9
Ethereum Architecture

This chapter is an introduction to the Ethereum blockchain, its architecture, and design. We will introduce the fundamentals and various theoretical concepts behind Ethereum. A discussion on the different components, protocols, and algorithms relevant to the Ethereum blockchain is also presented.

We cover different elements of Ethereum such as transactions, accounts, the world state, the **Ethereum Virtual Machine (EVM)**, the blockchain, the blockchain network, wallets, software clients, and supporting ecosystem protocols, with the goal of understanding the technical foundations on which Ethereum is built. The main topics we will explore in this chapter are as follows:

- Introducing Ethereum
- Cryptocurrency
- Keys and addresses
- Accounts
- Transactions and messages
- The EVM
- Blocks and blockchain
- Nodes and miners
- Networks
- Precompiled smart contracts
- Wallets and client software
- Supporting protocols

Let's begin with a brief overview of the foundation, architecture, and use of the Ethereum blockchain.

Introducing Ethereum

Vitalik Buterin conceptualized Ethereum in November 2013. The core idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and **Decentralized Applications (DApps)**.

This concept is in contrast to Bitcoin, where the scripting language is limited and only allows necessary operations. It has gone through some key developments since its creation:

- The first version of Ethereum, called Olympic, was released in May 2015.
- Two months later, a version of Ethereum called Frontier was released in July.
- Another version named Homestead with various improvements was released in March 2016.
- The latest Ethereum release is called Arrow Glacier, which delays the difficulty bomb, a difficulty adjustment mechanism that eventually forces all miners to stop mining on Ethereum 1 and move to Ethereum 2.

A timeline of all major milestones and events is available here: https://ethereum.org/en/history/

A list of all releases as announced is maintained at https://github.com/ethereum/go-ethereum/releases

The formal specification of Ethereum has been described in the *yellow paper*, which can be used to develop Ethereum client implementations.

The Ethereum yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf) was written by *Dr. Gavin Wood*, the founder of Ethereum and Parity (http://gavwood.com), and serves as a formal specification for the Ethereum protocol. Anyone can implement an Ethereum client by following the protocol specifications defined in the paper.

While this paper can be somewhat challenging to read, especially for those who do not have a background in algebra or mathematics and are not familiar with mathematical notation, it contains a complete formal specification for Ethereum. This specification can be used to implement a fully compliant Ethereum client. Therefore, it is necessary to understand this paper at a high level.

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 the 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, which allows interaction with the Geth client via the **Remote Procedure Call** (**RPC**) interface.

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

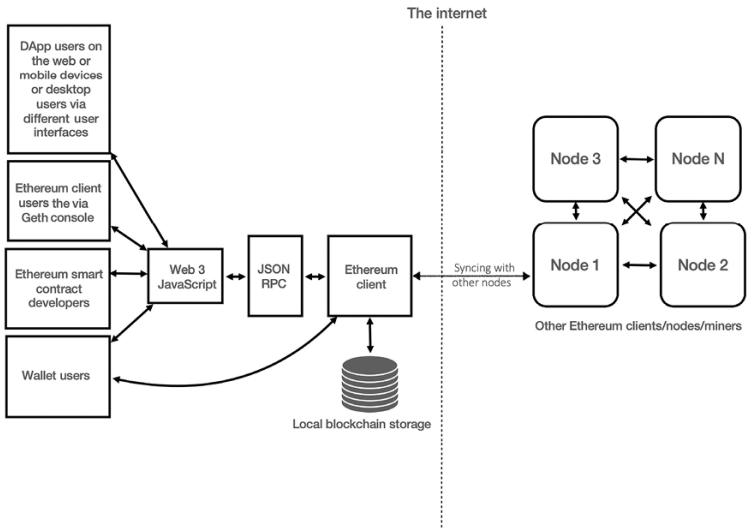

Figure 9.1: Ethereum's high-level ecosystem

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

- Ether cryptocurrency/tokens
- Keys and addresses
- Accounts
- Transactions and messages
- The EVM
- The Ethereum blockchain
- Nodes and miners
- The Ethereum network
- Smart contracts and native contracts
- Wallets and client software

In the following sections, we will discuss each of these one by one.

Cryptocurrency

As an incentive to the miners, Ethereum rewards its own native currency, called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization** (**DAO**) hack described in *Chapter 8*, *Smart Contracts*, a hard fork was proposed to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its own following with a dedicated community that is further developing ETC.

This chapter is focused on ETH, which is currently the most active and official Ethereum blockchain.

Ether is minted by miners as a currency reward for the computational effort they spend to secure the network by verifying transactions and blocks. Ether is used within the Ethereum blockchain to pay for the execution of contracts on the EVM. Ether is used to purchase gas as *crypto fuel*, which is required to perform computation on the Ethereum blockchain.

> The denomination table is available at: https://ethdocs.org/en/latest/ether.html#denominations

---

Next, let's have a look at keys and addresses.

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 the 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) under the rules defined by the elliptic curve `secp256k1` specification (in the range [`1` , `secp256k1n − 1`]).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function. We

will discuss this in the *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

An description of how keys and addresses look in Ethereum is as follows:

- A private key is basically a valid nonzero 256-bit integer randomly picked up from the range of the elliptic curve's field size (order). Ethereum client software makes use of the operating system's random number generator to generate this number.
  Once this integer is picked up, it is represented in hex format (64 hexadecimal characters) as shown here:
  `b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4`.

- A public key is an *x*, *y* coordinate, a point on an elliptic curve that satisfies the elliptic curve equation. The public key is derived from the private key. Once we have the randomly chosen private key, it is multiplied by a preset point on the elliptic curve called the generator point, which produces another point somewhere on the curve. This other point is the public key and is represented in Ethereum as a hexadecimal string of 130 characters:
  `3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260`.
- An address is a unique identifier on the Ethereum blockchain, which is derived from the public key by hashing it through the Keccak-256 hash function and keeping the last 20 bytes of the hash produced. An example is shown here:
  `0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32`.

> Note that the preceding private key is shown here only as an example and should not be reused.

The public and private key pair generated by the new account creation process is stored in key files located locally on the disk. These key files are stored in the `keystore` directory present in the relevant path according to the OS in use.

On the Linux OS, its location is as follows: `~/.ethereum/keystore`

The content of a `keystore` JSON file is shown here as an example. Can you correlate some of the names with what we have already learned in *Chapter 3*, *Symmetric Cryptography*, and *Chapter 4*, *Asymmetric Cryptography*? In the following example, it has been formatted for better visibility:

```
{
"address":"ba94fb1f306e4d53587fcdcd7eab8109a2e183c4",
"crypto":
{
    "cipher":"aes-128-ctr",
    "ciphertext":
        "b2ab4f94f5f44ce98e61d99641cd28eb00fd794129be25beb8a5fae89ef93241",
    "cipherparams":
        {"iv":"a0fdfe0a6d314a62ba6a370f438faa57"},
```

```
    "kdf":"scrypt",
    "kdfparams":
        {"dklen":32,"n":262144,"p":1,"r":8,
"salt":"be3e99203c24ffcb71a6be2823fc7a211c8cc10d66bc6b448fef420fa0669068"},
"mac":"1b0a42d4bf7a8e96d308179e9714718e902727ead7041b97a646ef1c9d6f9ad7"},
"id":"7e0772e0-965e-4a05-ad93-fc5d11245ba3",
"version":3
}
```

The private key is stored in an encrypted format in the `keystore` file. It is generated when a new account is created using the password and private key. The `keystore` file is also referred to as a UTC file as the naming format of it starts with UTC with the date timestamp therein. A sample name of a `keystore` file is shown here:

```
UTC--2022-01-18T17-46-46.604174215Z--ba94fb1f306e4d53587fcdcd7eab8109a2e183c4
```

On Unix-type operating systems, the `keystore` file is usually located under the user's home directory. For example, on a macOS, the `keystore` file is stored at the following location: `~/Library/Ethereum/keystore/`. For other operating systems, refer to the installation instructions available at **https://geth.ethereum.org/docs/getting-started/installing-geth**.

> It is imperative to keep the associated passwords created at the time of creating the accounts and when the key files are produced safe.

As the `keystore` file is present on the disk, it is very important to keep it safe. It is recommended that it is backed up as well. If the `keystore` files are lost, overwritten, or somehow corrupted, there is no way to recover them. This means that any ether associated with the private key will be irrecoverable, too.

Now we have described the elements of the password `keystore` file and what they represent. The key purpose of this file is to store the configuration, which, when provided with the account password, generates a decrypted account private key. This private key is then used to sign transactions:

- `Address` : This is the public address of the account that is used to identify the sender or receiver.
- `Crypto` : This field contains the cryptography parameters.
- `Cipher` : This is the cipher used to encrypt the private key. In the following diagram, **AES-128-CTR** indicates the Advanced Encryption Standard, 128-bit, in counter mode. Remember that we covered this in *Chapter 3*, *Symmetric Cryptography*.
- `Ciphertext` : This is the encrypted private key.
- `Cipherparams` : This represents the parameters required for the encryption algorithm, **AES-128-CTR**.
- `IV` : This is the 128-bit initialization vector for the encryption algorithm.
- `KDF` : This is the key derivation function. It is `scrypt` in this case.
- `KDFParams` : These are the parameters for the **Key Derivation Function (KDF)**.

- `Dklen` : This is the derived key length. It is 32 in our example.
- `N` : This is the iteration count.
- `P` : This is the parallelization factor; the default value is 1.
- `R` : This is the block size of the underlying hash function. It is set to 8, which is the default setting.
- `Salt` : This is the random value of salt for the **KDF**.
- `Mac` : This is the Keccak-256 hash output obtained following the concatenation of the second leftmost 16 bytes of the derived key together with the ciphertext.
- `ID` : This is a random identification number.
- `Version` : The version number of the file format, currently version 3.

Now we will explore how all these elements work together. This whole process is visualized in the following diagram:

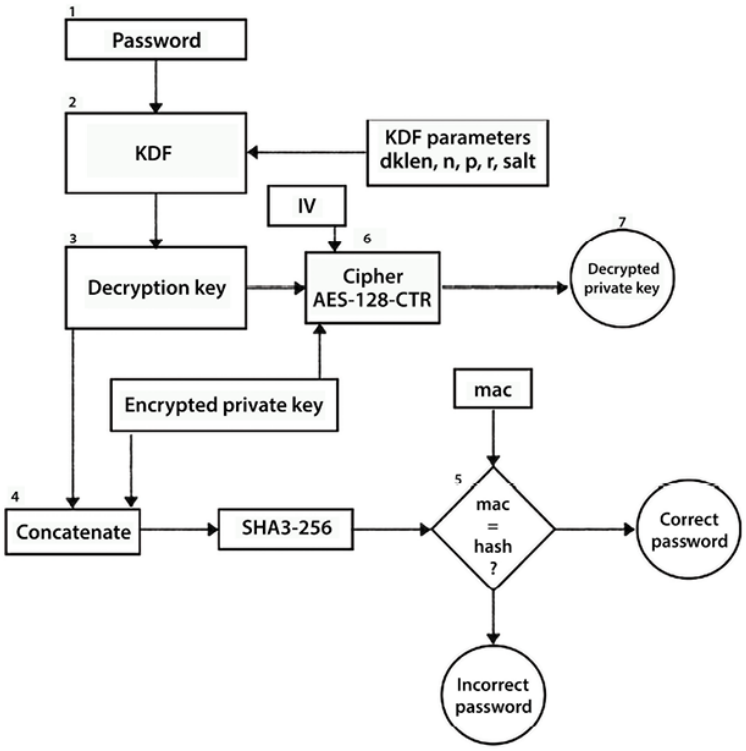

Figure 9.2: Private key decryption process

As shown in the preceding diagram, we can divide the process into different steps:

1. First, the password is fed into the **KDF**.
2. The KDF takes several parameters, namely, **dklen**, **n**, **p**, **r**, and **salt**, and produces a decryption key.
3. The decryption key is concatenated with an **encrypted private key** (ciphertext). The decryption key is also fed into the cipher algorithm, **AES-128-CTR**.
4. The concatenated **decryption key** and ciphertext are hashed using the **SHA3-256** hash function.
5. The **mac** is fed into the checking function where the hash produced from *step 4* is compared with the **mac**. If both match, then the password is valid, otherwise the password is incorrect.
6. The cipher function, which is fed with the **Initialization Vector** (**IV**), encrypted private key (ciphertext), and decryption key, decrypts the encrypted private key.

7. The **decrypted private key** is produced.

This decrypted private key is then used to sign transactions on the Ethereum network.

Another key element in Ethereum is an account, which is required to interact with the blockchain. It either represents a user or a smart contract.

Accounts

An account is one of the main building blocks of the Ethereum blockchain. It is 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. With 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. We will cover gas and relevant concepts shortly in this chapter. 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** (**OOG**) .
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.

Now, as we understand accounts in Ethereum generally, let's examine the two kinds of accounts that exist in Ethereum.

Externally owned account (**EOA**): EOAs are similar to accounts in Bitcoin that are controlled by a private key. The same as in Bitcoin, where an address is controlled by a private key, Ethereum addresses are also controlled by their corresponding private keys. They are defined by three elements: *address*, *balance*, and *nonce*. They have the following properties:

- They have a state.
- They are associated with a human user, hence are also called user accounts.
- EOAs have an ether balance.
- They can send transactions, i.e., can transfer value and can initiate contract code.
- They have no associated code.
- They are controlled by private keys.
- An EOA's public address is derived from its private key.
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

Contract account (CA): CAs are accounts that have code associated with them in addition to being controlled with the private key. They are defined by five elements: *address*, *balance*, *nonce*, *StorageRoot*, and *codeHash*. They have the following properties:

- 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 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. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter, in the *The EVM* section.
- A CA's public address is the combination of the public address of the EOA that created it and a transaction counter (how many transactions the EOA has sent) nonce.
- They have access to storage and can manipulate their storage.
- Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in future Ethereum releases, the distinction between EOAs and CAs may be eliminated.
- A CA can only be created from an EOA or from an already existing CA (contract).
- CAs cannot start transaction messages.
- CAs can initiate a call message, i.e., can call other contracts and contracts' functions.
- CAs can transfer ether.
- 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.

Accounts allow interaction with the blockchain via transactions.

Transactions and messages

A transaction in Ethereum is a digitally signed (using a private key) data packet that contains the instructions that, when completed, either result in a message call or contract creation. Transactions are constructed by an actor external to the Ethereum blockchain or some external software tool. Smart contracts cannot send a transaction.

Initially, in Ethereum, there were simple transactions, message call transactions, and contract creation transactions. Later, as the system evolved, a need was felt to introduce more types and also to make it easier to introduce new features and future transaction types and be able to distinguish different types of transactions. New features such as access lists and EIP-1559 have been introduced while remaining compatible with legacy transactions. Further innovation resulted in the introduction of typed transactions, which were introduced in EIP-2718. They define a new transaction type, which is an envelope for future transaction types.

We will shortly explain what an Ethereum transaction is and consider its different types. But, before that, let's look at an important concept, the **Merkle Patricia Tree** (**MPT**), which is the foundation of integrity in the blockchain.

MPTs

It's essential to understand what an MPT is and how it guarantees blockchain data integrity. An MPT is used in all data structures relevant to the state and transactions, such as the world state, receipts, storage, and the account state. Remember, we discussed the Merkle tree in *Chapter 3*, *Symmetric Cryptography*.

Leaf nodes of a Merkle tree are paired and transactions are concatenated from a block for which the Merkle tree is to be constructed. Once these leaf nodes are hashed together, another set of nodes is created. It is again paired and concatenated and hashed until a single final hash value is obtained, called the Merkle root. In each block's header, there is a field in which this Merkle root is stored and serves as proof for all the transactions included in the block. Due to the collision-resistant property of hash functions, it is proven that changing even a single bit in any transaction will result in changing the Merkle hash value and, consequently, the block header's hash. Because of this, the Merkle root, the block header's hash, is a unique fingerprint for the block and all its transactions. Each new block also points to the preceding block by including its header's hash, and this pattern repeats up to the genesis block, making a hash-linked chain of blocks or simply a blockchain. This hash linking results in transaction linking due to the transaction's Merkle root being present in each block; we can verify this whole data structure for integrity.

The way integrity is checked is quite simple:

1. We need a reference to the latest block (block header) and the genuine genesis block.
2. Each hash field in the hash pointer present in the block header is compared to the previous block header's hash, and this process runs until the genesis block is verified. If the verification fails at any point, this means that there is some modification in the transactions or block header that resulted in a changed Merkle root. This process verifies the integrity of the chain.
3. Moreover, during the block validation checks, it is checked if the Merkle root in the block header matches the hash of all the transactions in the block.

To verify if a given transaction is part of the blockchain or not, the Merkle root needs to be recomputed and verified. This process is called *Merkle*

path authentication or simply *Merkle proof*. Here's how it works:

If a transaction *t* is included in block *b*, then as long as the hash function is collision-resistant and the Merkle root value is authentic, *t* can be verified as follows:

1. First, calculate the hash of the element to be checked if it exists in the block. Let's say it is transaction *t*.
2. Next, get the Merkle root hash stored in the block header.
3. Calculate a candidate root hash value by using the transaction *t* for which the proof of inclusion is required and taking the off-path (i.e., off the authentication path) values. Note that the authentication path includes the siblings of nodes from transaction *t* to the root node (the Merkle root).
4. If the candidate root hash value matches the Merkle root hash stored in the block *b* header, it confirms that the transaction *t* is part of block *b*.

We now understand how Merkle proof of inclusion is performed and blockchain integrity is verified using Merkle trees; we can apply the same logic to MPTS. We can use the MPT to store, update, insert, delete, and retrieve key-value pairs. Each node in the tree can hold a part of a key, a value, and pointers to other nodes. An MPT is obtained by replacing the pointers with hash pointers, which means that the nodes are referenced by the hash of their contents instead of simple address pointers. The root of the MPT now serves the same purpose as the root of the Merkle tree we just saw, meaning that it represents the complete contents of the entire tree. This implies that the hash of the MPT's root node can be used to authenticate the entire key-value dataset of which the MPT is composed.

Ethereum goes a step further and customizes this MPT. The data of each node in the tree is stored in a database, and a hash value is used to look up this content. In Ethereum, there are three Merkle roots in the block header:

- **TransactionsRoot**: This is the root hash of the MPT of the transactions tree included in the block. Each block has its own separate transaction tree.
- **StateRoot:** This is the root hash of the MPT of the world state, i.e., persistent state.
- **ReceiptsRoot:** This is the root hash of the MPT of transaction receipts in the block. Each block has its own receipts tree.

Let's first now focus on transactions, after which we will discuss world state, account state, and transaction receipts, where the concept of *StateRoot* and *ReceiptsRoot* will become clear.

Transaction components

Transactions in Ethereum are composed of some standard fields, which are described as follows:

- Type: EIP-2718 transaction type 0 for legacy, 1 for EIP-2930, and 2 for EIP-1559. EIP-2718 defines a new generalized transaction envelope that allows the creation of different types of transactions. It is a simple format where the transaction type field is concatenated with a transaction payload, which can be of different types, depending on the

transaction type. There can be 128 possible transaction types, as the transaction type is a number defined as ranging from 0 to 0x7f. Defining generalized envelopes like this allows for backward compatibility and less complexity while being able to create different types of new transactions. The transaction payload is a byte array defined by the transaction type.

- There are also three standard legacy transaction types based on the output they produce:
  - **Simple transactions**: This is the standard transaction used for payments. It transfers ether (funds) between EOAs.
  - **Message call transactions**: This transaction is used to execute smart contracts, i.e., used to invoke methods in the deployed smart contracts.
  - **Contract creation transactions**: These transactions result in the creation of a new contract account, i.e., a smart contract—an account with the associated code.

We'll elaborate more on these core transaction types later, under the *Transaction types* section.

- **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. A nonce value can only be used once. This is used for replay protection on the network.
- **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. The concept of gas and gas limits will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the fee amount, in ether, that a user (for example, the sender of the transaction) is willing to pay for computation.
- **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; in the case of a CA, this represents the balance that the contract will hold.
- **Signature**: Transactions are signed using recoverable ECDSA signatures. 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 be determined from these values. The signature is based on the ECDSA scheme and makes use of the `secp256k1` curve. The theory of **Elliptic Curve Cryptography** (**ECC**) was discussed in *Chapter 4*, *Asymmetric Cryptography*. In this section, ECDSA will be presented in the context of its usage in Ethereum.

V is a single-byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. *V* is used in the ECDSA recovery contract as a recovery value. This value is used to recover (derive) the public key from the private key. In `secp256k1`, the recovery value is expected to be either 0 or 1. In Ethereum, this is offset by 27.

R is derived from a calculated point on the curve. First, a random number is picked, which is multiplied by the generator of the curve to calculate a point on the curve. The *x*-coordinate part of this point is *R*. *R* is encoded as a 32-byte sequence. *R* must be greater than 0 and less than the `secp256k1n` limit (`115792089237316195423570985008687907852837564279074904382605163141518161494337`).

S is calculated by multiplying *R* with the private key and adding it to the hash of the message to be signed, and then finally dividing it by the random number chosen to calculate *R*. *S* is also a 32-byte sequence. *R* and *S* together represent the signature.

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 internal code execution.
- **Data**: If the transaction is a message call (i.e., calling methods in a deployed smart contract), then the **Data** field is used instead of init. It contains the input data of the message call. In other words, it contains data such as the name of the function in the smart contract to be called and relevant parameters. It is unlimited in size and is organized as a byte array.

The **Type 1** transaction (EIP-2930) introduced the following fields:

- **ChainID:** The transaction is only valid on the blockchain network with this specific ChainID.
- **AccessList**: This field contains a list of access entries. Each access list is a tuple of account addresses and a list of storage keys that the transaction intends to access.
- **yParity:** This is signature Y parity, instead of V in a legacy transaction. The parity of the y-value of a `secp256k1` signature is 0 for even and 1 for odd.

Legacy transactions do not have an AccessList field. In addition, ChainID and yParity are combined into a single field, W, described below, in legacy transactions.

- **W:** This is a scalar value that encodes Y parity and ChainID. This is based on EIP-155.

EIP-1559 introduces a new transaction type. We call them **Type 2** transactions. Most of the fields remain the same as the legacy structure, however, some new fields are added, which are described below:

- **ChainID**: This field is now part of the transaction payload. It used to be encoded in the signature V value due to EIP-155.
- **maxPriorityFeePerGas** and **maxFeePerGas**: Two new fields introduced as a replacement for the legacy gasPrice field.
- **Gas Limit**, **destination**, **amount**, and **data** remain the same as legacy transactions.
- **AccessList:** This is the same as EIP-2930, where the access list can be defined.
- **signatureYparity:** The signature V value from legacy transactions is replaced with signatureYparity, which can be either a 0 or 1, depending on which *y*-coordinate on the elliptic curve is used. In legacy transactions, it used to be 27, 28 or 35, 36 in EIP-155 transactions for a signature recovery identifier.
- **signatureR** and **signatureS** remain the same as legacy and Type 1 transactions. Recall from *Chapter 4*, *Asymmetric Cryptography*, that these are simply two points on the elliptic curve.

When a transaction is successfully executed, a record in the transaction tree is created containing all fields related to the transaction. This structure is visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a **transaction trie** (a modified 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:

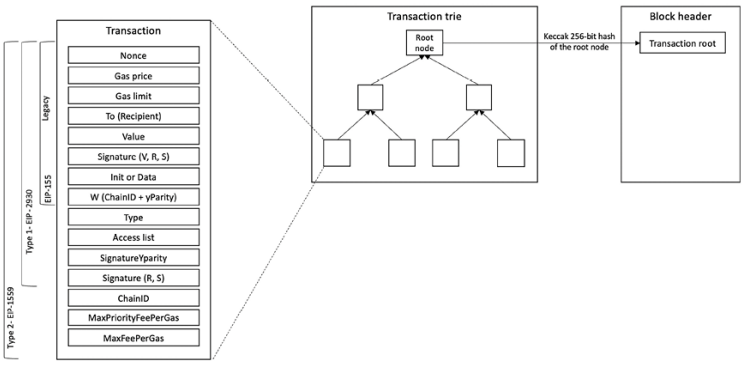

Figure 9.3: The relationship between the transaction, transaction trie, and block header

The question that arises here is how all these accounts, transactions, and related messages flow through the Ethereum network and how are they stored. So, before we move on to the different types of transactions and messages in Ethereum, let's explore how Ethereum data is encoded for storage and transmission. For this purpose, a new encoding scheme called **Recursive Length Prefix** (**RLP**) was developed, which we will cover here in detail.

Recursive Length Prefix

To define RLP, we first need to understand the concept of serialization. Serialization is simply a mechanism commonly used in computing to encode data structures into a format (a sequence of bytes) that is suitable for storage and/or transmission over the communication links in a network. Once a receiver receives the serialized data, it de-serializes it to obtain the original data structure. Serialization and deserialization are also referred to as marshaling and un-marshaling, respectively. Some commonly used serialization formats include XML, JSON, YAML, protocol buffers, and XDR. There are two types of serialization formats, namely *text* and *binary*. In a blockchain, there is a need to serialize and deserialize different types of data such as blocks, accounts, and transactions to support transmission over the network and storage on clients.

> Why do we need a new encoding scheme when there are so many different serialization formats already available? The answer to this question is that RLP is a deterministic scheme, whereas other schemes may produce different results for the same input, which is absolutely unacceptable on a blockchain. Even a small change will lead to a totally different hash and will result in data integrity problems that will render the entire blockchain useless.

RLP is an encoding scheme developed by Ethereum developers. It is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in an MPT on storage media. It is a deterministic and consistent binary encoding scheme used to serialize objects on the Ethereum blockchain such as account state, transactions, messages, and blocks. It operates on strings and lists to produce raw bytes that are suitable for storage and transmission. RLP is a minimalistic and simple-to-implement serialization format that does not define any data types and simply stores structures as nested arrays. In other words, RLP does not encode specific data types; instead, its primary purpose is to encode structures.

> More information on RLP is available on the Ethereum wiki, at `https://eth.wiki/en/fundamentals/rlp`.

Now, having defined RLP, we can delve deeper into transactions and other relevant elements of the Ethereum blockchain. Each operation on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas. We will now introduce this mechanism in detail.

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.

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

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 higher the fee, the higher the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if a 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 OOG 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 a 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. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or fewer operations than intended initially and can result in consuming more or less gas. If the execution runs OOG, everything is immediately rolled back; otherwise, if the execution is successful and some gas remains, then it is returned to the transaction originator.

A website that keeps track of the latest gas price and provides other valuable statistics and calculators is available at `https://ethgasstation.info`

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.

Transaction fees have been very high and at times the fee exceeded the value of the asset. On Ethereum, the high transaction fee is due to scalability constraints and how the economics of Ethereum works, instead of a design limitation. There are two aspects that one can think of when thinking about gas fees: opcodes and gas prices.

One idea that comes to mind is to reduce the EVM opcode gas cost, thereby lowering the fees. However, this would not work—the opcode gas cost estimates the time required to process an opcode, roughly calculated in milliseconds + some other constraints. In contrast, gas price is a function of Ethereum's fee market and supply and demand. This market exists because of the limited block size (maximum block size—12.5M gas—limit on the total gas that can be consumed by transaction in a block) where everyone is trying to get their transaction processed first. Therefore, it is a first-price auction market mechanism. All bidders bid simultaneously and the highest bidder wins. If the opcode cost is reduced, then due to supply and demand, the network will always have this high price fee due to this fee market mechanism.

Users are also willing to pay more to get their transaction processed first and miners can also ignore the low fee transaction and can order them as they like. So, the network gas price will remain high because, fundamentally, the Ethereum transaction fee market is a highest-bidder-wins market. Unless that is changed, only reducing the opcode cost will not result in any improvement. Opcodes' gas cost is there for a different reason. In comparison, gas price is a result of supply and demand and limited block size.

Other issues are DoS attacks and slower computers. Slower computers may not be able to process opcodes quick enough. For example, imagine a standard transfer instruction is 21,999 gas units. This is because it involved some ECC cooperation, which is estimated to take some time. Imagine the estimated time is hypothetically 21 milliseconds, because the gas cost is an estimate of the time it takes to process an opcode instruction. Now, if we reduce the gas cost, a faster computer will be able to do it quicker and within, let's say, 10, 000 gas untis. However, a slower computer will not be able to complete the whole operation in time.

Therefore, opcodes' cost cannot be just halved or reduced below a certain level. We can think of it as faster block times and bigger block sizes resulting in lower transaction fees. Some new layer 1 chains have introduced this, such as Solana.

Transaction types

We will continue by exploring different types of transactions on the Ethereum blockchain. We discussed this briefly before, under the section transaction components, however, now we'll dig deeper and learn about different elements of three core types of transactions in Ethereum.

Simple transactions

This is the standard value transfer transaction that Ethereum supports. It is used to transfer funds (ether) between accounts.

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)

The initialization EVM code from the init field in the transaction contains the EVM bytecode compiled by tools such as the Solidity compiler. Once the contract is deployed, it is added as a new item in the world state trie. Its bytecode is stored in the codeHash field in the account state. Moreover, a new storage trie is created and its root is saved in the `StorageRoot` field in the account state.

Addresses generated as a result of a contract creation transaction are 160 bits in length. Precisely as defined in the yellow paper, they are the right-most 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. The storage is also set to empty. The codeHash is a Keccak 256-bit hash of the empty string.

The new account is initialized when the EVM code (the initialization EVM code, mentioned earlier) is executed. In the case of any exception during code execution, such as not having enough gas (running **OOG**), the state does not change. If the execution is successful, then the account is created after the payment of the appropriate gas costs.

Since **Ethereum Homestead**, the result of a contract creation transaction is either a new contract with its balance or no new contract is created with no transfer of value. This contrasts with versions prior to Homestead, where the contract would be created regardless of the contract code deployment being successful or not due to an OOG exception.

Message call transactions

The state is altered by transactions. These transactions are created by external factors (users) and are signed and then broadcast to the Ethereum network. After a smart contract is deployed successfully, its functions are called by transactions called message calls. We can simply think of these as *contract execution* transactions. A message call transaction requires several parameters for execution, which are listed as follows:

- The sender
- The transaction originator

- The recipient (contract address)
- 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

Message calls result in a state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by EVM code, the output produced by the transaction execution is used. As defined in the yellow paper, a message call is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the EVM will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object (external actor), then the call passes back any data returned from the EVM operation.

Messages are passed between accounts using message calls. A description of messages and message calls is presented next.

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
- The 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

Messages are generated when the `CALL` or `DELEGATECALL` opcodes are executed by the contract running in the EVM.

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. It is akin to a dry run or a simulated run. Calls also do not allow ether transfer to CAs. Calls are executed locally on a node EVM and do not result in

any state change because they are never mined. 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 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 (i.e., 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 have a sufficient balance to cover the execution cost.

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. Events will be covered with practical examples in *Chapter 12*, *Web3 Development Using Ethereum*.
- **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 that are involved in 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 and storage in the Ethereum blockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This definition is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood.

The core idea is that in the Ethereum blockchain, a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state. In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition:

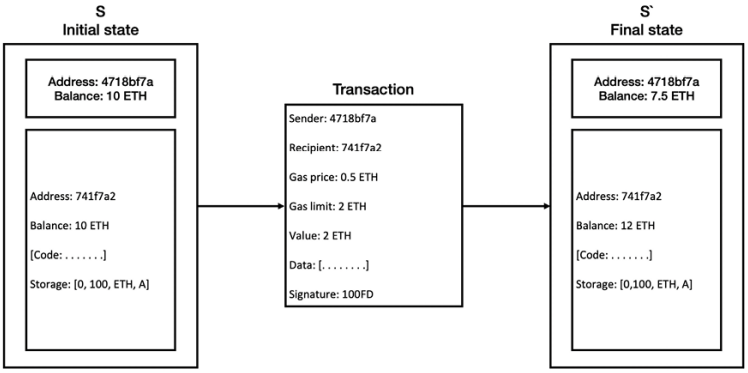

Figure 9.4: Ethereum state transition function

In the preceding example, a transfer of two ether from address **4718bf7a** to address **741f7a2** is initiated. The initial state represents the state before the transaction execution, and the final state is what the morphed state looks like. Mining plays a central role in state transition, and we will elaborate on the mining process in detail in later sections. The state is stored on the Ethereum network as the *world state*. This is the global state of the Ethereum blockchain.

The state needs to be stored permanently in the blockchain. For this purpose, the world state, the account state, transactions, and transaction receipts are stored.

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. State root is the hash of the root of the MPT of the world state, i.e., persistent state. Here an Ethereum address (from EOA—[address, balance, nonce]) is used as a key to find the leaf node (lookup for the value), the account state which is composed of four elements [the nonce, balance, StorageRoot, codeHash]. StorageRoot is the root of another trie called the storage trie, where the contract code is stored. We describe the account state next.

The account state

The account state consists of four fields: nonce, balance, StorageRoot, and codeHash, and is described in detail here:

- **Nonce**: This is a scalar value that is incremented every time a transaction is sent from the address. In the case of contract accounts, it represents the number of contracts created by the account.
- **Balance**: This value represents the number of Weis held by the given address.
- **Storage root**: This field represents the root node of the MPT that encodes the storage contents of the account. It is a mapping encoded in the trie, from the Keccak 256-bit hash of the 256-bit integer keys to the

RLP-encoded 256-bit integer values. (i.e., the hash of the integer keys to the RLP-encoded integer values).
- **CodeHash**: 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 world state and its relationship with the accounts trie, accounts, and block header are visualized in the following diagram. It shows the **account state**, or data structure, which contains a **storage root** hash derived from the **root node** of the **account storage trie** shown on the left. The account data structure is then used in the **world state trie**, which is a mapping between addresses and account states.

The accounts trie is an MPT used to encode the storage contents of an account. The contents are stored as a mapping between Keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Finally, the **root node** of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the **block header** data structure, which is shown on the right-hand side of the diagram as the **state root** hash:

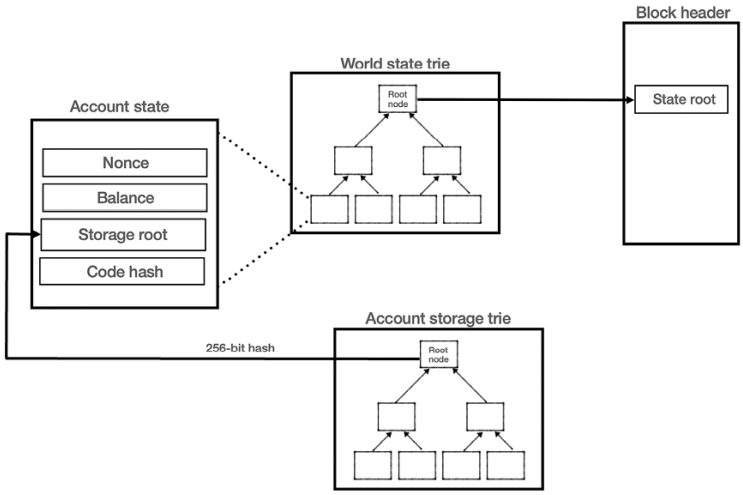

Figure 9.5: The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationships

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. They are 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 five elements, as follows:

- **Type of the transaction**: This item is the type of transaction based on EIP-2718. `0` represents a legacy transaction, `1` represents EIP-2930, and `2` represents EIP-1559.
- **Status code**: Since the Byzantium release, an additional field returning the success (`1`) or failure (`0`) of the transaction is also available.

- **Cumulative 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 a non-negative integer.
- **Series of log entries**: This field has a 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.
- **Bloom filter**: A bloom filter is created from the information contained in the set of logs discussed earlier. 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.

More information about this change is available at https://github.com/ethereum/EIPs/pull/658.

This transaction receipt and its relationship with the block header is visualized in the following diagram:

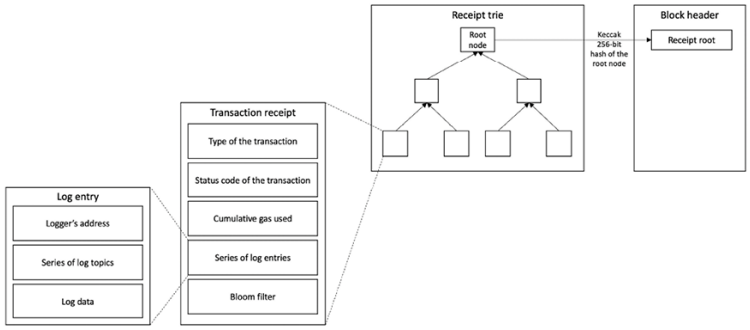

Figure 9.6: Transaction receipts and logs bloom

As a result of the transaction execution process, the state morphs from an initial state to a target state. This state needs to be stored and made available globally in the blockchain.

State is updated as a result of EVM execution, which is the core execution machine for the Ethereum blockchain and we'll discuss it next.

Ethereum virtual machine

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256 bits. The stack size is limited to 1,024 elements and is based on the **Last In, First Out** (**LIFO**) queue. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements.

The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external re-

sources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on the Ethereum blockchain.

As discussed earlier, the EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

There are six types of locations available to smart contracts and the EVM for reading and writing information:

- **Memory**: The first location is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. `write` operations to the memory can be of 8 or 256 bits, whereas `read` operations are limited to 256-bit words. Memory is unlimited but is constrained by gas fee requirements. It is available only for the duration of the transaction.
- **Storage**: The other location is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack**: EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1,024 elements and supports a word size of 256 bits. EVM opcodes `POP` data from and `PUSH` data to the stack. It is a temporary storage location, and it is empty once the transaction execution completes.
- **Call data**: This is the data field of a transaction. When the transaction is being executed, this acts as read-only memory that contains parameters to the message call.
- **Code**: This location contains the code currently being executed and is also used for static data storage.
- **Logs**: This is the write-only log and event output space.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory** (**virtual ROM**), which is accessible using the `CODECOPY` instruction. The `CODECOPY` instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM. There is another instruction called `EXTCODECOPY`, which can be used to copy program code from another contract to the memory.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into the main memory using the `CODECOPY` instruction. The main memory is then read by the EVM by referring to the program counter and executes instructions step by step.

The program counter and EVM stack are updated accordingly with each instruction execution:

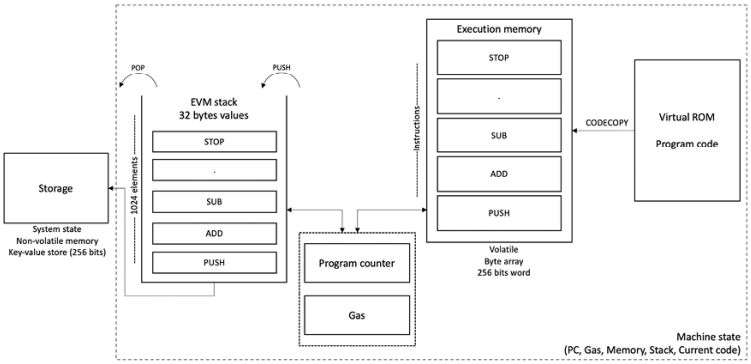

Figure 9.7: EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained and incremented with instructions being read from the main memory, with indexing the next EVM bytecode instruction to execute. There is also a gas register that keeps the count of available gas. The main memory gets the program code from the virtual ROM/storage via the `CODECOPY` instruction.

> The success and wide adoption of Ethereum sparked an interest in developing more EVM chains. There are various EVM chains available in the blockchain ecosystem now. A list of EVM networks with necessary details is available here: **https://chainlist.org**

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly (ewasm)**—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly (Wasm)** was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed. More information and the GitHub repository for ewasm is available at **https://github.com/ewasm**.

Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development. More information on this language can be found at **https://solidity.readthedocs.io/en/latest/yul.html**.

Execution environment

There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent, for example, a transaction. These are listed as follows:

- The system state.
- The remaining gas for execution.
- Accrued substate
- The address of the account that owns the executing code.
- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).

- The gas price of the transaction that initiated the execution.
- Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
- The address of the account that initiated the code execution or transaction sender. This is the address of the sender if the code execution is initiated by a transaction; otherwise, it is the address of the account.
- The value of the transaction. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed, presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- Depth—the number of message calls or contract creation transactions (`CALL` , `CREATE` or `CREATE2`) currently in execution.
- Permission to make modifications to the state.

The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is maintained internally and updated after each execution cycle of the EVM. An iterator function runs in the EVM, which outputs the results of a single cycle of the state machine. The iterator function performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

- It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
- It adds/removes (`PUSH/POP`) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/opcodes. It increments the **Program Counter** (**PC**).

The EVM is also able to halt in normal conditions if `STOP` , `SUICIDE` , or `RETURN` opcodes are encountered during the execution cycle.

Machine state can be viewed as a tuple (g, pc, m, i, s) that consists of the following elements:

- g: Available gas
- pc: The PC, which is a positive integer of up to 256
- m: The contents of the memory (a series of zeroes of size 2^{256})
- i: The active number of words in memory (counting continuously from position 0)
- s: The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions occur:

- Not enough gas remaining for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

With this, we complete our discussion on the EVM.

Blockchain is composed of blocks and blocks contain transactions. Next, we'll discuss what a block is, its structure, and how blocks are validated in the Ethereum blockchain.

Blocks and blockchain

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

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

An uncle block is a block that is the child of a parent but does not have a child block. Ommers or uncles are valid but stale blocks that are not part of the main chain but contribute to the security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

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 uncle) 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 has been 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.
- **Transaction 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. i.e., the block number.
- **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**: The 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.
- **BaseFeePerGas**: A new field introduced after the London upgrade—EIP-1559—to record the protocol calculated fee required for a transaction to be included in the block.

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

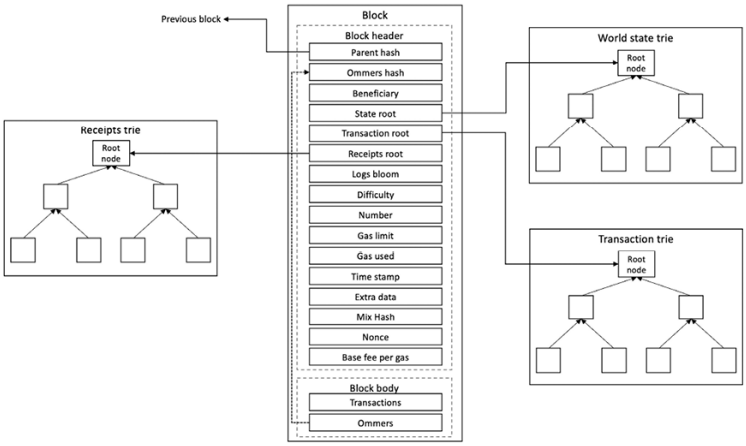

Figure 9.8: A detailed diagram of the block structure with a block header and relationship with tries

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is included in the block.

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. The key difference is that it is a hardcoded block in the Ethereum software client, whereas the rest of the blocks after it are mined. It contains 15 items that can be viewed on the Etherscan block explorer: `https://etherscan.io/block/0`.

Block validation, finalization, and processing

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. 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 – more than two
  - 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 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:

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 ethers. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum. Later, in the Constantinople release (`https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/`), it was reduced further to 2 ether. A block can have a maximum of two uncles.
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'll 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 we introduced that there are three MPTs in an Ethereum blockchain whose roots are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. The account contains another root for the storage trie where the contract data is stored, called StorageRoot. 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 **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.
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 decreases if the block time between two blocks increases. 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))*

Note that // is the integer division operator.

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 the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for the difficulty calculation. This new formula is shown here:

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

Soon after the **Istanbul** upgrade, the difficulty bomb was delayed once again, under the **Muir Glacier** network upgrade with a hard fork, `https://eips.ethereum.org/EIPS/eip-2384`, for roughly another 611 days. The change was activated at block number 9,200,000 on January 2, 2020. It has been delayed yet again under Arrow Glacier: `https://eips.ethereum.org/EIPS/eip-4345`. We will consider the difficulty time bomb in more detail in *Chapter 10, Ethereum in Practice*.

Nodes and miners

The Ethereum network contains different nodes. Some nodes act only as wallets, some are light clients, and a few are full clients running the full blockchain. One of the most important types of nodes is mining nodes. We will see what constitutes mining in this section.

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. As a result of the mining operation, currency (ether) is awarded to the nodes that perform mining operations as an incentive for them to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

In this process, the block header is repeatedly hashed until a valid nonce is found, such that once hashed with the block header, it results in a value less than the difficulty target. At a theoretical level, a miner node performs the following functions:

- It listens for the transactions broadcast on the Ethereum network and determines the transactions to be processed.
- It determines stale ommer blocks and includes them in the blockchain.
- It updates the account balance with the reward earned from successfully mining the block.
- Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.

Once a block is successfully mined, it is broadcast immediately to the network, claiming success, and will be verified and accepted by the network.

Miners play a vital role in reaching a consensus regarding the canonical state of the blockchain. The consensus mechanism that they contribute to is explained in the next section.

The consensus mechanism

The original method of mining in Ethereum was based on PoW, which is similar to that of Bitcoin. When a block is deemed valid, it must satisfy not only the general consistency requirements, but it must also contain the PoW for a given difficulty. Ethereum's PoW algorithm is called Ethash. The consensus mechanism in Ethereum is based on the **Greedy Heaviest Observed Subtree** (**GHOST**) protocol proposed initially by Zohar and Sompolinsky in December 2013.

> Readers interested in this topic can find more information in the paper at `http://eprint.iacr.org/2013/881.pdf`.

Ethereum uses a simpler version of this protocol, where the chain that has the most computational effort spent on it to build it is identified as the definite version. Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining efforts. The GHOST protocol was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to the creation of stale or orphaned blocks. In GHOST, stale blocks, orphan blocks, uncles, or ommers, are added in calculations to figure out the chain of blocks to be selected as the main chain, hence the term heaviest chain. This also means that this is the chain that has the most computational work done behind it. The following diagram shows the difference between the longest and heaviest chains:

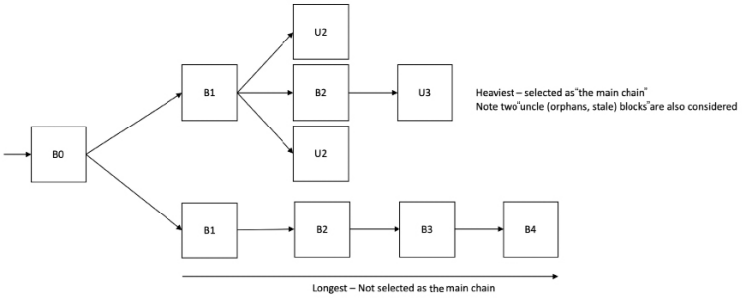

Figure 9.9: The longest versus the heaviest chain

The preceding diagram shows two rules for figuring out which blockchain is the canonical version of the truth. In the case of Bitcoin, shown on the left-hand side in the diagram, the longest chain rule is applied, which means that the active chain (true chain) is the one that has the most amount of PoW done. In the case of Ethereum, the concept is similar from the point of view of the longest chain, but it also includes ommers, the *orphaned* blocks, which means that it also rewards those blocks that were competing with other blocks during mining to be selected and performed significant PoW or were mined exactly at the same time as others but did not make it to the main chain. This makes the chain the *heaviest* instead of the *longest* because it also contains the *orphaned* blocks. This is shown on the right-hand side of the diagram.

Ethash was originally intended to be ASIC-resistant, where finding a nonce requires large amounts of memory. However, now some ASICs are available for *Ethash* too.

An algorithm named **Casper** has been developed that will replace the existing PoW algorithm in Ethereum. This is a security deposit based on the economic protocol where nodes are required to place a security deposit before they can produce blocks. Nodes have been named *bonded validators* in Casper, whereas the act of placing the security deposit is named *bonding*.

> More information about Casper can be found here: `https://github.com/ethereum/research/tree/master/casper4`.

After "the merge", the consensus protocol is replaced with proof-of-stake mechanics, which we will discuss in detail in *Chapter 13*, *The Merge and Beyond*.

As the blockchain progresses (more blocks are added to the blockchain) governed by the consensus mechanism, on occasion, the blockchain can split into two. This phenomenon is called **forking**.

Forks in the blockchain

A fork occurs when a blockchain splits into two. This can be intentional or unintentional. Usually, as a result of a major protocol upgrade, a hard fork is created, while an unintentional fork can be created due to bugs in the software.

It can also be temporary, as discussed previously. In other words, the longest and heaviest chain. This temporary fork occurs when a block is created almost at the same time and the chain splits into two, until it finds the longest or heaviest chain to achieve eventual consistency.

The release of **Homestead** involved major protocol upgrades, which resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum, known as **Frontier**, to the second version. Another version is called **Byzantium**. This was released as a hard fork at block number 4,370,000. The latest upgrade is the **London** hard fork under EIP-1559, activated at block number 12,965,000.

An unintentional fork, which occurred on November 24, 2016, at 14:12:07 UTC, was due to a bug in Ethereum's Geth client journaling mechanism. As a result, a network fork occurred at block number 2,686,351. This bug resulted in Geth failing to prevent empty account deletions in the case of the empty OOG exception. This was not an issue in **Parity** (another popular Ethereum client). This means that from block number 2,686,351, the Ethereum blockchain is split into two, one running with the Parity clients and the other with Geth. This issue was resolved with the release of Geth version 1.5.3. As a result of the DAO hack, the Ethereum blockchain was forked to recover from the attack.

Fundamentally, the Ethereum blockchain is a database that exists on every node in the Ethereum network. Let's now explore what the Ethereum network is.

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: the main net, test nets, and private nets.

Main net

The **main net** is the current live network of Ethereum. Its network ID is 1 and its chainID is also 1. The network and chainIDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at `https://etherscan.io`. This can be used to explore the Ethereum blockchain.

Test nets

There are two main networks 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 test net is called *Sepolia*, which is a PoW test net and contains all the features of the Ethereum main net. *Gorli* is another test net that is based on proof of authority.

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.

> Network IDs and chainIDs are used by Ethereum clients to identify the network. ChainIDs were introduced in EIP-155" as part of the replay protection mechanism. EIP155 is detailed at https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md. In EIP-155, ChainID was encoded with the signature V value. Currently, with EIP-1559, ChainID is a separate field in the transaction payload, identifying the blockchain network. A comprehensive list of Ethereum networks is maintained and available at https://chainlist.org

---

A private net allows the creation of an entirely new blockchain, usually on a local network. This is different from the testnet or main net in the sense that it uses its own genesis block and network ID.

On the main net, the Geth Ethereum client can discover **boot nodes** by default as they are hardcoded in the Geth client, and connects automatically. But on a private network, Geth needs to be configured by specifying appropriate flags and configurations in order for it to discover, or be discovered by, other peers. We will see how this is practically achieved in *Chapter 10*, *Ethereum in Practice*.

Let's get some theoretical aspects covered that are related to discovery and understand what actually happens when we disable node discovery. For this, first, we'll see how a node normally discovers other nodes on an Ethereum network and what protocols are involved.

The Ethereum network consists of four elements or layers: **Discovery**, **RLPx**, **DEVP2P**, and **application-level sub-protocols**:

- The **Discovery** protocol is responsible for discovering other nodes on the network by using a node discovery mechanism based on the Kademlia protocol's routing algorithm. This protocol works by using UDP. UDP is a connectionless and faster communication protocol as compared to TCP, which makes it suitable for time-sensitive applications and DNS lookups.
  In Ethereum, there are two discovery protocols, named DiscV4 and DiscV5. DiscV4 is currently production-ready and implemented in Ethereum, whereas DiscV5 is in development.

  > The specification for DiscV4 can be found here: https://github.com/ethereum/devp2p/blob/master/discv4.md
  >
  > The specification for DiscV5 can be found here: https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md

  ---

  For discovery, a new Ethereum node joining the network makes use of hardcoded bootstrap nodes, which provide an initial entry point into the network, from which further discovery processes then start. This list of bootstrap nodes can be found in the `bootnodes.go` file:

https://github.com/ethereum/go-ethereum/blob/2b0d0ce8b0a02634b90b02bc038523eacd2b220a/params/bootnodes.go#L23

- **RLPx** is a TCP-based transport protocol responsible for enabling secure communication between Ethereum nodes. It achieves this by using an asymmetric encryption mechanism called **Elliptic Curve Integrated Encryption Scheme** (**ECIES**) for handshaking and key exchange.

  > Note that RLPx is named after *RLP*, the serialization protocol introduced earlier. However, it is not related to the serialization protocol RLP, and the name is not an acronym.
  > Handshaking is a term used in computing to refer to a mechanism of exchanging initial information (signals, data, or messages) between different devices on a network to establish a connection for communication as per the protocol in use.

  More information on ECIES and the RLPx specification can be found here: https://github.com/ethereum/devp2p/blob/master/rlpx.md.

- **DEVP2P** (also called the wire protocol) is responsible for negotiating an application session between two nodes that have been discovered and have established a secure channel using RLPx. This is where the `HELLO` message is sent between nodes to provide each other with details of the version of the DEVP2P protocol: the client name, supported application sub-protocols, and the port numbers the nodes are listening on. `PING` and `PONG` messages are used to check the availability of the nodes and `DISCONNECT` is sent if a response from a node is not received.
- Finally, the fourth element of the Ethereum network stack is where different **Ethereum sub-protocols** exist. After discovering and establishing a secure transport channel and negotiating an application session, the nodes exchange messages using so-called "capability protocols" or application sub-protocols. This includes **Eth** (versions 62, 63, and 64), **Light Ethereum Sub-protocol** (**LES**), **Whisper**, and **Swarm**. These capability protocols are responsible for different application-level communications: for example, Eth is responsible for block synchronization. It makes use of several protocol messages such as `Status`, `Transactions`, `GetBlockHeaders`, and `GetBlockBodies` messages to exchange blockchain information between nodes.

Note that Eth is also referred to as the "Ethereum wire protocol."

> More detail on the Eth capability protocol can be found here: https://github.com/ethereum/devp2p/blob/master/caps/eth.md.

All four of these elements are visualized in the following diagram:

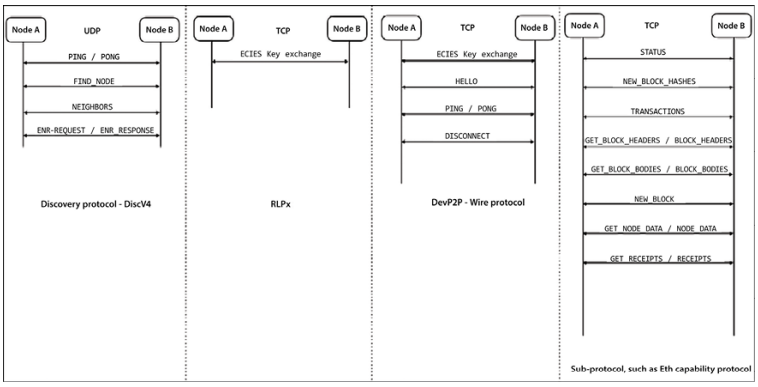

Figure 9.10: Ethereum node discovery, RLPx, DEVP2P, and capability protocols

When setting up private networks, it is advisable to disable the peer discovery mechanism, so that we can manually configure nodes that we need to connect to in our private network. We can then add nodes manually to the network by maintaining a static list of peers.

Now, after this introduction to some underlying theory, all the parameters required for creating a private network will be discussed in detail. In order to create a private net, three components are needed:

- Network ID
- The genesis file
- Data directory to store blockchain data

Let's consider each component in detail.

The **network ID** can be any positive number except any number that is already in use by another Ethereum network. For example, `1` and `3` are in use by Ethereum main net and test net (Ropsten), respectively.

> A list of Ethereum networks is maintained at `https://chainid.network` and network IDs chosen already should not be used to avoid conflicts.

The **genesis file** contains the necessary fields required for a custom genesis block. This is the first block in the network and does not point to any previous block. The Ethereum protocol performs checking in order to ensure that no other node on the internet can participate in the consensus mechanism unless they have the same genesis block. The ChainID is usually used for identification of the network.

The genesis file has a number of parameters. Let's see what each of these parameters means:

- `nonce` : This is a 64-bit hash used to prove that PoW has been sufficiently completed. This works in combination with the `mixhash` parameter.
- `timestamp` : This is the Unix timestamp of the block. This is used to verify the sequence of the blocks and for difficulty adjustment. For example, if blocks are being generated too quickly, that difficulty increases.

- `parentHash` : Being the genesis (first) block, this is always zero as there is no parent block.
- `extraData` : This parameter allows a 32-bit arbitrary value to be saved with the block.
- `gasLimit` : This is the limit on the expenditure of gas per block.
- `difficulty` : This parameter is used to determine the mining target. It represents the difficulty level of the hash required to prove the PoW.
- `mixhash` : This is a 256-bit hash that works in combination with `nonce` to prove that a sufficient amount of computational resources has been spent in order to complete the PoW requirements.
- `coinbase` : This is the 160-bit address where the mining reward is sent as a result of successful mining.
- `alloc` : This parameter contains the list of pre-allocated wallets. The long hex digit is the account to which the balance is allocated.
- `config` : This section contains various configuration information defining the chainID and blockchain hard fork block numbers. This parameter is not required to be used in private networks.

The third component, the **data directory**, does not strictly need to be mentioned. However, if there is more than one blockchain already active on the system, then the data directory should be specified so that a separate directory is used for the new blockchain.

This is the directory where the blockchain data for the private Ethereum network will be saved. A number of parameters are specified in order to run the Ethereum node, fine-tune the configuration, and launch the private network.

If connectivity to only specific nodes is required, which is usually the case in private networks, then a list of **static nodes** is provided as a JSON file. This configuration file is read by the Geth client at the time of starting up and the Geth client only connects to the peers that are present in the configuration file.

We will discuss how to connect to a test net and set up private nets in *Chapter 10*, *Ethereum in Practice*.

Precompiled smart contracts

We discussed smart contracts at length in *Chapter 8*, *Smart Contracts*. It is sufficient to say here that Ethereum supports the development of smart contracts that run on the EVM. There are also various contracts that are available in precompiled format in the Ethereum blockchain to support different functions. These contracts, known as **precompiled contracts** or **native contracts**, are described in the following subsection.

These are not strictly smart contracts in the sense of user-programmed Solidity smart contracts but are in fact functions that are available natively to support various computationally intensive tasks. They run on the local node and are coded within the Ethereum client; for example, Parity or Geth.

There are nine **precompiled contracts** or **native contracts** in the Ethereum Istanbul release. The following subsections outline these contracts and their details.

- The elliptic curve public key recovery function: `ECDSARECOVER` (the ECDSA recovery function) is available at address `0x1`. It is denoted as `ECREC` and requires 3,000 gas units in fees for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in ECC. The ECDSA recovery function is shown as follows:
  ECDSARECOVER (H, V, R, S) = Public Key
  It takes four inputs: *H*, which is a 32-byte hash of the message to be signed, and *V*, *R*, and *S*, which represent the ECDSA signature with the recovery ID and produce a 64-byte public key. *V*, *R*, and *S* have been discussed in detail previously in this chapter.

- The SHA-256-bit hash function: The SHA-256-bit hash function is a precompiled contract that is available at address `0x2` and produces a `SHA256` hash of the input. The gas requirement for SHA-256 (`SHA256`) depends on the input data size. The output is a 32-byte value.
- The RIPEMD-160-bit hash function: The RIPEMD-160-bit hash function is used to provide a RIPEMD-160-bit hash and is available at address `0x3`. The output of this function is a 20-byte value. The gas requirement is dependent on the amount of input data.
- The identity/datacopy function: The identity function is available at address `0x4` and is denoted by the `ID`. It simply defines output as input; in other words, whatever input is given to the `ID` function, it will output the same value. The gas requirement is calculated by a simple formula: *15 + 3 [Id/32]*, where *Id* is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data, albeit with some calculation performed, as shown in the preceding equation.
- The big mod exponentiation function: This function implements a native big integer exponential modular operation. This functionality allows for RSA signature verification and other cryptographic operations. This is available at address `0x05`.
- The BN256 Elliptic curve point addition function: We discussed elliptic curve addition in detail at a theoretical level in *Chapter 4*, *Asymmetric Cryptography*. This is the implementation of the same elliptic curve point addition function. This contract is available at address `0x06`.
- BN256 Elliptic curve scalar multiplication: We discussed elliptic curve multiplication (point doubling) in detail at a theoretical level in *Chapter 4*, *Asymmetric Cryptography*. This is the implementation of the same elliptic curve point multiplication function. Both elliptic curve addition and doubling functions allow for zk-SNARKs and the implementation of other cryptographic constructs. This contract is available at `0x07`.
- BN256 Elliptic curve pairing: The elliptic curve pairing functionality allows for performing elliptic curve pairing (bilinear maps) operations, which enables zk-SNARK verification. This contract is available at address `0x08`.
- Blake2 compression function 'F': This precompiled contract allows the `BLAKE2b` hash function and other related variants to run on the EVM. This improves interoperability with Zcash and other Equihash-based PoW chains. This precompiled contract is implemented at address `0x09`. The EIP is available at **https://eips.ethereum.org/EIPS/eip-152**.

More information on the Blake hash function is available at https://blake2.net

All the precompiled contracts can potentially become native extensions and might be included in the EVM opcodes in the future. There are more precompiled contracts proposed in EIP-2537 for BLS12-381 curve operations, however, the EIP is not implemented yet.

All pre-compiled contracts are present in the `contracts.go` file in the Ethereum source code. It is available at the following link: https://github.com/ethereum/go-ethereum/blob/master/core/vm/contracts.go

Information on EIP-2537 is available here: https://eips.ethereum.org/EIPS/eip-2537

Programming languages

Code for smart contracts in Ethereum is written in high-level languages such as Serpent, **Low-level Lisp-like Language (LLL)**, Solidity, or Vyper, and is converted into the bytecode that the EVM understands for it to be executed.

Solidity

Solidity is one of the high-level languages that has been developed for Ethereum. It uses JavaScript-like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called **solc**.

The official Solidity documentation is available at http://solidity.readthedocs.io/en/latest/

For example, a simple program in Solidity is shown as follows:

```
pragma solidity ^0.8.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

This program is converted into bytecode, as shown in the following subsection. Details on how to compile Solidity code with examples will be provided in *Chapter 11, Tools, Languages, and Frameworks for Ethereum Developers*.

Runtime bytecode

The runtime bytecode is what executes on the EVM. The smart contract code (`contract Test1`) from the previous section is translated into binary runtime and opcodes, as follows:

Binary of the runtime (raw hex codes):

```
6080604052348015600f57600080fd5b506004361060285760003560e01c80634d3e46e414 602d575b600080fd5b
```

Opcodes

There are many different **opcodes** that have been introduced in the EVM. The preceding runtime bytecode is translated into the following opcodes:

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE DUP1 ISZERO PUSH1 0x14 JUMP
UMPDEST PUSH1 0x0 PUSH1 0x2 PUSH1 0x0 SLOAD ADD SWAP1 POP SWAP1 JUMP INVALID LOG2 PUSH5 0x697
```

Opcodes are divided into multiple categories based on the operation they perform. A list of opcodes, their meanings, and usage is available here: `https://ethereum.org/en/developers/docs/evm/opcodes/`.

Now let's move onto another topic, clients and wallets, which are responsible for mining, payments, and management functions, such as account creation on an Ethereum network.

Wallets and client software

As Ethereum is undergoing heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years.

Wallets

A wallet is a generic program that can store private keys and, based on the addresses stored within it, it can compute the existing balance of ether associated with the addresses by querying the blockchain. It can also be used to deploy smart contracts. In *Chapter 10, Ethereum in Practice*, we will introduce the **MetaMask** wallet, which has become the tool of choice for developers.

Having discussed the role of wallets within Ethereum, let's now discuss some common clients.

Geth

This is the official **Go** implementation of the Ethereum client.

The latest version is available at the following link: `https://geth.ethereum.org/downloads/`.

There are other implementations, including C++ implementation Eth, and many others. A list is available here: `https://ethereum.org/en/developers/docs/nodes-and-clients/#execution-clients`.

Light clients

Simple Payment Verification (**SPV**) clients download only a small subset of the blockchain. This allows low-resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify transactions.

A complete Ethereum blockchain and node are not required in this case, and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to Bitcoin SPV clients.

The critical difference between clients and wallets is that clients are full implementations of the Ethereum protocol that support mining, account management, and wallet functions. In contrast, wallets only store the public and private keys, provide essential account management, and interact with the blockchain usually only for payment (transfer of funds) purposes.

In the next section, we will introduce several supporting protocols that are part of the complete decentralized ecosystem based on the Ethereum blockchain.

Supporting protocols

Various supporting protocols are available to assist with the complete decentralized ecosystem. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized to achieve a completely decentralized ecosystem. This includes decentralized messaging and decentralized storage. These are addressed by the Whisper and Swarm protocols, respectively.

Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network. In essence, Whisper is a communication protocol that DApps use to communicate with each other. The data and routing of messages are encrypted within Whisper communications. Whisper makes use of the **DEVP2P** wire protocol for exchanging messages between nodes on the network. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required.

Whisper is also designed to provide a communication layer that cannot be traced and provides *dark communication* between parties. Blockchain can be used for communication, but that is expensive, and a consensus is not really required for messages exchanged between nodes. Therefore, Whisper can be used as a protocol that allows censor-resistant communication. Whisper messages are ephemeral and have an associated **Time to Live** (**TTL**). Whisper is already available with Geth and can be enabled using the `-shh` option while running the Geth Ethereum client.

> The official Whisper documentation is available at `https://eth.wiki/concepts/whisper/whisper`.

Swarm

Swarm has been developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to-peer storage network. Files in this network are addressed by the hash of their content. This contrasts with traditional centralized services, where storage is available at a central location only. It has been developed as a native base layer service for the Ethereum Web3 stack. Swarm is integrated with DEVP2P, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide

a **Distributed Denial-of-Service** (**DDOS**)-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0. Similar to `shh` in Whisper, Swarm has a protocol called `bzz`, which is used by each Swarm node to perform various Swarm protocol operations.

The official Swarm documentation is available at:
https//swarm-guide.readthedocs.io/en/latest/

The following diagram gives a high-level overview of how Swarm and Whisper fit together and work with the Ethereum blockchain:

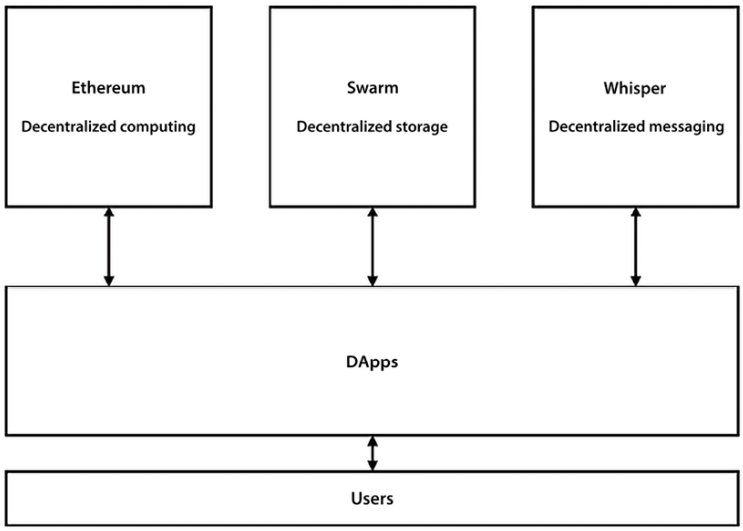

Figure 9.11: Blockchain, Whisper, and Swarm

With the development of Whisper and Swarm, a completely decentralized ecosystem emerges, where Ethereum serves as a decentralized computer, Whisper as a decentralized means of communication, and Swarm as a decentralized means of storage. In simpler words, Ethereum—or more precisely, the EVM—provides compute services, Whisper handles messaging, and Swarm provides storage.

If you recall, in *Chapter 2*, *Decentralization*, we mentioned that decentralization of the whole ecosystem is highly desirable as opposed to decentralization of just the core computation element. The development of Whisper for decentralized communication and Swarm for decentralized storage is a step toward the decentralization of the entire blockchain ecosystem.

Summary

This chapter mainly covered the Ethereum architecture and ecosystem. We introduced the core concepts of the Ethereum blockchain, such as the state machine model, the world and machine states, accounts, and transactions. Moreover, a detailed introduction to the core components of the EVM was also presented. In addition, the Ethereum blockchain and network, wallets, software clients, and supporting protocols were also discussed.

In the next chapter, we will continue to explore Ethereum development. We will look at more concepts, such as programming languages and developing programs for Ethereum blockchain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/ips2H