Foundations of Blockchain

Bitcoin, Ethereum, and Algorand

Matteo Nardelli
October, 2023
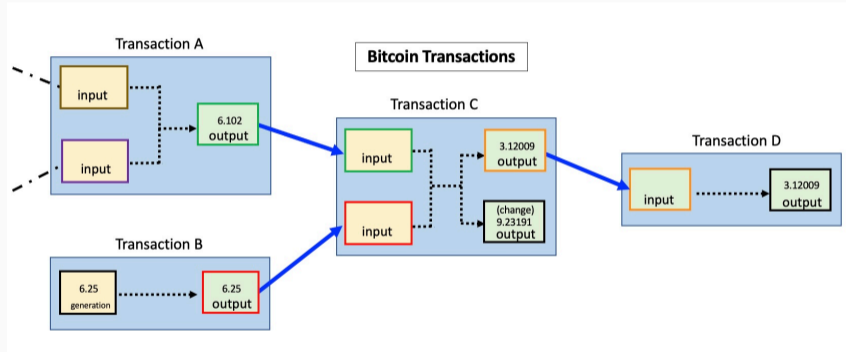
Bitcoin

The Bitcoin Network

REACHABLE BITCOIN NODES
Updated: Tue Aug 29 09:36:17 2023 CEST

16175 NODES [CHARTS]

IPv4: -8.9% / IPv6: -7.7% / .onion: -0.5%

Top 10 countries with their respective number of reachable nodes are as follows.

RANK	COUNTRY	NODES
1	n/a	10252 (63.38%)
2	United States	1494 (9.24%)
3	Germany	1268 (7.84%)
4	France	441 (2.73%)
5	Netherlands	311 (1.92%)
6	Canada	262 (1.62%)
7	Finland	221 (1.37%)
8	United Kingdom	167 (1.03%)
9	Russian Federation	158 (0.98%)
10	Switzerland	156 (0.96%)

Map shows concentration of reachable Bitcoin nodes found in countries around the world. [LIVE MAP]

Bitcoin

Transactions

Bitcoin: Transaction

- Transactions are the most important part of the Bitcoin system.
- They are data structures that encode the transfer of value between participants.
- Each transaction is a public entry in bitcoin's blockchain.
- Try a Bitcoin Explorer to analyze a transaction.

Bitcoin: Transaction

Transaction View information about a bitcoin transaction

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK (0.1 BTC - Output) →

1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA - (Unspent)   0.015 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK - (Unspent)   0.0845 BTC

97 Confirmations | 0.0995 BTC

Summary	
Size	258 (bytes)
Received Time	2013-12-27 23:03:05
Included In Blocks	277316 (2013-12-27 23:11:54 +9 minutes)

Inputs and Outputs	
Total Input	0.1 BTC
Total Output	0.0995 BTC
Fees	0.0005 BTC
Estimated BTC Transacted	0.015 BTC

Bitcoin: Transaction

- The unspent transaction output (UTXO) is the fundamental building block of a bitcoin transaction:
  - UTXOs are indivisible chunks of currency recorded in the blockchain;
  - The UTXO set grows as new UTXOs are created and shrinks when UTXOs are consumed;
  - Every transaction represents a change in the UTXO set;
- A user's "balance" is the sum of all UTXO that user's wallet can spend; it is scattered among hundreds of transactions and blocks.
  - The concept of a balance is created by the wallet application, it does not exist in the blockchain.

Bitcoin: UTXO Values

- The Bitcoin currency is the bitcoin BTC (26 487.02 EUR as of Oct 9th);
- A bitcoin can be divided down to eight decimal places as satoshis (10^{-8} BTC);
- An UTXO can have an arbitrary integer value:
    - Outputs are **discrete** and **indivisible** units of value;
    - They are denominated in **integer** satoshis;
    - An unspent output can only be consumed in its **entirety** by a transaction.
- Strategies to satisfy the purchase amount:
    - Combining several smaller units finding the exact change;
    - Using a single unit larger than the transaction value and making change.

Bitcoin: Coinbase Transaction

- A transaction consumes previously recorded UTXOs and creates new UTXOs that can be consumed by a future transaction.
- **Exception**: the coinbase transaction, which is the first transaction in each block:
    - This transaction is placed there by the "winning" miner;
    - It creates brand-new bitcoin payable to that miner as a reward;
    - It does not consume UTXO;
    - This is how bitcoins money supply is *created* during the mining process.
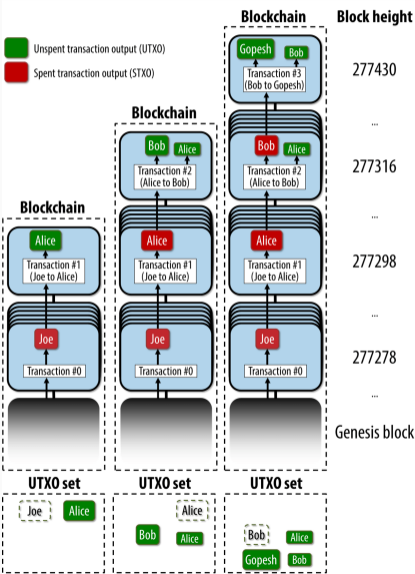
Bitcoin: Transaction Inputs and Outputs

Data structure representing a transaction:

Version	# Input	Inputs	# Output	Outputs	Locktime

Input

- Transaction Hash: Pointer to the transaction holding the UTXO to spend;
- Output Index: UTXO index in the previous transaction;
- Unlocking script: Script with unlocking conditions.

Output

- Value
- Locking script (or witness script): defines the spending conditions.

Bitcoin: Transaction Input

- Transaction inputs identify (by reference) which UTXO will be consumed;
- For each UTXO to consume, the wallet creates one input pointing to the UTXO and unlocks it with an unlocking script.
- Reference:
    - transaction hash;
    - sequence number where the UTXO is recorded in the blockchain;
    - e.g.: 5becc6f42c4796afd87980482c63f4069ad03e1e216ff9a1cb7af03a65d4d133:108
- Unlocking script:
    - Often, it is a digital signature and public key proving ownership of the bitcoin;
    - However, more complex scripts can be defined (e.g., those enabling layer-2 systems such as the Lightning Network).
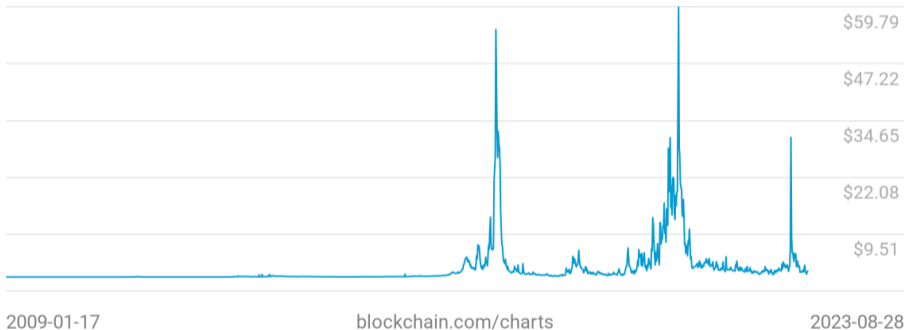
- Transactions fees:
    - Compensate the bitcoin miners for securing the network;
    - Serve as a security mechanism (making flooding economically infeasible);
    - Serve as an incentive to include (mine) a transaction into the next block;
    - Are **implied** as the difference between inputs and outputs;
    - Not mandatory;
- Fees are collected by the miner who mines the block that includes the transaction:
    - Miners prioritize transactions based on many different criteria, including fees.
- Most wallets calculate and include transaction fees automatically;
    - Based on the size of the transaction in kilobytes and other criteria.

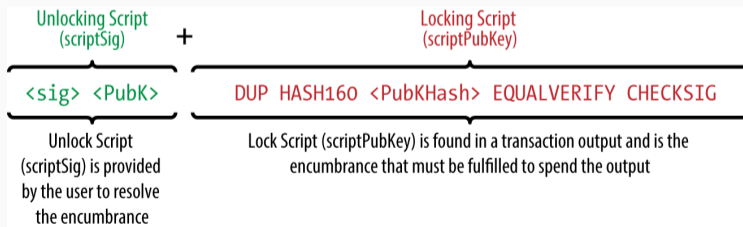# Bitcoin: Average Transaction Fee

FEES USD PER TRANSACTION
$1.34

$59.79
$47.22
$34.65
$22.08
$9.51

2009-01-17   blockchain.com/charts   2023-08-28

Source: blockchain.com/explorer

Bitcoin: Transaction Script

- The transactions script language is called Script;
- Designed to be limited in scope and executable on a range of hardware;
    - It's a stack-based execution language;
    - No loops or complex flow control capabilities other than condition flow control;
    - The language is not Turing Complete;
    - Not a general-purpose language.
    - Stateless: All information needed is contained within the script.

Unlocking Script (scriptSig) + Locking Script (scriptPubKey)

`<sig> <PubK>` `DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG`

Unlock Script (scriptSig) is provided by the user to resolve the encumbrance

Lock Script (scriptPubKey) is found in a transaction output and is the encumbrance that must be fulfilled to spend the output

Bitcoin: Script Example

- Locking script: *3 OP_ADD 5 OP_EQUAL*;
- Unlocking script: *2*;
- Resulting script: *2 3 OP_ADD 5 OP_EQUAL*;
- When executed, the result is *OP_TRUE*, i.e., the transaction is valid.

Bitcoin: Script Example

- Locking script: *3 OP_ADD 5 OP_EQUAL*;
- Unlocking script: *2*;
- Resulting script: *2 3 OP_ADD 5 OP_EQUAL*;
- When executed, the result is *OP_TRUE*, i.e., the transaction is valid.

Stack: 2 3
Script: *OP_ADD* 5 OP_EQUAL;

Bitcoin: Script Example

- Locking script: *3 OP_ADD 5 OP_EQUAL*;
- Unlocking script: *2*;
- Resulting script: *2 3 OP_ADD 5 OP_EQUAL*;
- When executed, the result is *OP_TRUE*, i.e., the transaction is valid.

Stack: 2 3   Stack: 5
Script: *OP_ADD* 5 OP_EQUAL; Script: *5* OP_EQUAL;

Bitcoin: Script Example

- Locking script: *3 OP_ADD 5 OP_EQUAL*;
- Unlocking script: *2*;
- Resulting script: *2 3 OP_ADD 5 OP_EQUAL*;
- When executed, the result is *OP_TRUE*, i.e., the transaction is valid.

Stack: 2 3
Script: *OP_ADD* 5 OP_EQUAL;

Stack: 5
Script: *5* OP_EQUAL;

Stack: 5 5
Script: *OP_EQUAL*;

Bitcoin: Script Example

- Locking script: *3 OP_ADD 5 OP_EQUAL*;
- Unlocking script: *2*;
- Resulting script: *2 3 OP_ADD 5 OP_EQUAL*;
- When executed, the result is *OP_TRUE*, i.e., the transaction is valid.

Stack: 2 3	Stack: 5	Stack: 5 5	Stack: TRUE
Script: *OP_ADD* 5 OP_EQUAL;	Script: *5* OP_EQUAL;	Script: *OP_EQUAL*;	Script: -

Bitcoin: Pay to Public Key Hash (P2PKH)

- The majority of transactions spend output locked with P2PKH;
- Outputs contain a locking script that locks the output to a public key hash (i.e., an address);
- The unlocking script includes a public key and a digital signature.

Example:

- Locking script:
  OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
- Unlocking script: *<Signature> <Public Key>*

Bitcoin: Advanced Scripting

- Multisignature:
  - *N* public keys are recorded in the script and at least *M* of those must provide signatures to unlock the funds (*M*-of-*N* scheme);
  - standard multisignature scripts are limited to at most 15 listed public keys;
  - locking script: *M <PubKey1> ... <PubKeyN> N CHECKMULTISIG*.
- Pay-to-Script-Hash (P2SH):
  - Simplifies the use of complex scripts;
  - Complex locking script is replaced with its cryptographic hash;
  - To spend the UTXO, a redeem script that matches the hash must be provided;
    - **Redeem Script:** 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
    - **Locking Script:** HASH160 <20-byte hash of redeem script> EQUAL
    - **Unlocking Script:** Sig1 Sig2 <redeem script>
  - The P2SH specification is not recursive.

Bitcoin: Benefits of P2SH

- P2SH allows to encode a script hash as an address (whose Base58Check-encoded starts with *3*);
- So, any wallet can transact to P2SH as if it were a bitcoin address;
- Benefits:
  - Complex scripts are replaced by shorter fingerprints;
  - Scripts can be coded as an address (the sender does not need complex engineering to implement P2SH);
  - The burden of constructing the script shifted to the recipient, not the sender;
  - The burden in storage for the long script moved from the output to the input;
  - The transaction fee cost moved from the sender to the recipient.

Bitcoin: Absolute Timelock

- Timelocks enable complex multi-step smart contracts;
- Absolute timelocks specify an absolute point in time
    - e.g., block height = 10000, timestamp = 7 760 000 s;
- *nLocktime* (a transaction field): block height or timestamp until which the transaction is not valid.
    - However, it does not make it impossible to spend the UTXOs in the transaction until that time (beware of double spending!);

Bitcoin: Absolute Timelock

- Timelocks enable complex multi-step smart contracts;
- Absolute timelocks specify an absolute point in time
    - e.g., block height = 10000, timestamp = 7 760 000 s;
- *nLocktime* (a transaction field): block height or timestamp until which the transaction is not valid.
    - However, it does not make it impossible to spend the UTXOs in the transaction until that time (beware of double spending!);
- *CHECKLOCKTIMEVERIFY* (CLTV) introduced by BIP-65:
    - Restricts specific UTXO so that it can only be spent in a future transaction (either block height or timestamp);
    - Example: <now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 …

Bitcoin: Relative Timelock

- They allow to impose a time constraint on one transaction that is dependent on the elapsed time from the confirmation of a previous transaction.
- *nSequence* (a transaction field, with value less than 2^{31}):
  - The transaction inputs is only valid once the input has aged by the relative timelock amount (e.g., it is only valid when at least 30 blocks have elapsed from the time the UTXO referenced in the input was mined).

Bitcoin: Relative Timelock

- They allow to impose a time constraint on one transaction that is dependent on the elapsed time from the confirmation of a previous transaction.
- *nSequence* (a transaction field, with value less than 2^{31}):
  - The transaction inputs is only valid once the input has aged by the relative timelock amount (e.g., it is only valid when at least 30 blocks have elapsed from the time the UTXO referenced in the input was mined).
- *CHECKSEQUENCEVERIFY* (CSV):
  - When evaluated in a UTXO's redeem script allows spending only in a transaction whose input nSequence value is greater than or equal to the CSV parameter.
- Relative timelocks with CSV are especially useful when several (chained) transactions are created and signed, but not propagated, when they are kept "off-chain".
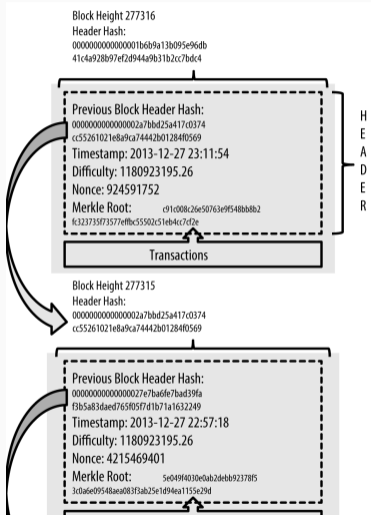
Bitcoin

Block Definition

Bitcoin: Transaction Pool

- As transactions are received and verified, they are added to the **transaction pool** and relayed to the neighboring nodes to propagate on the network.
- Some node implementations also maintain a separate pool of orphaned transactions, i.e., if a transaction's inputs refer to a transaction that is not yet known.
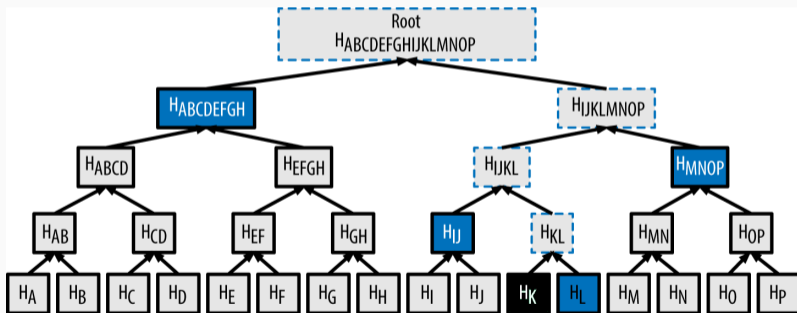
Bitcoin: Block

- A block groups transactions for inclusion in the blockchain;
- The block header is 80 bytes; the average transaction is at least 250 bytes and the average block contains more than 500 transactions.
- Block Header:
    - Reference to a previous block *header* hash;
    - Set of metadata (difficulty, timestamp, and nonce) related to the mining;
    - The Merkle tree root (summarizes all the transactions in the block).
- Genesis Block: The first block in the blockchain is called the genesis block

A Merkle tree is a binary tree used to efficiently summarize and verify the integrity of a large number of transactions.

A node can prove that transaction K is included in the block by producing a Merkle path, which consists of 4 hashes: H_L, H_{IJ}, H_{MNOP} e $H_{ABCDEFGH}$.

Only $\log_2(N)$ 32-byte hashes needed.

Ethereum

Ethereum: A General Purpose Blockchain

- Bitcoin tracks the state of units of bitcoin and their ownership.
- Ethereum:
    - Is a distributed state machine that tracks the changes a **general-purpose data store**, i.e., key–value tuples;
    - Has memory that stores both code and data;
    - Can load code into its state machine and run that code, storing the resulting state changes in its blockchain;

Ethereum: A General Purpose Blockchain

- **Transactions** include (among other things) a sender, recipient, value, and data payload;
- State transitions are processed by the **Ethereum Virtual Machine** (EVM)
    - A stack-based virtual machine that executes bytecode;
    - EVM programs are called "smart contracts";
    - EVM can compute any algorithm that can be computed by any Turing machine, given the limitations of finite memory (i.e., Turing completeness).
- Ethereum's state is **locally stored** on each node as a database;
    - Transactions and system state in a serialized hashed data structure called a Merkle Patricia Tree.

Ethereum: Merkle Patricia Tree

- A Merkle Tree helps to store static and immutable objects (like transactions);
- **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric (Patricia) Merkle Trie:
    - Combines the idea of Radix Trie (or, Patricia Trie) and a Merkle Tree;
    - A data structure to efficiently store key–value pairs;
    - Enables to check the inclusion of a key-value pair;
    - Enables to check the integrity of data;
    - Differently from a hash table, there is no key collision;
    - Ethereum Doc: Patricia Merkle Tree.

Ethereum: The ether Currency

- The Ethereum currency is the ether (ETH or Ξ);
- As of Oct 9th, 1 ETH = 1545.95 EUR;
- As in Bitcoin, we consider sub-multiples of ether;
- The smallest unit is called wei:

Value (in wei)	Exponent	Common name	SI name
1	1	wei	Wei
1,000	10^3	Babbage	Kilowei or femtoether
1,000,000	10^6	Lovelace	Megawei or picoether
1,000,000,000	10^9	Shannon	Gigawei or nanoether
1,000,000,000,000	10^{12}	Szabo	Microether or micro
1,000,000,000,000,000	10^{15}	Finney	Milliether or milli
1,000,000,000,000,000,000	*10^{18}*	*Ether*	*Ether*
1,000,000,000,000,000,000,000	10^{21}	Grand	Kiloether
1,000,000,000,000,000,000,000,000	10^{24}		Megaether

Ethereum: EVM and the World Computer

- Ethereum as a decentralized world computer;
- Smart contracts run on the EVM;
- EVM:
    - Operates **as** if it were a global, single instance computer, running everywhere;
    - Each node runs a local copy of the EVM to validate the contract execution;
    - The blockchain records the changing state of this world computer as it processes transactions and smart contracts;
    - Ether is meant to pay for running smart contracts on the EVM.

Ethereum: An Account-based blockchain

Ethereum stores an account-based state;

Externally Owned Accounts

- Have an address;
- Have a private key;
- Can control funds and contracts;
- Can't store smart contract code;
- A user can manage multiple accounts (e.g., to improve anonymity);

Contract Accounts

- Have an address;
- No private key (cannot initiate transactions, but can react by calling other contracts);
- Can send and receive ether;
- Have smart contract code;
- Incoming transactions causes the contract to run in the EVM;

Ethereum

Transactions

Ethereum: Transactions

- Transactions are signed messages originated by an externally owned account and recorded on the blockchain;
- They can trigger a change of state, or the execution of a smart contract;
- A Transaction contains:
    - **Nonce**: to prevent message replay
    - **Gas price**: *wei* the originator is willing to pay;
    - **Gas limit**: the maximum amount of gas the originator is willing to buy;
    - **Recipient**: destination address
    - **Value**: ether to send;
    - **Data**: binary payload;
    - v, r, s: Components of the sender's ECDSA digital signature;

Ethereum: Nonce

- A scalar value equal to the number of transactions sent from this address;
- Of utmost importance for account-based protocols;
    - Differently from UTXO-based protocols;
- Why?
    1. Ethereum processes transactions sequentially, based on the nonce;
        - Not in order transaction will be stored in the mempool, while waiting for the missing nonces to appear.
    2. Allows Ethereum to avoid processing twice the same transaction, and prevents replay attacks;

Ethereum: Gas

- The Ethereum's computation model requires counter-measures to avoid denial-of-service attacks or resource-devouring transactions;
- Gas allows to control the amount of resources that a transaction can use;
- Gas is a separate virtual currency with its own exchange rate against ether;
  - To protect the system from the ether value volatility;
  - The price is measured in wei per gas unit.
- In the gasPrice field, the originator sets the price he is willing to pay for gas;
  - The higher the gasPrice, the faster the transaction is likely to be confirmed;
  - Fee-free transaction are also allowed.
- The gasLimit gives the maximum number of units of gas the transaction originator is willing to buy in order to complete the transaction;

Ethereum: Gas

- If the transaction's destination is a contract, the amount of gas cannot be determined with accuracy, but only estimated;
- A contract can evaluate different conditions that lead to different execution paths (with different total gas costs);
- If the gas limit is exceeded during computation, the following series of events occurs:
    - An out-of-gas exception is thrown;
    - The state of the contract prior to execution is restored (reverted);
    - All ether used to pay for the gas is taken as a transaction fee; *it is not refunded.*

Ethereum: Recipient

- The recipient of a transaction is specified in the *to* field;
- The address can be an externally owned account or a contract address;
- Ethereum *does not* validate recipient addresses in transactions;
- Sending a transaction to the wrong address will probably **burn** the ether sent, rendering it forever inaccessible.

Ethereum: Value and Data

- Transactions can have:
    - Only value: transaction is a payment;
    - Only data: transaction is an invocation;
    - Value and Data: transaction is both a payment and an invocation;
    - None: possible, but it is only a waste of gas.
- If the transaction has a value:
    - Destination is a contract: the EVM will execute the contract and call the function named in the data payload (or a fallback function);
        - If the function is payable, will execute it to determine what to do next;
        - If there is no fallback, the transaction increase the balance of the contract;
        - A contract can reject incoming payments;
    - Otherwise: Value is added to the balance of the address (it is created if the address has not been seen before);

Ethereum: Value and Data

- If the transaction has data:
  - It is most likely addressed to a contract address;
  - It is interpreted as a function invocation, encoding:
    - A function selector;
    - The function arguments;
- Contract creation transactions are sent to a **special destination address** called the zero address (`0x0`);
  - The creation transaction contains the compiled contract bytecode as data payload;
  - Optionally, it can include an ether amount, to set the new contract with a starting balance.

Ethereum: Signature

- Ethereum uses ECDSA (Elliptic Curve Digital Signature Algorithm) to sign a digest (the hash) of the transaction;
- Only who manages the private key (through a wallet) can sign the transaction;
- The signature composed of two values, (r, s), that can be stored in the transaction;
- To verify the signature, we need: (r, s), the serialized transaction, and the corresponding public key.

Ethereum: Transaction Propagation

- The originating node creates a signed transaction;
- The transaction is validated and transmitted to all the other nodes directly connected to the node;
- On average, each node maintains connections to at least 13 other nodes;
- Valid transactions will eventually be included in a block;
- Once mined into a block, transactions either modify the balance of an account or invoke contracts that change their internal state.
- Changes are recorded in the form of a transaction receipt:
  - The transaction receipt contains log entries that provide information about the actions that occurred during the execution of the transaction.

Ethereum

Smart Contracts

Ethereum: Smart Contract

Smart contracts are programs running on the EVM

- Immutable: Once deployed, the code of a smart contract cannot change;
- Deterministic: The outcome of the execution is the same for everyone who runs it;
- EVM context: Smart contracts operate with a very limited execution context, they can access their own state, the context of the transaction that called them, and some information about the most recent blocks.

Ethereum: Smart Contract Life Cycle

- Smart contracts are written in a high-level language (e.g., Solidity, Vyper);
- They are compiled to the low-level bytecode that runs in the EVM;
- They are deployed using the contract creation transaction;
- Each contract is identified by an address, derived from the contract creation transaction as a function of the originating account and nonce;
  - Contracts only run if the first calling transaction is from an external account;
  - Contract can call other contracts;
  - Smart contracts are not executed in parallel; Transactions are atomic,
- If properly programmed, contracts can be deleted:
  - This removes the code and its internal state (storage) from the contract address;
  - To delete a contract, it has to call the opcode *SELFDESTRUCT*;
  - The operation costs negative gas (i.e., a gas refund);

```solidity
// Our first contract is a faucet!
contract Faucet {

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

         // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }

    // Accept any incoming amount
    function () public payable {}
}
```

```
$ solc --optimize --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binary:
6060604052341561000f57600080fd5b60cf8061001d6000396000f30060606040526004361060603e5
763ffffffff7c010000000000000000000000000000000000000000000000000000000006000350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e60043567016345785d8a00008111156
06357600080fd5b73ffffffffffffffffffffffffffffffffffffffff331681156108fc0282604051
6000604051808030381858888f19350505050505151560a057600080fd5b505600a165627a7a723058203
556d79355f2da19e773a9551e95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
```

Ethereum: ABI

- Ethereum uses the **application binary interface** (ABI) to encode contract calls for the EVM and to read data out of transactions.
- The purpose of an ABI is to define the functions in the contract that can be invoked and describe how each function will accept arguments and return its result;
- A contract's ABI is specified as a JSON array of:
  - function description, i.e., an object with fields type, name, inputs, outputs, constant, and payable;
  - An event description, with type, name, inputs, and anonymous.
- Events are objects used to construct logs, e.g., transaction receipt.

```
$ solc --abi Faucet.sol
======= Faucet.sol:Faucet =======
Contract JSON ABI
[{"constant":false,"inputs":[{"name":"withdraw_amount","type":"uint256"}], \
"name":"withdraw","outputs":[],"payable":false,"stateMutability":"nonpayable", \

"type":"function"},{"payable":true,"stateMutability":"payable", \
"type":"fallback"}]
```

Ethereum: Solidity

- Different data types: Boolean, Integer, Fixed point, Address, Byte array, Enum, Array, Struct, Mapping;
- Set of global objects: msg, transaction, block, address
- Address: contains methods to send value and call other functions;
- Solidity uses an object oriented programming approach:
    - principal data type is contract (includes data and methods);
    - interface (data and only function declaration);
    - library (contract to be deployed only once and used by other contracts);

Ethereum: Functions in Solidity

- Function syntax in Solidity:

  function FunctionName([parameters]) public|private|internal|external
  [pure|constant|view|payable] [modifiers] [returns (return types)]

- Visibility:
    - external functions are as public, but cannot be called within the contract;
    - internal are only accessible from within the contract; likewise, private functions, which however are not inherited from derived classes.
- Behavior:
    - pure: does neither read nor write any variable;
    - constant or view: does not modify any state;
    - payable: accepts payments; if not declared as payable, payments are rejected by default.
- Modifiers: allow to create conditions that apply to many functions, e.g., to restrict access

```solidity
contract owned {
        address owner;

        // Contract constructor: set owner
        constructor() {
                owner = msg.sender;
        }

        // Access control modifier
        modifier onlyOwner {
            require(msg.sender == owner);
            _;
        }
 }
contract mortal is owned {
        // Contract destructor
        function destroy() public onlyOwner {
                selfdestruct(owner);
        }
}
```

- The transaction receipt provides information about the actions occurred during execution;
- Events are used to construct these logs;
- Clients and DApp can "watch" for specific events;
- Event objects take arguments that are serialized and recorded in the transaction logs;
- Events are declared in the smart contract, then can be emitted;

```
contract Faucet is mortal {
        event Withdrawal(address indexed to, uint amount);
        event Deposit(address indexed from, uint amount);
```

```
function withdraw(uint withdraw_amount) public {
    [...]
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}
// Accept any incoming amount
function () public payable {
    emit Deposit(msg.sender, msg.value);
}
```

- Gas drives smart contract execution;
- Exhausting gas results in a out-of-gas exception, that reverts code execution (but fees are payed);
- Common practices:
    - Avoid dynamically sized arrays: operations or searches introduces the risk of using too much gas;
    - Avoid calls to other contracts (especially when the gas cost of their functions is not known);

- Many Ethereum smart contracts contain serious vulnerabilities:
  - Suicidal contracts: i.e., that can be killed by arbitrary addresses;
  - Greedy contracts: i.e., that can reach a state in which they cannot release ether;
  - Prodigal contracts: i.e., that can be made to release ether to arbitrary addresses;
- Vulnerabilities are not intentionally introduced;
- Vyper is designed to make it easier to write secure code;
- Vyper deliberately omits some of Solidity's features.

Ethereum

Fungible and Non-fungible Tokens

Tokens

- A tokens is a blockchain-based abstractions that can be **owned** and that **represent** assets, currency, or access rights;
- Uses:
    - Currency
    - Resource: earned or produced;
    - Asset: i.e., ownership of an intrinsic or extrinsic, tangible or intangible asset;
    - Access right;
    - Equity: A token can represent shareholder equity in a digital/legal entity;
    - Voting right;
    - Collectible: A digital collectible (e.g., CryptoPunks) or physical collectible (e.g., a painting);
    - Identity: Digital or legal identity;
    - Attestation: i.e., certification or attestation by a decentralized reputation system
    - Utility: to access or pay for a service.

Tokens

- Fungible: if we can substitute any single unit for another without any difference in its value or function.
- Intrinsicality:
  - Some tokens represent digital items that are intrinsic to the blockchain;
  - Other tokens can represent extrinsic things, whose ownership is not ruled by the blockchain protocol.
    - They carry additional counterparty risk because they are held by external custodians and registries;
    - Blockchain-based tokens can convert extrinsic assets into intrinsic assets, e.g., from equity in a corporation (extrinsic) to an equity or voting token in a DAO;

Ethereum: The ERC20 Token Standard

- The ERC20 standard defines a common interface for contracts implementing fungible tokens;
- Required functions and events:
    - totalSupply, transfer, transferFrom, approve (authorizes an address to execute several transfers up to an amount), allowance (remaining amount that the spender is approved to withdraw)
    - Transfer, Approval
- Any ERC20 implementation contains two data structures:
    - balances: tracks balances;
    - allowances: tracks allowances;

- Issues:
  - Subtle differences between tokens and ether itself: token transfers occur within the specific token contract state;
  - Difficulty to track balances of all ERC20 contracts;
  - To send tokens, you need to pay transactions in ether.

Ethereum: The ERC721 Token Standard

- The ERC721 proposal is for a standard for non-fungible tokens (NFTs);
- NFTs are also known as *deeds*;
- Whereas ERC20 tracks the balances that belong to each owner,
- ERC721 tracks each deed ID and who owns it.

Ethereum

Oracles

- Smart contract are *deterministic* and can access only data on the transaction or the blockchain;
- Oracles provide a trustless way of getting extrinsic (off-chain) information onto the blockchain;
- Pattern:
    - Collect data from an off-chain source;
    - Transfer the data on-chain with a signed message;
    - Make the data available by putting it in a smart contract's storage.

Algorand

Algorand

An Account-based Blockchain

- Algorand uses Ed25519 high-speed, high-security elliptic-curve signatures (i.e., EdDSA);
- A random seed is used to generate a pair of public key and private key;
- The public key is transformed into an Algorand address by adding a checksum and encoding in base32;

Algorand: Account

- Algorand currency is Algo (1 Algo = 0.087 EUR);
- Accounts are entities on the Algorand blockchain;
- They are associated with specific on-chain data, e.g., a balance;
- An address is the identifier for an Algorand account;
- After generating an address, sending Algos to the address will initialize its state on the blockchain.
  - Every account on Algorand must have a minimum balance of 0.1 Algos;
  - By default, they are set to offline (i.e., not participating in Algorand consensus);
  - A valid address can also be produced from a compiled smart contract;

Algorand

Transactions, Algos, and Assets

Algorand: Transactions

- There are 7 transaction types in Algorand:
    - **Payment**: sends Algos (the native currency) from one account to another;
    - **Key Registration**: registers an account to participate in Consensus (i.e., online);
    - **Asset Configuration**: to create, modify parameters, or destroy an asset;
    - **Asset Transfer**:
        - to opt-in to receive a specific type of Algorand Standard Asset (ASA);
        - transfer an ASA;
        - revoke an ASA from a specific account;
    - **Asset Freeze**: to freeze/unfreeze the asset holdings for a specific account. A frozen account cannot send or receive the frozen asset.
    - **Application Call**: to call smart contracts;
    - **State Proof**: submitted to the network during the consensus process.
- Atomic Transfer: the group of transactions that are part of the transfer either all succeed or all fail.

Algorand: Algorand Standard Asset (ASA)

- An Algorand Standard Assets (ASA): represents an on-chain assets;
- It can represent, e.g., stablecoins, loyalty points, system credits, and in-game points;
- It can also represent single, unique assets (i.e., NFT);
- There is also the possibility to place transfer restrictions on an ASA;
- For every asset an account creates or owns, its minimum balance is increased by 0.1 Algos;
- Before a new asset can be transferred to a specific account the receiver must **opt-in** to receive the asset.

Algorand

Smart Contracts

Algorand: Algorand Smart Contracts

- Algorand Smart Contracts are programs that operate on layer-1; Two categories:
    - Smart Contract (or, stateful contract);
    - Smart Signatures (or, stateless contract);
- Both written in Transaction Execution Approval Language (TEAL), which is an assembly-like language
    - PyTEAL allows to use Python to define and compile contracts;
- Interpreted by the Algorand Virtual Machine (AVM);

Algorand: Smart Contracts

- Once deployed, are remotely callable from any node in the blockchain;
- The on-chain instantiation is referred to as an Application and assigned an Application Id;
- Applications are triggered by a specific type of transaction (i.e., Application Call).
- They handle the decentralized logic of a dApp;
  - Applications can modify state associated with the application (global state) or a per (application, account) (local state);
  - Applications can access on-chain values (e.g., balances, asset configuration, block time).
  - Applications can execute application call transactions;
  - Applications have an associated Application Account that can hold Algos or ASAs balances (can be on-chain escrow accounts).

Algorand: Smart Signatures

- Smart signatures contain logic that is used to sign transactions;
  - A smart signature is submitted with a transaction;
  - The logic is not remotely callable;
  - If a smart signature's logic fails, the associated transaction will not be executed.
- When compiled, smart signatures produce an account;
  - These accounts can hold Algos or assets.
  - These funds are only allowed to leave the account only from the account that executes the smart signature.
- Smart signatures can also also be used to delegate some portion of authority to another account.
  - An account can sign the smart signature which can later be used to sign a transaction from the original signer's account.

Algorand

Properties of Algorand's Network Architecture

Algorand: Network Architecture

- Blockchain trilemma: Decentralization, Scalability, and Security.
- To optimize decentralization and transaction throughput, Algorand uses relay nodes and participation nodes:
    - Relay nodes:
        - Serve as network hubs:
        - maintain connections to many other nodes;
        - have high-bandwidth connection;
        - aimed to reduce the number of communication hops.
        - Accumulate messages from participation nodes and other relays;
        - Perform deduplication, signature checks, and other validation steps;
        - Propagating valid messages.
    - Participation nodes:
        - connected to fewer nodes (mostly relay nodes);
        - hold participation keys for proposing and voting on blocks;

Algorand: Network Architecture

- Blockchain trilemma: Decentralization, Scalability, and Security.
- To ensure security, relay nodes must be both diverse and decentralized.
  - Geographical distribution: Relay nodes must exist in different countries;
  - Should be located at key internet exchange points;
  - Organizations running nodes from network launch include universities, nonprofits, members of the cryptocurrency ecosystem, and traditional financial and Internet infrastructure providers;
  - Decentralized Consensus Protocol.

Matteo Nardelli