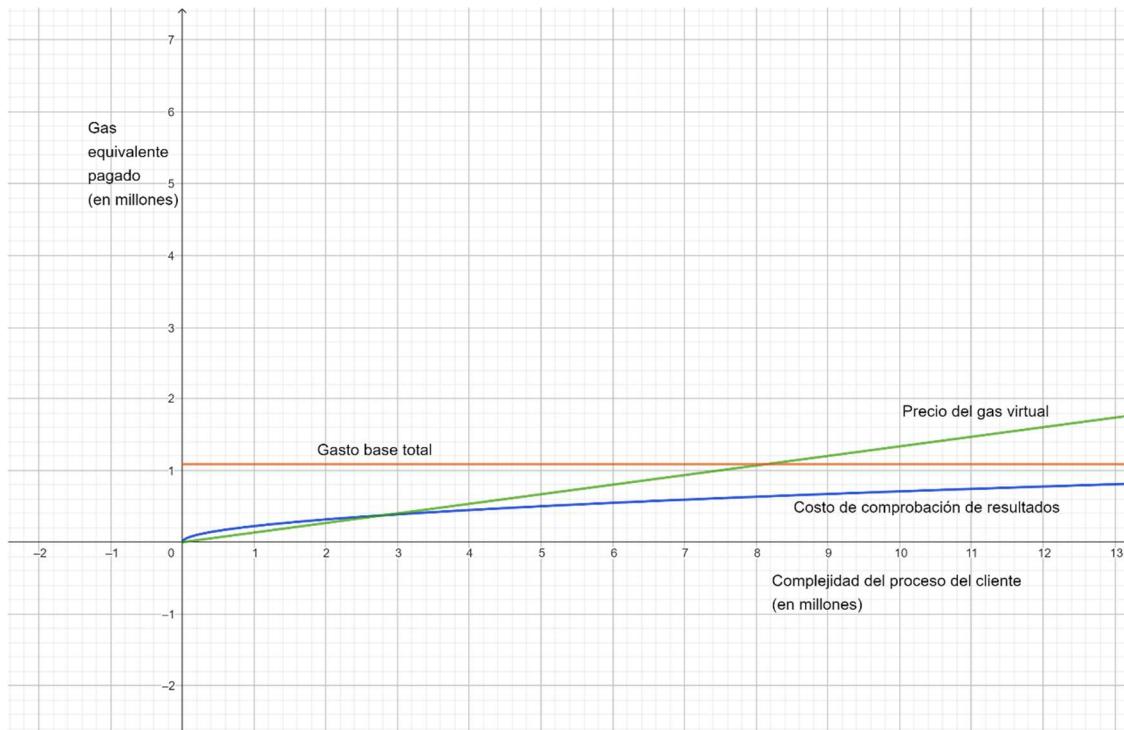


FIGURA XXXIII: Gráfico de los distintos componentes del costo del ZTR de TS3000.

Al igual que en el análisis de optimistic Execution, y por omisión en Open Chain Computing, debido a que un contrato cliente puede ser reutilizado infinitas veces, no se tendrá en cuenta su costo de despliegue para este análisis. En cuanto al cálculo del costo base (**recta naranja**) de gestión de una petición, se realizaron pequeñas modificaciones a la implementación de TS3000, para poder medir diferenciadamente el costo de creación de petición del costo de despliegue del contrato. Además, se forzó el tamaño de la estructura de resultado para que fuera de 1KB. Los resultados de correr el test de la Figura FIGURA XXVII luego de estas modificaciones, se muestran en la Figura FIGURA XXXIV a continuación.

```
Contract deployment cost: 1,778,046
Request creation cost: 222,387
-----
Fragment 0 Acceptance Gas: 73,370 Submission Gas: 1,000,180
Fragment 1 Acceptance Gas: 73,382 Submission Gas: 1,019,404
Fragment 2 Acceptance Gas: 73,382 Submission Gas: 1,019,416
Fragment 3 Acceptance Gas: 73,382 Submission Gas: 1,019,392
Fragment 4 Acceptance Gas: 73,382 Submission Gas: 788,909
-----
totalMiningCost: 5,214,199
finalKey: 0x90f1fbe211cc96d1ddedeccd2113dc32c31d712d12ce1f36d6a07e605dcf7d532
===== short test summary info =====
```

FIGURA XXXIV: impresión de la ejecución del test de la Figura FIGURA XXVII (adaptado) para una petición con tamaño de respuesta de 1KB en el ZTR de TS3000.

Si se suman los gastos de creación de la petición, el gasto de aceptación, y el gasto de publicación (se toman los 789.000 de la última petición ya que solo la última, por diseño de TS3000 no tiene lógica de procesamiento de resultados por ser la final), se obtiene que el costo base total es de 1.084.000 gas.

Es necesario tener en cuenta que, a diferencia de los dos protocolos anteriores, que tenían un mecanismo de incentivos para garantizar la integridad de las entregas, en un ZTR se necesita indefectiblemente una lógica de comprobación de resultados, lo que hace que los gastos generales no sean constantes, y dependan de la complejidad del problema a resolver. Sin embargo, como por definición los problemas NP tienen una comprobación de orden menor a la obtención de soluciones, la relación en complejidad de un resultado con su comprobación, se decidió modelar la curva de costo de procesamiento de resultados (**curva azul**, Figura FIGURA XXXIII), como una función logarítmica.

Sumando los costos totales, obtenemos una **curva negra** (de costo base total) que depende de una función constante (costo base de la solución), una función lineal (precio de mercado del gas virtual), y una función logarítmica (costo de procesamiento de entregas). De manera similar a los casos anteriores, el protocolo resulta rentable para cualquier problema con una complejidad superior a el punto de corte de las **curvas verdes** de precio/complejidad, con la **recta roja** (Figura FIGURA XXXV a continuación).

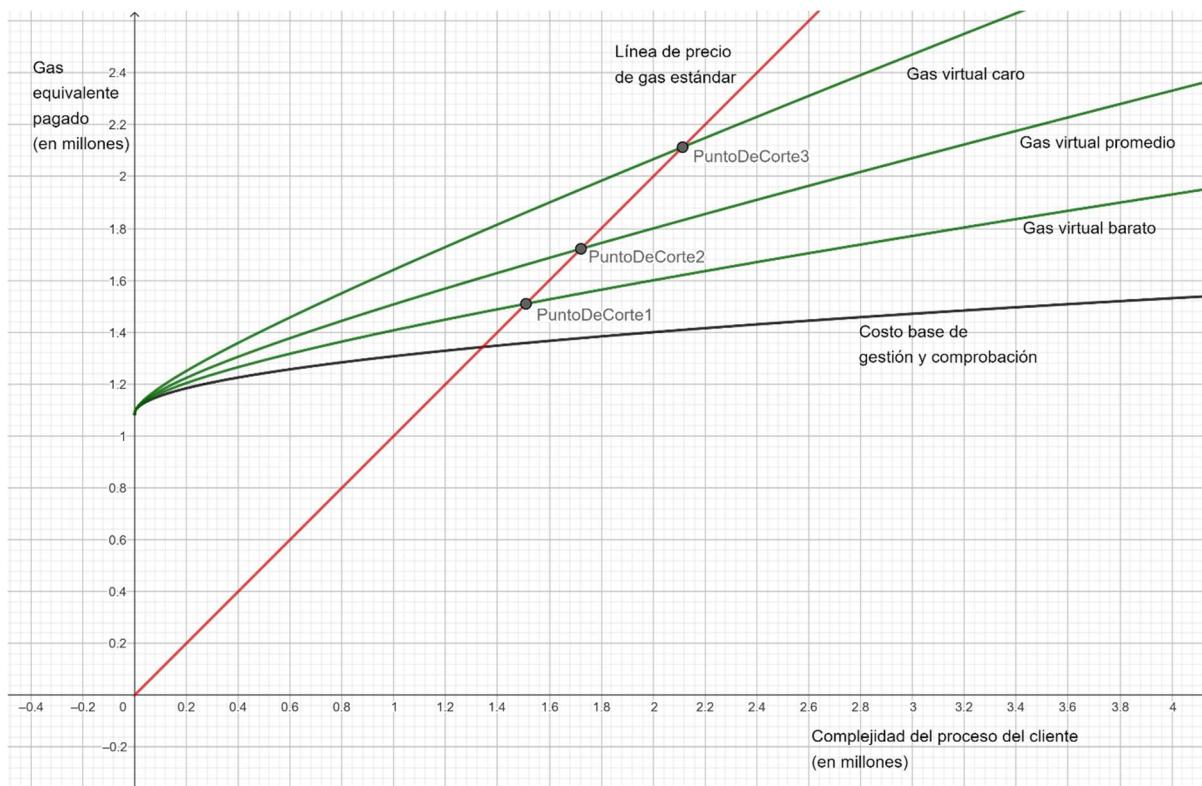


FIGURA XXXV: Gráfico comparativo entre los costos estándares de una ejecución onchain, y los costos del ZTR de TS3000 para distintos valores de mercado de gas virtual.

Nótese como, para un caso promedio, el punto de corte de rentabilidad se da en las **1.700.000** unidades de gas.

### 5.3.6 Pros y Contras

#### Pros:

- Amplía la capacidad de procesamiento de la red para problemas de tipo NP.
- Permite la ejecución de procesos criptográficamente asimétricos.
- El proceso a ejecutar puede ser arbitrariamente largo y complejo, siempre y cuando la comprobación de los resultados sea lo suficientemente simple, como para ejecutarse onchain.
- La comprobación de los resultados es determinista y garantiza su integridad.
- Los contratos son más sencillos ya que no hay lógica de desafío o redundancia y la lógica de los procedimientos a ejecutarse esta offchain.
- La definición de los precios de ejecución se da automáticamente de manera natural por un proceso de oferta y demanda.

- Si el cliente tiene la necesidad de ejecutar un proceso repetitivo, pero con valores de entrada cambiantes, se puede reutilizar el contrato cliente, reduciéndose enormemente el costo por petición.
- Debido a la infalibilidad e instantaneidad de la comprobación de resultados, no es necesario que la lógica de tanto el cálculo como la comprobación de resultados, sea 100% determinista. Esto le da más flexibilidad al protocolo, y permite que la función de comprobación de resultados sea de tipo “view”.

**Cons:**

- El protocolo solo es aplicable a un determinado tipo de problemas (aquellos de fácil comprobación).
- Al igual que en Optimistic Execution, los ejecutores deben tener algún mecanismo de estimación de complejidad de las peticiones para poder decidir si es conveniente resolverlas o no.
- Cada cliente debe programar su propia función de comprobación de resultados, debido a que cada problema se corrobora de una manera distinta y específica.

## 5.4 Análisis comparativo

En esta sección, se realizará una comparación plena, tanto cualitativa (diferencias de alcance y aplicabilidad), y cuantitativa (relación complejidad/costo) de cada una de las tres soluciones propuestas.

Al comparar los resultados medidos en la experimentación de los tres casos, se observa como los protocolos Optimistic Execution y Zero Trust Rollup tienen un costo base y promedio significativamente menor a Open Chain Computing. Esto se debe a la simplicidad de los mecanismos de garantía de integridad de los dos primeros respecto al último, además del hecho que en Open Chain Computing se debe pagar por múltiples ejecutores.

Sin embargo, hay que tener en cuenta que para el caso de Optimistic Execution, el costo base crece linealmente con la complejidad del proceso a ejecutar ya que es necesario que toda la lógica sea desplegada onchain. Por el contrario, en el caso ZTR, dicho costo crece a una tasa de orden menor debido a la naturaleza de los problemas NP. Otra cosa a tener en cuenta es que el límite superior de complejidad en OE, para que pueda

funcionar el sistema de desafíos, es de 30.000.000 de gas, mientras que para ZTR, dicho límite solo aplica a la lógica de comprobación de resultados. Por último, en ZTR el punto de corte de rentabilidad es aproximadamente 10 veces menor que en OE. Por consiguiente, el último protocolo es superior al primero en eficiencia. Sin embargo, este solo puede ser aplicado a un grupo reducido de problemas, y no a sistemas de cómputo general.

	Optimistic Execution	Open Chain Computing	Zero Trust Rollups
Escala el procesamiento de la red	X	X	X
Definición de precios por mercado	X		X
No necesita estimación de complejidad		X	
Funciona sin contrato cliente		X	
Permite procesos criptográficamente asimétricos			X
Tipo de problemas que resuelven	Cálculos Deterministas	Cálculos Deterministas	Problemas NP
Stake de los ejecutores	Alto	Mediano	Bajo
Límite de complejidad	30.000.000 gas	Ilimitado	Ilimitado
Confiabilidad de los resultados	Alta	Depende del número de ejecutores	Infalible

TABLA VII: Cuadro comparativo de las tres soluciones propuestas

Por otro lado, OCC, si bien es el más caro de los tres, no posee limitación alguna respecto a la complejidad de los procesos a ejecutar, ni el tipo de problemas a resolver. Otra ventaja de OCC es que es el único de los tres con costo base constante, ya que la única lógica onchain para este protocolo, es la lógica de distribución de tareas y pagos. Este protocolo, es también el más flexible de los tres, debido a que permite que el cliente elija el entorno de ejecución en el que se llevaran a cabo los procesos.

Independientemente, es importante notar que, para todos los casos, una vez superado el punto de corte de complejidad (con un precio de gas virtual/poder de ejecución promedio), que en OE ronda por los 2.000.000 gas, en OCC por los 6.000.000 gas, y en ZTR por los 1.700.000 gas, en todos los casos es más barato que la alternativa onchain.

## 6 Conclusión

A la hora de realizar un procesamiento de datos, cada alternativa conlleva limitaciones y costos asociados. Mantener la infraestructura onprem tiene un costo de adquisición y manutención de los equipos, además de que, bajo ese paradigma, la responsabilidad de mantener la seguridad, integridad y disponibilidad de la información recae en el cliente. Por otro lado, el paradigma Cloud tiene el problema que toda la infraestructura es controlada por una entidad centralizada. Y luego, BOINC carece de un sistema económico para incentivar el uso comercial. Todos estos inconvenientes son suplidos por la alternativa blockchain. Sin embargo, el gran problema de esta última vía, radica en el costo de utilización, cuando se requiere procesar grandes volúmenes de datos. Luego de el análisis y experimentación respecto a los tres mecanismos propuestos para realizar procesamiento en blockchain de manera delegada, con el fin de reducir los costos de uso, se concluyó lo siguiente:

- En cuanto a la factibilidad técnica, luego de la experimentación, se concluye que las tres soluciones son técnicamente factibles, con ciertas limitaciones. Cada uno de los tres protocolos propuestos en este trabajo, se enfoca en casos de uso diferentes. Optimistic Execution puede aplicarse para casos generales (siempre dentro del cómputo determinista), pero con una complejidad limitada debido a la necesidad de que el código a ejecutarse esté onchain. Por otro lado, Open Chain Computing se adapta bien a cualquier caso general de gran complejidad, pero tiene un costo base elevado, volviéndolo ineficiente para casos de uso de complejidad simple. Y por último, los Zero Trust Rollups, que poseen el costo más barato de todos y no poseen límite de complejidad, está limitado a un conjunto aún más reducido de casos de uso, es decir, aquellos cuyos resultados puedan ser comprobados de una manera más sencilla que el método de obtención de los mismos.
- En cuanto al costo, efectivamente cada una de las tres soluciones resulta más barata que la ejecución directa onchain, siempre y cuando se superen los respectivos puntos de corte de rentabilidad.

- En cuanto a la comparación de costos con plataformas comerciales de procesamiento delegado, cualquiera de las soluciones propuestas requiere un gasto de al menos unos millones de gas por petición. Si se utilizan los precios tomados para los diferentes análisis a lo largo del trabajo, se obtiene que como mínimo habría un gasto de más de 100 USD, es decir, el doble del costo de rentar una máquina virtual por un mes en los proveedores cloud más usados [31]. Por consiguiente, se concluye que, bajo las condiciones actuales de la red ethereum, ninguna de las tres soluciones es viable a nivel mercado.

Como se mencionó en las secciones introductorias, la intención central del trabajo es probar las posibilidades de la tecnología de las blockchain EVM para el cómputo general descentralizado. El objetivo no es desarrollar un producto competitivo a nivel mercado hoy. Sin embargo, debido a que la oferta de computo ocioso es real, y la arquitectura de las blockchains es descentralizada y posee una naturaleza económica, esta tecnología es una plataforma potencial para el desarrollo de un sistema de cómputo descentralizado. Si bien las soluciones propuestas en este trabajo, implementadas sobre la red ethereum, no son capaces de competir con los proveedores de plataformas de cómputo delegado tradicionales (cloud computing), se espera que el surgimiento de nuevas blockchains menos restrictivas que Ethereum puedan hacer uso del poder de procesamiento desperdiciado (aludido en la sección de motivación del trabajo), saturando el mercado de manera que el exceso de oferta de cómputo pueda reducir el costo de ejecución. Cabe destacar que, dada la complejidad del desarrollo para dispositivos heterogéneos, el estudio de viabilidad de las soluciones que se proponen en este proyecto sobre dispositivos con capacidad de cómputo ociosa, excede el alcance del trabajo y puede ser tomado como propuesta de desarrollos futuros.

En definitiva, a la hora de comparar estos protocolos con las alternativas tradicionales de procesamiento de datos, el problema general en los tres casos es el mismo. Bajo el estándar de EVM actual, el costo de procesamiento es muy alto, y solo se pueden procesar ejecuciones deterministas. No obstante, estas limitaciones nacen de las reglas y parámetros de las EVM, y no están ligados a la arquitectura del protocolo en sí.

## 7 Futuras Discusiones

Los protocolos propuestos a lo largo de este trabajo hararon el camino para investigar varias cuestiones relacionadas a la intención de resolver problemas de cómputo general, utilizando una arquitectura descentralizada de ejecutores bajo un protocolo blockchain.

Algunos ejemplos de discusiones futuras son:

- La implementación de un algoritmo que resuelva un caso de uso real en alguno de los tres protocolos.
- La implementación de un lenguaje de programación con un entorno de ejecución que cumpla con los requerimientos necesarios para ser utilizado en Open Chain Computing.
- La implementación de un ecosistema blockchain que contemple casos de computo generales de manera directa, en vez de estar montado como aplicaciones (en segunda capa) sobre una red ya existente (en este caso Ethereum).
- La implementación de un software destinado a correr en cualquier plataforma (PCs, celulares, dispositivos IoT), con el objetivo de acoplarlos a un sistema de distribución y resolución de tareas descentralizado.

## 8 Referencias

- [1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh y A. Hung Byers, «Big data: The next frontier for innovation, competition, and productivity,» 1 Mayo 2011. [En línea]. Available: <https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/big-data-the-next-frontier-for-innovation>. [Último acceso: 14 Enero 2023].
- [2] P. Fry, Print Punch: Artefacts From The Punch Card Era, Patrick Fry, 2019.
- [3] CoinMarketCap, «Today's Cryptocurrency Prices by Market Cap,» 2023. [En línea]. Available: <https://coinmarketcap.com/>.
- [4] E. Krithigashree, «Impact of gadgets in our life,» 30 Enero 2022. [En línea]. Available: <https://timesofindia.indiatimes.com/readersblog/silvercascade/impact-of-gadgets-in-our-life-40978/>.
- [5] N. Carr, The Big Switch: Rewiring the World, from Edison to Google, W. W. Norton & Company, 2008.
- [6] A. M. A. K. A. S. Trieu C. Chieu, «Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment,» *IEEE*, 2009.
- [7] R. Gellman, «Privacy in the Clouds: Risks to Privacy and Confidentiality from Cloud Computing,» World Privacy Forum, 2009.
- [8] Berkeley, «Volunteer Computing,» 15 Septiembre 2018. [En línea]. Available: <https://boinc.berkeley.edu/trac/wiki/VolunteerComputing>.
- [9] I. Parravicini, «Análisis comparativo de desempeño de plataformas distribuidas GRID y CLOUD para codificación de video bajo demanda,» Pontificia Universidad Católica Argentina, 2013.
- [10] J. K. David Anderson, «An Incentive System for Volunteer Computing,» 16 Abril 2007. [En línea]. Available: [https://boinc.berkeley.edu/boinc\\_papers/credit/text.php](https://boinc.berkeley.edu/boinc_papers/credit/text.php).
- [11] K. Bheemaiah, «Block Chain 2.0: The Renaissance of Money,» *Wired*, 2015.

- [12] V. Buterin, «Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform,» 2014.
- [13] TheEconomist, «Blockchains: The great chain of being sure about things,» *The Economist*, 2015.
- [14] D. Newman, «What Are Blockchain RPC Nodes and How Do They Work,» 18 Mayo 2023. [En línea]. Available: <https://beincrypto.com/learn/blockchain-rpc-nodes/>.
- [15] Baeldung, «Encryption: ECDSA vs. RSA Keys,» 20 Septiembre 2023. [En línea]. Available: [https://www.baeldung.com/cs/encryption-asymmetric-algorithms#:~:text=ECDSA%20\(Elliptic%20Curve%20Digital%20Signature,two%20pseudo%2Drandom%20prime%20numbers..](https://www.baeldung.com/cs/encryption-asymmetric-algorithms#:~:text=ECDSA%20(Elliptic%20Curve%20Digital%20Signature,two%20pseudo%2Drandom%20prime%20numbers..)
- [16] Infury, «Understanding Blockchain Layers: A Comprehensive Guide,» 8 Junio 2023. [En línea]. Available: <https://medium.com/@infuyIT/understanding-blockchain-layers-a-comprehensive-guide-27b6d184edeb>.
- [17] N. Lenga, «The Different Categories of Consensus Algorithms,» 2023 Febrero 6. [En línea]. Available: [https://zerocap.com/insights/research-lab/categories-of-consensus-algorithms/#:~:text=not%20be%20possible.-,Proof%20of%20Work%20\(PoW\),the%20total%20cryptocurrency%20market%20capitalisation..](https://zerocap.com/insights/research-lab/categories-of-consensus-algorithms/#:~:text=not%20be%20possible.-,Proof%20of%20Work%20(PoW),the%20total%20cryptocurrency%20market%20capitalisation..)
- [18] Binance, «Proof of Work (PoW) vs. Proof of Stake (PoS),» 12 Diciembre 2018. [En línea]. Available: <https://academy.binance.com/en/articles/proof-of-work-vs-proof-of-stake>.
- [19] T. Von Harz, «The Effects of the Ethereum Merge on GPU Prices,» 31 Julio 2023. [En línea]. Available: <https://history-computer.com/the-effects-of-the-ethereum-merge-on-gpu-prices/>.
- [20] MorpheusLabs, «Decentralized Applications (DApps) Explained,» 5 Septiembre 2018. [En línea]. Available: <https://medium.com/morpheus-labs/decentralized-applications-dapps-explained-5a67c6763ffa>.

- [21] V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden y A. Bakhta, «EIP-1559: Fee market change for ETH 1.0 chain,» 13 Abril 2019. [En línea]. Available: <https://eips.ethereum.org/EIPS/eip-1559>.
- [22] BlockchainCouncil, «Ethereum Rollups: A Comprehensive Guide,» 30 Junio 2022. [En línea]. Available: [https://www.blockchain-council.org/ethereum/ethereum-rollups/?gad\\_source=1&gclid=Cj0KCQiA3uGqBhDdARIsAFeJ5r02ScS62o55opKTQJO5iQMP8bLzN-o3SMR9dzwOLBbu3TKXe9KfJiAaAiAnEALw\\_wcB](https://www.blockchain-council.org/ethereum/ethereum-rollups/?gad_source=1&gclid=Cj0KCQiA3uGqBhDdARIsAFeJ5r02ScS62o55opKTQJO5iQMP8bLzN-o3SMR9dzwOLBbu3TKXe9KfJiAaAiAnEALw_wcB).
- [23] 0xMimir, «OPTIMISTIC ROLLUPS,» 15 Octubre 2023. [En línea]. Available: <https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/#:~:text=An%20optimistic%20rollup%20is%20an,data%20to%20Mainnet%20as%20calldata%20..>
- [24] J. Thaler, Proofs, Arguments, and Zero-Knowledge, Georgetown: Georgetown University, 2023.
- [25] Sheinix, «Zero Knowledge Proofs,» 31 Enero 2018. [En línea]. Available: <https://medium.com/coinmonks/zero-knowledge-proofs-14bb012c1ce9>.
- [26] Surfnomada, «zk-STARKs vs zk-SNARKs: Diferencias en las tecnologías de conocimiento cero,» 14 Septiembre 2022. [En línea]. Available: <https://medium.com/@surfnomada/zk-starks-vs-zk-snarks-diferencias-en-las-tecnolog%C3%ADAs-de-conocimiento-cero-2126079139e6#:~:text=As%C3%AD%2C%20los%20zk%2DSNARKs%20son,enfoque%20resistente%20a%20las%20colisiones..>
- [27] Karolina, «Arbitrum vs Optimism,» 29 Septiembre 2023. [En línea]. Available: <https://nextrope.com/arbitrum-vs-optimism/#:~:text=Fundamentally%2C%20Optimism%20implements%20single%20round,longer%20but%20are%20less%20expensive..>
- [28] QuickNode, «A Broad Overview of Reentrancy Attacks in Solidity Contracts,» 3 Julio 2023. [En línea]. Available: <https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-broad-overview-of-reentrancy-attacks-in-solidity-contracts>.

- [29] M. Ratcliffe, «GitHub - Zero-Trust-Rollups,» 10 Enero 2023. [En línea]. Available: <https://github.com/matiasratcliffe/Zero-Trust-Rollups>.
- [30] CoinTelegraph, «What is Polygon? MATIC explained,» 2023. [En línea]. Available: <https://cointelegraph.com/learn/polygon-blockchain-explained-a-beginners-guide-to-matic>.
- [31] D. Kechagias, «Top 10 Cloud Provider Comparison 2023: VM Performance / Price,» [En línea]. Available: <https://dev.to/dkechag/cloud-vm-performance-value-comparison-2023-perl-more-1kpp>.

## 9 Anexos

### Anexo I

```
@Logger.LogClassMethods()
class AESCipher(object):

    def __init__(self, passcode: int):
        self.bs = AES.block_size
        self.key = hashlib.sha256(passcode.hex().encode()).digest()

    def encryptFile(self, fileName):
        with open(fileName, "r") as f:
            fileContents = f.read()
        return self.encrypt(fileContents)

    def encrypt(self, raw):
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(self._pad(raw.encode())))

    def decryptFile(self, fileName):
        with open(fileName, "r") as f:
            fileContents = f.read()
        return self.decrypt(fileContents)

    def decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return self._unpad(cipher.decrypt(enc[AES.block_size:])).decode('utf-8')

    def _pad(self, s):
        return s + (self.bs - len(s) % self.bs) * bytes(0x01)

    def _unpad(self, s):
        for i in range(1, len(s) - 1):
            if s[-i] != 0:
                break
        i -= 1
        return s[:-i] if i > 0 else s
```

ANEXO I: código en python para encriptar un texto a partir de una clave con AESCipher.

## Anexo II

```
@Logger.LogClassMethods()
class TS3000Requestor:

    def __init__(self, broker, account, textFileToEncrypt, paymentPerFragment: int = 1e12,
                 numberOfKeyFragments: int = 50, difficulty: int = 10, timeFramePerFragment = 0, gas_price=0):
        self.broker = broker
        self.owner = account
        firstLocalHash, globalHashes, passcode = self._generateKeyFragments(numberOfKeyFragments, difficulty)
        self.cipher = AESCipher(passcode)
        with open(textFileToEncrypt + ".encrypted", "w") as f:
            f.write(str(self.cipher.encryptFile(textFileToEncrypt))[2:-1])
        self.client = TS3000.deploy(broker.address, textFileToEncrypt + ".encrypted", firstLocalHash, globalHashes,
                                    timeFramePerFragment, difficulty, {'from': account, 'value': paymentPerFragment, "gas_price": gas_price})

    def _generateKeyFragments(self, size: int, difficulty: int):
        globalHashes = []
        localHashes = [keccak(encode(['uint'], [randint(0, int('9' * difficulty))]))]
        for i in range(size):
            passcode = randint(0, int('9' * difficulty))
            globalHashes.append(keccak(encode(['uint', 'bytes32'], [passcode, localHashes[i]])))
            if i < size - 1:
                localHashes.append(keccak(encode(['uint'], [passcode])))
        return localHashes[0], globalHashes, keccak(encode(['uint'], [passcode]))

    def getInitialRequestID(self):
        return self.client.tx.events["requestCreated"]["requestID"]
```

*ANEXO II: código en python para generar una clave, encryptar un texto con ella, y publicar los fragmentos de dicha clave en la blockchain.*

## Anexo III

```
constructor(address brokerAddress, string memory _encryptedDataRefference, bytes32 firstLocalHash,
           bytes32[] memory globalHashes, uint _minTimeFramePerFragment, uint _fragmentDifficulty)
) BaseClient(brokerAddress) payable {
    postProcessingGas = 400000;
    postProcessingEnabled = true;
    fragmentDifficulty = _fragmentDifficulty;
    rewardPerFragment = msg.value / globalHashes.length;
    encryptedDataRefference = _encryptedDataRefference;
    minTimeFramePerFragment = _minTimeFramePerFragment;
    for (uint i = 0; i < globalHashes.length; i++) {
        KeyFragment memory fragment;
        fragment.globalHash = globalHashes[i];
        if (i == 0) {
            fragment.localHash = firstLocalHash;
        }
        keyFragments.push(fragment);
    }
    Input memory input = Input({
        fragmentIndex: 0,
        globalHash: globalHashes[0],
        localHash: firstLocalHash,
        minTimestamp: block.timestamp
    });
    _submitRequest(rewardPerFragment, abi.encode(input), postProcessingGas);
}
```

*ANEXO III: código en solidity para construir los fragmentos necesarios de una clave particionada al instanciarse el contrato*

## Anexo IV

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation. We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.--That to secure these rights, Governments are instituted among Men, deriving their just powers from the consent of the governed, -- That whenever any Form of Government becomes destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient causes; and accordingly all experience hath shewn, that mankind are more disposed to suffer, while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed. But when a long train of abuses and usurpations, pursuing invariably the same Object evinces a design to reduce them under absolute Despotism, it is their right, it is their duty, to throw off such Government, and to provide new Guards for their future security.--Such has been the patient sufferance of these Colonies; and such is now the necessity which constrains them to alter their former Systems of Government. The history of the present King of Great Britain is a history of repeated injuries and usurpations, all having in direct object the establishment of an absolute Tyranny over these States. To prove this, let Facts be submitted to a candid world.

ANEXO IV: texto extraído de la declaración de independencia de EEUU para probar con el cifrado y descifrado del protocolo TS3000.

## Anexo V

PlwnoCYfiXPAwCgaUgpEepJ7Qk+sVoEoP9dgxGbtooMvjQsgdv4nPgTXDFSGPoelA1h  
SntsIhUrpv3e5KMQECLrvAJ+qoQDpsz49WQieul5JY8Eb1Ai3uu7Wi5vXzPkWMXwdT0  
hBXX7hLbQY2A9t6BLbc0kH/MahTQdSBG0gJVI1VHF4x+IntuoKO/gQtGt/ZXK5pUQZT  
yY+M+FZAs6S0q8ZxjYSQUnlhlimjoze1kWebRs7XlpZKRPN1T7hXYL9wF3pxHkheEVA  
m1ng7ELeZT/TfjH4wsTvd/e0cpdf8Jhd81J2rtfySyDgGb0cKQyqrBoTeskqlm11ba1Ai5Fly  
TdeoNKA15z478JOwYoS3zE8BjuzLtk4XgWGom/HnNgzhHBjxIKm5/252MDh80iO5QX  
XI1lw6j+X5uqSPGVdmJevo5bzU6JlIFWRrauC/qR+pqjB6GO9D38WEqNrmSGxz4dwc  
oEjbYK1bBAa+AOSsJ5M2ZV6RPY6idgls37iNTFRtASxbB1iNQkeXF+Ze9ZjcsiEnBmPe  
aiKd9z8bXiMhxOGbdPPcp03fmZwwXXLTaQRMYZRBSdOtJxjlcjPa8bRCn3Juu+xnXRs  
zM2AkdnrcuwrA3bMfU1Dudkjy4Wb3sAjSL4/3SqjDuOHvnrtAbutss8I1RjrZP9oMjSOcVU  
2/z8IKgTnJ8SRxIVNCkAMnT/WOOObJa+YZ/AcFd3LUjXzvbPHBZErlJotKDPyzAw4DQz+  
noBMfWXW211BAiH5Z9uDiMcRml+41nxxz9hJJmVI8Xqlq5NNDGfCSW+LmOz08z83v  
FEXP9eNLrszMVXBj5+mh2+wWdVAy4T+h7iGM4SyajqymANsozyuO0Aa3Av1mQoeg  
R80xkDiYLJsp9b8i7Sc2yvGXsf7aSMAWhfDil+42yFNdL6a7kl92NezyJRJd8Wt6Wwf+0  
m4jcXwynPd+hq5VpKCXxNPoSOMTr2h6eabJr/7QnXRsf1djfKJ+msbMOyWDQ2Zon  
Pfx7KijowJoOB7HtyrWXdiNcZpli0am4K+Z3tdnUJY0nZ9kEDnD8TW4MzFJP20f1qkLtJE  
hJ6DAIJpY0YtVQlcvfJhJWJkilX7fJwa8R6YYIp9wQhNY6eKmFWOPL7Sbr/AkzhJjb/kC  
Mrdv8bu+VQ8dJ2f6rZwSWKVHaxsReGG8hITomononBNaeNGENuts/ErSmYpvxf2vgJ  
NLgPLpFAePtQ+buLeOvh1Nlj+xhuoj4OCKkb6JZDCCx+0AR5UALFH6ONz6KEUiixmo  
wJz9OVcyCXCWMJ62MulxzwjTowp2V/ODNwvFQZqlrkmpzjfQhc6NmSkiva5qicF+x2an  
L5AziWVK/Af2aybRlpXKwbX5rMmdqBOJATy6o4LuaYnDuMPooFhnF89RHCvvy5J/Eo  
ouyCWXPP2QntRzG/Rp5nhHem/nX3K8tyRrJCJdgJGqbSuHJ7puBXFGgvogC0IMDjVs  
55qaEYDBd89+d3Jlz1y2u1GtwpPj5JyJRA3DVyWkRHh0SpQcB5b82KejkPVPezP/BFzl  
R5hrgh+LY61jQkgsloyq1mYA+hru4jleUaZp93QtZpCNIKEiaEG6Ez+PhTiyyH6Y6SA5U7  
XhUxgmHAfU6UyjTxHRoYDLr32+LX817DY0mQUgbImBnnrWteLFjl08DCspdRNhGq9D  
nGMYcwQdHA+FeH+e5pHw2W4a6qKHxv8Ry9RN8bYiAyfUgdAdj7HHeRMwUvebDwA  
5f+IZZz1q+pId46l0yX6Yzj+pIHjt9norfSVK3jF4eJmUQz4HWUsZvMj7W4nqd8Ad9HKXw  
xjejuRckdjHLL6KeJrLhwIJKeHMvbsrlqHHg2gk0SHux0Clvl8ps5MeKioqRkHTQApO1oW  
xAYqooDjKXBEMIz2t0dAn98TYwZ2I5jsQADnahT8M8hEm/YzcGWk3DIUbbTZiZXWQr  
HzKUd/2sf+qxn93C/keASZfp7rRHdqnrBBnnwfgZchhyMtVYBcpT1s6SfcWy1GygKj5xe

kXmsRsI+Q5efqO+vQyQ6kQtILyEDEhT3Hsl/lyk8ZALqx2jT/wNt9YwueLR50YAWUZbSmGezeMRFVwuMtGa+U09MQRTrfPhy32ETo8HP8hjY/8T64StznLRVbp6eUxhKZ6Ix6D8n7j045rq9bw3asTvkIZ/T0597JhLbj2JtAFvFXtpctMehNFLvCft2VUwJ3YSr6VyaqclpIVYo+JOIG6J0H57vgZn073r5Mcup0WWmBHHZfZ3KaFYUny5wpsBQ3mw2AN9qy0jsfVYgEQUVTYFJlZ5XkPug/tUtXO2FtZoevaqa0wFMFSgO+XzCRaPT1RZCDRE03j1XQHBZp00u5uYnK35sUSCwKq2CuaGederkb8sEBIHrxu/0/NYWoda1kYpWPOF6xNqixerMyXxmPJ BsM6uJzkF96oGXEG5QI3liua/+HFyU99M1E9PjeR2rCSVsBIQ+dZz5laH0LbI7NG8phr/hdywU/50jHbB726L7NQMGRawEa4Sp0y1ZOLQ3B+0VellVRs85dcKzEB9FLSsnpvCtFPkxtqyT94DIJGEfydf7qQjlyHSNCflLT15opbLHkG1bdiMXdac9uMPsiXeLYuoxWB2Fok6FUDs96FxwVDfwrhw5/ogoQESgTBcD3RMA6s9m9yvuiNmkenUFe7jjhR4Sy6csDsixYbnzzEOG0WNntVNr5i9HIdCriGeGrA5w94/lfGkVcMN3vLnouOsSg

ANEXO V: versión encriptada del texto del Anexo IV, utilizando AESCipher con la clave

[0xa6c825f473ff3ad15e48ccd3e5c35d6b1e6b3ae7623a44ed3bcf3e1c4d4054ac]