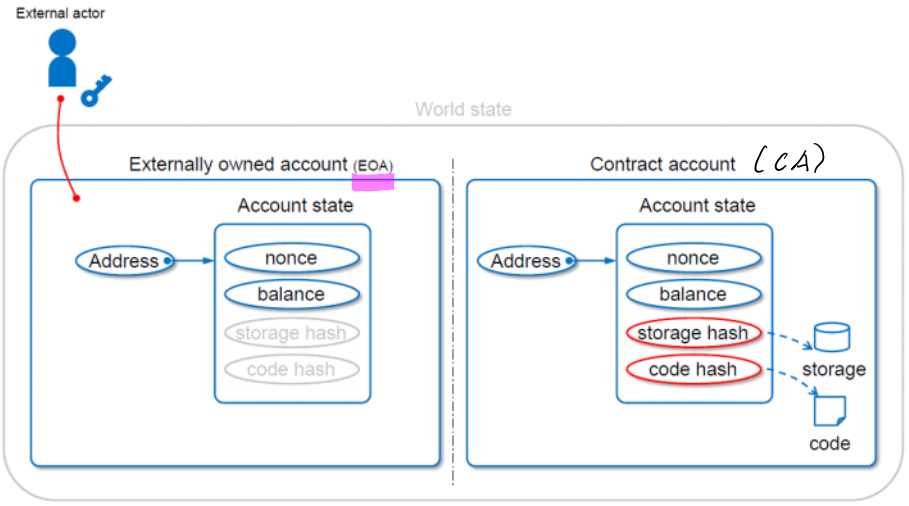
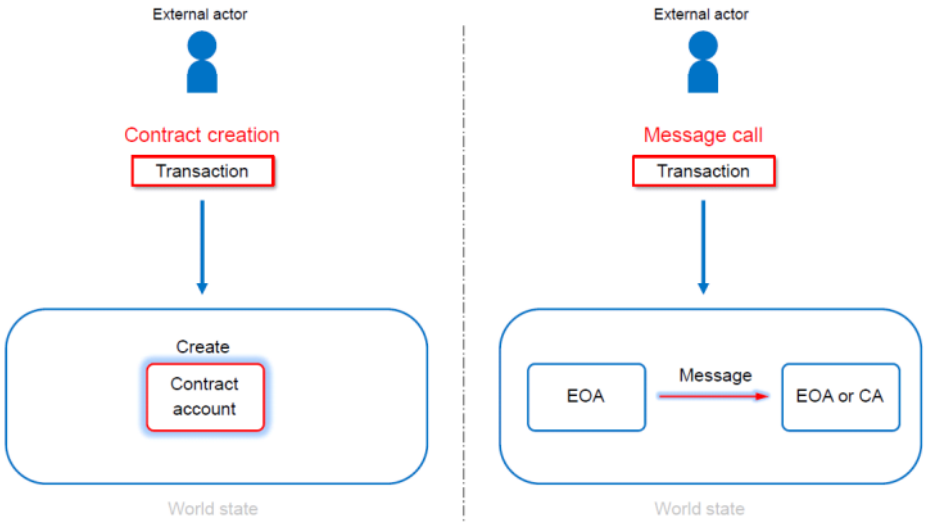
Two practical types of account



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

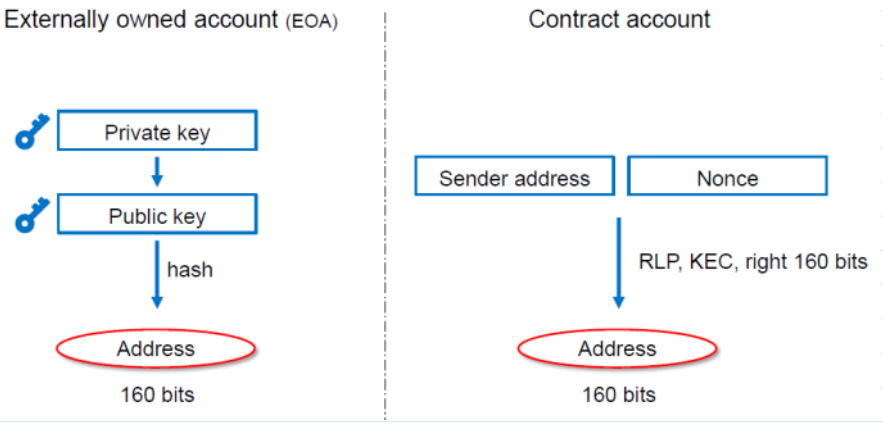
Contract contains EVM code.
Contract is controlled by EVM code.

Two practical types of transaction



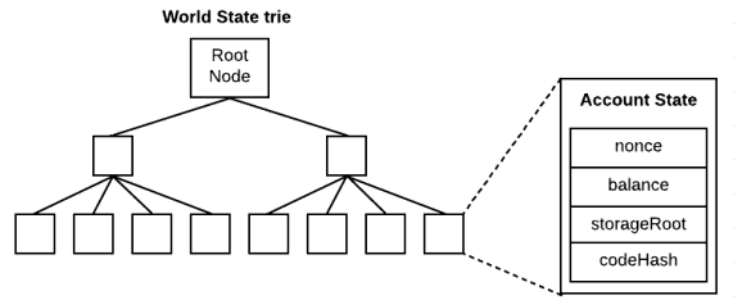
There are two practical types of transaction, contract creation and message call.

Address of account



KECCAK-256 hash, which Ethereum uses.

Account state and Account Storage trie
World state trie and Account storage trie



Account State

In Ethereum, there are two types of accounts: External Owned Accounts (EOA) and Contract Accounts (CA). 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.

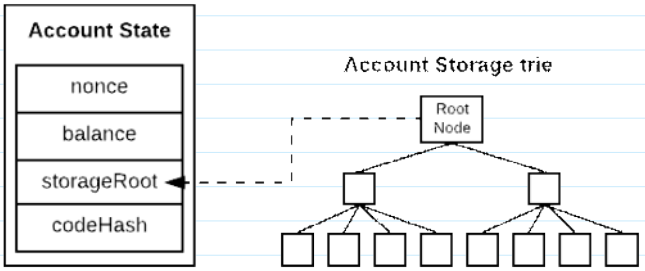
One important details about the account state is that **all fields (except the codeHash) are mutable**. For example, when one account sends some Ether to another, the nonce will be incremented and the balance will be updated to reflect the new balance.

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.

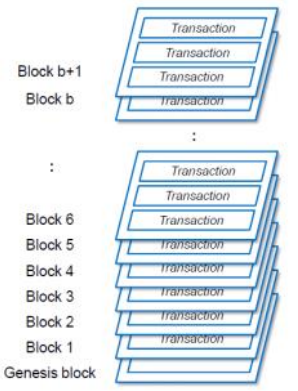
2^{256}

10^{80}

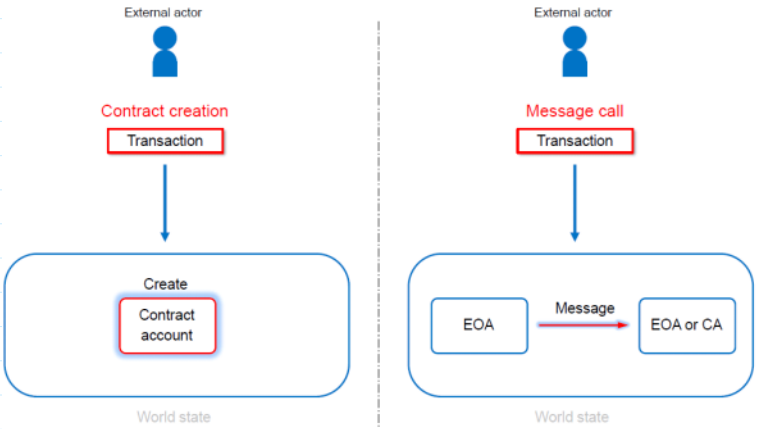


Ethereum can also be seen as a stack of transactions.

Stack of transactions : Ledger



Two practical types of transaction



T**ransactions 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)

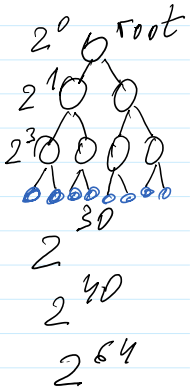
(technically, types 1 and 2 are the same... transactions that send message calls that affect an account state, either EOA or contract accounts. But is it easier to think about them as three different types)

These are the fields of a transaction:

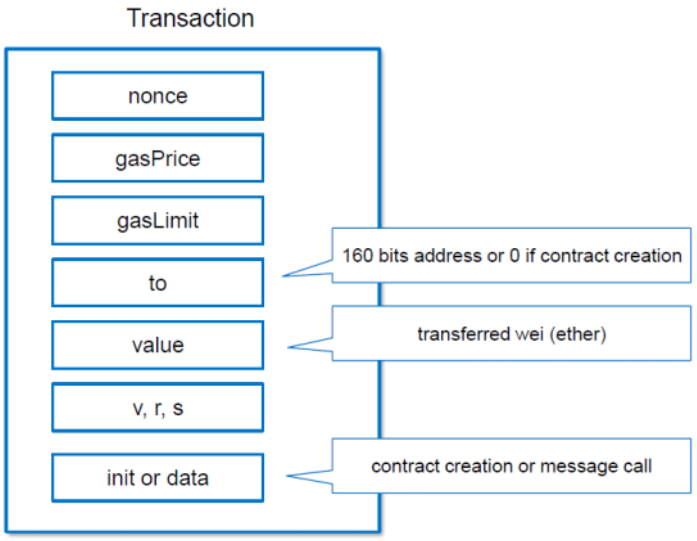
- **nonce**
  Number of transactions sent by the account that created the transaction.
- **gasPrice**
  Value (in Wei) that will be paid per unit of gas for the computation costs of executing this transaction.
- **gasLimit**
  Maximum amount of gas to be used while executing this transaction.
- **to**
  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.

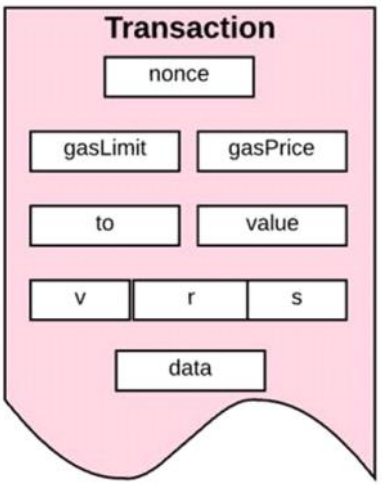
Don't try to grasp all of this at once... Some fields like the *data* field or the *init* field require you to have a deeper understanding of the internals of Ethereum to really understand what they mean and how to use them. This is not the time to deeply understand any of these fields.

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.

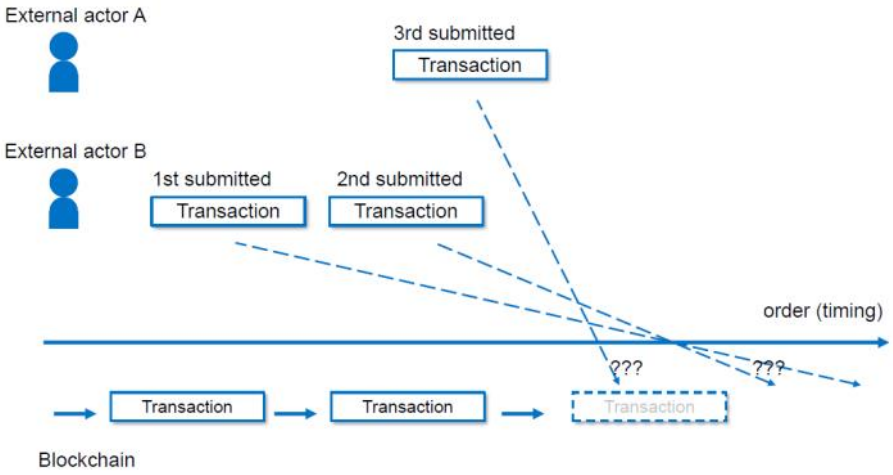


Fields of a transaction



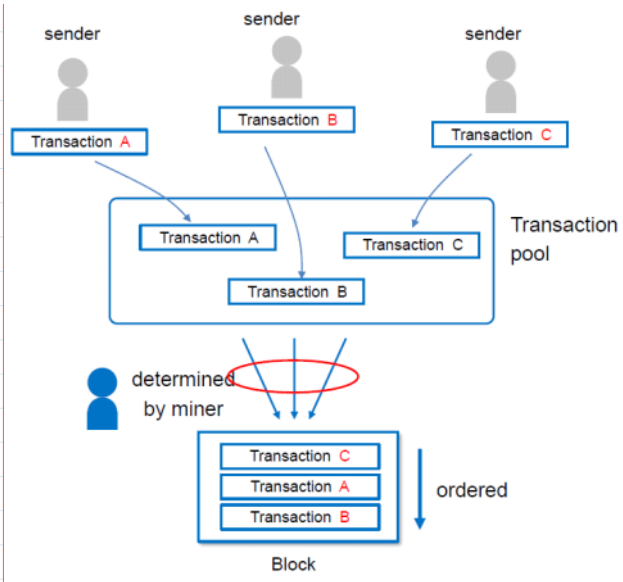


Order of transactions



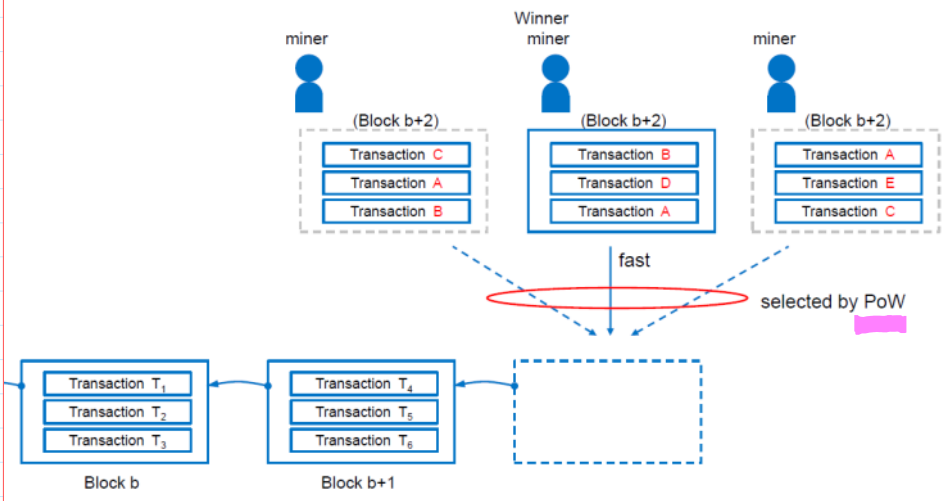
Transaction order is not guaranteed.

Ordering inner block

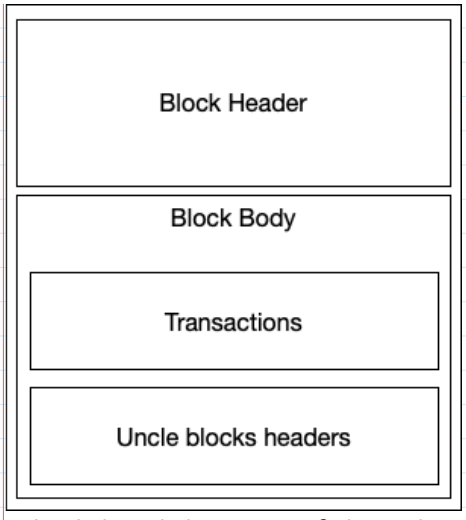


Miner can determine the order of transactions in a block.

Ordering inter blocks

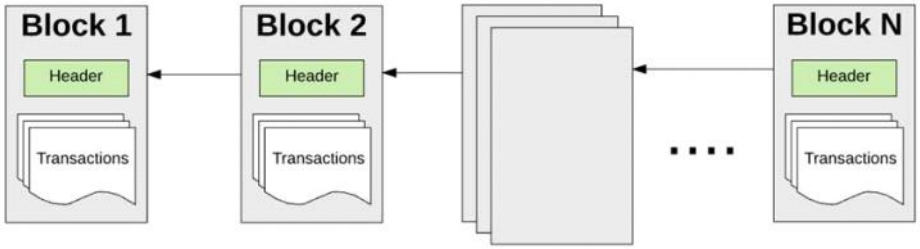


The order between blocks is determined by a consensus algorithm such as PoW or PoS or PoA.



A high level diagram of the Ethereum block

Block
The **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).



Fields in the block header:

- **parentHash**
  Hash of the block header from the previous block. Each block contains a hash of the previous block, all the way to the first block in the chain. This is how all the data is protected against modifications (any modification in a previous block would change the hash of all blocks after the modified block).
- **ommersHash**
  Hash of the uncle blocks headers part of the block body.
- **beneficiary**
  Ethereum account that will get fees for mining this block.
- **stateRoot**
  Hash of the root node of the world state trie (after all transactions are executed).
- **transactionsRoot**

Hash of the root node of the transactions trie. This trie contains all transactions in the block body.

- **receiptsRoot**
  Every time a transactions is executed, Ethereum generates a transaction receipt that contains information about the transaction execution. This field is the hash of the root node of the transactions receipt trie.
- **logsBloom**
  Bloom filter that can be used to find out if logs were generated on transactions in this block (if you want more details check this Stack Overflow answer). This avoids storing of logs in the block (saving a lot of space).
- **Difficulty:** **PoW** mining
  Difficulty level of this block. This is a measure of how hard it was to mine this block (I'm not diving into the details of how this is calculated in this post).
- **number**
  Number of ancestor blocks. This represents the height of the chain (how many blocks are in the chain). The genesis block has number zero.
- **gasLimit**
  Each transaction consumes gas. The gas limit specifies the maximum gas that can be used by the transactions included in the block. It is a way to limit the number of transactions in a block.
- **gasUsed**
  Sum of the gas cost of each transaction in the block.
- **timestamp**
  Unix timestamp when the block was created. Note that due to the decentralized nature of Ethereum, we can't trust in this value (specially when implementing smart contracts that have time related business logic). PoW???
- **extraData**
  Arbitrary byte array that can contain anything. When a miner is creating the block, it can choose to add anything in this field.
- **mixHash**
  Hash used to verify that a block has been mined properly (if you want to really understand this, read about the Ethash proof-of-work function).
- **nonce**
  Same as the mixHash, this value is used to verify that a block has been mined properly.

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
- Validation --> PoS

I am referring to the KECCAK-256 hash, which Ethereum uses.

Till this place

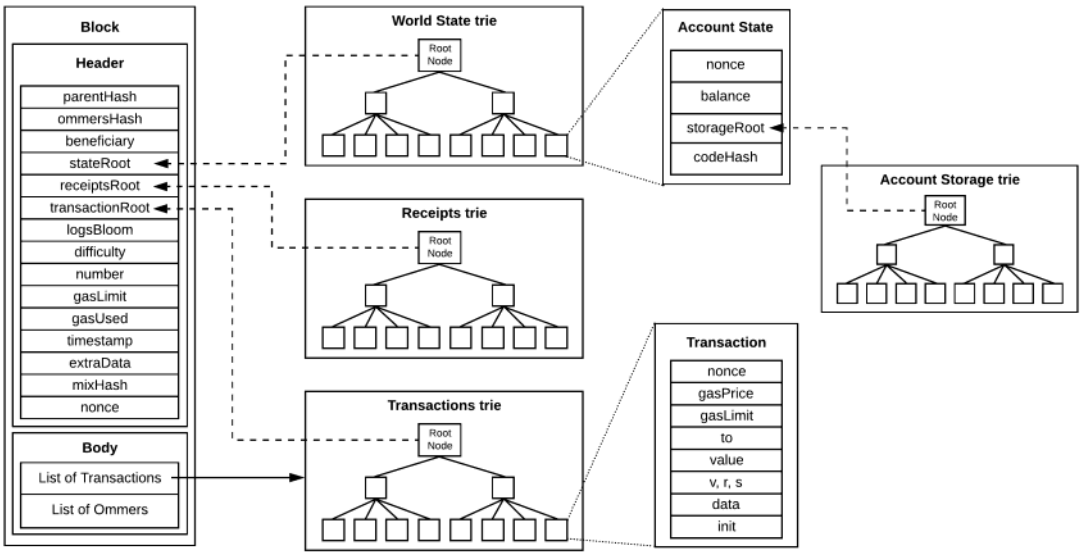
Conclusion

Basically, Ethereum has 4 types of tries:

1. **The world state trie contains the mapping between addresses and account states**. The hash of the root node of the world state trie is included in a block (in the **stateRoot** field) to represent the current state when that block was created. We only have one world state trie.
2. **The account storage trie contains the data associated to a smart contract**. The hash of the root node of the Account storage trie is included in the sccount state (in the **storageRoot** field). We have one Account storage trie for each account.
3. **The transaction trie contains all the transactions included in a block**. The hash of the root node of the Transaction trie is included in the block header (in the **transactionsRoot** field). We have one transaction trie per block.
4. **The transaction receipt trie contains all the transaction receipts for the transactions included in a block**. The hash of the root node of the transaction receipts trie is included in also included in the block header (in the **receiptsRoot** field); We have one transaction receipts trie per block.

And the objects that we discussed are:

1. **World state**: the hard drive of the distributed computer that is Ethereum. It is a mapping between addresses and account states.
2. **Account state**: stores the state of each one of Ethereum's accounts. It also contains the storageRoot of the account state trie, that contains the storage data for the account.
3. **Transaction**: represents a state transition in the system. It can be a funds transfer, a message call or a contract deployment.
4. **Block**: contains the link to the previous block (parentHash) and contains a group of transactions that, when executed, will yield the new state of the system. It also contains the **stateRoot**, the **transactionRoot** and the **receiptsRoot**, the hash of the root nodes of the world state trie, the transaction trie and the transaction receipts trie, respectively.



Block, transaction, account state objects and Ethereum tries

Global State — Sawtooth v0.8.13 documentation (hyperledger.org)

Global State

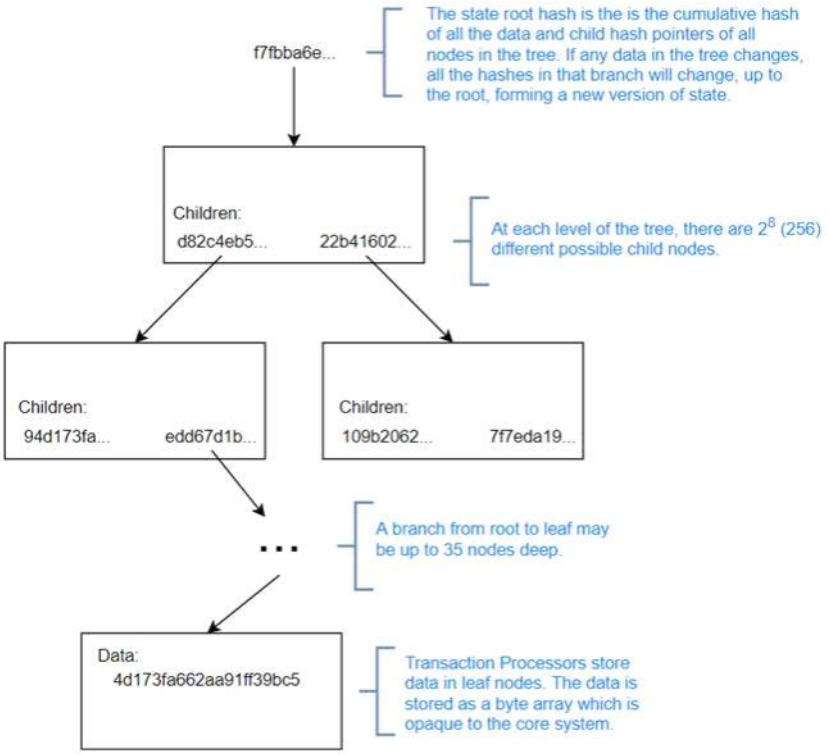
One goal of a distributed ledger like Sawtooth, indeed the *defining* goal, is to distribute a ledger among participating nodes. The ability to ensure a consistent copy of data amongst nodes in Byzantine consensus is one of the core strengths of blockchain technology.

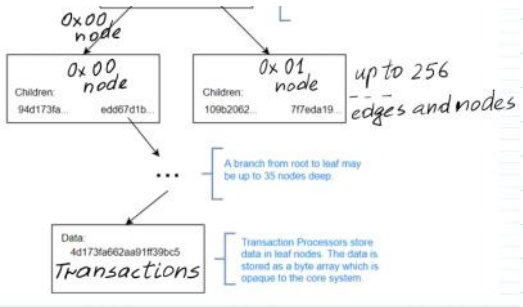
In a distributed system of n processes, where each process has an initial value, **Byzantine consensus** is the problem of agreeing on a common value, even though some of the processes may fail in arbitrary, even malicious, ways.

From <https://www.google.com/search?client=opera&hs=6S&biw=1240&bih=579&ei=dSkEYNCOD9a73APemJjACw&q=byzantine+consensus+blockchain&oq=Byzantine+consensus&gs_lcp=CgZwc3ktYWIQARgCMgQIABATMgQIABATMgQIABATMgQIABATMggIABAWEB4QEzIICAAQFhAeEBMyCAgAEBYQHhATMggIABAWEB4QEzIICAAQFhAeEBMyCAgAEBYQHhATOgQIABBHUMU-WMU-YN95aABwAngAgAGCAYgBggGSAQMwLjGYAQCgAQKgAQGqAQdnd3Mtd2l6yAEEwAEB&sclient=psy-ab>

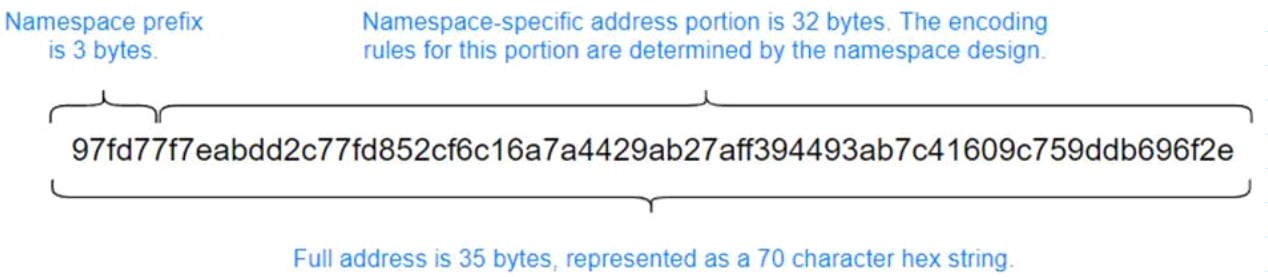
Sawtooth represents state for all transaction families in a single instance of a Radix Merkle Tree on each validator. The process of block validation on each validator ensures that the same transactions result in the same state transitions and that the resulting data is the same for all participants in the network.

The **state** is split into **namespaces** which allow flexibility for transaction family authors to define, share, and reuse global state data between transaction processors.





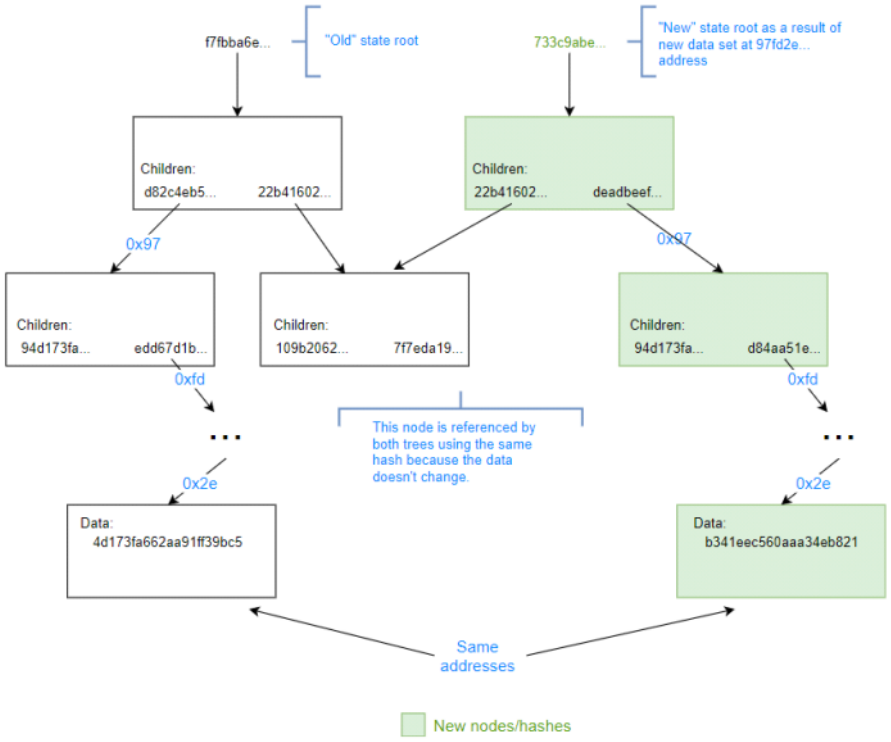
Radix Addresses

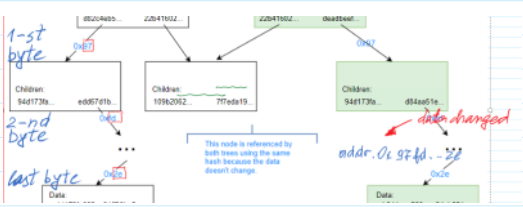


The tree is an addressable Radix tree because addresses uniquely identify the paths to leaf nodes in the tree where information is stored. An address is a hex-encoded 70 character string representing 35 bytes. In the tree implementation, each byte is a Radix path segment which identifies the next node in the path to the leaf containing the data associated with the address. The address format contains a 3 byte (6 hex character) namespace prefix which provides 2^{24} (16,777,216) possible different namespaces in a given instance of Sawtooth. The remaining 32 bytes (64 hex characters) are encoded based on the specifications of the designer of the namespace, and may include schemes for subdividing further, distinguising object types, and mapping domain-specific unique identifiers into portions of the address. For more information about general concepts, see the Radix page on wikipedia.

A **PATRICIA** trie is a special variant of the radix 2 (binary) trie, in which rather than explicitly store every bit of every key, the nodes store only the position of the first bit which differentiates two sub-trees. During traversal the algorithm examines the indexed bit of the search key and chooses the left or right sub-tree as appropriate. Notable features of the

PATRICIA trie include that the trie only requires one node to be inserted for every unique key stored, making PATRICIA much more compact than a standard binary trie. Also, since the actual keys are no longer explicitly stored it is necessary to perform one full key comparison on the indexed record in order to confirm a match. In this respect PATRICIA bears a certain resemblance to indexing using a hash table.[10]





Working with Ethereum protocol is interesting. Every day, I have the opportunity to learn something new (and also relearn things that I thought that I already knew). If you spot anything in this post that isn't quite clear (or even something that is clearly wrong), please comment and I'll try to address it as soon as possible. After all, we don't want to propagate misconceptions about the protocol.

References

- Merkle Trees
- Merkle Proofs
- How data is stored in Ethereum?
- Diving into Ethereum's world state
- How does Ethereum work anyway?
- A (Practical) Walkthrough of Smart Contract Storage

- Inside an Ethereum transaction
- Life Cycle of an Ethereum Transaction
- Ethereum Design Rationale