INTRODUCTION TO ETHEREUM

INTRODUCTION TO ETHEREUM

Ethereum was conceptualized by Vitalik Buterin in November 2013.

The key idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and decentralized applications.

This is in contrast to bitcoin, where the scripting language is very limited and allows basic and necessary operations only.

Ethereum clients and releases

Various Ethereum clients have been developed using different languages and currently most popular are go-Ethereum and parity.

Mist is a user-friendly Graphical User Interface (GUI) wallet that runs geth in the background to sync with the network

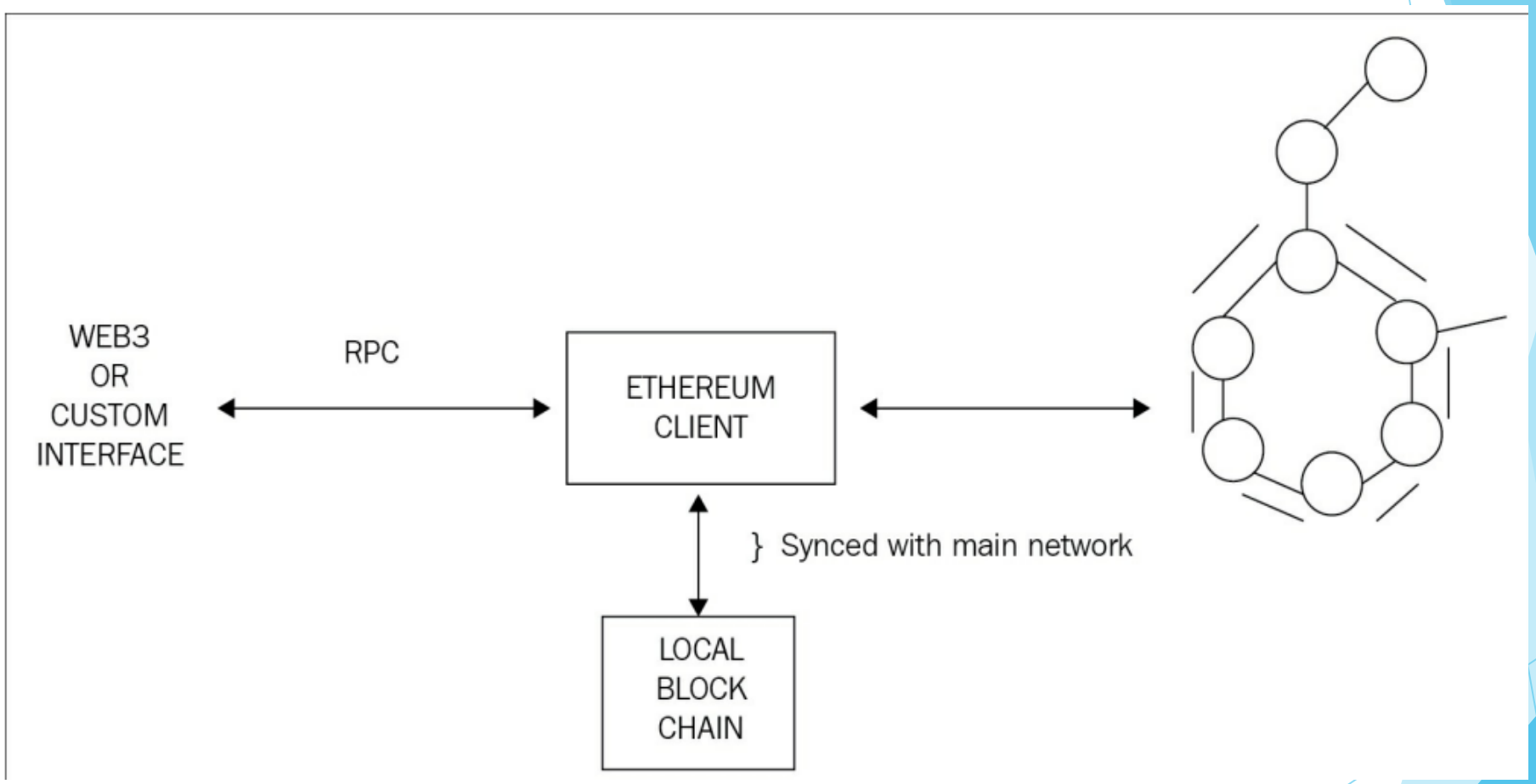
The first release of Ethereum was known as Frontier, and

The current release of Ethereum is called homestead release.

The next version is named metropolis and it focuses on protocol simplification and performance improvement.

The final release is named serenity, which is envisaged to have a Proof of Stake algorithm (Casper) implemented with it. Other areas of research targeted with serenity include scalability, privacy, and Ethereum virtual machine (EVM) upgrade

The Ethereum stack



The Ethereum stack consists of various components.

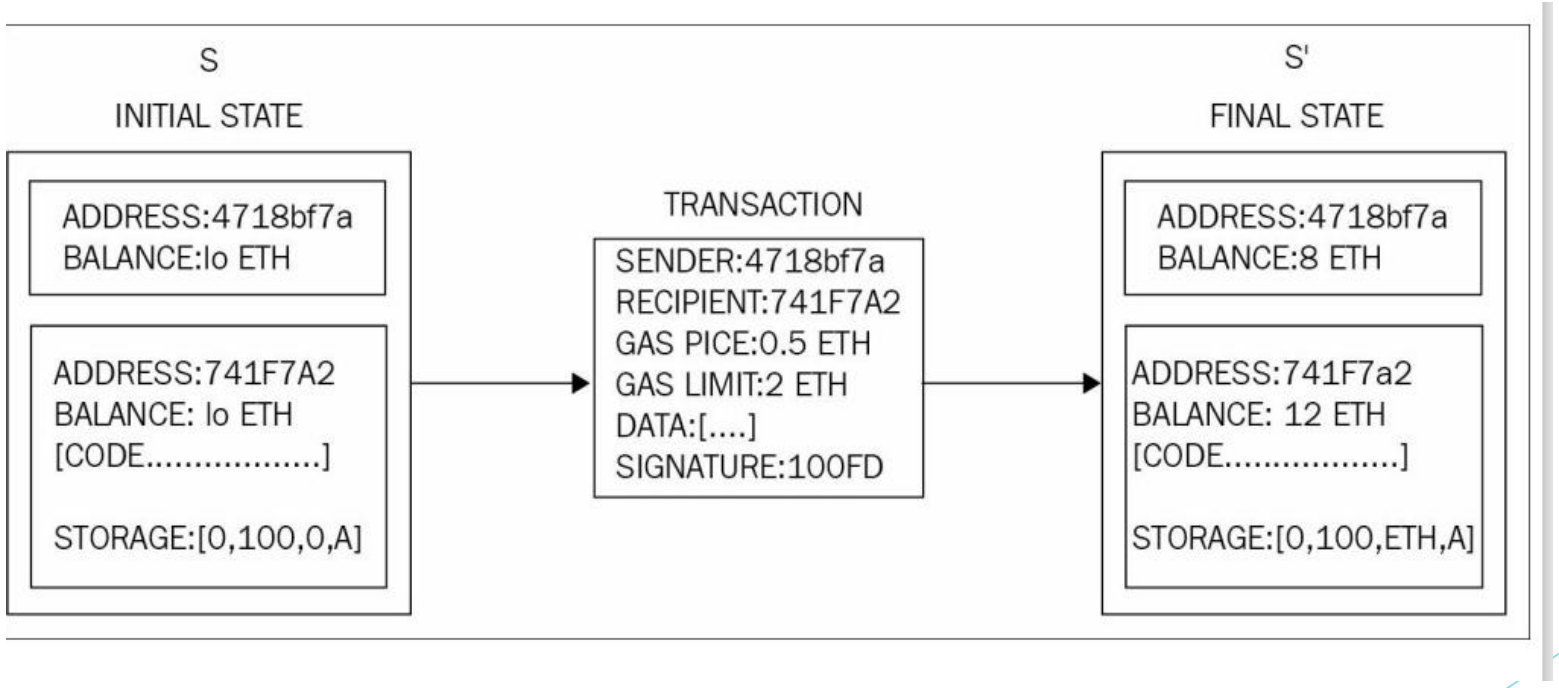
At the core, there is the Ethereum blockchain running on the P2P Ethereum network.

Secondly, there's an Ethereum client (usually geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network.

Another component is the web3.js library that allows interaction with geth via the Remote Procedure Call (RPC) interface.

Ethereum blockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood. The idea is that a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state.



Currency (ETH and ETC) :

As an incentive to the miners, Ethereum also rewards its native currency called Ether, abbreviated as ETH.

The massive hack would go on to drain as much as $60 million worth of Ether, or one-third of the funds contributed by would-be DAO(Decentralized Autonomous Organizations) participants. Even after a white-hat counterattack, the stolen funds would ultimately amount to around 5% of all the Ethereum tokens in existence at the time

After the DAO hack ,a hard fork was proposed in order to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum classic and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out

Forks

A fork is **a change to the blockchain protocol.**

However, many times, the nodes in the network cannot come to a unanimous consensus regarding the **future state of the blockchain**. This event leads to **forks**, meaning that it leads to a point in which the ideal single chain of blocks is split into two or more chains, that are all valid.

Hard Forks

When there is a change in the software that runs on full nodes to function as a network participant, the new blocks mined based on the new rules in the blockchain protocol are not considered valid by the old version of the software. When hard forks occur, new currency comes into existence. An equivalent quantity of currency is distributed to the full nodes that choose to upgrade their software so that no material loss occurs.

Example: Suppose, there is a new update in the Ethereum Blockchain in which the consensus protocol will change from a type of proof-of-work to a type of proof-of-stake. The full nodes that install the update will use the new consensus protocol, and the ones that do not choose to install the update will become incompatible in the blockchain.

2. Soft Forks

When there is a change in the software that runs on full nodes to function as a network participant, new blocks are mined **based on new rules in the blockchain protocol** and are also **considered valid by the old version of the software**. This feature is also called backward compatibility.

Example: Suppose, there is a new update in the Ethereum blockchain in which the consensus protocol will change from a type of proof-of-work to a type of proof-of-stake. The full nodes that will install the update will use the new consensus protocol, and the ones that choose not to install the update will still stay compatible with the other nodes in the blockchain.

With the latest release of homestead, due to major protocol upgrades, it resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum known as Frontier to the second version of Ethereum called homestead.

A recent unintentional fork that occurred on November 24, 2016, at 14:12:07 UTC was due to a bug in the geth client's journaling mechanism. Network fork occurred at block number 2,686,351. This bug resulted in geth failing to revert empty account deletions in the case of the empty out-of-gas exception. This was not an issue in parity (another popular Ethereum client). This means that from block number 2686351, the Ethereum blockchain is split into two, one running with parity clients and the other with geth. This issue was resolved with the release of geth version 1.5.3

Gas

Another key concept in Ethereum is that of gas. All transactions on the Ethereum blockchain are required to cover the cost of computation they are performing.

The cost is covered by something called gas or crypto fuel, which is a new concept introduced by Ethereum.

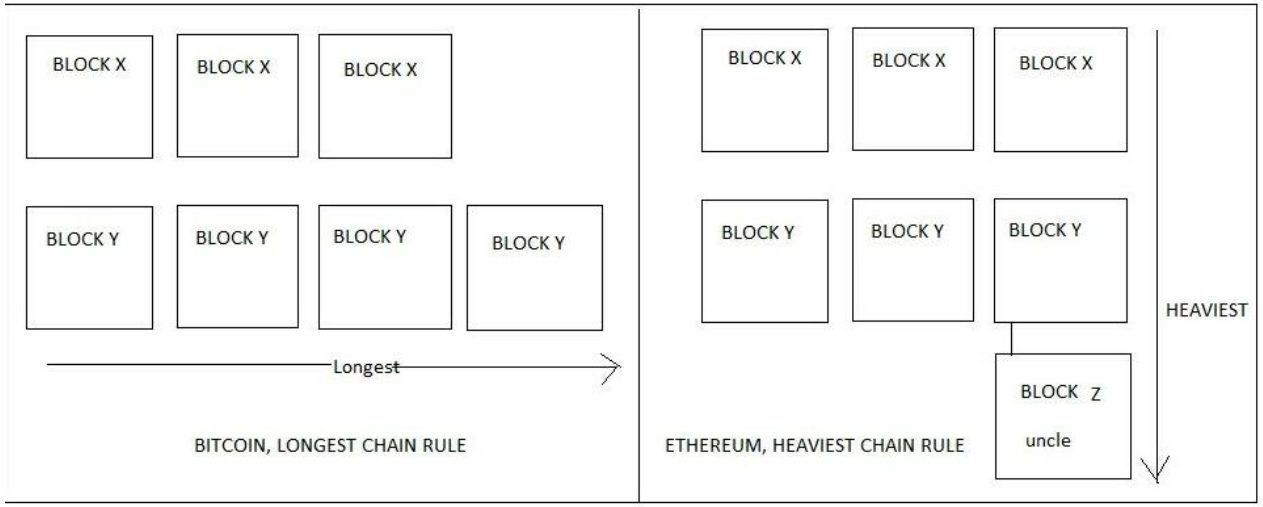
This gas as execution fee is paid upfront by the transaction originators. The fuel is consumed with each operation. Each operation has a predefined amount of gas associated with it. Each transaction specifies the amount of gas it is willing to consume for its execution.

If it runs out of gas before the execution is completed, any operation performed by the transaction up to that point is rolled back. If the transaction is successfully executed, then any remaining gas is refunded to the

The consensus mechanism

Ethereum uses a simpler version of this protocol, where the chain that has most computational effort spent on it

Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining effort. Greedy Heaviest Observed Subtree (GHOST) was first introduced as a mechanism to alleviate the issues. In GHOST, stale blocks are added in calculations to figure out the longest and heaviest chain of blocks. Stale blocks are called Uncles or Ommers in Ethereum. The following diagram shows a quick comparison between the longest and heaviest chain:



The world state

The world state in Ethereum represents the global state of the Ethereum blockchain. It is basically a mapping between Ethereum addresses and account states. The addresses are 20 bytes long. This mapping is a data structure that is serialized using Recursive Length Prefix (RLP)

The account state:

The account state consists of four fields: nonce, balance, storageroot and codehash

Nonce

This is a value that is incremented every time a transaction is sent from the address. In case of contract accounts, it represents the number of contracts created by the account. Contract accounts are one of the two types of accounts that exist in Ethereum.

Balance

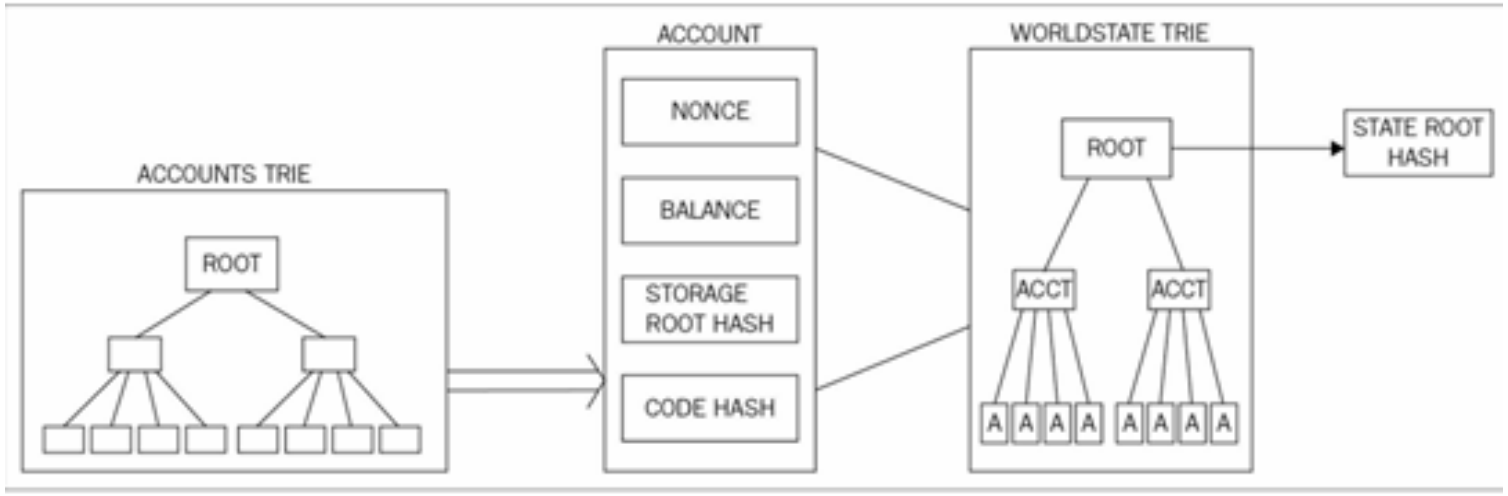
This value represents the number of Weis which is the smallest unit of the currency (Ether) in Ethereum held by the address.

Storageroot :

This field represents the root node of a Merkle Patricia tree that encodes the storage contents of the account.

Codehash

This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.

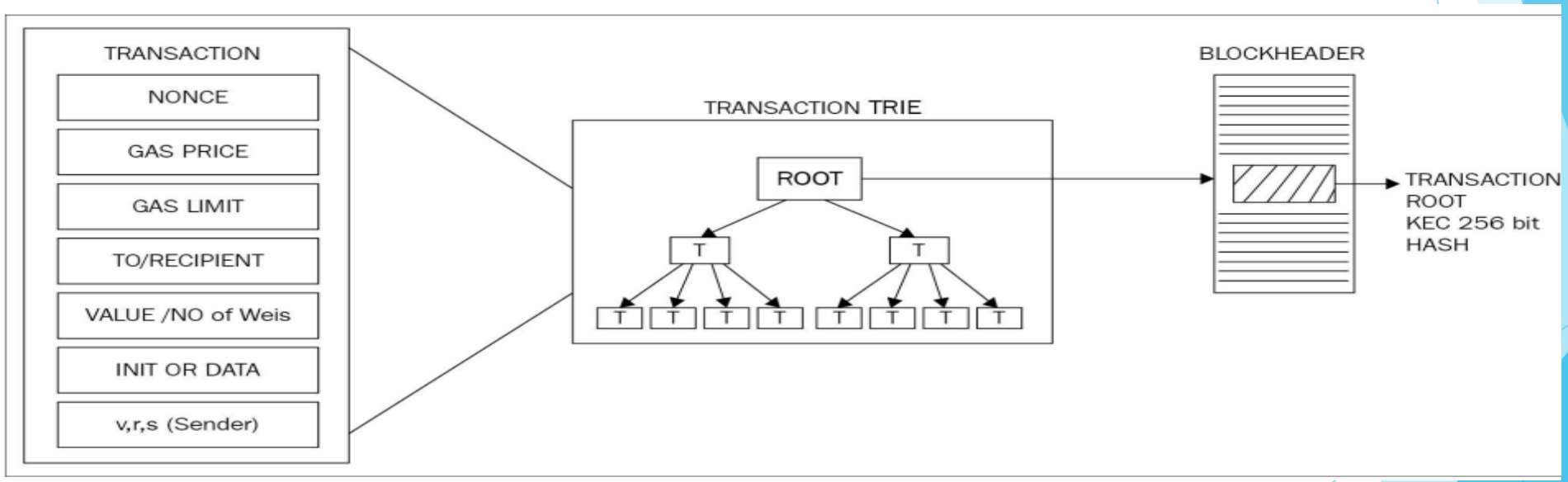


Transactions:

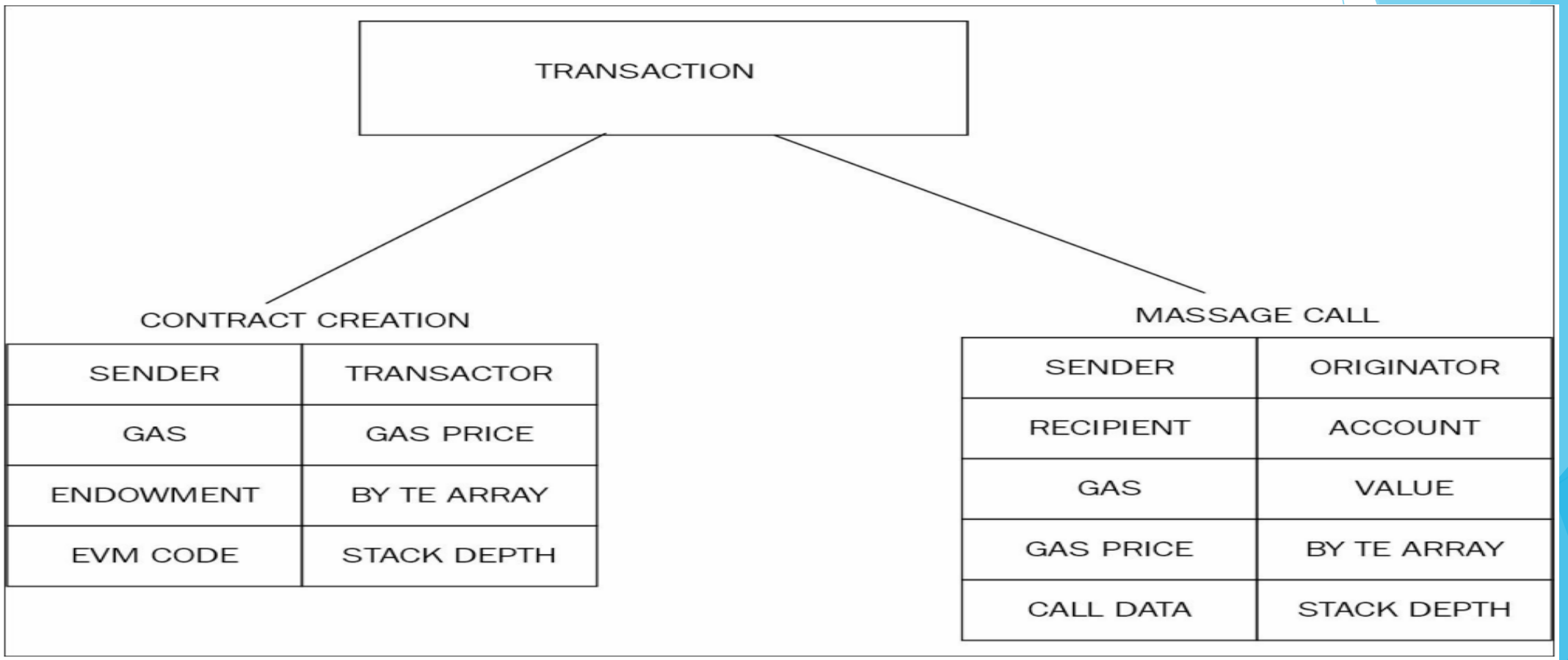
A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation. Transactions can be divided into two types based on the output they produce:

Message call transactions: This transaction simply produces a message call that is used to pass messages from one account to another.

Contract creation transactions: As the name suggests, these transactions result in the creation of a new contract. This means that when this transaction is executed successfully, it creates an account with the



- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- Current depth means the number of items that are already there in the stack



Elements of the Ethereum blockchain:

Ethereum virtual machine (EVM)

EVM is a simple stack-based execution machine that runs bytecode instructions in order to transform the system state from one state to another. The word size of the virtual machine is set to 256-bit. The stack size is limited to 1024 elements and is based on the LIFO (Last in First Out) queue.

EVM is a fully isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources, such as a network or filesystem

EVM is a stack-based architecture. EVM is big-endian by design and it uses 256-bit wide words. This word size allows for Keccak 256-bit hash and elliptic curve cryptography computations.

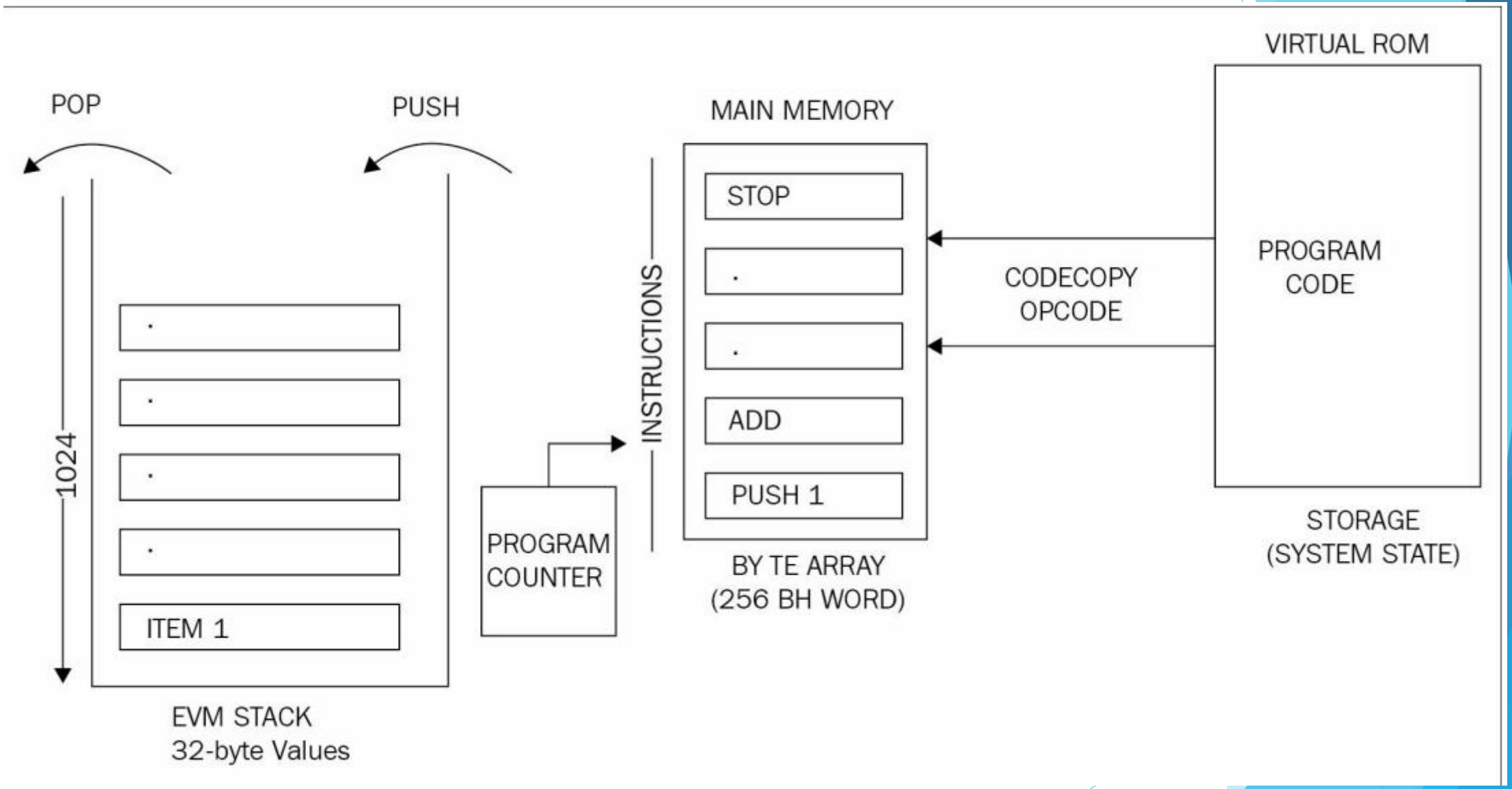
There are two types of storage available to contracts and EVM.

The first one is called memory, which is a byte array. When a contract finishes the code execution, the memory is cleared. It is akin to the concept of RAM.

The other type, called storage, is permanently stored on the blockchain. It is a key value store.

Memory is unlimited but constrained by gas fee requirements. The storage associated with the virtual machine is a word addressable word array that is nonvolatile and is maintained as part of the system state.

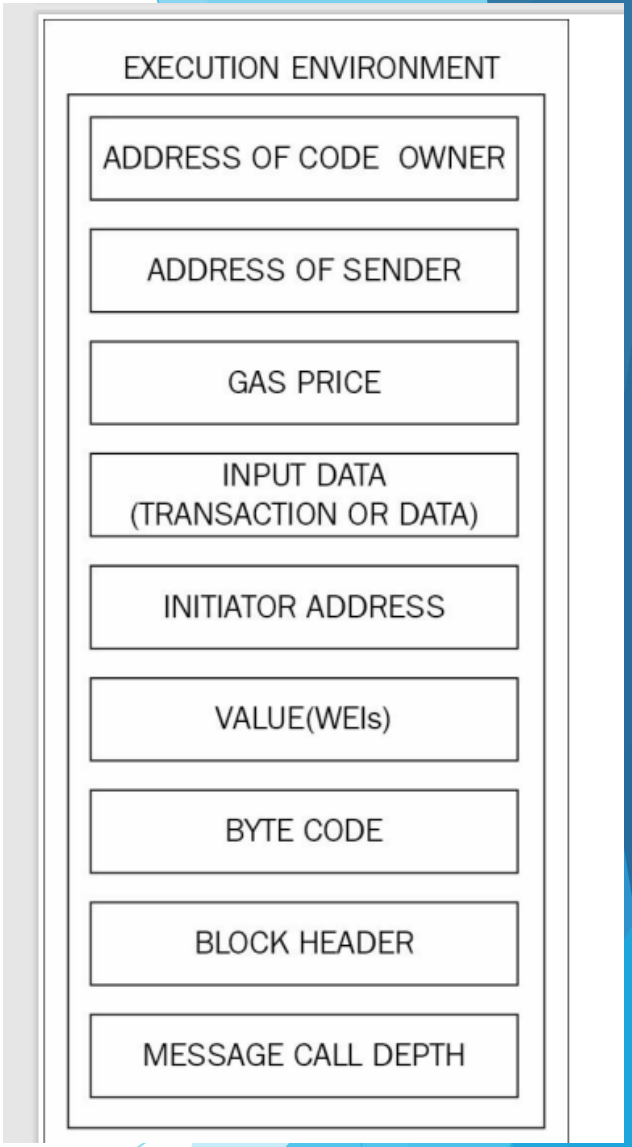
Keys and value are 32 bytes in size and storage. The program code is stored in a virtual readonly memory (virtual ROM) that is accessible using the CODECOPY instruction. The CODECOPY instruction is used to copy the program code into the main memory. Initially, all storage and memory is set to zero in the EVM.


VIRTUAL ROM
POP
PUSH
MAIN MEMORY
PROGRAM
CODE
STOP
.
.
ADD
PUSH 1
INSTRUCTIONS
CODECOPY
OPCODE
1024
.
.
.
ITEM 1
PROGRAM
COUNTER
STORAGE
(SYSTEM STATE)
BY TE ARRAY
(256 BH WORD)
EVM STACK
32-byte Values

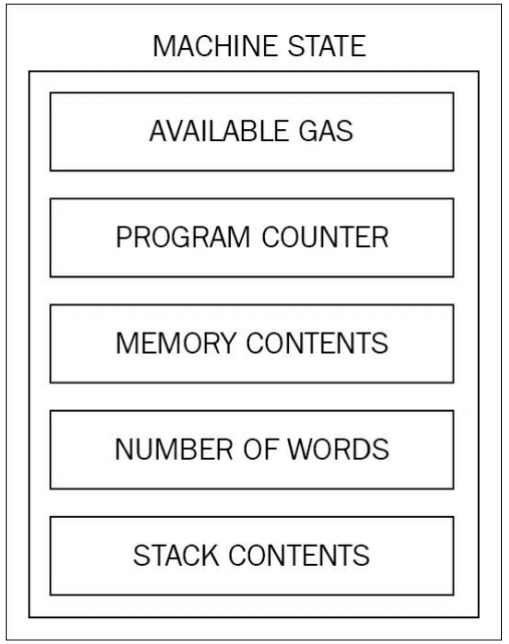
EVM operation :

WASM is developed by Google, Mozilla, and Microsoft and is now being designed as an open standard by the W3C community group. The aim of WASM is to be able to run machine code in the browser that will result in execution at native speed. Similarly, the aim of EVM 2.0 is to be able to run the EVM instruction set (Opcodes) natively in CPUs, thus making it faster and efficient.

1.The address of the account that owns the executing code.
2. The address of the sender of the transaction and the originating address of this execution.
3. The gas price in the transaction that initiated the execution.
4. Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
5. The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it's the address of the account.
6. The value or transaction value. This is the amount in **Wei(Smallest denomination of ether).** If the execution agent is a transaction, then it is the transaction value.
7. The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
8. The block header of the current block
9. The number of message calls or contract creation transactions currently in execution. In other words, this is the number of CALLs or CREATEs currently in execution



Machine state



Machine state is also maintained internally by the EVM. Machine state is updated after each execution cycle of EVM. Machine state is a tuple that consist of the following elements:
Available gas
The program counter, which is a positive integer up to 256
Memory contents
Active number of words in memory
Contents of the stack

The iterator function:
The iterator function mentioned earlier performs various important functions that are used to set the next state of the machine and eventually the world state.
It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
It adds/removes (PUSH/POP) items from the stack accordingly.
It increments the program counter (PC).

Opcodes and their meaning :There are different opcodes that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. The list of opcodes with their meaning and usage is presented here.

Arithmetic operations

All arithmetic in EVM is modulo 2^256. This group of opcodes is used to perform basic arithmetic operations. The value of these operations starts from 0x00 up to 0x0b.

Mnemonic	Value	POP	PUSH	Gas	Description
STOP	0x00	0	0	0	Halts execution
ADD	0x01	2	1	3	Adds two values
MUL	0x02	2	1	5	Multiplies two values
SUB	0x03	2	1	3	Subtraction operation
DIV	0x04	2	1	5	Integer division operation
SDIV	0x05	2	1	5	Signed integer division operation
MOD	0x06	2	1	5	Modulo remainder operation
SMOD	0x07	2	1	5	Signed modulo remainder operation
ADDMOD	0x08	3	1	8	Modulo addition operation
MULMOD	0x09	3	1	8	Module multiplication operation
EXP	0x0a	2	1	10	Exponential operation (repeated multiplication of the base)
SIGNEXTEND	0x0b	2	1	5	Extends the length of 2s complement signed integer

Logical Operations

Mnemonic	Value	POP	PUSH	Gas	Description
LT	0x10	2	1	3	Less than
GT	0x11	2	1	3	Greater than
SLT	0x12	2	1	3	Signed less than comparison
SGT	0x13	2	1	3	Signed greater than comparison
EQ	0x14	2	1	3	Equal comparison
ISZERO	0x15	1	1	3	Not operator
AND	0x16	2	1	3	Bitwise AND operation
OR	0x17	2	1	3	Bitwise OR operation
XOR	0x18	2	1	3	Bitwise exclusive OR (XOR) operation
NOT	0x19	1	1	3	Bitwise NOT operation
BYTE	0x1a	2	1	3	Retrieve single byte from word

Cryptographic operations

There is only one operation in this category named SHA3. It is worth noting that this is not the standard SHA3 standardized by NIST but the original Keccak implementation.

Mnemonic	Value	POP	PUSH	Gas	Description
SHA3	0x20	2	1	30	Used to calculate Keccak 256-bit hash.

Mnemonic	Value	POP	PUSH	Gas	Description
ADDRESS	0x30	0	1	2	Used to get the address of the currently executing account
BALANCE	0x31	1	1	20	Used to get the balance of the given account
ORIGIN	0x32	0	1	2	Used to get the address of the sender of the original transaction
CALLER	0x33	0	1	2	Used to get the address of the account that initiated the execution
CALLVALUE	0x34	0	1	2	Retrieves the value deposited by the instruction or transaction
CALLDATALOAD	0x35	1	1	3	Retrieves the input data that was passed a parameter with the message call
CALLDATASIZE	0x36	0	1	2	Used to retrieve the size of the input data passed with the message call
CALLDATACOPY	0x37	3	0	3	Used to copy input data passed with the message call from the current environment to the memory.
CODESIZE	0x38	0	1	2	Retrieves the size of running the code in the current environment
CODECOPY	0x39	3	0	3	Copies the running code from current environment to the memory
GASPRICE	0x3a	0	1	2	Retrieves the gas price specified by the initiating transaction.
EXTCODESIZE	0x3b	1	1	20	Gets the size of the specified account code
EXTCODECOPY	0x3c	4	0	20	Used to copy the account code to the memory.

Block Information

This set of instructions is related to retrieving various attributes associated with a block:

Mnemonic	Value	POP	PUSH	Gas	Description
BLOCKHASH	0x40	1	1	20	Gets the hash of one of the 256 most recently completed blocks
COINBASE	0x41	0	1	2	Retrieves the address of the beneficiary set in the block
TIMESTAMP	0x42	0	1	2	Retrieves the time stamp set in the blocks
NUMBER	0x43	0	1	2	Gets the block's number
DIFFICULTY	0x44	0	1	2	Retrieves the block difficulty
GASLIMIT	0x45	0	1	2	Gets the gas limit value of the block

Stack, memory, storage and flow operations

Mnemonic	Value	POP	PUSH	Gas	Description
POP	0x50	1	0	2	Removes items from the stack
MLOAD	0x51	1	1	3	Used to load a word from the memory.
MSTORE	0x52	2	0	3	Used to store a word to the memory.
MSTORE8	0x53	2	0	3	Used to save a byte to the memory
SLOAD	0x54	1	1	50	Used to load a word from the storage
SSTORE	0x55	2	0	0	Saves a word to the storage
JUMP	0x56	1	0	8	Alters the program counter
JUMPI	0x57	2	0	10	Alters the program counter based on a condition
PC	0x58	0	1	2	Used to retrieve the value in the program counter before the increment.
MSIZE	0x59	0	1	2	Retrieves the size of the active memory in bytes.
GAS	0x5a	0	1	2	Retrieves the available gas amount
JUMPDEST	0x5b	0	0	1	Used to mark a valid destination for jumps with no effect on the machine state during the execution.

Push operations

These operations include PUSH operations that are used to place items on the stack. The range of these instructions is from 0x60 to 0x7f. There are 32 PUSH operations available in total in the EVM. PUSH operation, which reads from the byte array of the program code.

Mnemonic	Value	POP	PUSH	Gas	Description
PUSH1 . . . PUSH 32	0x60 ... 0x7f	0	1	3	Used to place *N* right-aligned big-endian byte item(s) on the the stack. *N* is a value that ranges from 1 byte to 32 bytes (full word) based on the mnemonic used.

Duplication operations

As the name suggests, duplication operations are used to duplicate stack items. The range of values is from 0x80 to 0x8f. There are 16 DUP instructions available in the EVM. Items placed on the stack or removed from the stack also change incrementally with the mnemonic used; for example, DUP1 removes one item from the stack and places two items on the stack, whereas DUP16 removes 16 items from the stack and places 17 items.

Mnemonic	Value	POP	PUSH	Gas	Description
DUP1 . . . DUP16	0x80 ... 0x8f	X	Y	3	Used to duplicate the nth stack item, where *N* is the number corresponding to the DUP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Exchange operations

SWAP operations provide the ability to exchange stack items. There are 16 SWAP instructions available and with each instruction, the stack items are removed and placed incrementally up to 17 items depending on the type of Opcode used.

Mnemonic	Value	POP	PUSH	Gas	Description
SWAP1 . . . SWAP16	0x90 ... 0x9f	X	Y	3	Used to swap the nth stack item, where *N* is the number corresponding to the SWAP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Logging operations

Logging operations provide opcodes to append log entries on the sub-state tuple's log series field. There are four log operations available in total and they range from value 0x0a to 0xa4.

Mnemonic	Value	POP	PUSH	Gas	Description
LOG0 . . . LOG4	0x0a ... 0xa4	X	Y (0)	375, 750, 1125, 1500, 1875	Used to append log record with *N* topics, where *N* is the number corresponding to the LOG Opcode used. For example, LOG0 means a log record with no topics, and LOG4 means a log record with four topics. *X* and *Y* represent the items removed and placed on the stack, respectively. *X* and *Y* change incrementally, starting from 2, 0 up to 6, 0 according to the LOG operation used.

System operations

System operations are used to perform various system-related operations, such as account creation, message calling, and execution control. There are six Opcodes available in total in this category.

Mnemonic	Value	POP	PUSH	Gas	Description
CREATE	0xf0	3	1	32000	Used to create a new account with the associated code.
CALL	0xf1	7	1	40	Used to initiate a message call into an account.
CALLCODE	0xf2	7	1	40	Used to initiate a message call into this account with an alternative account's code.
RETURN	0xf3	2	0	0	Stops the execution and returns output data.
DELEGATECALL	0xf4	6	1	40	The same as CALLCODE but does not change the current values of the sender and the value.
SUICIDE	0xff	1	0	0	Stops (halts) the execution and the account is registered for deletion later

In this section, all EVM opcodes have been discussed. There are 129 opcodes available in the EVM of the homestead release of Ethereum in total.

Setting up Ethereum Development Tools:
Ethereum Development - Setting up a development environment

Test Net is called Ropsten and is used by developers or users as a test platform to test smart contracts and other blockchain-related proposals.
The Private Net option in Ethereum allows the creation of an independent private network that can be used as a distributed ledger between participating entities and for the development and testing of smart contracts.

Test Net (Ropsten) The Ethereum Go client, geth, can be connected to the test network using the following command:
$ geth --TestNet

Setting up a Private Net :

Private Net allows the creation of an entirely new blockchain. This is different from Test Net or Main Net in the sense that it uses its on-genesis block and Network ID. In order to create Private Net, three components are needed: 1. Network ID. 2. Genesis file. 3. Data directory to store blockchain data. Even though data directory is not strictly required to be mentioned, if there is more than one blockchain already active on the system, then data directory should be specified so that a separate directory is used for the new blockchain.

Network ID :
Network ID can be any positive number except 1 and 3, which are already in use by Ethereum Main Net and Test Net (Ropsten), respectively.

The genesis file :

The genesis file contains necessary fields required for a custom genesis block. This is the first block in the network and does not point to any previous block

Starting up the private network :

The initial command to start the private network is shown as follows:

```
$ geth --datadir ~/.ethereum/privatenet init ./privether/privategenesis.json
```

Private network initialization

This output indicates that a genesis block has been created successfully. In order for geth to start, the following command can be issued:

```
$ geth --datadir .ethereum/privatenet/ --networkid 786
```

Starting geth for a private network:

Now geth can be attached via IPC to the running geth client on a private network using the following command. This will allow you to interact with the running geth session on the private network:

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```

The following command creates a new account. In this context, the account will be created on the Private Network ID 786:

- personal.newAccount("Password123")

"0x76f11b383dbc3becf8c5d9309219878caae265c3“

Once the account is created, the next step is to set it as an Etherbase/coinbase account so that the mining reward goes to this account. This can be achieved using the following command:

> miner.setEtherbase(personal.listAccounts[0])
True

Currently, the etherbase account has no balance, as can be seen using the following command:

> eth.getBalance(eth.coinbase).toNumber();
0

In the JavaScript console, the current balance of total ether can be queried, as shown here. After mining, a significant amount can be seen in the following example. Mining is extremely fast as it is a
private network and in the genesis file, the network difficulty has also been set quite low:

```
> eth.getBalance(eth.coinbase).toNumber();
2.72484375e+21
```

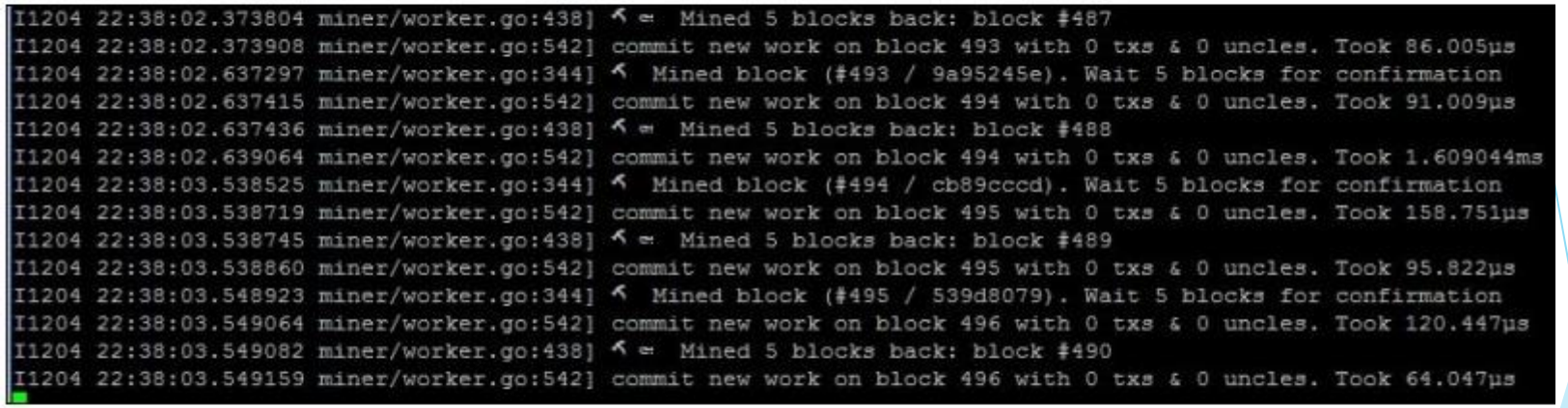
After mining starts, the first DAG generation is carried out and output similar to the following is
produced:

```
I0211 23:58:50.380089 eth/backend.go:479] Automatic pregeneration of ethash DAG ON (ethash dir: /home/imran/.e
thash)
I0211 23:58:50.380097 miner/miner.go:136] Starting mining operation (CPU=2 TOT=3)
I0211 23:58:50.380138 eth/backend.go:486] checking DAG (ethash dir: /home/imran/.ethash)
I0211 23:58:50.380257 miner/worker.go:542] commit new work on block 1 with 0 txs & 0 uncles. Took 139.49µs
I0211 23:58:50.380292 vendor/github.com/ethereum/ethash/ethash.go:259] Generating DAG for epoch 0 (size 107373
9904) (0000000000000000000000000000000000000000000000000000000000000000)
I0211 23:58:51.166755 vendor/github.com/ethereum/ethash/ethash.go:276] Done generating DAG for epoch 0, it too
k 786.458657ms
```

efficiency in data storage and processing of online transactions.

DAG generation

Once DAG generation is finished and mining starts, `geth` will produce output similar to that shown in the following screenshot. It can be clearly seen that blocks are being mined successfully with the `Mined 5 blocks . . .` message.



Mining output

Mining can be stopped using the following command:

```
> miner.stop
true
```

List of methods
There are a few other commands that can be used to query the private network. Some examples are
shown as follows:

Get the current gas price:

```
    eth.gasPrice
20000000000
```

Get the latest block number:

```
> eth.blockNumber
587
```

Unlock the account before sending transaction:

> personal.unlockAccount
("0x76f11b383dbc3becf8c5d9309219878caae265c3")

Unlock account 0x76f11b383dbc3becf8c5d9309219878caae265c3
Passphrase: ****

Send transactions:
> eth.sendTransaction({from:
"0x76f11b383dbc3becf8c5d9309219878caae265c3", to:
"0xcce6450413ac80f9ee8bd97ca02b92c065d77abc", value: 1000})

Another way is to use listAccounts[] method, this can be done as shown below:
> eth.sendTransaction({from: personal.listAccounts[0], to:
personal.listAccounts[1], value: 1000})

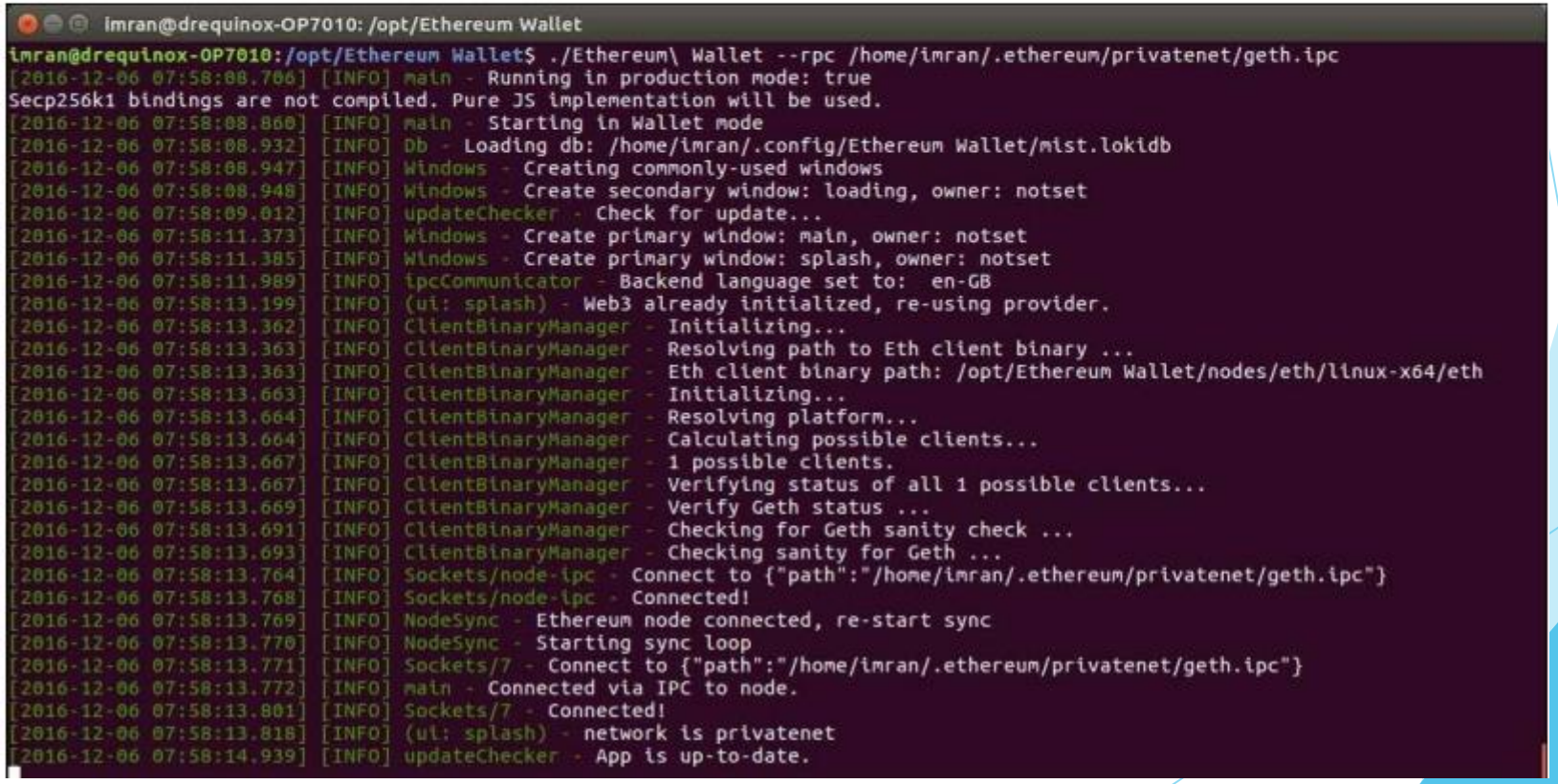
Get a list of compilers.
> web3.eth.getCompilers()
["Solidity"]

Running Mist on Private Net

It is possible to run Mist on Private Net by issuing the following command. This binary is usually available in the `home` folder after the installation of `/opt/Ethereum`:

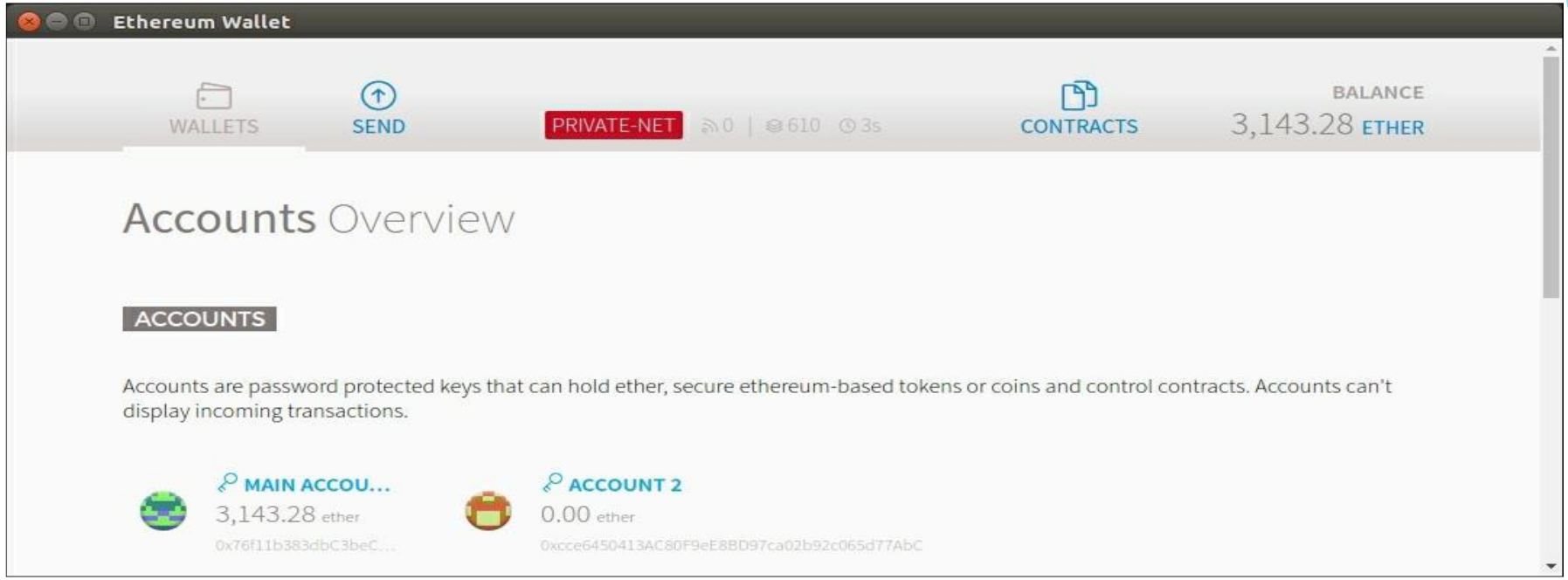
```
$ ./Ethereum\ Wallet --rpc ~/.ethereum/privatenet/geth.ipc
```

This will allow a connection to the running Private Net `geth` session, and it provides all features, such as wallet, account management, and contract deployment on Private Net via Mist.



Running Ethereum Wallet to connect to Private Net

Once Ethereum is launched, it will show the interface shown here, indicating clearly that it's running in the **PRIVATE-NET** mode.



Deploying contracts using Mist

It is very easy to deploy new contracts using Mist. Mist provides an interface where contracts can be written in solidity and then deployed on the network.

In the exercise, a simple contract that can perform various simple arithmetic calculations on the input parameter will be used.
Steps on how to use Mist to deploy this contract are shown here. As solidity
has not been introduced yet, the aim here is to allow users to experience the contract deployment and interaction process. More information on coding and solidity will be provided later in the chapter, after which it will become easy to understand the code shown. Those of you who are already familiar with JavaScript or any other similar language will find the code almost self-explanatory.

Smart Contracts:

History :

Smart contracts were first theorized by Nick Szabo in the late 1990s, but it was almost 20 years before the true potential and benefits of them were truly appreciated. Smart contracts are described by Szabo as follows: "A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs." This idea of smart contracts was implemented in a limited fashion in bitcoin in 2009, where bitcoin transactions can be used to transfer the value between users, over a peer-to-peer network where users do not necessarily trust each other and there is no need for a trusted intermediary

Definition :

There is no consensus on a standard definition of smart contracts. It is essential to define what a smart contract is, and the following is the author's attempt to provide a generalized definition of a smart contract.

Note:

A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.

a smart contract has the following four properties:
Automatically executable
Enforceable
Semantically sound
Secure and unstoppable

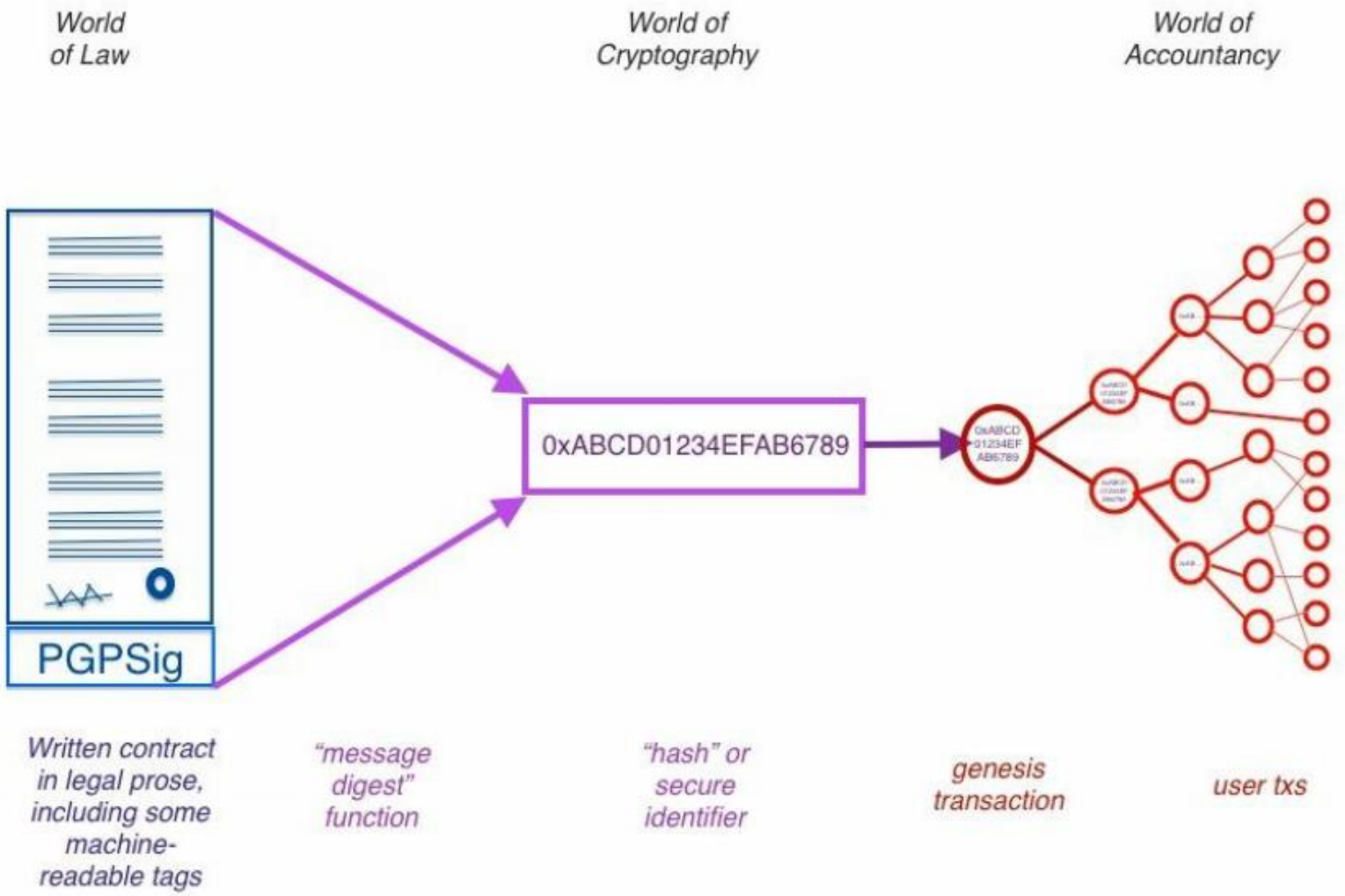
Ricardian contracts:

Ricardian contracts were originally proposed in the Financial Cryptography in 7 Layers paper by Ian Grigg in late 1990s. These contracts were used initially in a bond trading and payment system called Ricardo. The key idea is to write a document which is understandable and acceptable by both a court of law and computer software. Ricardian contracts address the challenge of issuance of value over the Internet. It identifies the issuer and captures all the terms and clauses of the contract in a document in order to make it acceptable as a legally binding contract.

- A contract offered by an issuer to holders
- A valuable right held by holders, and managed by the issuer
- Easily readable by people (like a contract on paper)
- Readable by programs (parseable, like a database)
- Digitally signed
- Carries the keys and server information
- Allied with a unique and secure identifier

The Ricardian Contract

the BowTie Model



A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a contract can be divided into two types: operational semantics and denotational semantics. The first type defines the actual execution, correctness and safety of the contract, and the latter is concerned with the real-world meaning of the full contract. Some researchers have differentiated between smart contract code and smart legal contracts where a smart contract is only concerned with the execution of the contract and the second type encompasses both the denotational and operational semantics of a legal agreement

At bitcoin, a very simple implementation of a smart contract can be observed which is fully oriented towards the execution of the contract, whereas a Ricardian contract is more geared towards producing a document that is understandable by humans, with some parts that a computer program can understand.

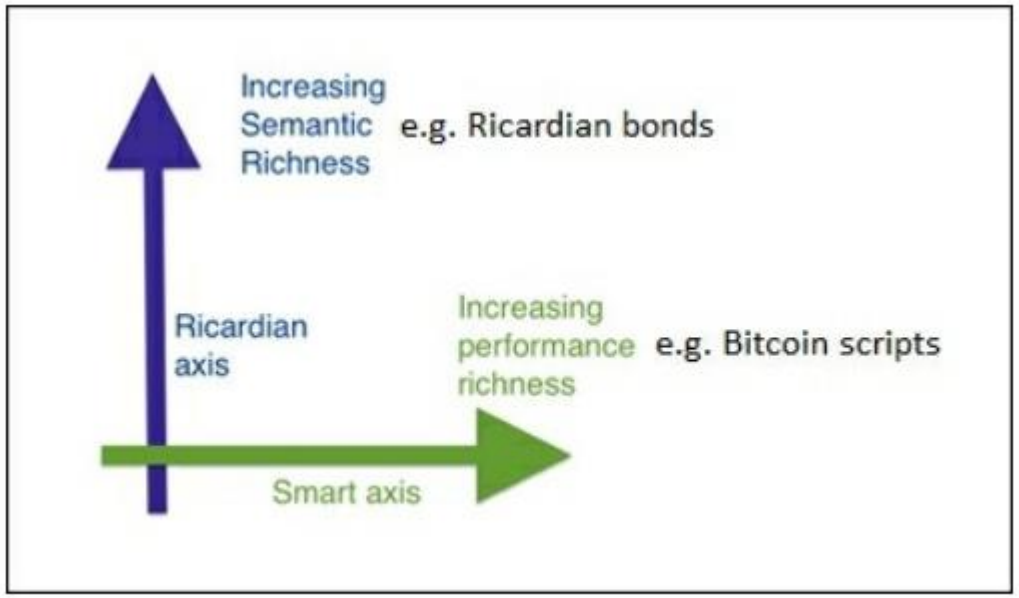


Diagram explaining performance v. semantics are orthogonal issues as described by Ian Grigg; slightly modified to show examples of different types of contracts on both axis

A smart contract is made up to have both of these elements (performance and semantics) embedded together, which completes an ideal model of a smart contract.

A Ricardian contract can be represented as a tuple of three objects, namely Prose, parameters and code. Prose represents the legal contract in regular language; code represents the program that is a computer-understandable representation of legal prose; and parameters join the appropriate parts of the legal contract to the equivalent code.

Ricardian contracts have been implemented in many systems, such as CommonAccord, OpenBazaar, OpenAssets, and Askemos