7
Ethereum 101

This chapter is intended to be an introduction to the Ethereum blockchain. You will be introduced to the fundamentals and advanced theoretical concepts behind Ethereum. A discussion on various components, protocols, and algorithms relevant to the Ethereum blockchain will be given in detail so that you can understand the theory behind this blockchain paradigm. Also, a practical and in-depth introduction to wallet software, mining, and setting up Ethereum nodes will be covered in this chapter. Some material on various challenges, such as security and scalability faced by Ethereum, will also be introduced. Additionally, trading and market dynamics will be discussed.

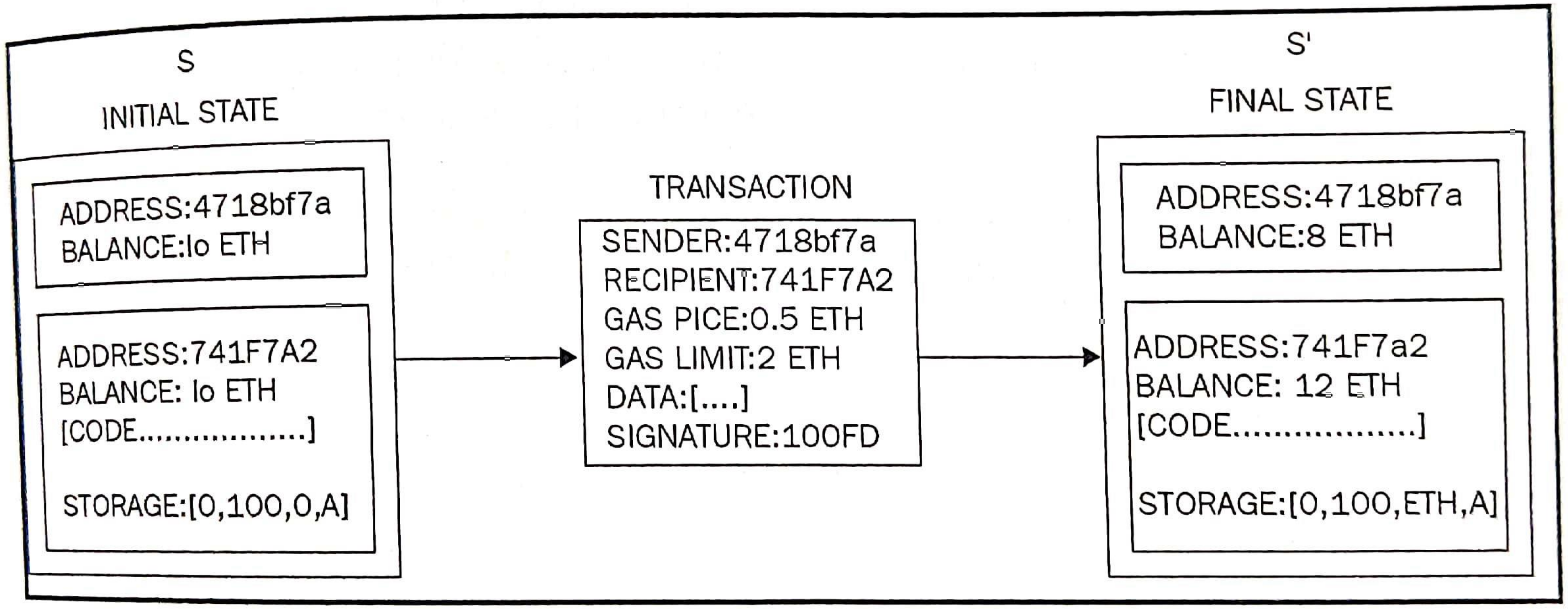
Introduction

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.

The Ethereum stack showing various components

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. In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition.



Ethereum State transition function

In the preceding example, a transfer of 2 Ether from **Address 4718bf7a** to Address 741f7a2 is initiated. The initial state represents the state before the transaction execution and the final state is what the morphed state looks like. This will be discussed in more detail later in the chapter, but the aim of this example is to introduce the core idea of state transition in Ethereum.

Elements of the Ethereum blockchain

In the following section, you will be introduced to various components of the Ethereum network and the blockchain. First, the basic concept of the EVM is given in the next section.

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 Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial of service attacks are not possible due to gas requirements. EVM also supports exception handling in case exceptions occur, such as not having enough gas or invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

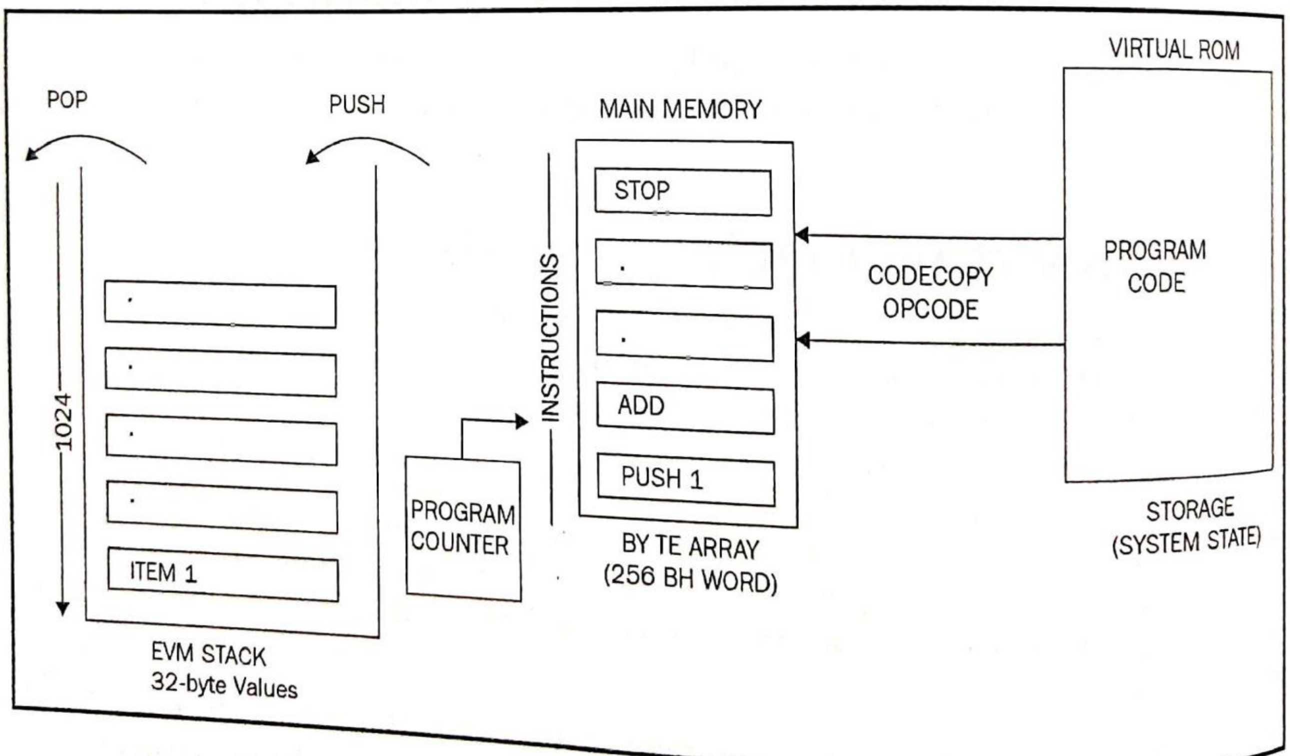
EVM is a fully isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources, such as a network or filesystem.

As discussed earlier, 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 read-only 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.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into main memory using **CODECOPY**. The main memory is then read by the EVM by referring to the program counter and executes instructions step by step. The program counter and EVM stack are updated accordingly with each instruction execution.



EVM operation

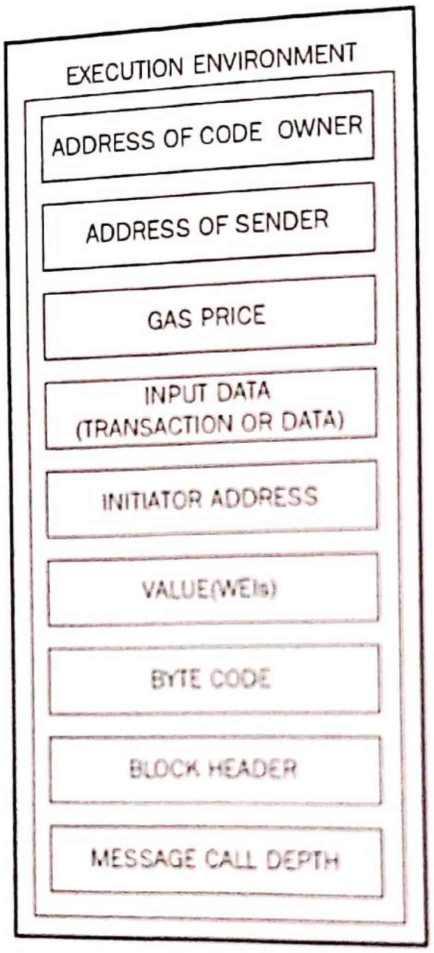
EVM optimization is an active area of research and recent research has suggested that EVM can be optimized and tuned to a very fine degree in order to achieve high performance. Research into the possibility of using **Web assembly (WASM)** is underway already. WASM is developed by Google, Mozilla, and 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.

Execution environment

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

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. 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.

The execution environment can be visualized as a tuple of nine elements, as follows:



Execution environment Tuple

In addition to the previously mentioned nine fields, system state and the remaining gas are also provided to the execution environment. The execution results in producing the resulting state, gas remaining after the execution, self-destruct or suicide set (described later), log series (described later), and any gas refunds.

Machine state

Machine state is also maintained internally by the EVM. Machine state is updated after each execution cycle of EVM. An iterator function (detailed in the next section) runs in the virtual machine, which outputs the results of a single cycle of the state machine. 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 EVM is designed to handle exceptions and will halt (stop execution) in case any of the following exceptions occur:

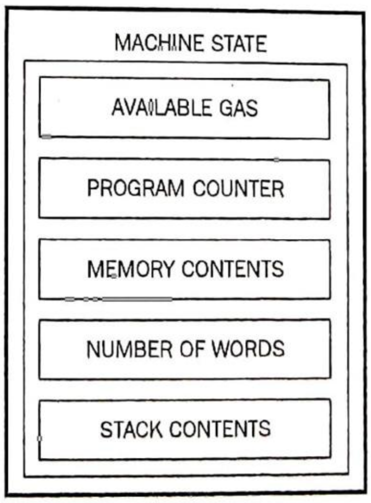
- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump op codes
- Invalid stack size (greater than 1024)

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. These functions include the following:

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

Machine state can be viewed as a tuple shown in the following diagram:



Machine state tuple

The virtual machine is also able to halt in normal conditions if STOP or SUICIDE or RETURN Opcodes are encountered during the execution cycle.

Code written in a high-level language such as serpent, LLL, or Solidity is converted into the byte code that EVM understands in order for it to be executed by the EVM. Solidity is the high-level language that has been developed for Ethereum with JavaScript such as syntax to write code for smart contracts. Once the code is written, it is compiled into byte code that's understandable by the EVM using the Solidity compiler called solc.

LLL (Lisp-like Low-level language) is another language that is used to write smart contract code. Serpent is a Python-like high-level language that can be used to write smart contracts for Ethereum.

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

```
pragma solidity ^0.4.0;
contract Test1
 {
    uint x=2;
    function addition1(uint x) returns (uint y) {
    y=x+2;
 }
}
```

This program is converted into bytecode, as shown here. Details on how to compile solidity code with examples will be given in the next chapter.

Runtime byte code

```
606060405260e060020a6000350463989e17318114601c575b6000565b34600057602960043
5603b565b604080519182525190819003602001 90f35b600281015b91905056
Opcodes PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE
PUSH1 0x0 JUMPI JUMPDEST PUSH1 0x45 DUP1 PUSH1 0x1A PUSH1 0x0 CODECOPY
PUSH1 0x0 RETURN PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0xE0 PUSH1 0x2 EXP
PUSH1 0x0 CALLDATALOAD DIV PUSH4 0x989E1731 DUP2 EQ PUSH1 0x1C JUMPI
JUMPDEST PUSH1 0x0 JUMP JUMPDEST CALLVALUE PUSH1 0x0 JUMPI PUSH1 0x29 PUSH1
0x4 CALLDATALOAD PUSH1 0x3B JUMP JUMPDEST PUSH1 0x40 DUP1 MLOAD SWAP2 DUP3
MSTORE MLOAD SWAP1 DUP2 SWAP1 SUB PUSH1 0x20 ADD SWAP1 RETURN JUMPDEST
PUSH1 0x2 DUP2 ADD JUMPDEST SWAP2 SWAP1 POP JUMP
```

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

Note that STOP is not an arithmetic operation but is categorized in this list of arithmetic operations due to the range of values (0s) it falls in.

Logical operations

Logical operations include operations that are used to perform comparisons and Boolean logic operations. The value of these operations is in the range of 0x10 to 0x1a.

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.

Environmental information

There are a total of 13 instructions in this category. These opcodes are used to provide information related to addresses, runtime environments, and data copy operations.

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.

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. The state is created or updated as a result of the interaction between accounts. Operations performed between and on the accounts represent state transitions. State transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. Transaction fee is calculated and the sending address is resolved using the signature. Furthermore, sender's account balance is checked and subtracted accordingly and nonce is incremented. An error is returned if the account balance is not enough.
3. Provide enough ether (gas price) to cover the cost of the transaction. This is charged per byte incrementally according to the size of the transaction.
4. In this step, the actual transfer of value occurs. The flow is from the sender's account to receiver's account. The account is created automatically if the destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.
5. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back with the exception of fee payment, which is paid to the miners.
6. Finally, the remainder (if any) of the fee is sent back to the sender as change and fee is paid to the miners accordingly. At this point, the function returns the resulting state.

Types of accounts

There are two types of accounts in Ethereum:

- Externally owned accounts
- Contract accounts

The first is **externally owned accounts** (**EOAs**) and the other is contract accounts. EOAs are similar to accounts that are controlled by a private key in bitcoin. Contract accounts are the accounts that have code associated with them along with the private key. An EOA has ether balance, is able to send transactions, and has no associated code, whereas a **Contract Account** (**CA**) has ether balance, associated code, and the ability to get triggered and execute code in response to a transaction or a message. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within contract accounts can be of any level of complexity. The code is executed by EVM by each mining node on the Ethereum network. In addition, contract accounts are able to maintain their own permanent state and can call other contracts. It is envisaged that in the serenity release, the distinction between externally owned accounts and contract accounts may be eliminated.

Ether

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

The denomination table is shown as follows:

Unit	Wei Value	Weis
Wei	1 Wei	1
Babbage	1e3 Wei	1,000
Lovelace	1e6 Wei	1,000,000
Shannon	1e9 Wei	1,000,000,000
Szabo	1e12 Wei	1,000,000,000,000
Finney	1e15 Wei	1,000,000,000,000,000
Ether	1e18 Wei	1,000,000,000,000,000,000

Fees are charged for each computation performed by the EVM on the blockchain. A detailed fee schedule is shown in the upcoming section.

Gas

Gas is required to be paid for every operation performed on the ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as some amount of Ether and is taken from the account balance of the transaction originator. A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block and the transaction originator will not get any refund.

Transaction cost can be estimated using the following formula:

$$Total\ cost = gasUsed * gasPrice$$

Here, *gasUsed* is the total gas that is supposed to be used by the transaction during the execution and *gasPrice* is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in Ether. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or less operations than originally intended and can result in consuming more or fewer gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and there is some remaining gas, then it is returned to the transaction originator.

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

Operation Name	Gas Cost
step	1
stop	0
suicide	0
sha3	30
sload	20
txdata	5
transaction	500
contract creation	53000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA3 operation can be calculated as follows:

- SHA3 costs 30 gas
- Current gas price is 25 GWei, which is 0.000000025 Ether
- Multiplying both: *0.000000025 * 30 = 0.00000075* Ether

In total, 0.00000075 Ether is the total gas that will be charged.

Fee schedule

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

- The computation of an operation
- For contract creation or message call
- Increase in the usage of memory

A list of instructions and various operations with the gas values has been provided previously in the chapter.

Messages

Messages, as defined in the yellow paper, are the data and value that are passed between two accounts. A message is a data packet passed between two accounts. This data packet contains data and value (amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (externally owned account) in the form of a transaction that has been digitally signed by the sender.

Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions; however, the main difference is that they are produced by the contracts, whereas transactions are produced by entities external (externally owned accounts) to the Ethereum environment.

A message consists of the components mentioned here:

1. Sender of the message
2. Recipient of the message
3. Amount of Wei to transfer and message to the contract address
4. Optional data field (Input data for the contract)
5. Maximum amount of gas that can be consumed

Messages are generated when `CALL` or `DELEGATECALL` Opcodes are executed by the contracts.

Calls

A call does not broadcast anything to the blockchain; instead, it is a local call to a contract function and runs locally on the node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. It is akin to a dry run. Calls are executed locally on a node and generally do not result in any state change. As defined in the yellow paper, this is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the virtual machine will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object, then the call passes any data returned from the virtual machine operation.

State is altered by transactions. These are created by external factors and are signed and then broadcasted to the Ethereum network.

Mining

Mining is the process by which new currency is added to the blockchain. This is an incentive for the miners to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

At a theoretical level, a miner performs the following functions:

1. Listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
2. Determines stale blocks called Uncles or Ommers and includes them in the block.
3. Updates the account balance with the reward earned from successfully mining the block.
4. Finally, a valid state is computed and block is finalized, which defines the result of all state transitions.

The current method of mining is based on Proof of Work, which is similar to that of bitcoin. When a block is deemed valid, it has to satisfy not only the general consistency requirements, but it must also contain the Proof of Work for a given difficulty.

The Proof of Work algorithm is due to be replaced with the Proof of Stake algorithm with the release of serenity. Considerable research work has been carried out in order to build the Proof of Stake algorithm suitable for the Ethereum Network.

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

Ethash

Ethash is the name of the Proof of Work algorithm used in Ethereum. Originally, this was proposed as the Dagger-Hashimoto algorithm, but much has changed since the first implementation and the PoW algorithm has now evolved into what's known as Ethash now. Similar to bitcoin, the core idea behind mining is to find a nonce that once hashed the result in a predetermined difficulty level. Initially, the difficulty was low when Ethereum was new and even CPU and single GPU mining was profitable to a certain extent, but that is no longer the case. Now either pooled mining is profitable, or large GPU mining farms are used for mining purposes.

Ethash is a memory-hard algorithm, which makes it difficult to be implemented on specialized hardware. As in bitcoin, ASICs have been developed, which have resulted in mining centralization over the years, but memory-hard Proof of Work algorithms are one way of thwarting this threat and Ethereum implements Ethash to discourage ASIC development for mining. This algorithm requires choosing subsets of a fixed resource called **DAG** (**Directed Acyclic Graph**) depending on the nonce and block headers. DAG is around 2 GB in size and changes every 30000 blocks. Mining can only start when DAG is completely generated the first time a mining node starts. The time between every 30000 blocks is around 5.2 days and is called epoch. This DAG is used as a seed by the Proof of Work algorithm called Ethash. According to current specifications, the epoch time is defined as 30,000 blocks.

The current reward scheme is 5 Ether for successfully finding a valid nonce. In addition to receiving 5 Ethers, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (Uncles) in the block. A maximum of two Uncles are allowed per block and are rewarded 7/8 of the normal block reward. In order to achieve a 12 second block time, block difficulty is adjusted at every block. The rewards are directly proportional to the miner's hash rate, which basically means how fast a miner can hash.

Mining can be performed by simply joining the Ethereum network and running an appropriate client. The key requirement is that the node should be fully synced with the main network before mining can start.

In the upcoming section, various methods of mining are mentioned.

CPU mining

Even though not profitable on the main net, CPU mining is still valuable on the test network or even a private network to experiment with mining and contract deployment. Private and test networks will be discussed with practical examples in the next chapter. A geth example is shown on how to start CPU mining here. Geth can be started with mine switch in order to start mining:

```
geth --mine --minerthreads <n>
```

CPU mining can also be started using the web 3 geth console. Geth console can be started by issuing the following command:

```
geth attach
```

After this, the miner can be started by issuing the following command, which will return true if successful, or false otherwise. Take a look at the following command:

```
Miner.start(4)
True
```

The preceding command will start the miner with four threads. Take a look at the following command:

```
Miner.stop
True
```

The preceding command will stop the miner. The command will return true if successful.

GPU mining

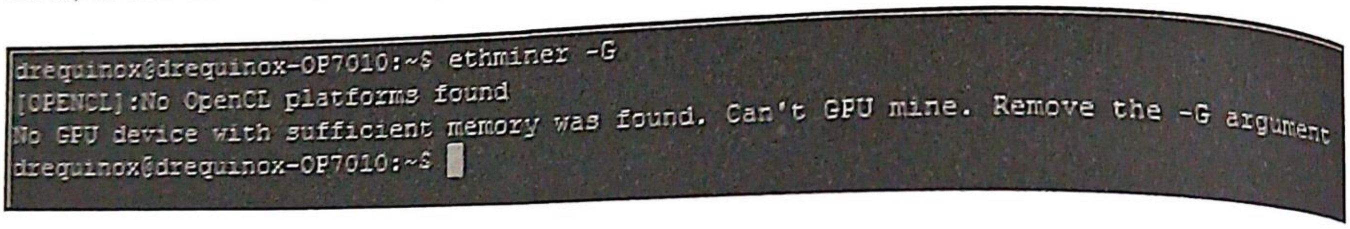
At a basic level, GPU mining can be performed easily by running two commands:

```
geth --rpc
```

Once geth is up and running and the blockchain is fully downloaded, Ethminer can be run in order to start mining. Ethminer is a standalone miner that can also be used in the farm mode to contribute to mining pools. It can be downloaded from `https://github.com/Genoil/cpp-ethereum/tree/master/releases`:

```
ethminer -G
```

Running with `G` switch assumes that the appropriate graphics card is installed and configured correctly. If no appropriate graphics cards are found, ethminer will return an error, as shown in the following screenshot:



```
drequinox@drequinox-OP7010:~$ ethminer -G
[OPENCL]:No OpenCL platforms found
No GPU device with sufficient memory was found. Can't GPU mine. Remove the -G argument
drequinox@drequinox-OP7010:~$ 
```

Error in case no appropriate GPUs can be found

GPU mining requires an AMD or Nvidia graphics card and an applicable OpenCL SDK. For NVidia chipset, it can downloaded from `https://developer.nvidia.com/cuda-downloads`. For AMD chipsets, it is available at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk`.

Once the graphics cards are installed and configured correctly, the process can be started by issuing the `ethminer -G` command.

Ethminer can also be used to run benchmarking, as shown in the following screenshot. There are two modes that can be invoked for benchmarking. It can either be CPU or GPU. The commands are shown here.

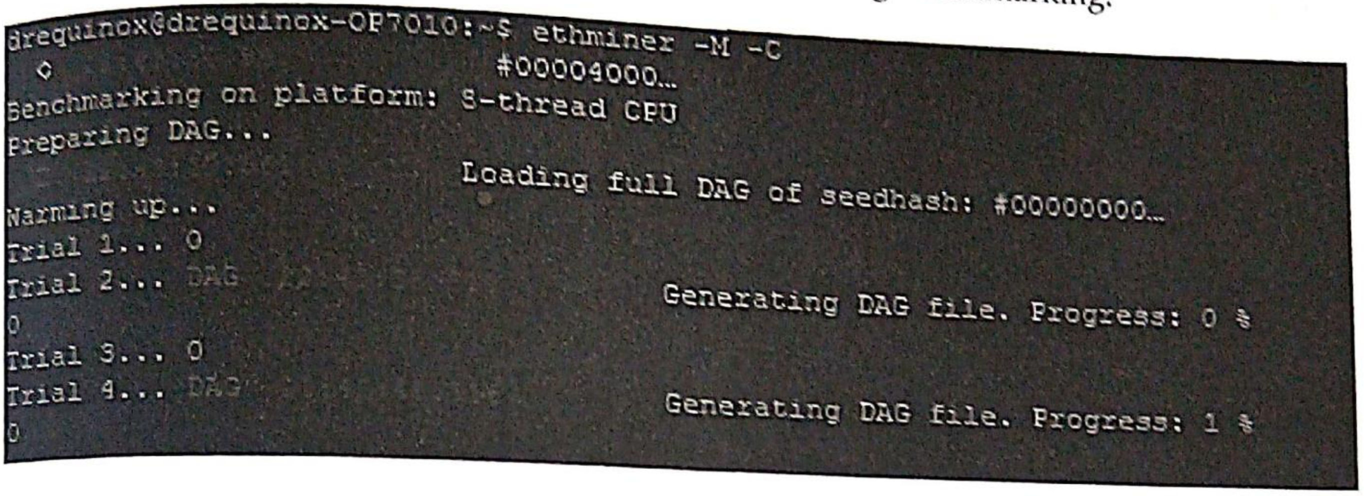
CPU benchmarking

```
$ ethminer -M -C
```

GPU benchmarking

```
$ ethminer -M -G
```

The following screenshot example is shown for CPU mining benchmarking:



```
drequinox@drequinox-OP7010:~$ ethminer -M -C
  ◊                              #00004000…
Benchmarking on platform: 8-thread CPU
Preparing DAG...
                                 Loading full DAG of seedhash: #00000000…
Warming up...
Trial 1... 0
Trial 2... DAG
                                              Generating DAG file. Progress: 0 %
0
Trial 3... 0
Trial 4... DAG
                                              Generating DAG file. Progress: 1 %
0
```

CPU benchmarking

The GPU device to be used can also be specified in the command line:

```
$ ethminer -M -G --opencl-device 1
```

As GPU mining is implemented using OpenCL AMD, chipset-based GPUs tend to work faster as compared to NVidia GPUs. Due to the high memory requirements (DAG creation), FPGAs and ASICs will not provide any major advantage over GPUs. This is done on purpose in order to discourage the development of specialized hardware for mining.

Mining rigs

As difficulty increased over time for mining Ether, mining rigs with multiple GPUs were starting to be built by the miners. A mining rig usually contains around five GPU cards, and all of them work in parallel for mining, thus improving the chances of finding valid nonces for mining.

Mining rigs can be built with some effort and are also available commercially from various vendors. A typical mining rig configuration includes the components discussed in the upcoming sections.

Motherboard

A specialized motherboard with multiple PCI-E x1 or x16 slots, for example, BIOSTAR Hi-Fi or ASRock H81, is required.

SSD hard drive

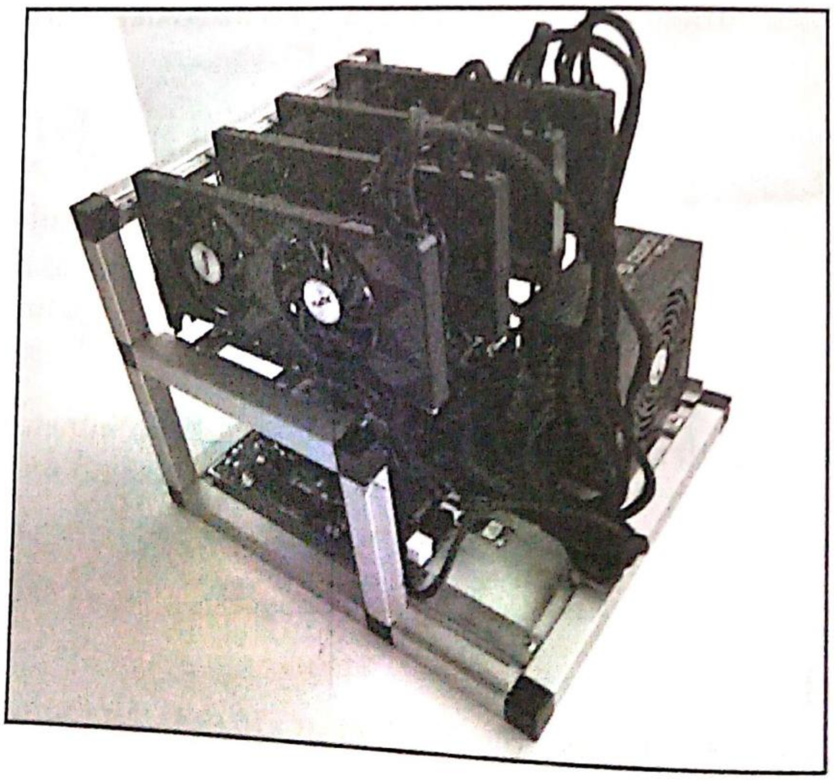
An SSD hard drive is required. The SSD drive is recommended because of its much faster performance over the analog equivalent. This will be mainly used to store the blockchain.

GPU

The GPU is the most important component of the rig as it is the main workhorse that will be used for mining. For example, it can be a Sapphire AMD Radeon R9 380 with 4 GB RAM.

Linux Ubuntu's latest version is usually chosen as the operating system for the rig. There is also another variant of Linux available, called EthOS (available at `http://ethosdistro.com/`), that is especially built for Ethereum mining and supports mining operations natively.

Finally, mining software such as Ethminer and geth are installed. Additionally, some remote monitoring and administration software is also installed so that rigs can be monitored and managed remotely, if required. It is also important to put appropriate air conditioning or cooling mechanisms in place as running multiple GPUs can generate a lot of heat. This also necessitates the need for using an appropriate monitoring software that can alert users if there are any problems with the hardware, for example, if the GPUs are overheating.



A mining rig for Ethereum for sale at eBay

Mining pools

There are many online mining pools that offer Ethereum mining. Ethminer can be used to connect to a mining pool using the following command. Each pool publishes its own instructions, but generally, the process of connecting to a pool is similar. An example from `ethereumpool.co` is shown here:

```
ethminer -C -F
http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e60c20eb1459
c@DrEquinox
```

```
drequinox@drequinox-OP7010:~$  ethminer -C -F http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e60c20eb1459c@DrEquinox
                                Getting work package...
```

Screenshot of ethminer

Clients and wallets

As Ethereum is under heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years. The following is a list of all main components, client software, and tools that are available with Ethereum. This list is provided in order to reduce the ambiguity around many tools and clients available for Ethereum. The list provided here also explains the usage and significance of various components.

Geth

This is the Go implementation of the Ethereum client.

Eth

This is the C++ implementation of the Ethereum client.

Pyethapp

This is the Python implementation of the Ethereum client.

Parity

This implementation is built using Rust and developed by EthCore. EthCore is a company that works on the development of the parity client. Parity can be downloaded from `https://ethcore.io/parity.html`.

Light clients

SPV clients download only a small subset of the blockchain. This allows low resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify the transactions. A complete ethereum blockchain and node are not required in this case and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to bitcoin SPV clients. There is a wallet available from Jaxx (`https://jaxx.io/`), which can be installed on iOS and Android, which provides the SPV (**Simple Payment Verification**) functionality.

Installation

The following installation procedure describes the installation of various Ethereum clients on Ubuntu systems. Instructions for other operating systems are available on Ethereum Wikis. As Ubuntu systems will be used in examples later on, only installation on Ubuntu has been described here.

Geth client can be installed by using the following command on an Ubuntu system:

```
> sudo apt-get install -y software-properties-common
> sudo add-apt-repository -y ppa:ethereum/ethereum
> sudo apt-get update
> sudo apt-get install -y ethereum
```

After installation is completed. Geth can be launched simply by issuing the `geth` command at the command prompt, as it comes preconfigured with all the required parameters to connect to the live Ethereum network (mainnet):

```
> geth
```

Eth installation

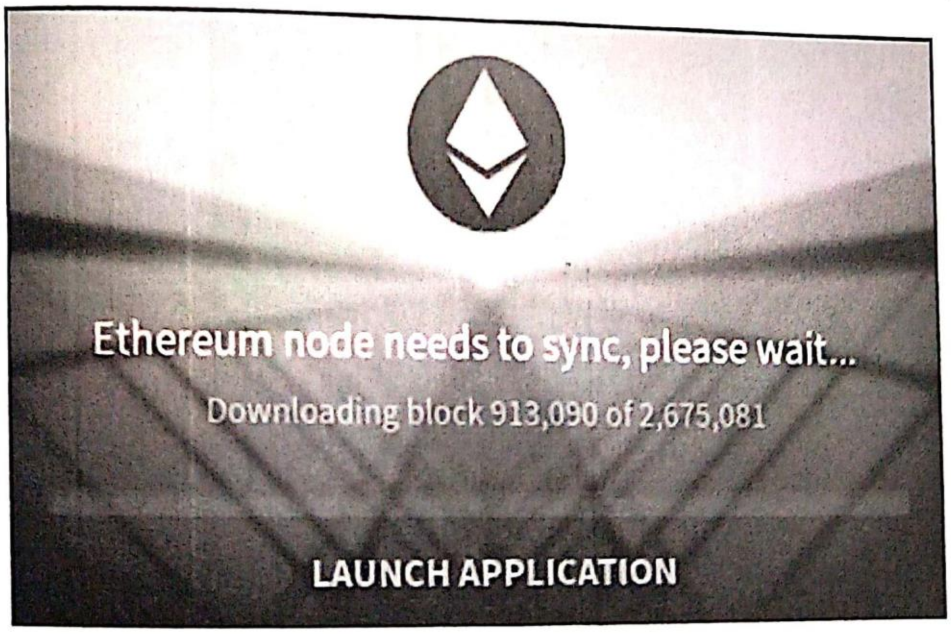
Eth is the C++ implementation of the Ethereum client and can be installed using the following command on Ubuntu:

```
> sudo apt-get install cpp-ethereum
```

Mist browser

Mist browser is a user-friendly interface for end users with a feature-rich graphical user interface that is used to browse DAPPS and for account management and contract management. Mist installation is covered in the next chapter.

When Mist is launched for the first time, it will initialize geth in the background and will sync with the network. It can take from a few hours to a few days depending on the speed and type of the network to fully synchronize with the network. If TestNet is used, then syncing completes relatively faster as the size of TestNet (Ropsten) is not as big as MainNet. More information on how to connect to TestNet will be provided in the next chapter.


Ethereum node needs to sync, please wait...
Downloading block 913,090 of 2,675,081
LAUNCH APPLICATION

Mist browser starting up and syncing with the main network

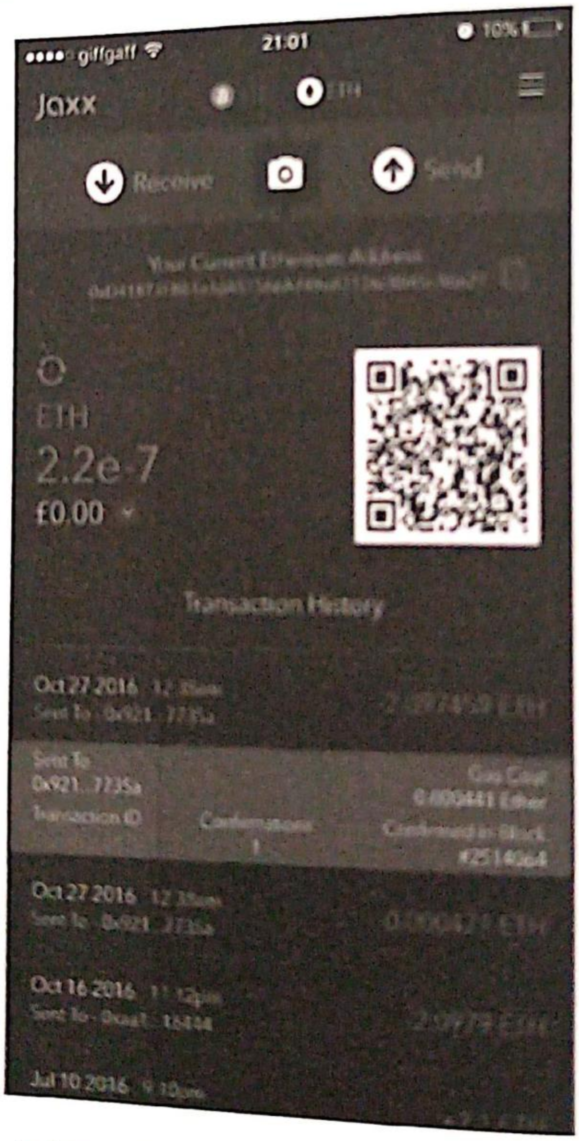
Mist browser is not a wallet; in fact, it is a browser of DAPPS and provides a user-friendly user interface for the creation and management of contracts, accounts, and browsing decentralized applications. Ethereum wallet is a DAPP that is released with Mist.

Wallet is a generic program that can store private keys and associated accounts and, based on the addresses stored within it, it can compute the existing balance of Ether associated with the addresses by querying the blockchain.

Other wallets include but are not limited to MyEtherWallet, which is an open source ether wallet developed in JavaScript. MyEtherWallet runs in the client browser. This is available at `https://www.myetherwallet.com`.

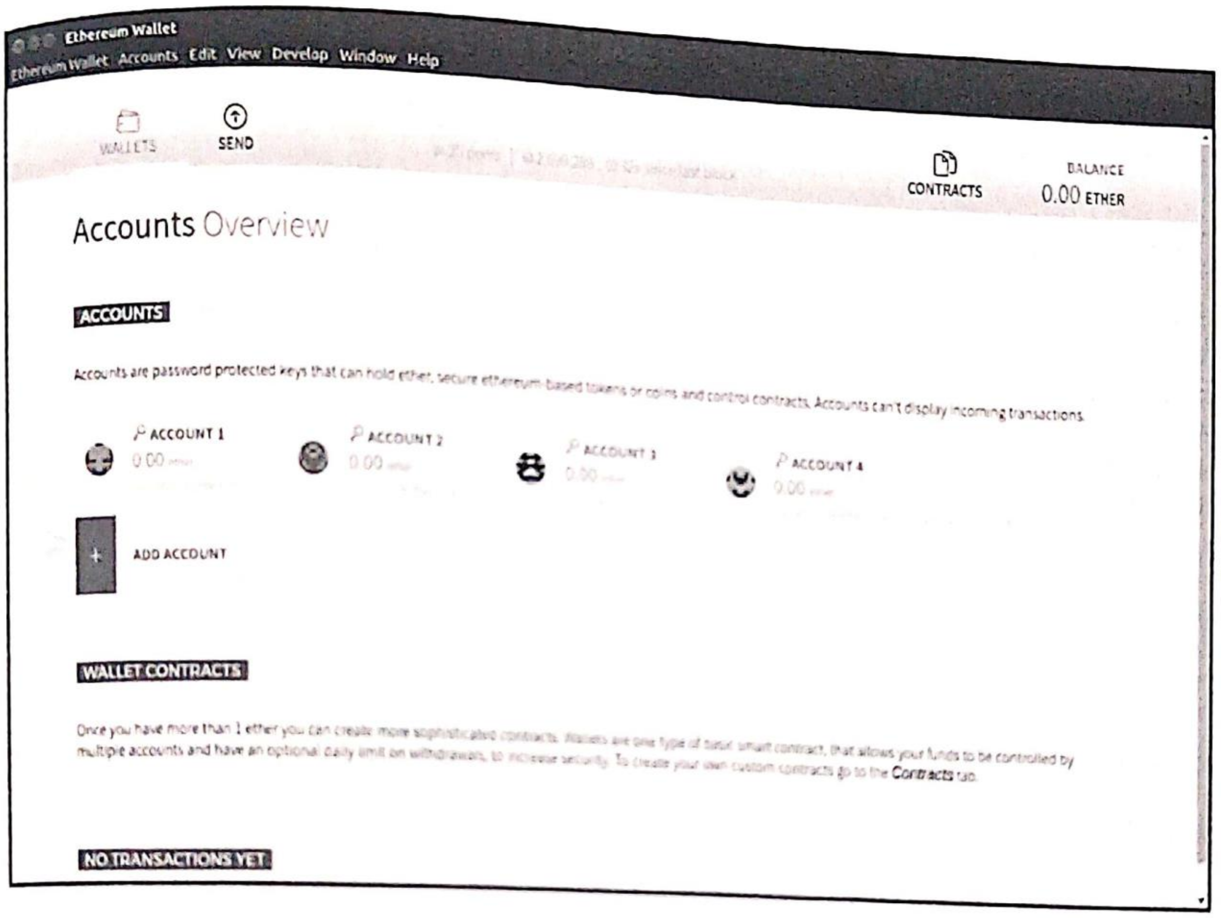
Icebox is developed by Consensys. This is a cold storage browser that provides secure storage of Ether. It depends on whether the computer on which Icebox is run is connected to the Internet or not.

Various wallets are available for ethereum for desktop, mobile, and web platforms. A popular Ethereum iOS Wallet named Jaxx is shown in the following image:



Jaxx Ethereum wallet for iOS showing transactions and current balance

Once the blockchain is synchronized, Mist will launch and show the following interface. In this example, four accounts are displayed with no balance:



Accounts Overview

ACCOUNTS

Accounts are password protected keys that can hold ether, secure ethereum-based tokens or coins and control contracts. Accounts can't display incoming transactions.

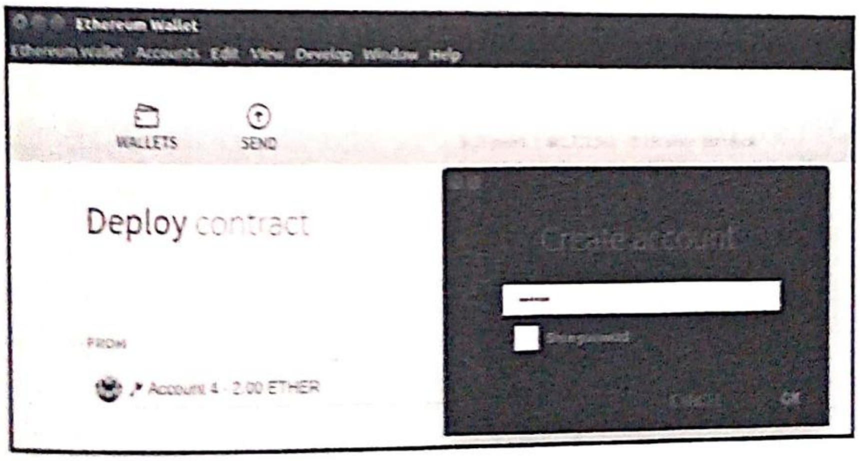
WALLET CONTRACTS

Once you have more than 1 ether you can create more sophisticated contracts. Wallets are one type of basic smart contract, that allows your funds to be controlled by multiple accounts and have an optional daily limit on withdrawals, to increase security. To create your own custom contracts go to the *Contracts* tab.

NO TRANSACTIONS YET

Mist browser

A new accounts can be created in a number of ways. In the Mist browser, it can be created by clicking on the **Accounts** menu and selecting the **New account** or by clicking on the **Add account** option in the Mist Accounts Overview screen.



Add new account

The account will need a password to be set, as shown in the preceding figure; once the account is set up, it will be displayed in the accounts overview section of the Mist browser.

Accounts can also be added via the command line using the geth or parity command-line interface. This process is shown in the next section.

Geth

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
Repeat passphrase:
Address: {21c2b52e18353a2cc8223322b33559c1d900c85d}
drequinox@drequinox-OP7010:~$
```

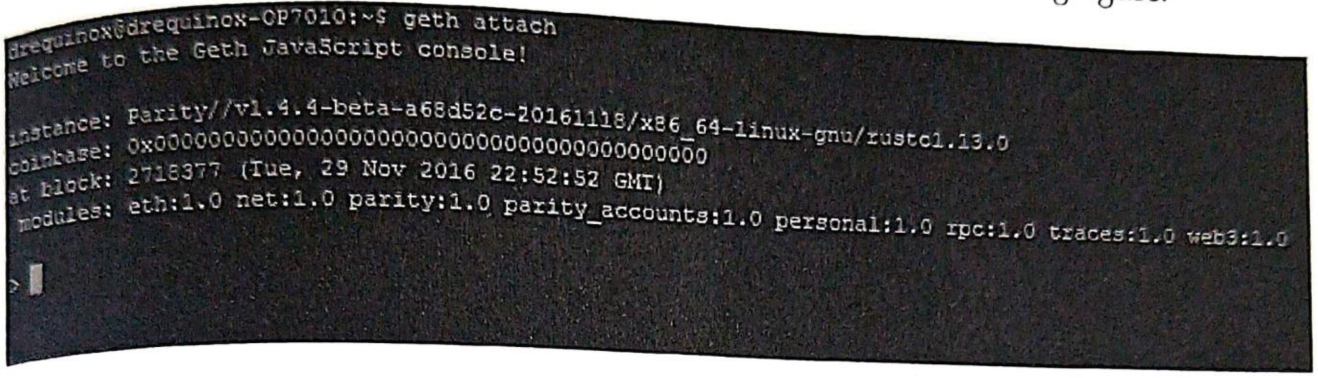
The list of accounts can be shown using geth using the following command:

```
$ geth account list

Account #0: {11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35}
/home/drequinox/.ethereum/keystore/UTC--2016-05-07T13-04-15.175558799Z-
-11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35

Account #1: {e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63}
/home/drequinox/.ethereum/keystore/UTC--2016-05-10T19-16-11.952722205Z--
e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63

Account #2: {21c2b52e18353a2cc8223322b33559c1d900c85d}
/home/drequinox/.ethereum/keystore/UTC--2016-11-29T22-48-09.825971090Z-
-21c2b52e18353a2cc8223322b33559c1d900c85d
```

The geth console

The geth JavaScript console can be used to perform various functions. For example, an account can be created by attaching geth.

Geth can be attached with the running daemon, as shown in the following figure:



```
drequinox@drequinox-OP7010:~$ geth attach
Welcome to the Geth JavaScript console!

instance: Parity//v1.4.4-beta-a68d52c-20161118/x86_64-linux-gnu/rustc1.13.0
coinbase: 0x0000000000000000000000000000000000000000
at block: 2718377 (Tue, 29 Nov 2016 22:52:52 GMT)
 modules: eth:1.0 net:1.0 parity:1.0 parity_accounts:1.0 personal:1.0 rpc:1.0 traces:1.0 web3:1.0

>
```

Once geth is successfully attached with the running instance of the ethereum client (in this case, parity), it will display command prompt '>', which provides an interactive command line interface to interact with the ethereum client using JavaScript notations.

For example, a new account can be added using the following command in the geth console:

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xc64a728a67ba67048b9c160ec39bacc5626761ce"
>
```

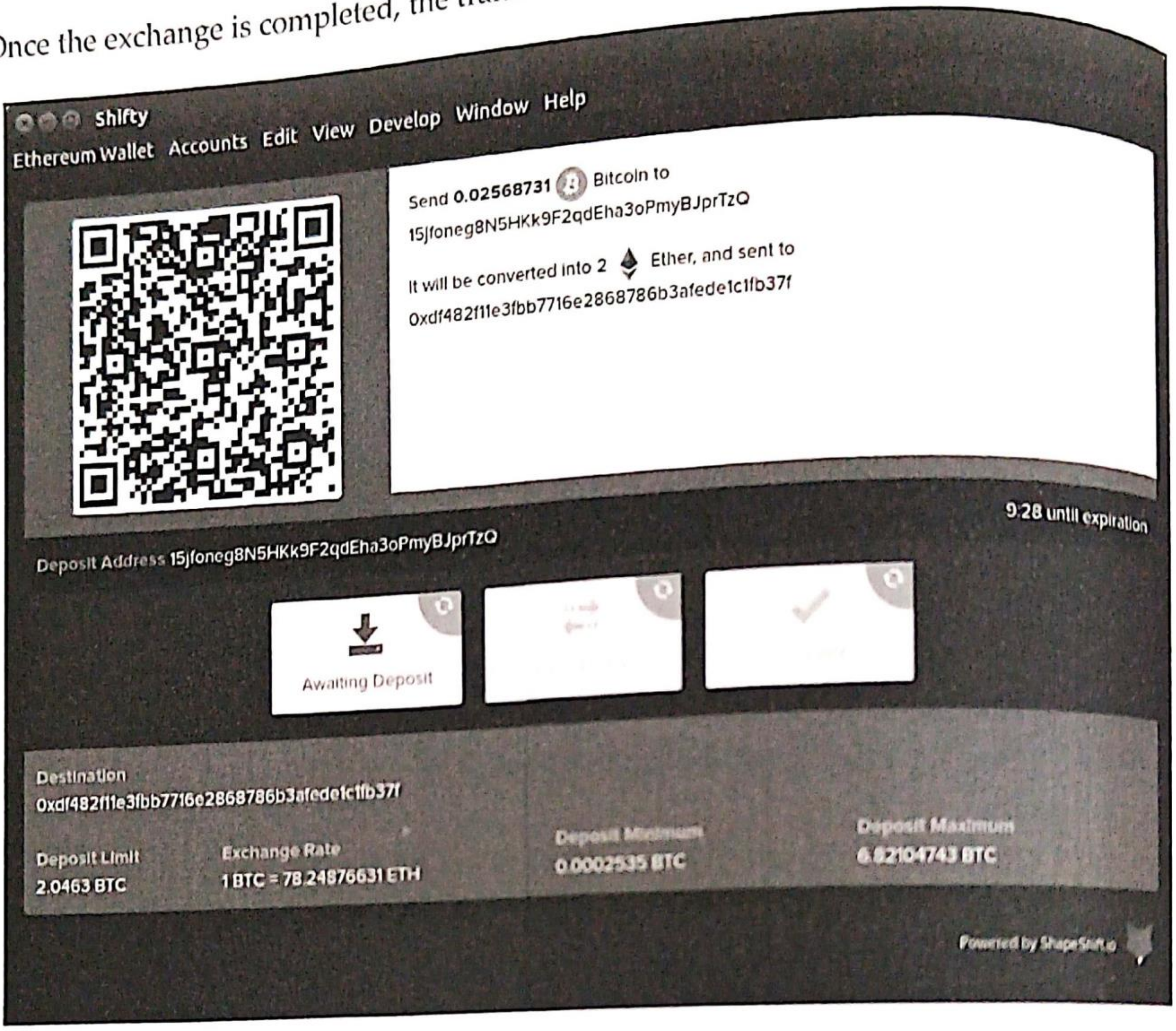
The list of accounts can also be displayed similarly:

```
> eth.accounts
["0x024a20cc5feba7f3dc3776075b3e60c20eb1459c",
"0x11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35",
"0xdf482f11e3fbb7716e2868786b3afede1c1fb37f",
"0xe49668b7ffbf031bbbdab7a222bdb38e7e3e1b63",
"0xf9834defb35d24c5a61a5fe745149e9470282495"]
```

Funding the account with bitcoin

This option is available with the Mist browser by clicking on the account and then selecting the option to fund the account. The backend engine used for this operation is shapeshift.io and can be used to fund the account from bitcoin or other currencies, including the fiat currency option as well.

Once the exchange is completed, the transferred Ether will be available in the account.



Send 0.02568731 Bitcoin to
15jfoneg8N5HKk9F2qdEha3oPmyBJprTzQ

It will be converted into 2 Ether, and sent to
0xdf482f11e3fbb7716e2868786b3afede1c1fb37f

9:28 until expiration

Deposit Address 15jfoneg8N5HKk9F2qdEha3oPmyBJprTzQ

Destination
0xdf482f11e3fbb7716e2868786b3afede1c1fb37f

Deposit Limit
2.0463 BTC

Exchange Rate
1 BTC = 78.24876631 ETH

Deposit Minimum
0.0002535 BTC

Deposit Maximum
6.82104743 BTC

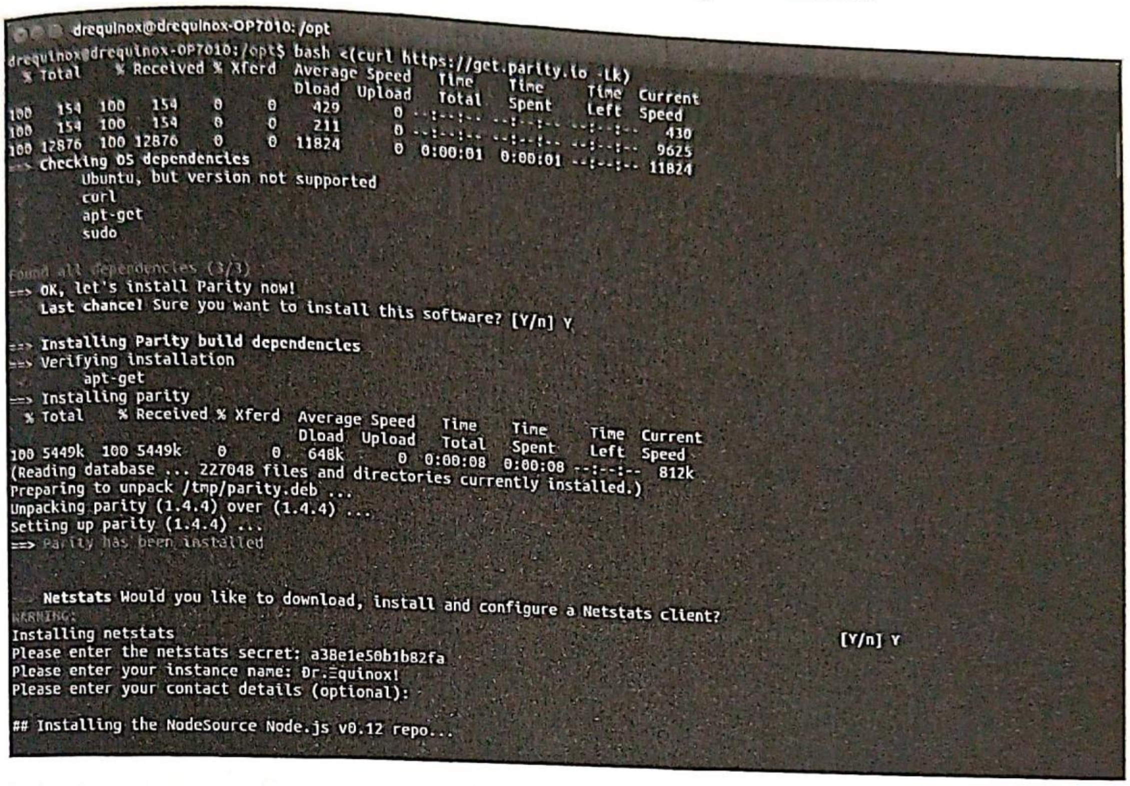
Parity installation

Parity is another implementation of the Ethereum client. It has been written using the Rust programming language. The main aim behind the development of parity is high performance, small footprint, and reliability. Parity can be installed using the following commands on an Ubuntu or Mac system:

```
bash <(curl https://get.parity.io -Lk)
```

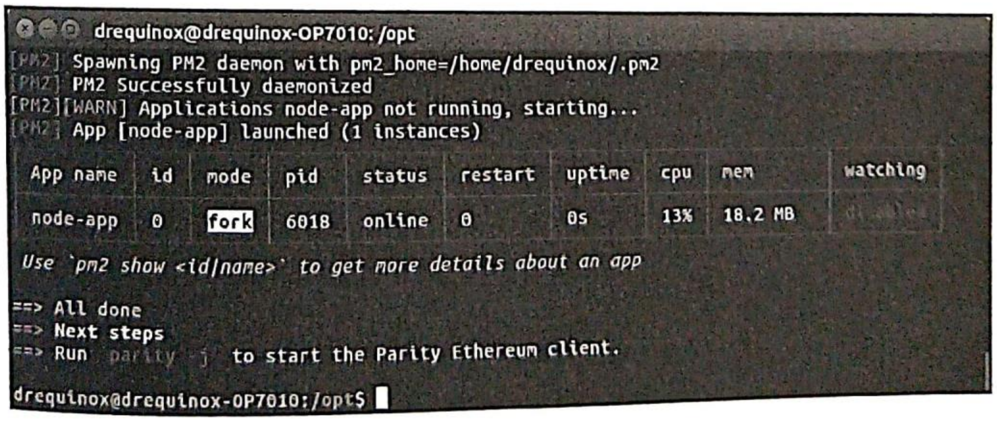
This will initiate the download and installation of the parity client. After the installation of parity is completed, the installer will also offer the installation of the netstats client. The netstat client is a daemon that runs in the background and collects important statistics and displays them on `stats.ethdev.com`.

A sample installation of parity is shown in the following screenshot:



```
drequinox@drequinox-OP7010: /opt
drequinox@drequinox-OP7010:/opt$ bash <(curl https://get.parity.io -Lk)
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   154  100   154    0     0    429      0 --:--:-- --:--:-- --:--:--   430
100   154  100   154    0     0    211      0 --:--:-- --:--:-- --:--:--  9625
100 12876  100 12876    0     0  11824      0  0:00:01  0:00:01 --:--:-- 11824
==> Checking OS dependencies
        Ubuntu, but version not supported
        curl
        apt-get
        sudo

Found all dependencies (3/3)
==> OK, let's install Parity now!
    Last chance! Sure you want to install this software? [Y/n] Y

==> Installing Parity build dependencies
==> Verifying installation
        apt-get
==> Installing parity
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5449k  100 5449k    0     0   648k      0  0:00:08  0:00:08 --:--:--  812k
(Reading database ... 227048 files and directories currently installed.)
Preparing to unpack /tmp/parity.deb ...
Unpacking parity (1.4.4) over (1.4.4) ...
Setting up parity (1.4.4) ...
==> Parity has been installed


    Netstats Would you like to download, install and configure a Netstats client?
WARNING:
Installing netstats                                                      [Y/n] Y
Please enter the netstats secret: a38e1e50b1b82fa
Please enter your instance name: Dr.Equinox!
Please enter your contact details (optional):

## Installing the NodeSource Node.js v0.12 repo...
```

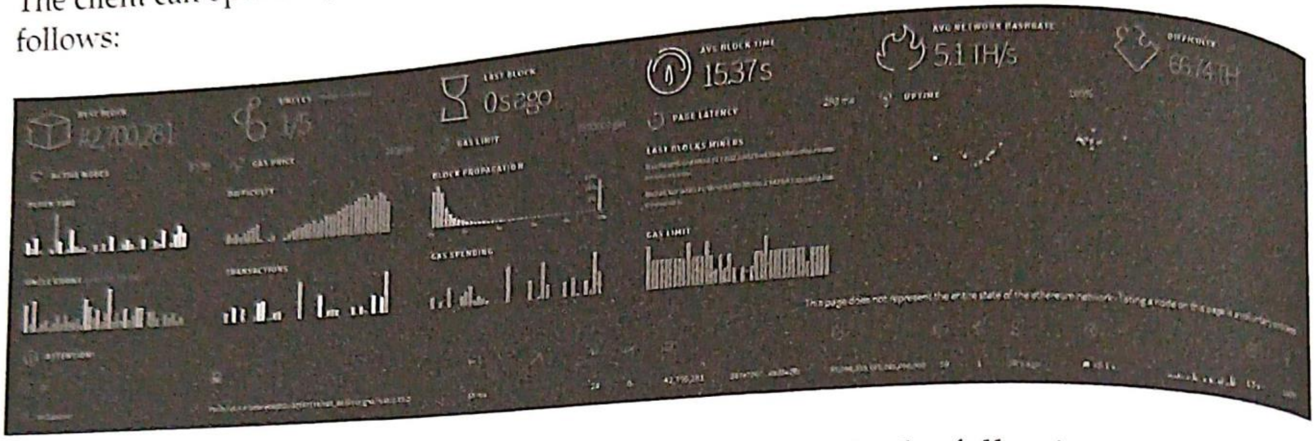
Once the installation is completed successfully, the following message is displayed. Ethereum parity node can then be started using `parity -j`. If compatibility with geth is required in order to use Ethereum wallet (Mist browser) with parity, then the `parity -geth` command should be used to run parity. This will run parity in compatibility mode with the geth client and will consequently allow Mist to run on top of parity.



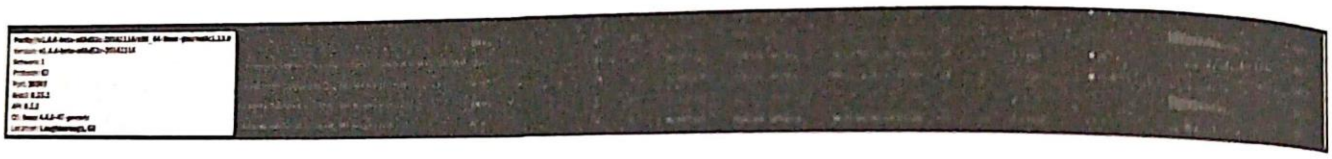
```
drequinox@drequinox-OP7010: /opt
[PM2] Spawning PM2 daemon with pm2_home=/home/drequinox/.pm2
[PM2] PM2 Successfully daemonized
[PM2][WARN] Applications node-app not running, starting...
[PM2] App [node-app] launched (1 instances)
│ App name │ id │ mode │ pid  │ status │ restart │ uptime │ cpu │ mem     │ watching │
│ node-app │ 0  │ fork │ 6018 │ online │ 0       │ 0s     │ 13% │ 18.2 MB │ disabled │
 Use `pm2 show <id|name>` to get more details about an app
==> All done
==> Next steps
==> Run   parity -j   to start the Parity Ethereum client.

drequinox@drequinox-OP7010:/opt$ █
```

Parity installation

The client can optionally be listed on `https://ethstats.net/`. An example is shown as follows:



All connected clients are listed on the ethstats.net, as shown in the following screenshot. These clients are listed with relevant attributes, such as the node name, node type, latency, mining status, number of peers, number of pending transactions, last block, difficultly, block transactions, and number of Uncles.


[illegible]

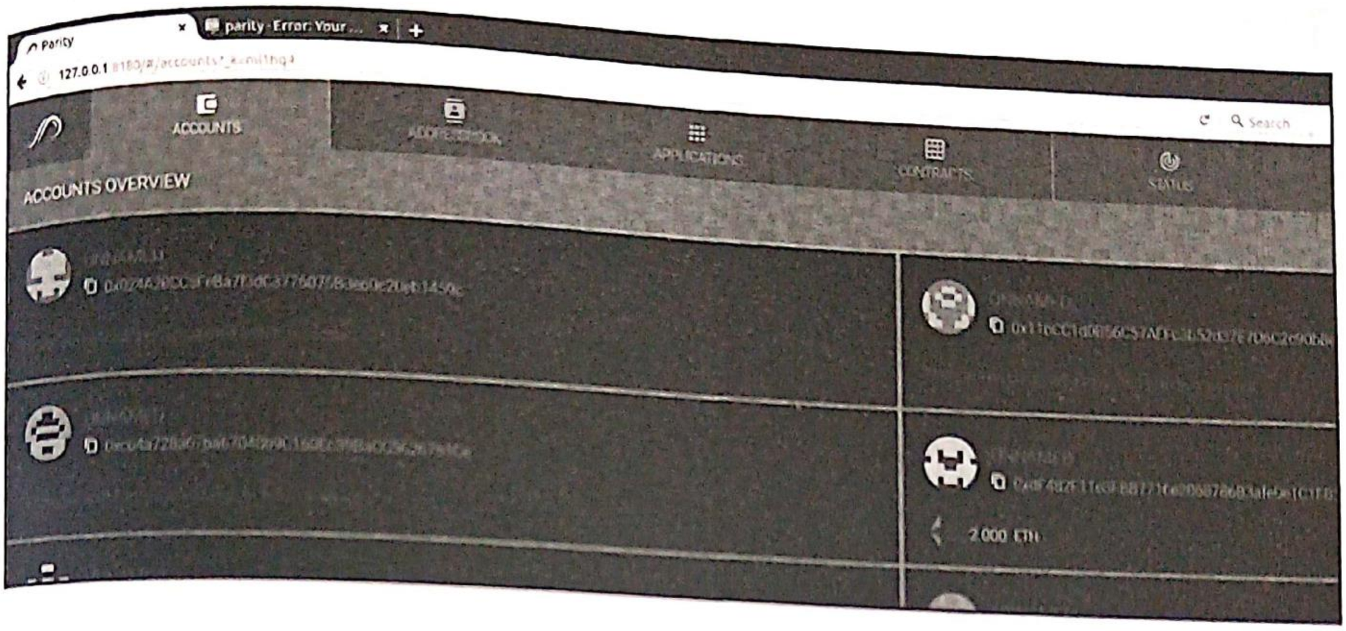
Client listed on https://ethstats.net/

Parity also offers a user-friendly web interface from where various tasks, such as account management, address book management, DAPP management, contract management, and status and signer operations, can be managed.

This is accessible by issuing the following command:

```
$ parity ui
```

This will bring up the interface shown as follows:



Parity user interface

If parity is running in the geth compatibility mode, the parity UI is disabled. In order to enable the UI along with geth compatibility, the following command can be used:

```
$ parity --geth --force-ui
```

The preceding command will start parity in the geth compatibility mode and also enable the web user interface.

Creating accounts using the parity command line

The following command can be used to create a new account using parity:

```
$ parity account new
Please note that password is NOT RECOVERABLE.
Type password:
Repeat password:
2016-11-30 02:18:55 UTC c8c92a910cfbce2e655c88d37a89b6657d1498fb
```

The Ethereum network

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

MainNet

MainNet is the current live network of ethereum. The current version of MainNet is homestead.

TestNet

TestNet is also called Ropsten and is the test network for the Ethereum blockchain. This blockchain is used to test smart contracts and DApps before being deployed to the production live blockchain. Moreover, being a test network, it allows experimentation and research.

Private net(s)

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

More discussion on how to connect to test net and how to set up private nets will be discussed in the next chapter.

Supporting protocols

There are various supporting protocols that are in development in order to support the complete decentralized ecosystem. This includes whisper and Swarm protocols. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized in order to achieve a complete decentralized ecosystem. This includes decentralized storage and decentralized messaging. Whisper, being developed for ethereum, is a decentralized messaging protocol, whereas Swarm is a decentralized storage protocol. Both of these technologies are being developed currently and have been envisaged to provide the basis for a fully decentralized web. In the following section, both technologies are discussed in detail.

Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the ethereum network. In essence, whisper is a communication protocol that nodes use in order to communicate with each other. The data and routing of messages are encrypted within whisper communications. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required. Whisper is also designed to provide a communication layer that cannot be traced and provides "dark communication" between parties. Blockchain can be used for communication, but that is expensive and consensus is not really required for messages exchanged between nodes. Therefore, whisper can be used as a protocol that allows

Whisper is already available with geth and can be enabled using the `--shh` option while running the geth ethereum client.

Swarm

Swarm is being developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to-peer storage network. Files in this network are addressed by the hash of their content. This is in contrast to the traditional centralized services, where storage is available at a central location only. This is developed as a native base layer service for the Ethereum web 3.0 stack. Swarm is integrated with DevP2P, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a **DDOS (Distributed Denial of service)**-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0. Both whisper and Swarm are under development and, even though Proof of Concept and alpha code has been released for Swarm, there is no stable production version available yet.