ethereum mechanics

Created	@October 8, 2021 10:30 AM
Created By	
Last Edited By	
Last Edited Time	@October 13, 2021 4:32 PM
Stakeholders	
Status	In Review
Type	Translate

Preethi Kasireddy

How does Ethereum work, anyway?READ STORY

Written On

September 13, 2017

in

¿Cómo funciona Ethereum?

Blockchain

Odds are you've heard about the Ethereum blockchain, whether or not you know what it is. It's been in the news a lot lately, including the cover of some major magazines, but reading those articles can be like gibberish if you don't have a foundation for what exactly Ethereum is. So what is it? In essence, a public database that keeps a permanent record of digital transactions. Importantly, this database doesn't require any central authority to maintain and secure it. Instead it operates as a "trustless" transactional system—a framework in which individuals can make peer-to-peer transactions without needing to trust a third party OR one another.

Still confused? That's where this post comes in. My aim is to explain how Ethereum functions at a technical level, without complex math or scary-looking formulas. Even if you're not a programmer, I hope you'll walk away with at least better grasp of the tech. If some parts are too technical and difficult to grok, that's totally fine! There's really no need to understand every little detail. I recommend just focusing on understanding things at a broad level.

Many of the topics covered in this post are a breakdown of the concepts discussed in the yellow paper. I've added my own explanations and diagrams to make understanding Ethereum easier. Those brave enough to take on the technical challenge can also read the Ethereum yellow paper.

Let's get started!

PS: If you are a developer interested in learning how to build Dapps on Ethereum, join my NEW cohort-based course.

*Cadena de bloques

Lo más probable es que hayas oído hablar de la cadena de bloques Ethereum, sepas o no lo que es. Últimamente ha salido mucho en las noticias, incluso en la portada de algunas revistas importantes, pero leer esos artículos puede parecer un galimatías si no se tiene una base de lo que es exactamente Ethereum. ¿Y qué es? En esencia, una base de datos pública que mantiene un registro permanente de las transacciones digitales. Lo más importante es que esta base de datos no requiere ninguna autoridad central para mantenerla y asegurarla. En su lugar, funciona como un sistema transaccional "sin confianza", un marco en el que las personas pueden realizar transacciones entre pares sin necesidad de confiar en un tercero o en los demás.

¿Todavía estás confundido? Ahí es donde entra este post. Mi objetivo es explicar cómo funciona Ethereum a un nivel técnico, sin matemáticas complejas ni fórmulas de aspecto aterrador. Incluso si no eres un programador, espero que te vayas con al menos una mejor comprensión de la tecnología. Si algunas partes son demasiado técnicas y difíciles de entender, ¡no pasa nada! No es necesario entender todos los detalles. Te recomiendo que te centres en comprender las cosas a un nivel amplio.

Muchos de los temas tratados en esta entrada son un desglose de los conceptos tratados en el documento amarillo. He añadido mis propias explicaciones y diagramas para facilitar la comprensión de Ethereum. Aquellos que sean lo suficientemente valientes como para asumir el reto técnico también pueden leer el libro amarillo de Ethereum.

¡Empecemos!

PD: Si eres un desarrollador interesado en aprender a construir Dapps en Ethereum, únete a mi NUEVO curso basado en cohortes. NEW cohort-based course.

Blockchain definition

A blockchain is a “**cryptographically secure transactional singleton machine with shared-state.”** [1] That’s a mouthful, isn’t it? Let’s break it down.

- **“Cryptographically secure”** means that the creation of digital currency is secured by complex mathematical algorithms that are obscenely hard to break. Think of a firewall of sorts. They make it nearly impossible to cheat the system (e.g. create fake transactions, erase transactions, etc.)
- **“Transactional singleton machine”** means that there’s a single canonical instance of the machine responsible for all the transactions being created in the system. In other words, there’s a single global truth that everyone believes in.
- **“With shared-state”** means that the state stored on this machine is shared and open to everyone.

Ethereum implements this blockchain paradigm.

Definicion de cadena de bloques (blockchain)

Un blockchain es una "**máquina transaccional criptográficamente segura con estado compartido"** Eso es un bocado, ¿no? Vamos a desglosarlo.

- **"Criptográficamente seguro "** significa que la creación de la moneda digital está asegurada por complejos algoritmos matemáticos que son obscenamente difíciles de romper. Piensa en una especie de cortafuegos. Hacen que sea casi imposible engañar al sistema (por ejemplo, crear transacciones falsas, borrar transacciones, etc.)
- **"Máquina transaccional única"** significa que hay una única instancia canónica de la máquina responsable de todas las transacciones que se crean en el sistema. En otras palabras, hay una única verdad global en la que todos creen.
- "**Con estado compartido**" significa que el estado almacenado en esta máquina es compartido y abierto a todos.

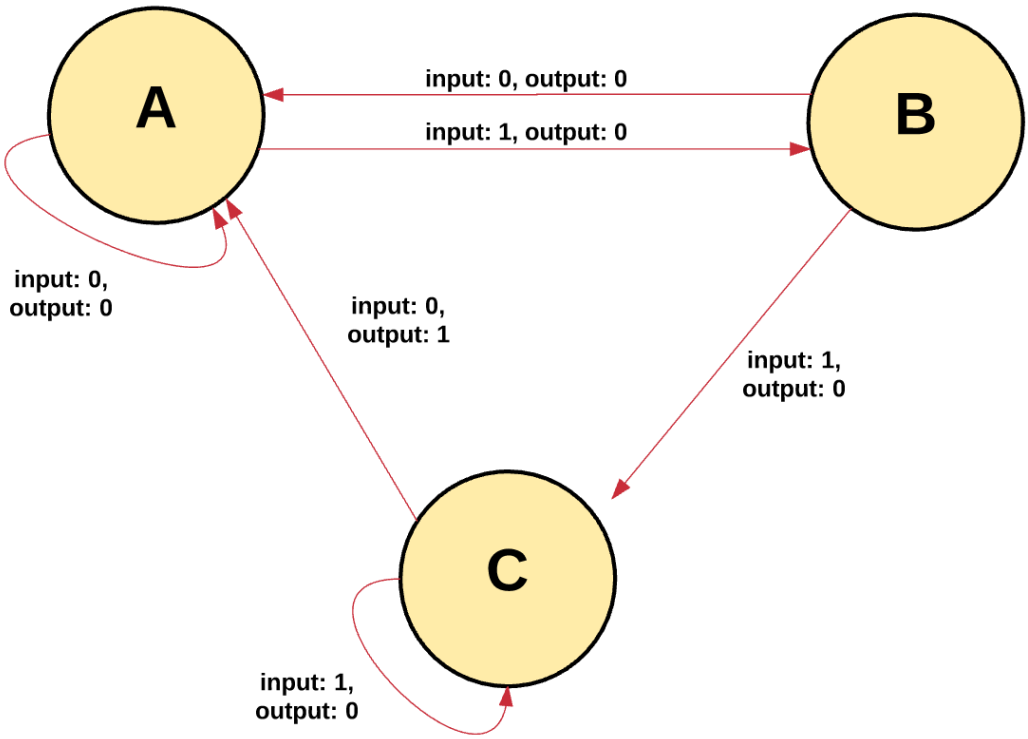
Ethereum implementa este paradigma de blockchain.

The Ethereum blockchain paradigm explained

The Ethereum blockchain is essentially a **transaction-based state machine**. In computer science, a *state machine* refers to something that will read a series of inputs and, based on those inputs, will transition to a new state.

El paradigma de la cadena de bloques de Ethereum explicado

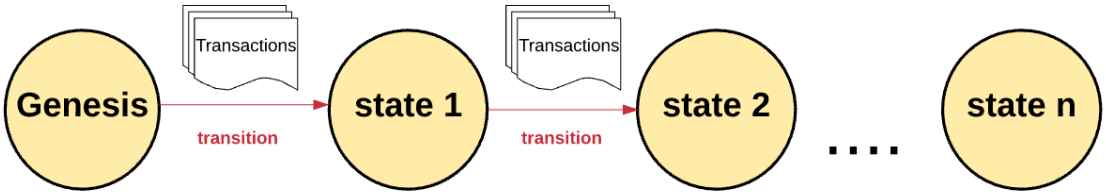
El blockchain de Ethereum es esencialmente una **máquina de estados** basada en transacciones. En informática, una *máquina de estados* se refiere a algo que leerá una serie de entradas y, basándose en esas entradas, hará la transición a un nuevo estado.



With Ethereum's state machine, we begin with a "genesis state." This is analogous to a blank slate, before any transactions have happened on the network. When

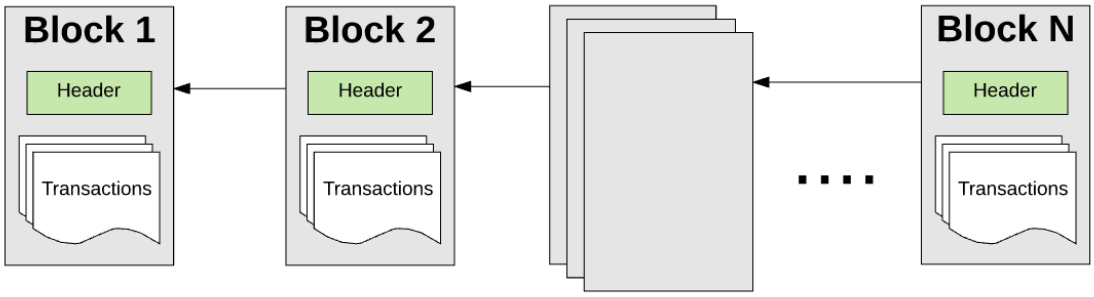
transactions are executed, this genesis state transitions into some final state. At any point in time, this final state represents the current state of Ethereum.

Con la máquina de estados de Ethereum, empezamos con un "estado de génesis". Esto es análogo a una pizarra en blanco, antes de que se haya producido ninguna transacción en la red. Cuando se ejecutan las transacciones, este estado de génesis pasa a un estado final. En cualquier momento, este estado final representa el estado actual de Ethereum.



The state of Ethereum has millions of transactions. These transactions are grouped into “blocks.” A block contains a series of transactions, and each block is chained together with its previous block.

El estado de Ethereum tiene millones de transacciones. Estas transacciones se agrupan en "bloques". Un bloque contiene una serie de transacciones, y cada bloque se encadena con su bloque anterior.



To cause a transition from one state to the next, a transaction must be valid. **For a transaction to be considered valid, it must go through a validation process known as mining.** Mining is when a group of nodes (i.e. computers) expend their compute resources to create a block of valid transactions.

Any node on the network that declares itself as a miner can attempt to create and validate a block. Lots of miners from around the world try to create and validate blocks at the same time. Each miner provides a mathematical “proof” when submitting a block to the blockchain, and this proof acts as a guarantee: if the proof exists, the block must be valid.

For a block to be added to the main blockchain, the miner must prove it faster than any other competitor miner. The process of validating each block by having a miner provide a mathematical proof is known as a **“proof of work.”**

A miner who validates a new block is rewarded with a certain amount of value for doing this work. What is that value? The Ethereum blockchain uses an intrinsic digital token called “Ether.” Every time a miner proves a block, new Ether tokens are generated and awarded.

You might wonder: what guarantees that everyone sticks to one chain of blocks? How can we be sure that there doesn’t exist a subset of miners who will decide to create their own chain of blocks?

Earlier, we defined a blockchain as a **transactional singleton machine with shared-state**. Using this definition, we can understand the correct current state is a single global truth, which everyone must accept. Having multiple states (or chains) would ruin the whole system, because it would be impossible to agree on which state was the correct one. If the chains were to diverge, you might own 10 coins on one chain, 20 on another, and 40 on another. In this scenario, there would be no way to determine which chain was the most “valid.”

Whenever multiple paths are generated, a “fork” occurs. We typically want to avoid forks, because they disrupt the system and force people to choose which chain they “believe” in.

Para pasar de un estado a otro, una transacción debe ser válida. **Para que una transacción se considere válida, debe pasar por un proceso de validación conocido como minería.** La minería se produce cuando un grupo de nodos (es decir, ordenadores) gastan sus recursos de computación para crear un bloque de transacciones válidas.

Cualquier nodo de la red que se declare minero puede intentar crear y validar un bloque. Muchos mineros de todo el mundo intentan crear y validar bloques al mismo tiempo. Cada minero proporciona una "prueba" matemática cuando envía un bloque a la cadena de bloques, y esta prueba actúa como garantía: si la prueba existe, el bloque debe ser válido.

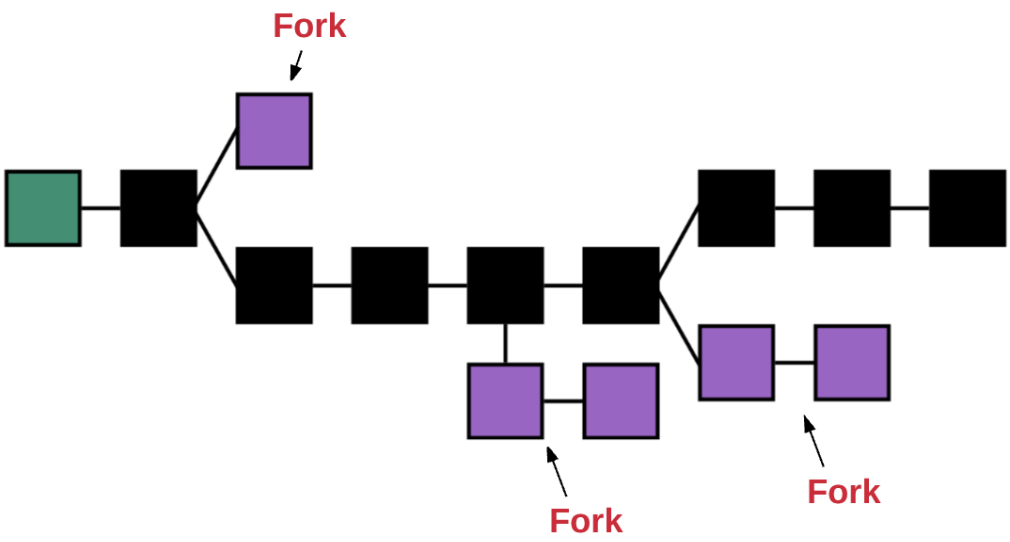
Para que un bloque se añada a la cadena de bloques principal, el minero debe probarlo más rápido que cualquier otro minero de la competencia. El proceso de validación de cada bloque mediante la presentación de una prueba matemática por parte de un minero se conoce como **"prueba de trabajo "**.

Un minero que valida un nuevo bloque es recompensado con una cierta cantidad de valor por hacer este trabajo. ¿Cuál es ese valor? La blockchain de Ethereum utiliza un token digital intrínseco llamado "Ether". Cada vez que un minero prueba un bloque, se generan y se otorgan nuevos tokens de Ether.

Te preguntarás: ¿qué garantiza que todo el mundo se ciña a una cadena de bloques? ¿Cómo podemos estar seguros de que no existe un subconjunto de mineros que decidan crear su propia cadena de bloques?

Anteriormente, definimos una cadena de bloques como una **máquina transaccional única con estado compartido**. Usando esta definición, podemos entender que el estado actual correcto es una única verdad global, que todos deben aceptar. Tener múltiples estados (o cadenas) arruinaría todo el sistema, porque sería imposible ponerse de acuerdo sobre qué estado es el correcto. Si las cadenas fueran divergentes, podrías tener 10 monedas en una cadena, 20 en otra y 40 en otra. En este escenario, no habría forma de determinar qué cadena es la más "válida".

Siempre que se generan múltiples rutas, se produce una "bifurcación". Normalmente queremos evitar las bifurcaciones, ya que perturban el sistema y obligan a la gente a elegir en qué cadena "creen".



To determine which path is most valid and prevent multiple chains, Ethereum uses a mechanism called the **“GHOST protocol.”**

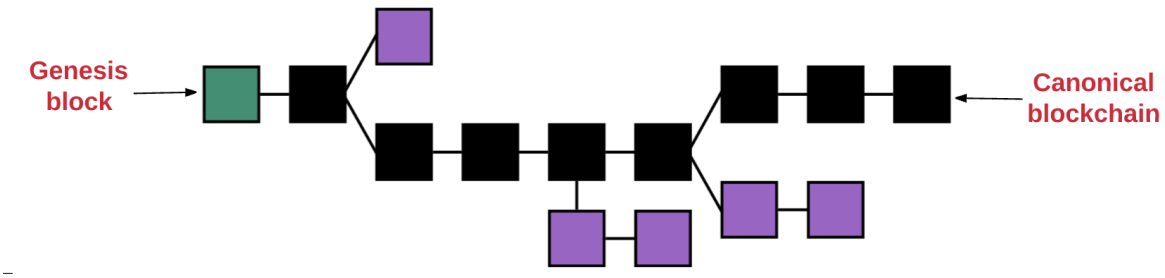
“GHOST” = “Greedy Heaviest Observed Subtree”

In simple terms, **the GHOST protocol says we must pick the path that has had the most computation done upon it**. One way to determine that path is to use the block number of the most recent block (the “leaf block”), which represents the total number of blocks in the current path (not counting the genesis block). The higher the block number, the longer the path and the greater the mining effort that must have gone into arriving at the leaf. Using this reasoning allows us to agree on the canonical version of the current state.

Para determinar qué camino es el más válido y evitar las cadenas múltiples, Ethereum utiliza un mecanismo llamado **"protocolo GHOST "**.

"GHOST" = "Greedy Heaviest Observed Subtree "

En términos simples, **el protocolo GHOST dice que debemos elegir el camino que ha tenido la mayor cantidad de cálculos realizados sobre él**. Una forma de determinar ese camino es utilizar el número de bloque del bloque más reciente (el "bloque hoja"), que representa el número total de bloques en el camino actual (sin contar el bloque génesis). Cuanto más alto sea el número de bloque, más larga será la ruta y mayor será el esfuerzo de extracción que se habrá realizado para llegar a la hoja. El uso de este razonamiento nos permite acordar la versión canónica del estado actual.



Now that you’ve gotten the 10,000-foot overview of what a blockchain is, let’s dive deeper into the main components that the Ethereum system is comprised of:

- accounts

- state
- gas and fees
- transactions
- blocks
- transaction execution
- mining
- proof of work

One note before getting started: whenever I say “hash” of X, I am referring to the KECCAK-256 hash, which Ethereum uses.

Ahora que ya tienes una visión general de lo que es una cadena de bloques, vamos a profundizar en los principales componentes del sistema Ethereum:

- cuentas
- estado
- gas y tarifas
- transacciones
- bloques
- ejecución de transacciones
- minería
- prueba de trabajo

Una nota antes de empezar: siempre que digo "hash" de X, me refiero al KECCAK-256 hash, que utiliza Ethereum.

Accounts

The global “shared-state” of Ethereum is comprised of many small objects (“accounts”) that are able to interact with one another through a message-passing framework. Each account has a **state** associated with it and a 20-byte **address**. An address in Ethereum is a 160-bit identifier that is used to identify any account.

There are two types of accounts:

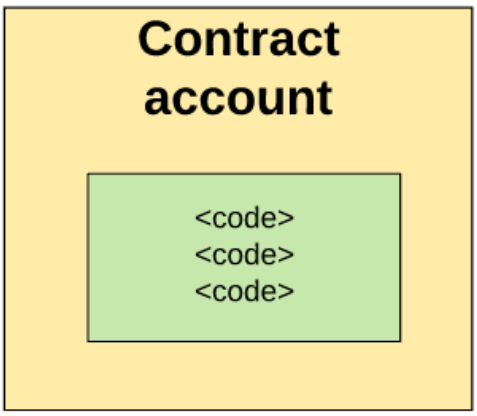
- Externally owned accounts, which are **controlled by private keys** and have **no code associated with them.**
- Contract accounts, which are **controlled by their contract code** and **have code associated with them.**

Cuentas

El "estado compartido" global de Ethereum se compone de muchos objetos pequeños ("cuentas") que pueden interactuar entre sí a través de un marco de paso de mensajes. Cada cuenta tiene un **estado** asociado y una **dirección** de 20 bytes. Una dirección en Ethereum es un identificador de 160 bits que se utiliza para identificar cualquier cuenta.

Hay dos tipos de cuentas:

- Cuentas de propiedad externa, que son **controladas por claves privadas** y no tienen **ningún código asociado**.
- Las cuentas de contrato, que están **controladas por su código de contrato** y **tienen un código asociado a ellas.**



Externally owned accounts vs. contract accounts

It’s important to understand a fundamental difference between externally owned accounts and contract accounts. **An externally owned account can send messages to other externally owned accounts OR to other contract accounts**

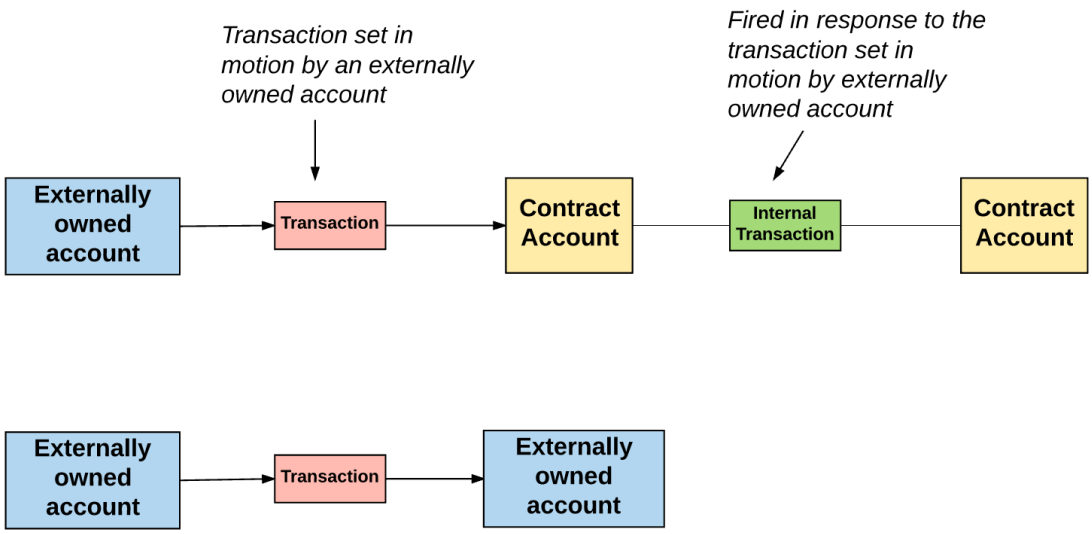
by creating and signing a transaction using its private key. A message between two externally owned accounts is simply a value transfer. But a message from an externally owned account to a contract account activates the contract account's code, allowing it to perform various actions (e.g. transfer tokens, write to internal storage, mint new tokens, perform some calculation, create new contracts, etc.).

Unlike externally owned accounts, contract accounts can't initiate new transactions on their own. Instead, contract accounts can only fire transactions in response to other transactions they have received (from an externally owned account or from another contract account). We'll learn more about contract-to-contract calls in the "*Transactions and Messages*" section.

Cuentas de propiedad externa vs. cuentas contractuales

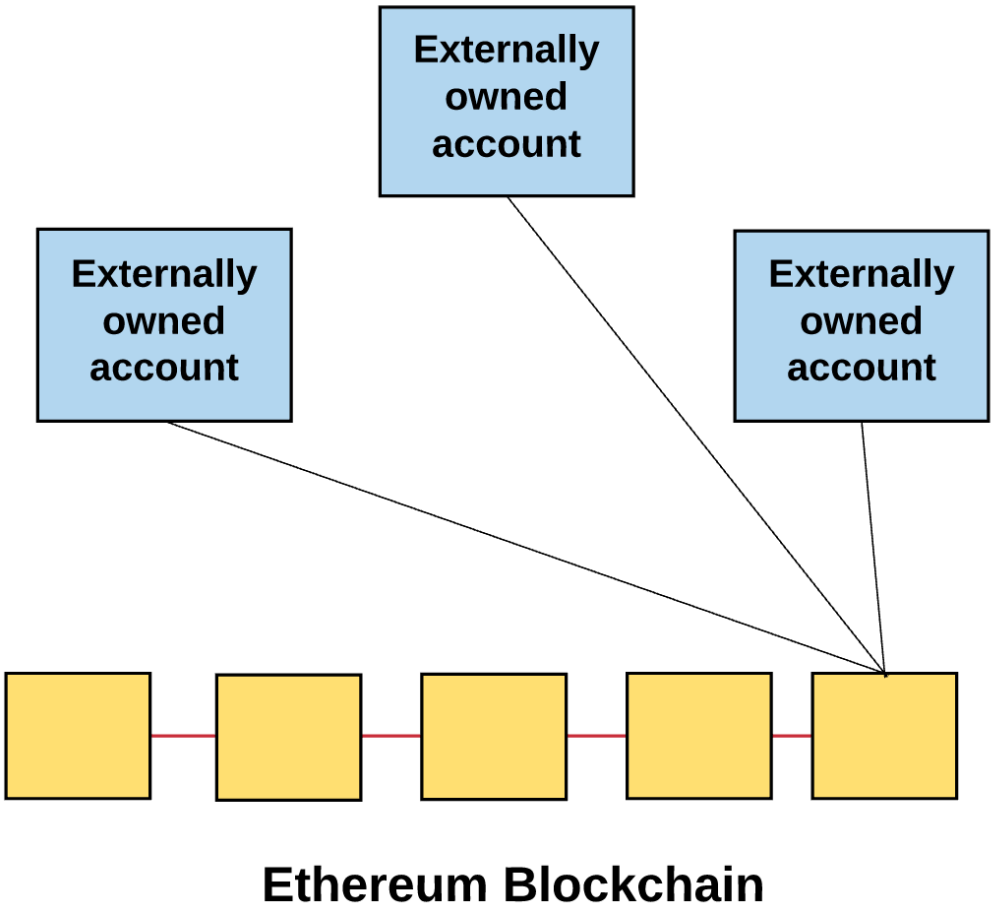
Es importante entender una diferencia fundamental entre las cuentas de propiedad externa y las cuentas de contrato. **Una cuenta de propiedad externa puede enviar mensajes a otras cuentas de propiedad externa O a otras cuentas de contrato creando y firmando una transacción con su clave privada**. Pero un mensaje de una cuenta de propiedad externa a una cuenta de contrato activa el código de la cuenta de contrato, permitiéndole realizar varias acciones (por ejemplo, transferir tokens, escribir en el almacenamiento interno, acuñar nuevos tokens, realizar algún cálculo, crear nuevos contratos, etc.).

- **A diferencia de las cuentas de propiedad externa, las cuentas de contrato no pueden iniciar nuevas transacciones por sí mismas,** sino que sólo pueden realizar transacciones en respuesta a otras transacciones que hayan recibido (de una cuenta de propiedad externa o de otra cuenta de contrato). Aprenderemos más sobre las llamadas de contrato a contrato en la sección "*Transacciones y mensajes*".



Therefore, any action that occurs on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts.

Por lo tanto, cualquier acción que se produzca en la blockchain de Ethereum siempre se pone en marcha mediante transacciones disparadas desde cuentas controladas externamente.



Account state

The account **state** consists of four components, which are present regardless of the type of account:

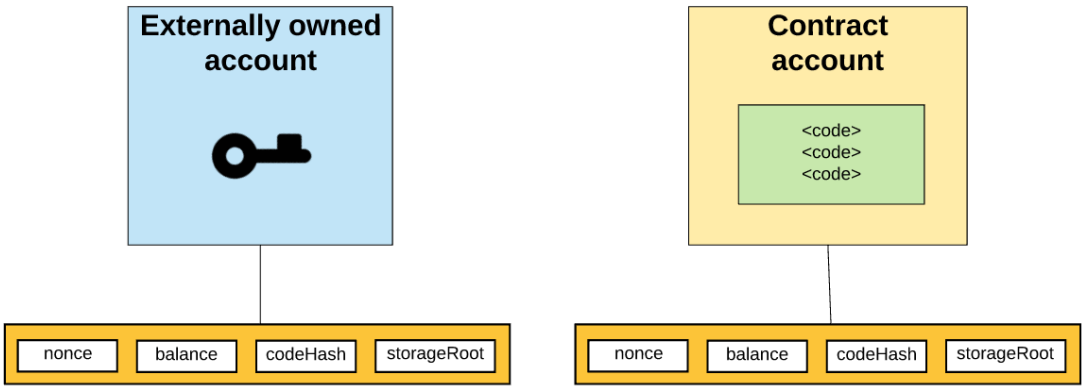
- **nonce**: If the account is an externally owned account, this number represents the number of transactions sent from the account's address. If the account is a contract account, the nonce is the number of contracts created by the account.
- **balance**: The number of Wei owned by this address. There are 1e+18 Wei per Ether.
- **storageRoot**: A hash of the root node of a Merkle Patricia tree (we'll explain Merkle trees later on). This tree encodes the hash of the storage contents of this account, and is empty by default.
- **codeHash**: The hash of the EVM (Ethereum Virtual Machine—more on this later) code of this account. For contract accounts, this is the code that gets

hashed and stored as the **codeHash**. For externally owned accounts, the **codeHash** field is the hash of the empty string.

Estado de la cuenta

El **estado de la cuenta** consta de cuatro componentes, que están presentes independientemente del tipo de cuenta:

- **nonce**: Si la cuenta es de propiedad externa, este número representa el número de transacciones enviadas desde la dirección de la cuenta. Si la cuenta es de contrato, el nonce es el número de contratos creados por la cuenta.
- **balance**: El número de Wei que posee esta dirección. Hay 1e+18 Wei por Ether.
- **storageRoot**: Un hash del nodo raíz de un árbol Merkle Patricia (explicaremos los árboles Merkle más adelante). Este árbol codifica el hash del contenido de almacenamiento de esta cuenta, y está vacío por defecto.
- **códigoHash**: El hash del código de la EVM (Máquina Virtual de Ethereum - más sobre esto más adelante) de esta cuenta. Para las cuentas de contrato, este es el código que se convierte en hash y se almacena como **codeHash**. Para las cuentas de propiedad externa, el campo **codeHash** es el hash de la cadena vacía.



World state

Okay, so we know that Ethereum's global state consists of a mapping between account addresses and the account states. This mapping is stored in a data structure known as a **Merkle Patricia tree**.

A Merkle tree (or also referred as “Merkle trie”) is a type of binary tree composed of a set of nodes with:

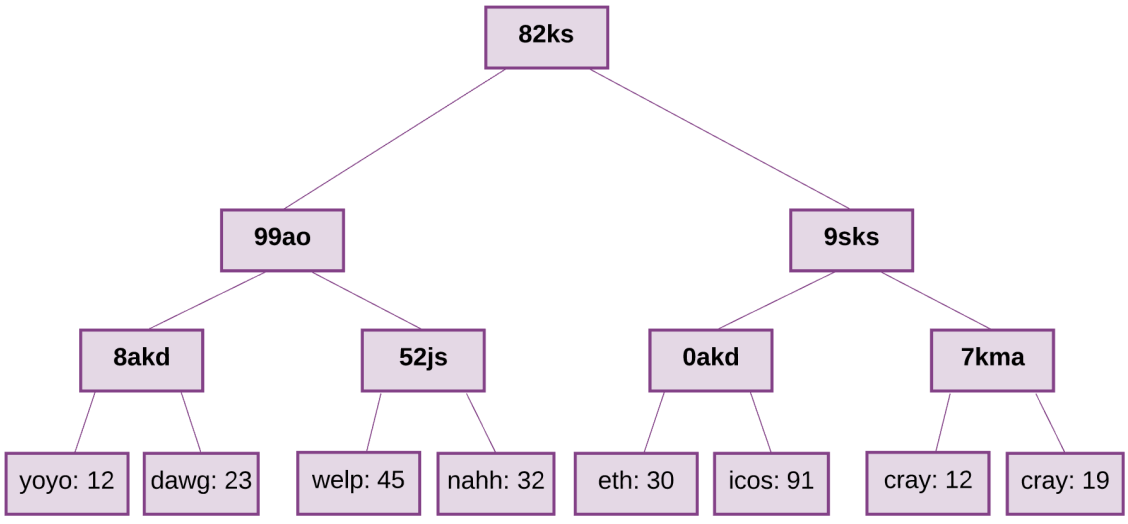
- a large number of leaf nodes at the bottom of the tree that contain the underlying data
- a set of intermediate nodes, where each node is the hash of its two child nodes
- a single root node, also formed from the hash of its two child node, representing the top of the tree

Estado mundial

Bien, sabemos que el estado global de Ethereum consiste en un mapeo entre las direcciones de las cuentas y los estados de las mismas. Este mapeo se almacena en una estructura de datos conocida como **árbol de Merkle Patricia**.

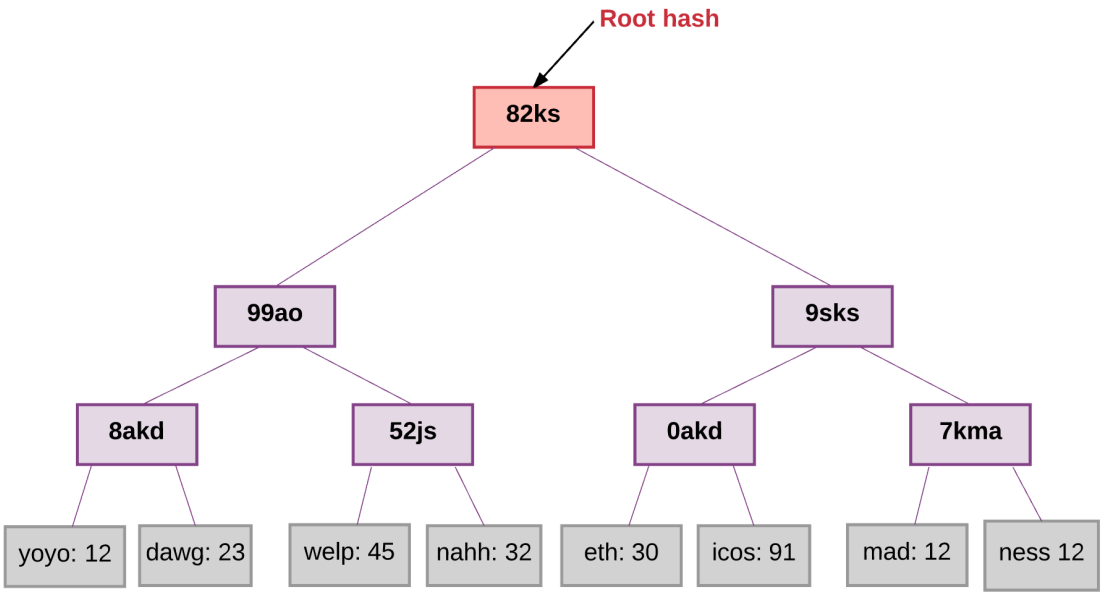
Un árbol de Merkle (o también conocido como "Merkle trie") es un tipo de árbol binario compuesto por un conjunto de nodos con:

- un gran número de nodos hoja en la parte inferior del árbol que contienen los datos subyacentes
- un conjunto de nodos intermedios, donde cada nodo es el hash de sus dos nodos hijos
- un único nodo raíz, también formado por el hash de sus dos nodos hijos, que representa la parte superior del árbol



The data at the bottom of the tree is generated by splitting the data that we want to store into *chunks*, then splitting the chunks into *buckets*, and then taking the hash of each bucket and repeating the same process until the total number of hashes remaining becomes only one: **the root hash.**

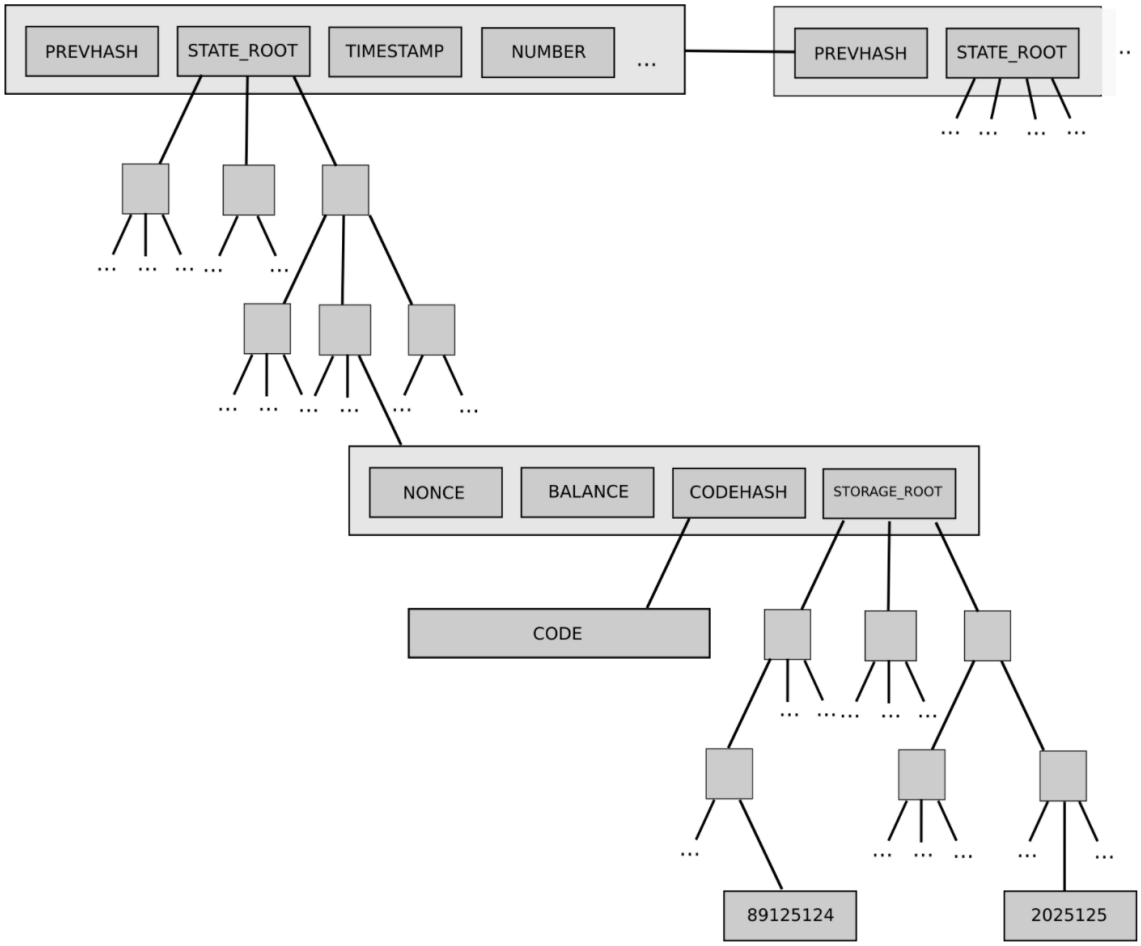
Los datos de la parte inferior del árbol se generan dividiendo los datos que queremos almacenar en *trozos,* luego dividiendo los trozos en *cubos*, y luego tomando el hash de cada cubo y repitiendo el mismo proceso hasta que el número total de hashes restantes sea sólo uno: **el hash de la raíz.**



This tree is required to have a key for every value stored inside it. Beginning from the root node of the tree, the key should tell you which child node to follow to get to the corresponding value, which is stored in the leaf nodes. In Ethereum's case, the key/value mapping for the state tree is between addresses and their associated accounts, including the balance, nonce, codeHash, and storageRoot for each account (where the storageRoot is itself a tree).

Este árbol debe tener una clave para cada valor almacenado en él. Empezando por el nodo raíz del árbol, la clave debe indicar qué nodo hijo hay que seguir para llegar al valor correspondiente, que se almacena en los nodos hoja. En el caso de Ethereum, el mapeo clave/valor para el árbol de estado es entre direcciones y sus

cuentas asociadas, incluyendo el balance, nonce, codeHash, y storageRoot para cada cuenta (donde el storageRoot es en sí mismo un árbol).



Source: Ethereum whitepaper

This same trie structure is used also to store transactions and receipts. More specifically, every block has a “header” which stores the hash of the root node of three different Merkle trie structures, including:

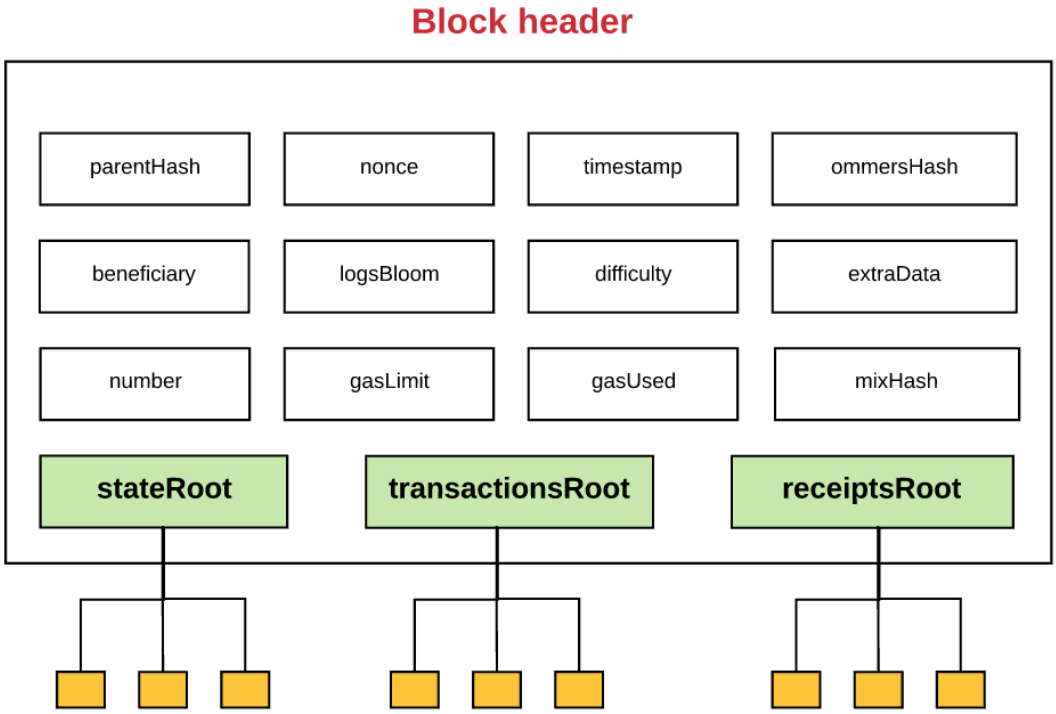
1. State trie
2. Transactions trie
3. Receipts trie

Fuente: Whitepaper de Ethereum.

Esta misma estructura trie se utiliza también para almacenar transacciones y recibos. Más concretamente, cada bloque tiene una "cabecera" que almacena el

hash del nodo raíz de tres estructuras Merkle trie diferentes, que incluyen

1. Trie de estado
2. Trie de transacciones
3. Ingresos trie



The ability to store all this information efficiently in Merkle tries is incredibly useful in Ethereum for what we call “light clients” or “light nodes.” Remember that a blockchain is maintained by a bunch of nodes. Broadly speaking, there are two types of nodes: full nodes and light nodes.

A full archive node synchronizes the blockchain by downloading the full chain, from the genesis block to the current head block, executing all of the transactions contained within. Typically, miners store the full archive node, because they are required to do so for the mining process. It is also possible to download a full node without executing every transaction. Regardless, any full node contains the entire chain.

But unless a node needs to execute every transaction or easily query historical data, there’s really no need to store the entire chain. This is where the concept of a light node comes in. **Instead of downloading and storing the full chain and executing**

all of the transactions, light nodes download only the chain of headers, from the genesis block to the current head, without executing any transactions or retrieving any associated state. Because light nodes have access to block headers, which contain hashes of three tries, they can still easily generate and receive verifiable answers about transactions, events, balances, etc.

The reason this works is because hashes in the Merkle tree propagate upward—if a malicious user attempts to swap a fake transaction into the bottom of a Merkle tree, this change will cause a change in the hash of the node above, which will change the hash of the node above that, and so on, until it eventually changes the root of the tree.

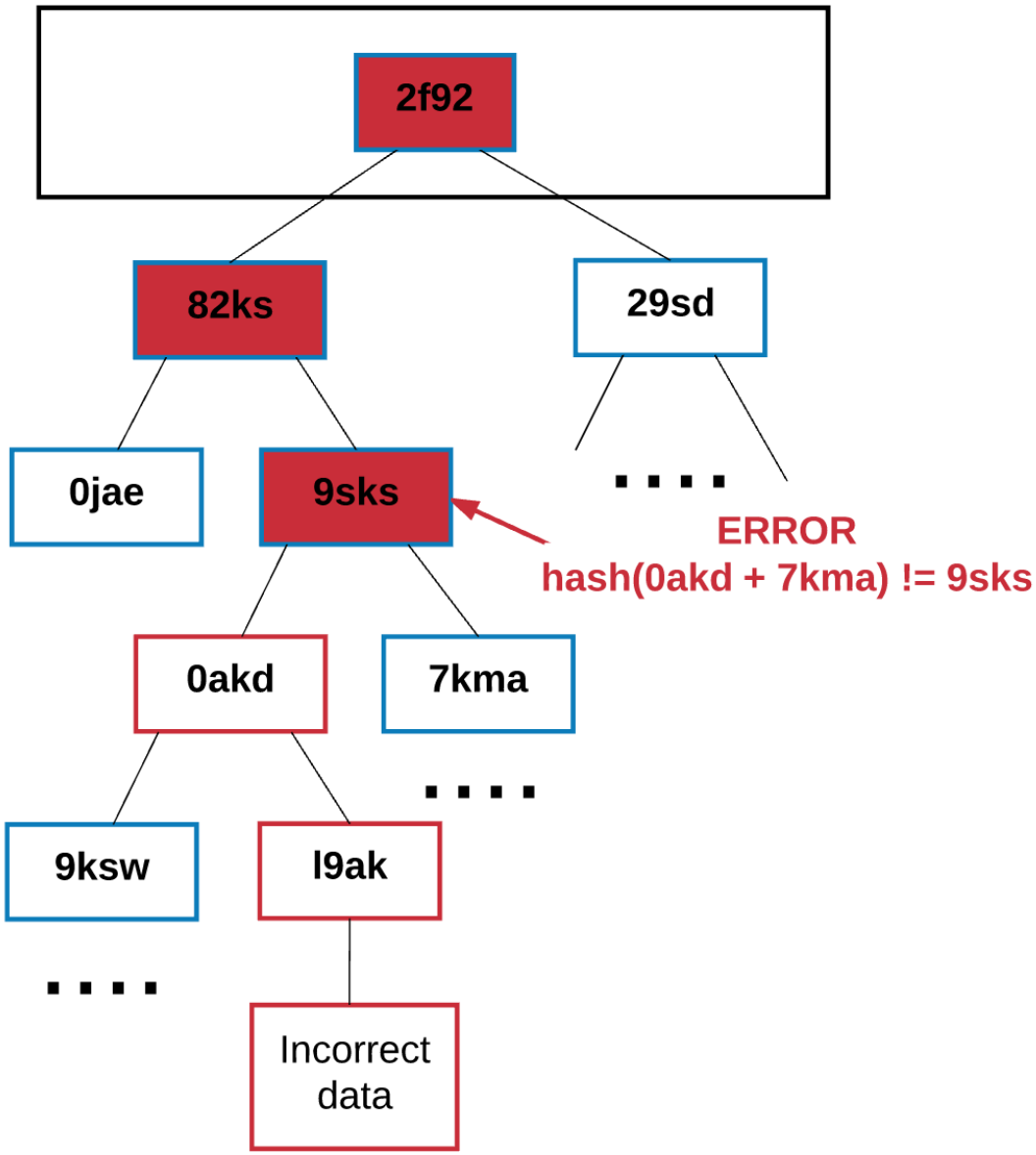
La capacidad de almacenar toda esta información de manera eficiente en Merkle tries es increíblemente útil en Ethereum para lo que llamamos "clientes ligeros" o "nodos ligeros". Recuerda que una blockchain es mantenida por un grupo de nodos. En términos generales, hay dos tipos de nodos: nodos completos y nodos ligeros.

Un nodo de archivo completo sincroniza la blockchain descargando la cadena completa, desde el bloque de génesis hasta el bloque de cabecera actual, ejecutando todas las transacciones contenidas en él. Normalmente, los mineros almacenan el nodo de archivo completo, porque se les exige hacerlo para el proceso de minería. También es posible descargar un nodo completo sin ejecutar todas las transacciones. En cualquier caso, cualquier nodo completo contiene toda la cadena.

Pero a menos que un nodo necesite ejecutar cada transacción o consultar fácilmente los datos históricos, no hay realmente necesidad de almacenar toda la cadena. Aquí es donde entra el concepto de nodo ligero. **En lugar de descargar y almacenar la cadena completa y ejecutar todas las transacciones, los nodos ligeros descargan sólo la cadena de cabeceras, desde el bloque génesis hasta la cabecera actual, sin ejecutar ninguna transacción ni recuperar ningún estado asociado.** Como los nodos ligeros tienen acceso a las cabeceras de los bloques, que contienen hashes de tres intentos, pueden seguir generando y recibiendo fácilmente respuestas verificables sobre transacciones, eventos, balances, etc.

La razón por la que esto funciona es porque los hashes en el árbol Merkle se propagan hacia arriba - si un usuario malicioso intenta intercambiar una transacción falsa en la parte inferior de un árbol Merkle, este cambio provocará un cambio en el

hash del nodo de arriba, que cambiará el hash del nodo de arriba, y así sucesivamente, hasta que finalmente cambie la raíz del árbol.

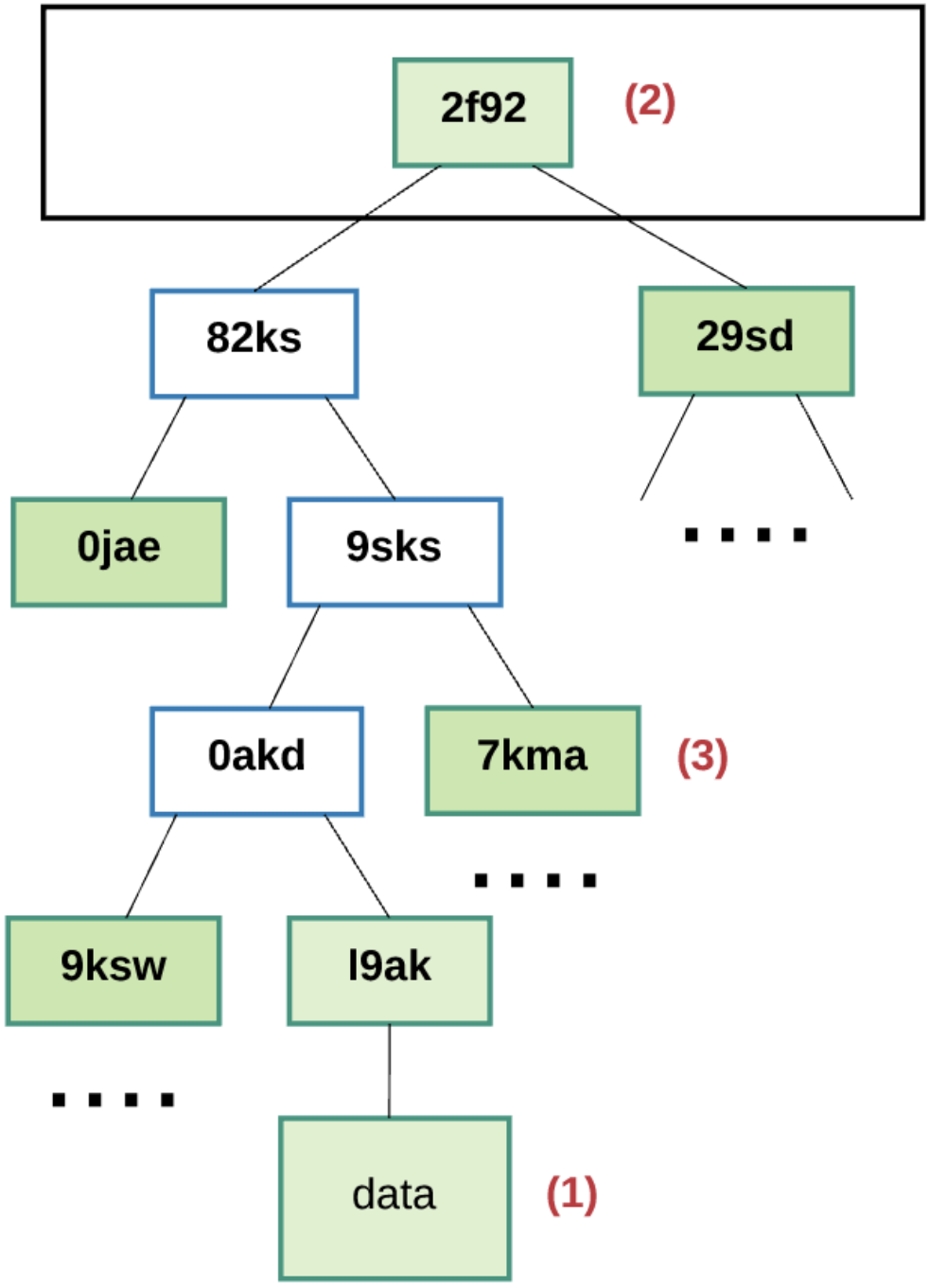


Any node that wants to verify a piece of data can use something called a “Merkle proof” to do so. A Merkle proof consists of:

1. A chunk of data to be verified and its hash
2. The root hash of the tree
3. The “branch” (all of the partner hashes going up along the path from the chunk to the root)

Cualquier nodo que quiera verificar un dato puede utilizar algo llamado "prueba Merkle" para hacerlo. Una prueba Merkle consiste en:

1. Un trozo de datos a verificar y su hash
2. El hash de la raíz del árbol
3. La "rama" (todos los hashes asociados que suben por el camino desde el trozo hasta la raíz)



Anyone reading the proof can verify that the hashing for that branch is consistent all the way up the tree, and therefore that the given chunk is actually at that position in the tree.

In summary, the benefit of using a Merkle Patricia tree is that the root node of this structure is cryptographically dependent on the data stored in the tree, and so the hash of the root node can be used as a secure identity for this data.

Since the block header includes the root hash of the state, transactions, and receipts trees, any node can validate a small part of state of Ethereum without needing to store the entire state, which can be potentially unbounded in size.

Cualquiera que lea la prueba puede verificar que el hash de esa rama es consistente a lo largo de todo el árbol, y por lo tanto que el trozo dado está realmente en esa posición del árbol.

En resumen, la ventaja de utilizar un árbol Merkle Patricia es que el nodo raíz de esta estructura depende criptográficamente de los datos almacenados en el árbol, por lo que el hash del nodo raíz puede utilizarse como una identidad segura para estos datos. Como la cabecera del bloque incluye el hash de la raíz de los árboles de estado, transacciones y recibos, cualquier nodo puede validar una pequeña parte del estado de Ethereum sin necesidad de almacenar todo el estado, cuyo tamaño puede ser potencialmente ilimitado.

Gas and payment

One very important concept in Ethereum is the concept of fees. **Every computation that occurs as a result of a transaction on the Ethereum network incurs a fee—there's no free lunch!** This fee is paid in a denomination called "gas."

Gas is the unit used to measure the fees required for a particular computation. **Gas price** is the amount of Ether you are willing to spend on every unit of gas, and is measured in "gwei." "Wei" is the smallest unit of Ether, where 1^{018} Wei represents 1 Ether. One gwei is 1,000,000,000 Wei.

With every transaction, a sender sets a **gas limit** and **gas price**. The product of **gas price** and **gas limit** represents the maximum amount of Wei that the sender is willing to pay for executing a transaction.

For example, let's say the sender sets the gas limit to 50,000 and a gas price to 20 gwei. This implies that the sender is willing to spend at most 50,000 x 20 gwei = 1,000,000,000,000,000 Wei = 0.001 Ether to execute that transaction.

Gastos y pagos

Un concepto muy importante en Ethereum es el de las tasas. **Cada cómputo que ocurre como resultado de una transacción en la red Ethereum incurre en una**

tarifa - ¡no hay almuerzo gratis! Esta tarifa se paga en una denominación llamada "gas".

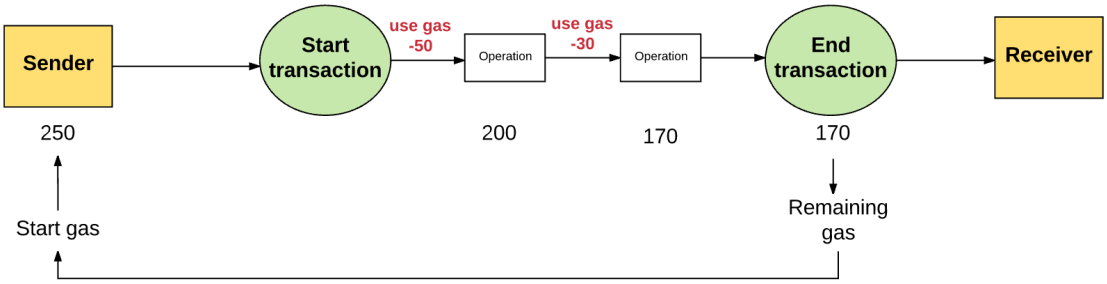
Gas es la unidad utilizada para medir las tasas requeridas para un cálculo particular. **El precio del gas** es la cantidad de éter que estás dispuesto a gastar por cada unidad de gas, y se mide en "gwei". "Wei" es la unidad más pequeña de Éter, donde 1^{018} Wei representa 1 Éter. Un gwei es 1.000.000.000 de Wei.

Con cada transacción, un emisor establece un **límite de gwei** y un **precio de gwei**. El producto del **precio del gas** y el **límite del gas** representa la cantidad máxima de Wei que el emisor está dispuesto a pagar por ejecutar una transacción.

Por ejemplo, digamos que el remitente fija el límite de gas en 50.000 y el precio del gas en 20 gwei. Esto implica que el remitente está dispuesto a gastar como máximo 50.000 x 20 gwei = 1.000.000.000.000 Wei = 0,001 Ether para ejecutar esa transacción.

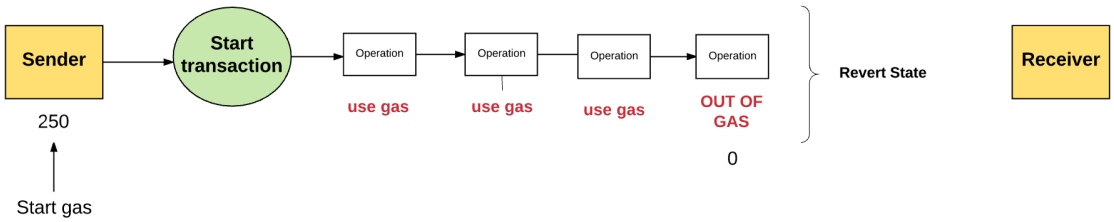
Remember that the gas limit represents the maximum gas the sender is willing to spend money on. If they have enough Ether in their account balance to cover this maximum, they’re good to go. The sender is refunded for any unused gas at the end of the transaction, exchanged at the original rate.

Recuerde que el límite de gas representa el máximo de gas que el remitente está dispuesto a gastar. Si el remitente tiene suficiente Ether en su cuenta para cubrir este máximo, puede seguir adelante. El remitente recibe el reembolso del gas no utilizado al final de la transacción, intercambiado a la tarifa original.



In the case that the sender does not provide the necessary gas to execute the transaction, the transaction runs “out of gas” and is considered invalid. In this case, the transaction processing aborts and any state changes that occurred are reversed, such that we end up back at the state of Ethereum prior to the transaction. Additionally, a record of the transaction failing gets recorded, showing what transaction was attempted and where it failed. And since the machine already expended effort to run the calculations before running out of gas, logically, **none of the gas is refunded to the sender.**

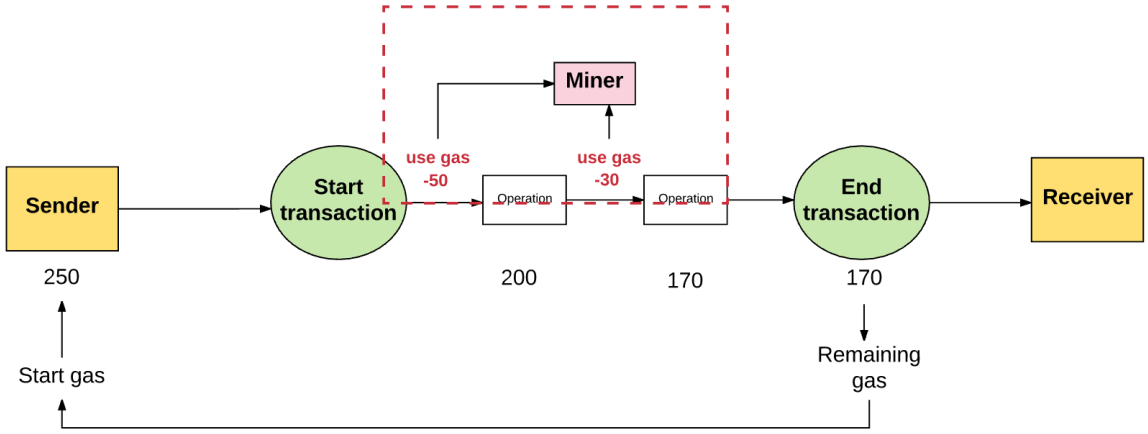
En el caso de que el emisor no proporcione el gas necesario para ejecutar la transacción, ésta se queda "sin gas" y se considera inválida. En este caso, el procesamiento de la transacción se aborta y cualquier cambio de estado que se haya producido se invierte, de forma que volvemos al estado de Ethereum anterior a la transacción. Además, se registra el fallo de la transacción, mostrando qué transacción se intentó y dónde falló. Y puesto que la máquina ya ha realizado el esfuerzo de ejecutar los cálculos antes de quedarse sin gasolina, lógicamente, **no se devuelve nada de la gasolina al remitente.**



Where exactly does this gas money go? **All the money spent on gas by the sender is sent to the “beneficiary” address, which is typically the miner’s**

address. Since miners are expending the effort to run computations and validate transactions, miners receive the gas fee as a reward.

¿A dónde va exactamente el dinero de la gasolina? **Todo el dinero gastado en gasolina por el remitente se envía a la dirección del "beneficiario", que suele ser la del minero.** Dado que los mineros se esfuerzan por realizar los cálculos y validar las transacciones, reciben la tasa de gas como recompensa.



Typically, the higher the gas price the sender is willing to pay, the greater the value the miner derives from the transaction. Thus, the more likely miners will be to select it. In this way, miners are free to choose which transactions they want to validate or ignore. In order to guide senders on what gas price to set, miners have the option of advertising the minimum gas price for which they will execute transactions.

Normalmente, cuanto mayor sea el precio del gas que el emisor está dispuesto a pagar, mayor será el valor que el minero obtenga de la transacción. Por lo tanto, es más probable que los mineros la seleccionen. De este modo, los mineros son libres de elegir qué transacciones quieren validar o ignorar. Para orientar a los emisores sobre el precio del gas que deben fijar, los mineros tienen la opción de anunciar el precio mínimo del gas por el que ejecutarán las transacciones.

There are fees for storage, too

Not only is gas used to pay for computation steps, it is also used to pay for storage usage. The total fee for storage is proportional to the smallest multiple of 32 bytes used.

Fees for storage have some nuanced aspects. For example, since increased storage increases the size of the Ethereum state database on *all* nodes, there's an incentive to keep the amount of data stored small. For this reason, if a transaction has a step that clears an entry in the storage, the fee for executing that operation of is waived, AND a refund is given for freeing up storage space.

También hay tarifas por el almacenamiento

El gas no sólo se utiliza para pagar los pasos de cálculo, sino también para pagar el uso del almacenamiento. La tarifa total por el almacenamiento es proporcional al menor múltiplo de 32 bytes utilizado.

Las tasas por almacenamiento tienen algunos aspectos matizados. Por ejemplo, dado que el aumento del almacenamiento incrementa el tamaño de la base de datos del estado de Ethereum en *todos* los nodos, existe un incentivo para que la cantidad de datos almacenados sea pequeña. Por esta razón, si una transacción tiene un paso que borra una entrada en el almacenamiento, la tarifa por ejecutar esa operación se queda exenta, Y se da un reembolso por liberar espacio de almacenamiento.

What's the purpose of fees?

One important aspect of the way the Ethereum works is that **every single operation executed by the network is simultaneously effected by every full node.** However, computational steps on the Ethereum Virtual Machine are very expensive. Therefore, Ethereum smart contracts are best used for simple tasks, like running simple business logic or verifying signatures and other cryptographic objects, rather than more complex uses, like file storage, email, or machine learning, which can put a strain on the network. **Imposing fees prevents users from overtaxing the network.**

Ethereum is a Turing complete language. (In short, a Turing machine is a machine that can simulate any computer algorithm (for those not familiar with Turing machines, check out this and this). This allows for loops and makes Ethereum susceptible to the halting problem, a problem in which you cannot dtermine whether or not a program will run infinitely. If there were no fees, a malicious actor could easily try to disrupt the network by executing an infinite loop within a transaction, without any repercussions. Thus, fees protect the network from deliberate attacks.

You might be thinking, "why do we also have to pay for storage?" Well, just like computation, storage on the Ethereum network is a cost that the entire network has

to take the burden of.

¿Cuál es el propósito de las tasas?

Un aspecto importante del funcionamiento de Ethereum es que **cada operación ejecutada por la red es efectuada simultáneamente por cada nodo completo.** Sin embargo, los pasos computacionales en la máquina virtual de Ethereum son muy costosos. Por lo tanto, los contratos inteligentes de Ethereum se utilizan mejor para tareas sencillas, como la ejecución de lógica comercial simple o la verificación de firmas y otros objetos criptográficos, en lugar de usos más complejos, como el almacenamiento de archivos, el correo electrónico o el aprendizaje automático, que pueden poner a prueba la red. **La imposición de tarifas evita que los usuarios sobrecarguen la red.**

Ethereum es un lenguaje completo de Turing. (En pocas palabras, una máquina de Turing es una máquina que puede simular cualquier algoritmo informático (para los que no estén familiarizados con las máquinas de Turing, consulten this y this. Esto permite los bucles y hace que Ethereum sea susceptible al problema de detención, un problema en el que no se puede determinar si un programa se ejecutará o no infinitamente. Si no existieran las tasas, un actor malicioso podría intentar fácilmente perturbar la red ejecutando un bucle infinito dentro de una transacción, sin ninguna repercusión. Así, las tasas protegen a la red de los ataques deliberados.

Quizá pienses: "¿por qué tenemos que pagar también por el almacenamiento?". Bueno, al igual que la computación, el almacenamiento en la red Ethereum es un coste que toda la red tiene que asumir.

Transaction and messages

We noted earlier that Ethereum is a **transaction-based state machine.** In other words, transactions occurring between different accounts are what move the global state of Ethereum from one state to the next.

In the most basic sense, a transaction is a cryptographically signed piece of instruction that is generated by an externally owned account, serialized, and then submitted to the blockchain.

There are two types of transactions: **message calls** and **contract creations** (i.e. transactions that create new Ethereum contracts).

All transactions contain the following components, regardless of their type:

- **nonce**: a count of the number of transactions sent by the sender.
- **gasPrice**: the number of Wei that the sender is willing to pay per unit of gas required to execute the transaction.
- **gasLimit**: the maximum amount of gas that the sender is willing to pay for executing this transaction. This amount is set and paid upfront, before any computation is done.
- **to**: the address of the recipient. In a contract-creating transaction, the contract account address does not yet exist, and so an empty value is used.
- **value**: the amount of Wei to be transferred from the sender to the recipient. In a contract-creating transaction, this value serves as the starting balance within the newly created contract account.
- **v, r, s**: used to generate the signature that identifies the sender of the transaction.
- **init** (only exists for contract-creating transactions): An EVM code fragment that is used to initialize the new contract account. **init** is run only once, and then is discarded. When **init** is first run, it returns the body of the account code, which is the piece of code that is permanently associated with the contract account.
- **data** (optional field that only exists for message calls): the input data (i.e. parameters) of the message call. For example, if a smart contract serves as a domain registration service, a call to that contract might expect input fields such as the domain and IP address.

Transacciones y mensajes

En otras palabras, las transacciones que se producen entre diferentes cuentas son las que mueven el estado global de Ethereum de un estado a otro.

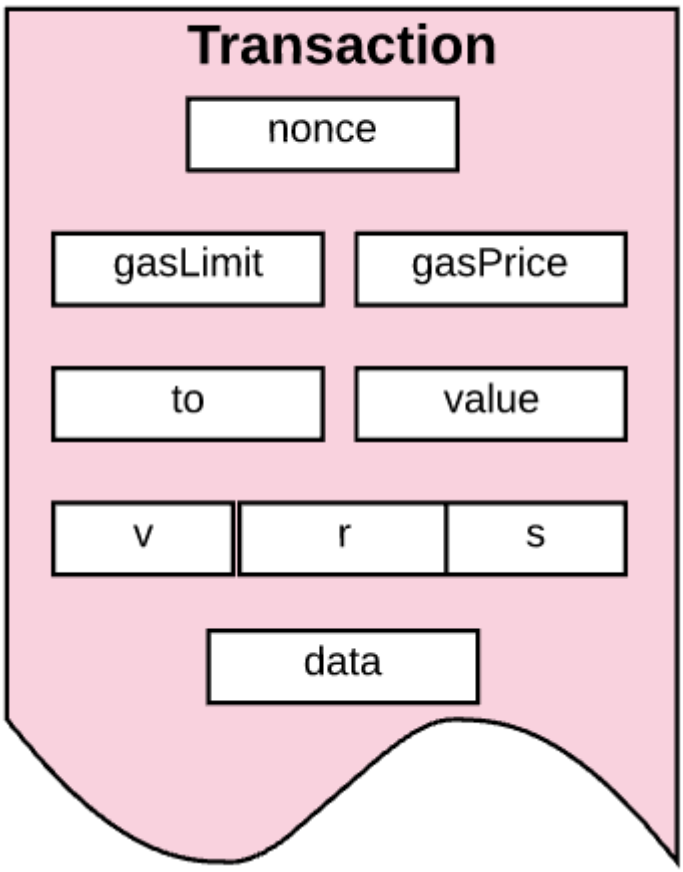
- **En el sentido más básico, una transacción es una pieza de instrucción firmada criptográficamente que es generada por una cuenta externa, serializada, y luego enviada a la cadena de bloques.**

Hay dos tipos de transacciones: **Llamadas a mensajes** y **creaciones de contratos** (es decir, transacciones que crean nuevos contratos de Ethereum).

Todas las transacciones contienen los siguientes componentes, independientemente de su tipo:

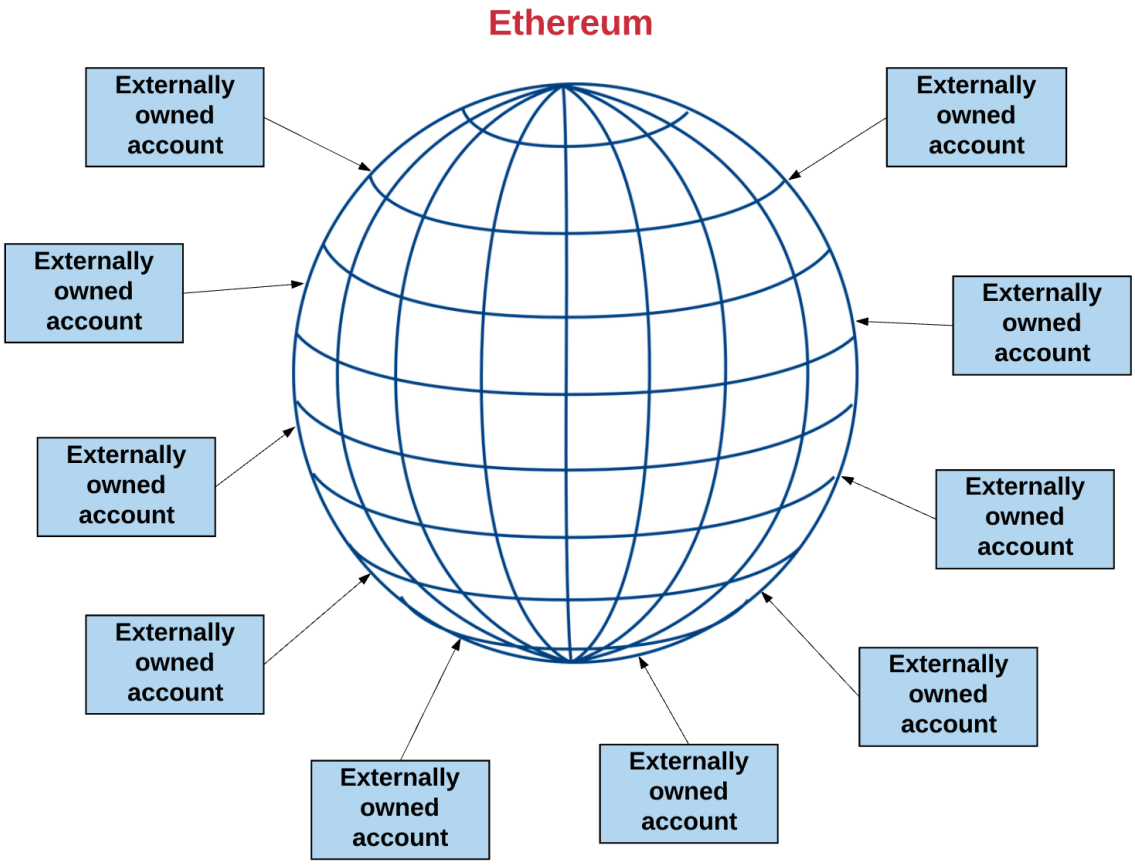
- **nonce**: un recuento del número de transacciones enviadas por el emisor.

- **gasPrice**: el número de Wei que el emisor está dispuesto a pagar por unidad de gas necesaria para ejecutar la transacción.
- **gasLimit**: la cantidad máxima de gas que el emisor está dispuesto a pagar para ejecutar esta transacción. Esta cantidad se establece y se paga por adelantado, antes de realizar cualquier cálculo.
- **a**: la dirección del destinatario. En una transacción de creación de contrato, la dirección de la cuenta del contrato aún no existe, por lo que se utiliza un valor vacío.
- **Valor**: la cantidad de Wei que se va a transferir del remitente al destinatario. En una transacción de creación de contrato, este valor sirve como saldo inicial dentro de la cuenta de contrato recién creada.
- **v, r, s**: se utiliza para generar la firma que identifica al remitente de la transacción.
- **init** (sólo existe para las transacciones de creación de contratos): Un fragmento de código de EVM que se utiliza para inicializar la nueva cuenta de contrato. **init** se ejecuta sólo una vez y luego se descarta. Cuando **init** se ejecuta por primera vez, devuelve el cuerpo del código de la cuenta, que es el fragmento de código que se asocia permanentemente a la cuenta del contrato.
- **datos** (campo opcional que sólo existe para las llamadas a mensajes): los datos de entrada (es decir, los parámetros) de la llamada a mensajes. Por ejemplo, si un contrato inteligente sirve como servicio de registro de dominios, una llamada a ese contrato podría esperar campos de entrada como el dominio y la dirección IP.



We learned in the “*Accounts*” section that transactions—both message calls and contract-creating transactions—are always initiated by externally owned accounts and submitted to the blockchain. Another way to think about it is that transactions are what bridge the external world to the internal state of Ethereum.

Hemos aprendido en la sección "*Cuentas*" que las transacciones -tanto las llamadas de mensajes como las transacciones de creación de contratos- siempre son iniciadas por cuentas de propiedad externa y enviadas a la blockchain. Otra forma de pensar en ello es que las transacciones son las que unen el mundo externo con el estado interno de Ethereum.



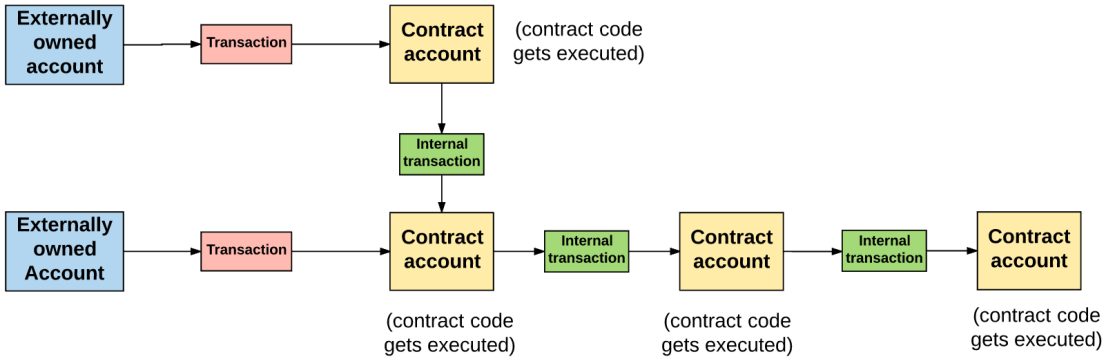
But this doesn't mean that contracts can't talk to other contracts. **Contracts that exist within the global scope of Ethereum's state can talk to other contracts within that same scope. The way they do this is via "messages" or "internal transactions" to other contracts.** We can think of messages or internal transactions as being similar to transactions, with the major difference that they are NOT generated by externally owned accounts. Instead, they are generated by contracts. They are virtual objects that, unlike transactions, are not serialized and only exist in the Ethereum execution environment.

When one contract sends an internal transaction to another contract, the associated code that exists on the recipient contract account is executed.

Pero esto no significa que los contratos no puedan hablar con otros contratos. **Los contratos que existen dentro del ámbito global del estado de Ethereum pueden hablar con otros contratos dentro de ese mismo ámbito. La forma en que lo hacen es a través de "mensajes" o "transacciones internas" a otros contratos.** Podemos pensar que los mensajes o transacciones internas son

similares a las transacciones, con la gran diferencia de que NO son generados por cuentas de propiedad externa. En cambio, son generados por contratos. Son objetos virtuales que, a diferencia de las transacciones, no se serializan y sólo existen en el entorno de ejecución de Ethereum.

Cuando un contrato envía una transacción interna a otro contrato, se ejecuta el código asociado que existe en la cuenta del contrato receptor.



One important thing to note is that internal transactions or messages don't contain a **gasLimit.** This is because the gas limit is determined by the external creator of the original transaction (i.e. some externally owned account). The gas limit that the externally owned account sets must be high enough to carry out the transaction, including any sub-executions that occur as a result of that transaction, such as contract-to-contract messages. If, in the chain of transactions and messages, a particular message execution runs out of gas, then that message's execution will revert, along with any subsequent messages triggered by the execution. However, the parent execution does not need to revert.

Una cosa importante a tener en cuenta es que las transacciones o mensajes internos no contienen un **límite de gas.** Esto se debe a que el límite de gas está determinado por el creador externo de la transacción original (es decir, alguna cuenta de propiedad externa). El límite de gas que la cuenta de propiedad externa establece debe ser lo suficientemente alto como para llevar a cabo la transacción, incluyendo cualquier sub-ejecución que ocurra como resultado de esa transacción, como los mensajes de contrato a contrato. Si, en la cadena de transacciones y mensajes, la ejecución de un mensaje en particular se queda sin gas, entonces la ejecución de ese mensaje se revertirá, junto con cualquier mensaje posterior

desencadenado por la ejecución. Sin embargo, la ejecución principal no necesita revertirse.

Blocks

All transactions are grouped together into “blocks.” A blockchain contains a series of such blocks that are chained together.

In Ethereum, a block consists of:

- the **block header**
- information about the **set of transactions** included in that block
- a **set of other block headers for the current block’s ommers**.

Bloques

Todas las transacciones se agrupan en "bloques". Una cadena de bloques contiene una serie de tales bloques que están encadenados.

En Ethereum, un bloque consiste en

- la **cabecera del bloque**
- información sobre el **conjunto de transacciones** incluidas en ese bloque
- un **conjunto de otras cabeceras de bloque para los omeros del bloque actual**.

Ommers explained

What the heck is an “ommer?” An ommer is a block whose parent is equal to the current block’s parent’s parent. Let’s take a quick dive into what ommers are used for and why a block contains the block headers for ommers.

Because of the way Ethereum is built, block times are much lower (~15 seconds) than those of other blockchains, like Bitcoin (~10 minutes). This enables faster transaction processing. However, one of the downsides of shorter block times is that more competing block solutions are found by miners. These competing blocks are also referred to as “orphaned blocks” (i.e. mined blocks do not make it into the main chain).

The purpose of ommers is to help reward miners for including these orphaned blocks. The ommers that miners include must be “valid,” meaning within the sixth generation or smaller of the present block. After six children, stale orphaned blocks can no longer be referenced (because including older transactions would complicate things a bit).

Ommer blocks receive a smaller reward than a full block. Nonetheless, there’s still some incentive for miners to include these orphaned blocks and reap a reward.

Explicación de los ommers

¿Qué diablos es un "ommer"? Un ommer es un bloque cuyo padre es igual al padre del bloque actual. Veamos rápidamente para qué se utilizan los ommers y por qué un bloque contiene las cabeceras de bloque para los ommers.

Debido a la forma en que Ethereum está construido, los tiempos de los bloques son mucho más bajos (~15 segundos) que los de otras cadenas de bloques, como Bitcoin (~10 minutos). Esto permite un procesamiento más rápido de las transacciones. Sin embargo, una de las desventajas de los tiempos de bloque más cortos es que los mineros encuentran más soluciones de bloque que compiten entre sí. Estos bloques competidores también se denominan "bloques huérfanos" (es decir, los bloques minados no llegan a la cadena principal).

El propósito de los ommers es ayudar a recompensar a los mineros por incluir estos bloques huérfanos. Los ommers que los mineros incluyen deben ser "válidos", es decir, dentro de la sexta generación o menor del bloque actual. Después de seis hijos, los bloques huérfanos y obsoletos ya no pueden ser referenciados (porque incluir transacciones más antiguas complicaría un poco las cosas).

Los bloques huérfanos reciben una recompensa menor que un bloque completo. No obstante, los mineros siguen teniendo algún incentivo para incluir estos bloques huérfanos y obtener una recompensa.

Block header

Let’s get back to blocks for a moment. We mentioned previously that every block has a block “header,” but what exactly is this?

A block header is a portion of the block consisting of:

- **parentHash**: a hash of the parent block’s header (this is what makes the block set a “chain”)

- **ommersHash**: a hash of the current block’s list of ommers
- **beneficiary**: the account address that receives the fees for mining this block
- **stateRoot**: the hash of the root node of the state trie (recall how we learned that the state trie is stored in the header and makes it easy for light clients to verify anything about the state)
- **transactionsRoot**: the hash of the root node of the trie that contains all transactions listed in this block
- **receiptsRoot**: the hash of the root node of the trie that contains the receipts of all transactions listed in this block
- **logsBloom**: a Bloom filter (data structure) that consists of log information
- **difficulty**: the difficulty level of this block
- **number**: the count of current block (the genesis block has a block number of zero; the block number increases by 1 for each each subsequent block)
- **gasLimit**: the current gas limit per block
- **gasUsed**: the sum of the total gas used by transactions in this block
- **timestamp**: the unix timestamp of this block’s inception
- **extraData**: extra data related to this block
- **mixHash**: a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce**: a hash that, when combined with the mixHash, proves that this block has carried out enough computation

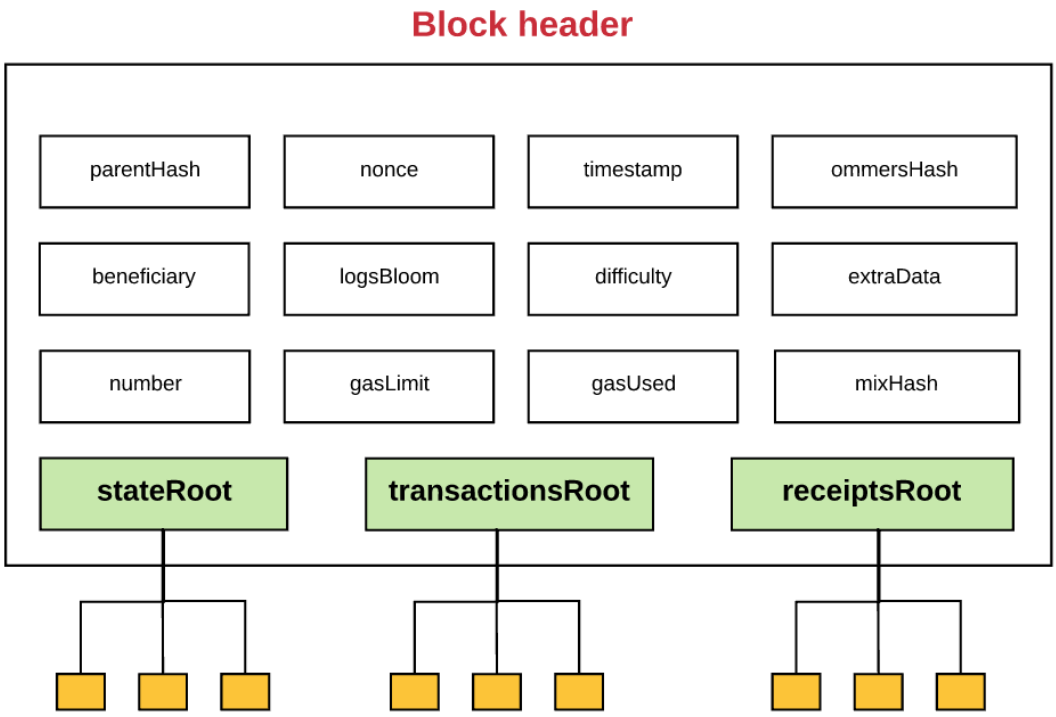
Cabecera de bloque

Volvamos a los bloques por un momento. Anteriormente hemos mencionado que cada bloque tiene una "cabecera" de bloque, pero ¿qué es esto exactamente?

Una cabecera de bloque es una parte del bloque que consiste en

- **parentHash**: un hash de la cabecera del bloque padre (esto es lo que hace que el conjunto de bloques sea una "cadena")
- **ommersHash**: un hash de la lista de ommers del bloque actual

- **beneficiario**: la dirección de la cuenta que recibe las tasas por minar este bloque
- **stateRoot**: el hash del nodo raíz de la triada de estado (recuerde que aprendimos que la triada de estado se almacena en la cabecera y facilita a los clientes ligeros la verificación de cualquier cosa sobre el estado)
- **TransactionsRoot:** el hash del nodo raíz del trie que contiene todas las transacciones listadas en este bloque
- **ReceiptsRoot**: el hash del nodo raíz del trie que contiene los recibos de todas las transacciones listadas en este bloque
- **logsBloom**: un filtro Bloom (estructura de datos) que consiste en información de registro
- **difficulty**: el nivel de dificultad de este bloque
- **número**: el recuento del bloque actual (el bloque génesis tiene un número de bloque de cero; el número de bloque aumenta en 1 para cada bloque posterior)
- **gasLimit**: el límite de gas actual por bloque
- **gasUsed**: la suma del gas total utilizado por las transacciones en este bloque
- **timestamp**: la marca de tiempo unix del inicio de este bloque
- **extraData**: datos adicionales relacionados con este bloque
- **mixHash**: un hash que, combinado con el nonce, demuestra que este bloque ha realizado suficientes cálculos
- **Noce:** un hash que, cuando se combina con el mixHash, demuestra que este bloque ha realizado suficientes cálculos



Notice how every block header contains three trie structures for:

- state (**stateRoot**)
- transactions (**transactionsRoot**)
- receipts (**receiptsRoot**)

These trie structures are nothing but the Merkle Patricia tries we discussed earlier.

Additionally, there are a few terms from the above description that are worth clarifying. Let's take a look.

Obsérvese que cada cabecera de bloque contiene tres estructuras trie para:

- estado (**stateRoot**)
- transacciones (**transactionsRoot**)
- recibos (**receiptsRoot**)

Estas estructuras trie no son más que los intentos de Merkle Patricia de los que hemos hablado antes.

Además, hay algunos términos de la descripción anterior que vale la pena aclarar. Echemos un vistazo.

Logs

Ethereum allows for logs to make it possible to track various transactions and messages. A contract can explicitly generate a log by defining “events” that it wants to log.

A log entry contains:

- the logger’s account address,
- a series of topics that represent various events carried out by this transaction, and
- any data associated with these events.

Logs are stored in a bloom filter, which stores the endless log data in an efficient manner.

Logs

Ethereum permite los registros para hacer posible el seguimiento de varias transacciones y mensajes. Un contrato puede generar explícitamente un registro definiendo los "eventos" que quiere registrar.

Una entrada de registro contiene:

- la dirección de la cuenta del registrador,
- una serie de temas que representan varios eventos realizados por esta transacción, y
- cualquier dato asociado a estos eventos.

Los registros se almacenan en un filtro de floración que almacena los interminables datos del registro de manera eficiente.

Transaction receipt

Logs stored in the header come from the log information contained in the transaction receipt. Just as you receive a receipt when you buy something at a store, Ethereum generates a receipt for every transaction. Like you’d expect, each receipt contains certain information about the transaction. This receipt includes items like:

- the block number
- block hash
- transaction hash
- gas used by the current transaction
- cumulative gas used in the current block after the current transaction has executed
- logs created when executing the current transaction
- ..and so on

Recibo de la transacción

Los registros almacenados en la cabecera provienen de la información de registro contenida en el recibo de la transacción. Al igual que recibes un recibo cuando compras algo en una tienda, Ethereum genera un recibo por cada transacción. Como es de esperar, cada recibo contiene cierta información sobre transacción. Este recibo incluye elementos como

- el número de bloque
- el hash del bloque
- el hash de la transacción
- el gas utilizado por la transacción actual
- gas acumulado utilizado en el bloque actual tras la ejecución de la transacción actual
- registros creados al ejecutar la transacción actual
- ..y así sucesivamente

Block difficulty

The “difficulty” of a block is used to enforce consistency in the time it takes to validate blocks. The genesis block has a difficulty of 131,072, and a special formula is used to calculate the difficulty of every block thereafter. If a certain block is validated more quickly than the previous block, the Ethereum protocol increases that block’s difficulty.

The difficulty of the block affects the **nonce,** which is a hash that must be calculated when mining a block, using the proof-of-work algorithm.

The relationship between the block's **difficulty** and **nonce** is mathematically formalized as:

Dificultad de los bloques

La "dificultad" de un bloque se utiliza para mantener la coherencia en el tiempo de validación de los bloques. El bloque de génesis tiene una dificultad de 131,072, y se utiliza una fórmula especial para calcular la dificultad de cada bloque a partir de entonces. Si un bloque determinado se valida más rápidamente que el anterior, el protocolo de Ethereum aumenta la dificultad de ese bloque.

La dificultad del bloque afecta al **nonce,** que es un hash que debe calcularse al minar un bloque, utilizando el algoritmo de prueba de trabajo.

La relación entre la **dificultad** del bloque y el **nonce** se formaliza matemáticamente como:

$$n \leqslant \frac{2^{256}}{H_d}$$

*where **Hd** is the difficulty.*

*donde **Hd** es la dificultad.*

The only way to find a nonce that meets a difficulty threshold is to use the proof-of-work algorithm to enumerate all of the possibilities. The expected time to find a solution is proportional to the difficulty—the higher the difficulty, the harder it becomes to find the nonce, and so the harder it is to validate the block, which in turn increases the time it takes to validate a new block. **So, by adjusting the difficulty of a block, the protocol can adjust how long it takes to validate a block.**

If, on the other hand, validation time is getting slower, the protocol decreases the difficulty. In this way, the validation time self-adjusts to maintain a constant rate—on average, one block every 15 seconds.

La única manera de encontrar un nonce que cumpla un umbral de dificultad es utilizar el algoritmo de prueba de trabajo para enumerar todas las posibilidades. El tiempo esperado para encontrar una solución es proporcional a la dificultad: cuanto más alta sea la dificultad, más difícil será encontrar el nonce y, por tanto, más difícil será validar el bloque, lo que a su vez aumenta el tiempo necesario para validar un nuevo bloque. **Así, ajustando la dificultad de un bloque, el protocolo puede ajustar el tiempo que se tarda en validar un bloque.**

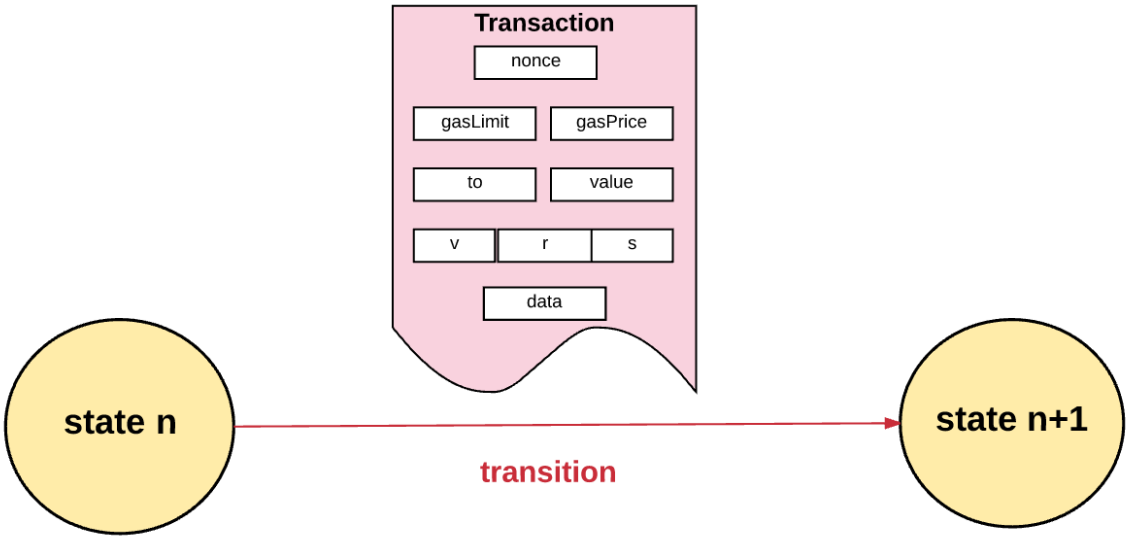
Si, por el contrario, el tiempo de validación es cada vez más lento, el protocolo disminuye la dificultad. De este modo, el tiempo de validación se autoajusta para mantener un ritmo constante: una media de un bloque cada 15 segundos.

Transaction Execution

We've come to one of the most complex parts of the Ethereum protocol: the execution of a transaction. Say you send a transaction off into the Ethereum network to be processed. What happens to transition the state of Ethereum to include your transaction?

Ejecución de la transacción

Hemos llegado a una de las partes más complejas del protocolo de Ethereum: la ejecución de una transacción. Digamos que envías una transacción a la red Ethereum para que sea procesada. ¿Qué sucede para la transición del estado de Ethereum para incluir su transacción?



First, all transactions must meet an initial set of requirements in order to be executed. These include:

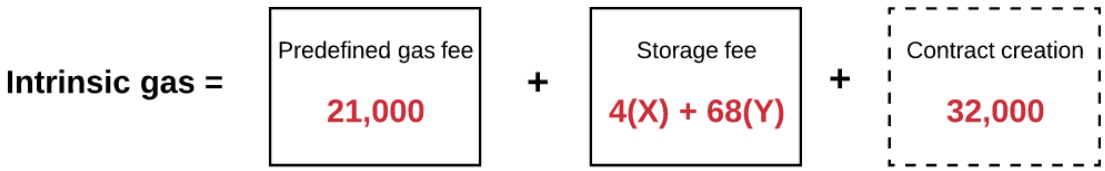
- The transaction must be a properly formatted **RLP**. “RLP” stands for “Recursive Length Prefix” and is a data format used to encode nested arrays of binary data. RLP is the format Ethereum uses to serialize objects.
- Valid transaction signature.
- Valid transaction nonce. Recall that the nonce of an account is the count of transactions sent from that account. To be valid, a transaction nonce must be equal to the sender account’s nonce.
- The transaction’s gas limit must be equal to or greater than the **intrinsic gas** used by the transaction. The intrinsic gas includes:
- a predefined cost of 21,000 gas for executing the transaction
- a gas fee for data sent with the transaction (4 gas for every byte of data or code that equals zero, and 68 gas for every non-zero byte of data or code)
- if the transaction is a contract-creating transaction, an additional 32,000 gas

En primer lugar, todas las transacciones deben cumplir una serie de requisitos iniciales para poder ser ejecutadas. Estos incluyen:

- La transacción debe ser un **RLP** correctamente formateado. "RLP" significa "Recursive Length Prefix" (prefijo de longitud recursiva) y es un formato de

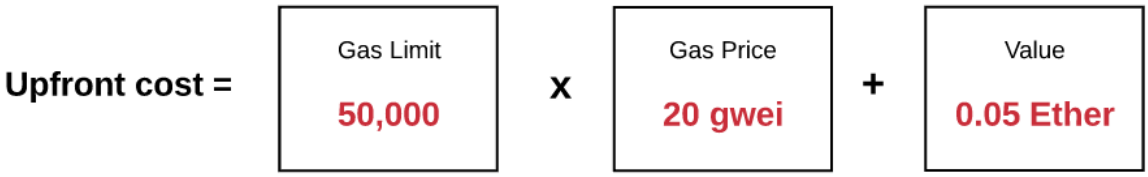
datos utilizado para codificar matrices anidadas de datos binarios. RLP es el formato que Ethereum utiliza para serializar objetos.

- Firma de transacción válida.
- Nonce de transacción válido. Recordemos que el nonce de una cuenta es el recuento de las transacciones enviadas desde esa cuenta. Para ser válido, el nonce de una transacción debe ser igual al nonce de la cuenta remitente.
- El límite de gas de la transacción debe ser igual o mayor que el **gas intrínseco** utilizado por la transacción. El gas intrínseco incluye
- un coste predefinido de 21.000 gas para ejecutar la transacción
- una tasa de gas por los datos enviados con la transacción (4 gas por cada byte de datos o código que sea igual a cero, y 68 gas por cada byte de datos o código que no sea cero)
- si la transacción es una transacción de creación de contratos, 32.000 gas adicionales



- The sender's account balance must have enough Ether to cover the **"upfront" gas costs** that the sender must pay. The calculation for the upfront gas cost is simple: First, the transaction's **gas limit** is multiplied by the transaction's **gas price** to determine the maximum gas cost. Then, this maximum cost is added to the total value being transferred from the sender to the recipient.

El saldo de la cuenta del remitente debe tener suficiente Éter para cubrir los **costes iniciales de gas** que el remitente debe pagar. El cálculo del coste inicial del gas es sencillo: Primero, el **límite de gas** de la transacción se multiplica por el **precio del gas** de la transacción para determinar el coste máximo del gas. A continuación, este coste máximo se añade al valor total que se transfiere del remitente al destinatario.

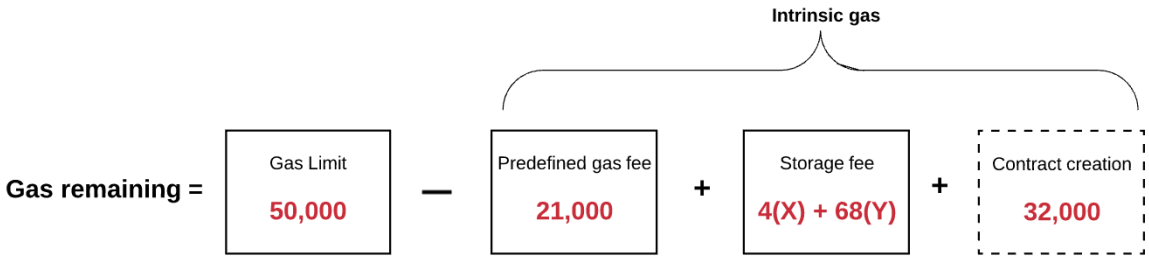


If the transaction meets all of the above requirements for validity, then we move onto the next step.

First, we deduct the upfront cost of execution from the sender's balance, and increase the nonce of the sender's account by 1 to account for the current transaction. At this point, we can calculate the gas remaining as the **total gas limit for the transaction minus the intrinsic gas used.**

Si la transacción cumple todos los requisitos de validez anteriores, pasamos al siguiente paso.

En primer lugar, deducimos el coste inicial de ejecución del saldo del remitente y aumentamos el nonce de la cuenta del remitente en 1 para tener en cuenta la transacción actual. En este punto, podemos calcular el gas restante como el **límite total de gas para la transacción menos el gas intrínseco utilizado.**



Next, the transaction starts executing. Throughout the execution of a transaction, Ethereum keeps track of the "substate." This substate is a way to record information accrued during the transaction that will be needed immediately after the transaction completes. Specifically, it contains:

- **Self-destruct set**: a set of accounts (if any) that will be discarded after the transaction completes.
- **Log series**: archived and indexable checkpoints of the virtual machine's code execution.

- **Refund balance**: the amount to be refunded to the sender account after the transaction. Remember how we mentioned that storage in Ethereum costs money, and that a sender is refunded for clearing up storage? Ethereum keeps track of this using a refund counter. The refund counter starts at zero and increments every time the contract deletes something in storage.

Next, the various computations required by the transaction are processed.

Once all the steps required by the transaction have been processed, and assuming there is no invalid state, the state is finalized by determining the amount of unused gas to be refunded to the sender. In addition to the unused gas, the sender is also refunded some allowance from the “refund balance” that we described above.

Once the sender is refunded:

- the Ether for the gas is given to the miner
- the gas used by the transaction is added to the block gas counter (which keeps track of the total gas used by all transactions in the block, and is useful when validating a block)
- all accounts in the self-destruct set (if any) are deleted

Finally, we’re left with the new state and a set of the logs created by the transaction.

Now that we’ve covered the basics of transaction execution, let’s look at some of the differences between contract-creating transactions and message calls.

A continuación, la transacción comienza a ejecutarse. A lo largo de la ejecución de una transacción, Ethereum hace un seguimiento del "subestado". Este subestado es una forma de registrar la información acumulada durante la transacción que se necesitará inmediatamente después de que ésta se complete. En concreto, contiene:

- **Serie de autodestrucción**: un conjunto de cuentas (si las hay) que se descartarán después de que la transacción se complete.
- **Serie de registros**: puntos de control archivados e indexables de la ejecución del código de la máquina virtual.
- **Saldo de reembolso**: la cantidad que se reembolsará a la cuenta del emisor después de la transacción. ¿Recuerdas que mencionamos que el almacenamiento en Ethereum cuesta dinero, y que un remitente recibe un reembolso por liquidar el almacenamiento? Ethereum hace un seguimiento de

esto utilizando un contador de reembolsos. El contador de reembolsos comienza en cero y se incrementa cada vez que el contrato borra algo en el almacenamiento.

A continuación, se procesan los distintos cálculos requeridos por la transacción.

Una vez que se han procesado todos los pasos requeridos por la transacción, y asumiendo que no hay ningún estado inválido, el estado se finaliza determinando la cantidad de gas no utilizado que se debe reembolsar al remitente. Además del gas no utilizado, también se devuelve al remitente una cantidad del "saldo de devolución" que hemos descrito anteriormente.

Una vez que se reembolsa al remitente:

- el Ether del gas se entrega al minero
- el gas utilizado por la transacción se añade al contador de gas del bloque (que lleva la cuenta del gas total utilizado por todas las transacciones del bloque, y es útil cuando se valida un bloque)
- se eliminan todas las cuentas del conjunto de autodestrucción (si las hay)

Finalmente, nos queda el nuevo estado y un conjunto de los registros creados por la transacción.

Ahora que hemos cubierto los fundamentos de la ejecución de transacciones, veamos algunas de las diferencias entre las transacciones de creación de contratos y las llamadas de mensajes.

Contract creation

Recall that in Ethereum, there are two types of accounts: contract accounts and externally owned accounts. When we say a transaction is “contract-creating,” we mean that the purpose of the transaction is to create a new contract account.

In order to create a new contract account, we first declare the address of the new account using a special formula. Then we initialize the new account by:

- Setting the nonce to zero
- If the sender sent some amount of Ether as *value* with the transaction, setting the account balance to that value
- Deducting the value added to this new account’s balance from the sender’s balance
- Setting the storage as empty

- Setting the contract’s codeHash as the hash of an empty string

Once we initialize the account, we can actually create the account, using the **init code** sent with the transaction (see the “Transaction and messages” section for a refresher on the init code). What happens during the execution of this init code is varied. Depending on the constructor of the contract, it might update the account’s storage, create other contract accounts, make other message calls, etc.

As the code to initialize a contract is executed, it uses gas. **The transaction is not allowed to use up more gas than the remaining gas. If it does, the execution will hit an out-of-gas (OOG) exception and exit. If the transaction exits due to an out-of-gas exception, then the state is reverted to the point immediately prior to transaction. The sender is *not* refunded the gas that was spent before running out.**

Boo hoo.

However, if the sender sent any Ether value with the transaction, the Ether value will be refunded even if the contract creation fails. Phew!

If the initialization code executes successfully, a final contract-creation cost is paid. This is a storage cost, and is proportional to the size of the created contract’s code (again, no free lunch!) If there’s not enough gas remaining to pay this final cost, then the transaction again declares an out-of-gas exception and aborts.

If all goes well and we make it this far without exceptions, then any remaining unused gas is refunded to the original sender of the transaction, and the altered state is now allowed to persist!

Hooray!

Creación de contratos

Recordemos que en Ethereum hay dos tipos de cuentas: las cuentas de contrato y las cuentas de propiedad externa. Cuando decimos que una transacción es de "creación de contrato", queremos decir que el propósito de la transacción es crear una nueva cuenta de contrato.

Para crear una nueva cuenta de contrato, primero declaramos la dirección de la nueva cuenta utilizando una fórmula especial. A continuación, inicializamos la nueva cuenta mediante:

- Poniendo el nonce a cero

- Si el remitente envió alguna cantidad de Ether como *valor* con la transacción, estableciendo el saldo de la cuenta a ese valor
- Deduciendo el valor añadido al saldo de esta nueva cuenta del saldo del remitente
- Establecer el almacenamiento como vacío
- Estableciendo el codeHash del contrato como el hash de una cadena vacía

Una vez que inicializamos la cuenta, podemos crearla realmente, utilizando el **código de init** enviado con la transacción (ver la sección "Transacción y mensajes" para refrescar el código de init). Lo que ocurre durante la ejecución de este código init es variado. Dependiendo del constructor del contrato, podría actualizar el almacenamiento de la cuenta, crear otras cuentas del contrato, hacer otras llamadas a mensajes, etc.

Mientras se ejecuta el código para inicializar un contrato, se utiliza gas. **La transacción no puede consumir más gas que el que queda.** Si lo hace, la ejecución se encontrará con una excepción de falta de gas (OOG) y saldrá. Si la transacción sale debido a una excepción de falta de gas, entonces el estado se revierte al punto inmediatamente anterior a la transacción. Al remitente no se le devuelve el gas gastado antes de agotarse.

Boo hoo.

Sin embargo, si el remitente envió algún valor de Ether con la transacción, el valor de Ether será reembolsado incluso si la creación del contrato falla. ¡Ufff!

Si el código de inicialización se ejecuta con éxito, se paga un coste final de creación de contrato. Si no hay suficiente gas para pagar este coste final, la transacción vuelve a declarar una excepción de falta de gas y se cancela.

Si todo va bien y llegamos hasta aquí sin excepciones, entonces cualquier gas restante que no se haya utilizado se devuelve al remitente original de la transacción, y el estado alterado puede persistir.

¡Hurra!

Message calls

The execution of a message call is similar to that of a contract creation, with a few differences.

A message call execution does not include any init code, since no new accounts are being created. However, it can contain input data, if this data was provided by the

transaction sender. Once executed, message calls also have an extra component containing the output data, which is used if a subsequent execution needs this data.

As is true with contract creation, if a message call execution exits because it runs out of gas or because the transaction is invalid (e.g. stack overflow, invalid jump destination, or invalid instruction), none of the gas used is refunded to the original caller. Instead, all of the remaining unused gas is consumed, and the state is reset to the point immediately prior to balance transfer.

Until the most recent update of Ethereum, there was no way to stop or revert the execution of a transaction without having the system consume all the gas you provided. For example, say you authored a contract that threw an error when a caller was not authorized to perform some transaction. In previous versions of Ethereum, the remaining gas would still be consumed, and no gas would be refunded to the sender. **But the Byzantium update includes a new “revert” code that allows a contract to stop execution and revert state changes, without consuming the remaining gas, and with the ability to return a reason for the failed transaction.** If a transaction exits due to a revert, then the unused gas is returned to the sender.

Llamadas de mensaje

La ejecución de una llamada de mensaje es similar a la de la creación de un contrato, con algunas diferencias.

La ejecución de una llamada de mensaje no incluye ningún código init, ya que no se están creando nuevas cuentas. Sin embargo, puede contener datos de entrada, si estos datos fueron proporcionados por el emisor de la transacción. Una vez ejecutadas, las llamadas de mensaje también tienen un componente extra que contiene los datos de salida, que se utiliza si una ejecución posterior necesita estos datos.

Al igual que ocurre con la creación de contratos, si una ejecución de llamada a un mensaje finaliza porque se queda sin gas o porque la transacción no es válida (por ejemplo, desbordamiento de pila, destino de salto no válido o instrucción no válida), no se devuelve nada del gas utilizado a la persona que realizó la llamada original. En su lugar, todo el gas restante no utilizado se consume, y el estado se restablece al punto inmediatamente anterior a la transferencia de saldo.

Hasta la actualización más reciente de Ethereum, no había forma de detener o revertir la ejecución de una transacción sin que el sistema consumiera todo el gas

que usted proporcionó. Por ejemplo, digamos que usted es el autor de un contrato que lanza un error cuando una persona que llama no está autorizada a realizar alguna transacción. En versiones anteriores de Ethereum, el gas restante se seguiría consumiendo, y no se devolvería ningún gas al emisor. **Pero la actualización de Byzantium incluye un nuevo código de "reversión" que permite a un contrato detener la ejecución y revertir los cambios de estado, sin consumir el gas restante, y con la capacidad de devolver una razón para la transacción fallida.** Si una transacción sale debido a una reversión, entonces el gas no utilizado se devuelve al remitente.

Execution model

So far, we've learned about the series of steps that have to happen for a transaction to execute from start to finish. Now, we'll look at how the transaction actually executes within the VM.

The part of the protocol that actually handles processing the transactions is Ethereum's own virtual machine, known as the Ethereum Virtual Machine (EVM).

The EVM is a Turing complete virtual machine, as defined earlier. The only limitation the EVM has that a typical Turing complete machine does not is that the EVM is intrinsically bound by gas. Thus, the total amount of computation that can be done is intrinsically limited by the amount of gas provided.

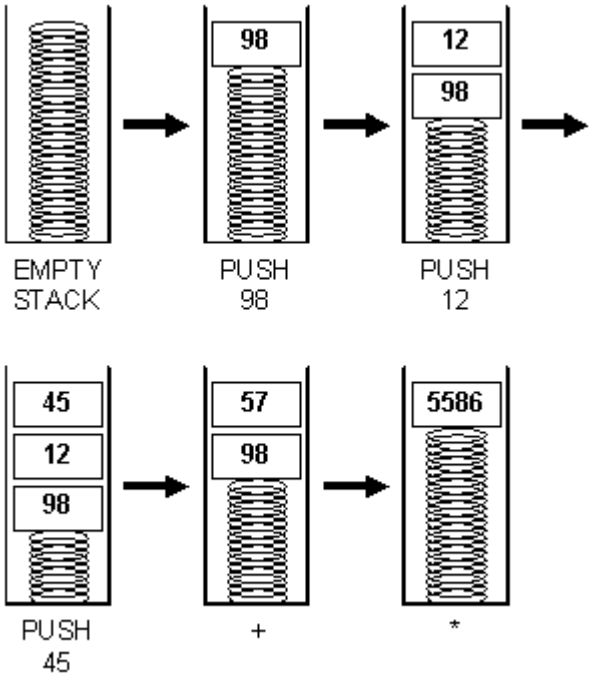
Modelo de ejecución

Hasta ahora, hemos aprendido sobre la serie de pasos que tienen que ocurrir para que una transacción se ejecute de principio a fin. Ahora, vamos a ver cómo la transacción realmente se ejecuta dentro de la VM.

- **La parte del protocolo que realmente maneja el procesamiento de las transacciones es la propia máquina virtual de Ethereum, conocida como la Máquina Virtual de Ethereum (EVM).**

La EVM es una máquina virtual completa de Turing, como se ha definido anteriormente. La única limitación que tiene la EVM que no tiene una máquina completa de Turing típica es que la EVM está intrínsecamente limitada por el gas.

Por lo tanto, la cantidad total de computación que se puede realizar está intrínsecamente limitada por la cantidad de gas suministrado.



Source: CMU

Moreover, the EVM has a stack-based architecture. A stack machine is a computer that uses a last-in, first-out stack to hold temporary values.

The size of each stack item in the EVM is 256-bit, and the stack has a maximum size of 1024.

The EVM has memory, where items are stored as word-addressed byte arrays. Memory is volatile, meaning it is not permanent.

The EVM also has storage. Unlike memory, storage is non-volatile and is maintained as part of the system state. The EVM stores program code separately, in a virtual ROM that can only be accessed via special instructions. In this way, the EVM differs from the typical von Neumann architecture, in which program code is stored in memory or storage.

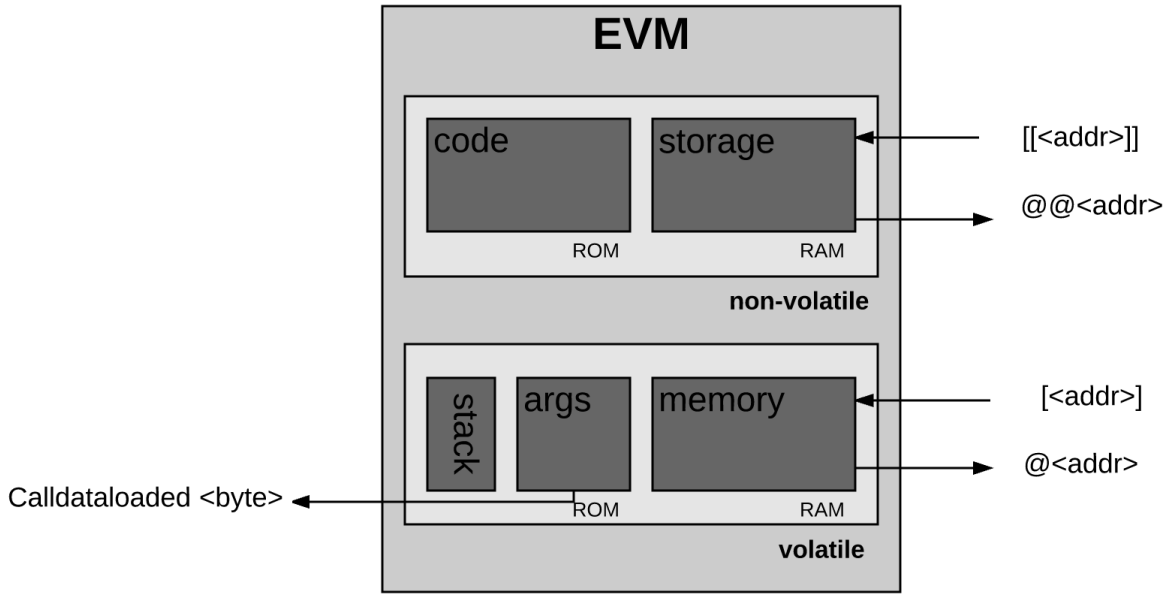
Fuente: CMU

Además, la EVM tiene una arquitectura basada en la pila. Una máquina de pila es un ordenador que utiliza una pila de último en entrar, primero en salir para mantener valores temporales.

El tamaño de cada elemento de la pila en la EVM es de 256 bits, y la pila tiene un tamaño máximo de 1024.

La EVM tiene memoria, donde los elementos se almacenan como matrices de bytes con direcciones de palabras. La memoria es volátil, es decir, no es permanente.

El EVM también tiene almacenamiento. A diferencia de la memoria, el almacenamiento no es volátil y se mantiene como parte del estado del sistema. El EVM almacena el código del programa por separado, en una ROM a la que sólo se puede acceder mediante instrucciones especiales. De este modo, el EVM se diferencia de la típica arquitectura von Neumann, en la que el código del programa se almacena en la memoria o en el almacenamiento.



The EVM also has its own language: “EVM bytecode.” When a programmer like you or me writes smart contracts that operate on Ethereum, we typically write code in a higher-level language such as Solidity. We can then compile that down to EVM bytecode that the EVM can understand.

Okay, now on to execution.

Before executing a particular computation, the processor makes sure that the following information is available and valid:

- System state
- Remaining gas for computation

- Address of the account that owns the code that is executing
- Address of the sender of the transaction that originated this execution
- Address of the account that caused the code to execute (could be different from the original sender)
- Gas price of the transaction that originated this execution
- Input data for this execution
- Value (in Wei) passed to this account as part of the current execution
- Machine code to be executed
- Block header of the current block
- Depth of the present message call or contract creation stack

At the start of execution, memory and stack are empty and the program counter is zero.

PC: 0 STACK: [] MEM: [], STORAGE: {}

The EVM then executes the transaction recursively, computing the **system state** and the **machine state** for each loop. The system state is simply Ethereum's global state. The machine state is comprised of:

- gas available
- program counter
- memory contents
- active number of words in memory
- stack contents.

Stack items are added or removed from the leftmost portion of the series.

On each cycle, the appropriate gas amount is reduced from the remaining gas, and the program counter increments.

At the end of each loop, there are three possibilities:

1. The machine reaches an exceptional state (e.g. insufficient gas, invalid instructions, insufficient stack items, stack items would overflow above 1024, invalid JUMP/JUMPI destination, etc.) and so must be halted, with any changes discarded
2. The sequence continues to process into the next loop

3. The machine reaches a controlled halt (the end of the execution process)

Assuming the execution doesn't hit an exceptional state and reaches a "controlled" or normal halt, the machine generates the resultant state, the remaining gas after this execution, the accrued substate, and the resultant output.

Phew. We got through one of the most complex parts of Ethereum. Even if you didn't fully comprehend this part, that's okay. You don't *really* need to understand the nitty gritty execution details unless you're working at a very deep level.

El EVM también tiene su propio lenguaje: "EVM bytecode". Cuando un programador como tú o yo escribe contratos inteligentes que operan en Ethereum, normalmente escribimos código en un lenguaje de alto nivel como Solidity. Luego podemos compilarlo en bytecode EVM que el EVM puede entender.

Bien, ahora pasamos a la ejecución.

Antes de ejecutar un determinado cálculo, el procesador se asegura de que la siguiente información está disponible y es válida:

- Estado del sistema
- Gas restante para el cómputo
- Dirección de la cuenta que posee el código que se está ejecutando
- Dirección del remitente de la transacción que originó esta ejecución
- Dirección de la cuenta que ha provocado la ejecución del código (puede ser diferente del remitente original)
- Precio del gas de la transacción que originó esta ejecución
- Datos de entrada para esta ejecución
- Valor (en Wei) pasado a esta cuenta como parte de la ejecución actual
- Código de la máquina que se va a ejecutar
- Cabecera del bloque actual
- Profundidad de la pila de llamada a mensajes o de creación de contratos actual

Al inicio de la ejecución, la memoria y la pila están vacías y el contador del programa es cero.

PC: 0 PILA: [] MEM: [], ALMACENAMIENTO: {}

A continuación, el EVM ejecuta la transacción recursivamente, calculando el **estado del sistema** y el **estado de la máquina** en cada bucle. El estado del sistema es simplemente el estado global de Ethereum. El estado de la máquina se compone de

- gas disponible.
- contador de programa.
- contenido de la memoria.
- número activo de palabras en la memoria.
- contenido de la pila.

Los elementos de la pila se añaden o eliminan de la parte más a la izquierda de la serie.

En cada ciclo, la cantidad de gas apropiada se reduce del gas restante, y el contador de programa se incrementa.

Al final de cada bucle, hay tres posibilidades:

1. La máquina alcanza un estado excepcional (por ejemplo, gas insuficiente, instrucciones inválidas, elementos de pila insuficientes, los elementos de pila se desbordarían por encima de 1024, destino JUMP/JUMPI inválido, etc.) y por lo tanto debe detenerse, con cualquier cambio descartado
2. La secuencia continúa el proceso hacia el siguiente bucle
3. La máquina llega a una parada controlada (el final del proceso de ejecución)

Suponiendo que la ejecución no llegue a un estado excepcional y alcance una parada "controlada" o normal, la máquina genera el estado resultante, el gas restante después de esta ejecución, el subestado acumulado y la salida resultante.

Uf. Hemos superado una de las partes más complejas de Ethereum. Incluso si usted no comprendió completamente esta parte, eso está bien. No necesitas *realmente* entender los detalles de la ejecución a menos que estés trabajando a un nivel muy profundo.

How a block gets finalized

Finally, let’s look at how a block of many transactions gets finalized.

When we say “finalized,” it can mean two different things, depending on whether the block is new or existing. If it’s a new block, we’re referring to the process required for mining this block. If it’s an existing block, then we’re talking about the process of

validating the block. In either case, there are four requirements for a block to be “finalized”:

1) Validate (or, if mining, determine) ommers Each ommer block within the block header must be a valid header and be within the sixth generation of the present block.

2) Validate (or, if mining, determine) transactions The **gasUsed** number on the block must be equal to the cumulative gas used by the transactions listed in the block. (Recall that when executing a transaction, we keep track of the block gas counter, which keeps track of the total gas used by all transactions in the block).

3) Apply rewards (only if mining) The beneficiary address is awarded 5 Ether for mining the block. (Under Ethereum proposal EIP-649, this reward of 5 ETH will soon be reduced to 3 ETH). Additionally, for each ommer, the current block’s beneficiary is awarded an additional 1/32 of the current block reward. Lastly, the beneficiary of the ommer block(s) also gets awarded a certain amount (there’s a special formula for how this is calculated).

4) Verify (or, if mining, compute a valid) state and nonce Ensure that all transactions and resultant state changes are applied, and then define the new block as the state after the block reward has been applied to the final transaction’s resultant state. Verification occurs by checking this final state against the state trie stored in the header.

Cómo se finaliza un bloque

Por último, veamos cómo se finaliza un bloque de muchas transacciones.

Cuando decimos "finalizado", puede significar dos cosas diferentes, dependiendo de si el bloque es nuevo o existente. Si se trata de un bloque nuevo, nos referimos al proceso necesario para minar este bloque. Si se trata de un bloque existente, nos referimos al proceso de validación del bloque. En cualquier caso, hay cuatro requisitos para que un bloque sea "finalizado":

1) Validar (o, en caso de minado, determinar) los ommers Cada bloque ommer dentro de la cabecera del bloque debe ser una cabecera válida y estar dentro de la sexta generación del bloque actual.

2) Validar (o, en caso de minería, determinar) las transacciones. El número **gasUsed** del bloque debe ser igual al gas acumulado utilizado por las transacciones que figuran en el bloque. (Recordemos que cuando se ejecuta una transacción,

hacemos un seguimiento del contador de gas del bloque, que lleva la cuenta del gas total utilizado por todas las transacciones del bloque).

3) Aplicar las recompensas (sólo si se minan) La dirección beneficiaria recibe 5 Ether por minar el bloque. (Según la propuesta de Ethereum EIP-649, esta recompensa de 5 ETH se reducirá pronto a 3 ETH). Además, por cada ommer, el beneficiario del bloque actual recibe 1/32 adicional de la recompensa del bloque actual. Por último, el beneficiario del bloque ommer también recibe una cantidad determinada (hay una fórmula especial para calcularla).

4) Verificar (o, si se trata de minería, calcular un estado y un nonce válidos. Asegurarse de que todas las transacciones y los cambios de estado resultantes se aplican, y luego definir el nuevo bloque como el estado después de que la recompensa del bloque se haya aplicado al estado resultante de la transacción final. La verificación se realiza cotejando este estado final con la prueba de estado almacenada en la cabecera.

Mining proof of work

The “*Blocks*” section briefly addressed the concept of block difficulty. The algorithm that gives meaning to block difficulty is called Proof of Work (PoW).

Ethereum’s proof-of-work algorithm is called “Ethash” (previously known as Dagger-Hashimoto).

The algorithm is formally defined as:

Prueba de trabajo de la minería

En la sección "*Bloques*" se abordó brevemente el concepto de dificultad de los bloques. El algoritmo que da sentido a la dificultad de los bloques se llama Prueba de Trabajo (PoW).

El algoritmo de prueba de trabajo de Ethereum se llama "Ethash" (antes conocido como Dagger-Hashimoto).

El algoritmo se define formalmente como:

$$(m, n) = \mathtt{PoW}(H_{\not n}, H_n, \mathbf{d})$$

*where **m** is the **mixHash**, **n** is the **nonce**, **Hn** is the new block's header (excluding the **nonce** and **mixHash** components, which have to be computed), **Hn** is the nonce of the block header, and **d** is the DAG, which is a large data set.*

*donde **m** es el* ***mixHash**,**n** es el* ***nonce**,**Hn** es la nueva cabecera del bloque (excluyendo los **nonce** y **mixHash*** *componentes, que tienen que ser calculados),**Hn** es el nonce de la cabecera del bloque, y **d** es el DAG, que es un gran conjunto de datos.*

In the "*Blocks*" section, we talked about the various items that exist in a block header. Two of those components were called the **mixHash** and the **nonce**. As you may recall:

- **mixHash** is a hash that, when combined with the nonce, proves that this block has carried out enough computation
- **nonce** is a hash that, when combined with the mixHash, proves that this block has carried out enough computation

The PoW function is used to evaluate these two items.

How exactly the **mixHash** and **nonce** are calculated using the PoW function is somewhat complex, and something we can delve deeper into in a separate post. But at a high level, it works like this:

A "seed" is calculated for each block. This seed is different for every "epoch," where each epoch is 30,000 blocks long. For the first epoch, the seed is the hash of a series of 32 bytes of zeros. For every subsequent epoch, it is the hash of the previous seed hash. Using this seed, a node can calculate a pseudo-random "cache."

This cache is incredibly useful because it enables the concept of "light nodes," which we discussed previously in this post. The purpose of light nodes is to afford certain nodes the ability to efficiently verify a transaction without the burden of storing the entire blockchain dataset. A light node can verify the validity of a transaction based solely on this cache, because the cache can regenerate the specific block it needs to verify.

Using the cache, a node can generate the DAG "dataset," where each item in the dataset depends on a small number of pseudo-randomly-selected items from the cache. In order to be a miner, you must generate this full dataset; all full clients and miners store this dataset, and the dataset grows linearly with time.

Miners can then take random slices of the dataset and put them through a mathematical function to hash them together into a "**mixHash.**" A miner will

repeatedly generate a **mixHash** until the output is below the desired target **nonce**. When the output meets this requirement, this nonce is considered valid and the block can be added to the chain

En la sección "*Bloques*", hablamos de los distintos elementos que existen en la cabecera de un bloque. Dos de esos componentes se llamaban el **mixHash** y el **nonce**. Como recordará:

- **mixHash** es un hash que, cuando se combina con el nonce, demuestra que este bloque ha realizado suficientes cálculos
- El **nonce** es un hash que, cuando se combina con el mixHash, demuestra que este bloque ha realizado suficientes cálculos

La función PoW se utiliza para evaluar estos dos elementos.

Cómo se calculan exactamente el **mixHash** y el **nonce** utilizando la función PoW es algo complejo, y es algo en lo que podemos profundizar en otro post. Pero a alto nivel, funciona así:

Se calcula una "semilla" para cada bloque. Esta semilla es diferente para cada "época", donde cada época tiene 30.000 bloques. Para la primera época, la semilla es el hash de una serie de 32 bytes de ceros. Para cada época posterior, es el hash de la semilla anterior. Con esta semilla, un nodo puede calcular una "caché" pseudoaleatoria.

Esta caché es increíblemente útil porque permite el concepto de "nodos ligeros", del que ya hablamos en este post. El propósito de los nodos ligeros es permitir a ciertos nodos la capacidad de verificar eficientemente una transacción sin la carga de almacenar todo el conjunto de datos de la cadena de bloques. Un nodo ligero puede verificar la validez de una transacción basándose únicamente en esta caché, ya que ésta puede regenerar el bloque específico que necesita verificar.

Utilizando la caché, un nodo puede generar el "conjunto de datos" DAG, donde cada elemento del conjunto de datos depende de un pequeño número de elementos seleccionados de forma pseudo-aleatoria de la caché. Para ser un minero, debe generar este conjunto de datos completo; todos los clientes y mineros completos almacenan este conjunto de datos, y el conjunto de datos crece linealmente con el tiempo.

Los mineros pueden entonces tomar trozos aleatorios del conjunto de datos y someterlos a una función matemática para hacer un hash que los una en un "**mixHash.**" Un minero generará repetidamente un **mixHash** hasta que la salida

esté por debajo del objetivo deseado **noce**. Cuando el resultado cumple este requisito, este nonce se considera válido y el bloque puede añadirse a la cadena.

Mining as a security mechanism

Overall, the purpose of the PoW is to prove, in a cryptographically secure way, that a particular amount of computation has been expended to generate some output (i.e. the **nonce**). **This is because there is no better way to find a nonce that is below the required threshold other than to enumerate all the possibilities.** The outputs of repeatedly applying the hash function have a uniform distribution, and so we can be assured that, on average, **the time needed to find such a nonce depends on the difficulty threshold.** The higher the difficulty, the longer it takes to solve for the nonce. In this way, **the PoW algorithm gives meaning to the concept of difficulty, which is used to enforce blockchain security.**

What do we mean by blockchain security? It's simple: we want to create a blockchain that EVERYONE trusts. As we discussed previously in this post, if more than one chain existed, users would lose trust, because they would be unable to reasonably determine which chain was the "valid" chain. In order for a group of users to accept the underlying state that is stored on a blockchain, we need a single canonical blockchain that a group of people believes in.

This is exactly what the PoW algorithm does: it ensures that a particular blockchain will remain canonical into the future, making it incredibly difficult for an attacker to create new blocks that overwrite a certain part of history (e.g. by erasing transactions or creating fake transactions) or maintain a fork. To have their block validated first, an attacker would need to consistently solve for the nonce faster than anyone else in the network, such that the network believes their chain is the heaviest chain (based on the principles of the GHOST protocol we mentioned earlier). This would be impossible unless the attacker had more than half of the network mining power, a scenario known as the majority 51% attack.

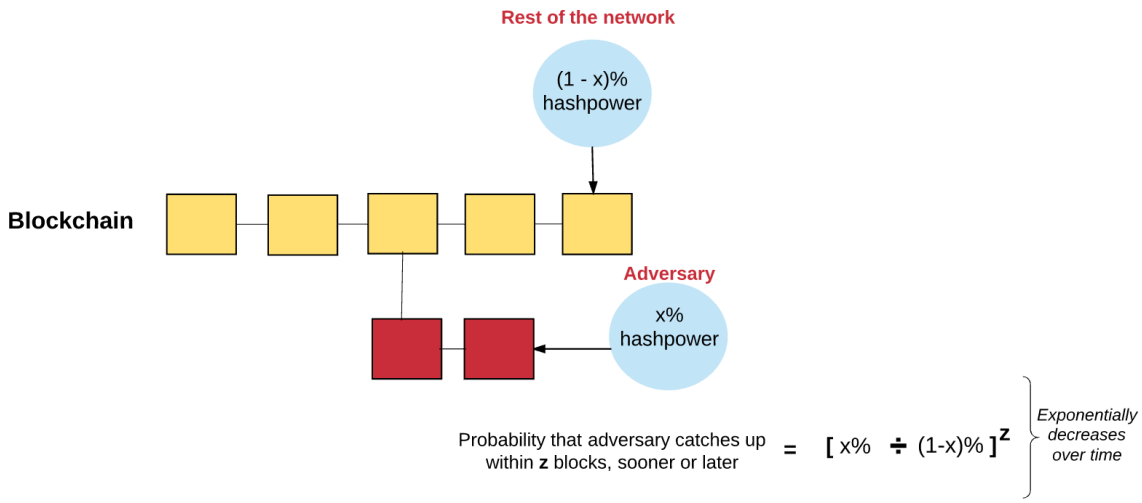
La minería como mecanismo de seguridad

- En general, el propósito del PoW es probar, de una manera criptográficamente segura, que una cantidad particular de computación ha sido gastada para generar alguna salida (es decir, el **nonce**). **Esto se debe a que no hay una forma mejor de encontrar un nonce que esté por debajo del umbral requerido que no sea enumerar todas las posibilidades.** Los resultados de aplicar repetidamente la función hash tienen una distribución uniforme, y por lo

tanto podemos estar seguros de que, en promedio, **el tiempo necesario para encontrar tal nonce depende del umbral de dificultad.** Cuanto mayor sea la dificultad, más tiempo se necesita para resolver el nonce. De este modo, **el algoritmo PoW da sentido al concepto de dificultad, que se utiliza para reforzar la seguridad de la cadena de bloques.**

¿Qué entendemos por seguridad de la cadena de bloques? Es sencillo: queremos crear una blockchain en la que TODOS confíen. Como hemos comentado anteriormente en este post, si existiera más de una cadena, los usuarios perderían la confianza, porque no podrían determinar razonablemente cuál es la cadena "válida". Para que un grupo de usuarios acepte el estado subyacente que se almacena en una blockchain, necesitamos una única blockchain canónica en la que un grupo de personas crea.

Esto es exactamente lo que hace el algoritmo PoW: asegura que una determinada blockchain seguirá siendo canónica en el futuro, haciendo increíblemente difícil para un atacante crear nuevos bloques que sobrescriban una determinada parte de la historia (por ejemplo, borrando transacciones o creando transacciones falsas) o mantener una bifurcación. Para que su bloque sea validado en primer lugar, un atacante necesitaría resolver sistemáticamente el nonce más rápido que cualquier otra persona de la red, de forma que la red crea que su cadena es la más pesada (basándose en los principios del protocolo GHOST que hemos mencionado antes). Esto sería imposible a menos que el atacante tuviera más de la mitad del poder de minería de la red, un escenario conocido como el ataque de la mayoría del 51%.



Mining as a wealth distribution mechanism

Beyond providing a secure blockchain, PoW is also a way to distribute wealth to those who expend their computation for providing this security. Recall that a miner receives a reward for mining a block, including:

- a *static block reward* of 5 ether for the “winning’” block (soon to be changed to 3 ether)
- the cost of gas expended within the block by the transactions included in the block
- an extra reward for including ommers as part of the block

In order to ensure that the use of the PoW consensus mechanism for security and wealth distribution is sustainable in the long run, Ethereum strives to instill these two properties:

- Make it accessible to as many people as possible. In other words, people shouldn’t need specialized or uncommon hardware to run the algorithm. The purpose of this is to make the wealth distribution model as open as possible so that anyone can provide any amount of compute power in return for Ether.
- Reduce the possibility for any single node (or small set) to make a disproportionate amount of profit. Any node that can make a disproportionate amount of profit means that the node has a large influence on determining the canonical blockchain. This is troublesome because it reduces network security.

In the Bitcoin blockchain network, one problem that arises in relation to the above two properties is that the PoW algorithm is a SHA256 hash function. The weakness with this type of function is that it can be solved much more efficiently using specialized hardware, also known as ASICs.

In order to mitigate this issue, Ethereum has chosen to make its PoW algorithm (Ethhash) sequentially memory-hard. This means that the algorithm is engineered so that calculating the nonce requires a lot of memory AND bandwidth. The large memory requirements make it hard for a computer to use its memory in parallel to discover multiple nonces simultaneously, and the high bandwidth requirements make it difficult for even a super-fast computer to discover multiple nonce simultaneously. This reduces the risk of centralization and creates a more level playing field for the nodes that are doing the verification.

One thing to note is that Ethereum is transitioning from a PoW consensus mechanism to something called “proof-of-stake”. This is a beastly topic of its own that we can hopefully explore in a future post. ☺

La minería como mecanismo de distribución de la riqueza

Además de proporcionar una blockchain segura, PoW es también una forma de distribuir riqueza a aquellos que gastan su computación para proporcionar esta seguridad. Recordemos que un minero recibe una recompensa por minar un bloque, que incluye

- una *recompensa de bloque estática* de 5 éteres por el bloque "ganador" (que pronto será cambiado a 3 éteres)
- el coste del gas gastado en el bloque por las transacciones incluidas en el mismo
- una recompensa extra por incluir ommers como parte del bloque

Para garantizar que el uso del mecanismo de consenso PoW para la seguridad y la distribución de la riqueza sea sostenible a largo plazo, Ethereum se esfuerza por inculcar estas dos propiedades

- Hacerlo accesible al mayor número de personas posible. En otras palabras, la gente no debería necesitar un hardware especializado o poco común para ejecutar el algoritmo. El propósito de esto es hacer que el modelo de distribución de la riqueza sea lo más abierto posible para que cualquiera pueda proporcionar cualquier cantidad de potencia de cálculo a cambio de Ether.
- Reducir la posibilidad de que un solo nodo (o un pequeño conjunto) obtenga una cantidad desproporcionada de beneficios. Cualquier nodo que pueda obtener una cantidad desproporcionada de beneficios significa que el nodo tiene una gran influencia en la determinación del blockchain canónico. Esto es problemático porque reduce la seguridad de la red.

En la red blockchain de Bitcoin, un problema que surge en relación con las dos propiedades anteriores es que el algoritmo PoW es una función hash SHA256. El punto débil de este tipo de función es que puede resolverse de forma mucho más eficiente utilizando hardware especializado, también conocido como ASIC.

Para mitigar este problema, Ethereum ha optado por hacer que su algoritmo PoW (Ethhash) sea secuencialmente difícil de memorizar. Esto significa que el algoritmo está diseñado para que el cálculo del nonce requiera mucha memoria Y ancho de banda. Los grandes requisitos de memoria hacen que sea difícil para un ordenador utilizar su memoria en paralelo para descubrir múltiples nonces simultáneamente, y los altos requisitos de ancho de banda hacen que sea difícil incluso para un

ordenador súper rápido descubrir múltiples nonce simultáneamente. Esto reduce el riesgo de centralización y crea una mayor igualdad de condiciones para los nodos que realizan la verificación.

Una cosa a tener en cuenta es que Ethereum está pasando de un mecanismo de consenso PoW a algo llamado "proof-of-stake". Este es un tema bestial por sí mismo que esperamos poder explorar en un futuro post.

Conclusion

…Phew! You made it to the end. I hope?

There’s a lot to digest in this post, I know. If it takes you multiple reads to fully understand what’s going on, that’s totally fine. I personally read the Ethereum yellow paper, white paper, and various parts of the code base many times before grokking what was going on.

Nonetheless, I hope you found this overview helpful. If you find any errors or mistakes, I’d love for you to write a private note or post it directly in the comments. I look at all of ’em, I promise ;)

And remember, I’m human (yep, it’s true) and I make mistakes. I took the time to write this post for the benefit of the community, for free. So please be constructive in your feedback, without unnecessary bashing.

Conclusión

...¡Uf! Has llegado al final. Espero...

Hay mucho que digerir en este post, lo sé. Si te toma varias lecturas para entender completamente lo que está pasando, eso está totalmente bien. Yo personalmente leí el libro amarillo de Ethereum, el libro blanco y varias partes del código base muchas veces antes de entender lo que estaba pasando.

No obstante, espero que esta visión general te resulte útil.

8 october 2021