**Capítulo 9**

Arquitetura do Ethereum

Este capítulo é uma introdução ao blockchain Ethereum, sua arquitetura e design. Vamos apresentar os fundamentos e diversos conceitos teóricos por trás do Ethereum. Uma discussão sobre os diferentes componentes, protocolos e algoritmos relevantes ao blockchain Ethereum também é apresentada.

Cobrimos diferentes elementos do Ethereum, como transações, contas, o estado mundial, a Máquina Virtual Ethereum (EVM), o blockchain, a rede blockchain, carteiras, clientes de software e protocolos de ecossistema de suporte, com o objetivo de compreender as bases técnicas sobre as quais o Ethereum é construído. Os principais tópicos que exploraremos neste capítulo são os seguintes:

* Introdução ao Ethereum
* Criptomoeda
* Chaves e endereços
* Contas
* Transações e mensagens
* A EVM
* Blocos e blockchain
* Nós e mineradores
* Redes
* Contratos inteligentes pré-compilados
* Carteiras e software cliente
* Protocolos de suporte

Vamos começar com uma breve visão geral da fundação, arquitetura e uso do blockchain Ethereum.

**Introdução ao Ethereum**

Vitalik Buterin conceituou o Ethereum em novembro de 2013. A ideia central proposta foi o desenvolvimento de uma linguagem Turing-completa que permite o desenvolvimento de programas arbitrários (contratos inteligentes) para blockchain e Aplicações Descentralizadas (DApps).

Esse conceito contrasta com o Bitcoin, onde a linguagem de script é limitada e permite apenas operações necessárias. Desde sua criação, passou por alguns desenvolvimentos-chave:

* A primeira versão do Ethereum, chamada Olympic, foi lançada em maio de 2015.
* Dois meses depois, uma versão do Ethereum chamada Frontier foi lançada em julho.
* Outra versão chamada Homestead, com várias melhorias, foi lançada em março de 2016.
* A versão mais recente do Ethereum é chamada Arrow Glacier, que adia a bomba de dificuldade, um mecanismo de ajuste de dificuldade que eventualmente força todos os mineradores a pararem de minerar no Ethereum 1 e migrarem para o Ethereum 2.

A especificação formal do Ethereum foi descrita no yellow paper, que pode ser usado para desenvolver implementações de cliente Ethereum.

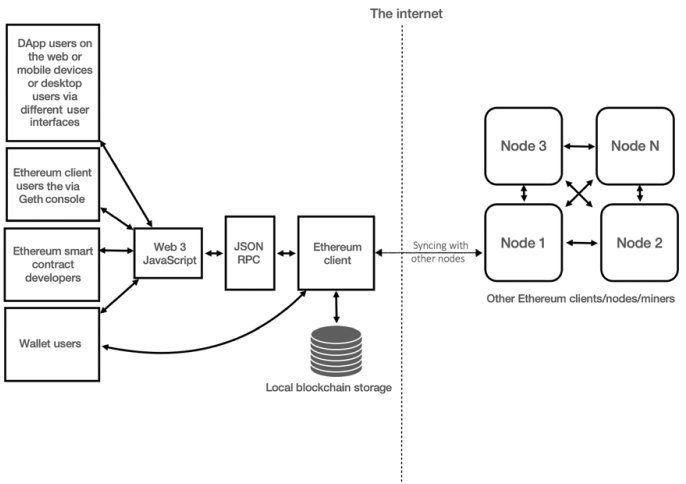
O yellow paper do Ethereum (<https://ethereum.github.io/yellowpaper/paper.pdf>) foi escrito pelo Dr. Gavin Wood, o fundador do Ethereum e da Parity (<http://gavwood.com>), e serve como uma especificação formal para o protocolo Ethereum. Qualquer pessoa pode implementar um cliente Ethereum seguindo as especificações do protocolo definidas no paper.

Embora este documento possa ser um pouco desafiador de ler, especialmente para aqueles que não têm formação em álgebra ou matemática e não estão familiarizados com notações matemáticas, ele contém uma especificação formal completa para o Ethereum. Essa especificação pode ser usada para implementar um cliente Ethereum totalmente compatível. Portanto, é necessário entender esse documento em um nível elevado.

A pilha do blockchain Ethereum consiste em vários componentes. No núcleo, há o blockchain Ethereum executando na rede peer-to-peer do Ethereum. Em segundo lugar, há um cliente Ethereum (geralmente o Geth) que roda nos nós e se conecta à rede peer-to-peer do Ethereum de onde o blockchain é baixado e armazenado localmente. Ele fornece várias funções, como mineração e gerenciamento de contas. A cópia local do blockchain é sincronizada regularmente com a rede. Outro componente é a biblioteca web3.js, que permite a interação com o cliente Geth via interface de Chamada de Procedimento Remoto (RPC).

Uma linha do tempo com todos os marcos e eventos principais está disponível aqui:  
<https://ethereum.org/en/history/>

Uma lista de todos os lançamentos anunciados é mantida em:  
<https://github.com/ethereum/go-ethereum/releases>

Uma lista de elementos presentes no blockchain Ethereum é apresentada aqui:

* Criptomoeda/token ether
* Chaves e endereços
* Contas
* Transações e mensagens
* A EVM
* O blockchain Ethereum
* Nós e mineradores
* A rede Ethereum
* Contratos inteligentes e contratos nativos
* Carteiras e software cliente

**Criptomoeda**

Como incentivo aos mineradores, o Ethereum recompensa com sua própria moeda nativa, chamada ether (abreviado como ETH). Após o ataque à Organização Autônoma Descentralizada (DAO), descrito no Capítulo 8, *Contratos Inteligentes*, um hard fork foi proposto para mitigar o problema; portanto, agora existem dois blockchains Ethereum: um chamado Ethereum Classic, cuja moeda é representada por ETC, enquanto a versão bifurcada é o ETH, que continua a crescer e sobre o qual o desenvolvimento ativo está sendo conduzido. ETC, no entanto, possui seus próprios seguidores com uma comunidade dedicada que continua desenvolvendo o ETC.

Este capítulo é focado no ETH, que atualmente é o blockchain Ethereum mais ativo e oficial.

O ether é cunhado por mineradores como recompensa em moeda pelo esforço computacional que gastam para proteger a rede, verificando transações e blocos. O ether é usado dentro do blockchain Ethereum para pagar pela execução de contratos na EVM. O ether é usado para comprar gás como combustível criptográfico, o qual é necessário para realizar computações no blockchain Ethereum.

A seguir, vamos analisar chaves e endereços.

**Chaves e endereços**

Chaves e endereços são usados no blockchain Ethereum para representar propriedade e transferir ether. As chaves utilizadas são compostas por pares de partes privadas e públicas. A chave privada é gerada aleatoriamente e mantida em segredo, enquanto a chave pública é derivada da chave privada. Endereços são derivados das chaves públicas e são códigos de 20 bytes usados para identificar contas.

O processo de geração de chaves e derivação de endereços é o seguinte:

1. Primeiro, uma chave privada é escolhida aleatoriamente (um número inteiro positivo de 256 bits) sob as regras definidas pela especificação da curva elíptica secp256k1 (no intervalo [1, secp256k1n − 1]).
2. A chave pública é então derivada dessa chave privada usando a função de recuperação do Algoritmo de Assinatura Digital de Curvas Elípticas (ECDSA). Discutiremos isso na seção *Transações e mensagens*, no contexto de assinaturas digitais.
3. Um endereço é derivado da chave pública, especificamente, dos 160 bits mais à direita do hash Keccak da chave pública.

Uma descrição de como chaves e endereços se parecem no Ethereum é a seguinte:

* Uma chave privada é basicamente um inteiro válido de 256 bits diferente de zero, escolhido aleatoriamente do intervalo do tamanho do campo da curva elíptica (ordem). O software cliente Ethereum utiliza o gerador de números aleatórios do sistema operacional para gerar esse número.

Uma tabela de denominação está disponível em:  
<https://ethdocs.org/en/latest/ether.html#denominations>

**Exemplo:**

Uma vez escolhido esse inteiro, ele é representado em formato hexadecimal (64 caracteres hexadecimais), como mostrado aqui:  
b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4.

* Uma chave pública é uma coordenada x, y — um ponto em uma curva elíptica que satisfaz a equação da curva elíptica. A chave pública é derivada da chave privada. Uma vez que temos a chave privada escolhida aleatoriamente, ela é multiplicada por um ponto predefinido na curva elíptica chamado ponto gerador, o que produz outro ponto em algum lugar da curva. Esse outro ponto é a chave pública e é representado no Ethereum como uma string hexadecimal de 130 caracteres:  
  3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260.
* Um endereço é um identificador único no blockchain Ethereum, derivado da chave pública ao fazer seu hash com a função Keccak-256 e manter os últimos 20 bytes do hash produzido. Um exemplo é mostrado aqui:  
  0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32.

O par de chaves pública e privada gerado durante o processo de criação de nova conta é armazenado em arquivos de chave localmente no disco. Esses arquivos de chave são armazenados no diretório *keystore* presente no caminho relevante de acordo com o sistema operacional em uso.

No sistema operacional Linux, sua localização é a seguinte: ~/.ethereum/keystore

O conteúdo de um arquivo JSON do keystore é mostrado aqui como exemplo. Você consegue correlacionar alguns dos nomes com o que já aprendemos no Capítulo 3, *Criptografia Simétrica*, e no Capítulo 4, *Criptografia Assimétrica*? No exemplo a seguir, ele foi formatado para melhor visibilidade:



Observe que a chave privada acima é mostrada apenas como exemplo e **não deve** ser reutilizada.

A chave privada é armazenada em formato criptografado no arquivo keystore. Ela é gerada quando uma nova conta é criada utilizando a senha e a chave privada. O arquivo keystore também é chamado de arquivo UTC, pois o formato de nomeação começa com "UTC" seguido por um timestamp de data. Um exemplo de nome de arquivo keystore é mostrado aqui:

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

Em sistemas operacionais do tipo Unix, o arquivo keystore normalmente está localizado no diretório pessoal do usuário. Por exemplo, no macOS, o arquivo keystore é armazenado na seguinte localização:  
~/Library/Ethereum/keystore/  
Para outros sistemas operacionais, consulte as instruções de instalação disponíveis em:  
<https://geth.ethereum.org/docs/getting-started/installing-geth>

Como o arquivo keystore está presente no disco, é muito importante mantê-lo seguro. É recomendável que ele seja também copiado como backup. Se os arquivos keystore forem perdidos, sobrescritos ou corrompidos de alguma forma, **não há como recuperá-los**. Isso significa que qualquer ether associado à chave privada também será irrecuperável.

Agora, descrevemos os elementos do arquivo keystore com senha e o que eles representam. O principal propósito deste arquivo é armazenar a configuração que, quando fornecida com a senha da conta, gera uma chave privada de conta descriptografada. Esta chave privada é então usada para assinar transações:

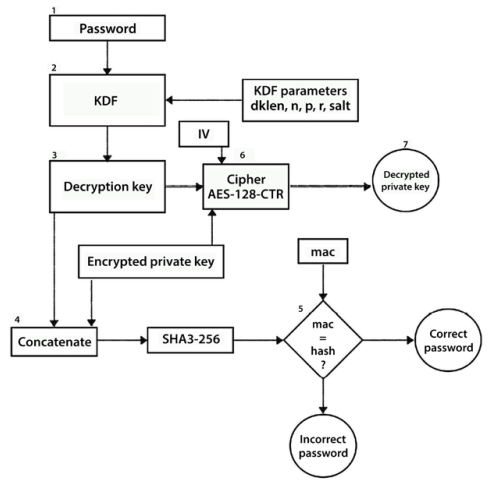
* **Address**: Este é o endereço público da conta, usado para identificar o remetente ou o destinatário.
* **Crypto**: Este campo contém os parâmetros de criptografia.
* **Cipher**: Este é o cifrador usado para criptografar a chave privada. No diagrama a seguir, AES-128-CTR indica o Padrão de Criptografia Avançada de 128 bits em modo contador. Lembre-se de que cobrimos isso no Capítulo 3, *Criptografia Simétrica*.
* **Ciphertext**: Esta é a chave privada criptografada.
* **Cipherparams**: Representa os parâmetros exigidos pelo algoritmo de criptografia, AES-128-CTR.
* **IV**: Este é o vetor de inicialização de 128 bits para o algoritmo de criptografia.
* **KDF**: Esta é a função de derivação de chave. Neste caso, é o *scrypt*.
* **KDFParams**: São os parâmetros para a função de derivação de chave (KDF).
  + **Dklen**: Este é o comprimento da chave derivada. No exemplo, é 32.
  + **N**: É o número de iterações.
  + **P**: Fator de paralelização; o valor padrão é 1.
  + **R**: Este é o tamanho do bloco da função hash subjacente. É definido como 8, que é a configuração padrão.
  + **Salt**: Este é o valor aleatório de sal para o KDF.
* **Mac**: Este é o resultado do hash Keccak-256 obtido após a concatenação dos 16 bytes à esquerda da chave derivada com o texto cifrado.

**É imperativo manter em segurança as senhas associadas criadas no momento da criação das contas e quando os arquivos de chave forem produzidos.**

<https://geth.ethereum.org/docs/getting-started/installing-geth>

* **ID**: Este é um número de identificação aleatório.
* **Version**: O número da versão do formato de arquivo, atualmente versão 3.

Agora vamos explorar como todos esses elementos funcionam juntos. Todo esse processo é visualizado no diagrama a seguir:



Como mostrado no diagrama anterior, podemos dividir o processo em diferentes etapas:

1. Primeiro, a senha é alimentada na função KDF.
2. A KDF recebe diversos parâmetros, a saber: dklen, n, p, r e salt, e produz uma chave de descriptografia.
3. A chave de descriptografia é concatenada com a chave privada criptografada (ciphertext). A chave de descriptografia também é passada para o algoritmo de cifra, AES-128-CTR.
4. A chave de descriptografia concatenada com o texto cifrado é hasheada usando a função de hash SHA3-256.
5. O mac é passado para a função de verificação, onde o hash produzido na etapa 4 é comparado com o mac. Se ambos coincidirem, a senha é válida; caso contrário, a senha está incorreta.
6. A função de cifra, que recebe como entrada o Vetor de Inicialização (IV), o texto cifrado (ciphertext) e a chave de descriptografia, descriptografa a chave privada.
7. A chave privada descriptografada é produzida.

Essa chave privada descriptografada é então usada para assinar transações na rede Ethereum.

Outro elemento fundamental no Ethereum é a conta, que é necessária para interagir com o blockchain. Ela representa um usuário ou um contrato inteligente.

**Contas**

Uma conta é um dos principais blocos de construção do blockchain Ethereum. Ela é definida por pares de chaves privadas e públicas. As contas são usadas por usuários para interagir com o blockchain por meio de transações. Uma transação é digitalmente assinada por uma conta antes de ser submetida à rede por meio de um nó. Como o Ethereum é uma máquina de estados impulsionada por transações, o estado é criado ou atualizado como resultado da interação entre contas e da execução de transações. Todas as contas possuem um estado que, quando combinado, representa o estado da rede Ethereum. A cada novo bloco, o estado da rede Ethereum é atualizado. As operações realizadas entre e nas contas representam transições de estado. A transição de estado é alcançada utilizando o que é chamado de função de transição de estado do Ethereum, que funciona da seguinte forma:

1. Confirma a validade da transação verificando a sintaxe, a validade da assinatura e o nonce.
2. A taxa de transação é calculada e o endereço de envio é resolvido usando a assinatura. Além disso, o saldo da conta do remetente é verificado e subtraído de forma correspondente, e o nonce é incrementado. Um erro é retornado se o saldo da conta for insuficiente.
3. Fornece ETH suficiente (o preço do gás) para cobrir o custo da transação. Veremos gás e conceitos relevantes em breve neste capítulo. Isso é cobrado por byte e é proporcional ao tamanho da transação. Nesta etapa, ocorre a transferência real de valor. O fluxo é da conta do remetente para a conta do destinatário. A conta é criada automaticamente se a conta de destino especificada na transação ainda não existir. Além disso, se a conta de destino for um contrato, então o código do contrato é executado. Isso também depende da quantidade de gás disponível. Se houver gás suficiente, o código do contrato será executado totalmente; caso contrário, ele será executado até o ponto em que ocorrer um erro de “Out of Gas” (OOG).
4. Em casos de falha da transação devido a saldo de conta insuficiente ou gás insuficiente, todas as alterações de estado são revertidas, exceto pelo pagamento da taxa, que é pago aos mineradores.
5. Por fim, o restante (se houver) da taxa é devolvido ao remetente como troco, e a taxa é paga aos mineradores de forma correspondente. Neste ponto, a função retorna o estado resultante, que também é armazenado no blockchain.

Agora que entendemos contas no Ethereum de maneira geral, vamos examinar os dois tipos de contas que existem no Ethereum.

**Conta de propriedade externa (Externally Owned Account — EOA)**

As EOAs são semelhantes às contas no Bitcoin que são controladas por uma chave privada. Assim como no Bitcoin, onde um endereço é controlado por uma chave privada, os endereços do Ethereum também são controlados por suas respectivas chaves privadas. Elas são definidas por três elementos: endereço, saldo e nonce. Possuem as seguintes propriedades:

* Possuem um estado.
* Estão associadas a um usuário humano, sendo por isso também chamadas de contas de usuário.
* As EOAs têm um saldo em ether.
* Elas podem enviar transações, ou seja, podem transferir valor e iniciar código de contrato.
* Não têm código associado.
* São controladas por chaves privadas.
* O endereço público de uma EOA é derivado de sua chave privada.
* EOAs não podem iniciar mensagens do tipo “call message”.
* As contas contêm uma estrutura de chave-valor.
* As EOAs podem iniciar mensagens de transação.

**Conta de contrato (Contract Account — CA)**

As CAs são contas que têm código associado a elas, além de serem controladas com uma chave privada. Elas são definidas por cinco elementos: endereço, saldo, nonce, StorageRoot e codeHash. Possuem as seguintes propriedades:

* Possuem um estado.
* Não estão intrinsecamente associadas a nenhum usuário ou ator no blockchain.
* As CAs têm um saldo em ether.
* Elas possuem código associado que é mantido na memória/armazenamento no blockchain. Elas podem ser acionadas e executar código em resposta a uma transação ou a uma mensagem de outros contratos. Vale notar que, devido à propriedade de completude de Turing do blockchain Ethereum, o código dentro das CAs pode ter qualquer nível de complexidade. O código é executado pela EVM por cada nó de mineração na rede Ethereum. A EVM será discutida mais adiante neste capítulo, na seção *A EVM*.
* O endereço público de uma CA é a combinação do endereço público da EOA que a criou com o nonce do contador de transações (quantas transações a EOA já enviou).
* Elas têm acesso ao armazenamento e podem manipular seu armazenamento.
* Além disso, as CAs podem manter seus estados permanentes e chamar outros contratos. Prevê-se que em futuras versões do Ethereum a distinção entre EOAs e CAs possa ser eliminada.
* Uma CA só pode ser criada a partir de uma EOA ou a partir de uma CA já existente (contrato).
* As CAs não podem iniciar mensagens de transação.
* As CAs podem iniciar mensagens do tipo “call message”, ou seja, podem chamar outros contratos e funções de contratos.
* As CAs podem transferir ether.
* As CAs contêm uma estrutura de chave-valor.
* Os endereços das CAs são gerados quando elas são implantadas. Esse endereço do contrato é usado para identificar sua localização no blockchain.

As contas permitem interação com o blockchain por meio de transações.

**Transações e mensagens**

Uma transação no Ethereum é um pacote de dados assinado digitalmente (usando uma chave privada) que contém as instruções que, quando concluídas, resultam em uma chamada de mensagem ou criação de contrato. As transações são construídas por um ator externo ao blockchain Ethereum ou por alguma ferramenta de software externa. Contratos inteligentes não podem enviar uma transação.

Inicialmente, no Ethereum, havia transações simples, transações de chamada de mensagem e transações de criação de contrato. Posteriormente, à medida que o sistema evoluiu, surgiu a necessidade de introduzir mais tipos e também de tornar mais fácil a introdução de novos recursos e tipos de transação futuros, bem como ser capaz de distinguir diferentes tipos de transações. Novos recursos, como listas de acesso (*access lists*) e o EIP-1559, foram introduzidos mantendo a compatibilidade com transações legadas. Inovações adicionais resultaram na introdução de transações tipadas, que foram introduzidas no EIP-2718. Elas definem um novo tipo de transação, que é um envelope para tipos de transação futuros.

Explicaremos em breve o que é uma transação Ethereum e consideraremos seus diferentes tipos. Mas, antes disso, vejamos um conceito importante, a *Merkle Patricia Tree* (MPT), que é a base da integridade no blockchain.

**MPTs**

É essencial entender o que é uma MPT e como ela garante a integridade dos dados do blockchain. Uma MPT é usada em todas as estruturas de dados relevantes para o estado e as transações, como o estado mundial, recibos, armazenamento e o estado da conta. Lembre-se, discutimos a árvore de Merkle no Capítulo 3, *Criptografia Simétrica*.

Os nós-folha de uma árvore de Merkle são emparelhados e as transações são concatenadas a partir de um bloco para o qual a árvore de Merkle será construída. Uma vez que esses nós-folha são hasheados juntos, um outro conjunto de nós é criado. Ele é novamente emparelhado, concatenado e hasheado até que um único valor final de hash seja obtido, chamado de raiz de Merkle (*Merkle root*). No cabeçalho de cada bloco, há um campo no qual essa raiz de Merkle é armazenada e serve como prova de todas as transações incluídas no bloco. Devido à propriedade de resistência a colisões das funções de hash, é possível provar que mudar até mesmo um único bit em qualquer transação resultará na alteração do valor de hash de Merkle e, consequentemente, no hash do cabeçalho do bloco. Por causa disso, a raiz de Merkle, ou o hash do cabeçalho do bloco, é uma impressão digital única para o bloco e todas as suas transações. Cada novo bloco também aponta para o bloco anterior incluindo o hash de seu cabeçalho, e esse padrão se repete até o bloco gênese, formando uma cadeia de blocos vinculados por hash — ou simplesmente um blockchain. Esse encadeamento por hash resulta em vinculação das transações, já que a raiz de Merkle das transações está presente em cada bloco; podemos verificar toda essa estrutura de dados quanto à integridade.

A maneira como a integridade é verificada é bastante simples:

1. Precisamos de uma referência ao bloco mais recente (cabeçalho do bloco) e ao bloco gênese genuíno.
2. Cada campo de hash no ponteiro de hash presente no cabeçalho do bloco é comparado com o hash do cabeçalho do bloco anterior, e esse processo continua até que o bloco gênese seja verificado. Se a verificação falhar em qualquer ponto, isso significa que houve alguma modificação nas transações ou no cabeçalho do bloco que resultou na mudança da raiz de Merkle. Esse processo verifica a integridade da cadeia.
3. Além disso, durante as verificações de validação do bloco, é verificado se a raiz de Merkle no cabeçalho do bloco corresponde ao hash de todas as transações no bloco.

Para verificar se uma determinada transação faz parte do blockchain ou não, a raiz de Merkle precisa ser recomputada e verificada. Esse processo é chamado de autenticação de caminho de Merkle (*Merkle path authentication*) ou simplesmente *Merkle proof*. Veja como funciona:

Se uma transação **t** está incluída no bloco **b**, então, desde que a função de hash seja resistente a colisões e o valor da raiz de Merkle seja autêntico, **t** pode ser verificada da seguinte forma:

1. Primeiro, calcule o hash do elemento que se deseja verificar se existe no bloco. Suponhamos que seja a transação **t**.
2. Em seguida, obtenha o hash da raiz de Merkle armazenada no cabeçalho do bloco.
3. Calcule um valor candidato de raiz de Merkle usando a transação **t**, para a qual se requer a prova de inclusão, e tomando os valores fora do caminho (*off-path*, ou seja, fora do caminho de autenticação). Note que o caminho de autenticação inclui os irmãos dos nós desde a transação **t** até o nó raiz (a raiz de Merkle).
4. Se o valor de hash raiz candidato corresponder ao hash de raiz de Merkle armazenado no cabeçalho do bloco **b**, confirma-se que a transação **t** faz parte do bloco **b**.

Agora que entendemos como a prova de inclusão de Merkle é realizada e a integridade do blockchain é verificada usando árvores de Merkle, podemos aplicar a mesma lógica às MPTs. Podemos usar a MPT para armazenar, atualizar, inserir, excluir e recuperar pares chave-valor. Cada nó na árvore pode conter uma parte de uma chave, um valor e ponteiros para outros nós. Uma MPT é obtida substituindo os ponteiros por ponteiros de hash, o que significa que os nós são referenciados pelo hash de seus conteúdos em vez de ponteiros de endereço simples. A raiz da MPT agora serve ao mesmo propósito que a raiz da árvore de Merkle que acabamos de ver, o que significa que ela representa o conteúdo completo de toda a árvore. Isso implica que o hash do nó raiz da MPT pode ser usado para autenticar todo o conjunto de dados chave-valor do qual a MPT é composta.

O Ethereum vai um passo além e personaliza essa MPT. Os dados de cada nó da árvore são armazenados em um banco de dados, e um valor de hash é usado para buscar esse conteúdo. No Ethereum, há três raízes de Merkle no cabeçalho do bloco:

* **TransactionsRoot**: Este é o hash raiz da MPT da árvore de transações incluídas no bloco. Cada bloco possui sua própria árvore de transações separada.
* **StateRoot**: Este é o hash raiz da MPT do estado mundial, isto é, o estado persistente.
* **ReceiptsRoot**: Este é o hash raiz da MPT dos recibos de transações no bloco. Cada bloco possui sua própria árvore de recibos.

Vamos agora focar nas transações e, em seguida, discutiremos o estado mundial, o estado da conta e os recibos de transação, onde os conceitos de StateRoot e ReceiptsRoot ficarão claros.

**Componentes da transação**

As transações no Ethereum são compostas por alguns campos padrão, os quais são descritos a seguir:

* **Type**: Tipo de transação do EIP-2718 — tipo 0 para legado, 1 para EIP-2930 e 2 para EIP-1559. O EIP-2718 define um novo envelope generalizado de transações que permite a criação de diferentes tipos de transações. É um formato simples onde o campo de tipo de transação é concatenado com um *payload* de transação, que pode ser de tipos diferentes, dependendo do tipo de transação. Podem existir 128 tipos de transações possíveis, já que o tipo de transação é um número definido no intervalo de 0 a 0x7f. Definir envelopes generalizados como esse permite compatibilidade com versões anteriores e menor complexidade, ao mesmo tempo em que se pode criar diferentes tipos de novas transações. O *payload* da transação é um vetor de bytes definido pelo tipo da transação.

Há também três tipos padrão de transações legadas com base na saída que produzem:

* **Transações simples**: São as transações padrão usadas para pagamentos. Transferem ether (fundos) entre EOAs.
* **Transações de chamada de mensagem**: São usadas para executar contratos inteligentes, isto é, invocar métodos em contratos inteligentes já implantados.
* **Transações de criação de contrato**: Resultam na criação de uma nova conta de contrato, ou seja, um contrato inteligente — uma conta com código associado.

Elaboraremos mais sobre esses tipos centrais de transações adiante, na seção **Tipos de transação**.

* **Nonce**: O nonce é um número que é incrementado em um a cada vez que uma transação é enviada pelo remetente. Ele deve ser igual ao número de transações enviadas e é usado como identificador único da transação. Um valor de nonce só pode ser usado uma vez. Ele é usado para proteção contra reexecução (replay) na rede.
* **Gas price**: O campo *gas price* representa a quantidade de Wei necessária para executar a transação. Em outras palavras, é o valor em Wei que você está disposto a pagar por essa transação. É cobrado por unidade de gás para todos os custos de computação incorridos como resultado da execução dessa transação.
* **Gas limit**: O campo *gas limit* contém o valor que representa a quantidade máxima de gás que pode ser consumida para executar a transação. O conceito de gás e limites de gás será abordado mais detalhadamente posteriormente neste capítulo. Por ora, é suficiente dizer que esse é o valor da taxa, em ether, que o usuário (por exemplo, o remetente da transação) está disposto a pagar pela computação.
* **To**: Como o nome sugere, o campo *To* é um valor que representa o endereço do destinatário da transação. É um valor de 20 bytes.
* **Value**: O campo *value* representa o número total de Wei a serem transferidos para o destinatário; no caso de uma CA, isso representa o saldo que o contrato manterá.
* **Signature**: As transações são assinadas usando assinaturas ECDSA recuperáveis. A assinatura é composta por três campos: V, R e S. Esses valores representam a assinatura digital (R, S) e algumas informações que podem ser usadas para recuperar a chave pública (V). Além disso, o remetente da transação pode ser determinado a partir desses valores. A assinatura é baseada no esquema ECDSA e utiliza a curva secp256k1. A teoria da Criptografia de Curvas Elípticas (ECC) foi discutida no Capítulo 4, *Criptografia Assimétrica*. Nesta seção, a ECDSA será apresentada no contexto de seu uso no Ethereum.

**Wei** é a menor denominação do ether; portanto, é usado para contar o ether.

* **V** é um valor de um byte que representa o tamanho e o sinal do ponto da curva elíptica e pode ser 27 ou 28. V é usado no contrato de recuperação ECDSA como um valor de recuperação. Esse valor é usado para recuperar (derivar) a chave pública a partir da chave privada. Na secp256k1, o valor de recuperação é esperado como 0 ou 1. No Ethereum, isso é deslocado para 27.
* **R** é derivado de um ponto calculado na curva. Primeiro, um número aleatório é escolhido, que é multiplicado pelo gerador da curva para calcular um ponto na curva. A parte da coordenada x desse ponto é R. R é codificado como uma sequência de 32 bytes. R deve ser maior que 0 e menor que o limite secp256k1n (115792089237316195423570985008687907852837564279074904382605163141518161494337).
* **S** é calculado multiplicando R pela chave privada e somando ao hash da mensagem a ser assinada e, por fim, dividindo pelo número aleatório escolhido para calcular R. S também é uma sequência de 32 bytes. R e S juntos representam a assinatura.

Para assinar uma transação, a função **ECDSASIGN** é usada, a qual recebe a mensagem a ser assinada e a chave privada como entrada e produz:  
**V** (um valor de 1 byte),  
**R** (valor de 32 bytes)  
e **S** (outro valor de 32 bytes).  
A equação é a seguinte:

ECDSASIGN (Message, Private Key) = (V, R, S)

* **Init**: O campo *Init* é usado apenas em transações destinadas à criação de contratos, isto é, transações de criação de contrato. Ele representa um vetor de bytes de comprimento ilimitado que especifica o código da EVM a ser usado no processo de inicialização da conta. O código contido neste campo é executado apenas uma vez, quando a conta é criada pela primeira vez. Esse código (*init*) é destruído imediatamente após isso. *Init* também retorna outra seção de código, chamada de *body*, que persiste e é executada em resposta às chamadas de mensagem que a CA possa receber. Essas chamadas de mensagem podem ser enviadas por meio de uma transação ou de uma execução de código interna.
* **Data**: Se a transação for uma chamada de mensagem (isto é, chamada de métodos em um contrato inteligente implantado), então o campo *Data* é usado em vez de *init*. Ele contém os dados de entrada da chamada de mensagem. Em outras palavras, contém dados como o nome da função no contrato inteligente a ser chamada e os parâmetros relevantes. Tem tamanho ilimitado e é organizado como um vetor de bytes.

A transação do Tipo 1 (EIP-2930) introduziu os seguintes campos:

* **ChainID**: A transação só é válida na rede blockchain com este identificador de cadeia específico.
* **AccessList**: Este campo contém uma lista de entradas de acesso. Cada lista de acesso é uma tupla de endereços de contas e uma lista de chaves de armazenamento que a transação pretende acessar.
* **yParity**: Esta é a paridade Y da assinatura, usada em vez de V em uma transação legada. A paridade do valor Y de uma assinatura secp256k1 é 0 para par e 1 para ímpar.

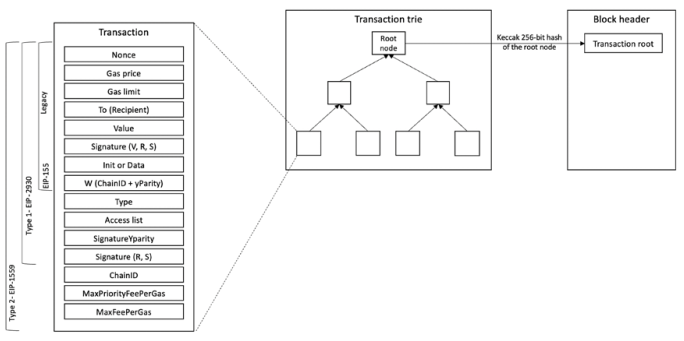
Transações legadas não possuem o campo *AccessList*. Além disso, *ChainID* e *yParity* são combinados em um único campo, **W**, descrito abaixo, nas transações legadas.

* **W**: Este é um valor escalar que codifica a paridade Y e o *ChainID*. Baseado no EIP-155.

O **EIP-1559** introduz um novo tipo de transação. Chamamos isso de transações do Tipo 2. A maioria dos campos permanece igual à estrutura legada, porém, alguns novos campos foram adicionados, conforme descrito abaixo:

* **ChainID**: Este campo agora faz parte do *payload* da transação. Antes, ele era codificado no valor da assinatura V devido ao EIP-155.
* **maxPriorityFeePerGas** e **maxFeePerGas**: Dois novos campos introduzidos como substitutos para o campo legado *gasPrice*.
* **Gas Limit**, **destination**, **amount** e **data** permanecem os mesmos das transações legadas.
* **AccessList**: Igual ao EIP-2930, onde a lista de acesso pode ser definida.
* **signatureYparity**: O valor da assinatura V nas transações legadas é substituído por *signatureYparity*, que pode ser 0 ou 1, dependendo de qual coordenada Y na curva elíptica é usada. Nas transações legadas, costumava ser 27, 28 ou 35, 36 nas transações do EIP-155 como identificador de recuperação da assinatura.
* **signatureR** e **signatureS** permanecem os mesmos das transações legadas e das transações do Tipo 1. Lembre-se do Capítulo 4, *Criptografia Assimétrica*, que estes são simplesmente dois pontos na curva elíptica.

Quando uma transação é executada com sucesso, um registro na árvore de transações é criado contendo todos os campos relacionados à transação. Essa estrutura é visualizada no diagrama a seguir, onde uma transação é uma tupla dos campos mencionados anteriormente, que é então incluída em uma *trie* de transações (uma MPT modificada) composta pelas transações a serem incluídas. Finalmente, o nó raiz da *trie* de transações é hasheado usando o algoritmo Keccak de 256 bits e incluído no cabeçalho do bloco junto com a lista de transações no bloco:



**Codificação Recursive Length Prefix (RLP)**

Para definir a RLP, primeiro precisamos entender o conceito de serialização. Serialização é simplesmente um mecanismo comumente utilizado em computação para codificar estruturas de dados em um formato (uma sequência de bytes) que seja adequado para armazenamento e/ou transmissão através de links de comunicação em uma rede. Uma vez que um receptor recebe os dados serializados, ele os desserializa para obter a estrutura de dados original. Serialização e desserialização também são referidas como *marshaling* e *un-marshaling*, respectivamente. Alguns formatos de serialização comumente utilizados incluem XML, JSON, YAML, *protocol buffers* e XDR. Existem dois tipos de formatos de serialização: texto e binário. Em um blockchain, há necessidade de serializar e desserializar diferentes tipos de dados, como blocos, contas e transações, para permitir sua transmissão pela rede e armazenamento nos clientes.

A RLP é um esquema de codificação desenvolvido pelos desenvolvedores do Ethereum. Trata-se de um esquema de codificação especialmente desenvolvido que é utilizado no Ethereum para serializar dados binários para armazenamento ou transmissão pela rede, bem como para salvar o estado em uma MPT em mídias de armazenamento. É um esquema de codificação binária determinístico e consistente usado para serializar objetos no blockchain Ethereum, como estado de contas, transações, mensagens e blocos. Ela opera sobre strings e listas para produzir bytes brutos que são adequados para armazenamento e transmissão. A RLP é um formato de serialização minimalista e de simples implementação, que não define nenhum tipo de dado e simplesmente armazena estruturas como arrays aninhados. Em outras palavras, a RLP não codifica tipos de dados específicos; em vez disso, seu principal propósito é codificar estruturas.

Agora que definimos a RLP, podemos nos aprofundar em transações e outros elementos relevantes do blockchain Ethereum. Cada operação na rede Ethereum custa uma certa quantidade de ETH e é cobrada por meio de um mecanismo de taxa. Essa taxa também é chamada de *gas*. Vamos agora introduzir esse mecanismo em detalhes.

Por que precisamos de um novo esquema de codificação quando já existem tantos formatos de serialização diferentes disponíveis?  
A resposta para essa pergunta é que a RLP é um esquema determinístico, enquanto outros esquemas podem produzir resultados diferentes para a mesma entrada, o que é absolutamente inaceitável em um blockchain. Mesmo uma pequena alteração levaria a um hash totalmente diferente e resultaria em problemas de integridade de dados que tornariam todo o blockchain inútil.

Mais informações sobre a RLP estão disponíveis na wiki do Ethereum, em:  
<https://eth.wiki/en/fundamentals/rlp>

**Gas**

Gas é necessário para cada operação realizada no blockchain Ethereum. Este é um mecanismo que garante que laços infinitos não possam fazer com que todo o blockchain pare, devido à natureza Turing-completa da EVM. Uma taxa de transação é cobrada como uma quantia de ether e é retirada do saldo da conta originadora da transação.

Gas é cobrado em três cenários como pré-requisito para a execução de uma operação:

* A computação de uma operação
* Para criação de contrato ou chamadas de mensagem
* Um aumento no uso de memória

Uma taxa é paga para que as transações sejam incluídas por mineradores para mineração. Se essa taxa for muito baixa, a transação pode nunca ser incluída; quanto maior a taxa, maiores as chances de que as transações sejam escolhidas pelos mineradores para inclusão no bloco. Por outro lado, se uma transação que tem uma taxa apropriada for incluída no bloco por mineradores, mas tiver muitas operações complexas a executar, isso pode resultar em uma exceção de OOG (*Out of Gas*) se o custo de gas não for suficiente. Nesse caso, a transação falhará, mas ainda será incluída no bloco, e o originador da transação não receberá reembolso.

Os custos de transação podem ser estimados utilizando a seguinte fórmula:

Custo total = gasUsado \* gasPreco

Aqui, *gasUsado* é o total de gas que deverá ser utilizado pela transação durante a execução, e *gasPreco* é especificado pelo originador da transação como incentivo para os mineradores incluírem a transação no próximo bloco. Isso é especificado em ETH. Cada opcode da EVM tem uma taxa atribuída. É uma estimativa, pois o gas utilizado pode ser mais ou menos que o valor originalmente especificado pelo originador da transação. Por exemplo, se a computação demorar muito ou o comportamento do contrato inteligente mudar em resposta a outros fatores, a execução da transação pode realizar mais ou menos operações do que o previsto inicialmente e pode acabar consumindo mais ou menos gas. Se a execução ficar sem gas (OOG), tudo é imediatamente revertido; caso contrário, se a execução for bem-sucedida e sobrar gas, ele é devolvido ao originador da transação.

Cada operação custa uma certa quantidade de gas; uma tabela de taxas de alto nível de algumas operações é mostrada como exemplo aqui:

| **Nome da operação** | **Custo em gas** |
| --- | --- |
| Stop | 0 |
| SHA3 | 30 |
| SLOAD | 800 |
| Transação | 21000 |
| Criação de contrato | 32000 |

Um site que mantém rastreamento do preço mais recente do gas e fornece outras estatísticas e calculadoras valiosas está disponível em:  
<https://ethgasstation.info>

Com base na tabela de taxas anterior e na fórmula discutida anteriormente, um exemplo de cálculo da operação SHA-3 pode ser feito da seguinte forma:

* SHA-3 custa 30 unidades de gas.
* Suponha que o preço atual do gas seja 25 GWei, e converta isso para ETH, o que resulta em 0,000000025 ETH. Após multiplicar ambos, 0,000000025 \* 30, obtemos 0,00000075 ETH.
* No total, 0,00000075 ETH é o valor de gas que será cobrado.

As taxas de transação têm sido muito altas e, por vezes, a taxa excede o valor do ativo. No Ethereum, a alta taxa de transação se deve a restrições de escalabilidade e ao funcionamento da economia do Ethereum, em vez de uma limitação de projeto. Há dois aspectos a se considerar ao pensar em taxas de gas: opcodes e preços de gas.

Uma ideia que pode vir à mente é reduzir o custo de gas dos opcodes da EVM, reduzindo assim as taxas. No entanto, isso não funcionaria — o custo de gas dos opcodes estima o tempo necessário para processar um opcode, calculado grosseiramente em milissegundos mais algumas outras restrições. Em contraste, o preço do gas é uma função do mercado de taxas do Ethereum e da oferta e demanda. Esse mercado existe devido ao tamanho limitado dos blocos (tamanho máximo do bloco — 12,5M de gas — limite de gas total que pode ser consumido por transações em um bloco), onde todos estão tentando fazer com que sua transação seja processada primeiro. Portanto, trata-se de um mecanismo de mercado de leilão de primeiro preço. Todos os participantes fazem lances simultaneamente e o maior lance vence. Se o custo dos opcodes for reduzido, então, devido à oferta e demanda, a rede sempre terá esse preço de taxa elevado por causa desse mecanismo de mercado de taxas.

Os usuários também estão dispostos a pagar mais para que suas transações sejam processadas primeiro, e os mineradores também podem ignorar transações com taxa baixa e ordená-las como quiserem. Assim, o preço do gas na rede continuará alto porque, fundamentalmente, o mercado de taxas de transações do Ethereum é um mercado onde o maior lance vence. A menos que isso mude, apenas reduzir o custo dos opcodes não resultará em nenhuma melhoria. O custo dos opcodes existe por uma razão diferente. Em comparação, o preço do gas é resultado da oferta, demanda e do tamanho limitado dos blocos.

Outros problemas incluem ataques de negação de serviço (DoS) e computadores mais lentos. Computadores mais lentos podem não ser capazes de processar opcodes rapidamente o suficiente. Por exemplo, imagine que uma instrução padrão de transferência custa 21.999 unidades de gas. Isso ocorre porque envolve alguma cooperação de ECC (criptografia de curvas elípticas), que se estima levar algum tempo. Imagine que o tempo estimado seja, hipoteticamente, 21 milissegundos, pois o custo de gas é uma estimativa do tempo necessário para processar uma instrução de opcode. Agora, se reduzirmos o custo de gas, um computador mais rápido será capaz de fazer isso mais rapidamente e dentro de, digamos, 10.000 unidades de gas. No entanto, um computador mais lento pode não conseguir completar toda a operação a tempo.

Portanto, os custos dos opcodes não podem ser simplesmente cortados pela metade ou reduzidos abaixo de certo nível. Podemos pensar em tempos de bloco mais rápidos e blocos maiores como resultando em taxas de transação mais baixas. Algumas novas cadeias de camada 1 introduziram isso, como a Solana.

**Tipos de transação**

Vamos continuar explorando os diferentes tipos de transações no blockchain Ethereum. Discutimos isso brevemente antes, na seção *componentes da transação*, no entanto, agora nos aprofundaremos e aprenderemos sobre os diferentes elementos dos três principais tipos de transações no Ethereum.

**Transações simples**

Este é o tipo padrão de transação de transferência de valor que o Ethereum suporta. Ela é usada para transferir fundos (ether) entre contas.

**Transações de criação de contrato**

Uma transação de criação de contrato é usada para criar contratos inteligentes no blockchain. Existem alguns parâmetros essenciais exigidos para uma transação de criação de contrato. Esses parâmetros estão listados a seguir:

* O remetente
* O originador da transação
* Gás disponível
* Preço do gás
* Valor destinado (endowment), que é a quantidade de ether alocada
* Um vetor de bytes de comprimento arbitrário
* Código de inicialização da EVM
* A profundidade atual da pilha de chamadas de mensagem/criação de contrato (profundidade atual significa o número de itens que já estão presentes na pilha)

O código de inicialização da EVM do campo *init* na transação contém o bytecode da EVM compilado por ferramentas como o compilador Solidity. Uma vez que o contrato é implantado, ele é adicionado como um novo item na *trie* do estado mundial. Seu bytecode é armazenado no campo *codeHash* no estado da conta. Além disso, uma nova *trie* de armazenamento é criada e sua raiz é salva no campo *StorageRoot* no estado da conta.

Endereços gerados como resultado de uma transação de criação de contrato têm 160 bits de comprimento. Exatamente como definido no *yellow paper*, eles são os 160 bits mais à direita do hash Keccak da codificação RLP da estrutura contendo apenas o remetente e o nonce. Inicialmente, o nonce na conta é definido como zero. O saldo da conta é definido com o valor passado para o contrato. O armazenamento também é definido como vazio. O *codeHash* é um hash Keccak de 256 bits de uma string vazia.

A nova conta é inicializada quando o código da EVM (o código de inicialização mencionado anteriormente) é executado. Em caso de qualquer exceção durante a execução do código, como falta de gás (OOG), o estado não muda. Se a execução for bem-sucedida, então a conta é criada após o pagamento dos custos de gás apropriados.

Desde o Ethereum Homestead, o resultado de uma transação de criação de contrato é ou um novo contrato com seu saldo ou nenhum novo contrato é criado e nenhum valor é transferido. Isso contrasta com versões anteriores ao Homestead, onde o contrato seria criado independentemente do sucesso da implantação do código do contrato ou não, mesmo em caso de exceção OOG.

**Transações de chamada de mensagem (*message call transactions*)**

O estado é alterado por meio de transações. Essas transações são criadas por fatores externos (usuários), são assinadas e então transmitidas para a rede Ethereum. Após um contrato inteligente ser implantado com sucesso, suas funções são chamadas por meio de transações chamadas de chamadas de mensagem (*message calls*). Podemos simplesmente pensar nelas como transações de execução de contrato. Uma transação de chamada de mensagem requer diversos parâmetros para sua execução, listados a seguir:

* O remetente
* O originador da transação
* O destinatário (endereço do contrato)
* A conta cujo código deve ser executado (geralmente a mesma que o destinatário)
* Gás disponível
* O valor
* O preço do gás
* Um vetor de bytes de comprimento arbitrário
* Os dados de entrada da chamada
* A profundidade atual da pilha de chamadas de mensagem/criação de contrato

As chamadas de mensagem resultam em uma transição de estado. As chamadas de mensagem também produzem dados de saída, que não são utilizados se as transações forem executadas externamente. Nos casos em que as chamadas de mensagem são acionadas por código da EVM, a saída produzida pela execução da transação é usada. Conforme definido no *yellow paper*, uma chamada de mensagem é o ato de passar uma mensagem de uma conta para outra. Se a conta de destino tiver um código da EVM associado, então a EVM será iniciada ao receber a mensagem para realizar as operações requeridas.  
Se o remetente da mensagem for um objeto autônomo (ator externo), então a chamada retorna quaisquer dados retornados da operação da EVM.

As mensagens são passadas entre contas utilizando chamadas de mensagem. A seguir, é apresentada uma descrição de mensagens e chamadas de mensagem.

**Mensagens**

Mensagens, conforme definidas no *yellow paper*, são os dados e valores que são passados entre duas contas. Uma mensagem é um pacote de dados passado entre duas contas. Este pacote de dados contém dados e um valor (a quantidade de ether). Ele pode ser enviado por meio de um contrato inteligente (objeto autônomo) ou de um ator externo (uma conta de propriedade externa, ou EOA) na forma de uma transação que foi assinada digitalmente pelo remetente.

Contratos podem enviar mensagens para outros contratos. Mensagens existem apenas no ambiente de execução e nunca são armazenadas. As mensagens são semelhantes às transações; no entanto, a principal diferença é que elas são produzidas pelos contratos, enquanto as transações são produzidas por entidades externas ao ambiente Ethereum (EOAs).

Uma mensagem consiste nos seguintes componentes:

* O remetente da mensagem
* O destinatário da mensagem
* A quantidade de Wei a ser transferida e a mensagem a ser enviada ao endereço do contrato
* Um campo de dados opcional (dados de entrada para o contrato)
* A quantidade máxima de gas (*startgas*) que pode ser consumida

As mensagens são geradas quando os opcodes **CALL** ou **DELEGATECALL** são executados pelo contrato em execução na EVM.

Uma *call* não transmite nada para o blockchain; em vez disso, é uma chamada local e é executada localmente no nó Ethereum. É quase como uma chamada de função local. Ela não consome nenhum gas, pois é uma operação somente de leitura. É semelhante a uma execução simulada ou teste. *Calls* também não permitem transferência de ether para contas de contrato (CAs). Elas são executadas localmente na EVM do nó e não resultam em nenhuma mudança de estado porque nunca são mineradas. *Calls* são processadas de forma síncrona e normalmente retornam o resultado imediatamente.

**Validação e execução de transações**

As transações são executadas após sua validade ser verificada. As verificações iniciais são listadas a seguir:

* Uma transação deve estar bem formada e codificada em RLP, sem quaisquer bytes extras no final.
* A assinatura digital usada para assinar a transação deve ser válida.
* O *nonce* da transação deve ser igual ao *nonce* atual da conta do remetente.
* O *gas limit* não deve ser inferior ao gas utilizado pela transação.
* A conta do remetente deve ter saldo suficiente para cobrir o custo da execução.

Um subestado de transação é criado durante a execução da transação e é processado imediatamente após a execução ser concluída. Este subestado de transação é uma tupla composta por quatro itens. Esses itens são:

* **Conjunto de suicídio ou conjunto de autodestruição** (*Suicide set*): Este elemento contém a lista de contas (se houver) que são eliminadas após a execução da transação.
* **Série de logs**: Esta é uma série indexada de pontos de verificação que permite o monitoramento e notificação de chamadas de contrato para entidades externas ao ambiente Ethereum, como interfaces de aplicações. Funciona como um mecanismo de gatilho que é executado toda vez que uma função específica é invocada ou um evento específico ocorre. Logs são criados em resposta a eventos que ocorrem no contrato inteligente. Também pode ser usado como uma forma mais barata de armazenamento. Eventos serão abordados com exemplos práticos no Capítulo 12, *Desenvolvimento Web3 Usando Ethereum*.

**Não confunda uma *call* com uma transação de chamada de mensagem, que de fato resulta em mudança de estado.**  
Uma *call* basicamente executa localmente as transações de chamada de mensagem no cliente e nunca consome gas nem resulta em mudança de estado. Está disponível na API JavaScript *web3.js* e pode ser vista como um modo simulado da transação de chamada de mensagem. Por outro lado, uma transação de chamada de mensagem é uma operação de escrita e é usada para invocar funções em uma conta de contrato (CA), o que consome gas e resulta em mudança de estado.

**Estado e armazenamento no blockchain Ethereum**

O estado do Ethereum é uma coleção de objetos que são compostos de contas. Cada conta possui um estado, e todas essas contas juntas compõem o estado do blockchain Ethereum. O estado de uma conta é composto por um conjunto de pares chave-valor. Cada par chave-valor é armazenado utilizando uma Merkle Patricia Trie (MPT). A raiz da MPT de estado é então armazenada no cabeçalho de bloco. O estado pode ser alterado apenas com a execução de transações.

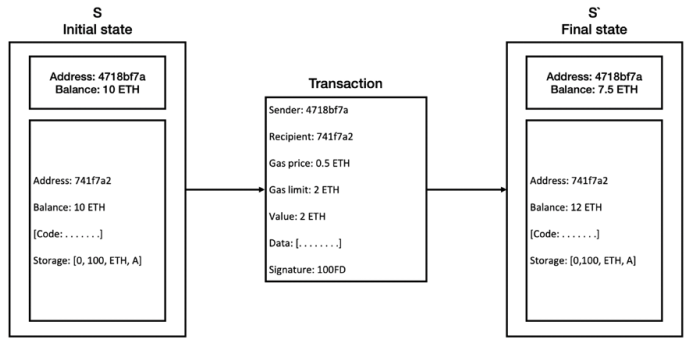
Cada conta no Ethereum possui os seguintes atributos em seu estado:

* Um **nonce**, que é o número de transações enviadas por esta conta.
* Um **saldo**, que é o número de Wei pertencente à conta.
* Um **StorageRoot**, que é o hash da raiz de uma árvore de Merkle Patricia (MPT), que codifica o conteúdo do armazenamento da conta (um armazenamento persistente e associado ao endereço).
* Um **CodeHash**, que é o hash do código EVM da conta. Para contas de propriedade externa (EOA), o código é vazio.

Todos esses atributos compõem o estado da conta. Esse estado da conta é armazenado em uma estrutura de árvore de Merkle chamada MPT. Cada nó da árvore é armazenado utilizando uma estrutura de banco de dados chamada LevelDB. Essa base de dados é uma estrutura de pares chave-valor muito eficiente, com baixo consumo de espaço e suporte à persistência. No Ethereum, ela é usada para armazenar todas as estruturas de dados relacionadas ao estado. O conteúdo dos nós da árvore MPT é armazenado no LevelDB com seus hashes como chaves.

**Armazenamento**

O armazenamento representa uma área de memória persistente alocada para cada conta de contrato. Este armazenamento é composto por uma estrutura de chave-valor que é armazenada em uma MPT. Cada contrato tem seu próprio armazenamento que é modificado por meio de transações que executam código no contrato. Uma vez que um contrato é criado, ele pode alterar seu próprio armazenamento interno. A raiz da MPT de armazenamento de uma conta é chamada de **StorageRoot**. Esta raiz é armazenada no estado da conta. A estrutura da MPT de armazenamento é mostrada no diagrama a seguir:

Conforme mostrado no diagrama, o estado da conta é armazenado em uma MPT. O campo StorageRoot dentro da conta contém o hash da raiz de outra MPT, que armazena a estrutura de chave-valor de armazenamento. Todos os dados do Ethereum são armazenados dessa forma.

Cada chave e valor armazenado em armazenamento é convertido em 32 bytes por meio de preenchimento com zero (zero padding) para a esquerda. Essas chaves e valores são então codificados usando codificação RLP. Por fim, o conjunto completo de pares chave-valor é armazenado em uma MPT. A raiz da MPT é então armazenada como **StorageRoot** no estado da conta.

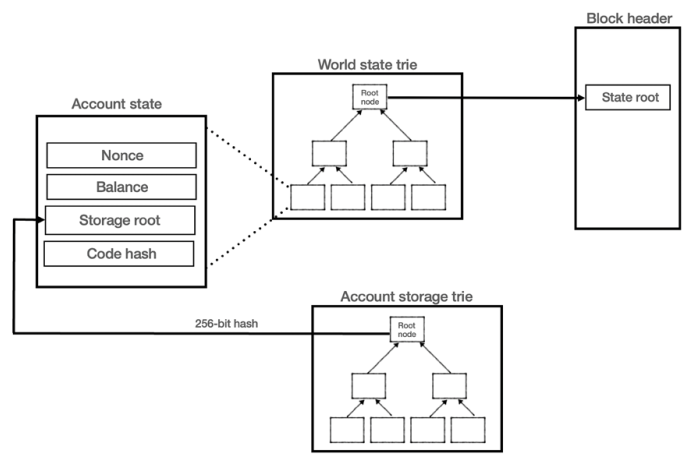
**Recibos de transação**

Um recibo de transação é um objeto no Ethereum que é criado para cada transação bem-sucedida incluída em um bloco. Os recibos de transação são usados para rastrear a execução da transação. Os recibos de transação são armazenados em uma árvore de Merkle Patricia (MPT) separada, chamada de **árvore de recibos**, cuja raiz é armazenada no campo **ReceiptsRoot** do cabeçalho do bloco.

Cada recibo de transação contém as seguintes informações:

* **Estado da execução da transação**: indica se a transação foi executada com sucesso (1) ou falhou (0).
* **Custo total de gas utilizado pela transação até este ponto no bloco**: essa é uma informação cumulativa.
* **Logs**: cada contrato pode emitir logs durante a execução, que são incluídos nos recibos.
* **Bloom filter**: uma estrutura de dados utilizada para verificar rapidamente se determinado log pode estar presente no conjunto de logs.

Como cada recibo está incluído em uma MPT, qualquer alteração em qualquer parte de um recibo mudaria o hash da raiz da MPT, o que por sua vez alteraria o hash do cabeçalho do bloco. Assim, isso fornece integridade e imutabilidade aos recibos de transação. O diagrama a seguir ilustra a estrutura de um recibo de transação:

A árvore de recibos no Ethereum é uma MPT com codificação RLP de todos os recibos incluídos no bloco. O campo **ReceiptsRoot** no cabeçalho do bloco contém o hash da raiz dessa árvore. Essa estrutura permite a verificação da inclusão de qualquer recibo usando provas de Merkle (Merkle proofs).

**Máquina virtual Ethereum**

A EVM é uma máquina de execução baseada em pilha simples que executa instruções de bytecode para transformar o estado do sistema de um estado para outro. O tamanho da palavra da EVM é definido como 256 bits. O tamanho da pilha é limitado a 1.024 elementos e é baseado em uma fila LIFO (Last In, First Out). A EVM é uma máquina Turing-completa, mas é limitada pela quantidade de gás necessária para executar qualquer instrução. Isso significa que laços infinitos que poderiam resultar em ataques de negação de serviço não são possíveis devido aos requisitos de gás.

Mais informações sobre essa mudança estão disponíveis em:  
<https://github.com/ethereum/EIPs/pull/658>

A EVM também suporta tratamento de exceções caso ocorram, como não haver gás suficiente ou fornecer instruções inválidas, caso em que a máquina será imediatamente interrompida e retornará o erro para o agente executor.

A EVM é um ambiente de execução completamente isolado e em sandbox. O código que roda na EVM não tem acesso a nenhum recurso externo, como rede ou sistema de arquivos. Isso resulta em maior segurança, execução determinística e permite que código não confiável (código que pode ser executado por qualquer pessoa) seja executado na blockchain do Ethereum.

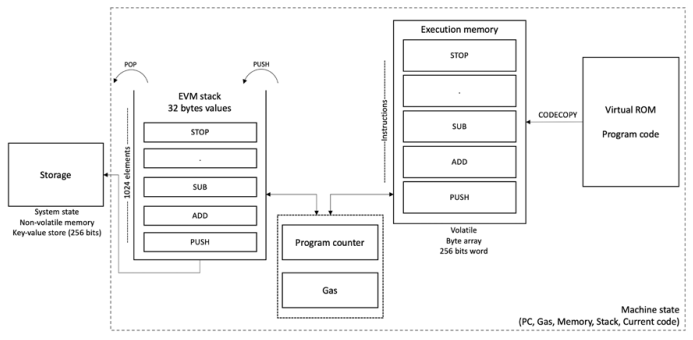
Como discutido anteriormente, a EVM é uma arquitetura baseada em pilha. A EVM é big-endian por design, e usa palavras de 256 bits. Esse tamanho de palavra permite o uso de hash Keccak de 256 bits e computações ECC.

Existem seis tipos de locais disponíveis para contratos inteligentes e para a EVM para leitura e escrita de informações:

* **Memória**: É chamada de memória volátil, uma matriz de bytes endereçada por palavra. Quando um contrato termina sua execução, a memória é limpa. É análoga à RAM. Operações de escrita podem ter 8 ou 256 bits, enquanto operações de leitura são limitadas a palavras de 256 bits. A memória é ilimitada, mas restrita pelos requisitos de taxa de gás. Está disponível apenas durante a duração da transação.
* **Armazenamento**: É uma estrutura de chave-valor e é permanentemente persistida na blockchain. As chaves e valores têm cada um 256 bits de largura. É alocada para todas as contas na blockchain. Como medida de segurança, o armazenamento é acessível apenas por seus respectivos contratos. Pode ser comparado ao armazenamento em disco rígido.
* **Pilha**: A EVM é uma máquina baseada em pilha, e assim realiza todas as computações em uma área de dados chamada pilha. Todos os valores em memória também são armazenados na pilha. Ela tem uma profundidade máxima de 1.024 elementos e suporta tamanho de palavra de 256 bits. Os opcodes da EVM retiram dados da pilha (POP) e colocam dados nela (PUSH). É um local de armazenamento temporário, que é esvaziado após a execução da transação.
* **Call data**: É o campo de dados de uma transação. Quando a transação está sendo executada, isso atua como memória somente leitura que contém os parâmetros para a chamada de mensagem.
* **Código**: Este local contém o código atualmente em execução e também é usado para armazenamento de dados estáticos.
* **Logs**: Este é o espaço de saída de evento e log somente escrita.

O armazenamento associado à EVM é uma matriz de palavras endereçável por palavra que é não-volátil e é mantida como parte do estado do sistema. Chaves e valores têm 32 bytes de tamanho. O código do programa é armazenado em memória virtual somente leitura (ROM virtual), que é acessível usando a instrução CODECOPY. A instrução CODECOPY copia o código do programa para a memória principal. Inicialmente, todo o armazenamento e memória são definidos como zero na EVM. Há outra instrução chamada EXTCODECOPY, que pode ser usada para copiar código de programa de outro contrato para a memória.

O diagrama a seguir mostra o design da EVM onde a ROM virtual armazena o código do programa que é copiado para a memória principal usando a instrução CODECOPY. A memória principal é então lida pela EVM referindo-se ao contador de programa e executa instruções passo a passo:



**Figura 9.7: Operação da EVM**

O diagrama anterior mostra uma pilha da EVM no lado esquerdo mostrando que os elementos são empilhados (push) e desempilhados (pop). Também mostra que um contador de programa é mantido e incrementado com as instruções sendo lidas da memória principal, com indexação da próxima instrução de bytecode da EVM a ser executada. Há também um registrador de gás que mantém a contagem do gás disponível. A memória principal recebe o código do programa da ROM virtual/armazenamento via instrução CODECOPY.

A otimização da EVM é uma área ativa de pesquisa, e estudos recentes sugeriram que a EVM pode ser otimizada e ajustada a um grau muito fino para alcançar alto desempenho. A pesquisa e o desenvolvimento no Ethereum WebAssembly (ewasm)—uma iteração do WebAssembly com sabor Ethereum—já estão em andamento. O WebAssembly (Wasm) foi desenvolvido pelo Google, Mozilla e Microsoft, e agora está sendo projetado como um padrão aberto pelo grupo da comunidade W3C. O objetivo do Wasm é poder executar código de máquina no navegador com velocidade nativa. Mais informações e o repositório GitHub para ewasm estão disponíveis em:  
<https://github.com/ewasm>

Outra linguagem intermediária chamada YUL, que pode compilar para vários backends como EVM e ewasm, está em desenvolvimento. Mais informações sobre essa linguagem podem ser encontradas em:  
<https://solidity.readthedocs.io/en/latest/yul.html>

O sucesso e ampla adoção do Ethereum despertaram interesse em desenvolver mais cadeias EVM. Há várias cadeias EVM disponíveis no ecossistema blockchain agora. Uma lista de redes EVM com detalhes necessários está disponível em:  
<https://chainlist.org>

**Ambiente de execução**

Existem alguns elementos-chave que são exigidos pelo ambiente de execução para executar o código. Os parâmetros principais são fornecidos pelo agente executor, por exemplo, uma transação. Estes são listados a seguir:

* O estado do sistema
* O gás restante para execução
* Subestado acumulado
* O endereço da conta que possui o código em execução
* O endereço do remetente da transação. Este é o endereço de origem desta execução (pode ser diferente do remetente)
* O preço do gás da transação que iniciou a execução
* Dados de entrada ou dados da transação dependendo do tipo de agente executor. Isto é uma matriz de bytes; no caso de uma chamada de mensagem, se o agente executor for uma transação, então os dados da transação são incluídos como dados de entrada
* O endereço da conta que iniciou a execução do código ou o remetente da transação
* O valor da transação (em Wei). Se o agente executor for uma transação, então é o valor da transação
* O código a ser executado, apresentado como uma matriz de bytes que a função iteradora seleciona em cada ciclo de execução
* O cabeçalho do bloco atual
* Profundidade—o número de chamadas de mensagem ou transações de criação de contrato (CALL, CREATE ou CREATE2) atualmente em execução
* Permissão para fazer modificações no estado

A execução resulta na produção do estado resultante, o gás restante após a execução, o conjunto de autodestruição (self-destruct), série de logs e quaisquer reembolsos de gás.

**Estado da máquina**

O estado da máquina é mantido internamente e atualizado após cada ciclo de execução da EVM. Uma função iteradora é executada na EVM, que gera os resultados de um único ciclo da máquina de estados. A função iteradora executa várias funções vitais que são usadas para definir o próximo estado da máquina e, eventualmente, o estado global. Essas funções incluem o seguinte:

* Buscar a próxima instrução de uma matriz de bytes onde o código de máquina é armazenado no ambiente de execução
* Adicionar/remover (PUSH/POP) itens da pilha adequadamente
* Reduzir o gás de acordo com o custo de gás das instruções/opcodes
* Incrementar o Contador de Programa (PC)

A EVM também é capaz de parar em condições normais se os opcodes STOP, SUICIDE ou RETURN forem encontrados durante o ciclo de execução.

O estado da máquina pode ser visto como uma tupla (g, pc, m, i, s) que consiste nos seguintes elementos:

* g: Gás disponível
* pc: O PC, que é um número inteiro positivo de até 256
* m: O conteúdo da memória (uma série de zeros de tamanho 2²⁵⁶)
* i: O número ativo de palavras na memória (contando continuamente a partir da posição 0)
* s: O conteúdo da pilha

A EVM é projetada para lidar com exceções e irá parar (interromper a execução) se ocorrerem quaisquer das seguintes exceções:

* Gás insuficiente restante para execução
* Instruções inválidas
* Itens insuficientes na pilha
* Destino inválido para opcodes de salto
* Tamanho inválido da pilha (maior que 1.024)

Com isso, concluímos nossa discussão sobre a EVM.

**Blocos e Blockchain**

Os blocos são a principal estrutura de construção de uma blockchain. Os blocos do Ethereum consistem em vários elementos, que são descritos da seguinte forma:

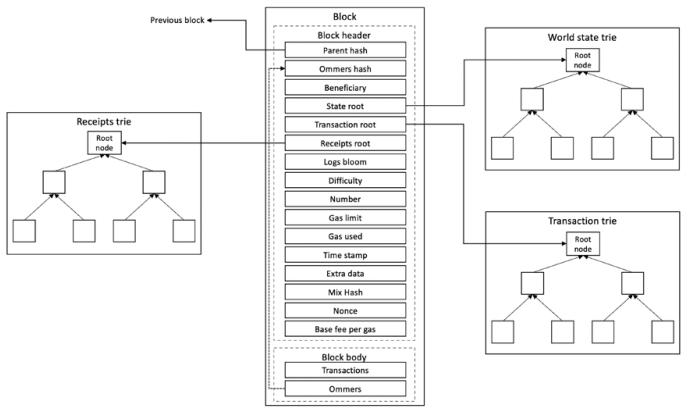
* A lista de cabeçalhos de ommers ou tios
* O cabeçalho do bloco
* A lista de transações

Um bloco tio é um bloco que é filho de um pai, mas que não possui um bloco filho. Ommers ou tios são blocos válidos, mas obsoletos, que não fazem parte da cadeia principal, mas contribuem para a segurança da cadeia. Eles também recebem uma recompensa por sua participação, mas não se tornam parte da verdade canônica.

**Cabeçalho de bloco**

Os cabeçalhos de bloco são os componentes mais críticos e detalhados de um bloco do Ethereum. O cabeçalho contém vários elementos, descritos em detalhes aqui:

* **Hash do pai (Parent hash)**: É o hash Keccak de 256 bits do cabeçalho do bloco pai (anterior)
* **Hash dos ommers (Ommers hash)**: É o hash Keccak de 256 bits da lista de blocos ommers (ou tios) incluídos no bloco
* **Beneficiário (Beneficiary)**: Contém o endereço de 160 bits do destinatário que receberá a recompensa de mineração uma vez que o bloco for minerado com sucesso
* **Raiz do estado (State root)**: Contém o hash Keccak de 256 bits do nó raiz da trie de estado. É calculada uma vez que todas as transações foram processadas e finalizadas
* **Raiz das transações (Transaction root)**: É o hash Keccak de 256 bits do nó raiz da trie de transações. Representa a lista de transações incluídas no bloco
* **Raiz dos recibos (Receipts root)**: É o hash Keccak de 256 bits do nó raiz da trie de recibos de transação. Esta trie é composta por recibos de todas as transações incluídas no bloco. Os recibos de transações são gerados após cada transação ser processada e contêm informações úteis pós-transação
* **Logs bloom**: É um filtro de bloom composto pelo endereço do registrador e tópicos de log da entrada de log de cada recibo de transação da lista incluída de transações no bloco. O sistema de logs é explicado em detalhe na próxima seção
* **Dificuldade (Difficulty)**: O nível de dificuldade do bloco atual
* **Número (Number)**: O número total de todos os blocos anteriores; o bloco gênese é o bloco zero
* **Limite de gás (Gas limit)**: Representa o limite definido no consumo de gás por bloco
* **Gás utilizado (Gas used)**: Contém o total de gás consumido pelas transações incluídas no bloco
* **Carimbo de tempo (Timestamp)**: É o tempo Unix epoch no momento da inicialização do bloco
* **Dados extras (Extra data)**: Pode ser usado para armazenar dados arbitrários relacionados ao bloco. Apenas até 32 bytes são permitidos nesse campo
* **Mixhash**: Contém um hash de 256 bits que, quando combinado com o nonce, é usado para provar que esforço computacional adequado (Proof of Work, ou PoW) foi despendido para criar esse bloco
* **Nonce**: É um hash de 64 bits (um número) que é usado em conjunto com o mixhash para provar que o PoW foi realizado corretamente
* **BaseFeePerGas**: Campo novo introduzido após a atualização London—EIP-1559—para registrar a taxa calculada pelo protocolo exigida para que uma transação seja incluída no bloco



**Figura 9.8: Diagrama detalhado da estrutura do bloco com cabeçalho e relação com tries**

A lista de transações é simplesmente uma lista de todas as transações incluídas no bloco. Também, a lista de cabeçalhos de tios é incluída no bloco.

**Bloco gênese**

O bloco gênese é o primeiro bloco em uma rede blockchain. Ele varia ligeiramente dos blocos normais devido aos dados que contém e à forma como foi criado. A principal diferença é que ele é um bloco codificado no cliente de software Ethereum, enquanto os blocos seguintes são minerados. Ele contém 15 itens que podem ser visualizados no explorador de blocos Etherscan:  
<https://etherscan.io/block/0>

**Validação, finalização e processamento de blocos**

Um bloco do Ethereum é considerado válido se passar nas seguintes verificações:

* Se for consistente com tios e transações. Isso significa que todos os ommers satisfazem a propriedade de que são realmente tios. Além disso, se o PoW dos tios for válido
* Se o bloco anterior (pai) existe e é válido
* Se o timestamp do bloco é válido. Isso significa que o timestamp do bloco atual deve ser maior do que o do bloco pai. Além disso, ele deve ser inferior a 15 minutos no futuro. Todos os tempos de bloco são calculados em tempo Unix (epoch)

Se qualquer uma dessas verificações falhar, o bloco será rejeitado. Uma lista de erros pelos quais o bloco pode ser rejeitado é apresentada aqui:

* O timestamp é mais antigo que o do pai
* Há muitos tios – mais de dois
* Há um tio duplicado
* O tio é um ancestral
* O pai do tio não é um ancestral
* Há dificuldade não positiva
* Há um digest de mix inválido
* Há um PoW inválido

A finalização de bloco é um processo executado pelos mineradores para validar o conteúdo do bloco e aplicar as recompensas. Ele resulta na execução de quatro etapas:

1. **Validação dos ommers**: No caso da mineração, determinar os ommers. O processo de validação dos cabeçalhos de blocos obsoletos verifica se o cabeçalho é válido e se a relação entre o tio e o bloco atual satisfaz a profundidade máxima de seis blocos. Um bloco pode conter no máximo dois tios
2. **Validação da transação**: No caso da mineração, determinar as transações. Isso envolve verificar se o gás total usado no bloco é igual ao consumo final de gás após a última transação, ou seja, o gás cumulativo usado pelas transações incluídas no bloco
3. **Aplicação de recompensas**: Aplicar recompensas, o que significa atualizar a conta do beneficiário com um saldo de recompensa. No Ethereum, uma recompensa também é dada a mineradores por blocos obsoletos, que é 1/32 da recompensa do bloco. Tios incluídos nos blocos também recebem 7/8 da recompensa total do bloco. A recompensa atual por bloco é 2 ethers. Ela foi reduzida de 5 para 3 com o lançamento do Byzantium e depois para 2 com o lançamento do Constantinople
4. **Validação do estado e nonce**: Verificar o estado e o nonce do bloco. No caso da mineração, calcular um estado válido e nonce do bloco

**Mecanismo de dificuldade de bloco**

A dificuldade do bloco é aumentada se o tempo entre dois blocos diminuir, enquanto é reduzida se o tempo entre blocos aumentar. Isso é necessário para manter um tempo de geração de bloco aproximadamente constante. O algoritmo de ajuste de dificuldade na versão *Homestead* do Ethereum é o seguinte:

block\_diff = parent\_diff + parent\_diff // 2048 \* max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) + int(2\*\*((block.number // 100000) - 2))

Esse algoritmo significa que, se a diferença de tempo entre o bloco pai e o bloco atual for menor que 10 segundos, a dificuldade aumenta por parent\_diff // 2048 \* 1. Se a diferença de tempo for entre 10 e 19 segundos, o nível de dificuldade permanece o mesmo. Por fim, se a diferença for de 20 segundos ou mais, o nível de dificuldade diminui. Essa redução é proporcional à diferença de tempo, de parent\_diff // 2048 \* -1 até uma redução máxima de parent\_diff // 2048 \* -99.

Na versão *Byzantium*, a fórmula de ajuste de dificuldade foi alterada para levar em conta os tios no cálculo da dificuldade. A nova fórmula é a seguinte:



bservação: // é o operador de divisão inteira.

Logo após a atualização *Istanbul*, a bomba de dificuldade foi adiada mais uma vez, sob a atualização de rede *Muir Glacier* com um hard fork:  
<https://eips.ethereum.org/EIPS/eip-2384>  
A mudança foi ativada no bloco número 9.200.000 em 2 de janeiro de 2020. Ela foi adiada novamente sob *Arrow Glacier*:  
<https://eips.ethereum.org/EIPS/eip-4345>

**Nós e mineradores**

A rede Ethereum contém diferentes tipos de nós. Alguns nós atuam apenas como carteiras, alguns são clientes leves e outros são clientes completos executando toda a blockchain. Um dos tipos mais importantes são os nós de mineração. Vamos ver o que constitui a mineração nesta seção.

As transações podem ser encontradas em pools de transações ou blocos. Nos pools de transações, elas aguardam verificação por um nó, e nos blocos, são adicionadas após verificação bem-sucedida. Quando um nó minerador começa sua operação de verificação de blocos, ele inicia com as transações com maiores taxas no pool de transações e as executa uma por uma. Quando o limite de gás é alcançado, ou não há mais transações no pool, a mineração começa. Como resultado, a moeda (ether) é concedida aos nós que realizam operações de mineração como incentivo para validar e verificar blocos compostos por transações. O processo de mineração ajuda a proteger a rede ao verificar os cálculos.

Nesse processo, o cabeçalho do bloco é repetidamente hasheado até que um nonce válido seja encontrado, de forma que, quando combinado com o cabeçalho do bloco, resulte em um valor inferior ao alvo de dificuldade. Em termos teóricos, um nó minerador executa as seguintes funções:

* Ouve as transações transmitidas na rede Ethereum e determina quais serão processadas
* Determina os blocos tios obsoletos e os inclui na blockchain
* Atualiza o saldo da conta com a recompensa recebida pela mineração bem-sucedida do bloco
* Por fim, um estado válido é calculado e o bloco é finalizado, o que define o resultado de todas as transições de estado

Uma vez que um bloco é minerado com sucesso, ele é imediatamente transmitido à rede, reivindicando sucesso, e será verificado e aceito pela rede.

Os mineradores desempenham um papel vital na obtenção de consenso quanto ao estado canônico da blockchain. O mecanismo de consenso ao qual eles contribuem é explicado na próxima seção.

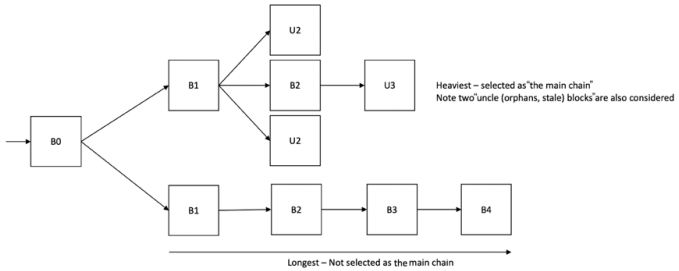
**O mecanismo de consenso**

O método original de mineração no Ethereum era baseado em Proof of Work (PoW), semelhante ao do Bitcoin. Quando um bloco é considerado válido, ele deve satisfazer não apenas os requisitos gerais de consistência, mas também conter o PoW para uma dada dificuldade. O algoritmo PoW do Ethereum é chamado de **Ethash**. O mecanismo de consenso no Ethereum é baseado no protocolo **Greedy Heaviest Observed Subtree (GHOST)**, proposto inicialmente por Zohar e Sompolinsky em dezembro de 2013.

Leitores interessados podem encontrar mais informações no artigo:  
<http://eprint.iacr.org/2013/881.pdf>

O Ethereum usa uma versão simplificada desse protocolo, onde a cadeia que tem o maior esforço computacional gasto em sua construção é identificada como a versão definitiva. Outra forma de visualizar isso é encontrar a cadeia mais longa, pois ela deve ter sido construída com esforço de mineração adequado.

O protocolo GHOST foi introduzido para mitigar os problemas decorrentes de tempos rápidos de geração de blocos que levavam à criação de blocos obsoletos ou órfãos. No GHOST, blocos obsoletos, órfãos, tios ou ommers, são incluídos nos cálculos para determinar a cadeia de blocos a ser selecionada como cadeia principal—daí o termo *cadeia mais pesada*. Isso também significa que essa é a cadeia com o maior trabalho computacional acumulado. O diagrama a seguir mostra a diferença entre as cadeias mais longas e mais pesadas:

****

**Figura 9.9: A cadeia mais longa versus a mais pesada**

O diagrama anterior mostra duas regras para descobrir qual blockchain é a versão canônica da verdade. No caso do Bitcoin (esquerda), aplica-se a regra da cadeia mais longa, o que significa que a cadeia ativa (verdadeira) é aquela com a maior quantidade de PoW. No caso do Ethereum, o conceito é semelhante, mas também inclui ommers, os blocos órfãos, que significa que também são recompensados os blocos que competiam com outros durante a mineração, mas não entraram na cadeia principal. Isso torna a cadeia *mais pesada* em vez de apenas mais longa.

O Ethash foi originalmente projetado para ser resistente a ASICs, onde encontrar um nonce exige grandes quantidades de memória. Contudo, agora já existem ASICs para Ethash também.

**Forks na blockchain**

Um *fork* ocorre quando uma blockchain se divide em duas. Isso pode ser intencional ou não intencional. Normalmente, como resultado de uma atualização importante do protocolo, é criado um *hard fork*, enquanto um *fork* não intencional pode ocorrer devido a bugs no software.

Ele também pode ser temporário, como discutido anteriormente. Em outras palavras, a cadeia mais longa e mais pesada. Esse *fork* temporário ocorre quando um bloco é criado quase ao mesmo tempo que outro e a cadeia se divide em duas, até encontrar a cadeia mais longa ou pesada para alcançar consistência eventual.

O lançamento do *Homestead* envolveu atualizações importantes de protocolo, o que resultou em um *hard fork*. O protocolo foi atualizado no bloco número **1.150.000**, resultando na migração da primeira versão do Ethereum, conhecida como *Frontier*, para a segunda versão. Outra versão é chamada *Byzantium*, que foi lançada como um *hard fork* no bloco número **4.370.000**. A atualização mais recente é o *London hard fork* sob a EIP-1559, ativada no bloco número **12.965.000**.

Um *fork* não intencional, que ocorreu em **24 de novembro de 2016 às 14:12:07 UTC**, foi causado por um bug no mecanismo de *journaling* do cliente Geth do Ethereum. Como resultado, um *fork* de rede ocorreu no bloco número **2.686.351**. Esse bug fez com que o Geth falhasse ao impedir exclusões de contas vazias no caso da exceção *out-of-gas (OOG)*. Isso não era um problema no Parity (outro cliente popular do Ethereum). Isso significa que, a partir do bloco número 2.686.351, a blockchain Ethereum foi dividida em duas: uma executando com clientes Parity e outra com Geth. Esse problema foi resolvido com o lançamento do Geth versão 1.5.3. Como resultado do ataque DAO, a blockchain Ethereum também foi bifurcada para se recuperar do ataque.

Fundamentalmente, a blockchain Ethereum é um banco de dados que existe em cada nó da rede Ethereum. Agora vamos explorar o que é a rede Ethereum.

**A rede Ethereum**

A rede Ethereum é uma rede ponto a ponto (*peer-to-peer*) na qual os nós participam para manter a blockchain e contribuir para o mecanismo de consenso. As redes podem ser divididas em três tipos, com base nos requisitos e uso: *main net*, *test nets* e *private nets*.

**Main net**

A *main net* é a rede ativa atual do Ethereum. Seu **network ID** é 1 e o **chainID** também é 1. Os IDs de rede e cadeia são usados para identificar a rede. Um explorador de blocos que mostra informações detalhadas sobre os blocos e outras métricas relevantes está disponível em:  
🔗 <https://etherscan.io>

**Redes de teste (Test nets)**

Existem duas redes principais disponíveis para teste no Ethereum. O objetivo dessas blockchains de teste é fornecer um ambiente para testar contratos inteligentes e DApps antes de serem implantados na blockchain de produção. Além disso, como redes de teste, elas também permitem experimentação e pesquisa.

* **Sepolia**: É a rede de teste principal baseada em PoW e contém todos os recursos da *main net* Ethereum.
* **Görli (ou Goerli)**: É outra rede de teste, baseada em prova de autoridade (*proof of authority*).

**Redes privadas (Private nets)**

Como o nome sugere, são redes privadas que podem ser criadas gerando um novo bloco gênese. Isso geralmente ocorre em redes blockchain privadas, nas quais um grupo privado de entidades inicia sua própria rede blockchain e a utiliza como uma blockchain com permissão (*permissioned*) ou consorcial (*consortium*).

Uma rede privada permite a criação de uma blockchain totalmente nova, geralmente em uma rede local. Isso difere das testnets ou mainnet no sentido de que ela usa seu próprio bloco gênese e ID de rede.

Na *main net*, o cliente Ethereum Geth pode descobrir *boot nodes* por padrão, pois eles estão codificados no cliente, e se conecta automaticamente. Mas em uma rede privada, o Geth precisa ser configurado especificando os parâmetros apropriados para descobrir ou ser descoberto por outros pares. Veremos como isso é feito na prática no Capítulo 10, *Ethereum na Prática*.

**Descoberta de nós e estrutura de rede**

Vamos agora cobrir alguns aspectos teóricos relacionados à descoberta de nós e entender o que realmente acontece quando desativamos a descoberta automática de nós. Para isso, primeiro veremos como um nó normalmente descobre outros nós na rede Ethereum e quais protocolos estão envolvidos.

A rede Ethereum consiste em quatro elementos ou camadas:

1. **Discovery (Descoberta)**
2. **RLPx**
3. **DEVP2P**
4. **Protocolos de subaplicação em nível de aplicação**

**Discovery protocol (Protocolo de descoberta)**

O protocolo de descoberta é responsável por descobrir outros nós na rede usando um mecanismo baseado no algoritmo de roteamento do protocolo Kademlia. Esse protocolo funciona usando **UDP**. O UDP é um protocolo de comunicação sem conexão e mais rápido que o TCP, o que o torna adequado para aplicações sensíveis ao tempo e buscas DNS.

Os IDs de rede (*network IDs*) e IDs de cadeia (*chainIDs*) são usados pelos clientes Ethereum para identificar a rede.  
Os *ChainIDs* foram introduzidos na **EIP-155** como parte do mecanismo de proteção contra repetição (replay protection).  
A EIP-155 está detalhada em:  
<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md>

Na Ethereum, há dois protocolos de descoberta chamados **DiscV4** e **DiscV5**.

* O **DiscV4** já está pronto para produção e é implementado no Ethereum
* O **DiscV5** ainda está em desenvolvimento

Para a descoberta, um novo nó Ethereum que entra na rede usa *bootnodes* codificados como ponto de entrada inicial na rede, a partir dos quais o processo de descoberta continua.

A lista desses *bootnodes* pode ser encontrada no arquivo bootnodes.go:  
<https://github.com/ethereum/go-ethereum/blob/.../params/bootnodes.go#L23>

**RLPx**

O RLPx é um protocolo de transporte baseado em TCP, responsável por permitir comunicação segura entre os nós Ethereum. Ele faz isso usando um mecanismo de criptografia assimétrica chamado **ECIES** (*Elliptic Curve Integrated Encryption Scheme*) para o handshake e troca de chaves.

Mais informações sobre o ECIES e a especificação RLPx podem ser encontradas em:  
<https://github.com/ethereum/devp2p/blob/master/rlpx.md>

**DEVP2P (ou protocolo de fio – wire protocol)**

O DEVP2P é responsável por negociar uma sessão de aplicação entre dois nós que foram descobertos e estabeleceram um canal seguro usando o RLPx. Nesse estágio, é enviada a mensagem **HELLO** entre os nós para trocar informações sobre:

* A versão do protocolo DEVP2P
* Nome do cliente
* Subprotocolos suportados
* Portas que os nós estão ouvindo

Mensagens **PING** e **PONG** são usadas para verificar a disponibilidade dos nós, e **DISCONNECT** é enviada se uma resposta não for recebida.

**Protocolos de subaplicação (capability protocols)**

É o quarto elemento da pilha de rede do Ethereum, onde diferentes subprotocolos residem. Após a descoberta e o estabelecimento de um canal seguro e a negociação da sessão, os nós trocam mensagens usando esses chamados "protocolos de capacidade" ou subprotocolos de aplicação.

Exemplos:

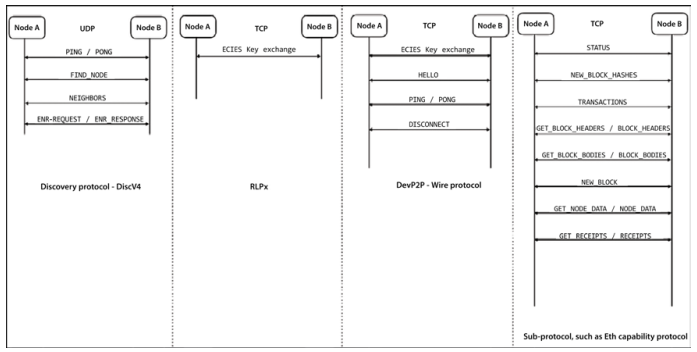
* **Eth** (versões 62, 63, 64)
* **LES** (*Light Ethereum Sub-protocol*)
* **Whisper**
* **Swarm**

Esses subprotocolos são responsáveis por comunicações de alto nível. Por exemplo, o **Eth** cuida da sincronização de blocos usando mensagens como **Status**, **Transactions**, **GetBlockHeaders**, **GetBlockBodies**, etc.

Especificações disponíveis em:

* DiscV4: <https://github.com/ethereum/devp2p/blob/master/discv4.md>
* DiscV5: <https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md>
* RLPx: <https://github.com/ethereum/devp2p/blob/master/rlpx.md>
* Eth capability: <https://github.com/ethereum/devp2p/blob/master/caps/eth.md>

⚠️ Apesar do nome, o **RLPx** não está relacionado ao protocolo de serialização RLP discutido anteriormente.

****

**Figura 9.10: Descoberta de nós Ethereum, RLPx, DEVP2P e protocolos de capacidade**

Quando configuramos redes privadas, é aconselhável desativar o mecanismo de descoberta automática de pares. Dessa forma, podemos configurar manualmente os nós que precisam se conectar. Podemos então adicionar nós manualmente à rede mantendo uma lista estática de pares.

Para criar uma rede privada, são necessários três componentes:

1. **Network ID**
2. **Arquivo gênese (genesis file)**
3. **Diretório de dados (data directory)**

**Criando uma rede privada: parâmetros e configuração**

**Network ID**

O **Network ID** pode ser qualquer número positivo, exceto aqueles já em uso por outra rede Ethereum. Por exemplo:

* 1 é usado pela mainnet
* 3 era usado pela antiga testnet Ropsten

Uma lista completa de IDs de rede pode ser encontrada em:  
<https://chainid.network>

**Arquivo gênese (genesis file)**

O **arquivo gênese** contém os campos necessários para criar um bloco gênese personalizado. Este é o primeiro bloco da rede e não aponta para nenhum bloco anterior. O protocolo Ethereum realiza verificações para garantir que nenhum outro nó na internet possa participar do consenso, a menos que tenha o mesmo bloco gênese.

O campo **chainId** normalmente é usado para identificar a rede.

**Os parâmetros mais importantes do arquivo gênese são:**

* **nonce**: Um hash de 64 bits usado para provar que o PoW foi concluído adequadamente. Funciona em combinação com mixhash
* **timestamp**: O timestamp Unix do bloco. É usado para verificar a ordem dos blocos e ajustar a dificuldade
* **parentHash**: Como este é o bloco gênese, esse valor é sempre zero
* **extraData**: Permite armazenar até 32 bytes de dados arbitrários com o bloco
* **gasLimit**: O limite de gasto de gás por bloco
* **difficulty**: Representa o nível de dificuldade para provar o PoW
* **mixhash**: Um hash de 256 bits usado junto com o nonce para provar que esforço computacional suficiente foi gasto
* **coinbase**: O endereço de 160 bits para onde a recompensa de mineração é enviada após a mineração bem-sucedida
* **alloc**: Lista de contas pré-alocadas. O valor em hexadecimal longo representa a conta que receberá saldo inicial
* **config**: Contém informações de configuração definindo o chainId e os números de blocos de hard forks. Este parâmetro não é obrigatório em redes privadas

**Diretório de dados (data directory)**

O **diretório de dados** não precisa necessariamente ser especificado, mas, se houver mais de uma blockchain ativa no sistema, deve-se usar um diretório separado para armazenar os dados da nova blockchain privada.

Esse diretório conterá os dados da blockchain da rede Ethereum privada. Diversos parâmetros podem ser especificados para executar o nó Ethereum, ajustar a configuração e iniciar a rede privada.

Se a conectividade com apenas nós específicos for desejada (como costuma ocorrer em redes privadas), então uma **lista estática de pares** deve ser fornecida como um arquivo JSON.

Esse arquivo de configuração será lido pelo cliente Geth durante a inicialização, e ele só se conectará aos nós especificados nesse arquivo.

**Contratos inteligentes pré-compilados (Precompiled smart contracts)**

Discutimos contratos inteligentes extensivamente no Capítulo 8, *Smart Contracts*. Aqui, é suficiente dizer que o Ethereum suporta o desenvolvimento de contratos inteligentes que são executados na EVM.

Além disso, existem vários contratos que estão disponíveis em formato pré-compilado na blockchain Ethereum para dar suporte a diferentes funções. Esses contratos são conhecidos como **contratos pré-compilados** ou **contratos nativos**.

Eles não são contratos inteligentes no sentido tradicional (programados pelo usuário em Solidity), mas sim **funções internas** disponíveis nativamente para suportar tarefas computacionalmente intensivas. Eles são executados no nó local e codificados no cliente Ethereum (por exemplo, Parity ou Geth).

Existem **nove contratos pré-compilados** no lançamento do Ethereum Istanbul. Abaixo, estão listados com seus endereços e finalidades:

**1. Recuperação de chave pública da curva elíptica – ECDSARECOVER**

* Endereço: 0x1
* Denotado como: ECREC
* Gás necessário: 3.000 unidades
* Retorna a chave pública a partir do hash da mensagem e da assinatura (H, V, R, S)

**Função:**

ECDSARECOVER(H, V, R, S) = Chave Pública

**2. Função hash SHA-256**

* Endereço: 0x2
* Gás: depende do tamanho da entrada
* Retorna um valor de 32 bytes

**3. Função hash RIPEMD-160**

* Endereço: 0x3
* Saída: valor de 20 bytes
* Gás: depende da entrada

**4. Função identidade / cópia de dados**

* Endereço: 0x4
* Retorna a própria entrada
* Fórmula para cálculo de gás:  
  15 + 3 \* [tamanho\_da\_entrada / 32]

**5. Exponenciação modular de grandes números**

* Endereço: 0x5
* Usado para operações como verificação de assinatura RSA

**6. Adição de ponto na curva elíptica BN256**

* Endereço: 0x6
* Implementa soma de pontos na curva elíptica, utilizada por zk-SNARKs

**7. Multiplicação escalar de ponto na curva BN256**

* Endereço: 0x7
* Implementa multiplicação de ponto (dobramentos) em curvas elípticas

**8. Emparelhamento de curva elíptica BN256**

* Endereço: 0x8
* Executa emparelhamentos bilineares, fundamentais para zk-SNARKs

**9. Função de compressão BLAKE2b “F”**

* Endereço: 0x9
* Suporte para interoperabilidade com Zcash e outras blockchains PoW baseadas em Equihash
* EIP: <https://eips.ethereum.org/EIPS/eip-152>

Todos os contratos pré-compilados poderão futuramente se tornar extensões nativas da EVM (opcodes).  
Novos contratos foram propostos na **EIP-2537** para operações na curva BLS12-381:  
<https://eips.ethereum.org/EIPS/eip-2537>

Código-fonte: todos os contratos pré-compilados estão no arquivo contracts.go do código-fonte do Ethereum:  
<https://github.com/ethereum/go-ethereum/blob/master/core/vm/contracts.go>

**Linguagens de programação**

O código para contratos inteligentes no Ethereum é escrito em linguagens de alto nível como:

* **Solidity**
* **Serpent**
* **LLL (Low-level Lisp-like Language)**
* **Vyper**

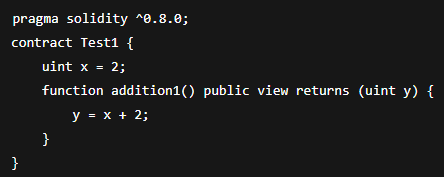
Esse código é então convertido para **bytecode** que a **EVM** entende e pode executar.

**Solidity**

A **Solidity** é uma das linguagens de alto nível desenvolvidas especificamente para o Ethereum. Sua sintaxe se assemelha à do JavaScript. Uma vez que o código é escrito, ele é compilado em bytecode pela ferramenta de compilação Solidity, chamada **solc**.

Documentação oficial:  
<http://solidity.readthedocs.io/en/latest/>

**Exemplo simples em Solidity:**

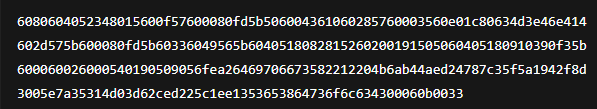


Esse programa será convertido para bytecode, mostrado a seguir.

**Bytecode de execução (Runtime bytecode)**

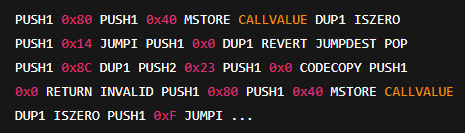
O bytecode de execução (runtime bytecode) é o que efetivamente roda na EVM.

**Representação binária do bytecode (hexadecimal bruto):**



**Opcodes**

Há muitos opcodes diferentes introduzidos na EVM. O bytecode acima é traduzido para os seguintes **opcodes**:



(continua com dezenas de outras instruções)

Lista completa e descrição dos opcodes da EVM:  
<https://ethereum.org/en/developers/docs/evm/opcodes/>

**Clientes e software de carteira**

Como o Ethereum está passando por desenvolvimento intenso e evolução constante, muitos componentes, clientes e ferramentas foram desenvolvidos e introduzidos nos últimos anos.

**Carteiras (Wallets)**

Uma **carteira** é um programa genérico que pode armazenar chaves privadas e, com base nos endereços armazenados nela, pode calcular o saldo de ether associado consultando a blockchain.

Também pode ser usada para **implantar contratos inteligentes**.

No Capítulo 10, *Ethereum na Prática*, será apresentada a carteira **MetaMask**, que se tornou a ferramenta preferida por desenvolvedores.

**Geth**

O **Geth** é a implementação oficial do cliente Ethereum escrita em Go.

Versão mais recente:  
<https://geth.ethereum.org/downloads/>

**Outros clientes**

Existem outras implementações, como:

* **Eth** (implementação em C++)
* Diversos outros, listados em:  
  <https://ethereum.org/en/developers/docs/nodes-and-clients/#execution-clients>

**Clientes leves (Light clients)**

Clientes **SPV** (*Simple Payment Verification*) baixam apenas um subconjunto da blockchain. Isso permite que **dispositivos com recursos limitados** (como celulares, dispositivos embarcados ou tablets) possam verificar transações.

Não é necessário manter a blockchain completa, e clientes SPV ainda podem **validar a execução de transações**. Eles são também chamados de **clientes leves**.

Essa ideia é semelhante aos clientes SPV do Bitcoin.

Diferença crítica entre clientes e carteiras:

* **Clientes**: implementações completas do protocolo Ethereum que suportam mineração, gerenciamento de contas e funções de carteira
* **Carteiras**: armazenam apenas as chaves públicas/privadas, oferecem funções básicas de conta e interagem com a blockchain principalmente para **transferência de fundos**

**Protocolos de suporte**

Diversos **protocolos complementares** estão disponíveis para auxiliar no ecossistema descentralizado completo. Além da camada de contratos (núcleo da blockchain), há **camadas adicionais** que precisam ser descentralizadas para atingir a **descentralização total**.

Essas camadas incluem:

* Mensagens descentralizadas
* Armazenamento descentralizado

São tratados pelos protocolos:

* **Whisper** (mensagens)
* **Swarm** (armazenamento)

**Whisper**

O **Whisper** fornece **comunicação peer-to-peer descentralizada** para a rede Ethereum.

Na essência, é um protocolo de comunicação usado por DApps para se comunicarem entre si.

* Os dados e o roteamento de mensagens são **criptografados**
* Utiliza o protocolo **DEVP2P**
* Ideal para pequenas transmissões, sem necessidade de consenso global
* Suporta **mensagens efêmeras** com tempo de vida (TTL)
* Proporciona comunicação **censura-resistente** e **anônima**

Pode ser ativado com o Geth usando o parâmetro --shh

Documentação oficial:  
<https://eth.wiki/concepts/whisper/whisper>

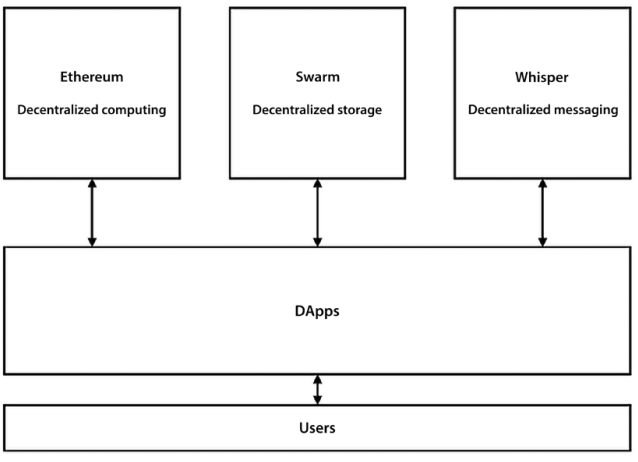
**Swarm**

O **Swarm** é uma **plataforma de armazenamento distribuído de arquivos**.

* Rede peer-to-peer de **armazenamento descentralizado**
* Os arquivos são identificados pelo **hash de seu conteúdo**
* Contrasta com serviços centralizados tradicionais
* Desenvolvido como **serviço de camada base** para o stack Web3 do Ethereum
* Integra-se ao protocolo DEVP2P
* Suporta resistência a DDoS e tolerância a falhas

Swarm utiliza o protocolo **bzz** para operações internas entre os nós.

Documentação oficial:  
<https://swarm-guide.readthedocs.io/en/latest/>

**Figura 9.11: Blockchain, Whisper e Swarm**  


Com o desenvolvimento de **Whisper** e **Swarm**, um ecossistema totalmente descentralizado emerge, onde:

* Ethereum (via EVM) → Computação
* Whisper → Comunicação
* Swarm → Armazenamento

Como mencionado no Capítulo 2 (*Decentralização*), **descentralizar todo o ecossistema** é mais desejável do que apenas a computação. O desenvolvimento desses protocolos representa um passo nessa direção.

**Conclusão**

Neste capítulo, exploramos em profundidade a arquitetura do Ethereum. Iniciamos com uma introdução à plataforma Ethereum e à sua principal característica: contratos inteligentes. Discutimos os dois tipos de contas no Ethereum — contas de propriedade externa (EOAs) e contas de contrato (CAs) — e como elas interagem por meio de transações.

Também cobrimos os componentes de uma transação Ethereum, os diferentes tipos de transações, como as transações de criação de contrato e chamadas de mensagem, bem como as estruturas de dados subjacentes usadas no Ethereum para armazenar estado e garantir integridade, como as Árvores de Merkle Patricia (MPTs) e o banco de dados LevelDB.

Além disso, examinamos como o Ethereum utiliza a codificação RLP para serializar dados, e como o mecanismo de *gas* regula e protege a execução de contratos inteligentes, atribuindo custos a cada operação da EVM.

Por fim, vimos como a Ethereum Virtual Machine (EVM) executa contratos inteligentes, gerencia memória, armazenamento e pilha, e como os contratos interagem entre si por meio de chamadas internas e criação dinâmica.

Este capítulo abordou principalmente a **arquitetura do Ethereum** e seu ecossistema.

Apresentamos os conceitos centrais da blockchain Ethereum, como:

* Modelo de **máquina de estados**
* Estados **global** e **local** da máquina
* **Contas** e **transações**

Além disso, foi apresentada uma introdução detalhada aos **componentes centrais da EVM**.

Também foram discutidos:

* A **blockchain do Ethereum**
* A **rede Ethereum**
* **Carteiras**
* **Clientes de software**
* **Protocolos de suporte** (Whisper e Swarm)