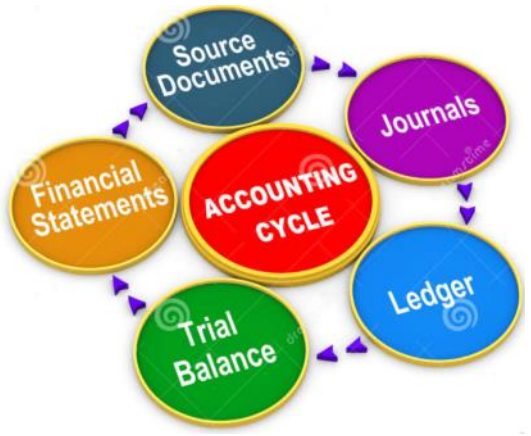
state machine
Transactions in Ethereum
ECIP-1049: Change the ETC Proof of Work Algorithm to Keccak256
Contracts and Transactions
Understanding How ECDSA Protects Your Data.
Signing and Verifying Ethereum Signatures
EIP-155 : Simple replay attack protection

Book-keeping --> accounting --> balance --> state
Bookkeeping is the recording of financial transactions, and is part of the process of accounting in business. Transactions include purchases, sales, receipts and payments by an individual person or an organization/corporation.
There are several standard methods of bookkeeping, including the single-entry and double-entry bookkeeping systems.
From <https://en.wikipedia.org/wiki/Bookkeeping>
https://www.dreamstime.com/stock-image-d-life-cycle-accounting-process-illustration-circular-flow-chart-image30625511



Authorized capital
Credit
Fixed Assets
Costs
Incomes

Op.No.	Input	Output	RemainingAmount
1	123	0	123
2	5	11	117

Compare with UOTX system
https://medium.com/@olxc/ethereum-and-smart-contracts-basics-e5c84838b19

State 0

Authorized Capital	Credit	Fixed Asset				**Balance 0**
12 000	9 000	-12 000				**9 000**

State 1

Authorized Capital	Credit		Electricity Cost 1	Mining 1	Percent for Credit	**Balance 1**
	9 000		-3 000	+31 000	-1 000	**36 000**

State 2

Authorized Capital	Credit		Electricity Cost 2	Mining 2	Percent for Credit	**Balance 2**
	8 000		-15 000	-	-1 000	**20 000**

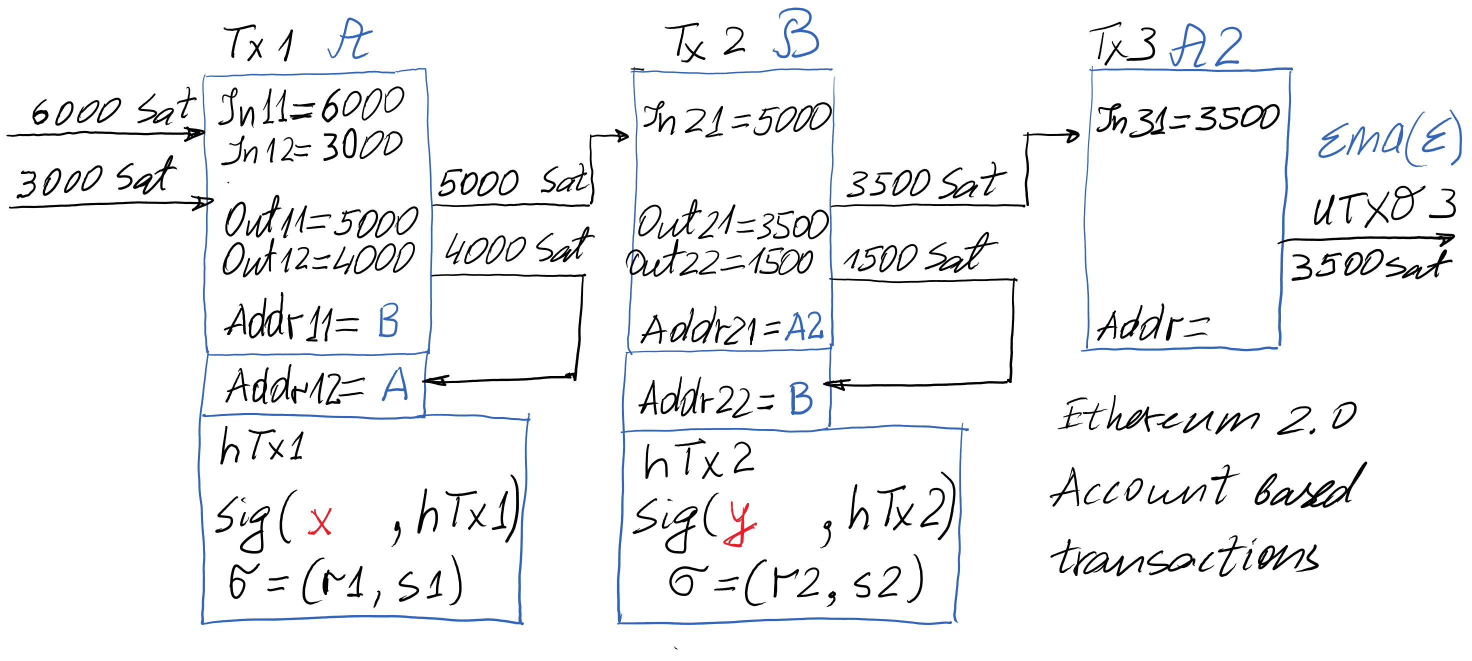
Book-keeping --> Accounting --> Balance --> State

UTxO

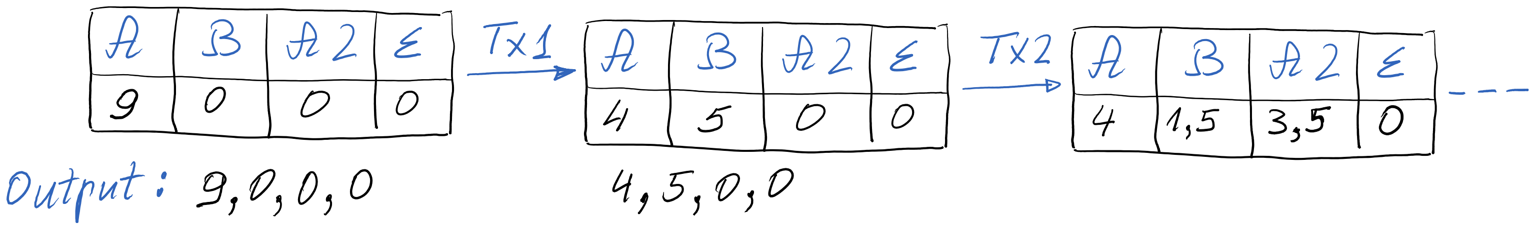
Tx 1 A

Tx 2 B

Tx 3 A2



State tranasition diagramm



Takenobu T. Ethereum EVM illustrated
[What is state machine? - Definition from WhatIs.com (techtarget.com)](https://techtarget.com)

State machine

In general, a state machine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change. A computer is basically a state machine and each machine instruction is input that changes one or more states and may cause other actions to take place. Each computer's data register stores a state. The read-only memory from which a boot program is loaded stores a state (the boot program itself is an initial state). The operating system is itself a state and each application that runs begins with some initial state that may change as it begins to handle input. Thus, at any moment in time, a computer system can be seen as a very complex set of states and each program in it as a state machine. In practice, however, state machines are used to develop and describe specific device or program interactions.
To summarize it, a state machine can be described as:

To summarize it, a state machine can be described as:

- An initial state or record of something stored someplace
- A set of possible input events
- A set of new states that may result from the input

- A set of possible actions or output events that result from a new state

In their book *Real-time Object-oriented Modeling*, Bran Selic & Garth Gullekson view a state machine as:

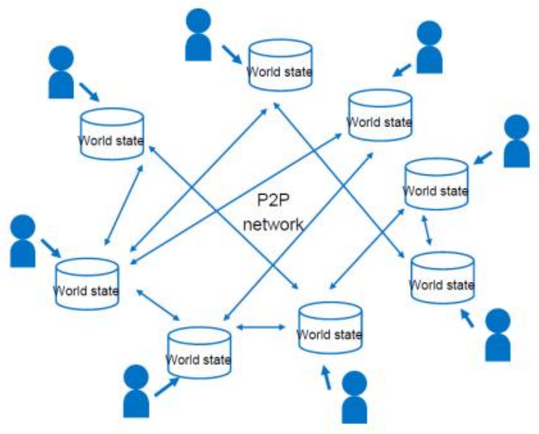
A set of input events	**1**.A description of the initial state
A set of output events	**2**.A set of **input** events
A set of states	**3**.A set of **states**
A function that maps states and input to output	**4**.A set of output events
A function that maps states and inputs to states	**5**.A function that maps **inputs** and **states** to **states**
A description of the initial state	**6**.A function that maps **inputs** and **states** to **output**

A finite state machine is one that has a limited or finite number of possible states. (An infinite state machine can be conceived but is not practical.) A finite state machine can be used both as a development tool for approaching and solving problems and as a formal way of describing the solution for later developers and system maintainers. There are a number of ways to show state machines, from simple tables through graphically animated illustrations.

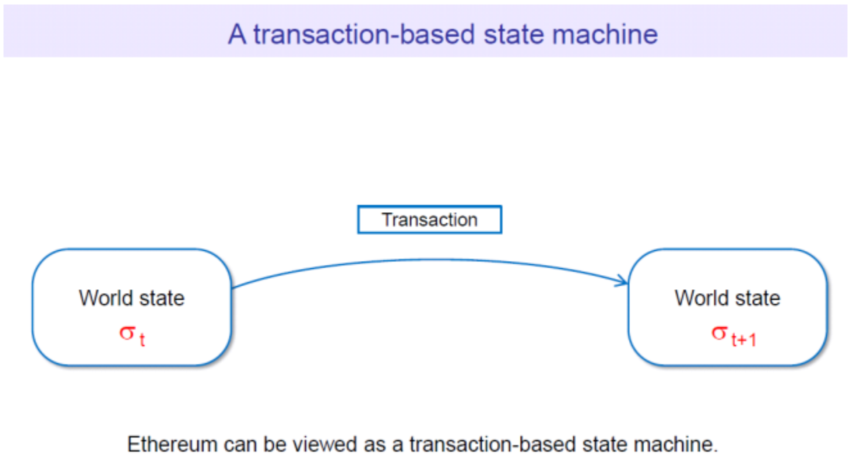
Continue Reading About state machine

- Desaware offers an "Introduction to State Machines."
- Xilinx offers a Finite State Machine Editor product.
- Architecture — Sawtooth v0.8.13 documentation (hyperledger.org)

Decentralised database

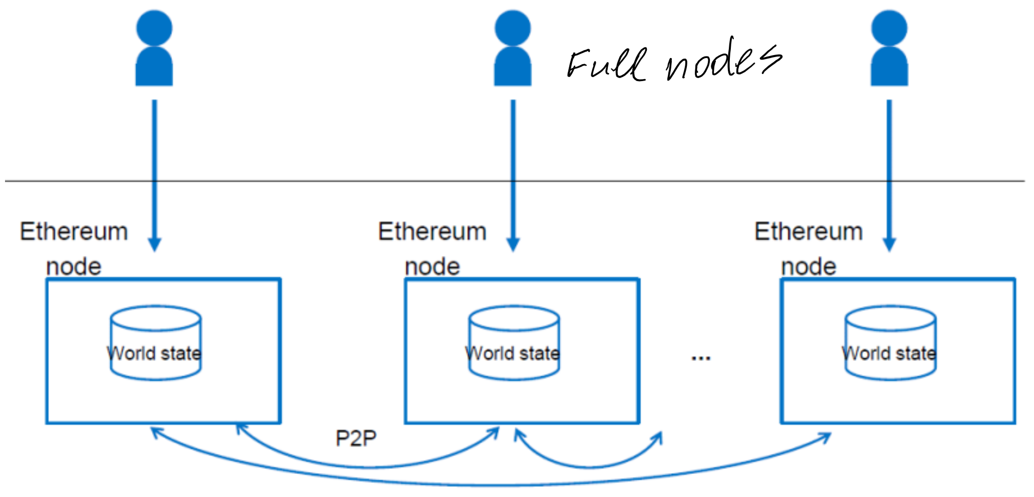


A blockchain is a globally shared, decentralised, transactional database.


A transaction-based state machine
Transaction
World state σ_t
World state σ_{t+1}
Ethereum can be viewed as a transaction-based state machine.

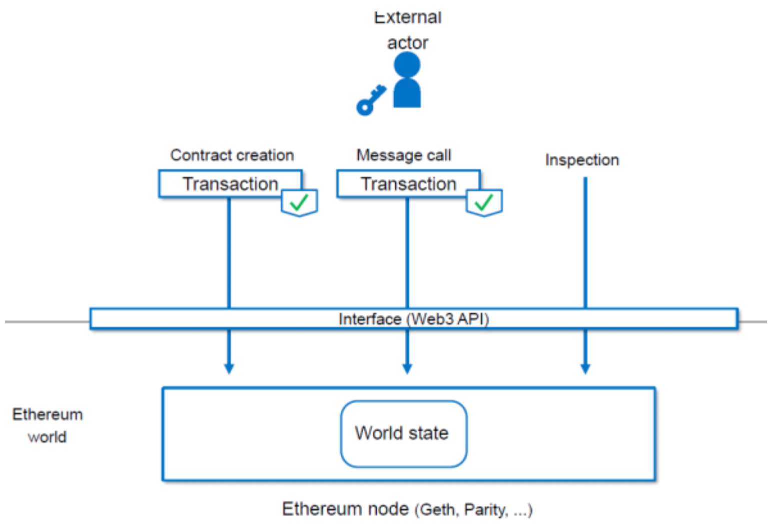
P2P network inter nodes

Light nodes



Decentralised nodes constitute Ethereum P2P network.

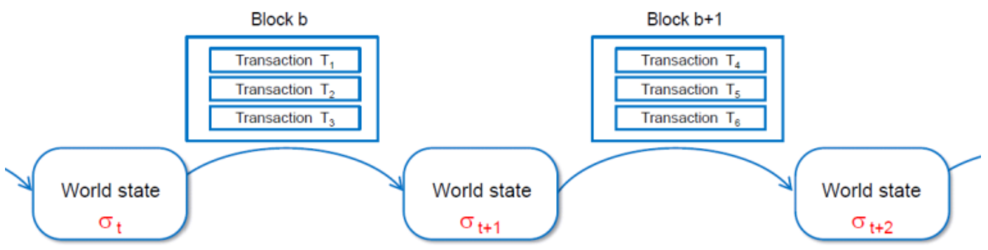
Interface to a node



External actors access the Ethereum world through Ethereum nodes.

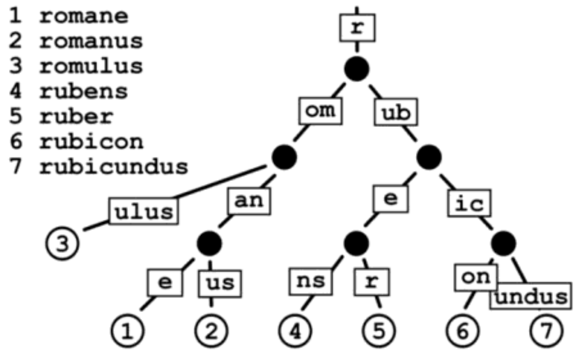
Chain of states

Ethereum can be seen as a state chain.



Radix tree - Wikipedia

Radix tree (also **radix trie** or **compact prefix tree**) is a data structure that represents a space-optimized trie (prefix tree - re**trie**ve) in which each node that is the only child is merged with its parent. The result is that the number of children of every internal node is at most the radix ***k*** of the radix tree, where ***k*** is a positive integer and a power *x* of 2, having *x* ≥ 1. Unlike regular trees, edges can be labeled with sequences of elements as well as single elements. This makes radix trees much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.



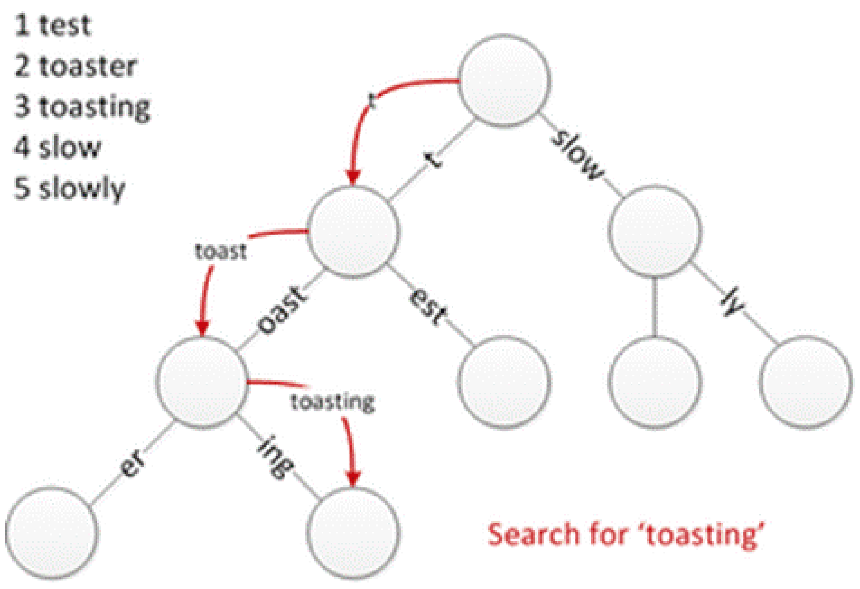
Unlike regular trees (where whole keys are compared *en masse* from their beginning up to the point of inequality), the key at each node is compared chunk-of-bits by chunk-of-bits, where the quantity of bits in that chunk at that node is the radix ***r*** of the radix trie. When the ***r*** is 2, the radix trie is binary (i.e., compare that node's 1-bit portion of the key), which minimizes sparseness at the expense of maximizing trie depth—i.e., maximizing up to conflation of nondiverging bit-strings in the key. When ***r*** is an integer power of 2 having ***r*** ≥ 4, then the radix trie is an ***r***-ary trie, which lessens the depth of the radix trie at the expense of potential sparseness.
Note that although the examples in this article show strings as sequences of characters, the type of the string elements can be chosen arbitrarily; for example, as a bit or byte of the string representation when using multibyte character encodings or Unicode.

Radix trees support insertion, deletion, and searching operations. Insertion adds a new string to the trie while trying to minimize the amount of data stored. Deletion removes a string from the trie. Searching operations include (but are not necessarily limited to) exact lookup, find predecessor, find successor, and find all strings with a prefix. All of these operations are O(***k***) where ***k*** is the maximum length of all strings in the set, where length is measured in the quantity of bits equal to the radix of the radix trie.

Lookup

Finding a string in a Patricia trie

The lookup operation determines if a string exists in a trie. Most operations modify this approach in some way to handle their specific tasks. For instance, the node where a string terminates may be of importance. This operation is similar to tries except that some edges consume multiple elements.



The following pseudo code assumes that these classes exist.

Edge

- *Node* targetNode
- *string* label

Node

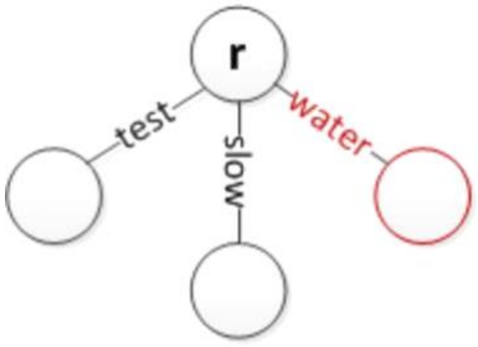
- *Array of Edges* edges
- *function* isLeaf()

```
function lookup(string x)
{
    // Begin at the root with no elements found
    Node traverseNode := root;
    int elementsFound := 0;

    // Traverse until a leaf is found or it is not possible to
continue
    while (traverseNode != null && !traverseNode.isLeaf() &&
elementsFound < x.length)
    {
        // Get the next edge to explore based on the elements not
yet found in x
        Edge nextEdge := select edge from traverseNode.edges where
edge.label is a prefix of x.suffix(elementsFound)
            // x.suffix(elementsFound) returns the last
(x.length - elementsFound) elements of x

        // Was an edge found?
        if (nextEdge != null)
        {
            // Set the next node to explore
            traverseNode := nextEdge.targetNode;

            // Increment elements found based on the label stored
at the edge
            elementsFound += nextEdge.label.length;
        }
        else
        {
            // Terminate loop
            traverseNode := null;
        }
    }

    // A match is found if we arrive at a leaf node and have used
up exactly x.length elements
    return (traverseNode != null && traverseNode.isLeaf() &&
elementsFound == x.length);
}
```

Insertion

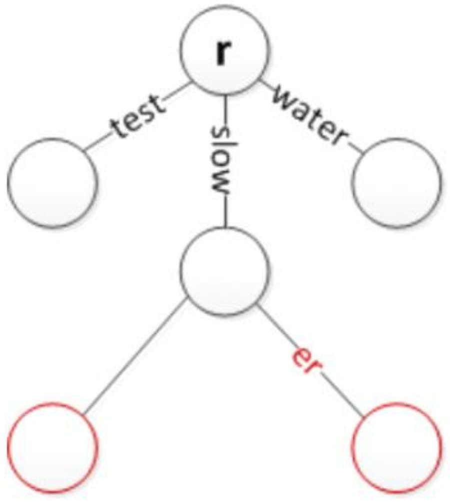
To insert a string, we search the tree until we can make no further progress. At this point we either add a new outgoing edge labeled with all remaining elements in the input string, or if there is already an outgoing edge sharing a prefix with the remaining input string, we split it into two edges (the first labeled with the common prefix) and proceed. This splitting step ensures that no node has more children than there are possible string elements.

Several cases of insertion are shown below, though more may exist. Note that ***r*** simply represents the root. It is assumed that edges can be labelled with empty strings to terminate strings where necessary and that the root has no incoming edge. (The lookup algorithm described above will not work when using empty-string edges.)

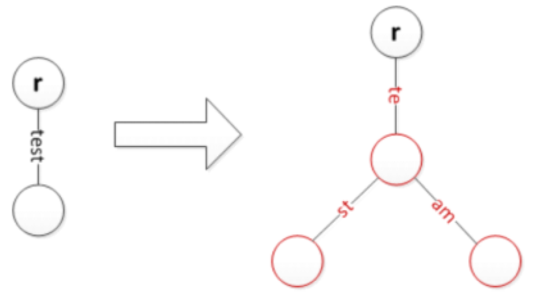


Insert 'water' at the root

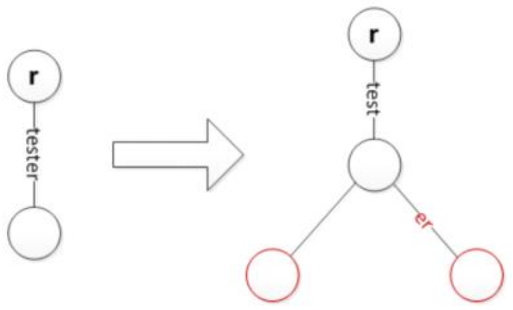
Insert 'water' at the root **r**



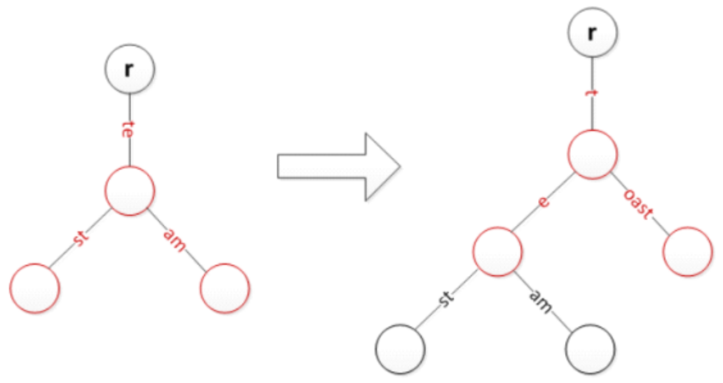
Insert 'slower' while keeping 'slow'



Insert 'team' while splitting 'test' and creating a new edge label 'st'



Insert 'test' which is a prefix of 'tester'



Insert 'toast' while splitting 'te' and moving previous strings a level lower

Deletion
To delete a string ***x*** from a tree, we first locate the leaf representing ***x***. Then, assuming ***x*** exists, we remove the corresponding leaf node. If the parent of our leaf node has only one other child, then that child's incoming label is appended to the parent's incoming label and the child is removed.

Additional operations

- Find all strings with common prefix: Returns an array of strings that begin with the same prefix.
- Find predecessor: Locates the largest string less than a given string, by lexicographic order.
- Find successor: Locates the smallest string greater than a given string, by lexicographic order.

Ethereum Yellow Paper Walkthrough (2/7) (lucassaldanha.com)

Merkle Tree and Ethereum Objects - Ethereum Yellow Paper Walkthrough (2/7)

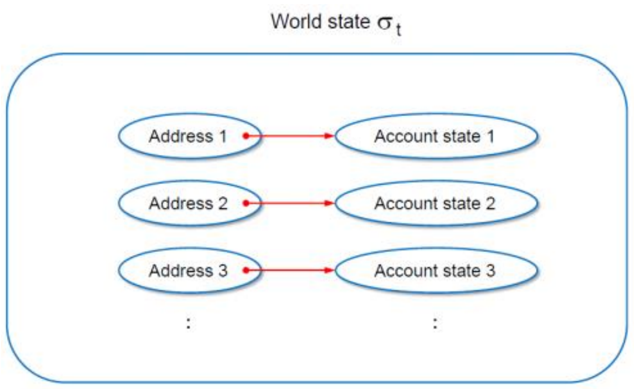
Lucas Saldanha

11 Dec 2018 • 11 min read

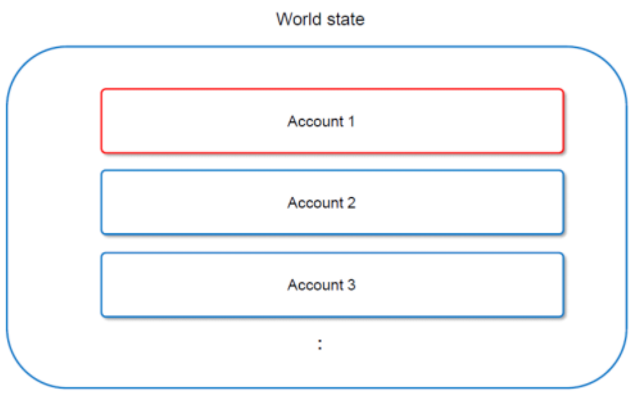
This is another post in our series exploring the Ethereum Yellow Paper. In this post, we will learn more about the main objects in Ethereum and their role. We'll also briefly discuss how Merkle trees are used in Ethereum.

(DISCLAIMER: this post is based on the Byzantium version of the Yellow Paper, version e94ebda from 5th June 2018)

World State



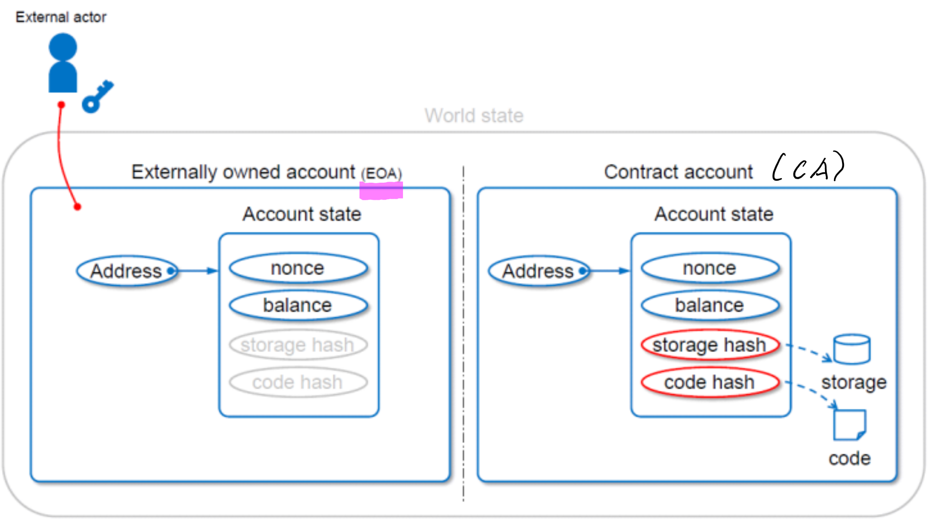
The **world state is a mapping between addresses (accounts) and account states**. The world state is not stored on the blockchain but the Yellow Paper states it is expected implementations store this data in a trie (also referred as the state database or state trie).The world state can be seen as the global state that is constantly updated by transaction executions. If you remember the discussion in the first post of the series about the Ethereum network being like a decentralized computer, the world state is considered this computer's hard drive.
All the information about Ethereum accounts live in the world state and is stored in the world state trie. If you want to know the balance of an account, or the current state of a smart contract, you query the world state trie to retrieve the account state of that account.



An account is an object in the world state.

Till this place

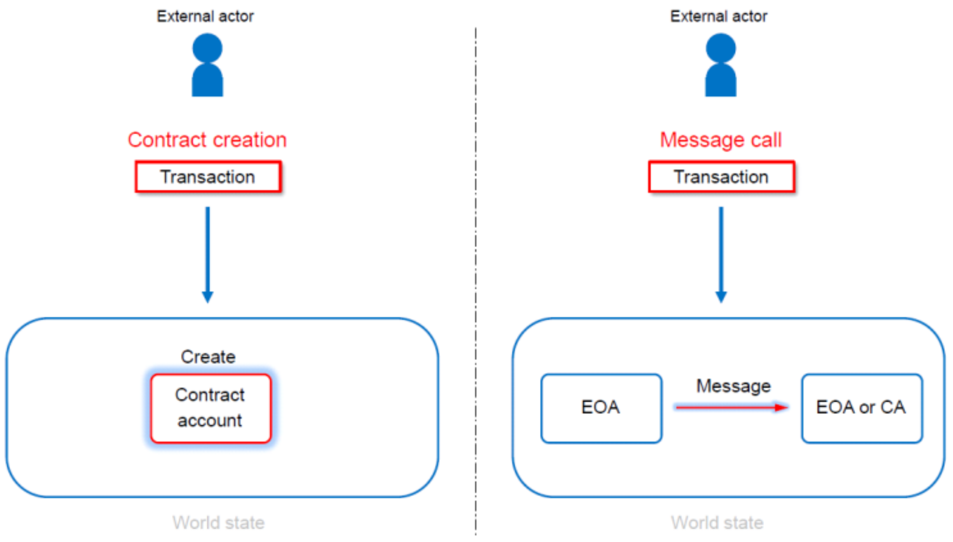
Two practical types of account



EOA is controlled by a private key.
EOA cannot contain EVM code.

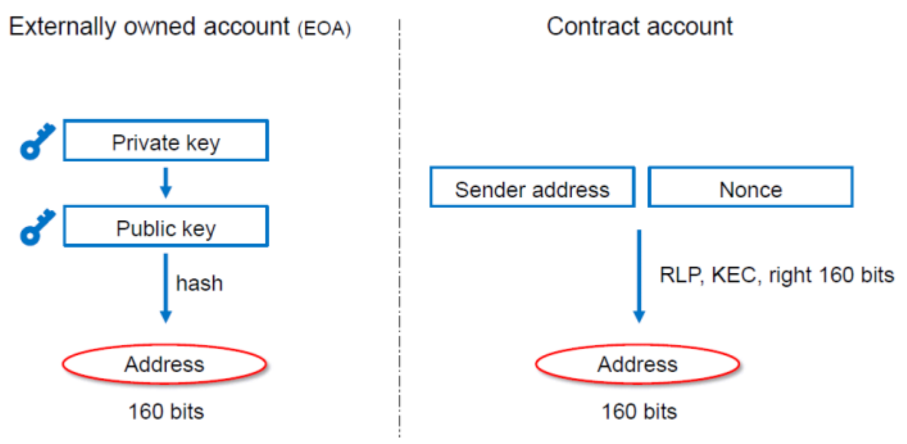
Contract contains EVM code.
Contract is controlled by EVM code.

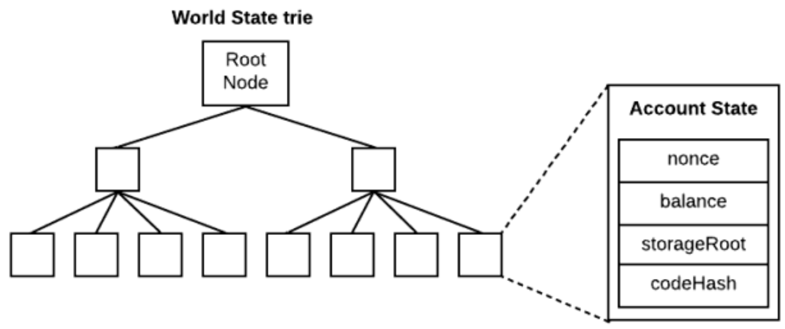
Two practical types of transaction



There are two practical types of transaction, contract creation and message call.

Address of account





World state trie and Account storage

Account State

In Ethereum, there are two types of accounts: External Owned Accounts (EOA) and Contract Accounts. An EOA account is the account that you and I would have, that we can use to send Ether to one another and deploy smart contracts. A contract account is the account that is created when a smart contract is deployed. Every smart contract has its own Ethereum account.
The account state contains information about an Ethereum account.
For example, it stores how much Ether an account has and the number of transactions sent by the account. Each account has an account state.
Let's take a look into each one of the fields in the account state:

- **nonce**
- Number of transactions sent from this address (if this is an External Owned Account - EOA) or the number of contract-creations made by this account (don't worry about what *contract-creations* means for now).
- **balance**
- Total Ether (in Wei) owned by this account.
- **storageRoot**
- Hash of the root node of the account storage trie (we'll see what the account storage is in a moment).
- **codeHash**
- For contract accounts, hash of the EVM code of this account. For EOAs, this will be empty.

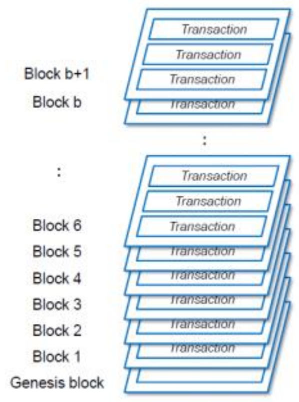
One important details about the account state is that **all fields (except the codeHash) are mutable**. For example, when one account sends some Ether to another, the nonce will be incremented and the balance will be updated to reflect the new balance.

One of the consequences of the codeHash being immutable is that if you deploy a contract with a bug, you can't update the same contract. You need to deploy a new contract (the buggy version will be available forever). This is why it is important to use Truffle to develop and test your smart contracts and follow the best practices when working with Solidity.

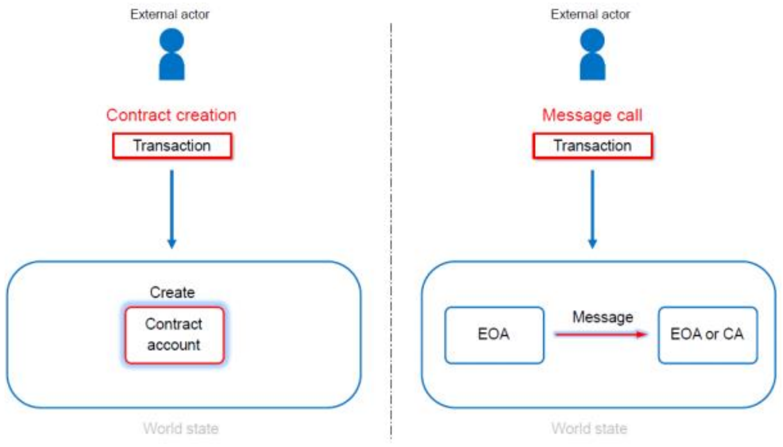
The **Account Storage trie is where the data associated with an account is stored**.

This is only relevant for Contract Accounts, as for EOAs the storageRoot is empty and the codeHash is the hash of an empty string. All smart contract data is persisted in the account storage trie as a mapping between 32-bytes integers. We won't discuss in details how the contract data is persisted in the account state trie. If you really want to learn about the internals, I suggest reading this post. The hash of an account storage root node is persisted in the storageRoot field in the account state of the respective account.

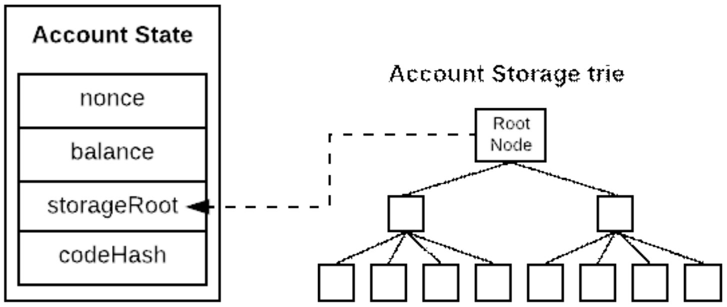
Ethereum can also be seen as a stack of transactions.
Stack of transactions : Ledger



Two practical types of transaction



Account state and Account Storage trie



Transactions are what makes the state change from the current state to the next state. In Ethereum, we have three types of transactions:

1. Transactions that **transfer value** between two EOAs (e.g, change the sender and receiver account balances)
2. Transactions that **send a message call** to a contract (e.g, set a value in the smart contract by sending a message call that executes a setter method)
3. Transactions that **deploy a contract** (therefore, create an account, the contract account)

(technically, types 1 and 2 are the same... transactions that send message calls that affect an account state, either EOA or contract accounts. But is it easier to think about them as three different types)

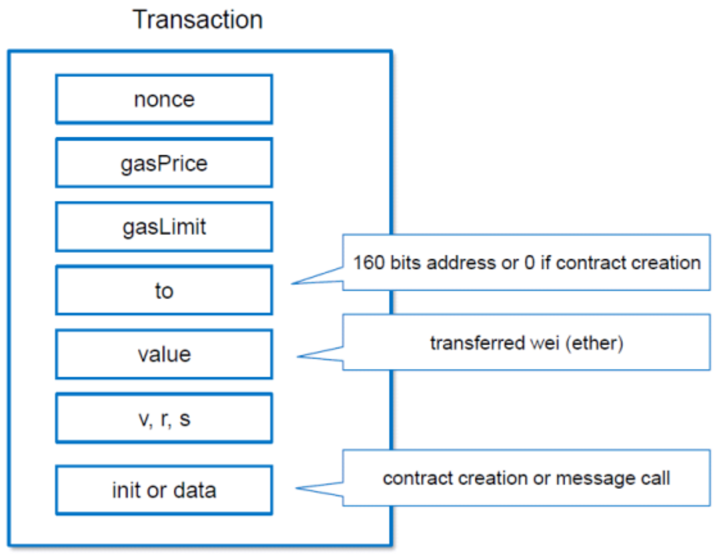
These are the fields of a transaction:

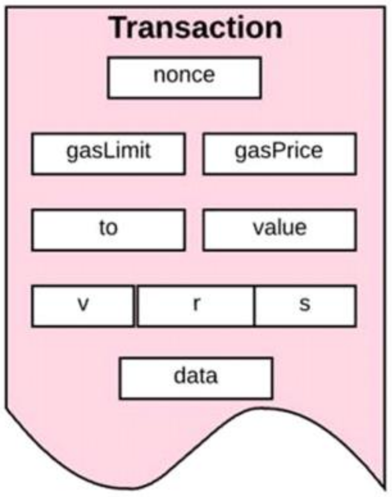
- **nonce**
  Number of transactions sent by the account that created the transaction.
- **gasPrice**
  Value (in Wei) that will be paid per unit of gas for the computation costs of executing this transaction.
- **gasLimit**
  Maximum amount of gas to be used while executing this transaction.
- **to**
  If this transaction is transfering Ether, address of the EOA account that will receive a value transfer.
  If this transaction is sending a message to a contract (e.g, calling a method in the smart contract), this is address of the contract.
  If this transactions is creating a contract, this value is always empty.
- **value**
  If this transaction is transfering Ether, amount in Wei that will be transferred to the recipient account.
  If this transaction is sending a message to a contract, amount of Wei payable by the smart contract receiving the message.
- If this transaction is creating a contract, this is the amount of Wei that will be added to the balance of the created contract.

- **v, r, s**
  Values used in the cryptographic signature of the transaction used to determine the sender of the transaction.
- **data** (only for value transfer and sending a message call to a smart contract)
  Input data of the message call (e.g, imagine you are trying to execute a setter method in your smart contract, the data field would contain the identifier of the setter method and the value that should be passed as parameter).
- **init** (only for contract creation)
  The EVM-code utilized for initialization of the contract.

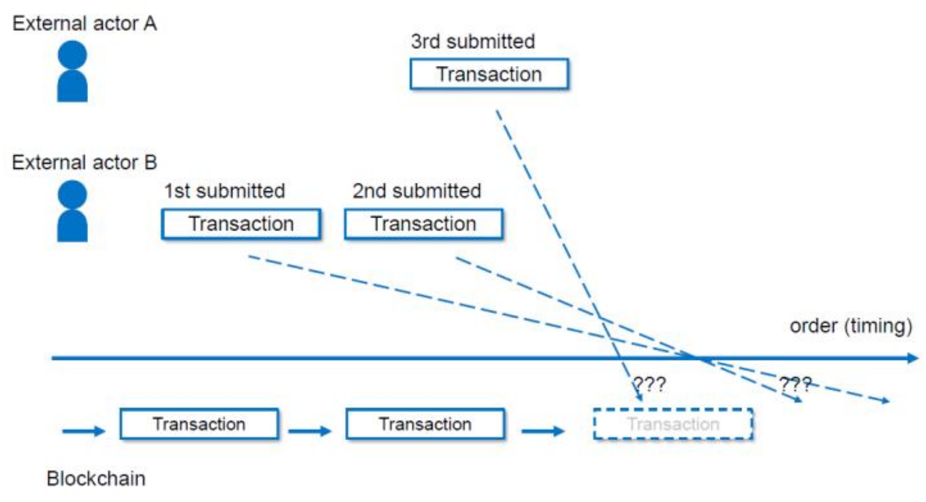
Don't try to grasp all of this at once... Some fields like the *data* field or the *init* field require you to have a deeper understanding of the internals of Ethereum to really understand what they mean and how to use them. This is not the time to deeply understand any of these fields.
Not surprisingly, all transactions in a block are stored in a trie. And the root hash of this trie is stored in the... block header! Let's take a look into the anatomy of an Ethereum block.

Fields of a transaction



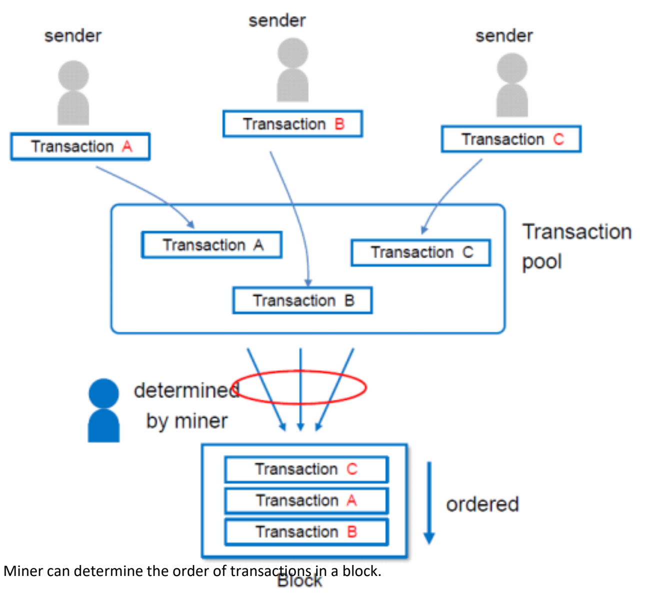


Order of transactions



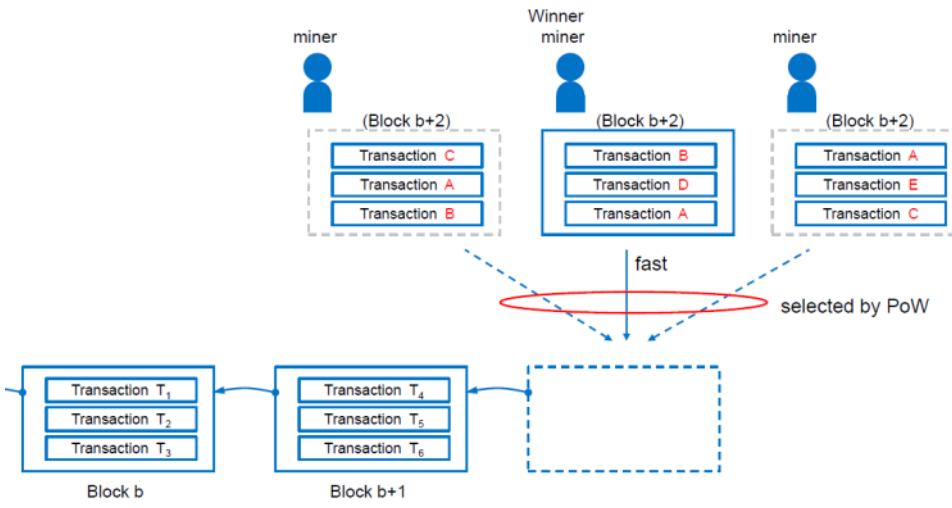
Transaction order is not guaranteed.

Ordering inner block



Miner can determine the order of transactions in a block.

Ordering inter blocks



The order between blocks is determined by a consensus algorithm such as PoW.