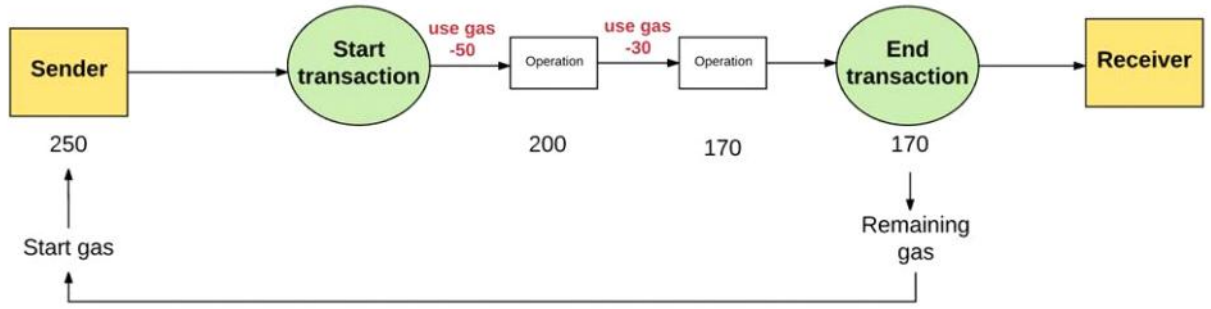
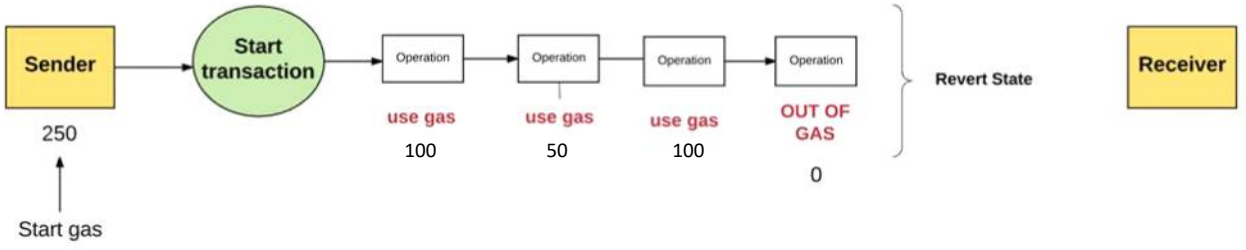
5.1.5. Gas and Payment.

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. 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 $\mathbf{10^{18}}$ 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 imit** 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 GWwei. This implies that the sender is willing to spend at most 50,000 x 20 GWwei = 1,000,000,000,000,000 Wei = 0.001 Ether to execute that transaction.

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.



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, **none of the gas is refunded to the sender.**



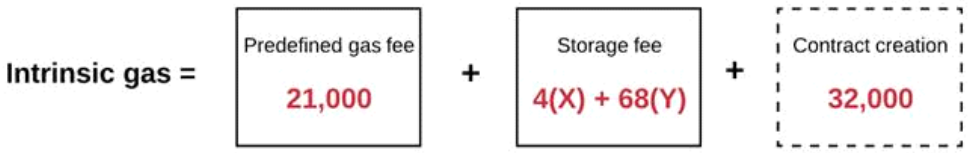
In this case, the transaction processing aborts and any state changes that occurred are reversed, such that we end up back at the state of Ethereum prior to the transaction. Additionally, a record of the transaction failing gets recorded, showing what transaction was attempted and where it failed.

And since the EVM already expended effort to run the calculations before running out of gas, logically, none of the gas is refunded to the sender.

- The transaction's gas limit must be equal to or greater than the **intrinsic gas** used by the transaction. The intrinsic gas includes:

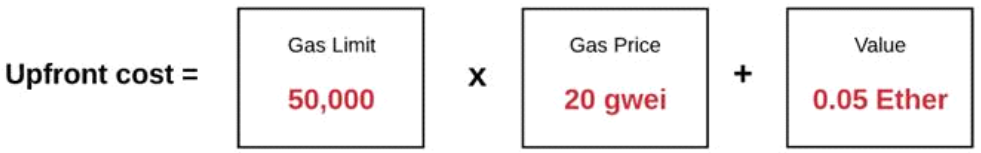
1. a predefined cost of 21,000 gas for executing the transaction
2. a gas fee for data sent with the transaction (4 gas for every byte of data or code, and 68 gas for every non-zero byte of data or code)
3. if the transaction is a contract-creating transaction, an additional 32,000 gas.

Vidinės dujos

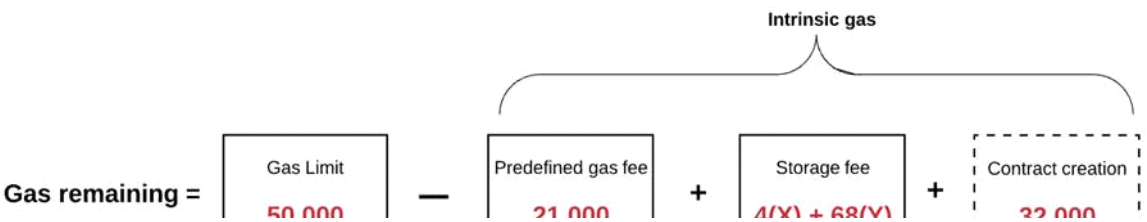


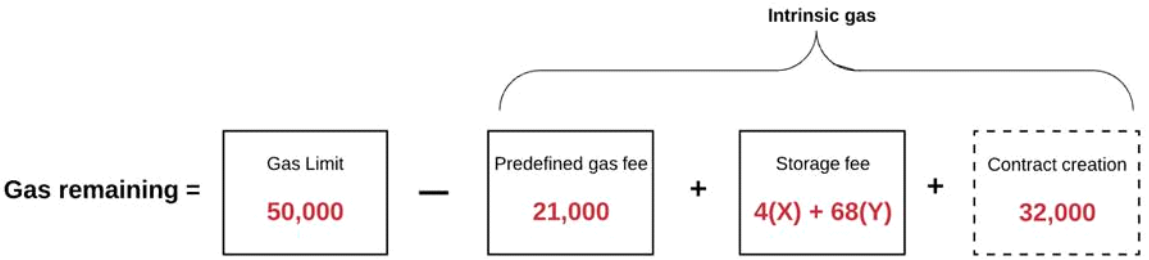
- The sender's account balance must have enough Ether to cover the **"upfront" gas costs** that the sender must pay. The calculation for the upfront gas cost is simple:
First, the transaction's **gas limit** is multiplied by the transaction's **gas price** to determine the maximum gas cost.
Then, this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

išankstinis mokestis



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.**





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.

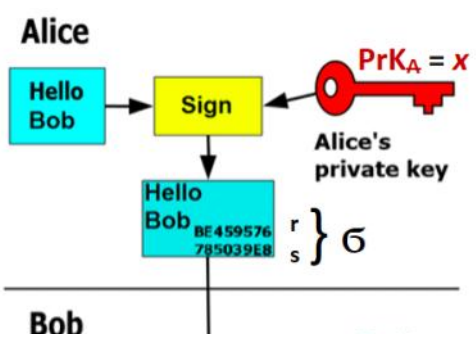
Next, the various computations required by the transaction are processed.
Once all the steps required by the transaction have been processed, and assuming there is no invalid state, the state is finalized by determining the amount of unused gas to be refunded to the sender. In addition to the unused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

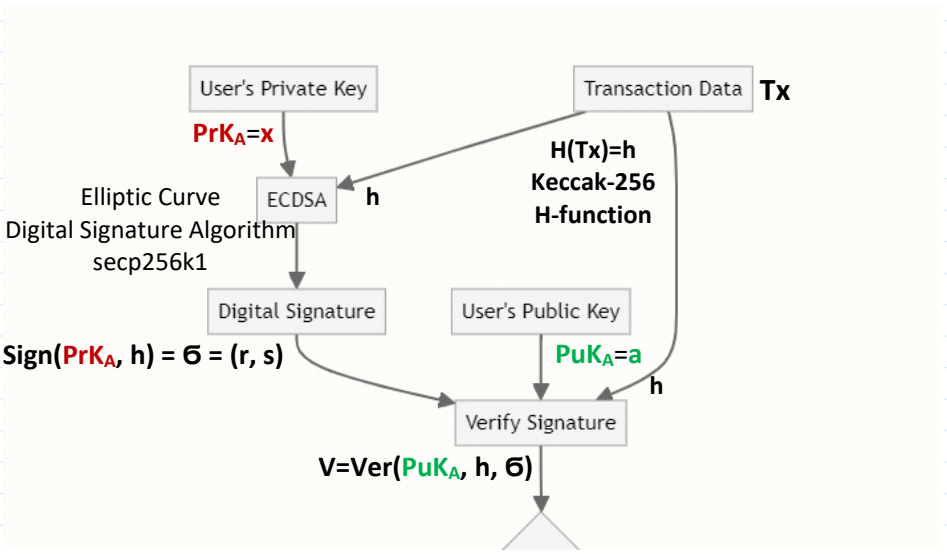
- the Ether for the gas is given to the miner-validator
- the gas used by the transaction is added to the block gas counter (which keeps track of the total gas used by all transactions in the block, and is useful when validating a block)
- all accounts in the self-destruct set (if any) are deleted

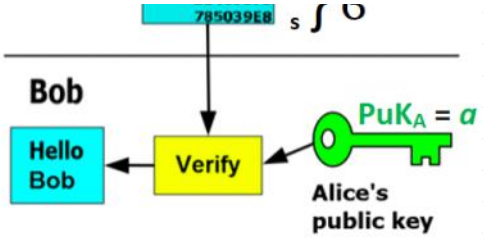
Finally, we’re left with the new state and a set of the logs created by the transaction.
Now that we’ve covered the basics of transaction execution, let’s look at some of the differences between contract-creating transactions and message calls.
From <https://preethikasireddy.medium.com/how-does-ethereum-work-anyway-22d1df506369>

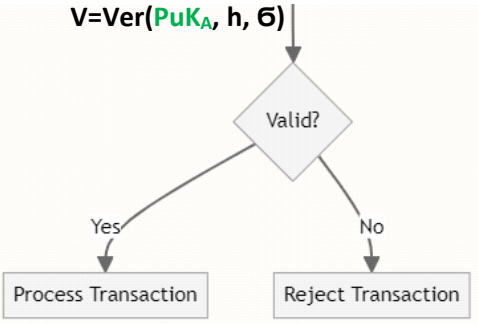
5.1.5. Signature.

Asymmetric Signing - Verification
Sign(PrK_A, m) = б = (r, s)
V=Ver(PuK_A, m, б), V∈{True, False}









Keccak-256
For signature creation **Ethereum** is using H-function **Keccak-256**.
Keccak-256 is widely used in blockchain technologies, for generating unique identifiers and ensuring data integrity being a "finger print" of data.
Keccak family, forms the basis of the SHA-3 (Secure Hash Algorithm 3) standard.

Key Features

- **Deterministic**: The same input always produces the same hash output.
- **Fixed Output Size**: Regardless of input size, the output is always 256 bits (64 hexadecimal characters).
- **Collision Resistance**: It is computationally infeasible to find two different inputs that produce the same hash.

Example Code in Python

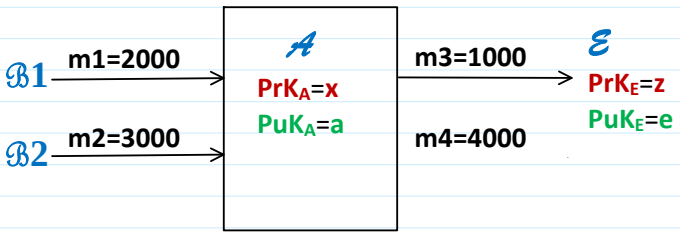
You can compute a Keccak-256 hash using the `pysha3` library in Python, for example:

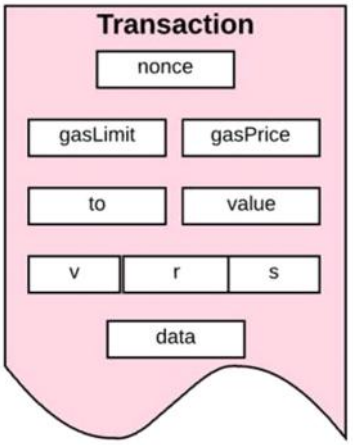
```
import hashlib
# Input data
data = "Hello, Keccak-256!"
# Compute Keccak-256 hash
hash_object = hashlib.new('sha3_256')
hash_object.update(data.encode('utf-8'))
keccak_hash = hash_object.hexdigest()
print(f"Keccak-256 Hash: {keccak_hash}")
```

For the input `"Hello, Keccak-256!"`, the output hash will look like:
`Keccak-256 Hash: 8b6f9c8e4c4b8e8f6e8c8b6f9c8e4c4b8e8f6e8c8b6f9c8e4c4b8e8f6e8c8b6`

Let us encode a simple transaction according to the template.

Transaction - **Tx** in blockchain



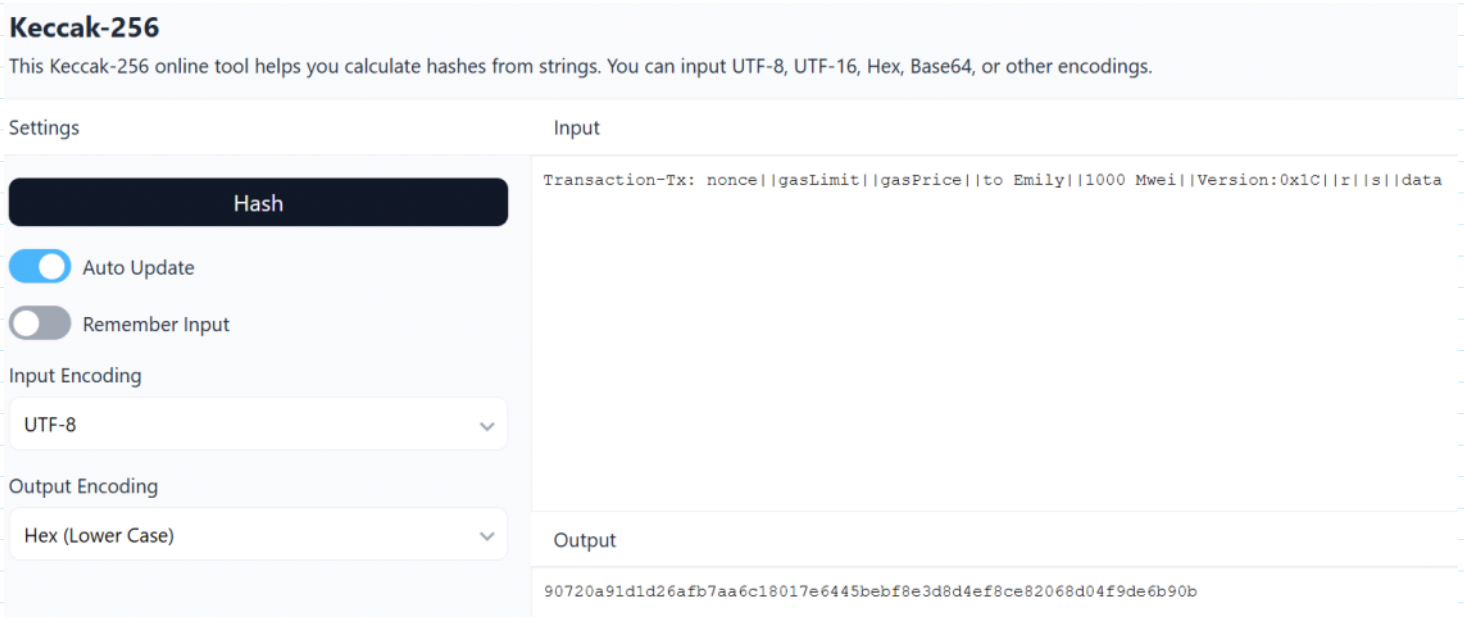


Signature field

Let us consider the depicted transaction written in symbolic form:

Transaction-Tx: nonce||gasLimit||gasPrice||to Emily||1000 Mwei||Version:0x1C||r||s||data

Then the Keccak-256 H-function can be computed using the following online tool
[Keccak-256 - Online Tools]

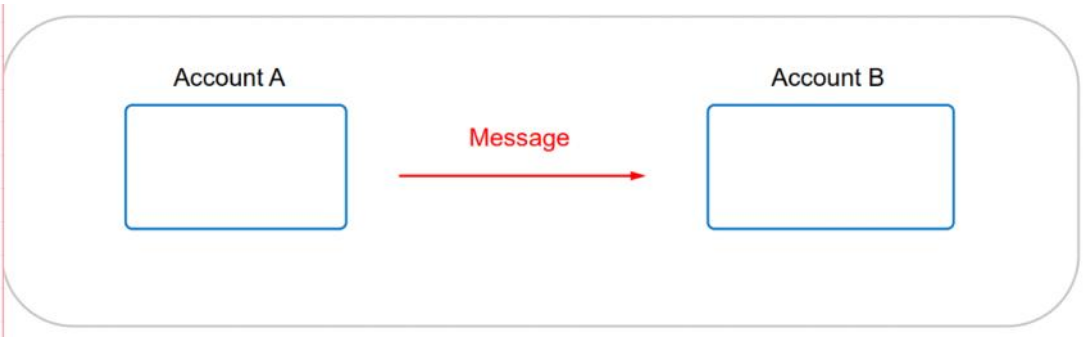


As we see the result in Output field by adding the prefix **0x** is the following:
0x90720a91d1d26afb7aa6c18017e6445bebf8e3d8d4ef8ce82068d04f9de6b90b

5.1.6. Message calls.

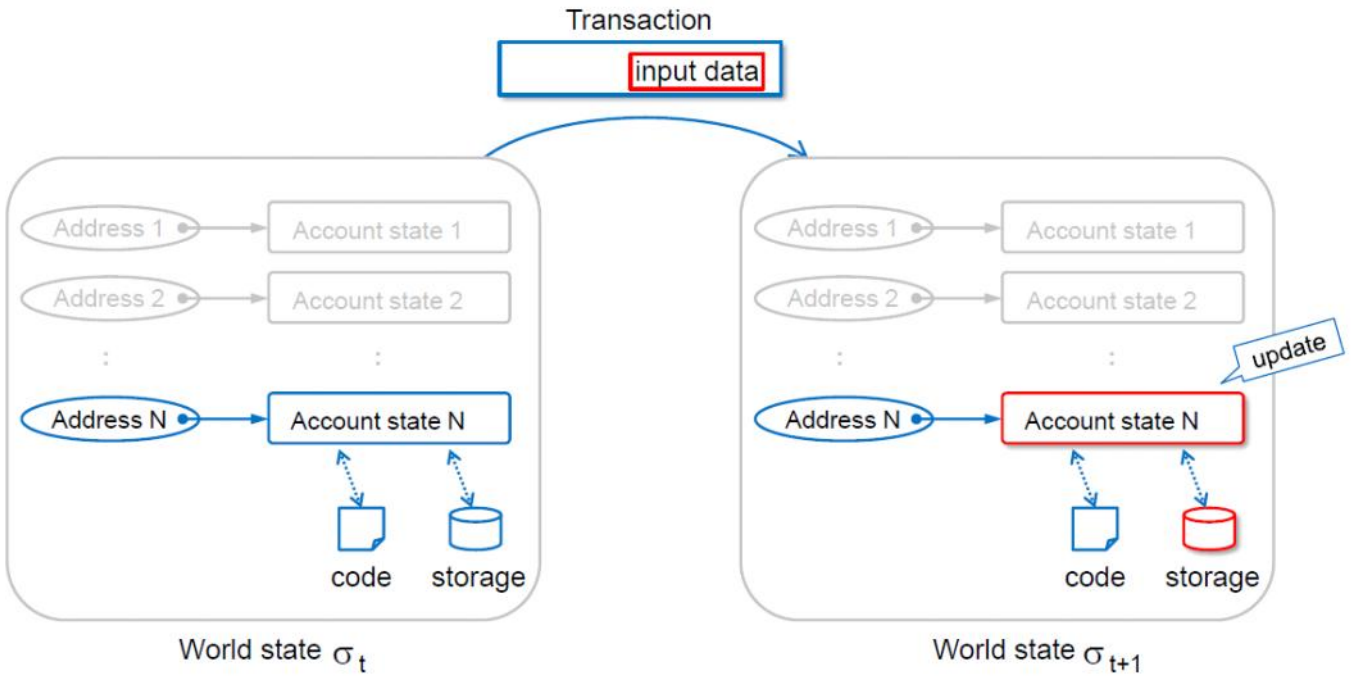
Message call transaction is a type of transaction in Ethereum that is used to interact with a smart contr act or another External Owned Account - EOA .

World State

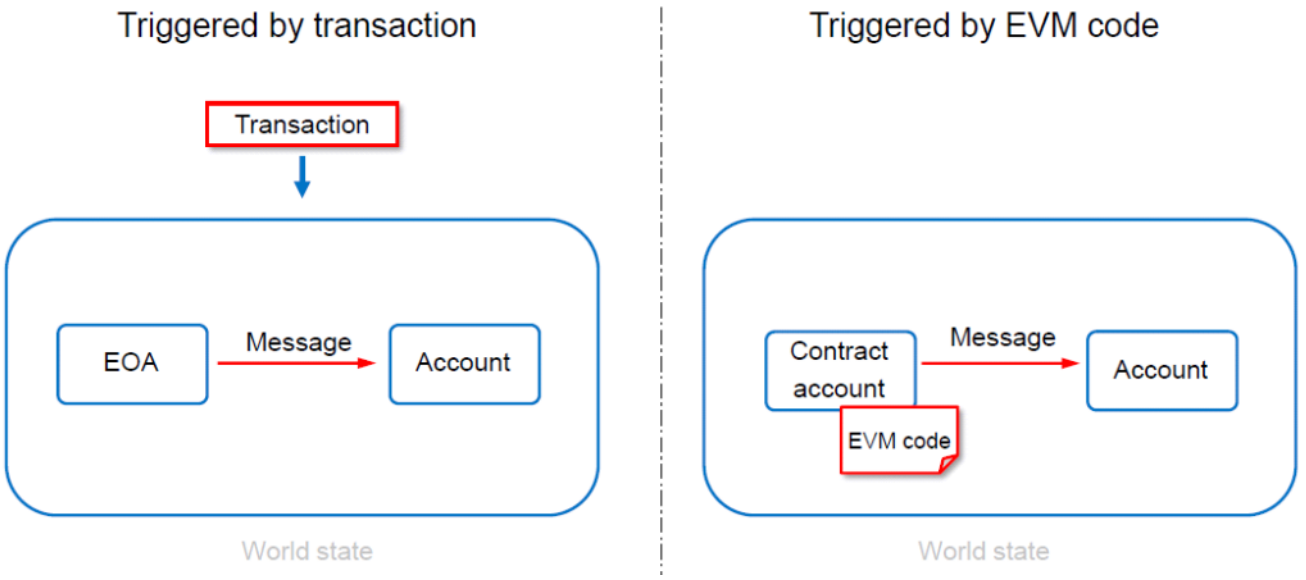


Message is passed between two Accounts.
Message is Data represented as a set of bytes and Value is specified as Ether.

Message call is initiated by an Externally Owned Account -
EOA and sends a message to the contract, triggering a function call within the contract.
From <https://www.bing.com/search?pglt=41&q=ethereum+message+calls&cvid=456c3e2325f540438c9fb561d01a6743&gs_lcrp=EgRlZGdlKgYIABBFGDkyBggAEEUYOTIICAEQ6QcY_FXSAQkxODAxN2owajGoAgCwAgA&FORM=ANNAB1&PC=U531>



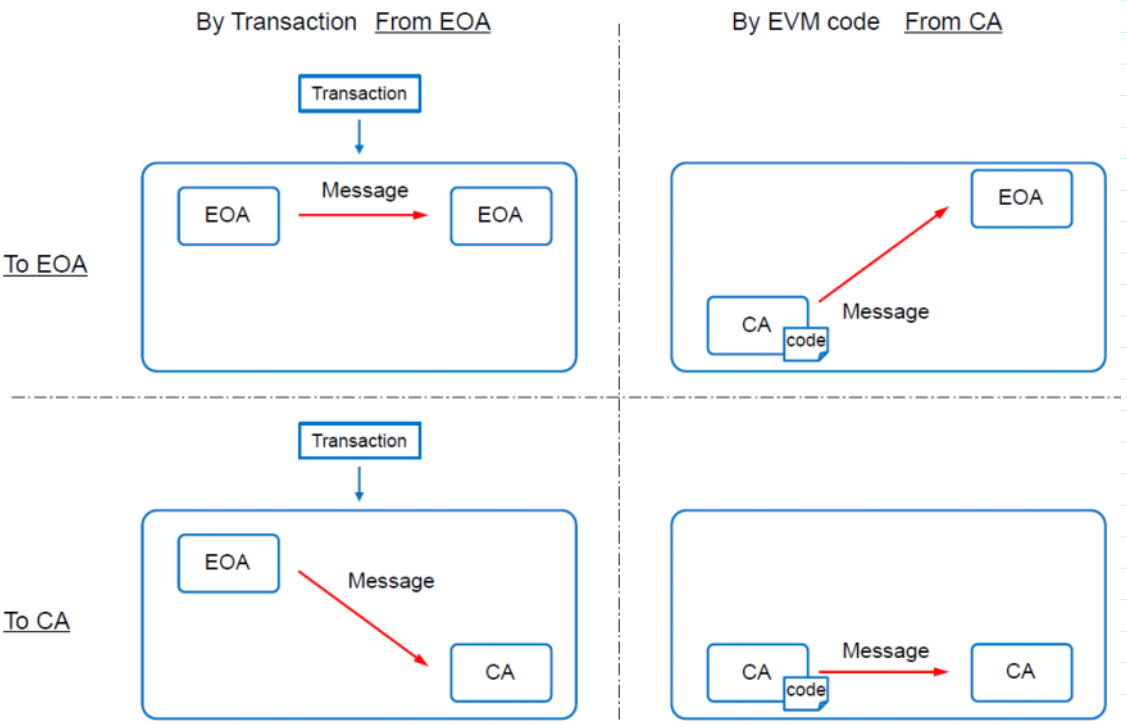
The execution of a **message call** is similar to that of a contract creation, with a few differences.
A message call execution does not include any **init code**, since no new accounts are being created.
However, it can contain **input data**, if this data was provided by the transaction sender.
Once 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.



Transaction triggers an associated message.

EVM can also send a message.

Four cases of message



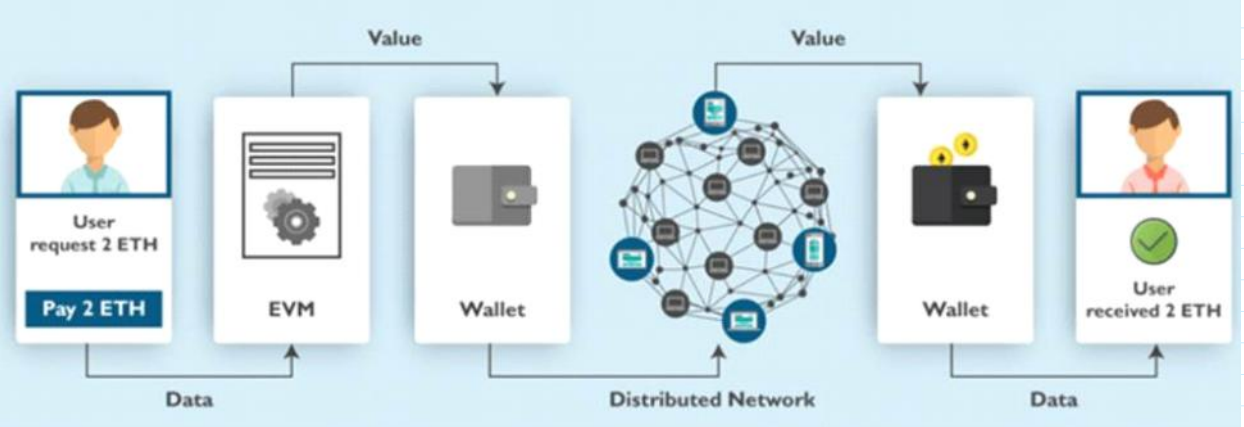
Until the most recent update of Ethereum, there was no way to stop or revert the execution of a transaction without having the system consume all the gas you provided. For example, say you authored a contract that threw an error when a caller was not authorized to perform some transaction. In previous 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.
An information concerning the Byzantine fault can be found in:

https://en.wikipedia.org/wiki/Byzantine_fault

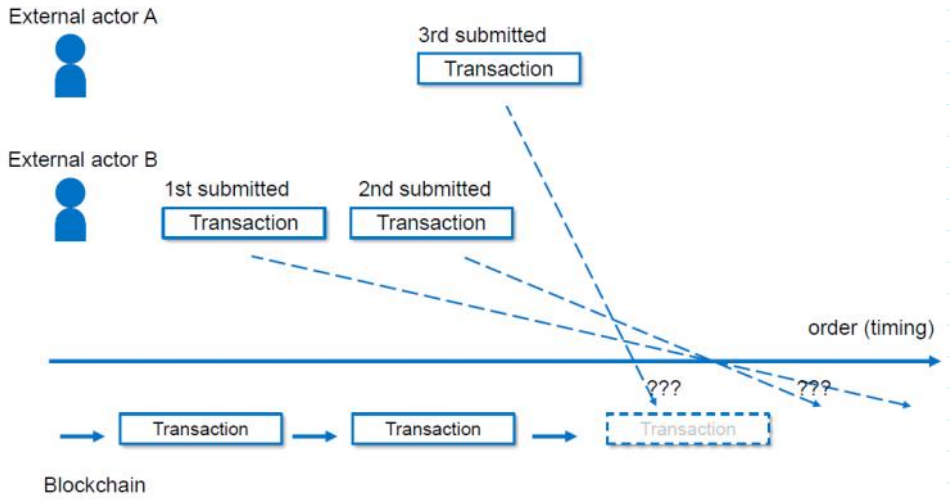
If a transaction exits due to a revert, then the unused gas is returned to the sender.

5.2.1. Ethereum blocks

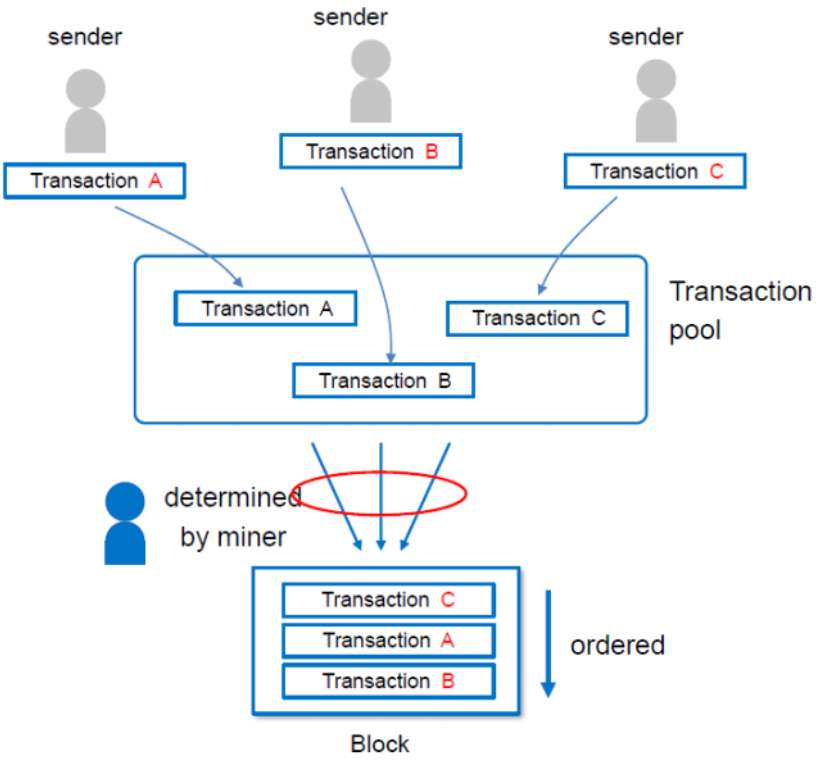
Transaction view



Ethereum transaction order is not guaranteed.



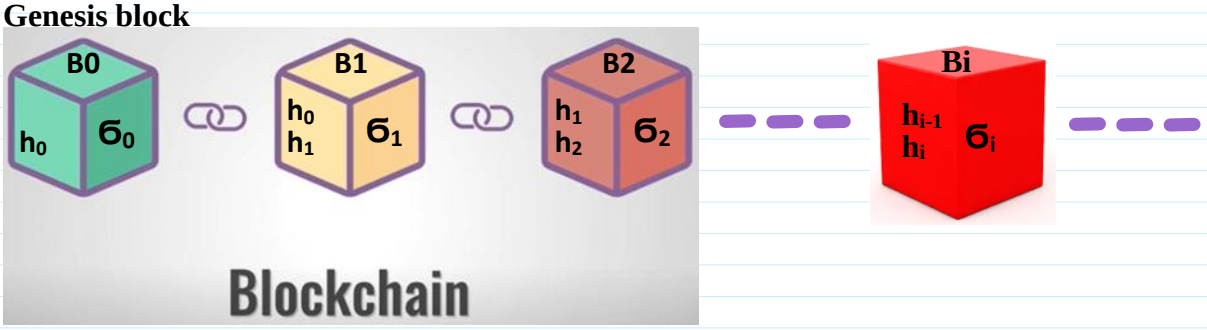
Ordering inner block



Miner can determine the order of transactions in a block.

Blockchain

Genesis block



$h_0=H(B0||)$
$h_1=H(B1||h_0)$
$h_2=H(B2||h_1)$

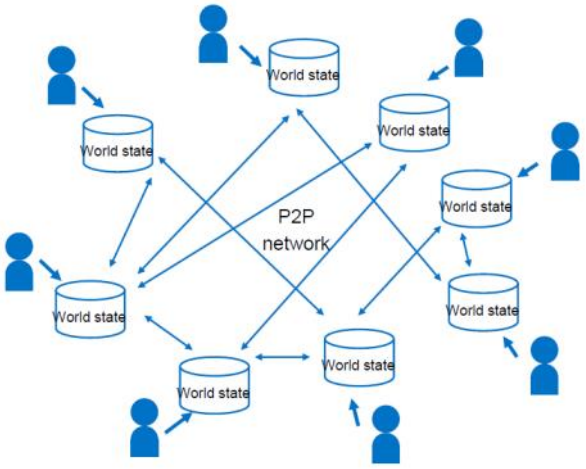
$h_i=H(Bi||h_{i-1})$

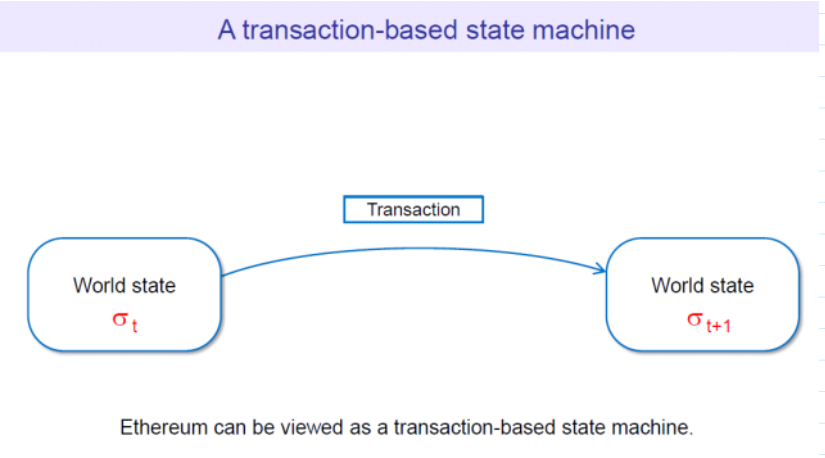
$\sigma_0 = Sign(PrK_0, h_0)$
$\sigma_1 = Sign(PrK_1, h_1)$
$\sigma_2 = Sign(PrK_2, h_2)$

$\sigma_i = Sign(PrK_i, h_i)$

Notations:
h_i - is a ***h***-value of **block Bi**
|| - is a concatenation operation of strings
Sign(,) - is a digital signing operation
σ_i - is a signature value of mined block number ***i*** consisting of two numbers denoted by (r_i, s_i)
PrK_i - Private Key of miner ***i*** for signature creation

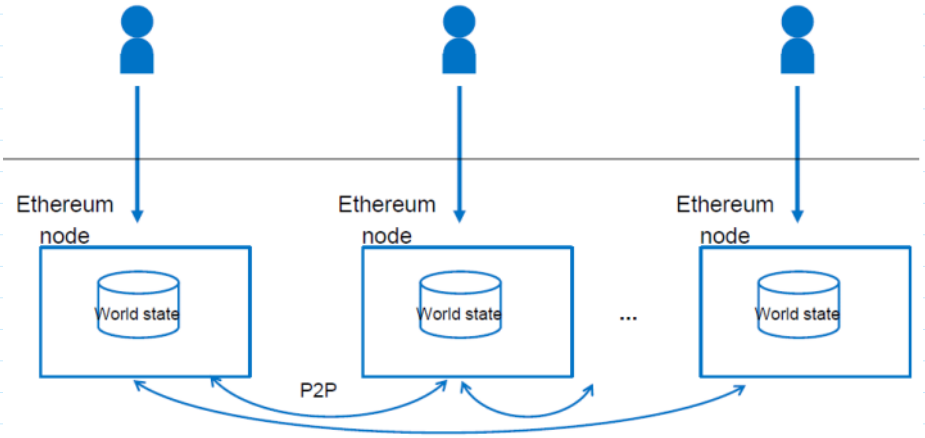
Decentralised database





A blockchain is a globally shared, decentralised, transactional database.

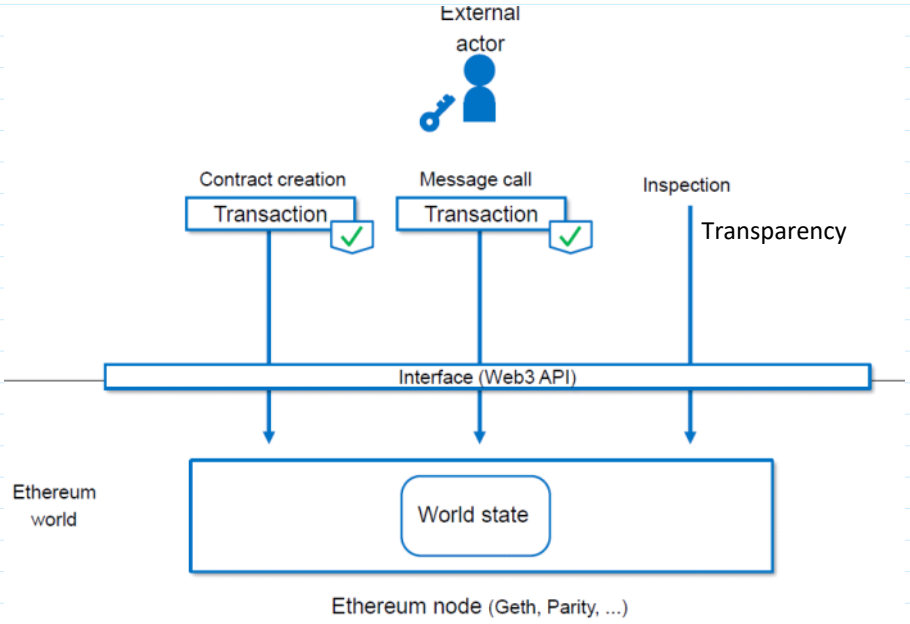
P2P network inter **nodes**



Decentralised nodes constitute Ethereum P2P network.

Interface to a **node**

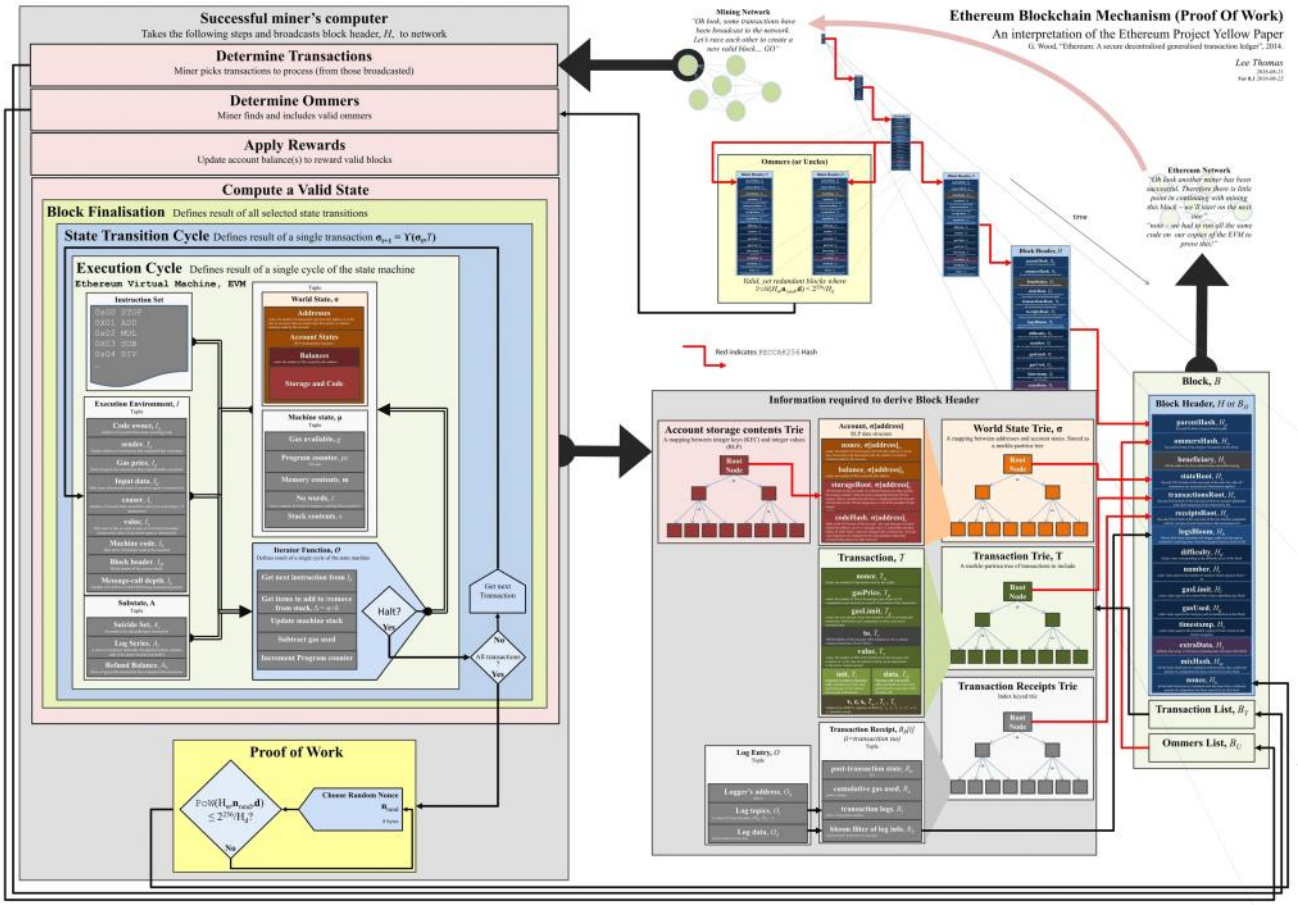
EOA - Externally Owned Account



External actors access the Ethereum world through Ethereum nodes.

Ethereum architecture formally-mathematically is presented in Ethereum **Yellow Paper**
Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain
The Ethereum Yellow Paper serves as the definitive technical document for the Ethereum protocol. It was init ially written by **Gavin Wood** and is now maintained by **Andrew Ashikhmin** along with contributions from vari ous experts globally.

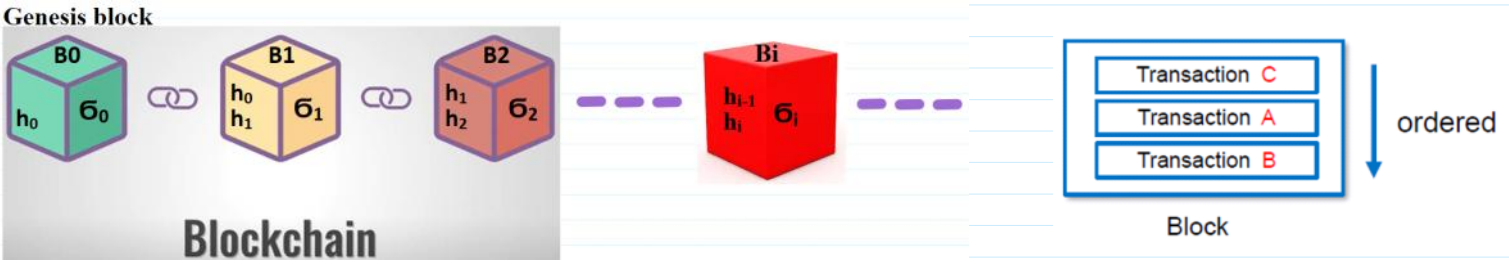
Ethereum architecture structurally is presented in:
blockchain - Ethereum block architecture - Ethereum Stack Exchange



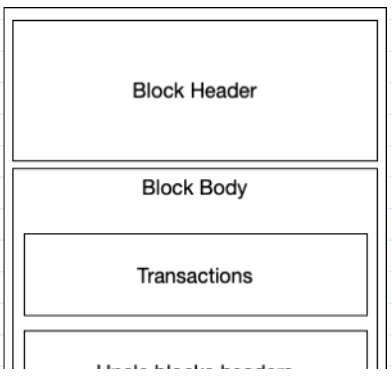
The source is Ethereum Guide: Understanding Ethereum's Structures
By: *Ricardo Santos* - VP EngineeringDate: 22/08/2023

From <https://parfin.io/en/blog/the-ethereum-guide-understanding-ethereums-structures>

Ethereum Blocks



A high level diagram of the Ethereum block.



Ethereum block is divided in two parts, the **block header** and the **block body**.
The block header is the blockchain part of Ethereum. This is the structure that contains the hash of its predecessor block (also known as parent block), building a cryptographically guaranteed chain.
The block body contains a **list of transactions** that have been included in this block and a **list of uncle (ommer) blocks headers** (if you want to know more about uncle blocks recommend this post).

The block body contains a **list of transactions** that have been included in this block and a **list of uncle (ommer) blocks headers** (if you want to know more about uncle blocks recommend this post).