Module 5

Ethereum

The Ethereum network

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types, based on the requirements and usage.

1. The mainnet

The **mainnet** is the current live network of Ethereum. Its network ID is 1 and its chain ID is also 1. The network and chain IDs are used to identify the network.

2. Testnets

- There is a number of testnets available for Ethereum testing.
- The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain.
- Moreover, being test networks, they also allow experimentation and research.
- The main testnet is called *Ropsten*, which contains all the features of other smaller and special-purpose testnets that were created for specific releases. For example, other testnets include *Kovan* and *Rinkeby*, which were developed for testing Byzantium releases. The changes that were implemented on these smaller testnets have also been implemented in Ropsten. Now the Ropsten test network contains all properties of Kovan and Rinkeby.

3. Private nets

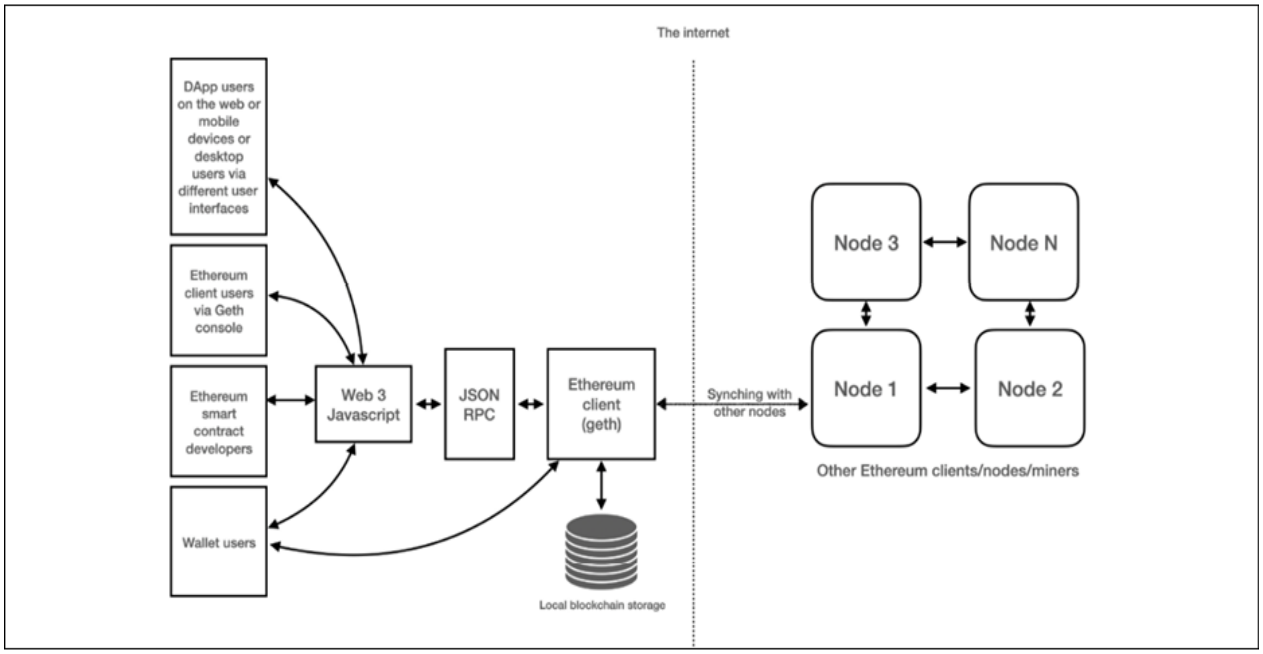
As the name suggests, these are private networks that can be created by generating a new genesis block. This is usually the case in private blockchain networks, where a private group of entities start their blockchain network and use it as a permissioned or consortium blockchain.

Components of the Ethereum ecosystem

- The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer Ethereum network.

- Secondly, there's an Ethereum client (usually Geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management.
- The local copy of the blockchain is synchronized regularly with the network. Another component is the web3.js library that allows interaction with the geth client via the **Remote Procedure Call (RPC)** interface.

The overall Ethereum ecosystem architecture is visualized in the following diagram:


The internet
DApp users on the web or mobile devices or desktop users via different user interfaces
Ethereum client users via Geth console
Ethereum smart contract developers
Wallet users
Web 3 Javascript
JSON RPC
Ethereum client (geth)
Local blockchain storage
Synching with other nodes
Node 3
Node N
Node 1
Node 2
Other Ethereum clients/nodes/miners

A list of elements present in the Ethereum blockchain is presented here:

- Keys and addresses
- Accounts
- Transactions and messages
- Ether cryptocurrency/tokens
- The EVM
- Smart contracts and native contracts

a. Keys and addresses

- Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether.

- The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas a public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

b. Accounts

- Accounts are one of the main building blocks of the Ethereum blockchain.
- They are defined by pairs of private and public keys.
- Accounts are used by users to interact with the blockchain via transactions.
- A transaction is digitally signed by an account before submitting it to the network via a node.
- Ethereum, being a *transaction-driven state machine*, the state is created or updated as a result of the interaction between accounts and transaction executions.
- All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. Operations performed between and on the accounts represent state transitions.
- The state transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. The transaction fee is calculated, and the sending address is resolved using the signature. Furthermore, the sender's account balance is checked and subtracted accordingly, and the nonce is incremented. An error is returned if the account balance is insufficient.
3. Provide enough ETH (the gas price) to cover the cost of the transaction. This is charged per byte and is incrementally proportional to the size of the transaction. In this step, the actual transfer of value occurs. The flow is from the sender's account to the receiver's account. The account is created automatically if the

destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.

4. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back except for the fee payment, which is paid to the miners.
5. Finally, the remainder (if any) of the fee is sent back to the sender as change and the fee is paid to the miners accordingly. At this point, the function returns the resulting state, which is also stored on the blockchain.

Types of accounts

Two kinds of accounts exist in Ethereum:

- **Externally Owned Accounts (EOAs)**
- **Contract Accounts (CAs)**

The first type is EOAs, and the other is CAs. EOAs are similar to accounts that are controlled by a private key in Bitcoin. CAs are the accounts that have code associated with them along with the private key.

The various properties of each type of account are as follows:

EOAs

- They have a state.
- They are associated with a human user, hence are also called user accounts.
- EOAs have an ether balance.
- They are capable of sending transactions.
- They have no associated code.
- They are controlled by private keys
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

CAs

- They have a state.
- They are not intrinsically associated with any user or actor on the blockchain.

- CAs have an ether balance.
- They have associated code that is kept in memory/storage on the blockchain. They have access to storage.
- They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within CAs can be of any level of complexity.
- CAs can maintain their permanent states and can call other contracts.
- CAs cannot start transaction messages.
- CAs can initiate a call message.
- CAs contain a key-value store.
- CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

c. Transactions and messages

A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation. Transactions can be divided into two types based on the output they produce:

- **Message call transactions**: This transaction simply produces a message call that is used to pass messages from one CA to another.
- **Contract creation transactions**: As the name suggests, these transactions result in the creation of a new CA. This means that when this transaction is executed successfully, it creates an account with the associated code.

Both of these transactions are composed of some standard fields, which are described as follows:

- **Nonce**: The nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction.
- **Gas price**: The **gas price** field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction.

  "Wei is the smallest denomination of ether; therefore, it is used to count ether"

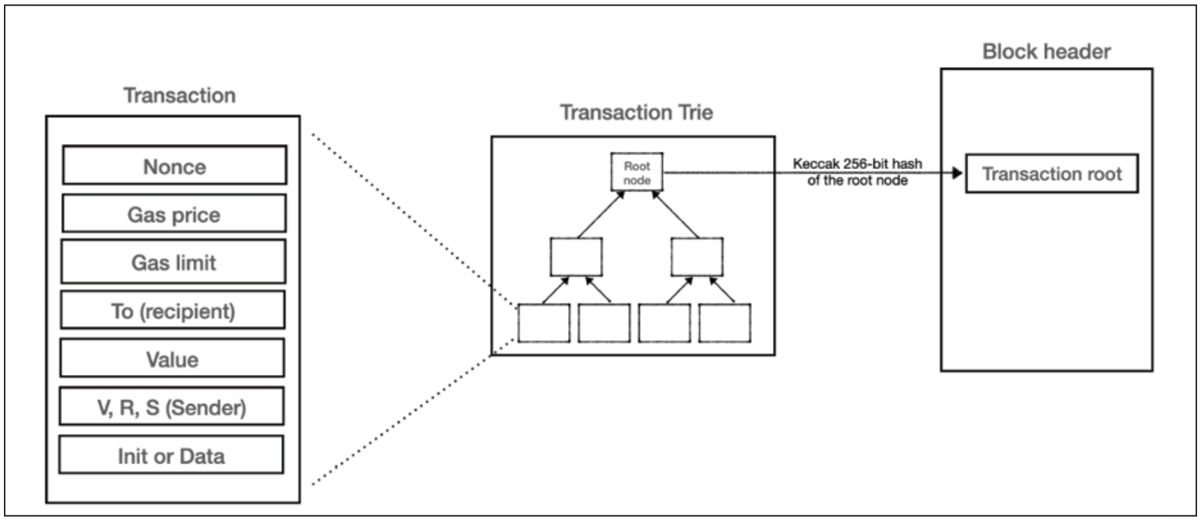
- **Gas limit**: The **gas limit** field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction

- **To**: As the name suggests, the **To** field is a value that represents the address of the recipient of the transaction. This is a 20 byte value.
- **Value**: Value represents the total number of Wei to be transferred to the recipient.
- **Signature**: The signature is composed of three fields, namely **V**, **R**, and **S**. These values represent the digital signature (*R*, *S*) and some information that can be used to recover the public key (*V*). Also, the sender of the transaction can also be determined from these values. The signature is based on the ECDSA scheme.

  To sign a transaction, the ECDSASIGN function is used, which takes the message to be signed and the private key as an input and produces *V*, a single-byte value; *R*, a 32-byte value; and *S*, another 32-byte value. The equation is as follows:

  ECDSASIGN (Message, Private Key) = (V, R, S)
- **Init**: The **Init** field is used only in transactions that are intended to create contracts, that is, contract creation transactions. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process. The code contained in this field is executed only once when the account is created for the first time, it (init) gets destroyed immediately after that. Init also returns another code section called the *body*, which persists and runs in response to message calls that the CA may receive. These message calls may be sent via a transaction or an internal code execution.
- **Data**: If the transaction is a message call, then the **Data** field is used instead of **init**, and represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This structure is visualized in the following diagram which is then included in a **transaction trie** (a modified **Merkle-Patricia tree** (**MPT**)) composed of the transactions to be included. Finally, the root node of the transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block:



- A block is a data structure that contains batches of transactions. Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. When a mining node starts its operation of verifying blocks, it starts with the highest-paying transactions in the transaction pool and executes them one by one. When the gas limit is reached, or no more transactions are left to be processed in the transaction pool, the mining starts.
- In this process, the block is repeatedly hashed until a valid nonce is found, such that once hashed with the block, it results in a value less than the difficulty target. Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network. This process is similar to Bitcoin's mining process.
- Ethereum's PoW algorithm is called Ethash, and it was originally intended to be ASIC-resistant where finding a nonce requires large amounts of memory.

Contract creation transactions

A contract creation transaction is used to create smart contracts on the blockchain. There are a few essential parameters required for a contract creation transaction. These parameters are listed as follows:

- The sender
- The transaction originator
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- The current depth of the message call/contract-creation stack (current depth means the number of items that are already present in the stack)

Message call transactions

A message call requires several parameters for execution, which are listed as follows:

- The sender
- The transaction originator
- The recipient

- The account whose code is to be executed (usually the same as the recipient)
- Available gas
- The value
- The gas price
- An arbitrary-length byte array
- The input data of the call
- The current depth of the message call/contract creation stack

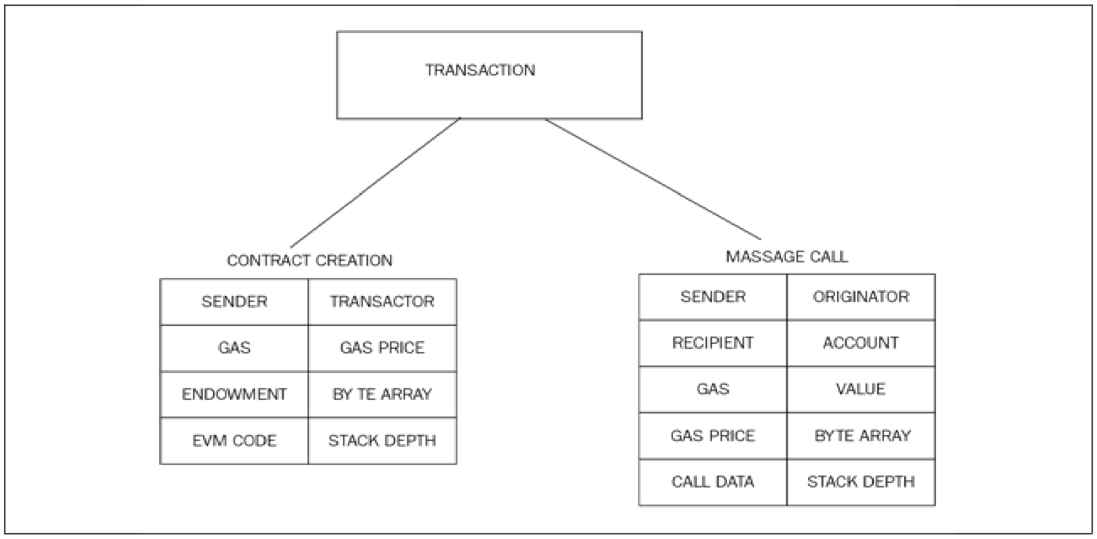
Messages

- Messages, as defined in the yellow paper, are the data and values that are passed between two accounts. A **message** is a data packet passed between two accounts. This data packet contains data and a value (the amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (an **Externally Owned Account**, or **EOA**) in the form of a transaction that has been digitally signed by the sender.
- Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions; however, the main difference is that they are produced by the contracts, whereas transactions are produced by entities external to the Ethereum environment (EOAs).

A message consists of the following components:

- The sender of the message
- The recipient of the message
- Amount of Wei to transfer and the message to be sent to the contract address
- An optional data field (input data for the contract)
- The maximum amount of gas (startgas) that can be consumed

In the following diagram, the segregation between the two types of transactions (**contract creation** and **message calls**) is shown:



Types of transactions and the required parameters for execution

Calls

A call does not broadcast anything to the blockchain; instead, it is a local call and executes locally on the Ethereum node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. Calls are processed synchronously and they usually return the result immediately.

Do not confuse a *call* with a *message call transaction*, which in fact results in a state change. A call basically runs message call transactions locally on the client and never costs gas nor results in a state change. It is available in the web3.js JavaScript API and can be seen as almost a simulated mode of the message call transaction. On the other hand, a *message call transaction* is a write operation and is used for invoking functions in a CA (Contract Account, or smart contract), which does cost gas and results in a state change.

Transaction validation and execution

Transactions are executed after their validity has been verified. The initial checks are listed as follows:

- A transaction must be well formed and RLP-encoded without any additional trailing bytes
- The digital signature used to sign the transaction must be valid
- The transaction nonce must be equal to the sender's account's current nonce
- The gas limit must not be less than the gas used by the transaction
- The sender's account must contain sufficient balance to cover the execution cost

The transaction substate

A transaction substate is created during the execution of the transaction and is processed immediately after the execution completes. This transaction substate is a tuple that is composed of four items. These items are as follows:

- **Suicide set or self-destruct set**: This element contains the list of accounts (if any) that are disposed of after the transaction executes.
- **Log series**: This is an indexed series of checkpoints that allows the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked, or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage
- **Refund balance**: This is the total price of gas for the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to offset the total execution cost partially.
- **Touched accounts**: Touched accounts can be defined as those accounts which are involved any potential state changing operation. Empty accounts from this set are deleted at the end of the transaction. The final state is reached after deleting accounts in the self-destruct set and emptying accounts from the touched accounts set.

State storage in the Ethereum blockchain

At a fundamental level, the Ethereum blockchain is a transaction- and consensus-driven state machine. The state needs to be stored permanently in the blockchain.
components next.

i. The world state

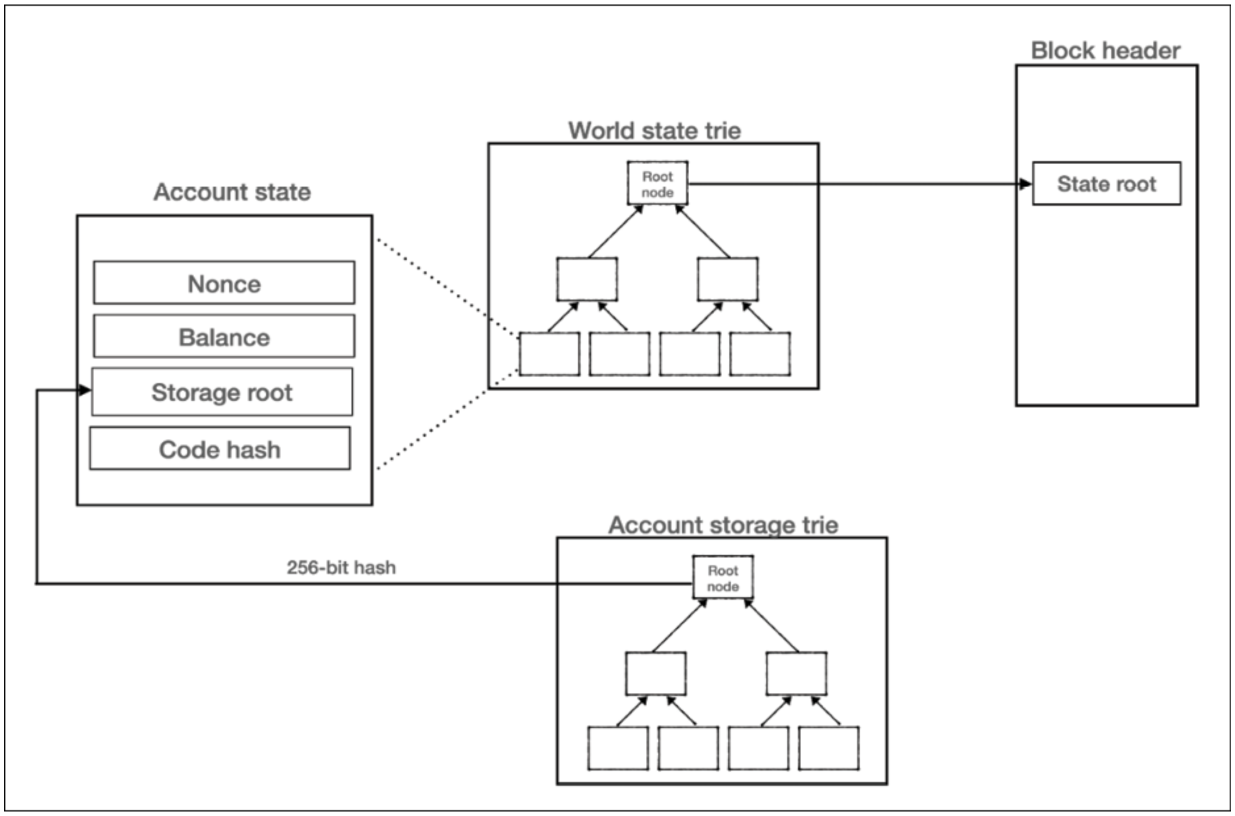
This is a *mapping* between Ethereum addresses and account states. The addresses are 20 bytes (160 bits) long. This mapping is a data structure that is serialized using RLP.

ii. The account state

The account state consists of four fields: nonce, balance, storage root, and code hash, and is described in detail here:

- **Nonce**: This is a value that is incremented every time a transaction is sent from the address. In the case of CAs, it represents the number of contracts created by the account.
- **Balance**: This value represents the number of weis, which is the smallest unit of the currency (ether) in Ethereum, held by the given address.

- **Storage root**: This field represents the root node of an MPT that encodes the storage contents of the account.
- **Code hash**: This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.



The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationship

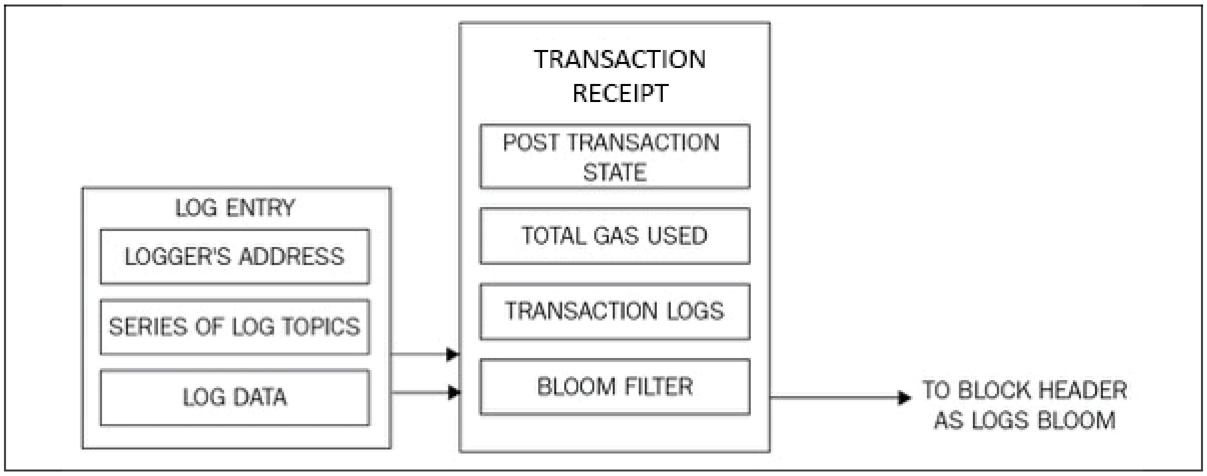
Transaction receipts

Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. It is produced after the execution of each transaction. All receipts are stored in an index-keyed trie. The hash (a 256-bit Keccak hash) of the root of this trie is placed in the block header as the receipt's root. It is composed of four elements as follows:

- **The post-transaction state**: This item is a trie structure that holds the state after the transaction has been executed. It is encoded as a byte array.

- **Gas used**: This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is expected to be a non-negative integer.
- **Set of logs**: This field shows the set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.
- **The bloom filter**: A bloom filter is created from the information contained in the set of logs. A log entry is reduced to a hash of 256 bytes, which is then embedded in the header of the block as the logs bloom. A log entry is composed of the logger's address, log topics, and log data. Log topics are encoded as a series of 32-byte data structures. The log data is made up of a few bytes of data.

This process of transaction receipt generation is visualized in the following diagram:



As a result of the transaction execution process, the state morphs from an initial state to a target state.

Ethereum Blocks and blockchain

Blocks are the main building structure of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

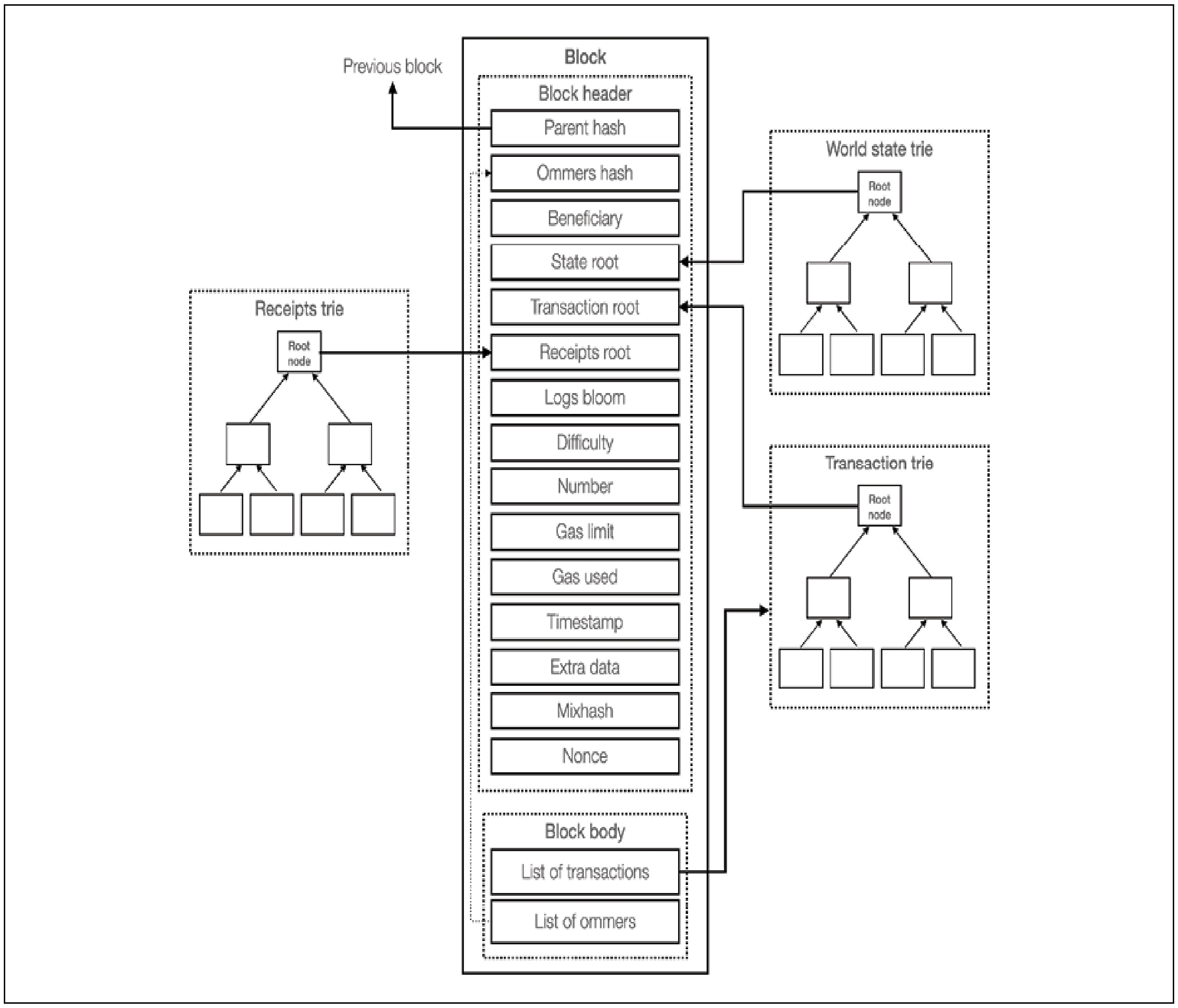
- The block header
- The transactions list
- The list of headers of ommers or uncles

Block header: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are described in detail here:

- **Parent hash**: This is the Keccak 256-bit hash of the parent (previous) block's header.
- **Ommers hash**: This is the Keccak 256-bit hash of the list of ommers (or uncles) blocks included in the block.
- **The beneficiary**: The beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.
- **State root**: The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated once all transactions have been processed and finalized.
- **Transactions root**: The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. The transaction trie represents the list of transactions included in the block.
- **Receipts root**: The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.
- **Logs bloom**: The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.
- **Difficulty**: The difficulty level of the current block.
- **Number**: The total number of all previous blocks; the genesis block is block zero.
- **Gas limit**: This field contains the value that represents the limit set on the gas consumption per block.
- **Gas used**: This field contains the total gas consumed by the transactions included in the block.
- **Timestamp**: The timestamp is the epoch Unix time of the time of block initialization.
- **Extra data**: The extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.
- **Mixhash**: The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **PoW**) has been spent in order to create this block.

- **Nonce**: Nonce is a 64-bit hash (a number) that is used to prove, in combination with the *mixhash* field, that adequate computational effort (PoW) has been spent in order to create this block.

The following diagram shows the detailed structure of the block and block header



The genesis block

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created. It contains 15 items that are described here.

Element	**Description**
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block
Hash	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
Parent hash	0x0000000000000000000000000000000000000000000000000000000000000000
SHA-3 uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined by	0x0000000000000000000000000000000000000000 in 15 seconds
Difficulty	17,179,869,184
Total difficulty	17,179,869,184
Size	540 bytes
Gas used	0
Nonce	0x0000000000000042
Block reward	5 ETH
Uncles reward	0
Extra data	In hex (0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)
Gas limit	5,000

The block validation mechanism

An Ethereum block is considered valid if it passes the following checks:

- If it is consistent with uncles and transactions. This means that all ommers satisfy the property that they are indeed uncles and also if the PoW for uncles is valid.
- If the previous block (parent) exists and is valid.

- If the timestamp of the block is valid. This means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).
- If any of these checks fails, the block will be rejected. A list of errors for which the block can be rejected is presented here:
  - The timestamp is older than the parent
  - There are too many uncles
  - There is a duplicate uncle
  - The uncle is an ancestor
  - The uncle's parent is not an ancestor
  - There is non-positive difficulty
  - There is an invalid mix digest
  - There is an invalid PoW

Block finalization

Block finalization is a process that is run by miners to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail:

1. **Ommers validation**. In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.
2. **Transaction validation**. In the case of mining, determine transactions. This process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction, in other words, the cumulative gas used by the transactions included in the block.
3. **Reward application**. Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 2 ether. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum.

4. **State and nonce validation**. Verify the state and block nonce. In the case of mining, compute a valid state and block nonce.

After the high-level view of the block validation mechanism, we now look into how a block is received and processed by a node. We will also see how it is updated in the local blockchain:

1. When an Ethereum full node receives a newly mined block, the header and the body of the block are detached from each other. Now remember, in the last chapter, when we introduced the fact that there are three **Merkle Patricia tries** (**MPTs**) in an Ethereum blockchain. The roots of those MPTs or tries are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. We will now learn how these tries are used to validate the blocks.
2. A new MPT is constructed that comprises all transactions from the block.
3. All transactions from this new MPT are executed one by one in a sequence. This execution occurs locally on the node within the **Ethereum Virtual Machine** (**EVM**). As a result of this execution, new transaction receipts are generated that are organized in a new receipts MPT. Also, the global state is modified accordingly, which updates the state MPT (trie).
4. The root nodes of each respective trie, in other words, the state root, transaction root, and receipts root are compared with the header of the block that was split in the first step. If both the roots of the newly constructed tries and the trie roots that already exist in the header are equal, then the block is verified and valid.
5. Once the block is validated, new transaction, receipt, and state tries are written into the local blockchain database.

Block difficulty mechanism

- Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's **Homestead** release is as follows:

  *block_diff = parent_diff + parent_diff // 2048 **

  max(1 - (block_timestamp - parent_timestamp) // 10, -99) +

  *int(2**((block.number // 100000) - 2))*

- The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up by *parent_diff // 2048 * 1*. If the time difference is between 10 and 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficulty level decreases. This decrease is proportional to the time difference from *parent_diff // 2048 * -1* to a maximum decrease of *parent_diff // 2048 * -99*.

- In addition to timestamp difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to **Proof of Stake** (**PoS**), since mining on the PoW chain will eventually become prohibitively difficult.
- According to the original estimates based on the algorithm, the block generation time would have become significantly higher during the second half of 2017, and in 2021, it would become so high that it would be virtually impossible to mine on the PoW chain, even for dedicated mining centers. This way, miners will have no choice but to switch to the PoS scheme proposed by Ethereum, called **Casper**.
- This ice age proposal has been postponed with the release of **Byzantium**. Instead, the mining reward has been reduced from 5 ETH to 3 ETH, in preparation for PoS implementation in Serenity.
- In the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for difficulty calculation. This new formula is shown here:

adj_factor = max((*2 if len*(*parent.uncles*) *else 1*) - ((*timestamp - parent.timestamp*) // *9*)*, -99*)

Blocks are composed of transactions and that there are various operations associated with transaction creation, validation, and finalization. Each of these operations on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas.

GAS

Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator.

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block.

Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund.

Transaction costs can be estimated using the following formula:

*Total cost = gasUsed * gasPrice*

Here, *gasUsed* is the total gas that is supposed to be used by the transaction during the execution, and *gasPrice* is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in ETH. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally.

Each operation costs some gas; a high-level fee schedule of a few operations is shown as an example here:

Operation name	**Gas cost**
Stop	0
SHA3	30
SLOAD	800
Transaction	21000
Contract creation	32000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA-3 operation can be calculated as follows:

- SHA-3 costs 30 gas.
- Assume that the current gas price is 25 GWei, and convert it into ETH, which is 0.000000025 ETH. After multiplying both, 0.000000025 * 30, we get 0.00000075 ETH.
- In total, 0.00000075 ETH is the total gas that will be charged.

Fee schedule

Gas is charged in three scenarios as a prerequisite to the execution of an operation:

- The computation of an operation
- For contract creation or message calls
- An increase in the use of memory