

基于以太坊平台的区块链学习

康嘉媛

院（系）：电气工程学院

专 业：电气工程及其自动化

学 号：1144210117

指导教师：Dr Shiping Chen

2019年7月

哈爾濱工業大學

毕业设计（论文）

题 目 基于以太坊平台的区块链学习

专 业 电气工程及其自动化

学 号 1144210117

学 生 康嘉媛

指导教师 Dr Shiping Chen

答辩日期 2018年11月8日

摘要

区块链范例与加密安全交易相结合，近年来通过许多项目证明了其实用性。尽管公众关注度日益增长，但我们对区块链基础协议的了解仍然非常有限。以太坊作为最值得注意的项目之一，以一种普遍的方式实施区块链技术。在本论文中，我们研究以太坊平台对区块链技术的底层实现，我们首先提供了以太坊的技术概述，研究重点基于理论研究和代码实现分析，围绕以太坊的三个核心模块展开：椭圆曲线数字签名，点对点通信，工作量证明。我们提出了一种技术，通过在以太坊网络中接收执行日志和运行信息来演示该过程，并深入了解区块链技术的底层实现。

关键词：区块链，以太坊，椭圆曲线数字签名，点对点，工作证明

Abstract

The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects in recent years. Despite the growth of the public's interests, we still have very limited knowledge of blockchain underlying protocols. Ethereum, as one the most notable projects, implements blockchain technology in a generalised manner. In this thesis, we research Ethereum platform to give an in-depth study of blockchain technology, we provide a technical overview of Ethereum, our research focus on three core modules: Elliptic Curve Digital Signature, Peer-to-peer Communication, Proof-of-work based on theory study and analysis of code implementation. We present a technique to demonstrate the process by receiving execution logs and running information in Ethereum network and gain insight into the underlying implementation of blockchain technology.

Keywords: Blockchain, Ethereum, Elliptic Curve Digital Signature, Peer-to-peer, Proof-of-Work

Table of Contents

摘要	1
Abstract	2
List of Figures	6
List of Tables	8
Glossary	9
Acronym	11
Chapter 1 Introduction	12
Chapter 2 Background	13
2.1 Blockchain History	13
2.2 Ethereum Platform	13
2.3 Design Philosophy	14
2.4 Memory and Storage	15
2.4.1 Merkle Patricia Trees	15
2.4.2 Recursive Length Prefix Encoding	16
2.4.3 The Block	17
2.5 Messages and Transactions	19
2.5.1 Transactions	19
2.5.2 Messages	19
2.6 Execution	20
Chapter 3 Core Protocols	21
3.1 Elliptic Curve Digital Signature	21
3.1.1 Elliptic Curve Point Multiplication	22
3.1.1.1 Background	22
3.1.1.2 Computation	24
3.1.2 Elliptic Curve Cryptography	25
3.1.3 Elliptic Curve Digital Signature Algorithm	25
3.1.4 Code Implementation In go-ethereum	27
3.1.4.1 Processing Digital Signatures	29
3.1.4.2 Recover the Public Key (address) From The Digital Signature	29
3.1.4.3 Digital Signature Generation	30

3.1.4.4 Public Key And Address	30
3.2 Peer-to-peer Communication	31
3.2.1 Devp2p Application Protocol	31
3.2.1.1 Low-Level	31
3.2.1.2 Message Contents	32
3.2.2 RLPx Transport Protocol	32
3.2.2.1 Node Identity	32
3.2.2.2 Encryption	32
3.2.2.3 Handshake	33
3.2.2.4 Framing	34
3.2.3 Node Discovery Protocol	34
3.2.3.1 Kademlia Network	34
3.2.3.2 Node Identity and Distance	35
3.2.3.3 Node Table	35
3.2.3.4 Wire Protocol	36
3.2.4 Code Implementation In go-ethereum	37
3.2.4.1 Node Discovery Based on UDP Implementation	37
3.2.4.2 Data Communication based on TCP Implementation	39
3.3 Proof of Work	44
3.3.1 Proof-of-Work System	44
3.3.2 Hash Function	45
3.3.3 Distributed Consensus	45
3.3.4 Proof of work and mining	47
3.3.5 Ethash Algorithm	48
3.3.5 Code Implementation In go-ethereum	50
3.3.5.1 Block Creation	51
3.3.5.2 Consensus Algorithm	56
3.3.5.3 Memory Mapping	58
3.3.5.4 Ethash Algorithm Summary	59
Chapter 4 Design	59
4.1 Private Network	59
4.1.1 Genesis file	60
4.1.2 Command Line Flags	60
4.1.3 Nodes	61
4.2 Http Server	63
4.2.1 Log Center	63

4.2.2 JSON API	63
Chapter 5 Analysis	65
5.1 Peer-to-peer network	65
5.2 Mining and Transaction	71
5.3 Proof of Work	74
Chapter 6 Conclusion and Future Work	75
Appendixes	77
Appendix A	77
Appendix B	80
Appendix C	80
Appendix D	80
Appendix E	81
Appendix F	81
Appendix G	82
Appendix H	82
Appendix I	82
Appendix J	83
Appendix K	83
Appendix L	83
Appendix M	84
Appendix N	84
Appendix O	85
Appendix P	86
Appendix Q	88
Appendix R	89
Appendix S	89
Appendix T	90
Appendix U	90
Bibliography	92
哈尔滨工业大学本科毕业设计（论文）原创性声明	95
致 谢	96

List of Figures

Figure 1. Merkle Tree	14
Figure 2. RLP mapping	15
Figure 3. Block Header	15
Figure 4. The Block	16
Figure 5. State Root	15
Figure 6. State Transition Function	17
Figure 7. Elliptic Curve	19
Figure 8. Elliptic Curve Secp256k1	22
Figure 9. Signer Interface Implementation	23
Figure 10. P2P Network Communication Overall Architecture	25
Figure 11. Binary Prefix Tree	28
Figure 12. Overall Architecture of Network Layers	30
Figure 13. Neighbour Node Discovery	31
Figure 14. TCP Connection Process	31
Figure 15. TCP Listening Process	32
Figure 16. TCP Encrypted Handshake Protocol	33
Figure 17. TCP Connecting Peers Process	34
Figure 18. Eth Protocol Process	35
Figure 19. Distributed System	36
Figure 20. Distributed System with complicated situations	36
Figure 21. The Byzantine Generals Problem	37
Figure 22. UML Diagram of miner packag	40
Figure 23. Miner Struct	40
Figure 24. process of worker.update()	41
Figure 25. process of worker.wait()	41
Figure 26. process of CpuAgent.update()	43
Figure 27. process of CpuAgent.mine()	43
Figure 28. consensus engine in Ethereum	44
Figure 29. Private Network	50
Figure 30. Execution Log	51
Figure 31. Node Inoformation	54
Figure 32. Peers Information	55
Figure 33. Node 1 Screehshot	55
Figure 34. Node 2 Screenshot	55
Figure 35. Transaction Screenshot	56

Figure 36. Pending Screenshot	56
Figure 37. Pending Screenshot After Mining	57
Figure 38. Transaction Receipt	57

List of Tables

Table.1 Etheruem UDP packet types	29
Table.2 Http Server Mapping Rules	50

Glossary

Name	Description
Hash	A hash function (or hash algorithm) is a process by which a piece of data of arbitrary size (could be anything; a piece of text, a picture, or even a list of other hashes) is processed into a small piece of data (usually 32 bytes) which looks completely random.
Encryption	Encryption is a process by which a document (plain text) is combined with a shorter string of data, called a key to produce an output (ciphertext) which can be "decrypted" back into the original plaintext by someone else who has the key, but which is incomprehensible and computationally infeasible to decrypt for anyone who does not have the key.
Public key encryption	A special kind of encryption where there is a process for generating two keys at the same time (typically called a private key and a public key).
Digital signature	A digital signing algorithm is a process by which a user can produce a short string of data called a "signature" of a document using a private key such that anyone with the corresponding public key.
Address	An address is essentially the representation of a public key belonging to a particular user.
Transaction	A transaction is a digitally signed message authorizing some particular action associated with the blockchain.
Block	A block is a package of data that contains zero or more transactions, the hash of the previous block ("parent"), and optionally other data.
State	The set of data that a blockchain network strictly needs to keep track of, and that represents data currently relevant to applications on the chain.
Account	An account is an object in the state.
Proof of work	One important property of a block in Bitcoin, Ethereum and many other crypto-ledgers is that the hash of the block must be smaller

	than some target value.
Mining	mining is the process of repeatedly aggregating transactions, constructing a block and trying different nonces until a nonce is found that satisfies the proof of work condition.

Acronym

Abbreviations	Description
P2P	Peer-to-peer
PoW	Proof-of-Work
PoA	Proof-of-Authority
PoS	Proof-of-Stack
API	Application Program Interface
JSON	JavaScript Object Notation
RPC	Remote Procedure Call

Chapter 1 Introduction

As the internet connections being ubiquitous around the world, global information transmission has become incredibly frugal. This brings about technology-rooted movements like Bitcoin in 2009, which have demonstrated the power of the default, consensus mechanisms and built a decentralised value-transfer system based on cryptographic proof.

With Bitcoin becoming booming in recent years, the underlying core technology of Bitcoin, blockchain, has attracted more and more attention from the world as well. With features of decentralisation, transparency and immutability, it is believed that blockchain technology has the potential to change the future, as there have already been many new products in global supply chains, financial transactions, asset ledgers and decentralised social networking based on blockchain technology.

Despite the significant breakthrough that blockchain technology has made, it is still complicated for most people to truly understand the underlying mechanism of blockchain in a peer-to-peer network since it implements cryptographic techniques to establish PoW protocol based on very tough math problems. Besides, there are many communication specification protocols between nodes hidden from client-side, and the specific standards of protocols vary with different blockchain-based cryptocurrencies. Ethereum, announced in 2014 and launched in 2015, is the second largest blockchain-based internet service platform. Ethereum aims to provide the end-developer with a more tightly integrated and compilable end-to-end system for building the decentralised applications with smart contracts on the blockchain. It is considered as a competitive alternative for Bitcoin in the future.

This thesis focuses on Ethereum as the most mature open blockchain platform. We aim to give an in-depth study of blockchain underlying protocols including cryptographic primitives, peer-to-peer network communication protocols and proof-of-work protocols. Based on the theoretical study, we present a technique to demonstrate the process of underlying peer-to-peer communication and proof-of-work protocol. We set up a private network in Ethereum and receive the most important execution logs and running information, which can be accessed from a local HTTP server, we analyse and summarize the results.

Chapter 2 Background

2.1 Blockchain History

In the past, almost all the online transactions needed to process electronic payments via financial institutions serving as the trusted third parties. This works well enough for most cases, while it still has trust issues considering the inherent weaknesses of this model. That is there is still a possibility that transactions can be tampered maliciously and it is impossible for financial institutions to avoid mediating disputes, which increases transaction costs as well. In real life, these costs and payment uncertainties can be avoided in person by using physical currency face to face, while there is no existing system to allow transactions happen in a person-to-person way without a trusted party.

This problem brings the concept of decentralised digital currency, which is an electronic payment system based on cryptographic proof instead of human trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. It is noted that this concept has been around for decades since the 1980s, some anonymous e-cash protocols have already implemented cryptographic primitive such as Chaumian blinding to provide a currency with high confidentiality, but they all finally failed due to reliance on a centralized intermediary. It was until 2009 that a decentralized currency, Bitcoin, was for the first time implemented in practice by Satoshi Nakamoto.

The blockchain is the technological basis of Bitcoin, it could be considered as a distributed ledger where each network node processes and stores the same transactions grouped in blocks. It is designed to add only one block at a time, and every block contains a mathematical proof to be verified if it follows in sequence from all the previous block, nodes are encouraged to maintain and verify the network by mathematically enforced economic incentives coded into the protocol.

2.2 Ethereum Platform

As introduced above, in Bitcoin's case, the blockchain technology is designed as a distributed ledger to facilitate trustless finance between individuals. However, with blockchain technology attracting greater attention from developers and technologists, novel projects began to use the blockchain for purposes other than transfers of value tokens.

One of these is Ethereum, Ethereum is proposed to be a decentralized and programmable blockchain platform for building decentralized applications. Rather than giving users a set of pre-defined operations like in Bitcoin transactions, the intent of Ethereum is to allow users to create their own operations of any complexity they wish with a suite of protocols that define a platform for decentralised applications. The heart of these protocols is called the Ethereum Virtual Machine (“EVM”), which can execute code of arbitrary algorithmic complexity. In this way, developers can write smart contracts and decentralized applications that run on the EVM with their own arbitrary rules for ownership, transaction formats and state transition functions.

As mentioned above, all the transactions are stored in the distributed block-grouped databases. This database is maintained and updated by many nodes connected to the network, which means every node in the Ethereum network runs the EVM and executes the same instructions. This is why Ethereum is also called “World Computer”. However, the entire Ethereum network has still been inefficient so far due to the massive parallelisation of computing, which means computation on Ethereum is far slower and more expensive than on a traditional “computer”. In addition, since every Ethereum node runs the EVM in order to maintain consensus across the blockchain, Ethereum itself would have extreme levels of fault tolerance, ensure zero downtime, and make data stored on the blockchain forever unchangeable and censorship-resistant.

2.3 Design Philosophy

As proposed by **Vitalik Buterin** in **2013**, The design philosophy behind Ethereum is intended to follow the following principles:

Simplicity, that is to say, the Ethereum protocol should be simple enough despite the cost of data storage and time inefficiency. In this way, even an average programmer could implement the Ethereum application layer to realize the unprecedented democratizing potential from the further version of Ethereum.

Universality, instead of having more features, Ethereum intends to provide an internal scripting language for constructing any smart contract or transaction type to be mathematically defined. This ensures users can define their own financial derivative and more than that.

Modularity, **Vitalik Buterin(2013)** suggested that the parts of the Ethereum protocol should be designed to be as modular and separable as possible. This is to improve the reusability of code so that even other protocols other than Ethereum can use these separate, feature-complete libraries from Ethereum as well, which would contribute to the development of the entire cryptocurrency ecosystem.

Agility, there is still a lot to improve in Ethereum protocol with computational tests later on in the development process, like the scalability or security of the protocol

architecture and the Ethereum Virtual Machine (EVM), which means the exploitation would still be needed.

2.4 Memory and Storage

Ethereum platform is proposed as the “world computer”, as each node of the network runs on the EVM and executes the same instructions. To achieve this distributed memory and storage, there is a so-called “world state” for every node. In this case, the world state is divided by blocks and each new block is representing a new world state.

The block is then mapped into RLP(e recursive length prefix standard) format to store the information in a Merkle-Patricia-Tree-like database. (**Micah Dameron,2018**)

2.4.1 Merkle Patricia Trees

According to **Satoshi Nakamoto(2009)**, Bitcoin implements a multi-level data structure to save disk space, that is to store the block hash with only the root included in the block's hash. In this way, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle Tree storing all transactions in the block. Based on Merkle Tree, **Micah Dameron(2018)** presented Merkle Patricia Trees to represent individual characters from hashes instead of each node representing an entire hash. Instead of only representing the intrinsically correct paths in the data, the state data structure will give the requisite cryptographic proofs that a piece of data was valid in the first place. **Micah Dameron(2018)** considered that such a Merkle Patricia Tree could verify the Blockchain by combining the structure of a standard Merkle Tree with the structure of a Radix Tree.

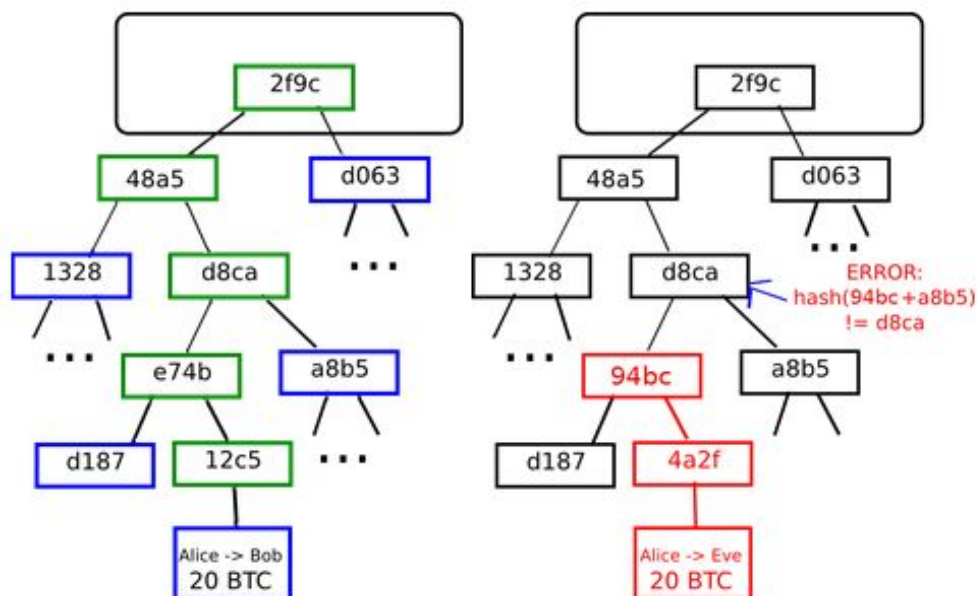


Fig.1 Merkle Tree (Vitalik Buterin,2014)

2.4.2 Recursive Length Prefix Encoding

Recursive Length Prefix Encoding (RLP) is a serialisation method for encoding arbitrarily structured binary data (byte arrays)(**DR. GAVIN WOOD FOUNDER, ETHEREUM & ETHCORE, 2018**). It imposes a structure on data that takes a prefixed hex value to index the data in the state database tree. In this way, the depth of a certain piece of data is determined by the hex value (**Micah Dameron,2018**). RLP is the main data serialization method in Ethereum, which only encodes the structure of data and does not pay heed to the particular types of data being encoded. In the world state database, RLP is implemented to encode each world state for fast traversal and verification of data, as shown in figure 2 below, RLP encoding creates a mapping between account states and world state. Since it is stored on the node operator's computers, the tree can be indexed and searched without network delay.

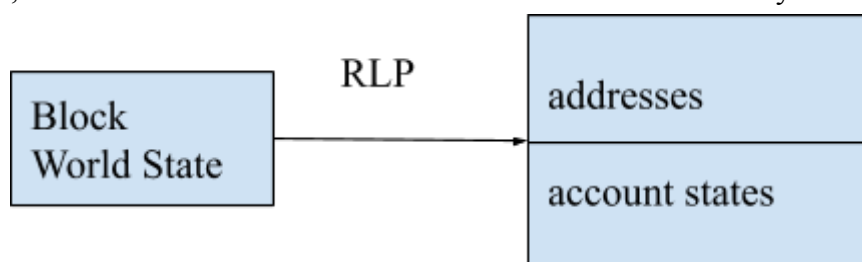


Fig.2 RLP mapping

2.4.3 The Block

Another core part of Ethereum in terms of memory and storage is the block. As **DR. GAVIN WOOD FOUNDER, ETHEREUM & ETHCORE(2018)** explained, the block in Ethereum is the collection of relevant pieces of information (known as the block header), H, together with information corresponding to the comprised transactions, T, and a set of other block headers U that are known to have a parent equal to the present block's parent's parent (such blocks are known as ommers²), the following table shows what information the block header contains:

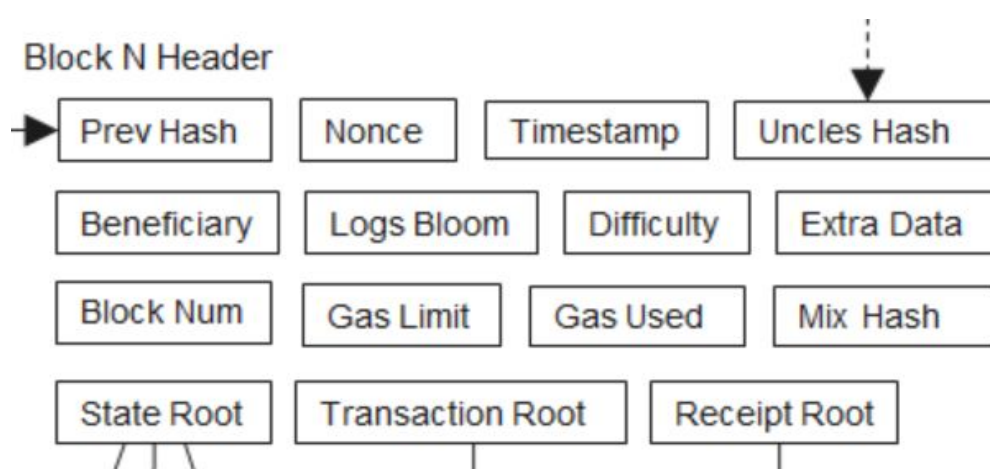


Fig.3 Block Header

The other two components in the block are simply a list of ommer block headers (of the same format as above) and a series of the transactions which is also called block footer. The whole structure is shown in figure 4 below.

Block Header	Parent Hash

	Nonce
Ommer Block Header	Parent Hash

	Nonce
Block Footer	A series of the transactions

Fig.4 The Block

As shown above, Ethereum, unlike Bitcoin, has the property that every block contains something called the “state root”: the root hash of a specialized kind of Merkle tree which stores the entire state of the system(Vitalik Buterin,2015).

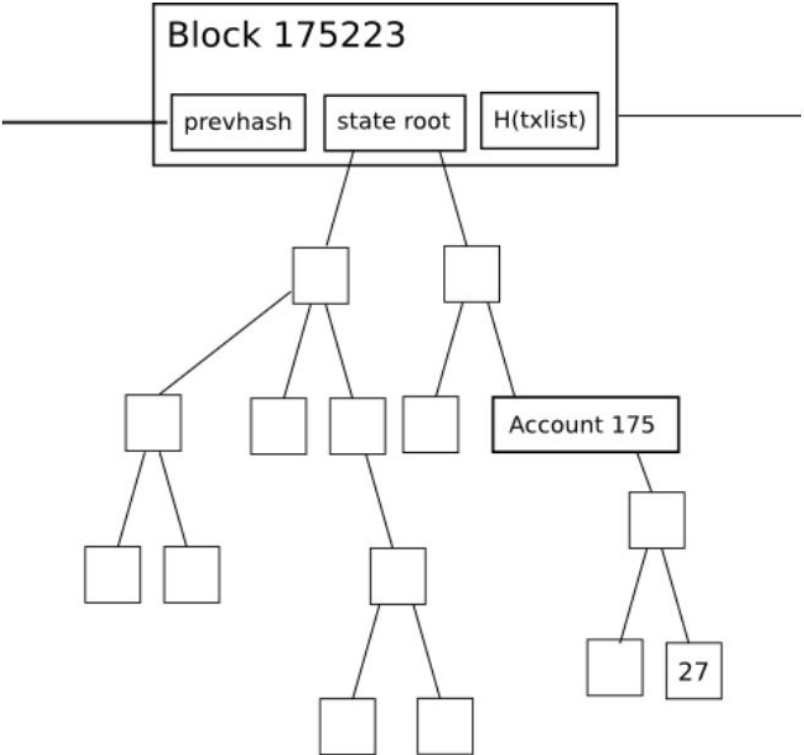


Fig.5 State Root(Vitalik Buterin,2014)

2.5 Messages and Transactions

2.5.1 Transactions

According to **Vitalik Buterin(2013)**, the term "transaction" used in Ethereum is to refer to the signed data package that stores a message to be sent from an externally owned account. There are two types of transactions: Message Calls and Contract Creations. Transactions play a key role in the Ethereum platform since each transaction applies the execution changes to the machine state, a temporary state which consists of all the required changes in computation that must be made before a block is finalized and added to the world state (**Micah Dameron,2018**).

A transaction contains the following parts:

- The recipient of the message
- A signature identifying the sender
- Value -The number of Wei to be transferred to the recipient of a message call
- An optional data field
- Gas Limit – The maximum amount of gas to be used while executing a transaction.
- Gas Price – The number of Wei to pay the network for a unit of gas.

It is noted that Gas Limit and Gas Price fields above are very crucial to prevent Ethereum from accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use (**Vitalik Buterin,2013**). This makes Ethereum an anti-denial of service model. Thus all programmable computation in Ethereum is subject to fees, the fee schedule is specified in units of “gas” so that given any type of transactions would have a universally agreed cost in terms of gas. The transaction would be considered invalid if the Gas Limit is less than Gas Price.

2.5.2 Messages

In Ethereum, Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment. A message contains:

- The sender of the message (implicit)
- The recipient of the message
- The amount of ether to transfer alongside the message
- An optional data field

- Gas Limit

Vitalik Buterin (2013) pointed out that a message is like a transaction, but it is created by a contract instead of external users. This message could then lead to the recipient account running its code. In this way, contracts can interact with other contracts in exactly the same way that external users can.

2.6 Execution

The execution of a transaction defines the state transition function in Ethereum. It can be defined as follows:

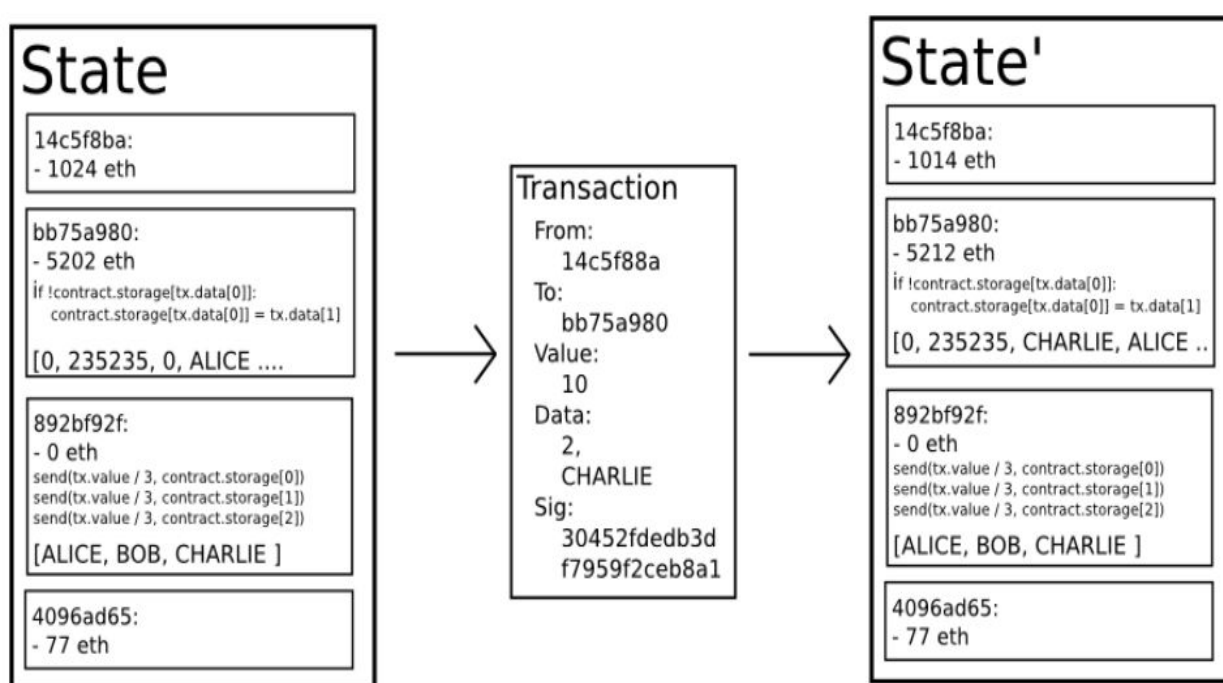


Fig.6 State Transition Function(Vitalik Buterin,2014)

1. Check if the transaction is well-formed, the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee.
3. Initialize Gas Limit, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

Before any transaction can be executed it needs to go through the initial tests of intrinsic validity(**Micah Dameron,2018**). During the execution, a certain amount of gas will be used and the accrued log items belonging to the transaction will be stored, the result of this transaction's execution is stored in the transaction receipt. As a result, the transaction receipt is a record of any given execution. **Micah Dameron (2018)** indicated that a valid transaction execution begins with a permanent change to the state, which means the nonce of the sender account is increased by one and the balance is decreased by the amount of gas a transaction is required to pay prior to its execution.

After a transaction is executed, there would be a pre-final state. As explained by **Micah Dameron (2018)**, Code execution always depletes gas. If gas runs out, an out-of-gas error is signaled (oog) and the resulting state defines itself as an empty set; it has no effect on the world state. This describes the transactional nature of Ethereum. In order to affect the world state, a transaction must go through completely or not at all.

Chapter 3 Core Protocols

3.1 Elliptic Curve Digital Signature

We attempt to first concisely discuss the widely used encryption scheme **ECDSA– Elliptic Curve Digital Signature Algorithm** – with the secp256k1 curve as it is currently implemented for all cryptographic operations in Ethereum. Hence it is necessary to introduce **Elliptic Curve Digital Signature Algorithm** first before we explain the other core protocols in the Ethereum.

There are at least three uses for **ECDSA** in Ethereum currently known:

1. Digitally sign the entire transaction object when generating each transaction object.
2. Digitally sign the newly created block in the implementation of the consensus algorithm.

Based on the use of **ECDSA** for digital signature, Ethereum also implements the public key used in the digital signature procedure as the address (ie the unique identifier of the account) in many use cases to achieve its own business requirements.

The relationship between several theoretical concepts of **ECDSA** is such that the elliptic curve digital encryption algorithm **ECDSA** is one type of the **Digital Signature Algorithm (DSA)**, which is based on **Elliptic-curve cryptography (ECC)**. While the theoretical premise of **ECC** is developed from the **elliptic curve point multiplication** of the elliptic curve. This section will start with the principles of these basic concepts, and explain in detail the Ethereum's implementation of **ECDSA**.

3.1.1 Elliptic Curve Point Multiplication

3.1.1.1 Background

The point multiplication of an elliptic curve refers to the operation in which a point on an elliptic curve is repeatedly added to itself along the curve. Here the term 'point multiplication' is the multiplication of a scalar and a point, which belongs to a scalar multiplication operation [9].

Suppose we have a starting point P on the elliptic curve and an end point R on the curve. Then we have the following point multiplication formula:

$$R = nP \quad (1)$$

The scalar is written to the left of the point. An elliptic curve E is always defined in the Cartesian coordinate system by an equation of the form:

$$y^2 = x^3 - ax + b \quad (2)$$

Where a and b are common scalar parameters. The geometric curve plot by the above equation is shown in the figure below, where the red curve represents the elliptic curve at $(a, b) = (-7, 6)$ and the blue curve represents $(a, b) = (-5, 6)$ Elliptic curve:

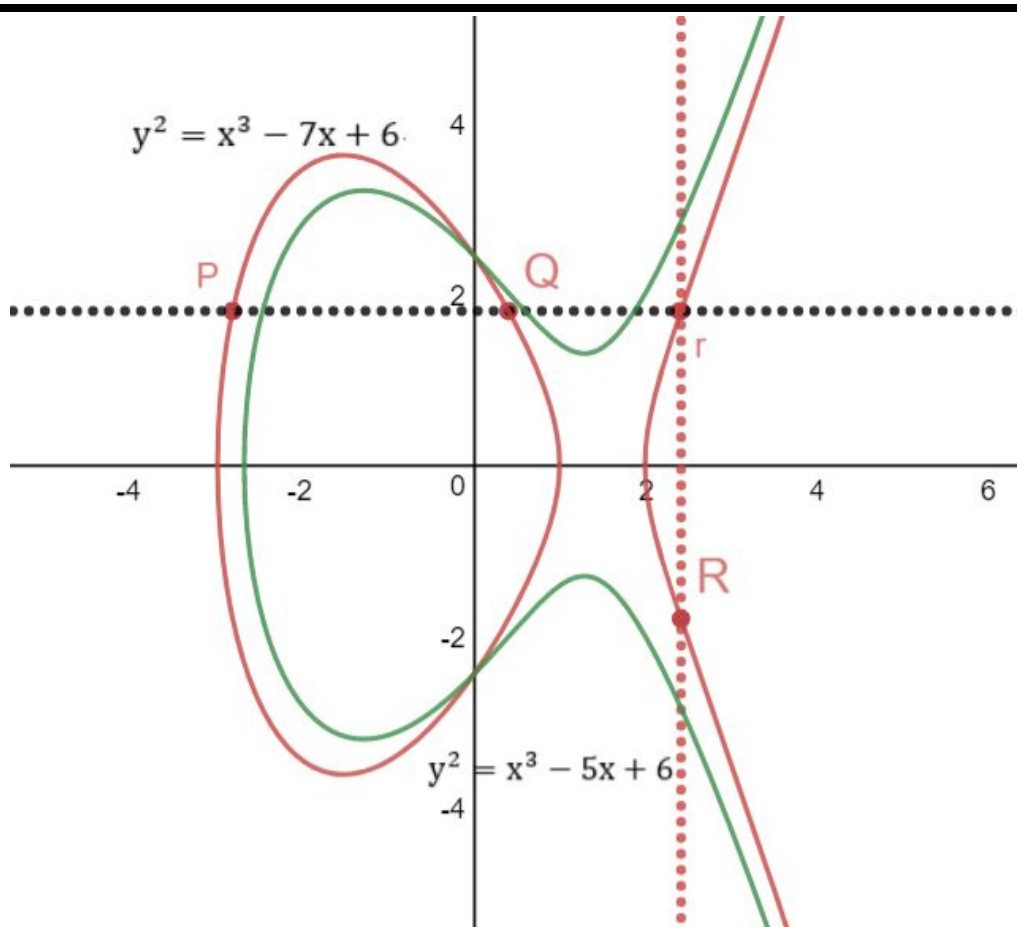


Fig.7 Elliptic Curve

It is shown that the values of a and b still have an effect on the shape of the curve. Now that we have the shape and equation of the elliptic curve, assuming that there is a point P on the curve mentioned above, we want to calculate its multiple product nP .

To do that, we still need to know more about elliptic curve operations first. One is that Point multiplication is achieved by two basic elliptic curve operations, point addition and point doubling. **Point addition** is defined to add two points to obtain another point. For example, there are two points P and Q on the red elliptic curve from the figure above. Adding these two points obtains the R point that is also on the curve. As the figure shows, the R point results from the conjugate point of the r point, where r point is derived from the intersection of the curve and the extension line of P, Q . This means r point is also the symmetry point of R point along the X -axis. Since the elliptic curve has such property that itself must be symmetrical along the X -axis, thus this R point must also be on the curve.

This can be represented in the algebra form as following:

$$P + Q = R \quad (3)$$

$$(x_p, y_p) + (x_q, y_q) = (x_r, y_r) \quad (4)$$

Then, we can use the Cartesian coordinate to indicate the P point, Q point, R point correspondingly, with the set elliptic curve formula $y^2 = x^3 + ax + b$, we can get the following:

$$x_r = \lambda^2 - x_p - x_q \quad (5)$$

$$y_r = \lambda(x_p - x_r) - y_p \quad (6)$$

$$\lambda = \frac{y_q - y_p}{x_q - x_p} \quad (7)$$

Hence, by introducing a parameter **lambda**, we can get the coordinates of P and Q to get the coordinates of the R point [9].

Let's take another step forward. If P and Q are coincident(at the same coordinates), and what is the addition of R? This is called **point doubling**. In this case, point doubling is the addition of a point P(or Q) on the elliptic curve to itself to obtain another point R on the same elliptic curve, this equals to double a point P(or Q) to get R, i.e. to find $R = 2P$. The addition is similar except that there is no well-defined straight line through P and Q, as P and Q are at the same point, hence the **lambda** mentioned above would become **the tangent** to the curve at P and Q using limiting case. As calculated as above, the corresponding x and y coordinates of the R point stays unchanged. The **lambda** is calculated as [9]:

$$\lambda = \frac{3x_p^2 + a}{2y_p} \quad (8)$$

where a is from the defining equation of the curve above.

4.1.1.2 Computation

Now we can take a look at how to compute the elliptic curve point multiplication $Q = dP$, that is, with known elliptic curve point P and the scalar d , to calculate the curve point Q . The simplest way is to combine these two operations, which is called the double-and-add method [9].

First, the scalar d needs to be represented in binary in order to apply the "point addition" and "point double" methods:

$$d = d_0 + 2d_1 + 4d_2 + \cdots + 2^m d_m \quad (9)$$

The formula above is a binary representation of d . In the code, each coefficient is represented as an array d of length $(m+1)$, and $d[i]$ corresponds to the value 0 or 1 of the i -th bit.

3.1.2 Elliptic Curve Cryptography

Elliptic-curve cryptography (ECC), like several other cryptographic types currently in use, is also an approach to implementing public-key cryptography for encryption-related operations. It operates on a specific elliptic curve over a finite field, the most important of which is the point multiplication of the elliptic curve. The elliptic curve point multiplication is the cornerstone of elliptic curve cryptography. This is because for the calculation of the point multiplication $Q = nP$, in the case of knowing the starting point P and the ending point Q , it is almost impossible to calculate n under the current calculation conditions.

This mathematical proof process is very complicated, so here we would just give an extreme example. Looking back at the figure in the previous chapter, if the red elliptic curve in the figure is used for the point multiplication operation, it is noticed that the left part of it is a closed irregular arc, if the end point Q of the double-and-add operation happens to fall on the left part, then the parameter n cannot be calculated forever, as Q will keep looping on the arc for more cycles with increasing n .

Parameters of Elliptic-curve cryptography

ECC can be used in many use cases including digital signatures, secure pseudo-random number generation and more. The application of elliptic curve is associated with a complete set of parameters called domain parameters. In general, this set of parameters defined by ECC can be expressed as

$$D = (q, a, b, G, n, h) \quad (10)$$

Where q is a maximal prime number used to represent the range of all points of the curve; a, b are the coefficients in the elliptic curve equation $y^2 = x^3 + ax + b$; G is the base point of the point multiplication of the elliptic curve, G can be regarded as their generator for all the points on the curve obtained by the point multiplication operation; n is the multiplicative order of the base point G , which is defined as the smallest integer n when the point multiplier nG is not existing; h is an integer constant that is related to the set of points in the elliptic curve operation and n , which is generally 1.

Next, we can see the application of these elliptic curve parameters in elliptic curve digital signatures.

3.1.3 Elliptic Curve Digital Signature Algorithm

As mentioned before, the Elliptic curve digital signature algorithm (ECDSA) is a kind of digital signature algorithm (DSA), which is based on elliptic curve

cryptography. Compared with RSA cryptography based DSA, ECDSA can greatly reduce the length of public key required for digital signature calculation. For example, for a digital signature with a security level of 80 bits, the public key length required by ECDSA is only 2 times that of security level, i.e. 160 bits, while the public key length required by RSA under the same security level is at least 1024 bits. At the same time, the signature length generated by the algorithm, whether ECDSA or RSA, is about 320 bits, which makes ECDSA has more advantage than RSA.

ECDSA has three phases, key generation, signature generation, and signature verification [8].

ECDSA Key Generation

Suppose Alice wants to send a signed message to Bob. Alice needs to generate her key pair from domain parameters $D = (q, a, b, G, n, h)$ first. Then Alice does as follows [8]:

1. Select a random integer d in the interval $[1, n-1]$.
2. Compute $Q = dP$.

Now Alice's public key is Q , her private key is d .

ECDSA Signature Generation

To sign the message, Alice does as follows:

1. Calculate $e = HASH(m)$, $HASH$ is a hash encryption function, such as SHA-126, or SHA-256.
2. Calculate z , the leftmost (the highest bit) L_n bits from the binary form of e , and L_n is the binary length of the multiplicative order n in the above elliptic curve parameter. The z may be greater than n , but the length will never be longer than n .
3. Select a random or pseudorandom integer k in the interval $[1, n-1]$.
4. Calculate the point on an elliptic curve:

$$(x_1, y_1) = k \times G \quad (11)$$

5. Calculate r from the following formula. If $r == 0$, return step 3 and recalculate

$$r = x_1 n \quad (12)$$

6. Calculate the s value from the following formula. If $s == 0$, return step 3 and recalculate

$$s = k^{-1} (z + rd_A) n \quad (13)$$

7. The generated digital signature is (r, s)

ECDSA Signature Verification

To verify Alice's signature (r, s) on m , Bob obtains an authenticated copy of Alice's domain parameters $D = (q, a, b, G, n, h)$ and public key Q and does the following:

1. Verify that both r and s are integers in the range $[1, n-1]$; otherwise, verification fails
2. Calculate $e = \text{HASH}(n)$, $\text{HASH}()$ is the hash function used in step 1 of the signature generation process.
3. Calculate z , the leftmost L_n bits from e
4. Calculate the parameter w :

$$w = s^{-1}n \quad (14)$$

5. Calculate two parameters u_1 and u_2 :

$$u_1 = zw \bmod n \quad (15)$$

$$u_2 = rw \bmod n \quad (16)$$

6. Calculate (x_1, y_1) , if (x_1, y_1) is not a point on an elliptic curve, then the validation fails:

$$(x_1, y_1) = u \times G + u_2 x Q_A \quad (17)$$

7. If the following equation does not hold, the verification fails:

$$r = (x_1 n) \quad (18)$$

3.1.4 Code Implementation In go-ethereum

The actual implementation of the ECDSA function in go-ethereum comes from the third-party library libsecp256k1, which is a C++ library that is used in the bitcoin code (github_bitcoin) and is considered as an optimized one for the elliptic curve secp256k1. Secp256k1 is designed as a specific set of elliptic curve digital signature parameters, including curve equations and a series of parameters required for signature operations and so on, secp256k1 is first applied in bitcoin, where the curve equation specified is $y^2 = x^3 + 7$, and its shape is shown below.

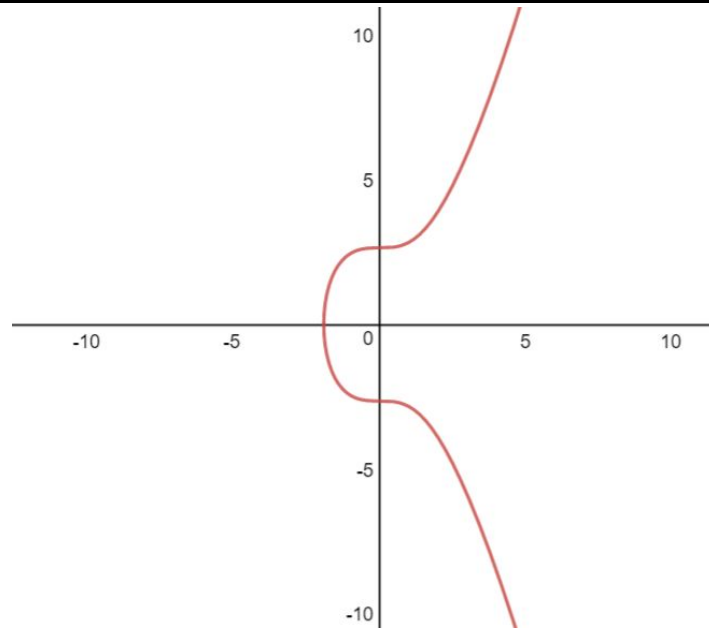


Fig.8 Elliptic Curve Secp256k1

In the go-ethereum source code, the code package under path `/crypto/` is responsible for all encryption-related operations. The source code of the `libsecp256k1` library is also stored in the subpath of `/secp256k1/`, which is to be compiled with C++ library files. The package directory is shown below:

```
crypto/
  bn256/
  ecies/
  randentropy/
  secp256k1/
  sha3/
  crypto.go
  crypto_test.go
  signature_cgo.go
  signature_nocgo.go
  signature_test.go
```

3.1.4.1 Processing Digital Signatures

Here we take the transaction object in go-ethereum as an example, the operations related to the ECDSA signature are placed in an interface called `Signer` and the interface implementation.

`/core/types/transaction_signing.go`

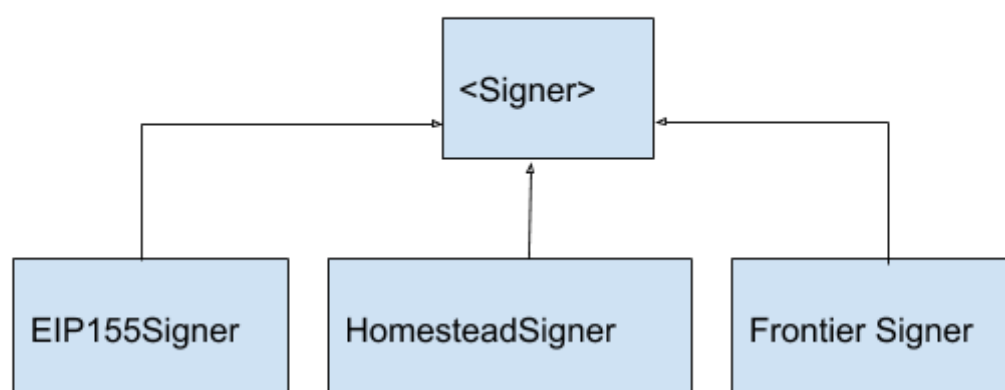


Fig.9 Signer Interface Implementation

In the method of interface `<Signer>`, `Sender()` is used to parse the public key from the digital signature carried by the Transaction object and convert it into an Address type variable; `SignatureValues()` takes the three parts of the digital signature from the tx(Transaction) object R, S, V; `Hash()` returns the content of current Transaction object which needs to be digitally signed, that is, to take the hash value of some variables in the Transaction object which are RLP encoded; `Equal()` is used to compare the object in Signer implementation. It can be seen that the `<Signer>` interface and its implementation mainly provide a method for operating the generated digital signature. See the related code in Appendix A

3.1.4.2 Recover the Public Key (address) From The Digital Signature

The function to recover (parse) the address variable from the digital signature is called `recoverPlain()`(see Appendix A)

In the function `recoverPlain()`, `crypto.ValidateSignatureValues()` is first called to verify that the digital signature is valid. The crypto package is calling the API of the libsecp256k1 library and implementing the digital signature verification part of the ECDSA algorithm theory. Next, R, S, V is jointed to the desired digital signature string. Next, `crypto.Ecrecover()` is called to recover the public key used for the digital signature with the digitally signed content 'sighash' and the signature

string 'sig'. The crypto package method still implements the libsecp256k1 library API to complete this procedure.

3.1.4.3 Digital Signature Generation

The function that generates a digital signature for a Transaction object is called `SignTx()` (see Appendix A)

As shown in the `SignTx()` function body, after the `Signer.Hash()` method provides the content to be signed (that is, the RLP-encoded hash of some members of the Transaction object), the main work of generating the signature is given to the `crypto.Sign()` function to be done. The function body of `Sign()` is shown in Appendix A

It can be seen that the `crypto.Sign()` function completes the generation of the elliptic curve digital signature by calling the API of the libsecp256k1 library.

3.1.4.4 Public Key And Address

Address type used in Ethereum, such as the address of each account, are derived from the public key used for elliptic curve digital signatures. In digital signatures, the public key can be reused in multiple signatures, which is indicated in the Ethereum account. Each Ethereum account has multiple transactions under one account, that is, multiple different Transaction objects, all of which use the same public key for digital signature.

In terms of the variable type, the Address type in Ethereum is a string of 20 bytes in length, while the public key in the elliptic curve digital signature should originally mean the coordinates (X, Y) of a point on the curve, so there must be a mutual conversion in the format for them. In the code, this includes three different formats (types): the address variable is an Address type, a string of 20 bytes in length; the publicKey variable is a string with an unknown length; the public key on the elliptic curve is the coordinates of a point. Here, go-ethereum implements the interface in the `crypto/ecdsa` from Golang built-in package, where `ecdsa.PublicKey{}` is represented by the members X, Y .

The conversion has following parts (see Appendix B):

1. Converting the publicKey variable to the Address type, which was introduced in the body of the `core.types.recoverPlain()` function mentioned earlier (at the end of the function, see Appendix A)
2. Converting the publicKey string type and the format function of the `ecdsa.PublicKey{}` type, defined by the crypto code package

3.2 Peer-to-peer Communication

All nodes in the Ethereum blockchain need to first join the underlying peer-to-peer (P2P) network of Ethereum. The node first finds other nodes using Node Discovery Protocol, then they establish the transport connection with RLPx Transport protocol. After the transport connection is established, Devp2p Application Protocol allows nodes to support sub-protocols in an application layer for later establishing Ethereum connections to participate in blockchain consensus.[26] In this chapter, we explain the related protocols according to the official protocol specifications, and the source code of the official Golang implementation of the protocols.

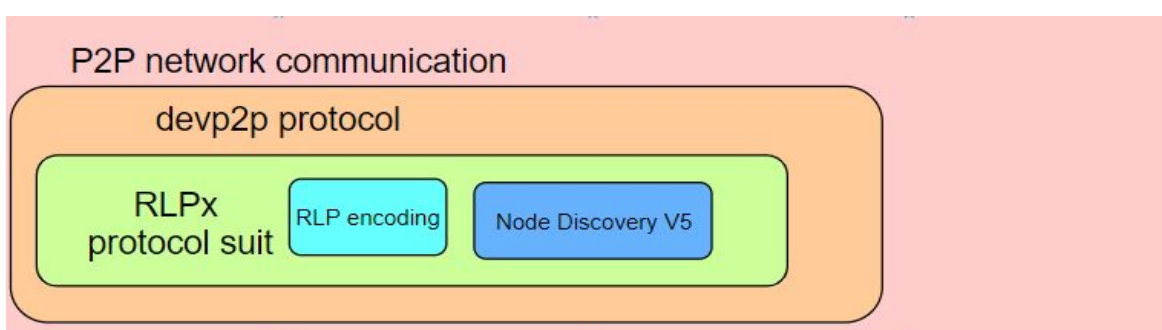


Fig.10 P2P network communication overall architecture

3.2.1 Devp2p Application Protocol

Peer-to-peer communication between nodes running Ethereum clients uses devp2p application protocol. The devp2p protocol is responsible for communication among nodes in a peer-to-peer network in an application layer, in where many sub-protocols can be supported by nodes, and devp2p is designed to allow negotiation of supported sub-protocols on both sides and carries their messages over a single connection [5].

3.2.1.1 Low-Level

The low-level transport of devp2p protocol is mainly implemented by two parts: **Node Discovery Protocol** and **The RLPx Transport Protocol**. The devp2p nodes first find peers via **discovery protocol** DHT(Distributed Hash Table), then nodes are able to communicate by sending messages with encrypted frames of arbitrary content using **RLPx**, the TCP connection will then be listened at a default port 30303 [5]. The detailed information of these two protocols will be introduced in the following sections.

3.2.1.2 Message Contents

All the messages in the Ethereum network are encoded using the **RLP(Recursive Length Prefix)** serialization format mentioned above. It is notable that **RLP** is only used to encode the structure of the data and is completely unaware of the type of object being encoded, there can be many different types of payload to be encoded within the message, the type is determined by the first entry(an integer) of the packet RLP [5].

In Ethereum, sub-protocols are also called capabilities, which indicates what capabilities a peer can have. The devp2p implements a user-friendly simple way to define any sub-protocols over the basic wire protocol, that is to abstract away code standardly shared by protocols.

Each sub protocol must statically specify how many message IDs they require. A message can only be sent once since the Payload reader is consumed during sending, this means both peers must have the same subprotocols they share (including versions). There are total 10 message IDs (from 0x00 to 0x10) are reserved for devp2p messages [5] (see Appendix C).

3.2.2 RLPx Transport Protocol

As mentioned above, RLPx Transport Protocol is designed to allow communication among Ethereum nodes from a low level. It is a TCP-based transport protocol to work with encrypted frames of arbitrary content [6].

3.2.2.1 Node Identity

As introduced earlier, all cryptographic operations in Ethereum are based on the secp256k1 elliptic curve. To connect to an Ethereum node and start communication, each node must have the following information:

- node id
- ip
- tcp port

Here, node id refers to the static private key which is saved and restored between session.

3.2.2.2 Encryption

ECIEC(Elliptic Curve Integrated Encryption Scheme) is implemented in RLPx handshake, which is a hybrid encryption system proposed by Victor Shoup in

2001 and employs the elliptic curves to establish an encryption key [8]. The Scheme of ECIES implemented by RLPx protocol is as below [7]:

1. A key derivation function $KDF(k, len)$: the NIST SP 800-56 Concatenation Key Derivation Function.
2. A symmetric key encryption scheme $AES(k, iv, m)$: the AES-128 encryption function in CTR mode.
3. A message authentication code $MAC(k, m)$: HMAC with hash function SHA25.

Suppose we have two users Alice and Bob, and Alice wants to send a message (an arbitrary bit string) encrypted with her private key to Bob. Knowing Alice's public key, Bob is expected to decrypt the message with his private key. Based on the agreed elliptic curve E and a base point $P \in E(Fq)$ of order r ,

Hence, Alice encrypts the messages as below:

1. Chooses a random integer $k \in \{1, \dots, r-1\}$.
2. Computes $R = kP$ and $Z = kQ$.
3. Derives $(k1||k2) \leftarrow KDF(Z)$, where $||$ denotes concatenation
4. Computes $C = ENC_{k1}(m)$ and $t = MAC_{k2}(C)$. User U1 then sends (R, C, t) .

Then Bob decrypts the messages as below:

1. Computes $Z = aR$
2. Derives $(k1||k2) \leftarrow KDF(Z)$.
3. Computes $t_0 = MAC_{k2}(C)$. If $t_0 = t$ then Bob rejects the ciphertext.
4. Computes $m = DEC_{k1}(C)$.

Now we can see Alice and Bob can have the shared secret message.

3.2.2.3 Handshake

Based on the ECIES encryption scheme above, the 'handshake' process is to establish key material to be used for both the initiator and recipient in the communication session. [6] The protocol process is as follows:

1. In the first phase of communication, the initiator initiates a TCP request encrypted with the recipient's public key (node_id) and sends its own public key and signature containing the temporary public key, and a randomly generated nonce.
2. The recipient receives the information, obtains the initiator's public key, and uses the ECIEC algorithm to finally obtain the initiator's temporary public key from the signature (to be used in step 4). The recipient encrypts and sends its temporary public key and random nonce with the initiator's public key

3. The initiator obtains the temporary public key and nonce of the recipient and uses the ECIEC algorithm to calculate the shared key from its own temporary private key and the temporary public key of the recipient. The shared key is used to encrypt the communication process and the nonce is used for verification of information sent by the peer.
4. The recipient performs the same calculation and obtains the shared key with its temporary key and the initiator's temporary public key.

The first phase of the key exchange is completed, now the initiator and the recipient has the same shared key, and the other's nonce. The shared key is used to encrypt future communications (symmetric encryption is more efficient than asymmetric encryption), and nonce is used to generate mac (message authentication code) to ensure complete information is received

1. The recipient sends the first payload frame
2. The initiator sends the first payload frame
3. The recipient receives and authenticates first payload frame
4. The initiator receives and authenticates first payload frame
5. The cryptographic handshake is complete if MAC of first payload frame is valid on both sides

3.2.2.4 Framing

After the key exchange is performed, the future communication must use the Framing format. The Framing contains the head and body, which is very similar to the format of the TCP packet.

There are two purposes for framing packets:

- Allowing multiple protocols over a single connection
- Allowing encrypted stream easier with authenticated frames

3.2.3 Node Discovery Protocol

Ethereum's underlying distributed network also referred to as the P2P network, implements the classic Kademlia network to store information about Ethereum nodes.

3.2.3.1 Kademlia Network

Kademlia Network was proposed by Petar P. Manmounkov and David Mazières of New York University in 2002. It is a distributed hash table (DHT) technology using XOR as the basis of distance measurement and has been applied in BitTorrent BitComet, Emule and other software.[10]

3.2.3.2 Node Identity and Distance

In the Kad network, each node has a unique node ID, which is the public key on the elliptic curve secp256k1. Here, the concept of 'distance' between two node IDs is not the physical distance but refers to the distance in logic, the calculation is to take xor operation on two node IDs (the hashes of the public keys). The distance between the two nodes A and B is calculated as:

$$D(A, B) = A.ID \text{ xor } B.ID. \quad (19)$$

XOR has an important property, that is if we suppose a, b, and c are any three numbers. If $a \text{ xor } b = a \text{ xor } c$ is true, then $b = c$. Therefore, if the node a and the distance L are given, then there is only one node b, which makes $D(a, b) = L$. In this way, the logical distance between different nodes in the Kad network can be effectively measured.

3.2.3.3 Node Table

Based on the XOR distance metric, Kad network can also organize the entire network topology into a binary prefix tree as shown in the following figure. Each Node ID will be mapped to a leaf on the binary tree [12].

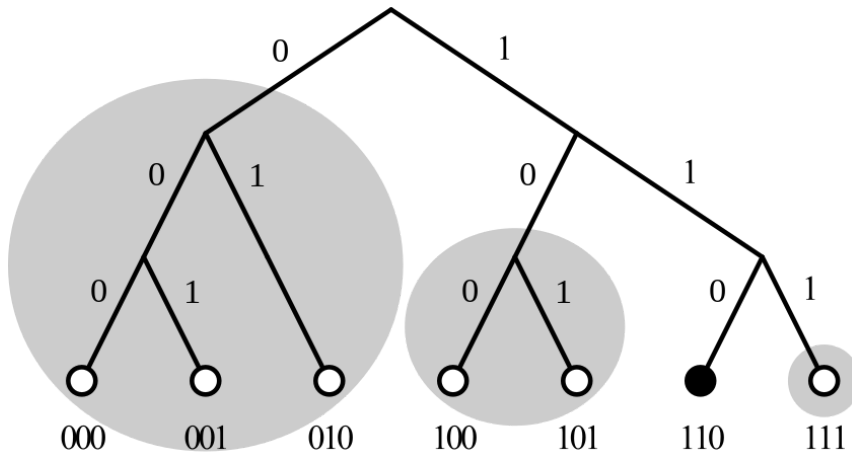


Fig.11 Binary Prefix Tree

With this binary tree structure, the closer the node XOR distance is, the closer they are near each node. Then, the entire binary tree can be split with the XOR distance. After the subtree split is completed, as long as one of the nodes in each subtree is known, recursive routing can be performed to implement traversal of all the nodes of the entire binary tree.

However, in practice, since the nodes are dynamically changed, generally multiple nodes need to be known instead of only one node of each subtree. Therefore, Kad implements a method called **K-bucket**, each bucket will record multiple nodes known in each subtree. In fact, a **K-bucket** is a routing table. If there are m subtrees split, the corresponding nodes will have m routing tables. Each node keeps its own m K-buckets. The node information recorded in each K-bucket generally includes the NodeID, IP, Endpoint, and Target node information(that is, the node that keeps the K-bucket).

In Ethereum, the number of K-buckets kept by each node is 256. These 256 K-buckets are sorted according to the XOR distance from the Target nodes. The maximum number of nodes saved in each K-bucket is 16 [11].

3.2.3.4 Wire Protocol

In the Ethereum Kad network, the communication between nodes is based on UDP datagrams and the maximum size of any packet is 1280 bytes. There are four main packet types are tabled below, if the PING-PONG handshake between two nodes establishes, the corresponding node is considered to be online.

No.	Types	Function	packet data
0x01	Ping	Used to detect if a node is online	[version, from, to, expiration] version = 4 from = [sender-ip, sender-udp-port, sender-tcp-port] to = [recipient-ip, recipient-udp-port, 0]
0x02	Pong	Used to respond to the ping command	[to, ping-hash, expiration]
0x03	FindNode	Used to find other nodes that are XORed closest to the Target node	[target, expiration]
0x04	Neighbours	Used to respond to the FindNode command, returning one or more nodes	[nodes, expiration] nodes = [[ip, udp-port, tcp-port, node-id], ...]

Table.1 Ethereum UDP packet types

3.2.4 Code Implementation In go-ethereum

Based on P2P related protocols mentioned above, now we dig into how Ethereum implements these protocols in the source code of client go-ethereum. First, an overall architecture figure of all network layers of Ethereum is given below. It is shown that all network functions are built on the transport layer of Ethernet, both TCP and UDP are applied.

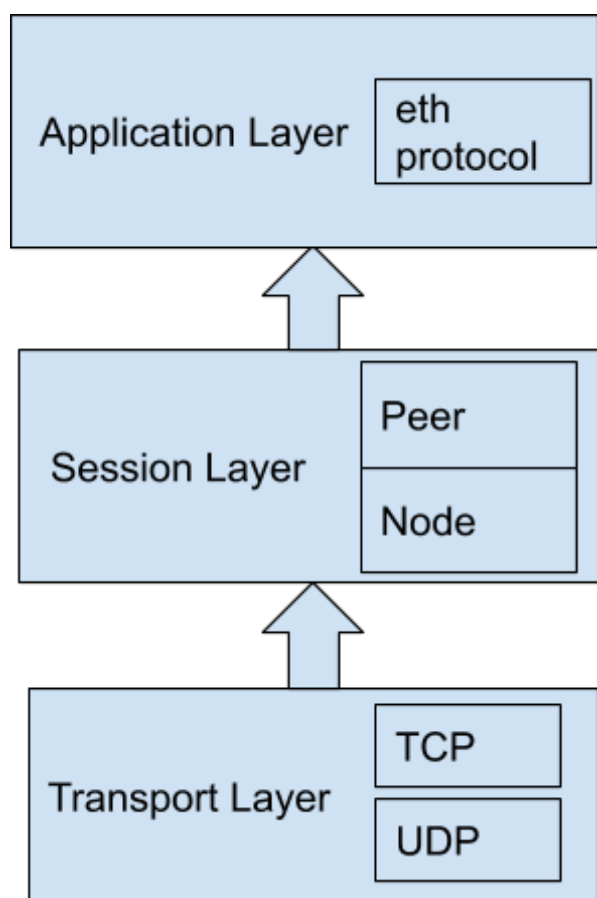


Fig.12 Overall Architecture of Network Layers

3.2.4.1 Node Discovery Based on UDP Implementation

The underlying P2P module of Ethereum is designed for the communication and service discovery between nodes, and the new node discovers the connection function.

For go-ethereum client, the P2P module has two parts:

1. Discover nodes: how to find other nodes nearby.
2. Connect nodes: how to connect to other nodes and communicate

Ethereum uses UDP streaming for node discovery service, since the communication content is relatively simple, there is no encryption. And TCP is used for real data transmission and interaction with the encrypted connection.

First, Ethereum will start from P2P package, and the discover of the P2P module will automatically discover nearby nodes, specifically in `p2p/server.go`. Node discover protocol is implemented in `p2p/discover` module, there are two important data structure UDP and Table.

UDP Structure

The udp structure is used to record the corresponding listening UDP connection, and most importantly, is to anonymously combine the Table structure. This structure is mainly used to handle protocol communication and handshake function (see Appendix D).

Table Structure

The Table structure is used to manage all neighbouring nodes, and to perform reconnection of neighbouring nodes. If the connection needs to be handshaked, the UDP function will be called to process. The Table structure will also retain a net pointer, which is actually udp type. This can call the interface of the upper layer to perform protocol handshake.

Neighbour node discovery

A neighbour node is a node that added into the K bucket and establishes handshakes through PING-PONG protocol mentioned above. The process can be shown as below.

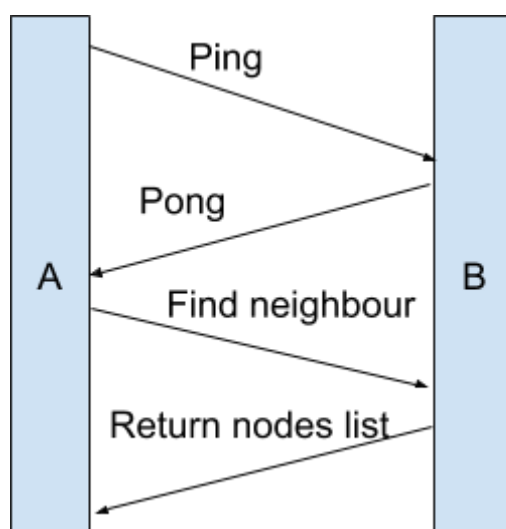


Fig.13 Neighbour Node Discovery

3.2.4.2 Data Communication based on TCP Implementation

As mentioned above, go-ethereum has two important parts, UDP-based node discovery module and TCP-based data communication module. The node discovery part is discussed above, which is used to discover more nodes of the whole network. In the node discovery part, the nodes are just found to be pinged, but they have not been really used yet. such as establishing a TCP connection for data transmission, protocol processing, etc. Here is an analysis of how the TCP connection of the P2P network is established and how it communicates with other nodes, the overall process is shown below.

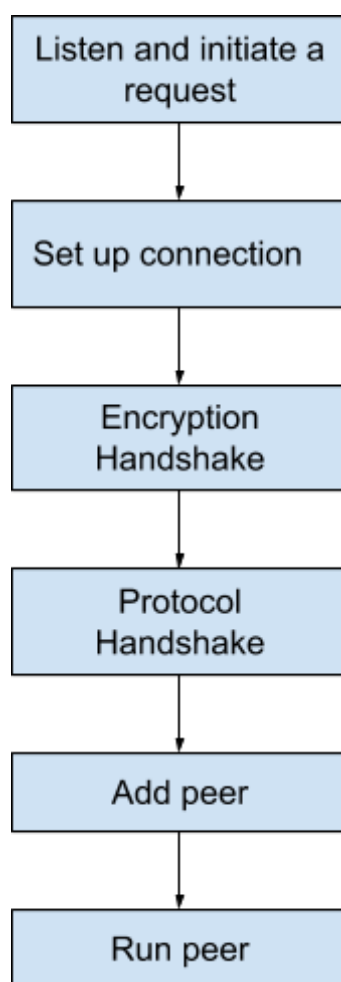


Fig.14 TCP Connection Process

TCP listening process

`Server.start()` in P2P will start `discover.ListenUDP` to enable the background coroutine to perform UDP listening, and continuously read the data packet

for processing, thus implementing the node discovery logic of UDP protocol. After that, it will start the node connection function of the TCP layer and listening for requests from other nodes to perform the encrypted handshake.

In the code(see Appendix F), `startListening()` is first called to start listening for TCP requests, then `listenLoop()` function is called to accept the connection with an infinite loop, and then a P2P communication link for the normal connection is established through the `SetupConn()` function. In `SetupConn()`, `setupConn()` is the actual implementation (see Appendix G).

The `setupConn()` function mainly does as follows:

1. exchanges keys with the client with the `doEncHandshake()` function
2. generates a temporary shared key for this communication encryption
3. creates a frame processor `RLPXFrameRW`
4. calls the `doProtoHandshake()` function for this communication to set the rules including the version number, name, capacity, port number, and so on.

The whole TCP listening process is as below:

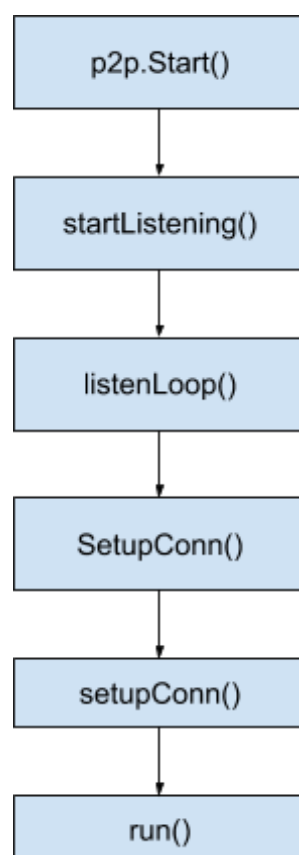


Fig.15 TCP Listening Process

RLPx Encrypted Handshake Protocol

As mentioned above, the TCP transport connection is encrypted. Before setting up the connection, the system needs to do handshake through RLPx encryption protocol. The first step is to obtain a shared key, in which the detailed mechanism is described before. In Ethereum source code, the shared key generation and key exchange are done by the `doEncHandshake()` method (see Appendix H).

From the code, we can see that if the server is listening for a connection, the `receiverEncHandshake()` function is called after receiving a new connection. If the client initiates a request to the server, the `initiatorEncHandshake()` function is called. These two functions are not much different, they will both exchange keys and generate a shared key. `initiatorEncHandshake()` is to initiate the connection and call `newRLPXFrameRW()` to create the frame processor. The `receiverEncHandshake()` function (see Appendix I) is called to create the shared key. The encrypted handshake protocol flow is as below:

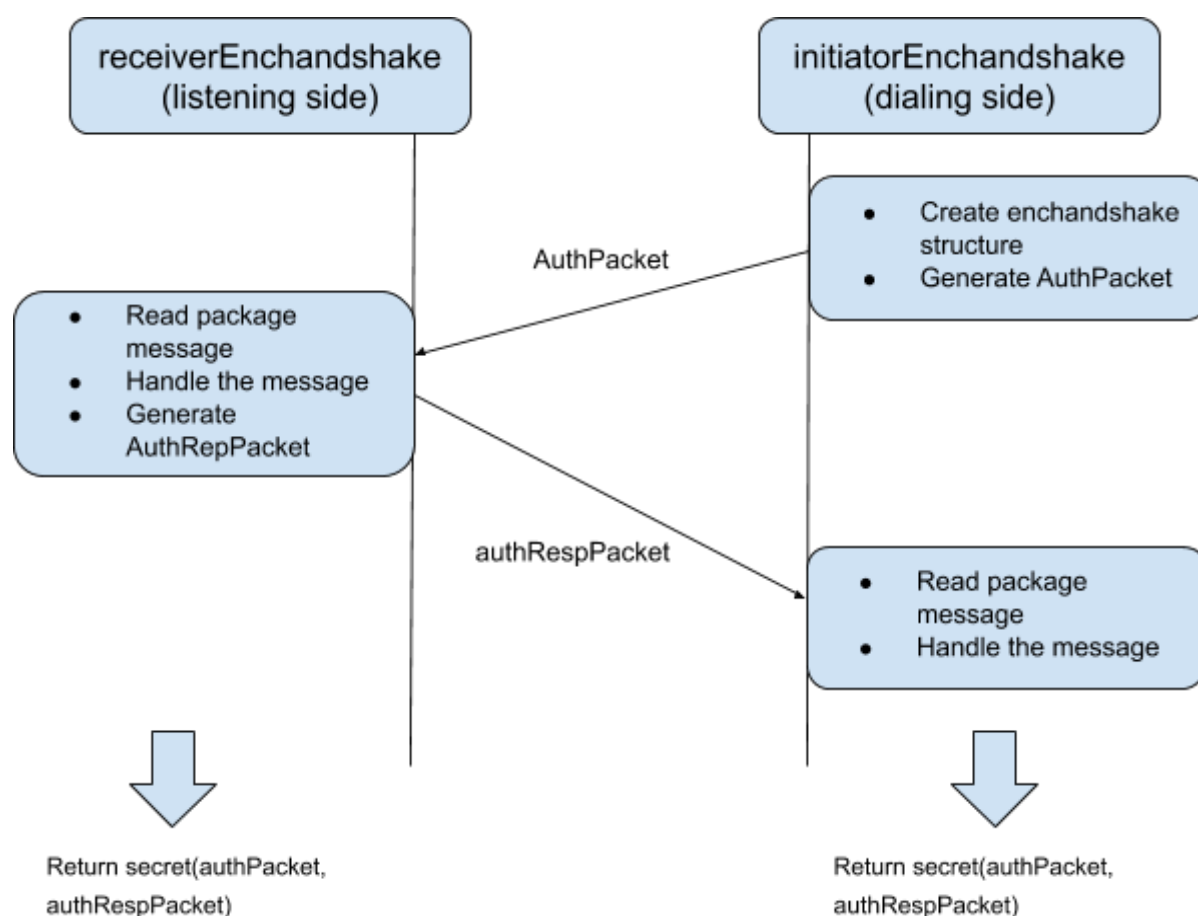


Fig.16 TCP Encrypted Handshake Protocol

After the communication connection is successfully established and the protocol handshake is completed, the processing flow is moved to connect peers module.

TCP connecting peers process

Next, `p2p.Start()` will handle the message by calling the `run()` function. The `run()` function (see Appendix J) waits for the transaction with an infinite loop. For example, after the new connection completes the handshake, it will be responsible for this function. The `run()` function supports the processing of multiple commands, including cleaning up service, sending handshake packets, adding new nodes, deleting nodes, and so on.

Here, we only discuss the `addpeer` branch: When a new node is added to the server node, it will go to this branch and generate an instance for the upper layer protocol based on the successful handshake information, then `runPeer()` is called to finally handle events through `p.run()`.

The `p.run()` function opens two coroutines to read data in loop and ping, which are used to process the received message and keep the connection, then it will call the `Run()` function of the specified protocol by calling the `startProtocols()` function.

The process flow is shown below:

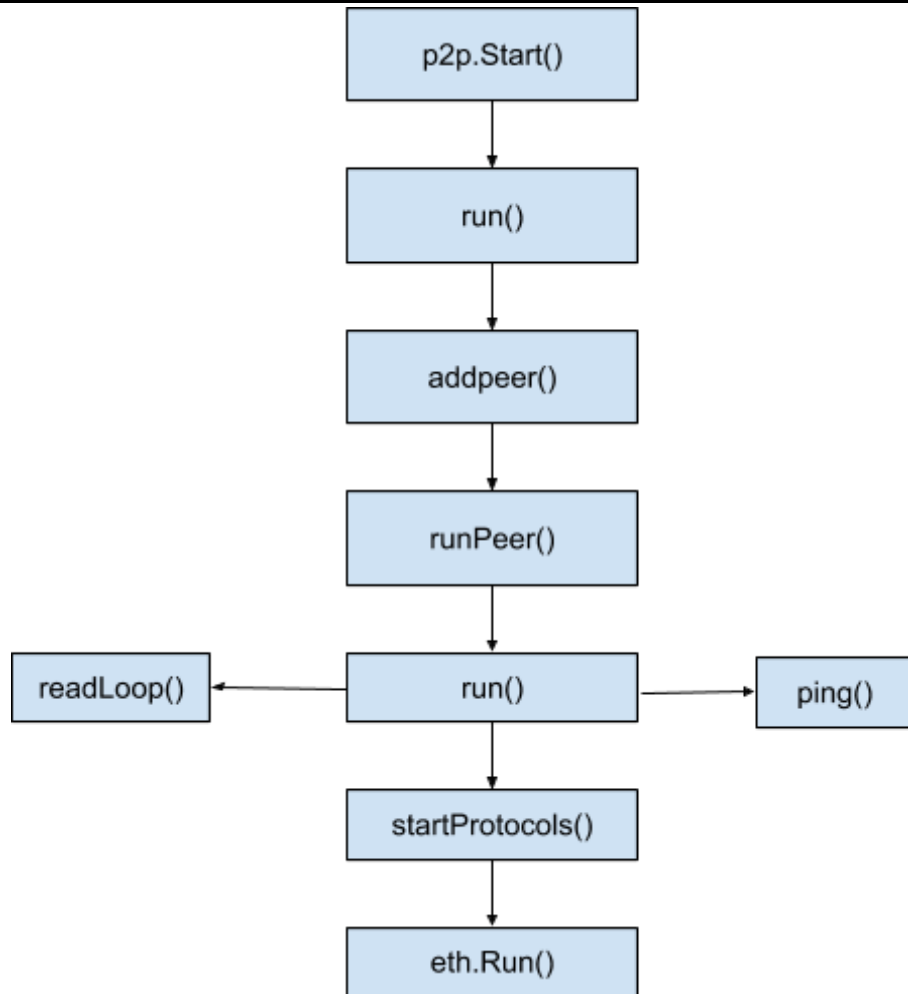


Fig.17 TCP Connecting Peers Process

At this point, the `P2P.Peer` module completes the data reading for the connection that just established with the `rlpx` encryption protocol, and finally puts the reading message into the pipeline and gives it to the corresponding protocol for later processing.

Then we take `eth` protocol as the example, the overall `eth` protocol processing flow is as below:

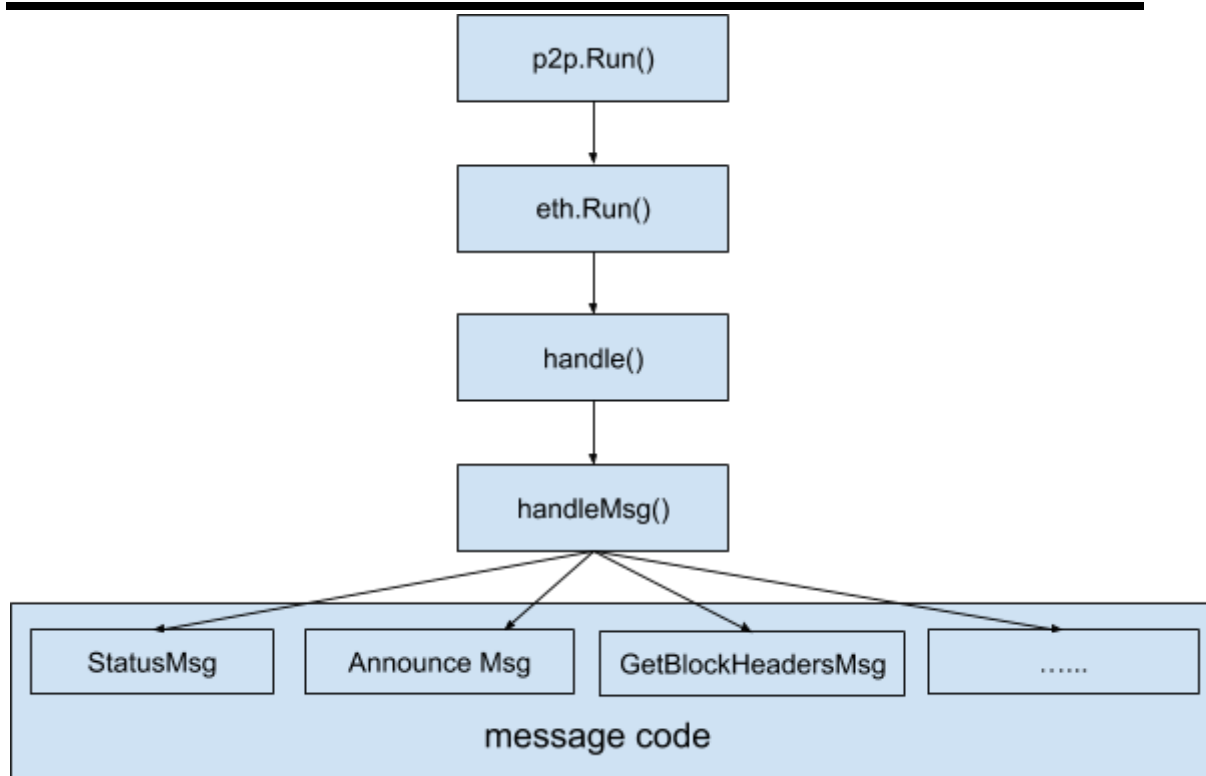


Fig.18 Eth Protocol Process

3.3 Proof of Work

3.3.1 Proof-of-Work System

Proof of Work (PoW), can be regarded as a proof to confirm that a certain amount of work has been done. Generally monitoring the entire process is extremely inefficient, and it is a very efficient way to prove the completion of the corresponding workload by certifying the results of the work. For example, in real life, things like diplomas, driver's licenses, etc, are also proofs obtained by verifying results (to pass relevant examinations).

The Proof of Work system (or protocol, function) is an economic response to the denial of service attacks and other service abuse. It requires the initiator to perform a certain amount of operations, which would take a certain amount of time. This concept was first proposed by Cynthia Dwork and Moni Naor in an academic paper in 1993. The term Proof of Work (POW) was actually proposed in the 1999 article by Markus Jakobsson and Ari Juels. [13]

3.3.2 Hash Function

A hash function is a function that computes the corresponding output $H(x)$ with the given input x . The main feature of the hash function is as below[14]:

1. Input x can be a string of any length
2. The output of $H(x)$ has a fixed length.
3. The process of computing $H(x)$ is efficient (for a string of length x , the time complexity of calculating $H(x)$ should be $O(n)$)

For the hash function used by Blockchain system, it needs to have the following additional properties[14]:

1. Collision-free, that is to say, there is no input $x \neq y$ to have $H(x) = H(y)$.
2. Non-invertible, that is to say, for a given output result $H(x)$, it is computationally impossible to reverse the input x .
3. There is no better way than the method of exhaustion to make the hash result $H(x)$ stays within a certain range.

The properties of hash functions above are the cornerstone of Blockchain systems including Ethereum's PoW mechanism.

3.3.3 Distributed Consensus

General speaking, a trustless or distributed consensus system is interconnected by distributed nodes in different locations and there is no central node in the system. In such a distributed system, how to synchronise the data with all nodes in the cluster and agree on a proposal is the core problem of the distributed system, and the consensus algorithm is used to achieve the consistency of the distributed system.

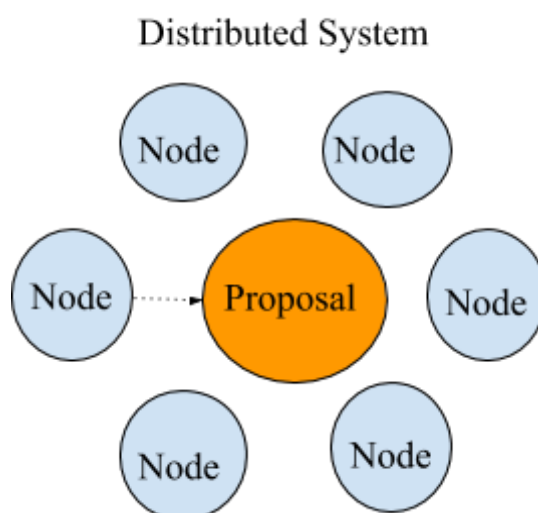


Fig.19 Distributed System

However, with multiple nodes, there can be many complicated situations in the distributed systems. As the number of nodes increases, node failure or downtime becomes a very common thing, boundary conditions and unexpected conditions in the system also increase the difficulty of solving distributed consistency problems.

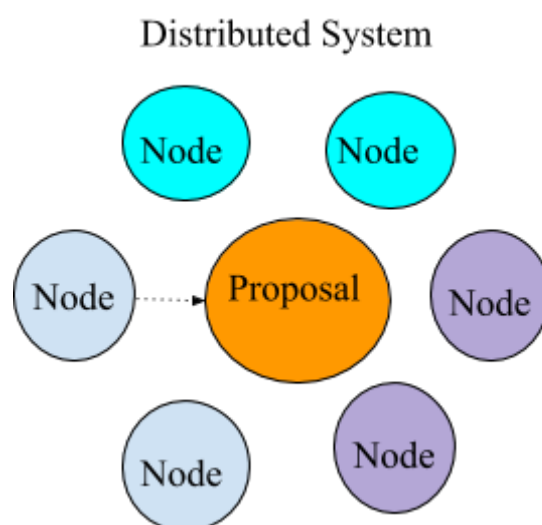


Fig.20 Distributed System with complicated situations

Based on those problems, **The Byzantine Generals Problem** is introduced to state synchronization problem of distributed systems. It is a fault-tolerant problem in the distributed domain proposed by Leslie Lamport in The Byzantine Generals Problem paper, which is the most complex and rigorous fault-tolerant model in the distributed domain. In this model, the system does not impose any restrictions on the nodes in the cluster. They can send random data, error data to other nodes, or choose not to respond to requests from other nodes. These unpredictable behaviours make fault tolerance. It has become more complicated [27].

The Byzantine General problem describes a scenario in which a group of generals directs a part of the army. Each general does not know whether the other generals are reliable or whether the information transmitted by other generals is reliable, but they need to vote to choose whether or not to attack or retreat. At this time, no matter whether the general is reliable or not, as long as all the generals have reached a unified plan, choosing to attack or retreat is actually without any problem. But if one of the generals tells that some of them will choose to attack and the other chooses to retreat, there will be very serious problems [27].

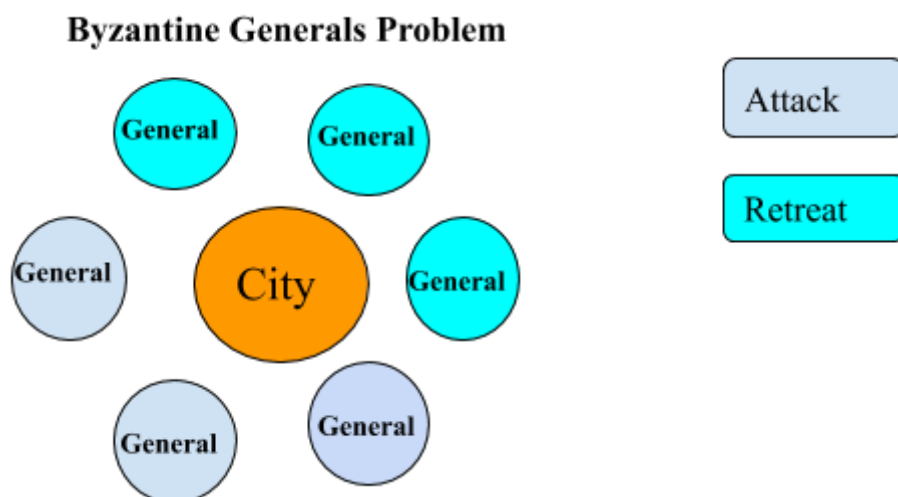


Fig.21 The Byzantine Generals Problem

3.3.4 Proof of work and mining

PoW(Proof of Work) is currently the most popular consensus algorithm for solving the General Byzantine problem above. Bitcoin, Ethereum and other mainstream blockchain digital currencies are all based on POW.

In order to solve the General Byzantine problem, it is necessary to first select a loyal "General" among these equal generals, and other generals can listen to his decision. This seems against the idea of decentralization. However, these generals were decentralized equal nodes before making this decision, and the selected general can only make the decision for this time, the general to make the decision will be selected again the next time. This is unlike centralizing, where the general to make decisions will be the same one every time [28]. The next thing to solve is how to select the general. The basic idea is as below:

- Before the decision, these generals were equal nodes, so they should judge by their speeches at the time of this decision. The generals use the known battlefield intelligence to estimate the current battlefield situation, calculate it, and broadcast it to other generals after the conclusion (The block).
- At the same time, the general will always listen to the broadcast content from other generals. Once receiving the conclusion broadcasts from other generals, the general will immediately stop the calculations in their hands to verify the content of the broadcast. If all the generals pass the verification, then the first general who starts this verifiable pass-through broadcast was selected to make the decision, and all other generals will listen to his conclusions (the content of the results already included in the broadcast)

There are two important factors in this process:

- Speed. Only the first one to pass the verification can be selected as the general.
- Correctness, the conclusions of the general are correct, which can be verified by other generals.

Here the speed is a problem of computing power, which is related to hardware specifications. While the PoW algorithm is used to solve the correctness problem. The PoW provides a way to calculate and verify easily, which is based on the properties of the hash function mentioned above.

With these properties, the PoW algorithm can send to each node a hash encryption function, and each node calculates a cryptographic hash by adding a nonce value to the sealed block information. This cryptographic hash needs to be satisfied certain rules (for example, the first four numbers must be 1111). Each node can only implement exhaustive method until the condition is met, that is, the block is successfully generated. At this time, the block hash is broadcasted again, and are verified by the other nodes, that is the predetermined rule is met. (The first four are indeed 1111), then the consensus is achieved, the block generated by the node just broadcast is added to the main chain. Here the term ‘work’ refers to the amount of work that the node is constantly trying to calculate. After obtaining and broadcasting the hash of the qualified block, the other nodes are doing the work at hand and the completed work is invalid (in fact, this is also a wasted calculation).

The word ‘mining’ refers to such a process, which is like the analogy between cryptocurrency and gold. Gold or precious metals are rare, as well as electronic tokens, the only way to increase the total amount is mining. The same is true of Ethereum, the only way to increase cryptocurrencies is to mine. Mining is also a way to protect the network by creating, verifying, distributing, and spreading blocks in the blockchain [18].

Ethereum, like all blockchain technologies, uses an incentive-driven security model. The consensus is based on selecting the block with the highest total difficulty. Miners create blocks to be checked for validity, where blocks are only valid when they contain a PoW difficulty [18].

3.3.5 Ethash Algorithm

Ethash is a PoW (Proof of Work) algorithm used in Ethereum, a modified version of Dagger-Hashimoto algorithm. The key feature of it is that the computational efficiency is basically independent of the CPU, but is positively related to memory size and memory bandwidth. Therefore, the large-scale deployment of mining machine chip through shared memory does not have linear or superlinear growth in mining efficiency.

Dagger Hashimoto is the original algorithm that the Ethereum mining algorithm Ethash has developed from, the purpose of Dagger Hashimoto is [19]:

- Resisting mining machine (ASIC)
- Light client authentication
- Full chain data storage

However, Dagger and Hashimoto are actually two algorithms. **Hashimoto algorithm** was invented by Thaddeus Dryja, it is designed to resist mining machines through IO restrictions. This can be achieved by setting the memory read limit conditions in the mining process. This is because the memory device will be cheaper and more common than the computing device with billions of dollars of research go into optimizing memory to adapt to different user cases, limiting memory read would make existing RAM(Random Access Memory) be moderately close to optimal for evaluating the algorithm. The Hashimoto algorithm uses the blockchain as the source data to meet the requirements of 1 and 3 above [19].

Dagger algorithm was invented by Vitalik Buterin. It utilizes the directed acyclic graph DAG to implement features of difficult memory computing and easy verification with the **Memory-Hard Function**. The main principle is [19]:

- Computing each nonce requires a small portion of the DAG.
- The mining process would store the complete DAG,
- Prohibit the recomputing of the corresponding subset of the DAG each time, and the verification process is allowed.

Memory-Hard Function can be regarded as a way to achieve ASIC resistant. As know that mining basically depends on our computer specifications, so some hardware manufacturers make hardware devices that are specifically used for mining. They are not a complete PC, such as ASICs, GPUs, and FPGAs. This equipment as mining machines are better than the ordinary PC mining, which is against the decentralization spirit of blockchain technology, so we must make the mining equipment equal. To achieve this, Dagger algorithm just measures the hardware performance in memory during the mining process, one computer has only one total memory no matter how good hardware specification the computer has, thus, multi-core parallel processing does not work better. In this way, whether it is a PC or an ASIC, GPU or FPGAs, the requirements for equal mining can be achieved [19].

Ethash algorithm process:

- First compute a seed based on the block information
- Using this seed, compute a 16MB cache data. The light client needs to store this cache.

- From this cache, we are able to generate a 1GB data set, each of which depends on a small portion of the cache. The complete client and miner store this data set, and the data set grows linearly over time
- Only a small amount of memory can be used to quickly compute the data at a specified location in the DAG based on the cache, the light client only needs to store the cache to perform efficient verification.

The dataset mentioned above is updated every 30,000 blocks, so the job of most miners is to read the data set instead of changing it. [20]

3.3.5 Code Implementation In go-ethereum

This part is intended to introduce the entire process of "mining" to get new blocks, as well as the implementation details of different consensus algorithms in the go-ethereum source code. For a process of mining, the code implementation is basically divided into two parts: one is to assemble a new block, the data of this block is basically complete, including some attributes of the member Header, the transaction list txs, uncle block, all transactions have been executed, all receipts (Receipt) have also been collected, this part is mainly done by the worker; the second is to fill the remaining member attributes of the block, such as Header, Difficulty, etc, and complete authorization, these tasks are achieved by the Agent <Engine> interface implementation with using the consensus algorithm. The UML diagram of the main structure is shown below.

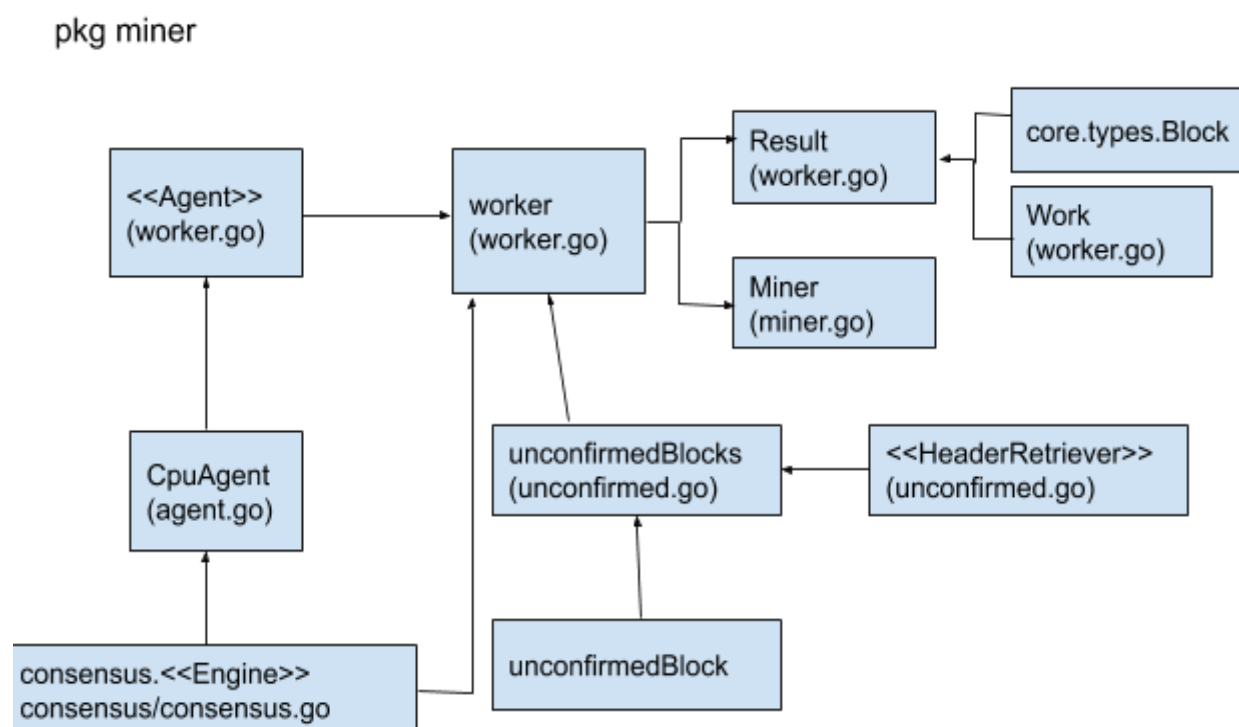


Fig.22 UML Diagram of miner package

3.3.5.1 Block Creation

The process of creating new blocks is in the interface `Miner`. The whole structure of `Miner` struct is as below:

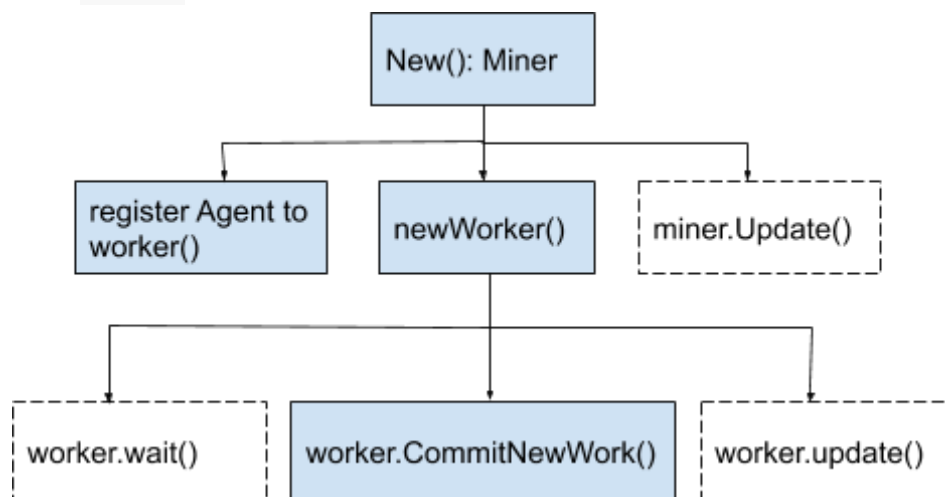


Fig.23 Miner Struct

Functions of Miner

The entire Ethereum mining-related operations are exposed through the `Miner` struct(see Appendix).

New()

In `New()`(see Appendix L), after the new object `miner` is initialized, the `worker` object will be created, then the `Agent` object is registered to the `worker`, and finally, a separate thread is used to run the `miner.Update()` function.

update()

`update()` (see Appendix M) will subscribe (listen) to several events related to `Downloader`. When receiving the `StartEvent` of the `Downloader`, it means that the node is downloading a new block from other nodes at this time. At this time, the `miner` will immediately stop the ongoing mining work and continue to listen; if the `DoneEvent` or `FailEvent` is received, it means the download task of the node has ended which means whether the download succeeds or fails, at this point you can start mining new blocks and exit the listener event listener at this time.

From the logic of `miner.Update()`, it can be seen that for any node in the Ethereum network, mining a new block, downloading and synchronizing a new block from other nodes are completely conflicting. Such a rule can guarantee that a new block may only have one source on a certain node, which can greatly reduce possible block conflicts and avoid waste of computing resources in the entire network.

Functions of worker

Here we mainly focus on `worker.updater()` and `wait()`

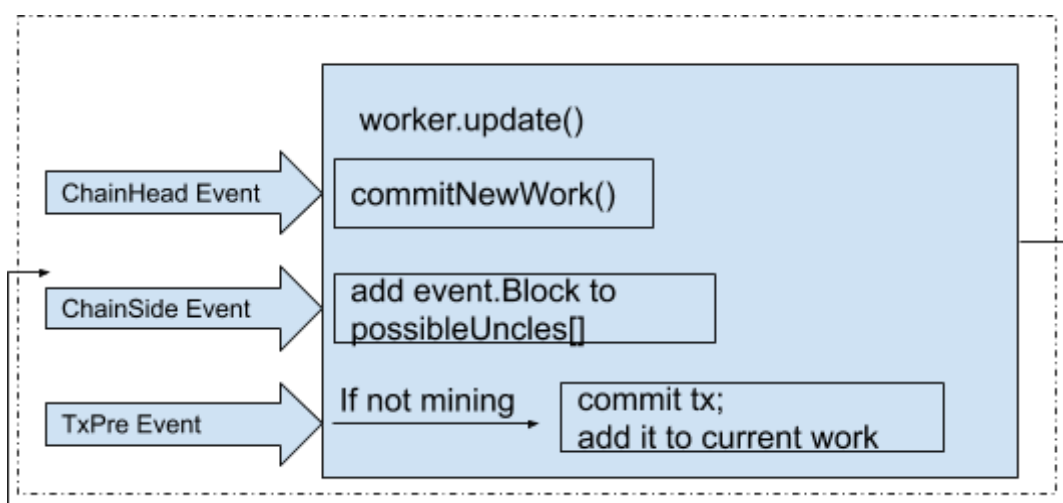


Fig.24 process of `worker.update()`

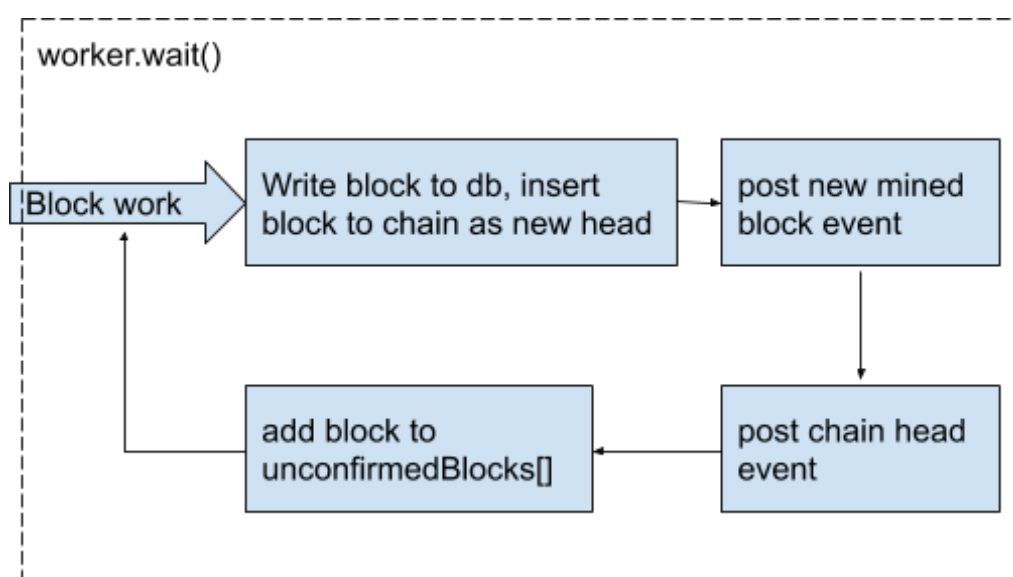


Fig.25 process of `worker.wait()`

update()

`worker.update()` (see Appendix N) will listen to `ChainHeadEvent`, `ChainSideEvent`, and `TxPreEvent` respectively, each event triggers a different reaction of the worker. These events are described as below:

- `ChainHeadEvent`: a new block has been added to the blockchain as the head of the entire chain. The worker will immediately start preparing to mine the next new block.
- `ChainSideEvent`: a new block is added to the blockchain as a side branch of the current chain head. The worker will put this block into the `possibleUncles[]` array as one of the next Uncles to mine the new block.
- `TxPreEvent`: it is issued by the `TxPool` object when a new transaction `tx` has been added to the `TxPool`. At this time, if the worker is not in the mining, then it will execute the `tx` and put it into the `Work.txs` array to reserve the new block for the next time.

wait()

`Worker.wait()` (see Appendix O) will wait for the Agent to finish mining the new `Block` and `Work` objects sent back at a channel. This block will be written to the database, and join the local blockchain to try to be the newest chain.

When this is done, the worker will send an event (`NewMinedBlockEvent{}`), which means announce to the world: a new block is mined by this node. In this way, other nodes that listen to the event will decide whether to accept the new block to become the new chain head in the whole network according to its own situation. As for how this consensus is implemented, it will be explained later in the consensus algorithm implementation.

commitNewWork()

`commitNewWork()` (see Appendix P) will be called many times in the worker. It is called directly every time and is not started in goroutine mode. `commitNewWork()` internally uses `sync.Mutex` to isolate all operations. The basic logic of this function is as follows:

1. Prepare the time attribute `Header.Time` of the new block, which is generally equal to the current time of the system,
2. Create a header object for the new block, but the properties such as `Difficulty`, `GasLimit`, etc., are left to be determined in the consensus algorithm.
3. Call the `Engine.Prepare()` function to complete the preparation of the Header object.

4. According to the location of the new block (Number), check if it is within the influence of DAO hard fork, and if so, assign it to header.Extra.
5. Create a new Work object and update the worker.current member variable with the existing Header object.
6. If the configuration information supports hard forks, apply hard forks in the StateDB of the Work object.
7. Prepare a list of transactions for the new block, sourced from those recently added tx in TxPool, and execute these transactions.
8. Prepare the new block's uncle block uncles[], the source is `worker.possibleUncles[]`, and each block in `possibleUncles[]` is collected from the event ChainSideEvent. Note that there are at most two uncle blocks.
9. Call the `Engine.Finalize()` function to "set" the new block, add several attributes such as Header.Root, TxHash, ReceiptHash, UncleHash, etc.
10. If the previous block (that is, the old chain block) is in unconfirmedBlocks, it means that it is also mined by this node, try to verify that it has been absorbed into the backbone chain.
11. The created Work object is sent to each registered Agent through the channel for subsequent mining.

In the steps above, only step 4 and step 6 are needed to support the DAO hard fork in the block configuration, the other steps are universal. Now that `commitNewWork()` completes the creation of the block to be mined, the block.Header is created, the transaction array txs and the unblock `Uncles[]` have been obtained, and since all transactions have been executed, the corresponding `Receipt[]` has also been obtained. Everything is ready, and Agent can start to "mine".

Functions of CpuAgent

Here we mainly focus on `CpuAgent.updater()` and `CpuAgent.mine()`

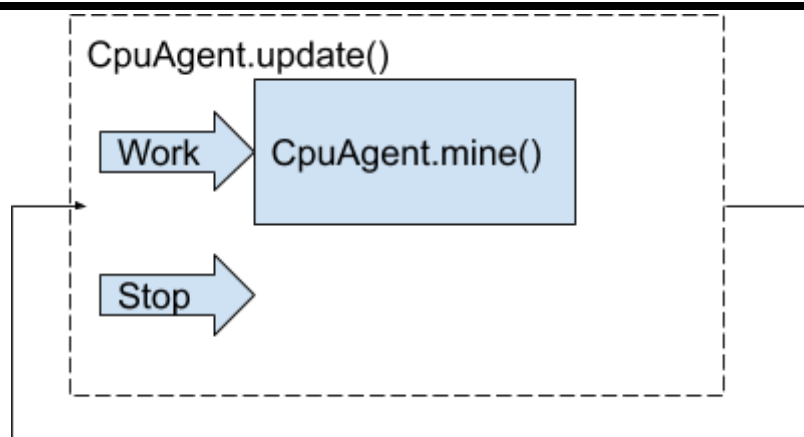


Fig.26 process of CpuAgent.update()

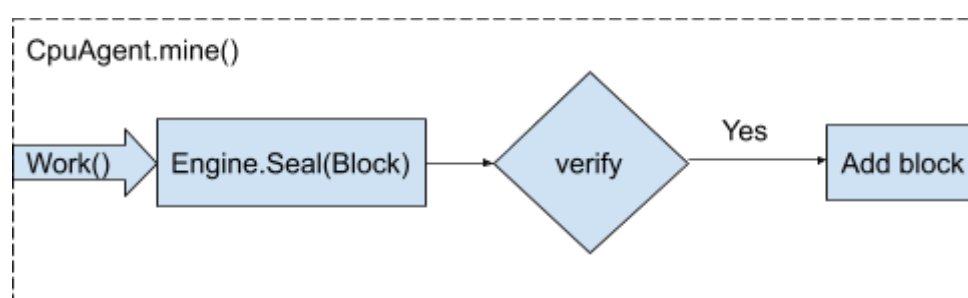


Fig.27 process of CpuAgent.mine()

`CpuAgent.update()` (see Appendix Q) is the worker object that is sent after the `worker.commitNewWork()` is finished. If the `Work` object is received (issued by the end of `worker.commitNewWork()`), the `mine()` function will be started; if it the message of stopping mining, it will quit all related operations.

`CpuAgent.mine()` (see Appendix R) will directly call the `Engine.Seal()` function, and use the consensus algorithm implementation of `<Engine>` interface to perform the final authorization on the incoming `Block`. If successful, the `Block` will be returned to the worker through the channel together with the `Work`. `Worker.wait()` will receive and process it.

Obviously, these two functions do not do any substantive work, they are only responsible for calling the `<Engine>` interface implementation, and sending the block data back to the worker after the authorization is completed. The real work of mining a block is in the consensus algorithm of the `Engine` implementation.

3.3.5.2 Consensus Algorithm

The consensus algorithm package exposes the Engine interface, which has two implementations, namely the Ethash algorithm based on PoW and the Clique algorithm based on PoA, in this part we mainly focus on Ethash implementation.

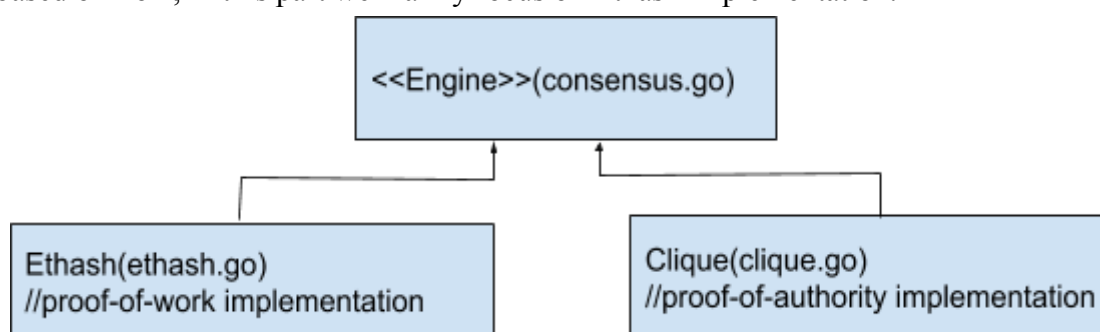


Fig.28 consensus engine in Ethereum

In the Engine interface, functions such as `VerifyHeader()`, `VerifyHeaders()`, `VerifyUncles()` are used to verify whether the corresponding data members of the block are reasonably correct and can be added in the block. The `Prepare()` function is often called when the Header is created to assign values to properties such as `Header.Difficulty`. The `Finalize()` function is called when the data members of the block are available. For example, the uncles are already available, all transactions have been executed, and all receipts (`Receipt[]`) have been collected, in such a condition `Finalize()` will eventually generate members such as `Root`, `TxHash`, `UncleHash`, `ReceiptHash`, etc.

`Seal()` and `VerifySeal()` are the most important of all the functions of the Engine interface. The `Seal()` function authorizes or seals a block that has been called `Finalize()` and assigns some values from the seal process to the remaining unassigned members in the block (`Header.Nonce`, `Header.MixDigest`). When `Seal()` succeeds, all the blocks returned are complete and can be regarded as a valid block, which can be broadcasted to the entire network, or added into the blockchain. Thus, for mining a new block, `Engine.Seal()` is the most important and most complicated step in all relevant code.

The `VerifySeal()` function is based on the same algorithm mechanism as `Seal()`. It determines whether the block has been processed by `Seal()` operation by verifying whether certain attributes of the block (`Header.Nonce`, `Header.MixDigest`, etc.) are correct.

mine() function

In the `Ethash.Seal()` function implementation, the `mine()` (see Appendix S) function is called in parallel through a multi-threaded (goroutine) manner. The number of threads is equal to `Ethash.threads`. If `Ethash.threads` is set to 0, `Ethash` will choose to use the total number of cores in the local CPU as the number of threads to open threads.

The main business logic of the `mine()` function. the `id` is the thread number, which is used to send the log to inform the upper layer; the function first defines a set of local variables, including the hash, nonce, huge auxiliary array dataset, and the result of the comparison when the `hashimeFull()` is called; then there is an infinite loop, each time `hashimeFull()` is called to perform a series of complex operations, once its return value meets the condition, it will copy the Header object (deep copy), and assign the Nonce, MixDigest attributes to return the authorized block; in each loop operation, nonce will increment by 1, making the calculations in each loop different.

Here the `hashimotoFull()` (see Appendix T) function will complete the operation by calling the `hashimoto()` function, while there is another similar function, the `hashimotoLight()` function.

These two functions above both called `hashimoto()`, but they use different parameters for `hashime()` function. Compared with `hashimotoLight()`, `hashimotoFull()` has a larger 'size' and a query function `lookup()` that obtains data from a larger array. Here, the `hashimotoFull()` function is called by `Seal()`, and `hashimotoLight()` is prepared for `VerifySeal()`

The core function `hashimoto()`

The `hashimoto()` (see Appendix U) function is the actual implementation of the Ethash consensus algorithm. The input parameters of the `hashimoto()` function including the block hash value-hash, the block nonce member - nonce, the hash function `lookup()` of the non-linear table lookup, the capacity of the non-linear table it finds - size. The return values digest and result are both 32 bytes long byte strings.

As the logic of `hashimoto()` is very complex including repeatedly hash operations. Here we briefly describe it from the perspective of data structure changes: The code flow is as below:

- First, the `hashimoto()` function merges hash and nonce arguments into a 40-byte array, taking its SHA-512 hash value as the seed with a length of 64 bytes.
- Then `seed[]` is converted into an array `mix[]` with each element of uint32 size.
- Next, the `lookup()` is constantly called `lookup()` in a loop to extract the uint32 element type array from the external dataset, and mix the unknown data into the `mix[]` array. The number of cycles can be set with parameters and is currently set to 64 times. In each loop, the change generates the parameter index, so that each

time the array that is called by the `lookup()` is different. The way to mix data here is a vector-like XOR operation developed from the FNV algorithm.

- After mixing data is completed, a completely different array `mix[]` with `uint32` length are obtained.
- Finally, the collapsed `mix[]` is directly converted into a byte array of length 8 from a `uint32` array of length 8. This is the return value - digest; also, the previous `seed[]` array is merged with the digest and the SHA- 256 hash value to get a byte array of length 32, which is the return value - result.

Finally, after a series of hash operations, `hashimoto()` returns two-byte arrays of length 32 - `digest[]` and `result[]`. As we analysed earlier, in the `ethash.mine()` function, the two return values of `hashimotoFull()`, the result and target, will be directly compared in the form of `big.int` integer; if the requirement is met, the digest will be stored in `Header.MixDigest` as the hash of SHA3-256 (256 Bits), which can be verified later by `Ethash.VerifySeal()`.

lookup()

In the `hashimoto()` function, the function type input parameter `lookup()` actually implements a hash operation in the non-linear table lookup mode. The `lookup()` takes the input parameter as key, and extracts 64 bytes of data from the related data set according to the defined logic and returns the hash value. The returned data will be used by other hash operations in the `hashimoto()` function.

The hashing by nonlinear table lookup is one implementation of the hash functions. It is widely used in the core of the Ethash algorithm. The nonlinear table used is defined into two structures, `cache{}` and `dataset{}`. The main difference between these two is the size of the table and the use case: the data size in `dataset{}` is much larger, so it will be called by `hashimotoFull()` to serve `Ethash.Seal()`; the size of `cache{}` is relatively small, which will be called by `hashimotoLight()` and will serve `Ethash.VerifySeal()`.

3.3.5.3 Memory Mapping

Since the dataset `cache{}` and `dataset{}` used in the Ethash (PoW) algorithm are too large, it is necessary to store it on disk as a file. Also, because these files are too large, they need to be read into the memory at one time (because the use of them is to randomly extract a piece of data), using memory mapping can greatly reduce I/O burden. In the `cache{}` and `dataset{}` structures, there is a `map` object to achieve the memory mapping, and a system file object dump to the open disk file.

3.3.5.4 Ethash Algorithm Summary

If we take the whole process of generating result[] as a conceptual function RAND(), generating an array in a more random and more evenly distributed way is the key to the security of the entire Ethash algorithm. So the Ethash consensus algorithm applies a very complex series of operations, including multiple different hash function operations such as:

- Widely use of SHA3 hash functions, including 256-bit and 512-bit forms, to hash data (groups), or to serve as a prototype for other more complex hash computing.
- The lookup() function provides a hash function for non-linear table lookups. The related dataset{} and cache{} both have huge size, so the hash function is also widely used in the data generation process.

Chapter 4 Design

4.1 Private Network

To research the underlying protocols of Ethereum platform mentioned above, it is necessary to run Ethereum client (geth) on a private test network. As it takes Ethereum to initiate transactions on the Ethereum's public chain. By modifying the configuration, we can build a private chain of Ethereum in the local machine, because it has nothing to do with the public chain, neither to synchronize the huge data of the public chain, nor to spend money to buy Ether, which satisfies research requirements. The tools and languages used are as below:

- go-ethereum
- Golang compiler
- JetBrains GoLand 2018.1.2

The configurations that need to be specified in the private blockchain are:

- Custom initial file
- Custom data directory
- Custom network ID

The requirements for each node to join the same private blockchain [21]:

- Each node will use a distinct data directory to store the database and the wallet.
- Each node must initialize a blockchain based on the same genesis file.

- Each node must join the same network id different from the one reserved by Ethereum (0 to 3 are already reserved).
- The port numbers must be different if different nodes are installed on the same computer.

Based on these, the configuration of our private network is described as following [21]:

4.1.1 Genesis file

Each blockchain starts with a genesis block that is used to initialize the blockchain and defines the terms and conditions to join the network. The genesis block is called “CustomGenesis.json” with the following content:

```
{
  "config": {
    "chainId": 987,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "difficulty": "0x400",
  "gasLimit": "0x8000000",
  "alloc": {}
}
```

Among the parameters, the most important ones are the following:

- difficulty: if the value is low, the transactions will be quickly processed within our private blockchain.
- gasLimit: define the limit of Gas expenditure per block. The gasLimit is set to the maximum to avoid being limited to our tests.

4.1.2 Command Line Flags

There are some required command line options (also known as "flags") to make sure the network is private. Flags used in this private network are following:

```
-- identity
```

This is to set an identity for the node, making it easier to recognize it in the list of endpoints.

```
-- init
```

This is to initialize the genesis block mentioned above

```
-- datadir
```

This is to set data directory for the databases and keystore

```
-- networkid
```

Network identifier (integer, 1=Frontier, 2=Morden (disused), 3=Ropsten, 4=Rinkeby)
(default: 1)

```
--rpc
```

This flag can activate the RPC interface on your node. It is usually activated by default in the geth

```
--rpcport
```

This is to set HTTP-RPC server listening port (default: 8545)

```
--rpcapi
```

API's offered over the HTTP-RPC interface, we will use this API to gather information of our private network

```
--port
```

This is to set Network listening port (default: 30303)

```
--ipcdisable
```

This is to disable the IPC-RPC server, as we are using HTTP-RPC server.

```
--nodiscover
```

This is to disable the discovery mechanism. As the discovery protocol is not working on a private blockchain.

4.1.3 Nodes

The private chain is set with three nodes, two nodes are designed to communicate and have a transaction, the other node(miners) will validate and propagate transactions and apply the notion of consensus within the blockchain.

The network structure is shown below:

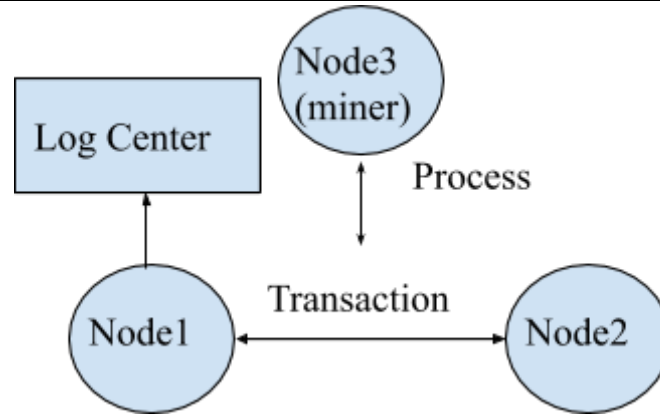


Fig.29 Private Network

The nodes are configured as follows:

Node1

```
geth --datadir=~\Ethereum1" init "~\CustomGenesis.json"
geth --identity "Node1" --networkid 42 --datadir=~\Ethereum1"
--rpc --rpcport "8041" --rpcapi
"admin,db,eth,net,web3,miner,txpool" --port "30303" --ipcdisable
--unlock 0 --nodiscover
geth attach http://127.0.0.1:8041
```

Node2

```
geth --datadir=~\Ethereum2" init "~\CustomGenesis.json"
geth --identity "Node2" --networkid 42 --datadir=~\Ethereum2"
--rpc --rpcport "8042" --rpcapi
"admin,db,eth,net,web3,miner,txpool" --port "30304" --ipcdisable
--unlock 0 --nodiscover
geth attach http://127.0.0.1:8042
```

Node3

```
geth --datadir=~\Ethereum3" init "~\CustomGenesis.json"
geth --identity "Node3" --networkid 42 --datadir=~\Ethereum3"
--rpc --rpcport "8043" --rpcapi
"admin,db,eth,net,web3,miner,txpool" --port "30305" --ipcdisable
--unlock 0 --nodiscover
```



```
geth attach http://127.0.0.1:8043
```

4.2 Http Server

4.2.1 Log Center

Also, to find out how p2p network and proof of work works with the underlying protocols, the research design is to set up a log center to receive the information from each transaction at the very beginning. This means we need to modify the source code to send specific information to the log center in related communication levels, the log center would then record all the events happened in order.

Once we get the record of the event of a complete transaction in Ethereum network, it is feasible to analyse the detailed implementation of Ethereum underlying protocols. **log15** is implemented to achieve the function, it is an open source tool to provide an opinionated, simple toolkit for best-practice logging in Go (golang) that is both human and machine readable. The **log15** is also implemented by official Go implementation of the Ethereum protocol(go-ethereum) with some minor modifications required by the go-ethereum codebase. Based on this, the log center is designed to be accessible as a http web server, so that we can obtain related logs by accessing the local web.

4.2.2 JSON API

With the log center recording the main events, we also want to know the detailed information during an event. The go-ethereum has provided official JSON APIs over the Geth RPC endpoints, so that we can access HTTP server to get JSON results.

The APIs we are using for research testing are listed as below:

```
admin_nodeInfo  
admin_peers()  
txpool_content  
eth_protocolVersion  
eth_syncing  
eth_hashrate  
eth_gasPrice  
eth_getTransactionCount  
eth_getTransactionByHash  
eth_getWork
```

The overall local HTTP server mapping rules are tabled as below [23]:

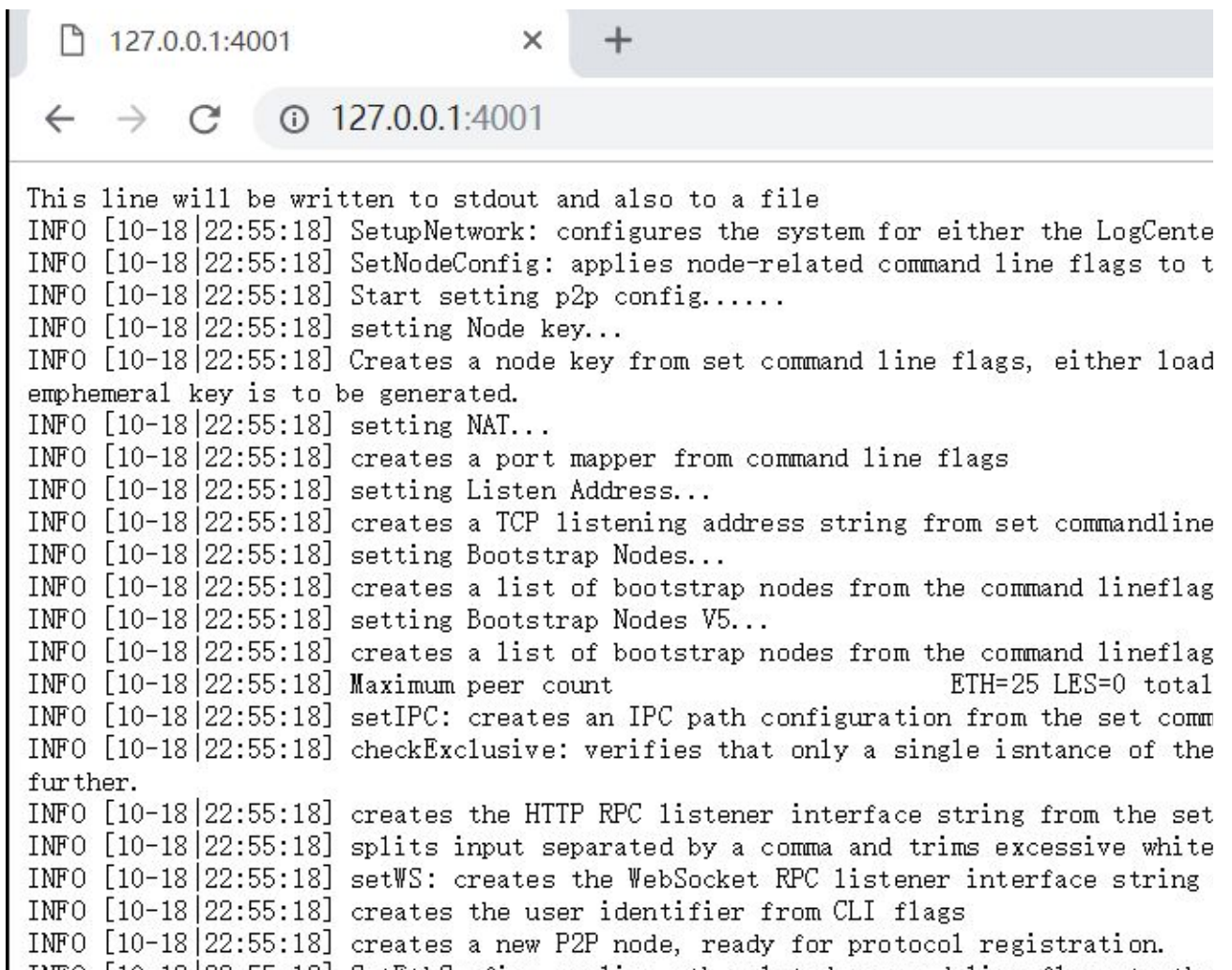
Mapp	Parameters	Return
/	none	Returns the execution logs of the running Geth node
/node	none	Returns all the information known about the running Geth node
/peers	none	Returns all the information known about the connected nodes at the networking granularity
/tx/tx_count	20 Bytes - address block parameter - the string "latest", "earliest" or "pending"	Returns the number of transactions sent from an address
/tx/tx_pending	none	Returns the exact details of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.
/eth/version	none	Returns the current Ethereum protocol version
/eth/syncing	none	Returns an object with data about the sync status or false.
/eth/hashrate	none	Returns the number of hashes per second that the node is mining with.
/eth/gasPrice	none	Returns the current price per gas in wei
/eth/getWork	none	Returns the hash of the current block, the seedHash, and the boundary condition to be met ("target").
/eth/getTransactionByHash	32 Bytes - hash of a transaction	A transaction object, or null when no transaction was found

Table.2 Http Server Mapping Rules

Chapter 5 Analysis

5.1 Peer-to-peer network

After we start the three nodes in the private network, the overview log of the whole p2p networking is shown below:



```

This line will be written to stdout and also to a file
INFO [10-18|22:55:18] SetupNetwork: configures the system for either the LogCente
INFO [10-18|22:55:18] SetNodeConfig: applies node-related command line flags to t
INFO [10-18|22:55:18] Start setting p2p config.....
INFO [10-18|22:55:18] setting Node key...
INFO [10-18|22:55:18] Creates a node key from set command line flags, either load
emphemeral key is to be generated.
INFO [10-18|22:55:18] setting NAT...
INFO [10-18|22:55:18] creates a port mapper from command line flags
INFO [10-18|22:55:18] setting Listen Address...
INFO [10-18|22:55:18] creates a TCP listening address string from set commandline
INFO [10-18|22:55:18] setting Bootstrap Nodes...
INFO [10-18|22:55:18] creates a list of bootstrap nodes from the command lineflag
INFO [10-18|22:55:18] setting Bootstrap Nodes V5...
INFO [10-18|22:55:18] creates a list of bootstrap nodes from the command lineflag
INFO [10-18|22:55:18] Maximum peer count ETH=25 LES=0 total
INFO [10-18|22:55:18] setIPC: creates an IPC path configuration from the set comm
INFO [10-18|22:55:18] checkExclusive: verifies that only a single isntance of the
further.
INFO [10-18|22:55:18] creates the HTTP RPC listener interface string from the set
INFO [10-18|22:55:18] splits input separated by a comma and trims excessive white
INFO [10-18|22:55:18] setWS: creates the WebSocket RPC listener interface string
INFO [10-18|22:55:18] creates the user identifier from CLI flags
INFO [10-18|22:55:18] creates a new P2P node, ready for protocol registration.

```

Fig.30 Execution Log

The detailed analysis of p2p communication is given in the previous chapter. Here we briefly go through how Ethereum client(geth) initializes and starts p2p networking based on the execution log.

The first stage is to **configure and set up the network**. As shown below, the system first configures log center to record all following events, then it sets related configurations in order, such as p2p config, NAT protocol, TCP listening address and so on. Finally, it creates a new P2P node to be ready for protocol registration.

```
INFO [10-18|23:35:46] SetupNetwork: configures the system for either the LogCenter net or
some test network.
INFO [10-18|23:35:46] SetNodeConfig: applies node-related command line flags to the
config.
INFO [10-18|23:35:46] Start setting p2p config.....
INFO [10-18|23:35:46] setting Node key...
INFO [10-18|23:35:46] Creates a node key from set command line flags, either loading it
from a file or as a specified hex value. If neither flags were provided, this method
returns nil and an ephemeral key is to be generated.
INFO [10-18|23:35:46] setting NAT...
INFO [10-18|23:35:46] creates a port mapper from command line flags
INFO [10-18|23:35:46] setting Listen Address...
INFO [10-18|23:35:46] creates a TCP listening address string from set commandline flags.
INFO [10-18|23:35:46] setting Bootstrap Nodes...
INFO [10-18|23:35:46] creates a list of bootstrap nodes from the command lineflags,
reverting to pre-configured ones if none have been specified.
INFO [10-18|23:35:46] setting Bootstrap Nodes V5...
INFO [10-18|23:35:46] creates a list of bootstrap nodes from the command lineflags,
reverting to pre-configured ones if none have been specified.
INFO [10-18|23:35:46] Maximum peer count ETH=25 LES=0 total=25
INFO [10-18|23:35:46] setIPC: creates an IPC path configuration from the set command line
flags,returning an empty string if IPC was explicitly disabled, or the set path.
INFO [10-18|23:35:46] checkExclusive: verifies that only a single instance of the
provided flags was set by the user. Each flag might optionally be followed by a string
type to specialize it further.
INFO [10-18|23:35:46] creates the HTTP RPC listener interface string from the setcommand
line flags, returning empty if the HTTP endpoint is disabled
INFO [10-18|23:35:46] splits input separated by a comma and trims excessive white space
from the substrings.
INFO [10-18|23:35:46] setWS: creates the WebSocket RPC listener interface string from the
setcommand line flags, returning empty if the HTTP endpoint is disabled.
INFO [10-18|23:35:46] creates the user identifier from CLI flags
INFO [10-18|23:35:46] creates a new P2P node, ready for protocol registration.
```

The second stage is basically **protocol and service registration**. As shown below, the system starts to configure eth protocol, transaction pool, ethash algorithm, ssh protocol and so on (marked below). Then the node is live and ready to communicate with other nodes, the p2p server will then create the discovery databases for node discover protocol.

```

INFO [10-18|23:35:46] SetEthConfig: applies eth-related command line flags to the config.
INFO [10-18|23:35:46] checkExclusive: verifies that only a single instance of the
provided flags was set by the user. Each flag might optionally be followed by a string
type to specialize it further.
INFO [10-18|23:35:46] checkExclusive: verifies that only a single instance of the
provided flags was set by the user. Each flag might optionally be followed by a string
type to specialize it further.
INFO [10-18|23:35:46] checkExclusive: verifies that only a single instance of the
provided flags was set by the user. Each flag might optionally be followed by a string
type to specialize it further.
INFO [10-18|23:35:46] checkExclusive: verifies that only a single instance of the
provided flags was set by the user. Each flag might optionally be followed by a string
type to specialize it further.
INFO [10-18|23:35:46] setEtherbase: retrieves the etherbase either from the directly
specified command line flags or from the keystore if CLI indexed.
INFO [10-18|23:35:46] setGPO
INFO [10-18|23:35:46] setTxPool
INFO [10-18|23:35:46] setEthash
INFO [10-18|23:35:46] makeDatabaseHandles: raises out the number of allowed file handles
per process for Geth and returns half of the allowance to assign to the database.
INFO [10-18|23:35:46] SetShhConfig: applies ssh-related command line flags to the config.
INFO [10-18|23:35:46] SetDashboardConfig: applies dashboard related command line flags to
the config.
INFO [10-18|23:35:46] RegisterEthService: adds an Ethereum client to the stack.
INFO [10-18|23:35:46] Register: injects a new service into the node's stack. The service
created by the passed constructor must be unique in its type with regard to sibling ones.
INFO [10-18|23:35:46] startNode.....
INFO [10-18|23:35:46] startNode: Start up the node itself
INFO [10-18|23:35:46] Start: create a live P2P node and starts running it.
INFO [10-18|23:35:46] Initialize the p2p server. This creates the node key and discovery
databases.
INFO [10-18|23:35:46] Starting peer-to-peer node
instance=Geth/Node1/v1.8.8-unstable/windows-amd64/go1.10

```

The third stage is more about **chain database configuration**. As shown below, the system creates the chain database based on given data directory, consensus engine is then created for mining, cache and DAG of ethash are also configured for disk storage. Then the node is ready to

```

INFO [10-18|23:35:46] creates a new Ethereum object
INFO [10-18|23:35:46] CreateDB: creates the chain database.
INFO [10-18|23:35:46] Allocated cache and file handles
database=C:\\Users\\K\\AppData\\Roaming\\Ethereum1\\geth\\chaindata cache=768
handles=1024
INFO [10-18|23:35:47] Initialised chain configuration          config="{ChainID: 987

```

```
Homestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: 0 Byzantium:
<nil> Constantinople: <nil> Engine: unknown}"
INFO [10-18|23:35:47] CreateConsensusEngine: creates the required type of consensus
engine instance for an Ethereum service
INFO [10-18|23:35:47] If proof-of-authority is requested, set it up
INFO [10-18|23:35:47] Otherwise assume proof-of-work
INFO [10-18|23:35:47] Disk storage enabled for ethash caches
dir=C:\\Users\\K\\AppData\\Roaming\\Ethereum1\\geth\\ethash count=3
INFO [10-18|23:35:47] Disk storage enabled for ethash DAGs
dir=C:\\Users\\K\\AppData\\Ethash count=2
INFO [10-18|23:35:47] Initialising Ethereum protocol versions="[63 62]"
network=42
INFO [10-18|23:35:47] Loaded most recent local header number=452
hash=52af36''2f0cff td=63719066
INFO [10-18|23:35:47] Loaded most recent local full block number=452
hash=52af36''2f0cff td=63719066
INFO [10-18|23:35:47] Loaded most recent local fast block number=452
hash=52af36''2f0cff td=63719066
INFO [10-18|23:35:47] Loaded local transaction journal transactions=1 dropped=0
INFO [10-18|23:35:47] Regenerated local transaction journal transactions=1 accounts=1
INFO [10-18|23:35:47] NewProtocolManager: returns a new Ethereum sub protocol manager.
INFO [10-18|23:35:47] newPeerSet: creates a new peer set to track the active
participants.
WARN [10-18|23:35:47] Blockchain not empty, fast sync disabled
INFO [10-18|23:35:47] prepare header for mining
INFO [10-18|23:35:47] Create the new block to seal with the consensus engine
INFO [10-18|23:35:47] sends a new work task to currently live miner agents.
INFO [10-18|23:35:47] Gather the protocols and start the freshly assembled P2P server
INFO [10-18|23:35:47] Start: starts running the server.
INFO [10-18|23:35:47] Starting P2P networking
```

The last stage is to start **running the application layer**, in this stage, the node is capable to start application protocols, such as Ethereum protocol, RLPx protocol and all API related protocols.

```
INFO [10-18|23:35:47] startListening
INFO [10-18|23:35:47] Launch the TCP listener.
INFO [10-18|23:35:47] update keeps track of the downloader events
Miner=0x0000000000000000000000000000000000000000000000000000000000000000 mining=0
INFO [10-18|23:35:47] Start Ethereum protocol
INFO [10-18|23:35:47] Lastly start the configured RPC interfaces
INFO [10-18|23:35:47] startRPC: start all the various RPC endpoint during node startup.
INFO [10-18|23:35:47] startInProc: initializes an in-process RPC endpoint.
INFO [10-18|23:35:47] Mined broadcast loop
INFO [10-18|23:35:47] syncer: periodically synchronise with the network,
INFO [10-18|23:35:47] RLPx listener up
self="enode://3ca61227e873e5598bf740fd4d3e7a1a0e6120ca779a3ac4120d38f9885af218a51ff57b620"
```

```
426dec32c22b9bff66c2b85f820f19af570a80a7cae6821cd6611@[::]:30303?discport=0"
INFO [10-18|23:35:47] startIPC: initializes and starts the IPC RPC endpoint.
INFO [10-18|23:35:47] startHTTP: initializes and starts the HTTP RPC endpoint.
Unlocking account 0 | Attempt 1/3
!! Unsupported terminal, password will be echoed.
Passphrase: INFO [10-18|23:35:47] HTTP endpoint opened
url=http://127.0.0.1:8041 cors= vhosts=localhost
INFO [10-18|23:35:47] startWS: initializes and starts the websocket RPC endpoint.
INFO [10-18|23:35:47] All API endpoints started successfully
INFO [10-18|23:35:47] Finish initializing the startup
```

After the p2p networking is start up successfully, we can then check the node information using HTTP server described above, the results are shown below:



```
{
  "jsonrpc": "2.0",
  "id": 42,
  "result": {
    "id": "3ca61227e873e5598bf740fd4d3e7a1a0e6120ca779a3ac4120d38f9885af218a51f",
    "name": "Geth/Node1/v1.8.8-unstable/windows-amd64/go1.10",
    "enode": "enode://3ca61227e873e5598bf740fd4d3e7a1a0e6120ca779a3ac4120d38f98",
    "ip": "::",
    "ports": {
      "discovery": 0,
      "listener": 30303
    },
    "listenAddr": "[::]:30303",
    "..."
  }
}
```

Fig.31 Node Information

The results above give a detailed description of the running node, including id, name, ip, port and protocols. As this is a p2p network, we can also check peers of the running node, and the results are shown below:


```

{
  "jsonrpc": "2.0",
  "id": 42,
  "result": [
    {
      "id": "2ecc7ea4a727d8c490fc5613099af818ffdaa6cf82a40513b180cc81ddba3daabe8b20046191",
      "name": "Geth/Node2/v1.8.17-stable-8bbe7207/windows-386/go1.11.1",
      "caps": [
        "eth/62",
        "eth/63"
      ],
      "network": {
        "localAddress": "192.168.1.101:59556",
        "remoteAddress": "192.168.1.101:30304",
        "inbound": false,
        "trusted": false,
        "static": true
      },
      "protocols": {
        "eth": {
          "version": 63,
          "difficulty": 63719066,
          "head": "0x52af366d10fcafe811d62606dfeed146666af561165ec18b53cb2475772f0cf1"
        }
      }
    },
    {
      "id": "b07ea7f39a02d531186a35937a11ddb8fc9e1af53f92009c174ec6b83518a29afb522c11d5c",
      "name": "Geth/Node3/v1.8.17-stable-8bbe7207/windows-386/go1.11.1",
      "caps": [
        "eth/62",
        "eth/63"
      ],
      "network": {
        "localAddress": "192.168.1.101:59554",
        "remoteAddress": "192.168.1.101:30305",

```

Fig.32 Peers Information

The results above give a detailed description of peers including id, name, ip, port and protocols.

5.2 Mining and Transaction

As described above, in the private chain, two nodes are designed to make a transaction and the third node will be the miner to validate and propagate the transaction. The following shows how mining contributes to a transaction.

Check balances

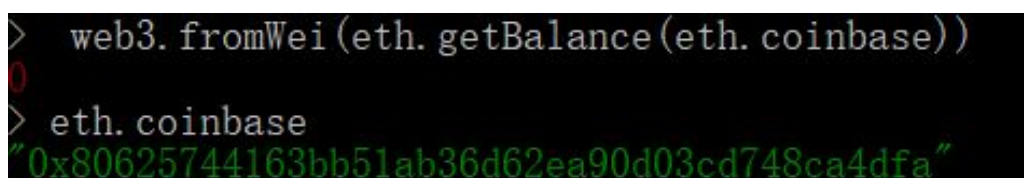
For node1, the balance is as following:



```
> web3.fromWei(eth.getBalance(eth.coinbase))
45
```

Fig.33 Node 1 Screenshot

For node2, the balance is as following:



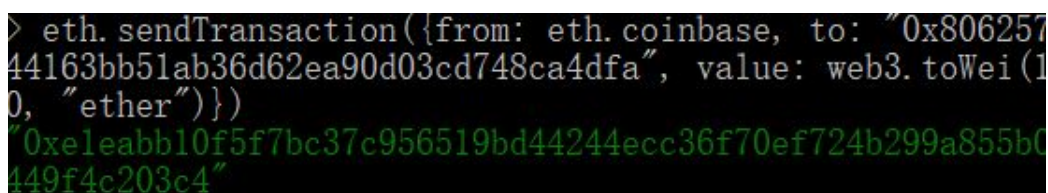
```
> web3.fromWei(eth.getBalance(eth.coinbase))
0
> eth.coinbase
"0x80625744163bb51ab36d62ea90d03cd748ca4dfa"
```

Fig.34 Node 2 Screenshot

Send ethers from node1 to node2

Through the Geth console, now we send 10 ethers from node1 to node2, Then we get a transaction hash:

"0xe1eabb10f5f7bc37c956519bd44244ecc36f70ef724b299a855b0449f4c203c4"



```
> eth.sendTransaction({from: eth.coinbase, to: "0x80625744163bb51ab36d62ea90d03cd748ca4dfa", value: web3.toWei(10, "ether")})
"0xe1eabb10f5f7bc37c956519bd44244ecc36f70ef724b299a855b0449f4c203c4"
```

Fig.35 Transaction Screenshot

Here, "0x80625744163bb51ab36d62ea90d03cd748ca4dfa" is the default address of node 2.

As now there is no miner working, this transaction will be pending to be processed and the balance of node2 is still zero when we visit the URL in the screenshot below, the JSON API will return the pending transaction, which is the transaction we just sent.

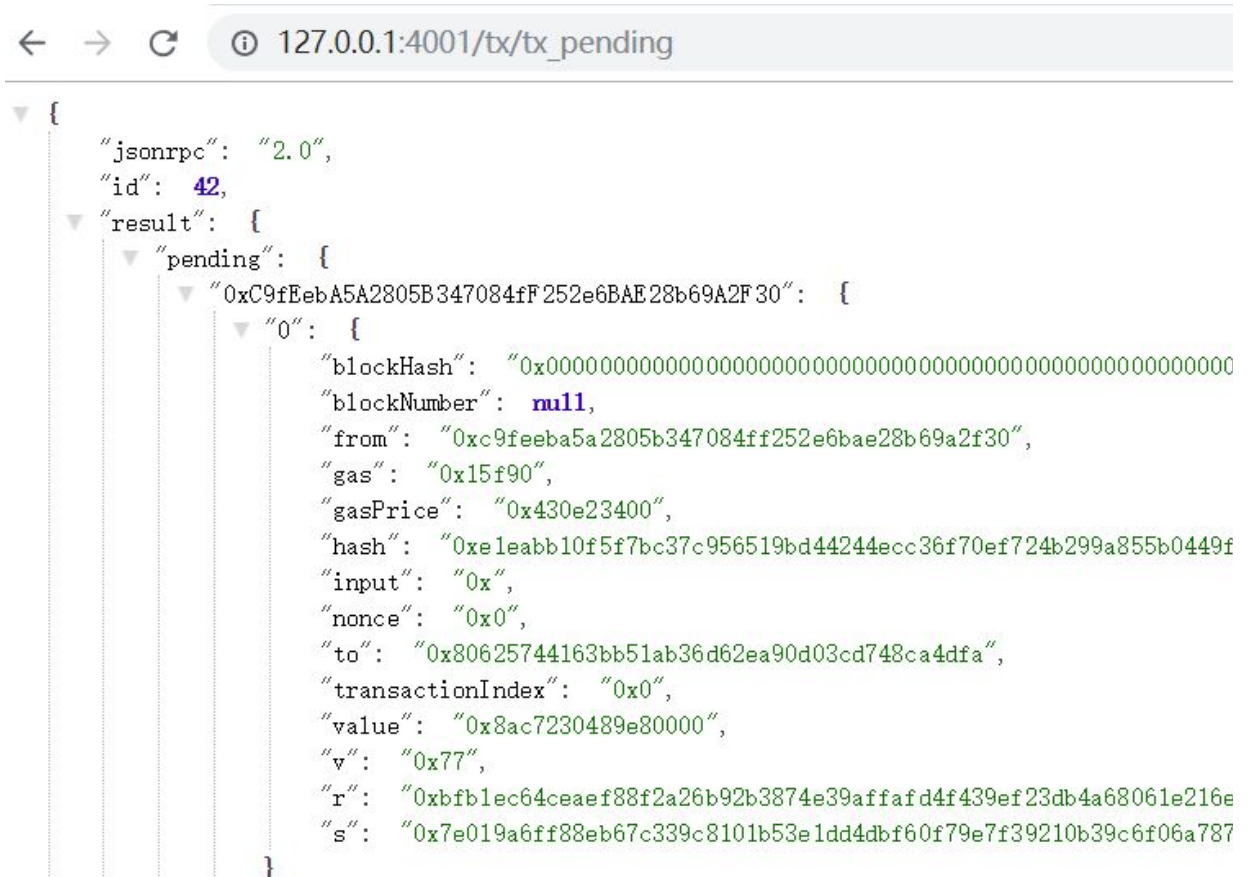


Fig.36 Pending Screenshot

Mining for processing transaction

Now we make node3 start mining so that the transaction can be valid and praprocated. After node 3 starts mining, We can see there is no more pending transaction.

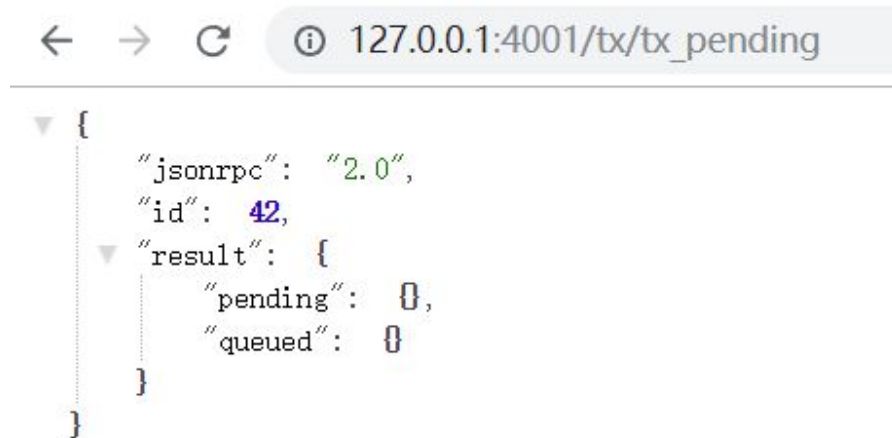


Fig.37 Pending Screenshot After Mining

We can get the detailed transaction receipt with the given transaction hash, as shown below, the results include sender address, receiver address, gas price and so on.

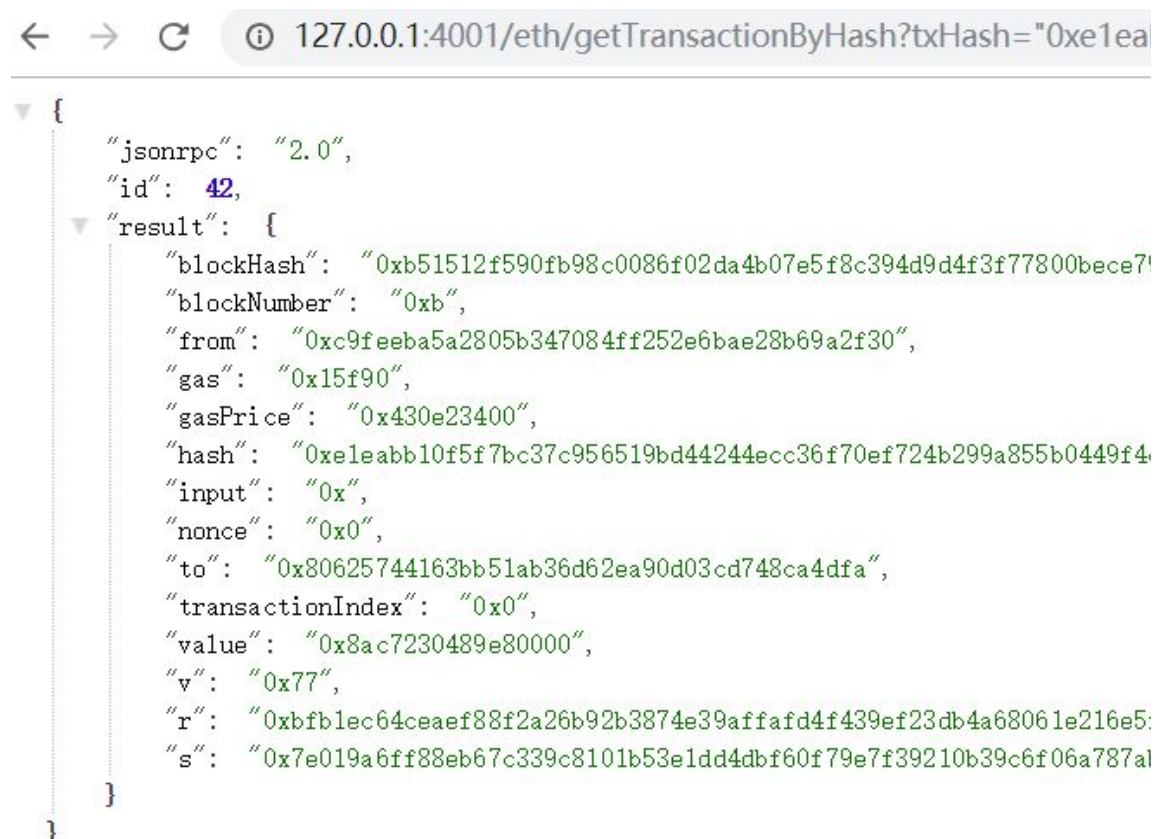


Fig.38 Transaction Receipt

5.3 Proof of Work

When node 3 starts mining, we also get the execution log of the mining process, the detailed analysis of proof of work and mining is given in the previous chapter. Here we briefly go through how Ethereum client(eth) implements mining based on the execution log.

Before mining, the running node keeps retrieving the current hash of the network status, the frequency is three times per ten minutes as shown below:

```
INFO [10-19|08:29:08] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:08] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:18] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:18] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:18] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:28] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:28] Head retrieves a copy of the current head hash and total difficulty of thepeer.
INFO [10-19|08:29:28] Head retrieves a copy of the current head hash and total difficulty of thepeer.
```

After mining is started, a new block will be created for committing mining work, this work will be done by miner agents with a series of hash computing operations.

```
INFO [10-19|08:29:28] Starting mining operation,
Worker=0xC9fEebA5A2805B347084fF252e6BAE28b69A2F30
INFO [10-19|08:29:28] prepare header for mining
INFO [10-19|08:29:28] Create the new block to seal with the consensus engine
INFO [10-19|08:29:28] Commit new mining work number=1 txs=0 uncles=0
elapsed=0s
INFO [10-19|08:29:28] sends a new work task to currently live miner agents.
INFO [10-19|08:29:28] Started ethash search for new nonces seed=4602246591336105676
INFO [10-19|08:29:28] Started ethash search for new nonces seed=6440097846019695234
INFO [10-19|08:29:28] Started ethash search for new nonces seed=3645540802371180141
INFO [10-19|08:29:28] Started ethash search for new nonces seed=6628262593409525697
INFO [10-19|08:29:28] Calculate the number of theoretical rows
INFO [10-19|08:29:28] Combine header+nonce into a 64 byte seed
INFO [10-19|08:29:28] Calculate the number of theoretical rows
INFO [10-19|08:29:28] Start the mix with replicated seed
INFO [10-19|08:29:28] Mix in random dataset nodes
```

```
INFO [10-19|08:29:28] Combine header+nonce into a 64 byte seed
```

Finally, the new block is successfully mined, and this will be broadcasting in the whole network.

```
INFO [10-19|08:29:45] Successfully sealed new block          number=2
hash=122146'b6fe08
INFO [10-19|08:29:45] ? mined potential block              number=2
hash=122146'b6fe08
INFO [10-19|08:29:45] BroadcastBlock: either propagate a block to a subset of it's peers,
orwill only announce it's availability (depending what's requested).
INFO [10-19|08:29:45] prepare header for mining
INFO [10-19|08:29:45] SendNewBlock: propagates an entire block to a remote peer.
INFO [10-19|08:29:45] Create the new block to seal with the consensus engine
```

Chapter 6 Conclusion and Future Work

This project provides us with a chance to in-depth study the most open blockchain platform Ethereum in a way of both theory and practice. Based on the open resource including documents and source code of Ethereum, we build an overall understanding of Ethereum platform, we mainly analysis core protocols of the three most important underlying modules: Elliptic Curve Digital Signature, Peer-to-peer Communication, Proof-of-Work and Mining.

For Elliptic Curve Digital Signature, we find that the digital signatures in Ethereum all use elliptic curve digital encryption algorithm (ECDSA), its theoretical basis is elliptic curve cryptography (ECC), and the theoretical basis of ECC is the point multiplication formula $Q = dP$ where the private key d almost cannot be deciphered. Compared with RSA based on large prime decomposition, ECC only needs shorter public key when providing the same security level. The elliptic curve digital signature algorithm used in Ethereum is implemented by libsecp256k1 library, which is an optimized C++ library for a specific elliptic curve secp256k1 adopted by the Bitcoin system. Also, the type of Address used in Ethereum, such as the address of each account, all comes from the public key of the elliptic curve digital signature.

For Peer-to-peer communication, we focus on three protocols: Devp2p application protocol, RLPx transport protocol, Node Discovery protocol. Ethereum network implements XOR distance, binary prefix tree, K-bucket, etc., which is quite complicated in structure, but it is indeed much faster than the bitcoin on the node route. For Proof-of-Work and Mining, in Ethereum, the term “mining a new block” actually consists of two parts. In the first stage, all the data members of the new block are assembled, including the transaction list txs, the uncle block uncles, etc, and all transactions have been executed, and the status of each account is updated; the second

stage of the block is to seal, the block without a successful seal cannot be broadcast to other nodes. The computing resources consumed in the second phase far exceed the first phase. The seal process is performed by a consensus algorithm cluster, including Ethash and Clique. The former is actually used in the product environment, and the latter is used for the test network (testnet). The Ethash algorithm (PoW) define the winners of the mining block based on computing power. The algorithm uses multiple hash functions to mining machine such as ASIC with extremely high computing resource consumption.

Due to lack of time and complexity of Ethereum platform, many analysis have been left for the future. one is that Ethereum has more than one consensus algorithms, PoW(Proof-of-Work) is just one of them. The are PoA(Proof-of-Authority) and PoS(Proof-of-Stack), Ethereum is now switching from Proof-of-Work mining to 'Proof of Stake', in the future we can research more about these two algorithms based on existing knowledge. Considering smart contract is the most important feature of Ethereum platform, another interesting area could be smart contract, as this project focus on underlying technology, the next step to follow this work is to research upper layer application.

Appendixes

Appendix A

```
// Signer encapsulates transaction signature handling. Note that this interface is not a
// stable API and may change at any time to accommodate new protocol rules.
type Signer interface {
    // Sender returns the sender address of the transaction.
    Sender(tx *Transaction) (common.Address, error)
    // SignatureValues returns the raw R, S, V values corresponding to the
    // given signature.
    SignatureValues(tx *Transaction, sig []byte) (r, s, v *big.Int, err error)
    // Hash returns the hash to be signed.
    Hash(tx *Transaction) common.Hash
    // Equal returns true if the given signer is the same as the receiver.
    Equal(Signer) bool
}

// EIP155Transaction implements Signer using the EIP155 rules.
type EIP155Signer struct {
    chainId, chainIdMul *big.Int
}

func NewEIP155Signer(chainId *big.Int) EIP155Signer {
    if chainId == nil {
        chainId = new(big.Int)
    }
    return EIP155Signer{
        chainId: chainId,
        chainIdMul: new(big.Int).Mul(chainId, big.NewInt(2)),
    }
}

func (s EIP155Signer) Equal(s2 Signer) bool {
    eip155, ok := s2.(EIP155Signer)
    return ok && eip155.chainId.Cmp(s.chainId) == 0
}

var big8 = big.NewInt(8)

func (s EIP155Signer) Sender(tx *Transaction) (common.Address, error) {
    if !tx.Protected() {
        return HomesteadSigner{}.Sender(tx)
    }
    if tx.ChainId().Cmp(s.chainId) != 0 {
        return common.Address{}, ErrInvalidChainId
    }
    V := new(big.Int).Sub(tx.data.V, s.chainIdMul)
    V.Sub(V, big8)
    return recoverPlain(s.Hash(tx), tx.data.R, tx.data.S, V, true)
}
```

```

// WithSignature returns a new transaction with the given signature. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (s EIP155Signer) SignatureValues(tx *Transaction, sig []byte) (R, S, V *big.Int, err error) {
    R, S, V, err = HomesteadSigner{}.SignatureValues(tx, sig)
    if err != nil {
        return nil, nil, nil, err
    }
    if s.chainId.Sign() != 0 {
        V = big.NewInt(int64(sig[64] + 35))
        V.Add(V, s.chainId.Mul)
    }
    return R, S, V, nil
}

// Hash returns the hash to be signed by the sender.
// It does not uniquely identify the transaction.
func (s EIP155Signer) Hash(tx *Transaction) common.Hash {
    return rlpHash([]interface{} {
        tx.data.AccountNonce,
        tx.data.Price,
        tx.data.GasLimit,
        tx.data.Recipient,
        tx.data.Amount,
        tx.data.Payload,
        s.chainId, uint(0), uint(0),
    })
}

// HomesteadTransaction implements TransactionInterface using the
// homestead rules.
type HomesteadSigner struct { FrontierSigner }

func (s HomesteadSigner) Equal(s2 Signer) bool {
    _, ok := s2.(HomesteadSigner)
    return ok
}

// SignatureValues returns signature values. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (hs HomesteadSigner) SignatureValues(tx *Transaction, sig []byte) (r, s, v *big.Int, err error) {
    return hs.FrontierSigner.SignatureValues(tx, sig)
}

func (hs HomesteadSigner) Sender(tx *Transaction) (common.Address, error) {
    return recoverPlain(hs.Hash(tx), tx.data.R, tx.data.S, tx.data.V, true)
}

type FrontierSigner struct {}

func (s FrontierSigner) Equal(s2 Signer) bool {
    _, ok := s2.(FrontierSigner)
    return ok
}

```



```

// SignatureValues returns signature values. This signature
// needs to be in the [R || S || V] format where V is 0 or 1.
func (fs FrontierSigner) SignatureValues(tx *Transaction, sig []byte) (r, s, v *big.Int, err error) {
    if len(sig) != 65 {
        panic(fmt.Sprintf("wrong size for signature: got %d, want 65", len(sig)))
    }
    r = new(big.Int).SetBytes(sig[:32])
    s = new(big.Int).SetBytes(sig[32:64])
    v = new(big.Int).SetBytes([]byte{sig[64] + 27})
    return r, s, v, nil
}

// Hash returns the hash to be signed by the sender.
// It does not uniquely identify the transaction.
func (fs FrontierSigner) Hash(tx *Transaction) common.Hash {
    return rlpHash([]interface{} {
        tx.data.AccountNonce,
        tx.data.Price,
        tx.data.GasLimit,
        tx.data.Recipient,
        tx.data.Amount,
        tx.data.Payload,
    })
}

func (fs FrontierSigner) Sender(tx *Transaction) (common.Address, error) {
    return recoverPlain(fs.Hash(tx), tx.data.R, tx.data.S, tx.data.V, false)
}

func recoverPlain(sighash common.Hash, R, S, Vb *big.Int, homestead bool) (common.Address, error) {
    if Vb.BitLen() > 8 {
        return common.Address{}, ErrInvalidSig
    }
    V := byte(Vb.Uint64() - 27)
    if !crypto.ValidateSignatureValues(V, R, S, homestead) {
        return common.Address{}, ErrInvalidSig
    }
    // encode the snature in uncompressed format
    r, s := R.Bytes(), S.Bytes()
    sig := make([]byte, 65)
    copy(sig[32-len(r):32], r)
    copy(sig[64-len(s):64], s)
    sig[64] = V
    // recover the public key from the snature
    pub, err := crypto.Ecrecover(sighash[:], sig)
    if err != nil {
        return common.Address{}, err
    }
    if len(pub) == 0 || pub[0] != 4 {
        return common.Address{}, errors.New("invalid public key")
    }
    var addr common.Address
    copy(addr[:], crypto.Keccak256(pub[1:])[12:])
    return addr, nil
}

```

```
// deriveChainId derives the chain id from the given v parameter
func deriveChainId(v *big.Int) *big.Int {
    if v.BitLen() <= 64 {
        v := v.Uint64()
        if v == 27 || v == 28 {
            return new(big.Int)
        }
        return new(big.Int).SetUint64((v - 35) / 2)
    }
    v = new(big.Int).Sub(v, big.NewInt(35))
    return v.Div(v, big.NewInt(2))
}
```

Appendix B

```
// crypto/crypto.go
func ToECDSAPub(pub []byte) *ecdsa.PublicKey {
    x, y := elliptic.Unmarshal(S256(), pub)
    return &ecdsa.PublicKey{Curve:S256(), X:x, Y:y}
}
func FromECDSAPub(pub *ecdsa.PublicKey) []byte {
    return elliptic.Marshal(S256(), pub.X, pub.Y)
}
```

Appendix C

```
StatusMsg      = 0x00
NewBlockHashesMsg = 0x01
TxMsg          = 0x02
GetBlockHeadersMsg = 0x03
BlockHeadersMsg  = 0x04
GetBlockBodiesMsg = 0x05
BlockBodiesMsg   = 0x06
NewBlockMsg      = 0x07
GetNodeDataMsg   = 0x0d
NodeDataMsg      = 0x0e
GetReceiptsMsg   = 0x0f
ReceiptsMsg      = 0x10
```

Appendix D

```
type udp struct {
    conn      conn
    netrestrict *netutil.Netlist
    priv      *ecdsa.PrivateKey
    ourEndpoint rpcEndpoint

    addpending chan *pending
    gotreply   chan reply
}
```

```

closing chan struct{}
nat nat.Interface

*Table

}

```

Appendix E

```

type Table struct {
    mutex sync.Mutex // protects buckets, bucket content, nursery, rand
    buckets [nBuckets]*bucket // index of known nodes by distance
    nursery []*Node // bootstrap nodes
    rand *mrnd.Rand // source of randomness, periodically reseeded
    ips netutil.DistinctNetSet

    db *nodeDB // database of known nodes
    refreshReq chan chan struct{}
    initDone chan struct{}
    closeReq chan struct{}
    closed chan struct{}

    bondmu sync.Mutex
    bonding map[NodeID]*bondproc
    bondslots chan struct{} // limits total number of active bonding processes

    nodeAddedHook func(*Node) // for testing

    net transport
    self *Node // metadata of the local node
}

```

Appendix F

```

Server.start()
// handshake
    srv.ourHandshake = &protoHandshake{Version: baseProtocolVersion, Name: srv.Name, ID:
discover.PubkeyID(&srv.PrivateKey.Public
Key))}
    for _, p := range srv.Protocols {
        srv.ourHandshake.Caps = append(srv.ourHandshake.Caps, p.cap())
    }
// listen/dial
    if srv.ListenAddr != "" {
        //Start listening for TCP requests, the TCP port needs to be encrypted, and the transmitted information is sensitive, so there is a
        handshake process.
        if err := srv.startListening(); err != nil {
            return err
        }
    }
    srv.loopWG.Add(1)
    go srv.run(dialer)
    srv.running = true

```

```
    return nil
}
```

Appendix G

```
/p2p/server.go
func (srv *Server) setupConn(c *conn, flags connFlag, dialDest *discover.Node) error {
    ...
    if c.id, err = c.doEncHandshake(srv.PrivateKey, dialDest); err != nil {
        srv.log.Trace("Failed RLPx handshake", "addr", c.fd.RemoteAddr(), "conn", c.flags, "err", err)
        return err
    }
    ...
    phs, err := c.doProtoHandshake(srv.ourHandshake)
    ...
}
```

Appendix H

```
/p2p/rlpx.go
func (t *rlpx) doEncHandshake(prv *ecdsa.PrivateKey, dial *discover.Node) (discover.NodeID, error) {
    ...
    if dial == nil {
        sec, err = receiverEncHandshake(t.fd, prv, nil)
    } else {
        sec, err = initiatorEncHandshake(t.fd, prv, dial.ID, nil)
    }
    ...
    t.rw = newRLPXFrameRW(t.fd, sec)
    ..
}
```

Appendix I

```
./p2p/rlpx.go
func receiverEncHandshake(conn io.ReadWriter, prv *ecdsa.PrivateKey, token []byte) (s secrets, err error) {
    authPacket, err := readHandshakeMsg(authMsg, encAuthMsgLen, prv, conn)
    ...
    authRespMsg, err := h.makeAuthResp()
    ...
    if _, err = conn.Write(authRespPacket); err != nil {
        return s, err
    }
    return h.secrets(authPacket, authRespPacket)
}
```

Appendix J

```
/p2p/server.go
func (srv *Server) run(dialstate dialer) {
    ...
    for {
        select {
            case <-srv.quit: ...
            case n := <-srv.addstatic: ...
            case n := <-srv.removestatic: ...
            case op := <-srv.peerOp: ...
            case t := <-taskdone: ...
            case c := <-srv.posthandshake: ...
            case c := <-srv.addpeer: ...
            case pd := <-srv.delpeer: ...
        }
    }
}
```

Appendix K

```
type Miner struct {
    mux *event.TypeMux //
    worker *worker //
    coinbase common.Address //
    mining int32 //
    eth Backend //
    engine consensus.Engine //
    canStart int32 //
    shouldStart int32 //
}
```

Appendix L

```
func New(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine consensus.Engine) *Miner {
    miner := &Miner{
        eth: eth,
        mux: mux,
        engine: engine,
        worker: newWorker(config, engine, common.Address{}, eth, mux),
        canStart: 1,
    }
    miner.Register(NewCpuAgent(eth.BlockChain(), engine))
    go miner.update()

    return miner
}
```

Appendix M

```
func (self *Miner) update() {
    log.Info("update keeps track of the downloader events", "Miner", self.coinbase, "mining", self.mining)
    events := self.mux.Subscribe(downloader.StartEvent{}, downloader.DoneEvent{}, downloader.FailedEvent{})
    out:

    for ev := range events.Chan() {
        switch ev.Data.(type) {
        case downloader.StartEvent:
            atomic.StoreInt32(&self.canStart, 0)
            if self.Mining() {
                self.Stop()
                atomic.StoreInt32(&self.shouldStart, 1)
                log.Info("Mining aborted due to sync")
            }

        case downloader.DoneEvent, downloader.FailedEvent:
            shouldStart := atomic.LoadInt32(&self.shouldStart) == 1

            atomic.StoreInt32(&self.canStart, 1)
            atomic.StoreInt32(&self.shouldStart, 0)
            if shouldStart {
                self.Start(self.coinbase)
            }
            // unsubscribe. we're only interested in this event once
            events.Unsubscribe()
            // stop immediately and ignore all further pending events
            break out
        }
    }
}
```

Appendix N

```
func (self *worker) update() {
    defer self.txSub.Unsubscribe()
    defer self.chainHeadSub.Unsubscribe()
    defer self.chainSideSub.Unsubscribe()

    for {
        // A real event arrived, process interesting content
        select {
        // Handle ChainHeadEvent
        case <-self.chainHeadCh:
            self.commitNewWork()

        // Handle ChainSideEvent
        case ev := <-self.chainSideCh:
            self.uncleMu.Lock()
            self.possibleUncles[ev.Block.Hash()] = ev.Block
            self.uncleMu.Unlock()
        }
    }
}
```

```

// Handle TxPreEvent
case ev := <-self.txCh:
    // Apply transaction to the pending state if we're not mining
    if atomic.LoadInt32(&self.mining) == 0 {
        self.currentMu.Lock()
        acc, _ := types.Sender(self.current.signer, ev.Tx)
        txs := map[common.Address]types.Transactions{acc: {ev.Tx}}
        txset := types.NewTransactionsByPriceAndNonce(self.current.signer, txs)

        self.current.commitTransactions(self.mux, txset, self.chain, self.coinbase)
        self.updateSnapshot()
        self.currentMu.Unlock()
    } else {
        // If we're mining, but nothing is being processed, wake on new transactions
        if self.config.Clique != nil && self.config.Clique.Period == 0 {
            self.commitNewWork()
        }
    }

// System stopped
case <-self.txSub.Err():
    return
case <-self.chainHeadSub.Err():
    return
case <-self.chainSideSub.Err():
    return
}
}
}

```

Appendix O

```

func (self *worker) wait() {
    for {
        mustCommitNewWork := true
        for result := range self.recv {
            atomic.AddInt32(&self.atWork, -1)

            if result == nil {
                continue
            }
            block := result.Block
            work := result.Work

            // Update the block hash in all logs since it is now available and not when the
            // receipt/log of individual transactions were created.
            for _, r := range work.receipts {
                for _, l := range r.Logs {
                    l.BlockHash = block.Hash()
                }
            }
            for _, log := range work.state.Logs() {
                log.BlockHash = block.Hash()
            }
        }
    }
}

```

```

    }
    stat, err := self.chain.WriteBlockWithState(block, work.receipts, work.state)
    if err != nil {
        log.Error("Failed writing block to chain", "err", err)
        continue
    }
    // check if canon block and write transactions
    if stat == core.CanonStatTy {
        // implicit by posting ChainHeadEvent
        mustCommitNewWork = false
    }
    // Broadcast the block and announce chain insertion event
    self.mux.Post(core.NewMinedBlockEvent{Block: block})
    var (
        events []interface{}
        logs    = work.state.Logs()
    )
    events = append(events, core.ChainEvent{Block: block, Hash: block.Hash(), Logs: logs})
    if stat == core.CanonStatTy {
        events = append(events, core.ChainHeadEvent{Block: block})
    }
    self.chain.PostChainEvents(events, logs)

    // Insert the block into the set of pending ones to wait for confirmations
    self.unconfirmed.Insert(block.NumberU64(), block.Hash())

    if mustCommitNewWork {
        self.commitNewWork()
    }
}
}
}

```

Appendix P

```

func (self *worker) commitNewWork() {
    self.mu.Lock()
    defer self.mu.Unlock()
    self.uncleMu.Lock()
    defer self.uncleMu.Unlock()
    self.currentMu.Lock()
    defer self.currentMu.Unlock()

    tstart := time.Now()
    parent := self.chain.CurrentBlock()

    tstamp := tstart.Unix()
    if parent.Time().Cmp(new(big.Int).SetInt64(tstamp)) >= 0 {
        tstamp = parent.Time().Int64() + 1
    }
    // this will ensure we're not going off too far in the future
    if now := time.Now().Unix(); tstamp > now+1 {
        wait := time.Duration(tstamp-now) * time.Second
        log.Info("Mining too far in the future", "wait", common.PrettyDuration(wait))
    }
}

```



```

        time.Sleep(wait)
    }

    num := parent.Number()
    header := &types.Header{
        ParentHash: parent.Hash(),
        Number:      num.Add(num, common.Big1),
        GasLimit:    core.CalcGasLimit(parent),
        Extra:       self.extra,
        Time:        big.NewInt(tstamp),
    }
    // Only set the coinbase if we are mining (avoid spurious block rewards)
    if atomic.LoadInt32(&self.mining) == 1 {
        header.Coinbase = self.coinbase
    }
    log.Info("prepare header for mining")
    if err := self.engine.Prepare(self.chain, header); err != nil {
        log.Error("Failed to prepare header for mining", "err", err)
        return
    }
    // If we are care about TheDAO hard-fork check whether to override the extra-data or not
    if daoBlock := self.config.DAOForkBlock; daoBlock != nil {
        // Check whether the block is among the fork extra-override range
        limit := new(big.Int).Add(daoBlock, params.DAOForkExtraRange)
        if header.Number.Cmp(daoBlock) >= 0 && header.Number.Cmp(limit) < 0 {
            // Depending whether we support or oppose the fork, override differently
            if self.config.DAOForkSupport {
                header.Extra = common.CopyBytes(params.DAOForkBlockExtra)
            } else if bytes.Equal(header.Extra, params.DAOForkBlockExtra) {
                header.Extra = []byte{} // If miner opposes, don't let it use the
reserved extra-data
            }
        }
    }

    // Could potentially happen if starting to mine in an odd state.
    err := self.makeCurrent(parent, header)
    if err != nil {
        log.Error("Failed to create mining context", "err", err)
        return
    }
    // Create the current work task and check any fork transitions needed
    work := self.current
    if self.config.DAOForkSupport && self.config.DAOForkBlock != nil &&
self.config.DAOForkBlock.Cmp(header.Number) == 0 {
        misc.ApplyDAOHardFork(work.state)
    }
    pending, err := self.eth.TxPool().Pending()
    if err != nil {
        log.Error("Failed to fetch pending transactions", "err", err)
        return
    }
    txs := types.NewTransactionsByPriceAndNonce(self.current.signer, pending)
    work.commitTransactions(self.mux, txs, self.chain, self.coinbase)

    // compute uncles for the new block.

```

```

var (
    uncles    []*types.Header
    badUncles []common.Hash
)
for hash, uncle := range self.possibleUncles {
    if len(uncles) == 2 {
        break
    }
    if err := self.commitUncle(work, uncle.Header()); err != nil {
        log.Trace("Bad uncle found and will be removed", "hash", hash)
        log.Trace(fmt.Sprintf(uncle))

        badUncles = append(badUncles, hash)
    } else {
        log.Debug("Committing new uncle to block", "hash", hash)
        uncles = append(uncles, uncle.Header())
    }
}
for _, hash := range badUncles {
    delete(self.possibleUncles, hash)
}
// Create the new block to seal with the consensus engine
log.Info("Create the new block to seal with the consensus engine")
if work.Block, err = self.engine.Finalize(self.chain, header, work.state, work.txs, uncles,
work.receipts); err != nil {
    log.Error("Failed to finalize block for sealing", "err", err)
    return
}
// We only care about logging if we're actually mining.
if atomic.LoadInt32(&self.mining) == 1 {
    log.Info("Commit new mining work", "number", work.Block.Number(), "txs",
work.tcount, "uncles", len(uncles), "elapsed", common.PrettyDuration(time.Since(tstart)))
    self.unconfirmed.Shift(work.Block.NumberU64() - 1)
}
self.push(work)
self.updateSnapshot()
}

```

Appendix Q

```

func (self *CpuAgent) update() {
out:
    for {
        select {
        case work := <-self.workCh:
            self.mu.Lock()
            if self.quitCurrentOp != nil {
                close(self.quitCurrentOp)
            }
            self.quitCurrentOp = make(chan struct{})
            go self.mine(work, self.quitCurrentOp)
            self.mu.Unlock()
        case <-self.stop:
            self.mu.Lock()

```

```

        if self.quitCurrentOp != nil {
            close(self.quitCurrentOp)
            self.quitCurrentOp = nil
        }
        self.mu.Unlock()
        break out
    }
}

```

Appendix R

```

func (self *CpuAgent) mine(work *Work, stop <-chan struct{}) {
    if result, err := self.engine.Seal(self.chain, work.Block, stop); result != nil {
        log.Info("Successfully sealed new block", "number", result.Number(), "hash",
result.Hash())
        self.returnCh <- &Result{work, result}
    } else {
        if err != nil {
            log.Warn("Block sealing failed", "err", err)
        }
        self.returnCh <- nil
    }
}

```

Appendix S

```

func (ethash *Ethash) mine(block *Block, id int, seed uint64, abort chan struct{}, found chan
*Block) {
    var (
        header = block.Header()
        hash    = header.HashNoNonce().Bytes()
        target = new(big.Int).Div(maxUint256, header.Difficulty)
        number = header.Number.Uint64()
        dataset = ethash.dataset(number)
        nonce  = seed
    )
    for {
        select {
        case <-abort:
            ...; return
        default:
            digest, result := hashimotoFull(dataset, hash, nonce) // compute the POW value of this
nonce
            if new(big.Int).SetBytes(result).Cmp(target) <= 0 { // x.Cmp(y) <= 0 means x <= y
                header = types.CopyHeader(header)
                header.Nonce = types.EncodeNonce(nonce)
                header.MixDigest = common.BytesToHash(digest)
                found<- block.WithSeal(header)
                return
            }
        }
        nonce++
    }
}

```

```
}
}
```

Appendix T

```
func hashimotoFull(dataset []uint32, hash []byte, nonce uint64) ([]byte, []byte) {
    lookup := func(index uint32) []uint32 {
        offset := index * hashWords
        return dataset[offset : offset+hashWords]
    }
    return hashimoto(hash, nonce, uint64(len(dataset))*4, lookup)
}
func hashimotoLight( size uint64, cache []uint32, hash []byte, nonce uint64 ) ([]byte, []byte) {
    lookup := func(index uint32) []uint32 {
        rawData := generateDatasetItem(cache, index, keccak512)
        data := make([]uint32, len(rawData)/4)
        for i := 0; i < len(data); i++ {
            data[i] = binary.LittleEndian.Uint32(rawData[i*4:])
        }
        return data
    }
    return hashimoto(hash, nonce, size, lookup)
}
```

Appendix U

```
func hashimoto(hash []byte, nonce uint64, size uint64, lookup func(index uint32) []uint32) ([]byte, []byte) {
    log.Info("Calculate the number of theoretical rows")
    rows := uint32(size / mixBytes)

    // Combine header+nonce into a 64 byte seed
    log.Info("Combine header+nonce into a 64 byte seed")
    seed := make([]byte, 40)
    copy(seed, hash)
    binary.LittleEndian.PutUint64(seed[32:], nonce)

    seed = crypto.Keccak512(seed)
    seedHead := binary.LittleEndian.Uint32(seed)

    // Start the mix with replicated seed
    log.Info("Start the mix with replicated seed")
    mix := make([]uint32, mixBytes/4)
    for i := 0; i < len(mix); i++ {
        mix[i] = binary.LittleEndian.Uint32(seed[i*16*4:])
    }
    // Mix in random dataset nodes
    log.Info("Mix in random dataset nodes")
    temp := make([]uint32, len(mix))

    for i := 0; i < loopAccesses; i++ {
        parent := fnv(uint32(i)^seedHead, mix[i%len(mix)]) % rows
        for j := uint32(0); j < mixBytes/hashBytes; j++ {
```

```
        copy(temp[j*hashWords:], lookup(2*parent+j))
    }
    fnvHash(mix, temp)
}
// Compress mix
log.Info("Compress mix")
for i := 0; i < len(mix); i += 4 {
    mix[i/4] = fnv(fnv(fnv(mix[i], mix[i+1]), mix[i+2]), mix[i+3])
}
mix = mix[:len(mix)/4]

digest := make([]byte, common.HashLength)
for i, val := range mix {
    binary.LittleEndian.PutUint32(digest[i*4:], val)
}
return digest, crypto.Keccak256(append(seed, digest...))
}
```

Bibliography

- [1] Dr. Gavin Wood & Parity Gavin. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER". ethereum.github.io. June,5. 2018.
Available at:
<https://ethereum.github.io/yellowpaper/paper.pdf>
- [2] Vitalik Buterin. "A Next-Generation Smart Contract and Decentralized Application Platform". github.com. June,2. 2013. Available at:
<https://github.com/ethereum/wiki/wiki/White-Paper#history>
- [3]Micah Dameron."An Ethereum Technical Specification". github.com. March. 2018.
Available at:
<https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf>
- [4]Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System".bitcoin.org. 2009.
Available at:
<https://bitcoin.org/bitcoin.pdf>
- [5]"ethereum/devp2p", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/devp2p/blob/master/devp2p.md>.
- [6]"ethereum/devp2p", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/devp2p/blob/master/rlpx.md>.
- [7]"Elliptic curve integrated encryption scheme - Crypto++ Wiki", *Cryptopp.com*, 2018. [Online]. Available:
https://www.cryptopp.com/wiki/Elliptic_curve_integrated_encryption_scheme.
- [8]*Blog.coinfabrik.com*, 2018. [Online]. Available:
https://blog.coinfabrik.com/wp-content/uploads/2016/06/some_comments_on_the_security_of_ecies_with_secp256k1.pdf.
- [9]"Elliptic curve point multiplication", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication. [Accessed: 19- Oct-2018].
- [10]"Kad network", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Kad_network.
- [11]"ethereum/devp2p", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/devp2p/blob/master/discv4.md>.
- [12]"Kademlia", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Kademlia#Routing_tables.
- [13]"Proof-of-work system", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Proof-of-work_system.

- [14]"Hash function", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Hash_function.
- [15]"Proof of Work vs Proof of Stake: Basic Mining Guide - Blockgeeks", *Blockgeeks*, 2018. [Online]. Available:
<https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/>.
- [16]"Distributed networking", *En.wikipedia.org*, 2018. [Online]. Available:
https://en.wikipedia.org/wiki/Distributed_networking.
- [17]"Wayback Machine", *Web.archive.org*, 2018. [Online]. Available:
<https://web.archive.org/web/20170205142845/http://lampport.azurewebsites.net/pubs/byz.pdf>.
- [18]"ethereum/wiki", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/wiki/wiki/Mining#introduction>.
- [19]"ethereum/wiki", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/wiki/wiki/Ethash-DAG>.
- [20]"ethereum/wiki", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/wiki/wiki/Ethash>. [Accessed: 31- Oct- 2018].
- [21] S. Eloudrhiri, "Set up the private chain – miners (3/6) | ChainSkills", *ChainSkills*, 2018. [Online]. Available:
<http://chainskills.com/2017/03/10/part-3-setup-the-private-chain-miners/>. [Accessed: 31- Oct- 2018].
- [22]"ethereum/go-ethereum", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options>. [Accessed: 31- Oct- 2018].
- [23]"ethereum/go-ethereum", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/go-ethereum/wiki/Management-APIs>. [Accessed: 31- Oct- 2018].
- [24]"ethereum/go-ethereum", *GitHub*, 2018. [Online]. Available:
<https://github.com/ethereum/go-ethereum/wiki/JSON-RPC>. [Accessed: 31- Oct- 2018].
- [25]E. Foundation, "Ethereum Scalability and Decentralization Updates", *Blog.ethereum.org*, 2018. [Online]. Available:
<https://blog.ethereum.org/2014/02/18/ethereum-scalability-and-decentralization-updates/>. [Accessed: 31- Oct- 2018].
- [26] *Ideals.illinois.edu*, 2018. [Online]. Available:
<https://www.ideals.illinois.edu/bitstream/handle/2142/99409/KIM-THESIS-2017.pdf?sequence=1&isAllowed=y>. [Accessed: 02- Nov- 2018].
- [27] *People.eecs.berkeley.edu*, 2018. [Online]. Available:
<https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf>. [Accessed: 02- Nov- 2018].

[28] Fischer M.J. (1983) The consensus problem in unreliable distributed systems (a brief survey). In: Karpinski M. (eds) Foundations of Computation Theory. FCT 1983. Lecture Notes in Computer Science, vol 158. Springer, Berlin, Heidelberg

哈尔滨工业大学本科毕业设计（论文）原创性声明

本人郑重声明：在哈尔滨工业大学攻读学士学位期间，所提交的毕业设计（论文）《基于以太坊平台的学习》，是本人在导师指导下独立进行研究工作所取得的成果。对本文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明，其它未注明部分不包含他人已发表或撰写过的研究成果，不存在购买、由他人代写、剽窃和伪造数据等作假行为。

本人愿为此声明承担法律责任。

作者签名：

日期：2019 年 4 月 9 日

致 谢

首先，我要感谢我的论文导师陈世平博士，感谢他对我的学习和相关研究的不断支持，以及他的指导。最后，我要感谢我的家人和朋友，在我学校的这些年里，给予我无尽的支持和鼓励。

没有他们，完成这篇论文这一成是不可能实现的。特此致谢！