Altcoins, Ethereum projects, and More – Bonus Content

Welcome to the bonus online content for Mastering Blockchain – Third Edition! This content complements the main text, with the intention to enhance the reader learning experience. The material covered here has not been included in the core book for reasons of brevity and scope, but it contains valuable information, and has thus been included here for reference and further learning. We'll list and briefly describe the topics covered in these pages here, as an idea of what will be covered in this package. The topics are best read in conjunction with the chapters specified here, but can also be accessed as a reference guide for various topics at any point!

- **Bitcoin forks**: It is recommended that this topic is read after *Chapter 7*, *Bitcoin Network and Payments*. It covers alternative Bitcoin projects such as Bitcoin Cash and Bitcoin Unlimited.
- **Alternative coins**: This is intended to be read in conjunction with *Chapter 9*, *Alternative Coins*. This topic covers some important cryptocurrency projects such as Zcash and Primecoin. We cover in detail, among other things, how various clients can be installed, and how mining can be performed.
- **Ethereum networks, trading and investment**: It's recommended to read this material with *Chapter 11*, *Ethereum 101*. This topic briefly introduces the different types of Ethereum networks and some technical details. Also, we introduce some basics regarding the trading and investment of ether, the Ethereum cryptocurrency.
- **Ethereum EVM opcodes**: This topic should be read in conjunction with *Chapter 12*, *Further Ethereum*. Here, we list all EVM opcodes, with descriptions and some technical details, such as their opcode representation and gas consumption. This information can be extremely beneficial when writing and debugging smart contracts.
- **Ethereum block explorer**: These pages should be read after the completion of *Chapter 13*, *Ethereum Development Environment*. This topic covers how to use a block explorer on the Ethereum network, which helps to monitor the blockchain network by providing vital information about blocks and transactions in a simple dashboard format.
- **IDEAP project**: This decentralized application project is intended to be completed after reading *Chapter 15*, *Introducing Web3*. In this project, readers will make use of technologies such as Solidity, HTML, and JavaScript, and use tools such as Ganache, Truffle, and Drizzle, to develop a complete DApp project. The core idea behind this program is to provide a Proof of Idea service to keep a record of ideas. This record can then be used as proof that at a specific time in the past, the claimant has had access to a particular piece of information. This information can be beneficial for patent documents.

- **Running Ethereum 2.0 clients**: This project should be completed after reading *Chapter 16*, *Serenity*. In it, readers will install and run an Ethereum 2.0 beacon chain client, and monitor it on the network using eth2stats.
- **Alternative blockchains, projects and tools**: This material is intended to be read with *Chapter 22*, *Current Landscape and What's Next*. This topic provides a list of some noteworthy blockchains and projects that have emerged over the last few years. This section also covers a few miscellaneous tools that can be used in blockchain development.

Let's start with an exploration of a couple of innovative Bitcoin protocols, before looking deeper into some of the alternative cryptocurrencies available today.

Chapter 7

Bitcoin Cash

There have been a number of updates introduced with **Bitcoin Cash** (**BCH**), primarily the increased the block limit to 8 MB. This change immediately increased the number of transactions that can be processed in one block to a much larger number compared to the 1 MB limit in the original Bitcoin protocol. BCH uses **Proof of Work** (**PoW**) as a consensus algorithm, and the mining hardware is still ASIC-based. It also provides replay protection and wipe-out protection, which means that because BCH uses a different hashing algorithm, it prevents it being replayed on the Bitcoin blockchain. It also has a different type of signature compared to Bitcoin to differentiate between two blockchains.

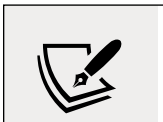
The Bitcoin Cash wallet and relevant information is available on their website: `https://www.bitcoincash.org`.

Bitcoin Unlimited

Bitcoin Unlimited (**BU**) increases the size of the block without setting a hard limit. Instead, miners come to a consensus on the block size cap over a period of time. Other concepts such as Xtreme thin blocks and parallel validation have also been proposed in Bitcoin Unlimited.

Xtreme thin blocks allow for a faster block propagation between Bitcoin nodes. In this scheme, the node requesting blocks sends a `getdata` request, along with a bloom filter, to another node. The purpose of this bloom filter is to filter out the transactions that already exist in the **memory pool** (**mempool**) of the requesting node. The node then sends back a **thin block** only containing the missing transactions. This fixes an inefficiency in Bitcoin whereby transactions are regularly received twice—once at the time of broadcast by the sender and then again when a mined block is broadcasted with the confirmed transaction.

Parallel validation allows nodes to validate more than one block, along with new incoming transactions, in parallel. This mechanism is in contrast to Bitcoin, where a node, during its validation period after receiving a new block, cannot relay new transactions or validate any blocks until it has accepted or rejected the block.



BU's client is available for download at `https://www.bitcoinunlimited.info`.

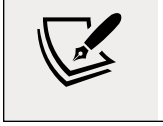
Chapter 9

Namecoin

Namecoin is the first fork of the Bitcoin source code. The key idea behind Namecoin is not to produce an altcoin but instead to provide improved decentralization, censorship resistance, privacy, security, and faster decentralized naming. Decentralized naming services are intended to respond to inherent limitations such as slowness and centralized control in the traditional **Domain Name System** (**DNS**) protocols used on the internet. Namecoin is also the first solution to Zooko's triangle, which was briefly discussed in *Chapter 1*, *Blockchain 101*.

Namecoin is used to essentially provide a service to register a key-value pair. One major use case of Namecoin is that it can provide a decentralized **Transport Layer Security** (**TLS**) certificate validation mechanism, driven by a blockchain-based distributed and decentralized consensus.

It is based on the same technology introduced with Bitcoin, but with its own blockchain and wallet software.



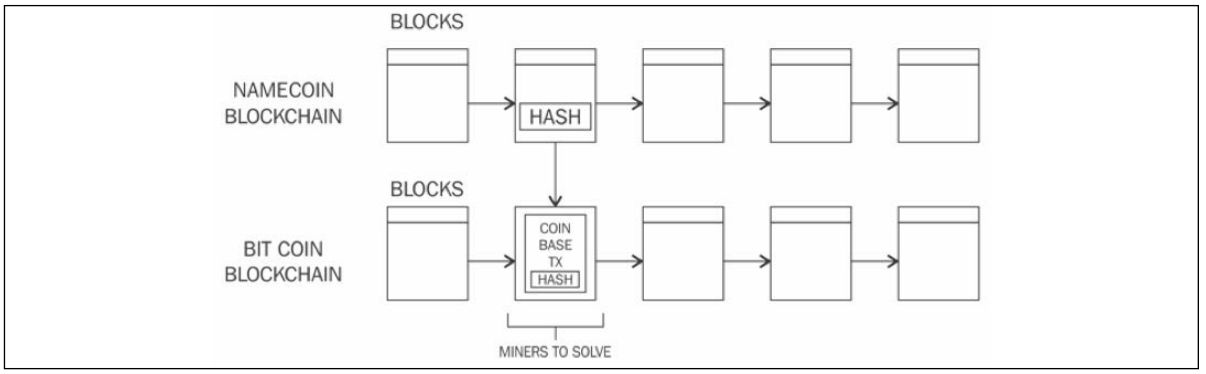
The source code for the Namecoin core is available at `https://github.com/namecoin/namecoin-core`.

In summary, Namecoin provides the following three services:

1. Secure storage and transfer of names (keys)
2. Attachment of some value to the names by attaching up to 520 bytes of data
3. Production of a digital currency (Namecoin)

Namecoin also, for the first time, introduced merged mining, which allows a miner to mine on more than one chain simultaneously. The idea is simple but very effective: miners create a Namecoin block and produce a hash of that block. Then, the hash is added to a Bitcoin block and miners solve that block at equal to or greater than the Namecoin block difficulty to prove that enough work has been contributed toward solving the Namecoin block.

The coinbase transaction is used to include the hash of the transactions from Namecoin (or any other altcoin if you've merged mining with that coin). The mining task is to solve Bitcoin blocks whose `scriptSig` coinbase contains a hash pointer to Namecoin (or any other altcoin) block. This is shown in the following diagram:



Merged mining visualization

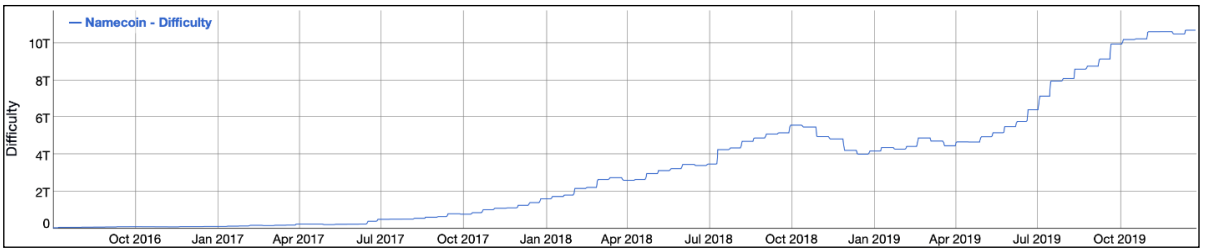
If a miner manages to solve a hash at the Bitcoin blockchain difficulty level, the Bitcoin block is built and becomes part of the Bitcoin network. In this case, the Namecoin hash is ignored by the Bitcoin blockchain. On the other hand, if a miner solves a block at the Namecoin blockchain difficulty level, a new block is created in the Namecoin blockchain. The core benefit of this scheme is that all the computational power spent by the miners contributes toward securing both Namecoin and Bitcoin.

Trading Namecoin

The current (June, 2020) market cap of Namecoin is US$ 6,656,288. It can be bought and sold at various exchanges. A list of exchanges where Namecoin can be traded is available here: `https://www.namecoin.org/exchanges/`.

Obtaining Namecoin

Even though Namecoin can be mined independently, they are usually mined as part of Bitcoin mining by utilizing the merged mining technique, as explained earlier. This way, Namecoin can be mined as a by-product of Bitcoin mining. Solo mining is no longer profitable, as is evident from the following difficulty graph; instead, it is recommended to use merged mining, use a mining pool, or even use a cryptocurrency exchange to buy Namecoin:



Namecoin difficulty, as shown at https://bitinfocharts.com/comparison/diffculty-nmc.html (since October 2016)

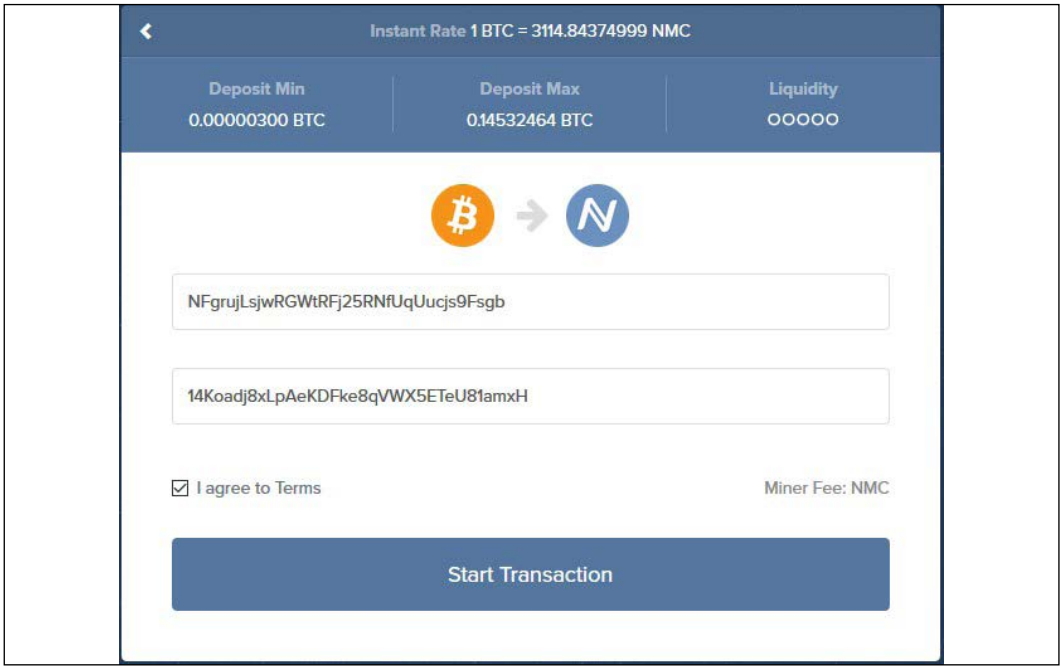
Various mining pools, such as `https://slushpool.com`, also offer the option of merged mining. This allows a miner to mine primarily Bitcoin but also, as a result, earn Namecoin too.

Another method that can be used to quickly get some Namecoins is to swap your existing coins with Namecoins; for example, if you already have some bitcoins or another cryptocurrency that can be used to exchange with Namecoin.

An online service, `https://shapeshift.io/`, is available that provides this service. This service allows conversion from one cryptocurrency to another using a simple user-friendly interface.

For example, paying BTC to receive Namecoin is done as follows:

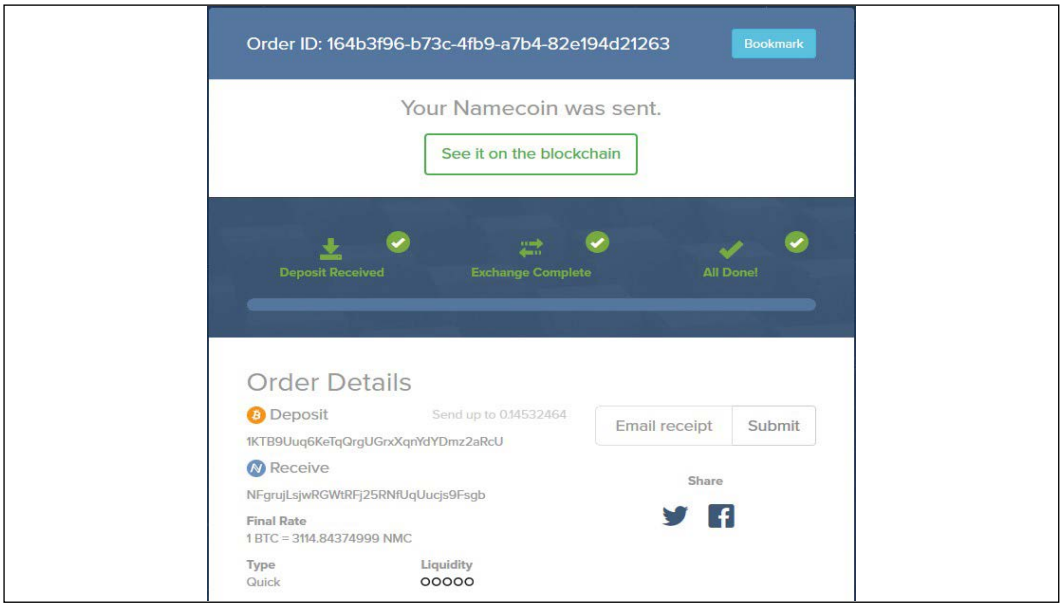
1. First, the deposit coin is selected, which in this case is Bitcoin, and the coin to be received is selected, which is Namecoin in this case. In the top edit box, the Namecoin address where you want to receive the exchanged Namecoin is entered. In the second edit box, at the bottom, the Bitcoin refund address is entered, where the coins will be returned to in case the transaction fails for any reason.
2. The exchange rate and miner fee are calculated instantly as soon as the deposit and exchange currency are chosen. The exchange rate is driven by the market conditions, whereas the miner fee is calculated algorithmically based on the target currency chosen and what the target network's miner would charge:



Bitcoin to Namecoin exchange

3. Once **Start Transaction** is clicked, the transaction starts and instructs the user to send the bitcoin to a specific Bitcoin address. When the user sends the required amount, the conversion process starts, as shown in the following screenshot.

This whole process takes a few minutes:

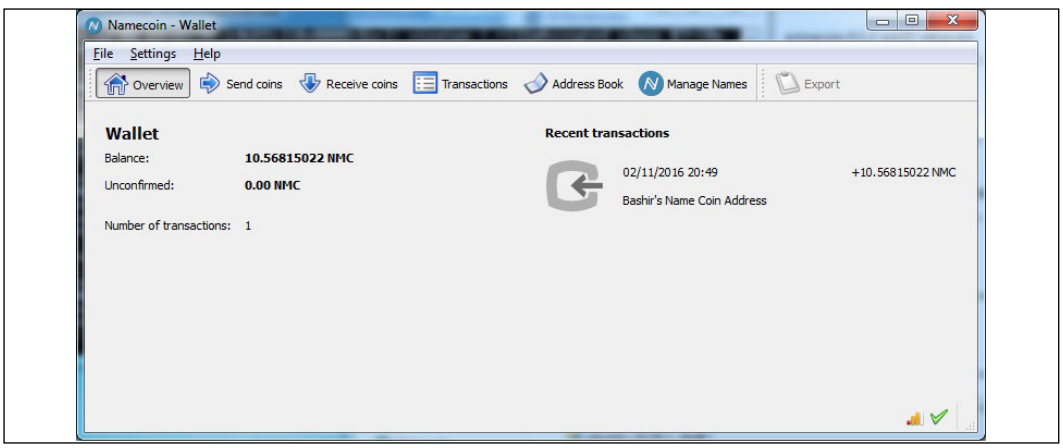


Notification of Namecoin delivery

The preceding screenshot shows that after sending the deposit, the exchange occurs and finally an **All Done!** message is displayed, indicating that the exchange has been successful.

4. A few other order details are displayed on the page, such as what currency was deposited and what was received after the exchange. In this case, it is a Bitcoin to Namecoin exchange. It's also worth noting that relevant addresses are also displayed under each coin icon. There are a few other options, such as **Email receipt**, which can be invoked to receive a receipt of the transaction via email.

   When the process completes, the transactions can be viewed in the Namecoin wallet, as shown here:



Namecoin wallet

5. It may take some time (usually around 1 hour) to confirm the transactions; until that time, it is not possible to use the Namecoins to manage names. However, when Namecoins are available in the wallet, the **Manage Names** option can be used to generate Namecoin records.

We'll cover this process next.

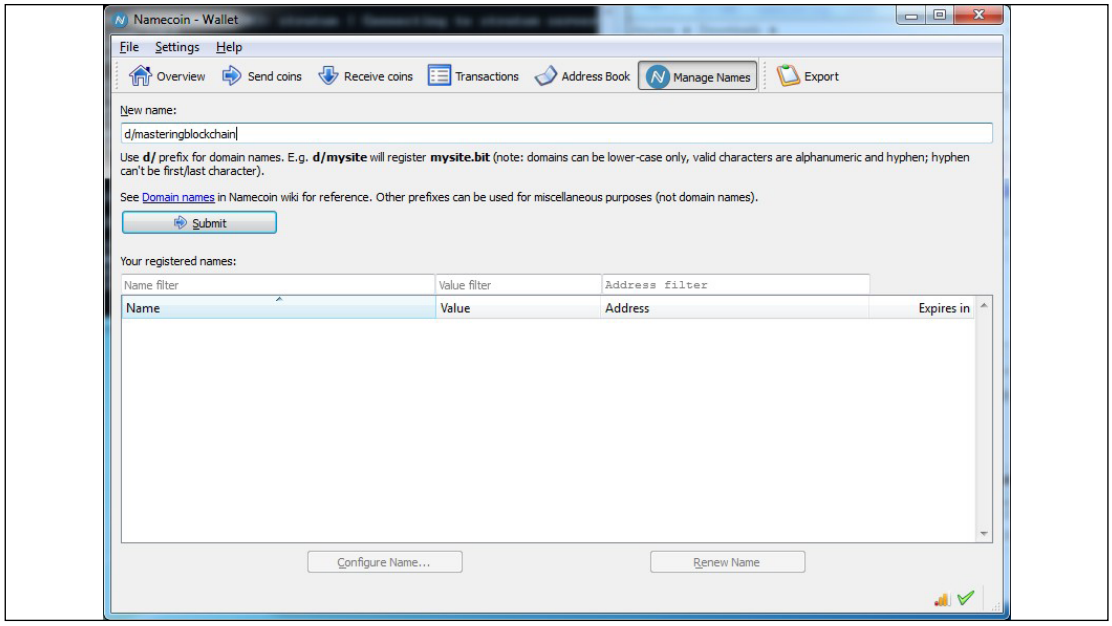
Generating Namecoin records

Namecoin records are in the form of key-value pairs. A name is a lowercase string of the form `d/examplename`, whereas a value is a case-sensitive, UTF-8 encoded JSON object that's a maximum of 520 bytes.

The name should be RFC1035 (`https://tools.ietf.org/html/rfc1035`) compliant.

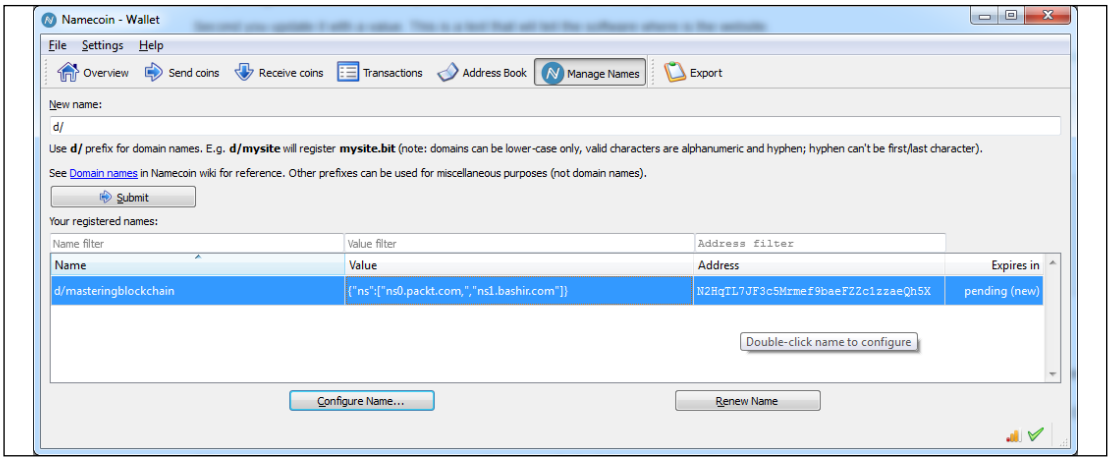
A general Namecoin name can be an arbitrary binary string up to 255 bytes long with 1,024 bits of associated identifying information. A record on a Namecoin chain is only valid for around 200 days or 36,000 blocks, after which it needs to be renewed. Namecoin also introduced `.bit` top-level domains, which can be registered using Namecoin and can be browsed using specialized Namecoin-enabled resolvers. Namecoin wallet software, as shown in the following screenshot, can be used to register `.bit` domain names:

1. The name is entered and, after the **Submit** button is pressed, it will ask for configuration information, such as DNS, IP, or identity:



Namecoin wallet: domain name configuration

2. As shown in the following screenshot, `masteringblockchain` will register as `masteringblockchain.bit` on the Namecoin blockchain:



Namecoin wallet: showing the registered name

Namecoin is the first fork of the Bitcoin source code. The second fork after Namecoin is Litecoin, another popular digital currency (altcoin) after Bitcoin, which the next section will cover in detail.

Litecoin

Litecoin is a fork of the Bitcoin source code and was released in 2011. It uses Scrypt as PoW, which was originally introduced in the Tenebrix coin (`https://bitcointalk.org/index.php?topic=45667.0`). Litecoin allows for faster transactions compared to Bitcoin due to its faster block generation time of 2.5 minutes. Also, difficulty readjustment is achieved every 3.5 days roughly due to faster block generation time. The total coin supply is 84 million.

The original announcement of Litecoin, along with an interesting discussion, is available at `https://bitcointalk.org/index.php?topic=47417.0`.

Scrypt is a sequentially memory hard function that is the first alternative to the SHA-256-based PoW algorithm. It was originally proposed as a **Password-Based Key Derivation Function** (**PBKDF**). The key idea is that if the function requires a significant amount of memory to run, then custom hardware such as ASICs will require more VLSI area, which would be infeasible to build. The Scrypt algorithm requires a large array of pseudorandom bits to be held in memory, and a key is derived from this in a pseudorandom fashion.

The algorithm is based on a phenomenon called **Time-Memory Trade-Off** (**TMTO**). If memory requirements are relaxed, then it results in increased computational cost. Put another way, TMTO shortens the running time of a program if more memory is given to it. This trade-off makes it unfeasible for an attacker to gain more memory because it is expensive and difficult to implement on custom hardware, or if the attacker chooses not to increase memory, then this results in the algorithm running slowly due to high processing requirements. This means that ASICs are difficult to build for this algorithm.

Scrypt uses the following parameters to generate a derived key (**Kd**):

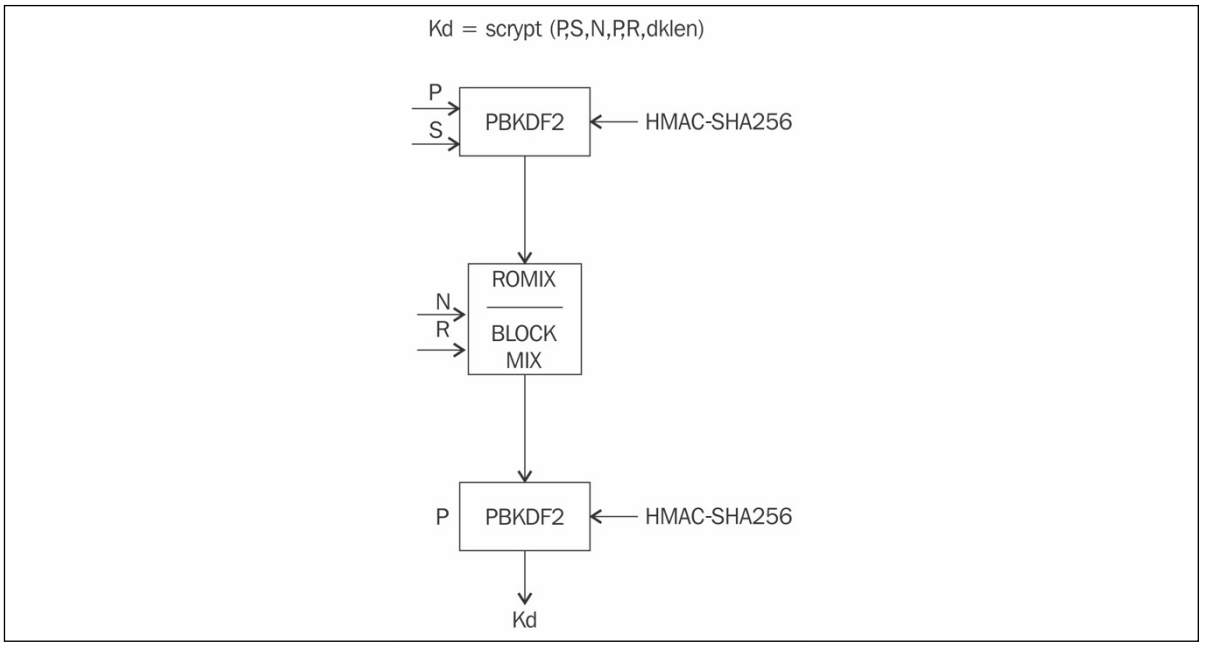
- **Passphrase**: This is a string of characters to hash.
- **Salt**: This is a random string that is provided to Scrypt functions (generally, all hash functions) in order to provide a defense against brute-force dictionary attacks using rainbow tables.
- **N**: This is a memory/CPU cost parameter that must be a power of 2 > 1.
- **P**: This is the parallelization parameter.
- **R**: This is the block size parameter.
- **dkLen**: This is the intended length of the derived key in bytes.

Formally, this function can be written as follows:

$$Kd = scrypt\ (P, S, N, P, R, dkLen)$$

Before applying the core Scrypt function, the algorithm takes *P* and *S* as input and applies PBKDF2 and SHA-256-based HMAC. Then, the output is fed to an algorithm called ROMix, which internally uses the Blockmix algorithm using the Salsa20/8 core stream cipher to fill up the memory, which requires a large memory to operate, thus enforcing the sequentially memory-hard property.

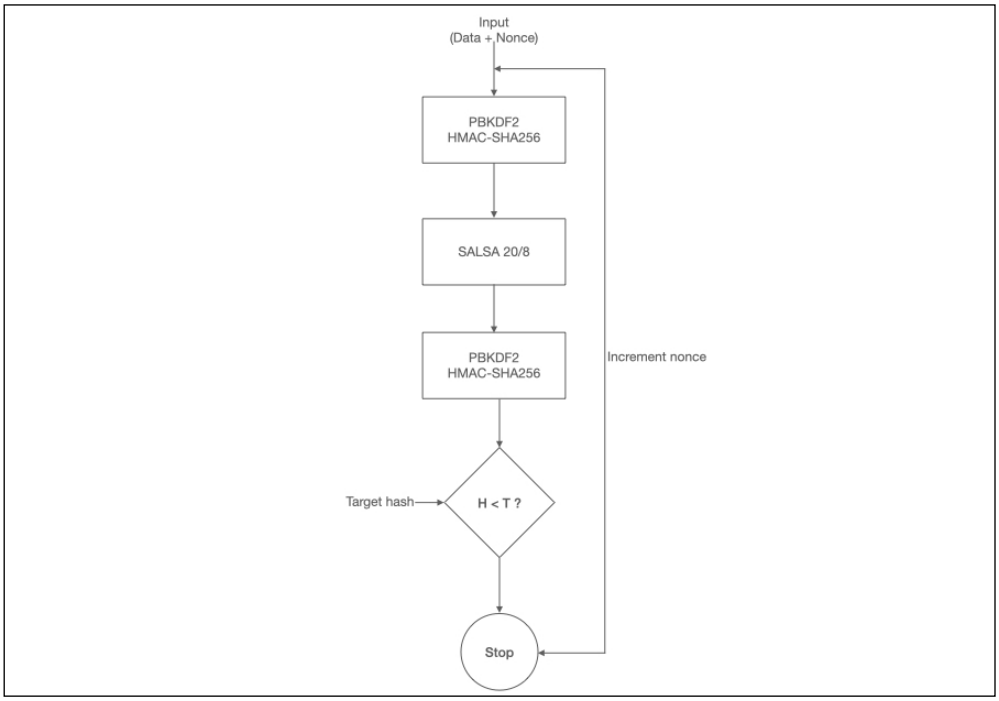
The output from this step of the algorithm is finally fed to the PBKDF2 function again in order to produce a derived key. This process is shown in the following diagram:



Scrypt algorithm

Scrypt is used in Litecoin mining with specific parameters where *N = 1024*, *R = 1*, *P = 1*, and *S = random 80 bytes*, thus producing a 256-bit output.

It appears that, due to the selection of these parameters, the development of ASICs for Scrypt for Litecoin mining turned out to be not very difficult. In ASIC for Litecoin mining, sequential logic can be developed that takes the data and nonce as input and applies the PBKDF2 algorithm with HMAC-SHA256. Then, the resultant bit stream is fed into the SALSA20/8 function, which produces a hash that, again, is fed down to the PBKDF2 and HMAC-256 functions to produce a 256-bit hash output. As is the case with Bitcoin PoW, in Scrypt, if the output hash is less than the target hash (already passed as input at the start, stored in memory, and checked with every iteration), then the function terminates; otherwise, the nonce is incremented and the process is repeated again until a hash is found that is lower than the difficulty target:



Scrypt ASIC design simplified flowchart

Trading Litecoin

As with other coins, trading Litecoin is easily carried out on various online exchanges. The current market cap of Litecoin is US$ 2,817,900,578. The current price (as of June, 2020) of Litecoin is US$ 43.36/LTC.

Litecoin mining on a CPU is no longer profitable, as is the case with many other digital currencies now. There are online cloud mining providers and ASIC miners available that can be used to mine Litecoin. Litecoin mining started from the CPU, progressed through GPU mining rigs, and now has reached a point where specialized ASIC miners, such as ASIC Scrypt Miner Wolf, are available from Ehsminer, which are now required to be used in the hope of being able to make some coins. Generally, it is true that even with ASICs, it is better to mine in pools instead of solo, as solo mining is not as profitable as mining in pools due to the proportional rewards scheme used by mining pools. Another miner is Antminer L3+, which is capable of producing 504MH/s.

Litecoin mining can be carried out solo or in pools. At the moment, ASICs for Scrypt are available that can be used to solo mine Litecoin. However, mining pools are also another option.

Software source code and wallet

The source code for Litecoin is available at `https://github.com/litecoin-project/litecoin`. The Litecoin wallet can be downloaded from `https://litecoin.org/` and can be used just like the Bitcoin core client software. A block explorer for Litecoin is available at `https://blockchair.com/litecoin`.

Litecoin and Bitcoin both use a PoW mechanism, whose sole purpose is to facilitate consensus via mining. It was realized quite early on that these protocols consume an extraordinary amount of electricity but only perform one function, which is not useful for anything else except mining. This is seen as a waste of energy and resources. The introduction of Primecoin was one of the first solutions to this problem, which the following section will discuss.

Primecoin

Primecoin is the first digital currency on the market that introduced a useful PoW, as opposed to Bitcoin's SHA256-based PoW. Primecoin uses searching prime numbers as a PoW. Not all types of prime number meet the requirements to be selected as PoW. Three types of prime numbers (known as the Cunningham chain of the first kind, the Cunningham chain of the second kind, and bi-twin chains) meet the requirements of a PoW algorithm to be used in cryptocurrencies. Primecoin is very interesting because it helps with prime number research. Several world records have been made so far. A list is available at `https://primes.zone`.

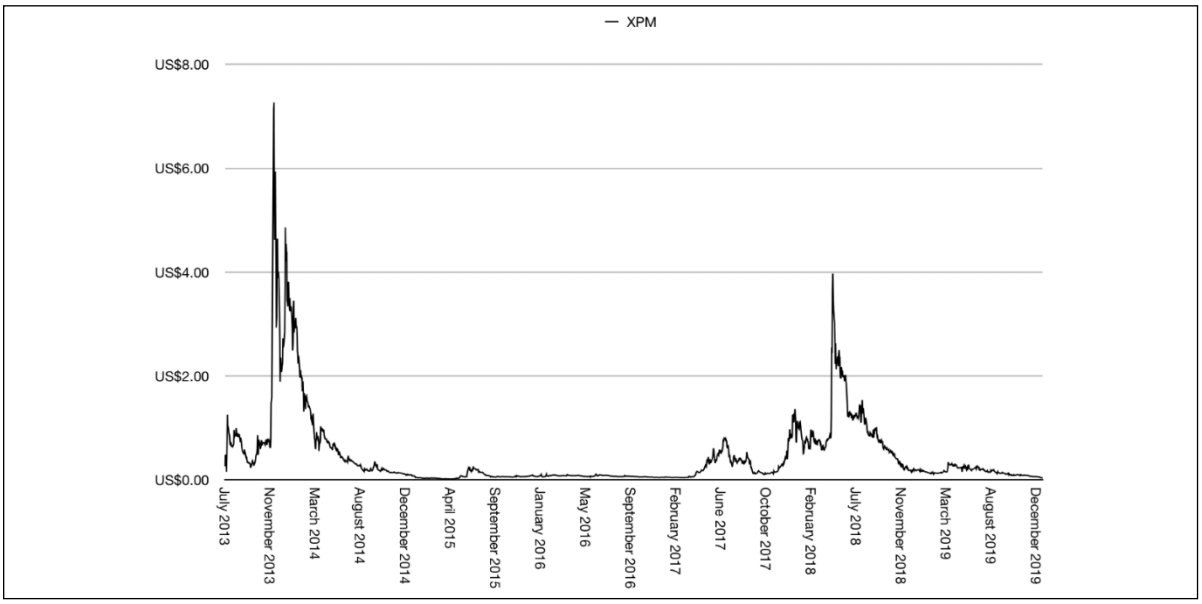
The difficulty is dynamically adjusted via a continuous difficulty evaluation scheme in the Primecoin blockchain. The efficient verification of PoW based on prime numbers is also of high importance because if verification is slow, then PoW is not suitable. Therefore, prime chains are selected as a PoW because finding prime chains gets difficult as the chain increases in length, whereas verification remains quick enough to warrant being used as an efficient PoW algorithm.

It is also important that once a PoW has been verified on a block, it must not be reused on another block. This is accomplished in Primecoin through a combination of PoW certificates and hashing it with the header of the parent block in the child block.

The PoW certificate is produced by linking the prime chain to the block header hash. It also requires that the block header's origin be divisible by the block header hash. If it is, it is divided and after division, the quotient is used as a PoW certificate. Another property of the adjustable difficulty of PoW algorithms is met by introducing difficulty adjustment every block instead of every 2,016, as is the case with Bitcoin. This is a smoother approach compared to Bitcoin and allows readjustment in the case of sudden increases in hash power. Also, the total number of coins generated is community-driven, and there is no definite limit on the number of coins Primecoin can generate.

Trading Primecoin

Primecoin can be traded on major virtual currency trading exchanges. The current market cap of Primecoin is US$ $1,099,646 at the time of writing (June, 2020). It is not very large but, because the currency is based on a novel idea and there is a dedicated community behind it, it continues to hold some market share:



Primecoin (XPM) price since 2013

Next, we'll explore a quick Primecoin mining guide.

Mining Primecoin

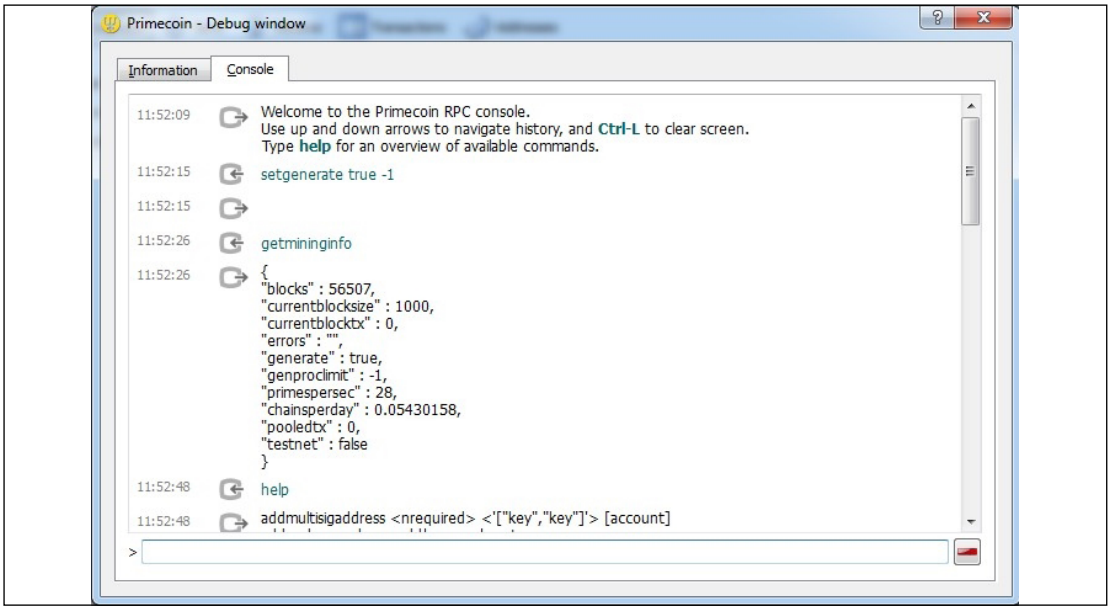
The first step is to download a wallet. Primecoin supports native mining within the wallet, just like original Bitcoin clients, but can also be mined on the cloud via various online cloud service providers.

A quick guide for Microsoft Windows is presented as follows. The Linux client is also available at `http://primecoin.io/downloads.php`:

1. The first step is to download the Primecoin wallet from `http://primecoin.io/index.php`.

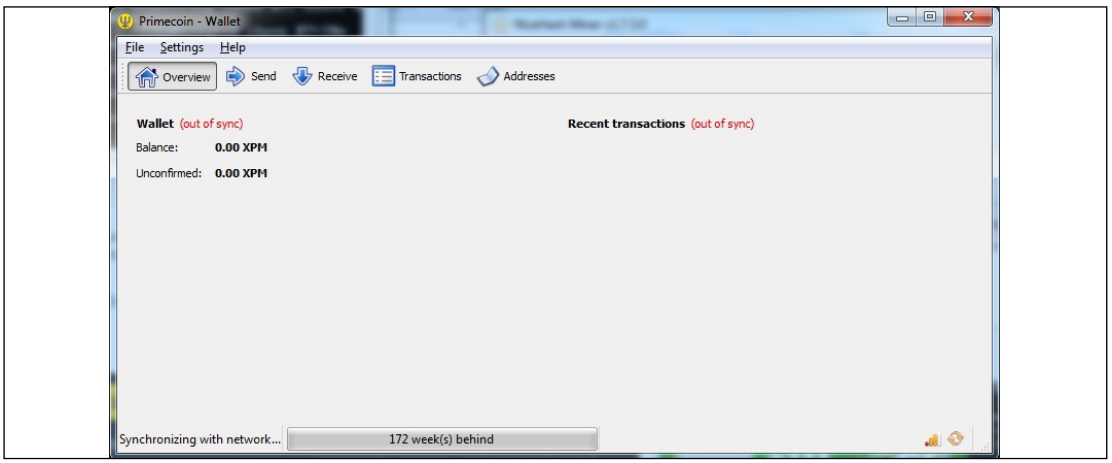
   Once the wallet has been installed and synched with the network, mining can be started.
2. A debug window can be opened in the Primecoin wallet by clicking on the **Help** menu and selecting the **Debug window** menu item.

Additional help can be invoked through typing `help` in the **Console** window of the **Debug window**, which is used to enable the Primecoin mining function:



Primecoin mining

3. Once the preceding commands are successfully executed, mining will start in solo mode. This may not be very fast and profitable if you have an entry-level PC with a slower CPU, but as this is a CPU-mined cryptocurrency, the miner can use PCs with powerful CPUs:



Primecoin wallet software syncing with the network

As an alternative, cloud services can be used, which host powerful server hardware. Primecoin is a novel concept and the PoW that it has introduced has great scientific significance. It is still in use with a market cap of US$ 17,034,198, but it appears that no active development is being carried out to further develop Primecoin, as is evident from its GitHub inactivity.

The Primecoin source code is available at `https://github.com/primecoin/primecoin`.

You can further explore Primecoin by reading the Primecoin whitepaper by Sunny King (a pseudonym) at `http://primecoin.io/bin/primecoin-paper.pdf`.

Having covered Primecoin, let's now move on and look at Zcash, a cryptocurrency that uses ZKPs to enhance its privacy features in a novel way.

Zcash

Zcash was launched on October 28, 2016. Zcash has used ZKPs in an innovative way, paving the way for future applications that require inherent privacy, such as banking, medicine, or the law. This is the first currency that uses a specific type of ZKP, known as **zero-knowledge Succinct Non-Interactive Arguments of Knowledge** (**zk-SNARK**), to provide complete privacy to the user. These proofs are concise and easy to verify; however, setting up the initial public parameters is a complicated process. The latter includes two keys: the proving key and the verifying key. The process requires sampling some random numbers to construct the public parameters. The issue is that these random numbers, also called toxic waste, must be destroyed after the parameter generation in order to prevent the counterfeiting of Zcash.

To address this issue, the Zcash team came up with a multi-party computation protocol to generate the required public parameters collaboratively from independent locations to ensure that toxic waste is not created. Because these public parameters are required to be created by the Zcash team, it means that the participants in the ceremony are trusted. This is the reason why the ceremony was very open and conducted by making use of a multi-party computation mechanism.

This mechanism has a property whereby all of the participants in the ceremony will have to be compromised to compromise the final parameters. When the ceremony is completed, all participants physically destroyed the equipment used for private key generation. This action eliminates any trace of the participants' part of the private key on the equipment.

Zk-SNARKS have the properties of **zero knowledge**, **succinctness**, **non-interactivity** and **arguments of knowledge**. We discussed these concepts in *Chapter 4*, *Public Key Cryptography*.

Zcash developers have introduced the concept of a **Decentralized Anonymous Payment scheme** (**DAP scheme**), which is used in the Zcash network to enable direct and private payments. The transactions reveal no information about the origin, destination, and amount of the payments. There are two types of addresses available in Zcash, Z address and T address. Z addresses are based on ZKPs and provide privacy protection, whereas T addresses are similar to those of Bitcoin.

A snapshot of various attributes of Zcash is shown as follows:

Attribute	**Description**
Name	Zcash
Currency code	ZEC
Launch date	28/10/2016
Main purpose	Cryptocurrency
Maximum coins	21 million
Block time	10 minutes
Consensus facilitation algorithm	Proof of Work–Equihash
Difficulty adjustment algorithm	DigiShield V3 (modified)
Difficulty adjustment interval	1 block
Mining hardware	CPU, GPU, ASIC

Zcash uses an efficient asymmetric PoW scheme called Equihash, which is based on the **Generalized Birthday Problem**. It allows for very efficient verification. It is a memory-hard and ASIC-resistant function.

The research article *Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem* is available at `http://ledger.pitt.edu/ojs/index.php/ledger/article/view/48/65`.

A novel idea, **initial slow mining**, was introduced with Zcash, which means that the block reward increases gradually over a period until it reaches the 20,000th block. This allows for initial scaling of the network and experimentation by early miners, and adjustment by Zcash developers if required. The slow start did have an impact on price due to scarcity as the price of ZEC on its first day of launch reached roughly USD 25,000.

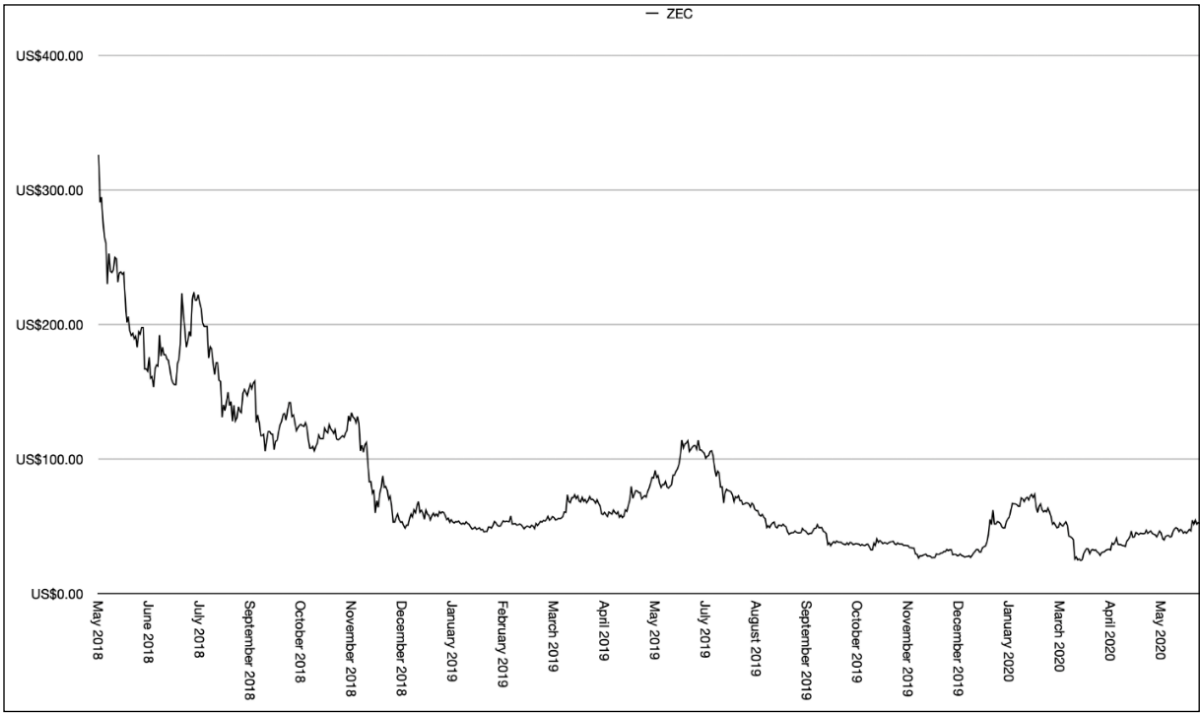
A slightly modified version of the DigiShield difficulty adjustment algorithm has been implemented in Zcash. The formula is shown as follows:

(Next difficulty) = (last difficulty) x *SQRT [(150 seconds) / (last solve time)]*

Trading Zcash

Zcash (**ZEC**) can be bought on many digital currency exchanges. A list is available here: `https://z.cash/exchanges/`.

When Zcash was introduced, its price was very high. As shown in the following graph, the price soared as high as approximately 10 bitcoins per ZEC. Some exchanges carried out orders as high as 2,500 BTC per ZEC. The price of ZEC is around US$ 51 at the time of writing (June, 2020):



Zcash price since 2018

In the following sections, we'll explore the process of mining Zcash. However, we'll preface this with an important consideration regarding the two possible address types that can be generated in Zcash.

Types of addresses

There are two types of addresses in Zcash. Addresses are either private, known as **t-addresses**, or transparent, known as **z-addresses**. T-addresses have a prefix letter of **t**, while z-addresses start with the letter **z**.

We can generate a t-addresses using the command shown here:

```
$ ./src/zcash-cli getnewaddress
```

This command will produce the output as shown, which shows the new address:

```
t1QBcRixAHLaSH5JHphBAX672eJoCTSLEw2
```

Similarly, a z-address can be generated using the following command:

```
$ ./src/zcash-cli z_getnewaddress
```

This command will produce the output as shown, which shows the new address:

```
zs1vtzhyawflspv6ej69rkjdvmun0y40lekkq8uzu5uc69f3djtpkyd4gfge38fuwkl8gc67yxed5l
```

Now, let's consider some ways that Zcash can be mined.

Mining Zcash

There are multiple methods to mine Zcash. Currently, only ASIC mining remains profitable, but it's worth noting that various commercial cloud mining pools offer Zcash mining contracts. However, CPU and GPU mining can still be performed for the purposes of experimentation. We'll consider these methods in the following sections.

Mining Zcash using a CPU

To perform solo mining using a **CPU**, the following steps can be followed on Ubuntu Linux. During this process, if you experience any errors, try to upgrade packages as shown here:

```
$ sudo apt-get update
```

After this, run:

```
$ sudo apt-get upgrade
```

The first `update` command will update the list of all available packages and their versions, whereas the second `upgrade` command will install the latest version of the packages. Both of these commands help to resolve some of the issues related to incompatibilities between different versions of the packages. With this in mind, follow these steps to establish a connection to `zcash-cli` using a CPU:

1. The first step is to install the prerequisites using the following command:

   ```
   $ sudo apt-get install \
     build-essential pkg-config libc6-dev m4 g++-multilib \
     autoconf libtool ncurses-dev unzip git python \
     zlib1g-dev wget bsdmainutils automake
   ```

   If the prerequisites are already installed, a message will be displayed, indicating that the components are already the newest versions. If they're not already installed or older than the latest package, then the installation will continue, the required packages will be downloaded, and the installation will be completed.

2. Next, run the commands to clone Zcash from Git. Note that if you are running `git` for the first time, you may have to accept a few configuration changes, which will automatically be done for you. The command should be entered as follows:

   ```
   $ git clone https://github.com/zcash/zcash.git
   ```

This will clone the Zcash Git repository locally. The output is shown in the following screenshot:

```
drequinox@drequinox-OP7010:~$ git clone https://github.com/zcash/zcash.git
Cloning into 'zcash'...
remote: Counting objects: 56593, done.
remote: Total 56593 (delta 0), reused 0 (delta 0), pack-reused 56593
Receiving objects: 100% (56593/56593), 42.78 MiB | 2.11 MiB/s, done.
Resolving deltas: 100% (43020/43020), done.
Checking connectivity... done.
drequinox@drequinox-OP7010:~$ cd zcash/
drequinox@drequinox-OP7010:~/zcash$ git checkout v1.0.0
Note: checking out 'v1.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 1feaefa... Update network magics for 1.0.0
```

Cloning the Zcash Git repository

3. The next step is to download the proving and verifying keys using the following command:

```
$ ./zcutil/fetch-param.sh
```

This command will produce an output similar to the one shown here:

```
drequinox@drequinox-OP7010:~/zcash$ ./zcutil/fetch-params.sh
Zcash - fetch-params.sh

This script will fetch the Zcash zkSNARK parameters and verify their
integrity with sha256sum.

The parameters are currently just under 911MB in size, so plan accordingly
for your bandwidth constraints. If the files are already present and
have the correct sha256sum, no networking is used.

Creating params directory. For details about this directory, see:
/home/drequinox/.zcash-params/README

Retrieving: https://z.cash/downloads/sprout-proving.key
--2016-10-28 21:46:21--  https://z.cash/downloads/sprout-proving.key
Resolving z.cash (z.cash)... 104.236.171.172
Connecting to z.cash (z.cash)|104.236.171.172|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://s3.amazonaws.com/zcashfinalmpc/sprout-proving.key [following]
--2016-10-28 21:46:22--  https://s3.amazonaws.com/zcashfinalmpc/sprout-proving.key
Resolving s3.amazonaws.com (s3.amazonaws.com)... 54.231.40.114
Connecting to s3.amazonaws.com (s3.amazonaws.com)|54.231.40.114|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 910173851 (868M) [application/octet-stream]
Saving to: '/home/drequinox/.zcash-params/sprout-proving.key.dl'

     0K ........ ........ ........ ........  3% 2.71M 5m8s
 32768K ........ ........ ........ ........  7% 3.58M 4m20s
 65536K ........ ........ ........ ........ 11% 2.53M 4m28s
 98304K ........ ........ ........ ........ 14% 1.75M 4m59s
131072K ........ ........ ........
```

Zcash setup fetching zk-SNARK parameters

4. When this command runs, it will download around 911 MBs of keys into the `~/.zcash-params/` directory. The directory contains files for proving and verifying keys:

```
$ pwd
/home/drequinox/.zcash-params
$ ls -ltr
sprout-verifying.key
sprout-proving.key
```

5. Once the preceding commands are completed successfully, the source code can be built using the following command:

```
$ ./zcutil/build.sh -j$(nproc)
```

   This will produce a very long output; if everything goes well, it will produce a `zcashd` binary file. Note that this command takes `nproc` as the parameter, which is essentially a command that finds the number of cores or processors in the system and displays that number. If you don't have that command, then replace `nproc` with the number of processors in your system.

6. Once the build is completed, the next step is to configure Zcash. This is achieved by creating a configuration file with the name `zcash.conf` in the `~/.zcash/` directory. A sample configuration file is shown as follows:

```
addnode=mainnet.z.cash
rpcuser=drequinox
rpcpassword=xxxxxxoJNo4o5c+F6E+J4P2C1D5izlzIKPZJhTzdW5A=
gen=1
genproclimit=8
equihashsolver=tromp
```

   The preceding configuration enables various features. The first line adds the mainnet node and enables mainnet connectivity. `rpcuser` and `rpcpassword` are the username and password for the RPC interface. `gen=1` is used to enable mining. `genproclimit` is the number of processors that can be used for mining. The last line enables a faster mining solver; this is not required if you want to use standard CPU mining.

7. Now, Zcash can be started using the following command:

```
$ ./zcashd --daemon
```

Once started, this will allow interaction with the RPC interface via the `zcash-cli` **command-line interface** (**CLI**). This is very similar to the Bitcoin CLI. Once the Zcash daemon is up and running, various commands can be run to query different attributes of Zcash. Transactions can be viewed locally using the CLI or a blockchain explorer.

A blockchain explorer for Zcash is available at `https://explorer.zcha.in/`.

Cloud mining contracts are also available from various online cloud mining providers. The cloud mining service providers perform mining on the customers' behalf. In addition to cloud mining contracts, miners can use their own equipment to mine via mining pools using stratum or other protocols.

One key example is the Zcash pool by NiceHash, which is available at `https://www.nicehash.com`. Using this pool, miners can sell their hash power.

An example of building and using a CPU miner on a Zcash mining pool is shown as follows.

Building and using a CPU miner on a mining pool

The idea here is to show how pool mining works. We'll be using `nheqminer`, a Linux miner for Zcash. `nheqminer` releases are available for Windows and Linux at the following link: `https://github.com/nicehash/nheqminer/releases`.

The following steps can be used to download and compile `nheqminer` on an Ubuntu Linux distribution:

```
$ sudo apt-get install cmake build-essential libboost-all-dev
$ git clone https://github.com/nicehash/nheqminer.git
$ cd nheqminer/nheqminer
$ mkdir build
$ cd build
$ cmake .. make
```

Once all the steps are completed successfully, `nhequminer` can be run using the following command:

```
$ ./nhequminer -l eu -u <btc address> -t <number of threads> -od 0
```

In the preceding command, `nheqminer` takes several parameters such as location (`-l`), username (`-u`), and the number of threads (which is usually equal to the number of processor cores available) to be used for mining (`-t`). (`-od`) indicates that AMD GPU device is enabled.

A sample run in Linux is shown as follows. In this screenshot, the payout is being made to a Bitcoin address for selling hash power:

```
drequinox@drequinox-OP7010:~/nheqminer/nheqminer/build$ ./nheqminer -l eu -u 1PL6gsm49xCFMvrXqgGcee5cdrG119GoWN.worker1 -t 6 -od 0
Equihash CPU Miner for NiceHash v0.1c
Thanks to Zcash developers for providing most of the code
Special thanks to tromp for providing optimized CPU equihash solver

Setting log level to 2
[09:28:53][0x00007f51009cd700] stratum | Connecting to stratum server equihash.eu.nicehash.com:3357
[09:28:53][0x00007f51009cd700] stratum | Connected!
[09:28:53][0x00007f51009cd700] stratum | Starting miner
[09:28:53][0x00007f50fafce700] miner#1 | Starting thread #1
[09:28:53][0x00007f50fb7cf700] miner#0 | Starting thread #0
[09:28:53][0x00007f50f8fca700] miner#5 | Starting thread #5
[09:28:53][0x00007f50fa7cd700] miner#2 | Starting thread #2
[09:28:53][0x00007f50f97cb700] miner#4 | Starting thread #4
[09:28:53][0x00007f50f9fcc700] miner#3 | Starting thread #3
[09:28:54][0x00007f51009cd700] stratum | Subscribed to stratum server
[09:28:54][0x00007f51009cd700] miner | Extranonce is 5000e5b800000000000000005000e5b9ab
[09:28:54][0x00007f51009cd700] stratum | Authorized worker 1PL6gsm49xCFMvrXqgGcee5cdrG119GoWN.worker1
[09:28:54][0x00007f51009cd700] stratum | Target set to 01e1e1e1e0000000000000000000000000000000000000000000000000000000
[09:28:54][0x00007f51009cd700] stratum | Received new job #000000329b82d287
[09:28:55][0x00007f50fa7cd700] stratum | Submitting share #4, nonce 02000000000000000000000000000000
[09:28:55][0x00007f51009cd700] stratum | Accepted share #4
[09:28:55][0x00007f51009cd700] stratum | Ignoring non-clean job #000000329b82d2cc
[09:28:57][0x00007f50fafce700] stratum | Submitting share #5, nonce 01000000000000000000000000000001
[09:28:57][0x00007f51009cd700] stratum | Accepted share #5
[09:28:59][0x00007f50f97cb700] stratum | Submitting share #6, nonce 04000000000000000000000000000005
```

Using the BTC address to receive payouts for selling hash power

The next screenshot shows a sample run of `nheqminer` in Windows, with payouts being made to a **Zcash t-address** for selling hash power:

```
C:\nheqminer_v0.3a>nheqminer_zcash.exe -l eu -u t1YiNeyxoLZcjDHnaMtq5WLEbKquTY11gK8.miner1 -t 6 -od 0

        ========== www.nicehash.com ==========
              Equihash CPU&GPU Miner for NiceHash v0.3a
        Thanks to Zcash developers for providing base of the code.
            Special thanks to tromp and xenoncat for providing
                optimized CPU and CUDA equihash solvers.
        ========== www.nicehash.com ==========

Setting log level to 2
[2016-10-29 22:37:18.196024][0x00001404]: Using SSE2: YES
[2016-10-29 22:37:18.203024][0x00001404]: Using AVX: YES
[2016-10-29 22:37:18.207025][0x00001404]: Using AVX2: YES
[2016-10-29 22:37:18.222026][0x00001590]: stratum | Starting miner
[2016-10-29 22:37:18.228026][0x000015c4]: miner#0 | Starting thread #0 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:18.228026][0x00000524]: miner#1 | Starting thread #1 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:18.243027][0x00000b20]: miner#3 | Starting thread #3 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:18.243027][0x00001570]: miner#2 | Starting thread #2 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:20.354147][0x00001330]: miner#5 | Starting thread #5 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:20.354147][0x000015f8]: miner#4 | Starting thread #4 (CPU-XENONCAT-AVX2)
[2016-10-29 22:37:20.354147][0x00001590]: stratum | Connecting to stratum server stratum.eu.zcash.nicehash.com:3357
[2016-10-29 22:37:20.354147][0x00001108]: miner#6 | Starting thread #6 (OCL_XMP) TODO
<info> found 1 devices
Using device 0 as GPU 0
<info> compiling ...
<info> source: 41162 bytes
[2016-10-29 22:37:28.796630][0x00001590]: stratum | Connected!
[2016-10-29 22:37:28.892636][0x00001590]: stratum | Subscribed to stratum server
[2016-10-29 22:37:28.899636][0x00001590]: miner | Extranonce is 1ffff7da000000000000000001ffff7d9
[2016-10-29 22:37:28.999642][0x00001590]: stratum | ←[35mTarget set to 03c3c3c000000000000000000000000000000000000000000000000000000000←[0m
[2016-10-29 22:37:29.016643][0x00001590]: stratum | ←[36mReceived new job #6←[0m
[2016-10-29 22:37:29.290659][0x00001590]: stratum | Authorized worker t1YiNeyxoLZcjDHnaMtq5WLEbKquTY11gK8.miner1
[2016-10-29 22:37:33.311889][0x00001404]: ←[33mSpeed [300 sec]: 0.154357 I/s, 0.231535 Sols/s←[0m
[2016-10-29 22:37:40.476298][0x00000524]: stratum | Submitting share #4, nonce 0100000000000000000000000000004
[2016-10-29 22:37:40.605306][0x00001590]: stratum | ←[32mAccepted share #4←[0m
[2016-10-29 22:37:43.411466][0x00001404]: ←[33mSpeed [300 sec]: 0.824045 I/s, 1.43124 Sols/s←[0m
```

Using Zcash t-address to receive payouts for selling hash power

Another excellent miner software for pool mining is called **bminer**. It can be downloaded from `https://www.bminer.me/releases/`.

As discussing all the different algorithms that **bminer** supports is not possible here for reasons of brevity, we encourage you to refer to following link for further information: `https://www.bminer.me/examples/`.

Mining Zcash using a GPU

Other than CPU mining, **GPU** mining options for Zcash are also available, which, along with ASICs, are still popular tools for mining Zcash. There is no official GPU miner yet; however, open source developers have produced various proofs of concepts and working miners. The Zcash company held an open competition to encourage developers to build and submit CPU and GPU miners, but no winning entry has been announced at the time of writing.

You can get more information on this by visiting the following website: `https://zcashminers.org/`.

Mining Zcash using ASICs

As expected, Zcash **ASIC** miners are now available. CPU mining is not profitable anymore and some GPUs are somewhat profitable, like GTX 1080 and 1070, but very soon, they will also be left behind. ASIC miners are now the tool of choice for mining Zcash.

ASICs such as Bitmain, Antminer Z9, and Antminer Z11 are available and provide a high rate of solving **Proof of Work** (**PoW**).

Running Zcash on Mac

In this section, we will describe the process of downloading, compiling, and running Zcash on macOS. This is quite useful if you want to run a node on a Mac, run a local wallet, or just experiment with the testnet.

A list of Zcash wallets is available at `https://z.cash/wallets/`. We will not use any wallets in this example; instead, we will use the native Zcash daemon for our experiment.

We need some prerequisites to be installed before we can download and compile Zcash. These prerequisites are listed here:

1. Homebrew: if you do not have it already installed on your Mac, we can install `Homebrew` using the following command:

```
$ /usr/bin/ruby -e "$(curl -fsSL \ https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

This command will show an output similar to the one shown here:

```
==> This script will install:
/usr/local/bin/brew
/usr/local/share/doc/homebrew
/usr/local/share/man/man1/brew.1
/usr/local/share/zsh/site-functions/_brew
/usr/local/etc/bash_completion.d/brew
/usr/local/Homebrew
==> The following new directories will be created:
/usr/local/sbin
/usr/local/Frameworks
==> The Xcode Command Line Tools will be installed.

Press RETURN to continue or any other key to abort
==> /usr/bin/sudo /bin/mkdir -p /usr/local/sbin /usr/local/Frameworks
Password:
```

Homebrew installation

Note that the complete outputs of these commands are not shown, for brevity.

This command will also download and install command-line tools for Xcode.

2. Once `homebrew` is installed, we can move on to the next step, which is to install various required packages using brew:

```
$ brew install git pkgconfig automake autoconf wget libtool coreutils
```

This command will install the required packages for Zcash installation, and show an output similar to the one shown here:

```
Warning: pkg-config 0.29.2 is already installed and up-to-date
To reinstall 0.29.2, run `brew reinstall pkg-config`
==> Installing dependencies for git: pcre2
==> Installing git dependency: pcre2
==> Downloading https://homebrew.bintray.com/bottles/pcre2-10.34.catalina.bottle.tar.gz
==> Downloading from https://akamai.bintray.com/af/af3bf030a455daf0a560f8d9e433f7a803d71
=b57f5fe13e84b7791535eaa96d
######################################################################## 100.0%
==> Pouring pcre2-10.34.catalina.bottle.tar.gz
🍺  /usr/local/Cellar/pcre2/10.34: 230 files, 5.9MB
==> Installing git
==> Downloading https://homebrew.bintray.com/bottles/git-2.24.1.catalina.bottle.tar.gz
==> Downloading from https://akamai.bintray.com/27/2779f9cea861ef4d906093ac7b7255b3eeb7b
=a1cde25ccd29678272e1db57d7
######################################################################## 100.0%
==> Pouring git-2.24.1.catalina.bottle.tar.gz
```

Brew package installation

3. Once finished successfully, the next step is to install `pip` (a Python package manager):

```
$ sudo easy_install pip
```

This will show the following output, indicating successful installation of `pip`:

```
Password:
Searching for pip
Reading https://pypi.org/simple/pip/
Downloading https://files.pythonhosted.org/packages/00/b6/9cfa56b4081ad13874b0c6
any.whl#sha256=6917c65fc3769ecdc61405d3dfd97afdedd75808d200b2838d7d961cebc0c2c7
Best match: pip 19.3.1
Processing pip-19.3.1-py2.py3-none-any.whl
Installing pip-19.3.1-py2.py3-none-any.whl to /Library/Python/2.7/site-packages
Adding pip 19.3.1 to easy-install.pth file
Installing pip script to /usr/local/bin
Installing pip3.7 script to /usr/local/bin
Installing pip3 script to /usr/local/bin
```

PIP installation

4. After this, we have to install the relevant Python packages for RPC tests:

```
$ sudo pip install pyblake2 pyzmq
```

This will show the following output, indicating successful installation of the necessary packages:

```
Collecting pyblake2
  Downloading https://files.pythonhosted.org/packages/a6/ea/
(126kB)
     |████████████████████████████████| 133kB 1.4MB/s
Collecting pyzmq
  Downloading https://files.pythonhosted.org/packages/73/ef/
m-macosx_10_6_intel.whl (1.4MB)
     |████████████████████████████████| 1.4MB 314kB/s
Building wheels for collected packages: pyblake2
  Building wheel for pyblake2 (setup.py) ... done
  Created wheel for pyblake2: filename=pyblake2-1.1.2-cp27-c
9701aee3e5e7b63a7ec9f885ff6b
  Stored in directory: /Users/drequinox/Library/Caches/pip/w
Successfully built pyblake2
Installing collected packages: pyblake2, pyzmq
Successfully installed pyblake2-1.1.2 pyzmq-18.1.1
```

Python packages installation

5. Now, we need to clone the Zcash source code from GitHub:

```
$ git clone https://github.com/zcash/zcash.git
```

This will show the following output:

```
Cloning into 'zcash'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 77342 (delta 5), reused 5 (delta 5), pack-reused 77334
Receiving objects: 100% (77342/77342), 67.52 MiB | 1.64 MiB/s, done.
Resolving deltas: 100% (58355/58355), done.
```

Zcash download from GitHub

6. Change the directory to `zcash`:

```
$ cd zcash
```

7. Check out the `git` branch:

```
$ git checkout v2.1.0-1
Note: checking out 'v2.1.0-1'.
.
.
HEAD is now at 253fcaa99 Merge pull request #4213 from str4d/
release-v2.1.0-1
```

If you see a message stating `detached HEAD`, you can safely ignore it.

8. Now, we can download the Zcash zk-SNARK parameters:

```
$ ./zcutil/fetch-params.sh
```

9. Finally, we can build:

```
$./zcutil/build.sh -j$(nproc)
```

Note that this step can take around 30-45 minutes to complete.

Once built successfully, the binaries will be located in the `src` folder.

10. Now, we can run the `zcash` daemon simply by running the following command:

```
$ ./src/zcashd
```

The command will produce the output shown below, indicating that Zcash server is starting:

```
Zcash server starting
```

Similarly to the `bitcoin-cli`, we can run several commands. An example of `getinfo` is shown here:

```
➜  zcash git:(253fcaa99) ./src/zcash-cli getinfo
{
  "version": 2010051,
  "protocolversion": 170009,
  "walletversion": 60000,
  "balance": 0.00000000,
  "blocks": 21535,
  "timeoffset": 0,
  "connections": 1,
  "proxy": "",
  "difficulty": 1081700.94708172,
  "testnet": false,
  "keypoololdest": 1510092406,
  "keypoolsize": 101,
  "paytxfee": 0.00000000,
  "relayfee": 0.00000100,
  "errors": ""
}
```

Zcash getinfo example

The output provides information on a number of parameters related to the `zcash-cli`.

Now, we can stop the `zcash` daemon using the following command:

```
$ ./src/zcash-cli stop
```

The command will produce the output shown below, indicating that Zcash server is stopping:

```
Zcash server stopping
```

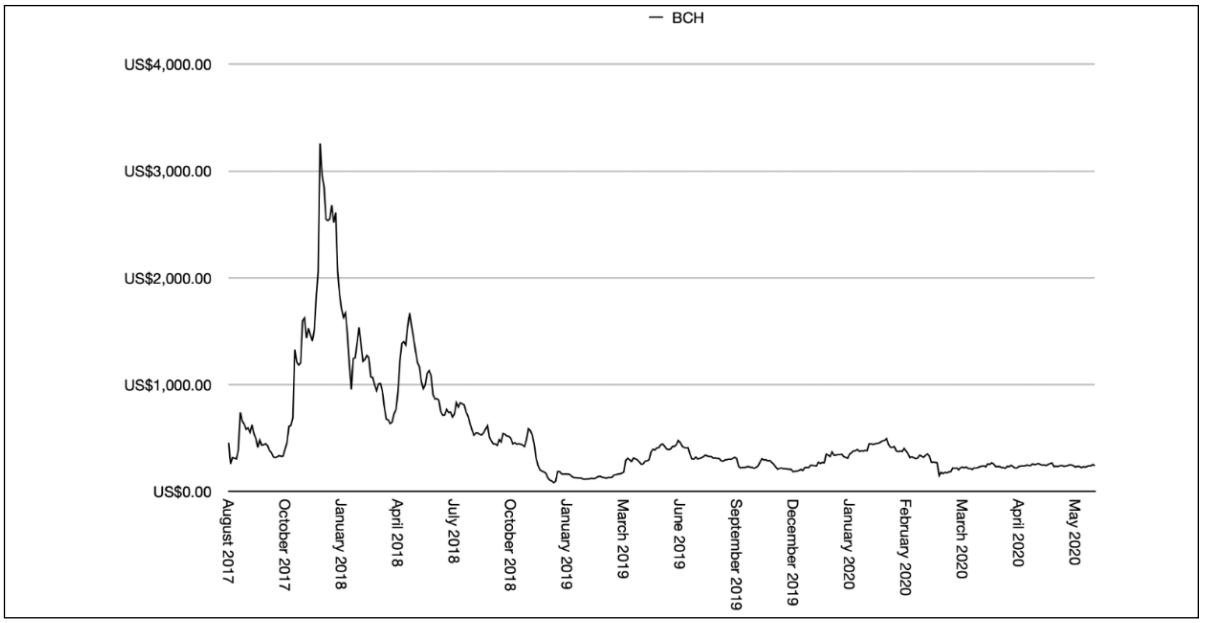
This section completes the introduction to Zcash. You can explore Zcash further online at `https://z.cash`.

Next, we will introduce Bitcoin Cash, a fork of Bitcoin that features a number of improvements over the original Bitcoin protocol.

Bitcoin Cash

Bitcoin Cash (**BCH**) was created on August 1, 2017 as a result of a hard fork of Bitcoin. The most significant change from Bitcoin is the size of the block, which was increased to 8 MB. Similar to Bitcoin, the total supply of BCH is limited to 21 million BCH.

On November 16, 2018, Bitcoin Cash was hard forked to create Bitcoin SV and Bitcoin ABC. Bitcoin ABC became the longest and dominant chain due to possessing more hash power and the majority of the nodes. Ultimately, exchanges renamed the Bitcoin ABC chain to Bitcoin Cash, and the Bitcoin Cash SV chain became what is known today as **Bitcoin SV** (**BSV**):



BCH price

In the preceding chart, the BCH price is shown from 2017. The price was at its highest around early 2018 and later dropped significantly. Currently, the price of BCH is US$ 233. Further details about the protocol and pricing can be found on the official website: `https://www.bitcoincash.org`.

There are a number of wallets available for different platforms for BCH. The wallet can be downloaded from `https://www.bitcoincash.org/wallets.html`.

A blockchain explorer for Bitcoin cash is available here: `https://explorer.bitcoin.com/bch`.

Further innovation and development did not stop with BCH. Even with the development of BCH, the need was felt to further improve and scale Bitcoin and as a result, Bitcoin Cash was forked into Bitcoin SV. We will discuss Bitcoin SV next.

Bitcoin SV

Bitcoin Satoshi's Vision (**BSV**) was created on November 15, 2018 by hard forking Bitcoin Cash (BCH). It was created after a need was felt to increase the performance, security, and scalability of the Bitcoin system by increasing the block size.

There are four fundamental areas that have been reported to be addressed by the Bitcoin SV team to shape a blockchain system that is suitable for mainstream usage. These four points are **stability**, **scalability**, **security**, and **safe instant transactions**:

- **Stability** is envisioned to be provided by restoring the Bitcoin protocol to its original design. A further innovation is expected to occur after a stable base protocol has been achieved.
- **Scalability** is expected to be provided by increasing the capacity of the blocks and relevant performance enhancements.
- **Security**, an important aspect, is expected to be achieved by introducing better change management practices, industry-standard security audits, and by offering a bug bounty program. With this bug bounty program, it is expected that researchers around the world will participate in finding and reporting security vulnerabilities.
- **Safe instant transactions** are noted as a key priority for BSV. Instant transactions or `0-conf` transactions are expected to allow faster payments for businesses. `0-conf` are the transactions that have been broadcast but are waiting to be included in a block. These transactions are seen as a method to enable faster payments in Bitcoin. This is possible because transaction propagation itself is a fast process, but including it in a block, mining it, and then gaining confirmations takes a long time. If, somehow, before including them in a block, these transactions can be considered final with appropriate security guarantees, then faster payments can be achieved. However, the biggest challenge in this case is double spending attacks, which is addressed in Bitcoin by the mining process. However, in BSV, `0-conf` transactions are believed to be safe because first, the transaction propagation is fast and miners will not accept a transaction that they see as a double spend attempt. Secondly, all attempts of double detection will be detectable due to fast propagation of the transactions on the network.

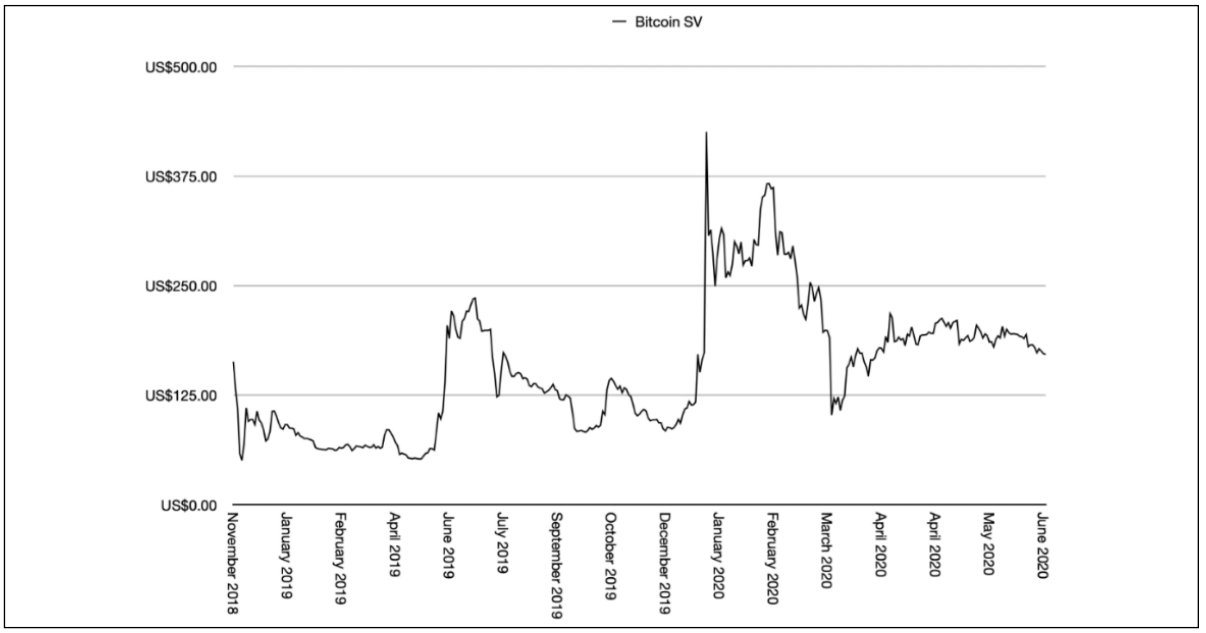
`0-conf`, generally, and on BSV, is debatable as there has been some evidence of double spending `0-conf` transactions. More information on this is available at the following link:

`https://cointelegraph.com/news/video-demonstrates-double-spending-possibility-in-bitcoin-cash-sv-0-conf-transactions`

Also, note that like any technical argument, there are points in favour of `0-conf` and points against it. You are advised to explore further and make a sound judgement based on the facts.

A number of wallets are available that supports BSV. A list is available here: `https://bitcoinsv.io/services/wallets-and-exchanges`.

A block explorer for BSV is available at `https://blockchair.com/bitcoin-sv`:



Price of BSV since November 2018

The preceding chart shows the price of BSV over 2018. The current market cap (as of June 2020) of BSV is USD 3,220,614,914, which makes it one of the top 10 cryptocurrencies by market cap.

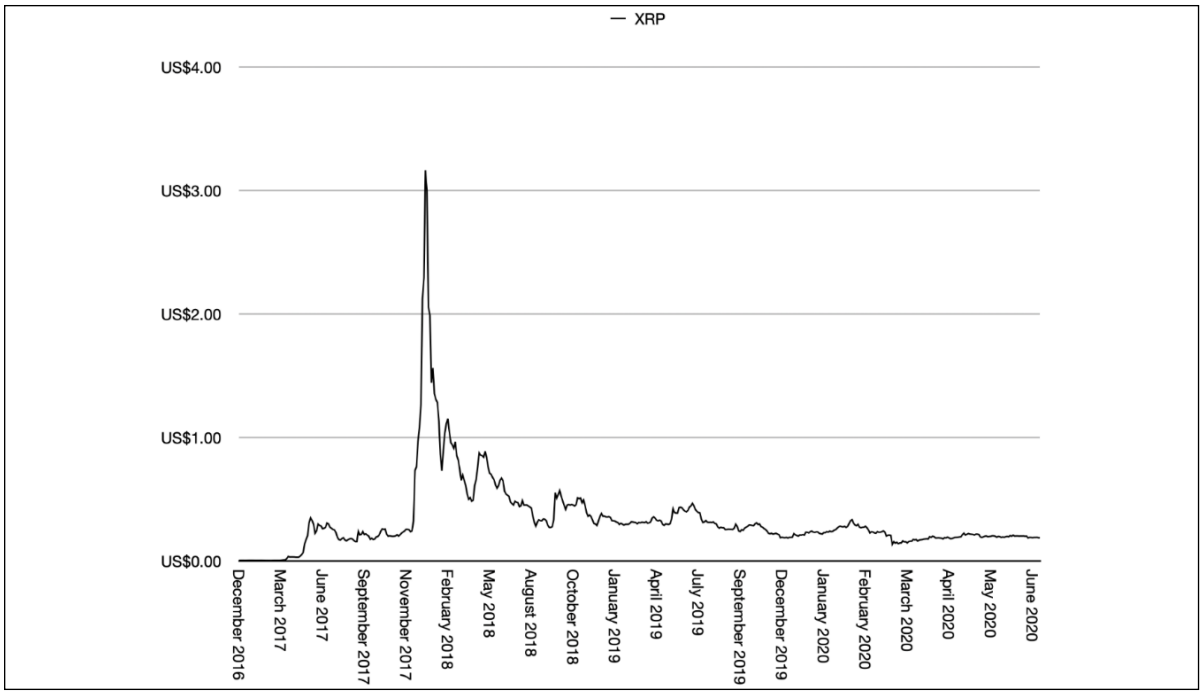
Ripple XRP

Ripple was developed by RippleLabs Inc. in 2012. XRP is the currency or token that is used on the Ripple network for payment processing. The **Ripple network** (**RippleNet**) facilitates money transfer from one entity to another by converting the money to be transferred into XRP, and then using XRPs to represent that money on the network. Once converted into XRP, the XRPs representing the money is transferred to the recipient, where it is concerted back to fiat money from XRP. XRP has a fixed supply of 100 billion units.

XRP's official site contains a wealth of information about the protocol. It can be accessed at `https://ripple.com/xrp/`.

RippleNet makes use of a Byzantine agreement algorithm for ensuring the safety and liveness of the distributed ledger. A paper on this topic is available here: `https://arxiv.org/abs/1802.07242`

Ripple software, which is predominantly written in C++, is available at `https://github.com/ripple/rippled`:



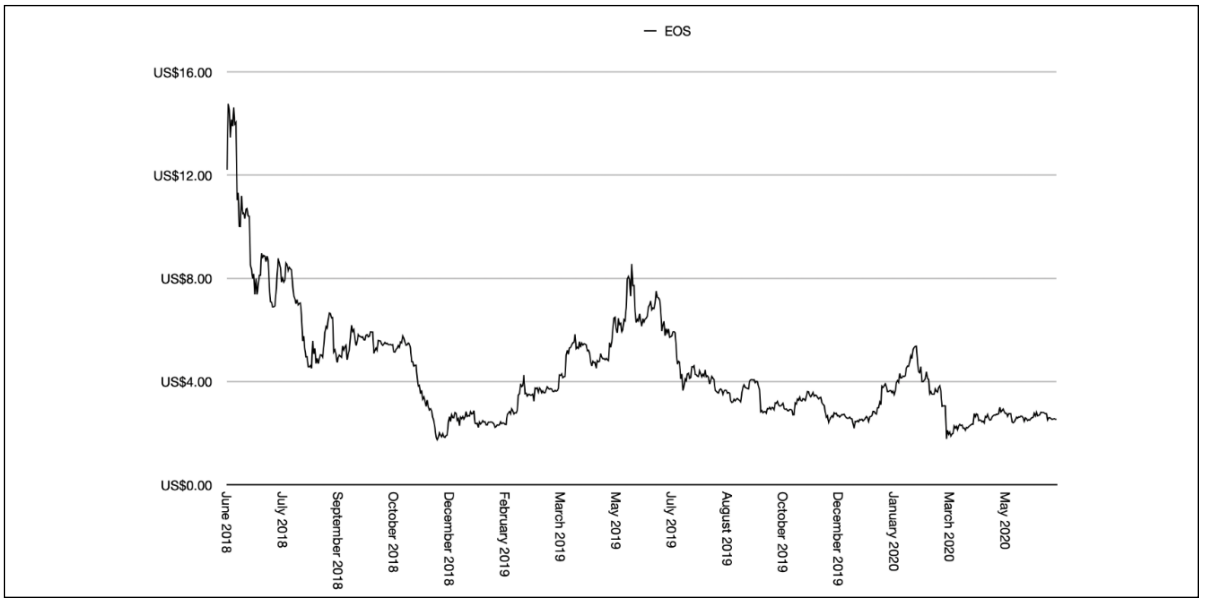
Price of XRP since 2016

The preceding chart shows the price of XRP since 2016. A sharp increase in the price is seen around early 2018. The current market cap (as of June 2020) of XRP is USD 8,443,721,935.

EOS

EOS is the native digital currency of the EOS.IO blockchain. EOSIO is based on the principles of real computer hardware architecture, including CPU, RAM, and storage. EOSIO is a decentralized operating system that supports smart contracts.

An interesting feature claimed by the EOSIO blockchain is that it does not charge any transaction fees and can scale up to millions of transactions per second:



Price of EOS since 2018

As shown in the preceding price history graph, the price of EOS has decreased significantly compared to the time when it was initially launched in 2018. The current price of EOS (as of June, 2020) is USD 2.53.

EOS was first released on January 31, 2018. The blockchain makes use of the delegated proof of stake algorithm to achieve consensus. EOS is developed by a company called block.one (`https://block.one`). It is implemented using C++.

The EOS blockchain's official website is `https://eos.io`. The EOS blockchain explorer is available at `https://bloks.io`. An interesting historic reference and the original announcement is available at `https://Bitcointalk.org/index.php?topic=1904415.0`.

The technical white paper is available here: `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`

EOS is traded at a number of exchanges, including Binance (`https://www.binance.com/en/trade/EOS_USDT`) , Changelly (`https://changelly.com`), and Coinbase (`https://www.coinbase.com`).

Tezos

Tezos was launched in September 2018. Tezos (XTZ), also called **tezzies** or **Tezos coin**, is the currency of the Tezos blockchain. XTZ is the native currency of the Tezos blockchain, which is created by smart contracts on the Tezos blockchain:

Tezos price chart since 2018

As shown in the preceding pricing chart, XTZ saw a major decrease in price in early 2019, but has somewhat recovered and currently (as of June, 2020), the price of XTZ is around US$ 2.66.

As a blockchain, it has introduced a number of interesting features such as self-amending and upgrading capability, liquid Proof of Stake mining (called baking in the Tezos blockchain), and increased smart contract security using formal verification to prove the correctness of the smart contract program. Moreover, it is implemented in a language called OCaml, which is quite different from other blockchains, which are usually developed in C++, Rust, or Golang.

OCaml is a general-purpose programming language for software development that focuses on safety and expressiveness. More details can be found here: `https://ocaml.org`.

As Tezos and EOS run on their respective native blockchains, features related to the underlying blockchain technology of these coins can be explored in more detail in the *Alternative Blockchains* section of this bonus content pack.

Chapter 11

Ethereum networks

When interacting with Ethereum, it's important to consider which network you'd like to connect to, and for what purpose. Some of the different Ethereum networks and their properties are presented in the following table, as a resource for further research:

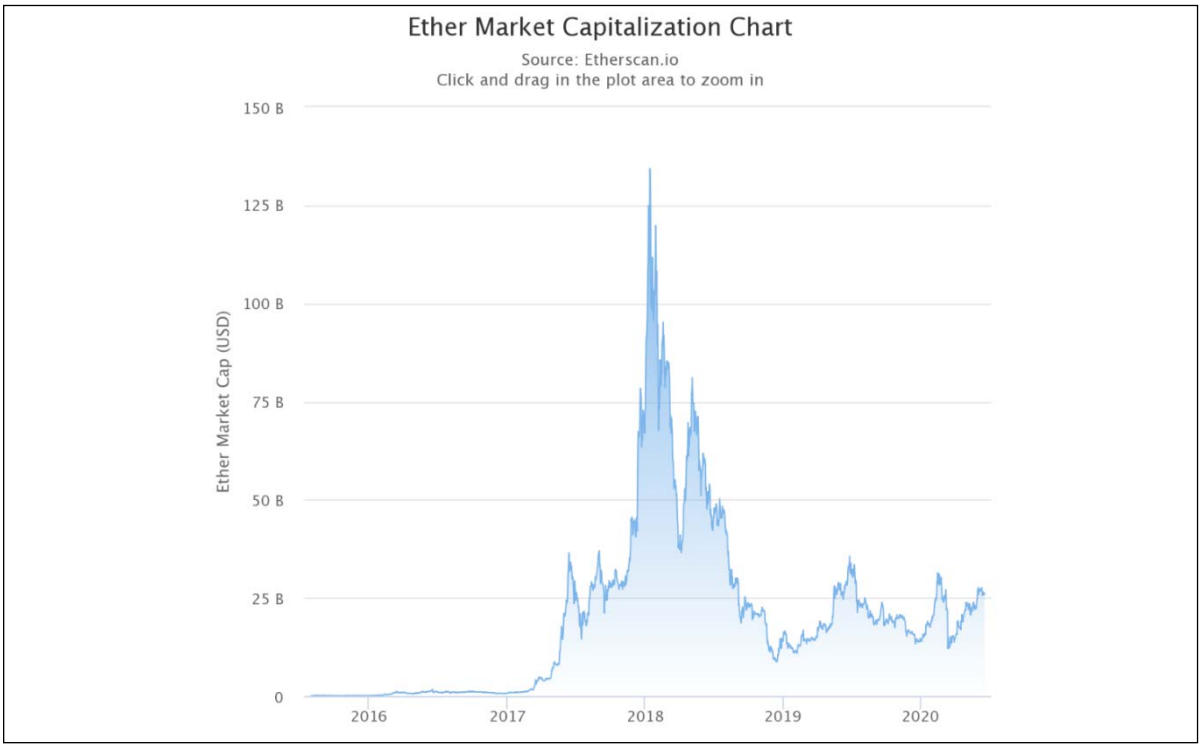
Network name	**Description**	**Network ID**	**Chain ID**	**Consensus**
Mainnet	The Ethereum main network `https://ethereum.org`	1	1	**Proof of Work** (**PoW**)
Ropsten	An Ethereum test network `https://ropsten.etherscan.io`	3	3	PoW
Rinkeby	An Ethereum test network `https://www.rinkeby.io/`	4	4	**Proof of Authority** (**PoA**)
Goerli	A cross-client Ethereum test network `https://goerli.net`	5	5	PoA
Kovan	An Ethereum test network `https://kovan.etherscan.io`	42	42	PoA
Ethereum Classic mainnet	The Ethereum classic main network `https://ethereumclassic.org`	1	61	PoW
Mordor	The Ethereum classic test network (replaced the Morden test network) `http://mordor.etccoopexplorer.com`	7	63	PoW
Private	Geth private chains	Default: 1337 User-configurable	1337	PoA and use case dependent

A major motivation for users connecting to the Ethereum mainnet is to obtain **ether** (**ETH**). To provide some economic context for this, we will move on from our technical analysis of Ethereum's network types, and briefly consider the logistics of buying and selling ETH.

Trading and investment

Ether can be mined, or alternatively purchased on various online exchanges for the purpose of trading. It is also available at various exchanges for buying and selling. The current market cap of Ethereum is USD 25,745,492,321 at the time of writing (June, 2020), and an individual ETH is worth approximately USD 231.

The following chart shows the historical market capitalization, giving an idea of the volatility of ether investment:



Ether historical market capitalization (source Etherscan.io)

There are online services available, such as `https://shapeshift.io`, which allow conversion from one currency to another.

Various online exchanges, such as Kraken, Coinbase, and many more, offer purchases of ETH with fiat currency using credit cards or another virtual currency, such as bitcoins.

Chapter 12

Opcodes and their meaning

These tables show various mnemonics, and some associated description. This includes the hex value of the mnemonic, the number of items that will be removed from the stack when this mnemonic executes (`POP`), the number of items that are added to the stack (`PUSH`) when this mnemonic executes, the gas cost associated with the mnemonic and purpose of the mnemonic.

Arithmetic operations

All arithmetic in the **Ethereum Virtual Machine** (**EVM**) is modulo 2256. This group of opcodes is used to perform basic arithmetical 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	Modulo multiplication operation.
`EXP`	`0x0a`	2	1	10	Exponential operation (repeated multiplication of the base).
`SIGNEXTEND`	`0x0b`	2	1	5	Extends the length of two's complement signed integer. **Two's complement** is a method used to represent signed integers.

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

Logical operations

Logical operations include operations that are used to perform comparisons and bitwise 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) operator.
NOT	0x19	1	1	3	Bitwise NOT operator.
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 SHA-3 standardized by NIST, but the original Keccak implementation.

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

Note that 30 is the cost of the operation. Then 6 gas is paid for each word. Therefore, the formula for SHA3 gas cost becomes *30 + 6 * (size of input in words)*.

Environmental information

There is 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	700	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 the 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 memory.
RETURNDATASIZE	0x3d	0	1	2	Size of data returned from the previous call.
RETURNDATACOPY	0x3e	3	0	3	Copy data returned from the previous call to memory.
EXTCODEHASH	0x3f	1	1	400	Returns the keccak256 hash of a contract's code.
CHAINID	0x46	0	1	2	Returns the ID of the executing/config chain.
SELFBALANCE	0x47	0	1	5	Returns the balance.

Block information

This set of instructions is related to retrieving various attributes associated with a block. These opcodes are available in the range of `0x40` to `0x45`:

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 timestamp 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

This set of instructions contains all mnemonics that are necessary to store items on stack and memory. Also, the instructions required to control program flow are also included in this range:

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	800	Used to load a word from storage.
SSTORE	0x55	2	0	0	Saves a word to 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 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. The PUSH operation reads bytes arrays of the program code.

Mnemonic	Value	POP	PUSH	Gas	Description
PUSH1 . . . PUSH32	0x60 ... 0x7f	0	1	3	Used to place N right-aligned big-endian byte item(s) on 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

The `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 `N`th 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 to the substate 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 the 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 nine opcodes available in total in this category:

Mnemonic	Value	POP	PUSH	Gas	Description
`CREATE`	`0xf0`	3	1	32,000	Used to create a new account with the associated code.
`CALL`	`0xf1`	7	1	40	Used to initiate a message call in a contract account.
`CALLCODE`	`0xf2`	7	1	40	Used to initiate a message call in this account with a different account's code.
`RETURN`	`0xf3`	2	0	0	Stops the execution and returns output data.

`DELEGATECALL`	`0xf4`	6	1	40	This is the same as `CALLCODE`, but does not change the current values of the sender and the value.
`STATICCALL`	`0xfa`	6	1	40	This is like the `CALL` instruction. The only exception is that state-changing operations are not permitted.
`CREATE2`	`0xf5`	4	1	32,000	Creates a contract where the address is known prior to deployment.
`REVERT`	`0xfd`	2	0	0	This stops execution and reverts any state changes without consuming all the gas provided.
`SELFDESTRUCT`	`0xff`	1	0	5,000	Stops (halts) execution and the account is registered for deletion later.

In this section, all EVM opcodes have been discussed. There are approximately 133 opcodes available in the EVM of the Istanbul release of Ethereum in total.

Chapter 13

Local Ethereum block explorer

Local Ethereum block explorer is a useful tool that can be used to explore the local privatenet blockchain. This is just one example of many open source block explorer projects.

The one used in this example has been picked up for its simplicity, as the aim is to show the usefulness of a block explorer. Whether a developer develops their own, uses a commercial one, or works with an open source one, the aim is the same—to view the transactions history in a consolidated fashion, in an easy-to-use interface.

There is an open source free block explorer available on GitHub at `https://github.com/etherparty/explorer`. We will use this software in our examples to visualize blocks and transactions. This project is not developed by the author, it has been picked up merely for its ease of use and what was found at the time of writing this. Full credit goes to the original developers, the author is merely using it to show the concept of block explorers and how they can be used in private blockchains.

This project can be installed by taking the following steps. Node.js and NPM installation is required on the computer you are using before we can install the explorer. You can check the official website for installation instructions and download Node.

The Node.js official website is: `https://nodejs.org/en/`.

On a Linux Ubuntu machine or macOS, run the following command in order to install the local Ethereum block explorer:

```
$ git clone https://github.com/etherparty/explorer
```

This will show an output similar to the following:

```
Cloning into 'explorer'...
remote: Counting objects: 269, done.
remote: Total 269 (delta 0), reused 0 (delta 0), pack-reused 269
Receiving objects: 100% (269/269), 59.41 KiB | 134.00 KiB/s, done.
Resolving deltas: 100% (139/139), done.
```

The next step is to change the directory to the explorer and run the following commands:

```
$ cd explorer/
$ npm start
```

Once the installation is finished (it may take almost 5 minutes), an output similar to the following will be shown, where the HTTP server for Ethereum explorer starts up:

```
> EthereumExplorer@0.1.0 prestart /Users/drequinox/explorer
> npm install


> EthereumExplorer@0.1.0 postinstall /Users/drequinox/explorer
> bower install

audited 768 packages in 3.165s
found 82 vulnerabilities (21 low, 24 moderate, 37 high)
  run `npm audit fix` to fix them, or `npm audit` for details

> EthereumExplorer@0.1.0 start /Users/drequinox/explorer
> http-server ./app -a localhost -p 8000 -c-1

Starting up http-server, serving ./app on port: 8000
Hit CTRL-C to stop the server
```

Ethereum explorer HTTP server

While messages regarding vulnerabilities are important in a production environment, we are not overly concerned with those for now as long as the server starts up after compilation.

We have covered the Geth installation process already in this book. If Geth is already running, we can simply browse to the localhost on `TCP 8000`.

Otherwise, Geth can be restarted using the following command:

```
$ geth --datadir ~/etherprivate/ --allow-insecure-unlock --networkid 786 --rpc
--rpcapi 'web3,eth,net,debug,personal'  --rpccorsdomain '*'
```

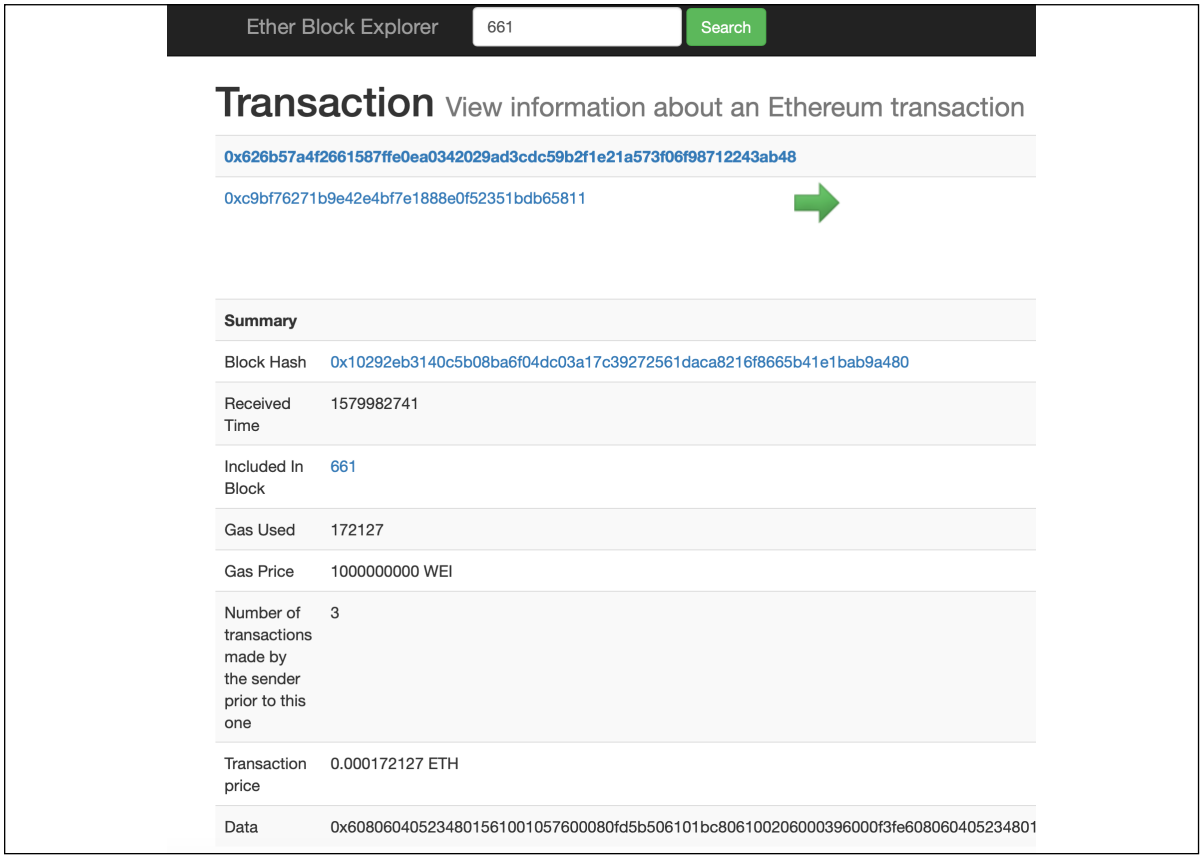
Alternatively, as a simpler option, we can use the following command, as we do not necessarily need the unlocking of accounts or RPC APIs. The purpose here is to just demonstrate how block explorers work:

```
$ geth --datadir ~/etherprivate/ --networkid 786 --rpc
--rpccorsdomain '*'
```

After a successful startup of `geth`, navigate to `localhost` on TCP port `8000`, as shown here, in order to access the local Ethereum block explorer.

We can put the same transaction hash that was generated when deploying a previous contract from *Chapter 13*, *Ethereum Development Environment*. The transaction is:

`0x626b57a4f2661587ffe0ea0342029ad3cdc59b2f1e21a573f06f98712243ab48`



Block explorer

Once entered, the explorer will display the relevant details, as shown in the preceding screenshot. If it does not, an error like the one shown in the following screenshot may occur:

Error message

In this case, restart `geth` to allow `rpccorsdomain`:

```
$ geth --datadir ~/etherprivate/ --networkid 786 --rpc --rpcapi
'web3,eth,net,debug,personal'--rpccorsdomain '*'
```

`'*'` means that any IP can connect. You can also use your computer's local loopback IP address, for example, `127.0.0.1`, or local private network address, in this case, `192.168.0.17`.

In this section, we covered how a block explorer can be used to view blocks and transactions. Usually, block explorers are available for public networks, however in this section, we downloaded an open source block explorer, installed it, and used it to view blocks and transactions of our privatenet. Block explorers are important visualization tools in any network, and especially on a local network, it is equally important, if not more, to monitor all transactions, and block explorers provide an easy way to visualize what's going on in the network. Here we only touched a few basic points but readers can further experiment with different options available in block explorers such as more details about the transactions, associated data, smart contracts, and block details.

Chapter 15

Developing a proof of idea project

In this section, we will create a complete end-to-end DApp project covering smart contract coding, deployment, and frontend design. The idea behind this program is to provide a service to keep a record of **ideas**. This can then be used as proof that at a certain time in the past, the claimant has had access to a certain piece of information. This can be very useful for patent documents.

For example, if someone has come up with an idea, they can then create a hash of that document and save it on the blockchain. Due to the immutable nature of blockchain, it can serve as permanent proof that a certain idea (documents) existed at a certain time. There are many ways in which this can be achieved, but the key idea is the same and it works on the principle that hash functions provide a digest of the text or document and are unique.

This can be achieved in several ways by using different hash functions; the key idea is to create a hash of the document or text string and save it on the blockchain. Once the text is hashed and saved, further requests to save that same text can be disallowed by comparing the hash of the document with the already-stored hash.

For this example, the Remix IDE, Truffle, and testnet (already running network ID 786, created *Chapter 13*, *Ethereum Development Environment*) will be used. This example will provide you with the opportunity to learn how a contract project can be developed from an idea into Solidity contract source code and finally to deployment.

Creating the contract

First, the code for the contract will be written. This can be done using any appropriate text editor or integrated development environment such as the Remix IDE or Visual Studio Code. The Remix IDE can also be used as that too provides a simulated environment for the test.

The complete contract source code is shown as follows:

```
pragma solidity ^0.5.0;
contract PatentIdea {
  mapping (bytes32 => bool) private hashes;
  bool alreadyStored;
  int tracker=0;
  event ideahashed(bool);
  function saveHash(bytes32 hash) private {
    hashes[hash] = true;
  }
  function SaveIdeaHash(string memory idea) public returns (bool){
    bytes32 hashedIdea = HashtheIdea(idea);
   if (alreadyHashed(HashtheIdea(idea)))  {
       alreadyStored = true;
       emit ideahashed(false);
       return alreadyStored;
    }
       saveHash(hashedIdea);
          tracker = tracker+1;
       emit ideahashed(true);
  }
  function alreadyHashed(bytes32 hash) private view returns(bool) {
    return hashes[hash];
  }
  function isAlreadyHashed(string memory idea) public view returns (bool) {
    bytes32 hashedIdea = HashtheIdea(idea);
    return alreadyHashed(hashedIdea);
  }
   function HashtheIdea(string memory idea) private pure returns (bytes32) {
      return  bytes32(keccak256(abi.encodePacked(idea)));
  }
    function getTracker() public view returns (int) {
      return tracker;
  }
}
```

Let's look at the code line by line. This statement ensures that the minimum compiler version is 0.4.0 and the maximum version cannot be greater than 0.4.9. This ensures compatibility between programs:

```
pragma solidity ^0.5.0;
```

This statement is the start of the contract with the name `PatentIdea`:

```
contract PatentIdea {
```

In the following code line, a mapping is defined, which maps `bytes32` to Boolean, which is basically a hashmap (dictionary) of `bytes32` mapping to a Boolean value:

```
mapping (bytes32 => bool) private hashes;
```

This is a variable declared with the `alreadyStored` name, which is a Boolean type and can have a `true` or `false` value. This variable is used to hold the return value from the `SaveIdeaHash` function:

```
bool alreadyStored;
```

An event is declared as well, which will be used to capture the failure or success of the hashing function (`SaveIdeaHash`). When the event is triggered, it will return a `true` or `false` Boolean value.

```
emit event ideahashed(bool);
```

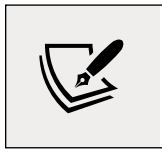
A function named `saveHash` is declared, which takes the hash variable of type `bytes32` as parameters and saves it in the hash map. This will result in a change of the state of the contract. Note that the function accessibility is changed to `private` as it is only required internally in the contract and does not need to be exposed publicly:

```
function saveHash(bytes32 hash) private {
  hashes[hash] = true;
}
```

Another function, `SaveIdeaHash`, is declared, and it takes the variable idea of type string and returns a Boolean (`true` or `false`) depending on the outcome of the function:

```
function SaveIdeaHash(string memory idea) public returns (bool){
  bytes32 hashedIdea = HashtheIdea(idea);
 if (alreadyHashed(HashtheIdea(idea)))  {
     alreadyStored = true;
     emit ideahashed(false);
     return alreadyStored;
  }
     saveHash(hashedIdea);
        tracker = tracker+1;
     emit ideahashed(true);
}
```

This function has a variable declared `hashedIdea`, which is assigned a value after calling the `HashtheIdea` function described later.



Note that this function can also return a value if saved, but it is not shown here for simplicity.

The next function is the `alreadyHashed` function, which is declared to take the variable named `hash` of type `bytes32` and returns a Boolean (either `true` or `false`) after checking the `hash` in the hash map. This is again declared as a constant and accessibility is set to `private`:

```
function alreadyHashed(bytes32 hash) private view returns(bool) {
  return hashes[hash];
}
```

The next function is `isAlreadyHashed`, which checks whether the "idea" is already hashed. This takes the input parameter idea of type string such as "my idea" (also declared as a constant, meaning that it cannot change the state of the contract), and returns either `true` or `false` based on the outcome of the execution of the function named `alreadyHashed`. This function then calls the `alreadyHashed` function described earlier to check from the hashes map whether the hash is already stored there. This would mean that the same string (idea) has already been hashed and stored (patented):

```
function isAlreadyHashed(string memory idea) public view  returns (bool) {
  bytes32 hashedIdea = HashtheIdea(idea);
  return alreadyHashed(hashedIdea);
}
```

Finally, the `HashtheIdea` function is shown here, which takes the `idea` variable of type `string` and is of constant type, which means that it cannot change the state of the contract. It is also declared as `private` as there is no need to expose this function publicly because it is only used internally in the contract. This function returns the `bytes32` type value:

```
function HashtheIdea(string memory idea) private pure returns (bytes32) {
    return  bytes32(keccak256(abi.encodePacked(idea)));
}
```

The preceding function calls Solidity's built-in function `sha3` and passes a string to it in a variable `idea`, then returns the SHA3 hash of the string. The `sha3` function is an alias for the `keccak256()` function available in Solidity, which computes the `keccak-256` hash of the string passed to it.

The SHA3 function used in Solidity is not the NIST standard SHA-3; instead, it is Keccak-256, which is the original proposal to NIST for the SHA-3 standard competition. It was later modified slightly and standardized as the SHA-3 standard by NIST. The actual SHA-3 standard hash function will return a different hash compared to Keccak-256 (Ethereum's `sha3` function).

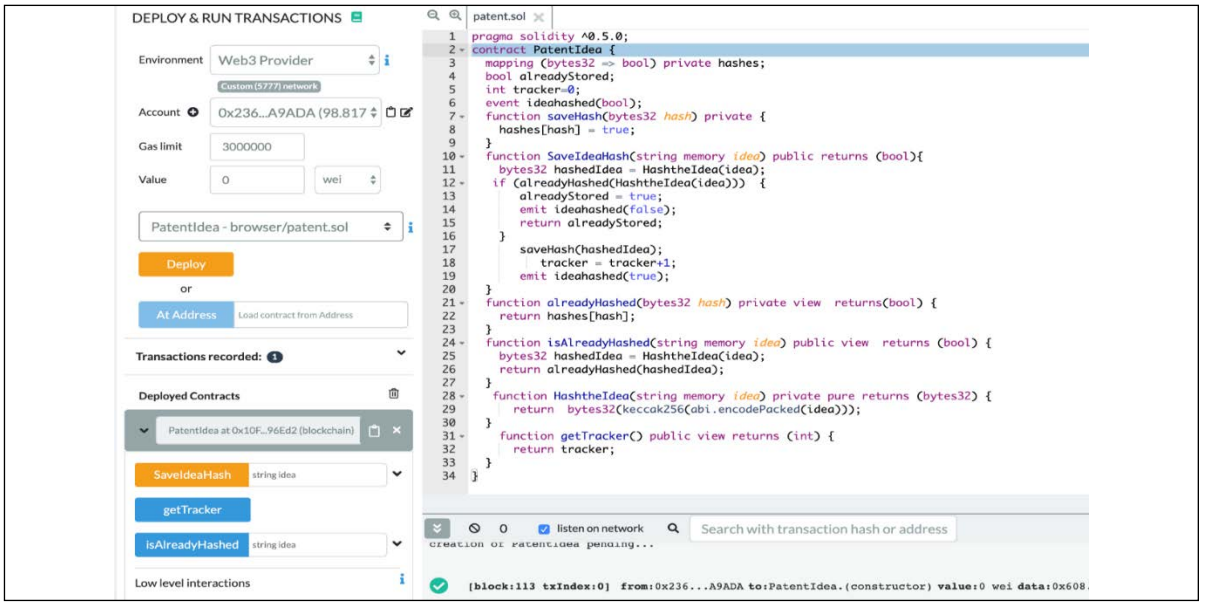
Finally, we have the following code, which is a function that returns the number of hashed ideas so far:

```
function getTracker() public view returns (int) {
      return tracker;
```

This function is based on the following code, where `tracker` variable is declared as an integer and initialized to `0`. The `tracker` increments with `1`, when a string is hashed successfully. This variable simply keeps track of all the hashed idea strings.

```
int tracker=0;

tracker = tracker+1;
```

This source code can be simulated in the Remix IDE and optionally, Ganache, or it can built in the JavaScript VM in the Remix IDE in order to verify that it is working correctly. An example is shown here:

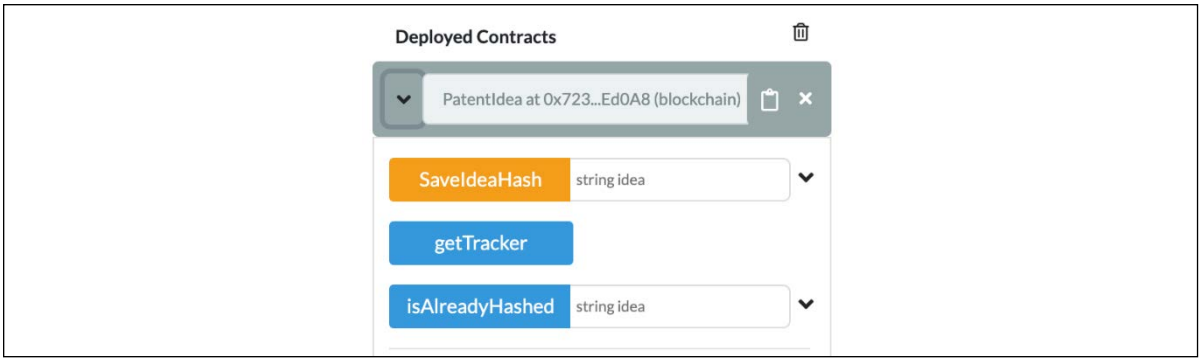


Create contract using browser Solidity

Once the contract source code is typed and syntax verification is complete, on the right-hand panel, a screen similar to the preceding screenshot will appear.

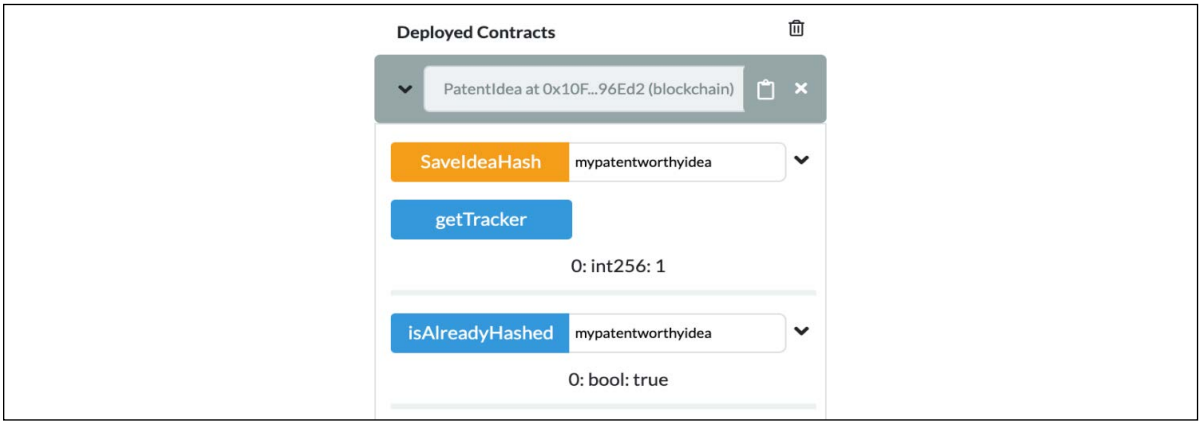
This code can be improved in many ways. For example, the date can also be stored in a mapping with the document hash and can be returned when queried. It can be expanded by adding structures and more information related to the patent, but this example was intended to be simple and easy to understand; therefore, too much complexity was avoided. The code provided here is complete and will provide you a full working example of how the "proof of idea" contract can be implemented. You can do further enhancements to this as an exercise.

After clicking on **Deploy**, two functions from the contract will be exposed, as shown in the following screenshot:



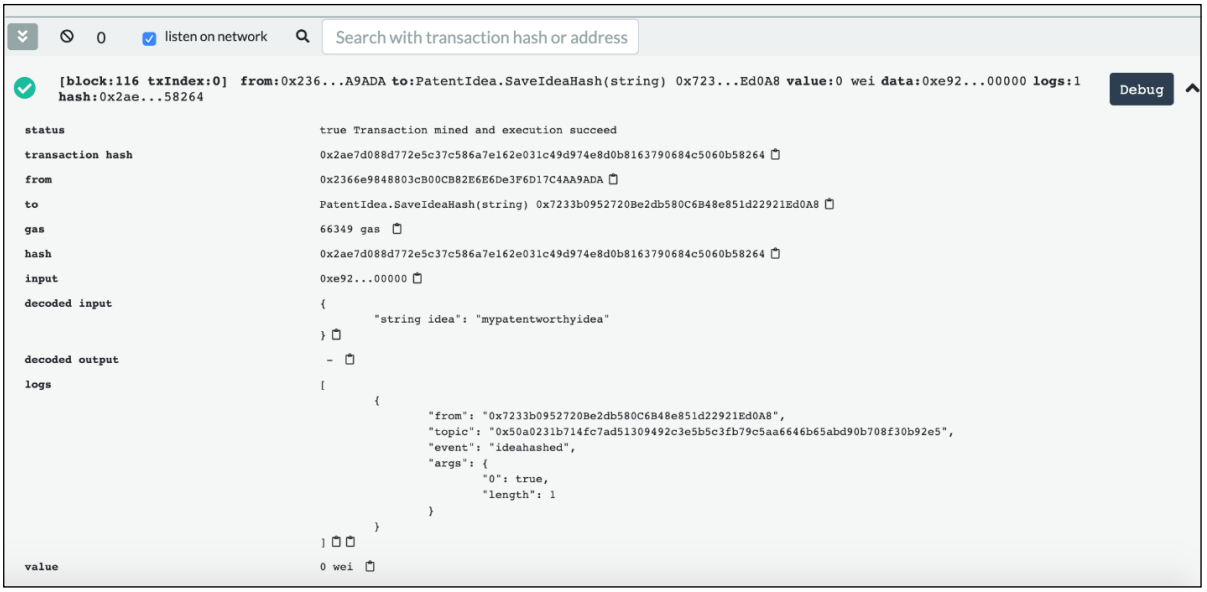
Available methods

Functions **isAlreadyHashed** (to check if the idea is already hashed) and **SaveIdeaHash** (to save the new idea string) can now be invoked. Function **getTracker** is used to track the number of patented ideas so far. This is shown in the following screenshot:



Deployed contracts

Now if we look at the logs produced in the Remix IDE shown at the bottom of the IDE, we can see helpful details:

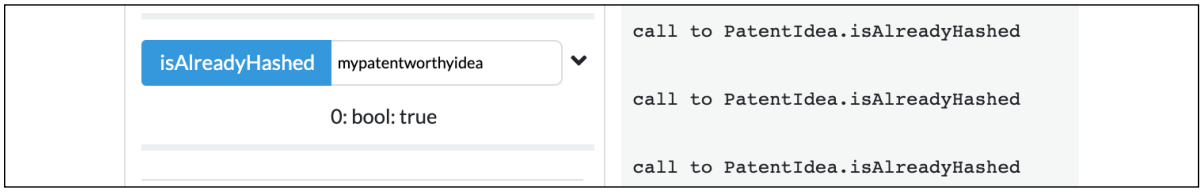


Logs

The log shows valuable information, such as:

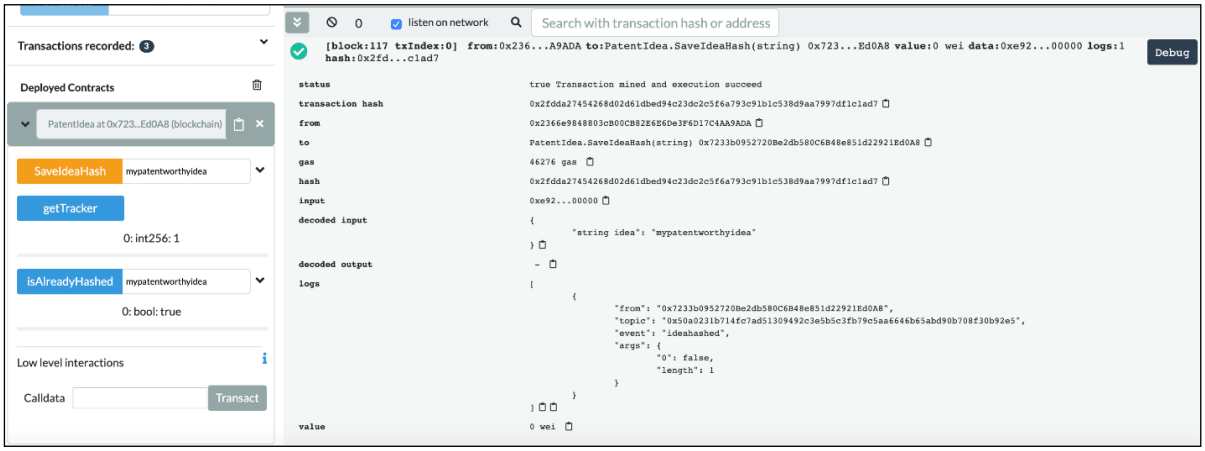
- `status`: This is `true` in the example, meaning that transaction has been mined and executed successfully.
- `transaction hash`: Hash of the transaction.
- `from`: This is the address of the account from which the contract was initiated.
- `to`: This is the address of the contract on the blockchain.
- `gas`: This shows how much gas is sent.
- `hash`: This is the hash of the contract.
- `input`: This is the input shown in hex.
- `decoded input`: This shows the decoded input.
- `decoded output`: This shows the decoded output.
- `logs`: This shows transaction logs with relevant events.
- `value`: This shows the value in Wei in the contract.

Similarly, **isAlreadyHashed** can be called. You can also explore the logs to find more details about the execution:



Execute function isAlreadyHashed

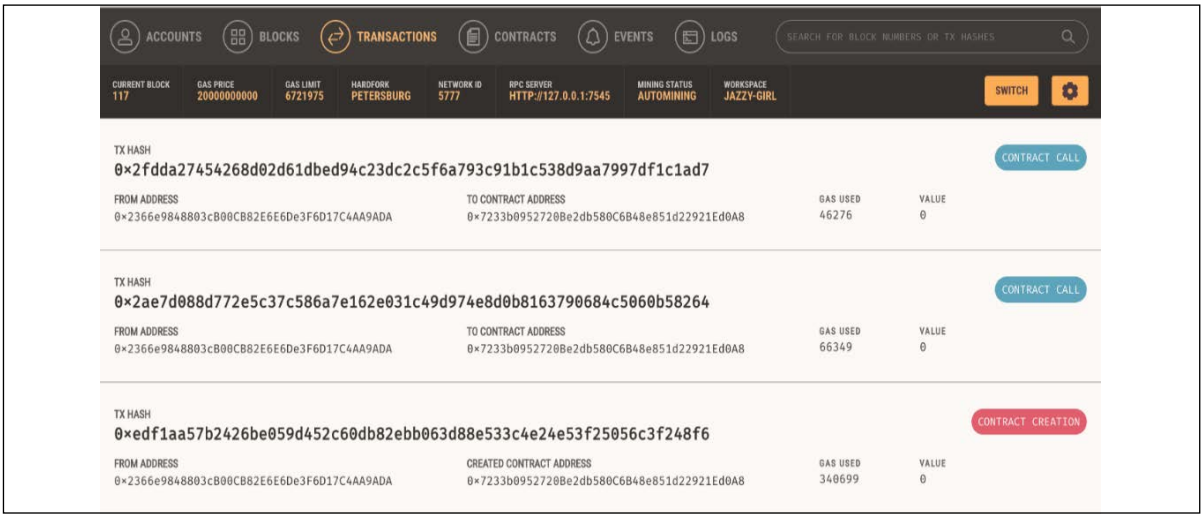
If the same string is passed to the function again, it will not be saved, as shown in the following screenshot:



Execute function SaveIdeaHash

Note that the event has returned `false`, indicating that the hash could not be saved and the function returned `true`, further indicating that the same hash is already saved.

As we have been using Ganache as the local blockchain providing Web3, we can see all the relevant transactions in the Ganache frontend as shown in the following screenshot:



Ganache showing all transactions issued via the Remix IDE

Now, as we have run and tested our smart contract on Ganache using the Remix IDE, we can deploy it to our privatenet. We will describe this process in the next section.

Deploying the contract with Truffle

Once the contract is written and simulated in the Remix IDE using Ganache, the next step is to use Truffle to initialize a new project and deploy and test it on the privatenet (ID `786`), created earlier in *Chapter 13*, *Ethereum Development Environment*:

1. The first step is to create a separate directory, `ideap`, for the project:

```
$ mkdir ideap
$ cd ideap
```

2. The next step is to initialize Truffle and create a new project:

```
$ truffle init
✓ Preparing to download box
✓ Downloading
✓ cleaning up temporary files
✓ Setting up box
```

   It will create a structure as shown here:

```
$ tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
└── truffle-config.js

3 directories, 3 files
```

3. Under the `contracts` folder, create a file named `PatentIdea.sol` and put the source code of `contract PatentIdea` (shown in the *Creating the contract* section) in it.
4. Edit `truffle-config.js` to point to the `localhost` HTTP endpoint. This is our private network `786`:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",     // Localhost (default: none)
      port: 8545,            // Standard Ethereum port (default: none)
      network_id: 786,       // Any network (default: none)
    }
  }
}
```

5. Under the `~/ideap/migrations` folder, create the `2_deploy_contracts.js` file so that it looks like the following:

```
var PatentContract = artifacts.require("PatentIdea");

module.exports = function(deployer) {
  deployer.deploy(PatentContract);
};
```

6. Next, run the compilation using Truffle, as shown here:

```
$ truffle compile
```

This will show the output shown following indicating the compilation of the contracts:

```
Compiling your contracts...
===========================
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/PatentIdea.sol
> Artifacts written to /Users/drequinox/ideap/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
```

7. Ensure that mining is running in the background and deploy to the network, as shown here:

```
$ truffle migrate
```

This will start the contract migration process and produce the following output:

```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.

Starting migrations...
======================
> Network name:    'development'
> Network id:      786
> Block gas limit: 0x92a4ac


1_initial_migration.js
======================

   Deploying 'Migrations'
   ----------------------
   > transaction hash:
0xb70804703a00829cd1a2eebd230cbc9238740b879a1f52d66e4819f6596763b8
   > Blocks: 0            Seconds: 12
```

```
   > contract address:    0x851231d78764c557E996037b233D2cE7E3c0387b
   > block number:        5718
   > block timestamp:     1581268685
   > account:             0xc9Bf76271b9E42E4bF7E1888e0F52351bDb65811
   > balance:             28489.9999999999999989
   > gas used:            194469
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00388938 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00388938 ETH


2_deploy_contracts.js
=====================

   Deploying 'PatentIdea'
   ----------------------
   > transaction hash:
0xf1d489b3df9369877701b7562ed9b5fd948286a82d28e87e3a18d40e49f79900
   > Blocks: 0            Seconds: 0
   > contract address:    0xe7E76d7317235D20a437f37568aBC30B4E9c6A23
   > block number:        5722
   > block timestamp:     1581268712
   > account:             0xc9Bf76271b9E42E4bF7E1888e0F52351bDb65811
   > balance:             28509.9999999999999989
   > gas used:            346685
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0069337 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0069337 ETH


Summary
=======
```

```
> Total deployments:   2
> Final cost:          0.01082308 ETH
```

Once the contract is deployed, it can be interacted with using the `truffle console`.

8. Start the `truffle console` by issuing the following command:

```
$ truffle console
```

9. Once the console is up and running, functions from the deployed contract can be called. For example, to register a new idea, type the following commands in the `geth` console:

```
truffle(development)> PatentIdea.deployed().then(function(instance){app =
instance})
undefined

>truffle(development)> app.SaveIdeaHash("hello1")
```

This will show an output similar to the following:

```
{ tx:
   '0x7c5d5a40bb905d8521e5c33c27699f229d303a917a9dd4dd6dfcb648c3dc8e7b',
  receipt:
   { blockHash:
      '0x24343abfbb320167a7c6b0ec4eda9d0e2ee8567c91f072ea97620824cd81c870',
     blockNumber: 5731,
     contractAddress: null,
     cumulativeGasUsed: 65583,
     from: '0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811',
     gasUsed: 65583,
     logs: [ [Object] ],
     logsBloom:
      '0x000000000000100000000000000000000000000000000000440000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000004000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000040000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000002000000000000000000000000',
     root:
      '0xefa61499a1cad32dc6b2bc95fd2a15639ecc0e4354b341f77aefce6b6d7bcb57',
     to: '0xe7e76d7317235d20a437f37568abc30b4e9c6a23',
     transactionHash:
      '0x7c5d5a40bb905d8521e5c33c27699f229d303a917a9dd4dd6dfcb648c3dc8e7b',
     transactionIndex: 0,
     rawLogs: [ [Object] ] },
  logs:
   [ { address: '0xe7E76d7317235D20a437f37568aBC30B4E9c6A23',
       blockNumber: 5731,
       transactionHash:
        '0x7c5d5a40bb905d8521e5c33c27699f229d303a917a9dd4dd6dfcb648c3dc8e7b',
       transactionIndex: 0,
       blockHash:
        '0x24343abfbb320167a7c6b0ec4eda9d0e2ee8567c91f072ea97620824cd81c870',
       logIndex: 0,
       removed: false,
       id: 'log_11ca5e88',
       event: 'ideahashed',
       args: [Result] } ] }
```

Invoking deployed contract methods in the Truffle console

10. If the account is locked, you can unlock it using the following command:

```
> personal.unlockAccount(web3.eth.coinbase, "Password123", 0)
```

Note that we have used another parameter here, `0`, which represents the duration in seconds for which we want to keep the account unlocked. If it is set to `0`, the account will remain unlocked until the `geth` client is restarted. If for example this parameter is set to `600`, the account will remain unlocked for 10 minutes.

11. Check whether `"hello1"` is hashed:

```
truffle(development)> app.isAlreadyHashed("hello1");
```

This will return an output of `true`.

12. Check whether another idea is hashed or not; in this example we've used string `"hello3"`:

```
truffle(development)> app.isAlreadyHashed("hello3");
```

    This time the output will be negative, `false`.

13. Check what the current number of the patented ideas is:

```
truffle(development)> app.getTracker()
```

    This will return an output of the number of patented ideas:

```
<BN: 1>
```

This example demonstrated how a contract can be created from scratch, simulated, and deployed on the privatenet. In order to deploy this on testnet or a live blockchain, a similar exercise can be performed. Simply point to the appropriate RPC and use `truffle migrate` to deploy on the blockchain of your choice.

So far, we have built web frontends manually using HTML and JavaScript. However, with the availability of new tools such as Drizzle, we can develop faster and more easily! We'll consider this in the next section.

Building a UI using Drizzle

In this section, we will build the UI frontend for our DApp. There are a few pre-requisites that we need to set up before we can use Drizzle.

Note that we are still in our `ideap` directory that we created in the previous section.

We installed `npm` and `node` in the *Chapter 14*, *Development Tools and Frameworks*. The versions that we are using are listed as follows:

```
$ npm -v
6.13.4
$ node -v
v10.18.0
```

Drizzle is the core library that instantiates Web3, accounts, and contracts, along with required synchronizations and contract functionality.

Drizzle installation is performed using the command as follows:

```
$ npm install drizzle –save
```

`drizzle-react` provides two helper methods to allow easy connectivity between the `react` app and Drizzle. `drizzle-react-components` is a library that makes it very easy to interact with the contracts. It contains components called `ContractData`, `ContractForm`, and `LoadingContainer`. These components provide common decentralized application (DApp) functions for interaction with the contracts.

Install these packages with the following commands:

```
$ npm install --save drizzle drizzle-react
$ npm install --save drizzle drizzle-react-components
```

Finally, install `create-react-app`:

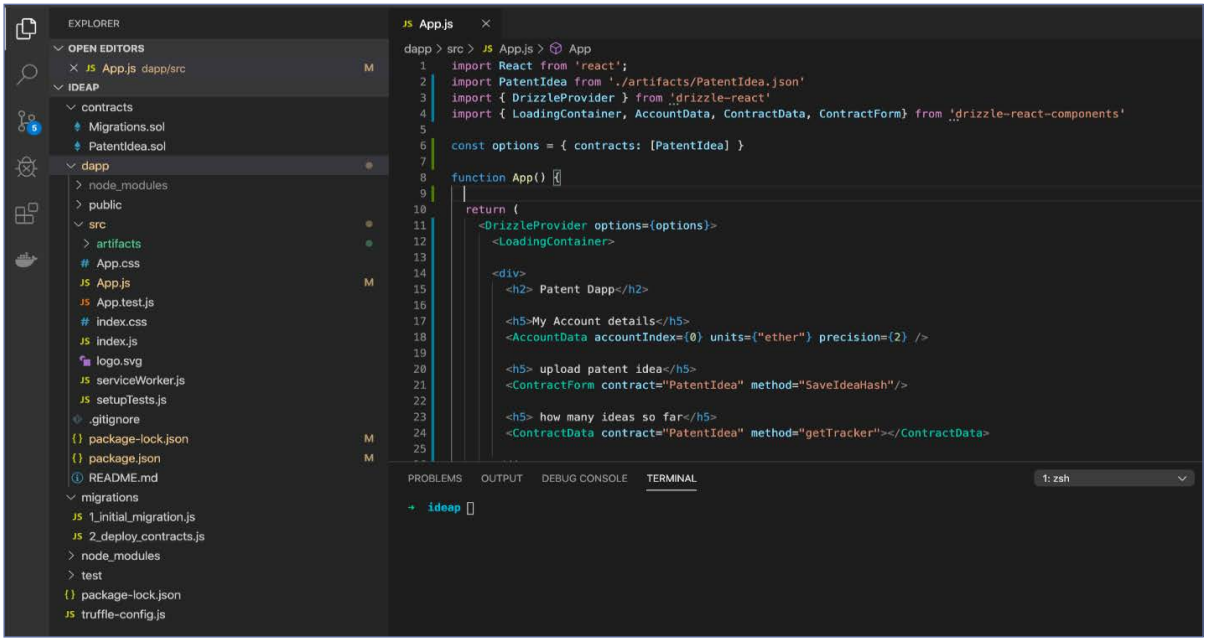
```
$ npm install create-react-app
```

Now we will initialize a new `react` app:

```
$ npm init react-app dapp
```

Now when the React app is created, within the `ideap` directory, a directory with the name `dapp` will appear. At this point, Visual Studio Code can be launched to further work on the project.

The Visual Studio Code IDE with the `ideap` directory and newly created `dapp` directory is shown in the following screenshot:



Visual Studio Code shows the ideap directory

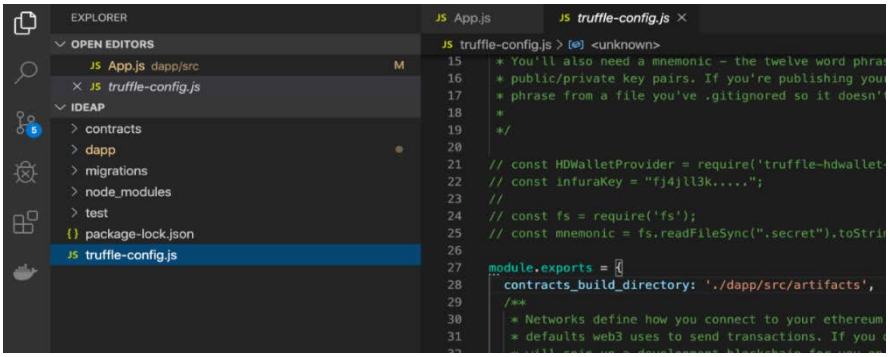
Change the contract compilation output location so that Truffle contracts can be compiled into this new directory.

Edit the `truffle-config.js` file that we introduced earlier in the *Compiling, testing, and migrating using Truffle* section. It is present in the `dapp` directory. Open this file in Visual Studio Code and add the following code:

```
contracts_build_directory: './dapp/src/artifacts',
```

This change is required because the DApp can only access the files within the `src` directory. Therefore we want Truffle to compile to the `/dapp/src/artifacts` directory so that the contract artifacts are accessible to our application.

This is shown in the following screenshot:



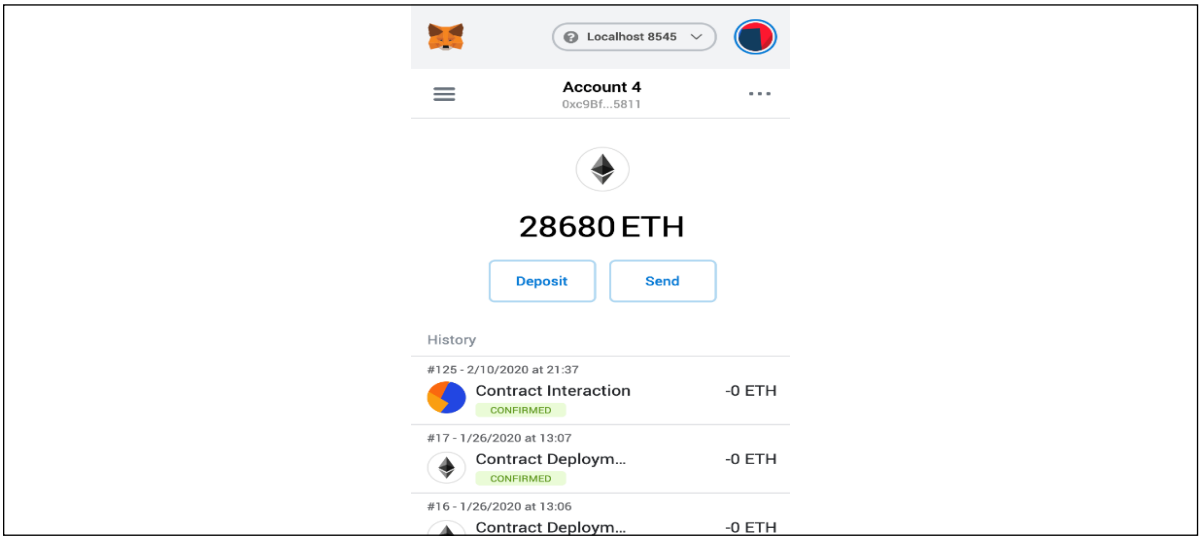
Visual Studio Code showing the Truffle config

Now we compile the contracts using Truffle again:

```
$ truffle compile
```

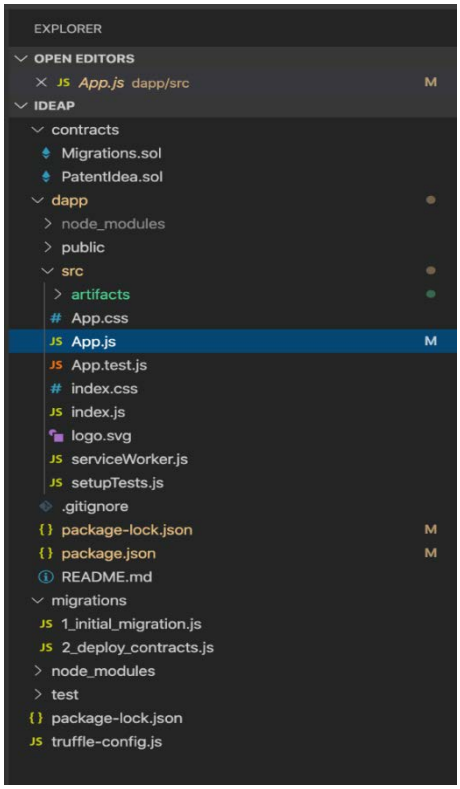
This will create a new directory under `./dapp/src/` called `artifacts`, and the output will be written there. This will allow the contract file to be accessible to code, as all of this is now under the `src` directory.

Now, ensure that MetaMask is running and fully functional in our web browser. Remember our private network ID `786`; we imported the account `0xc9Bf76271b9E42E4bF7E1888e0F52351bDb65811` earlier into MetaMask.



Account imported in MetaMask

In Visual Studio, the directory structure after installing all of the pre-requisites should look like this:

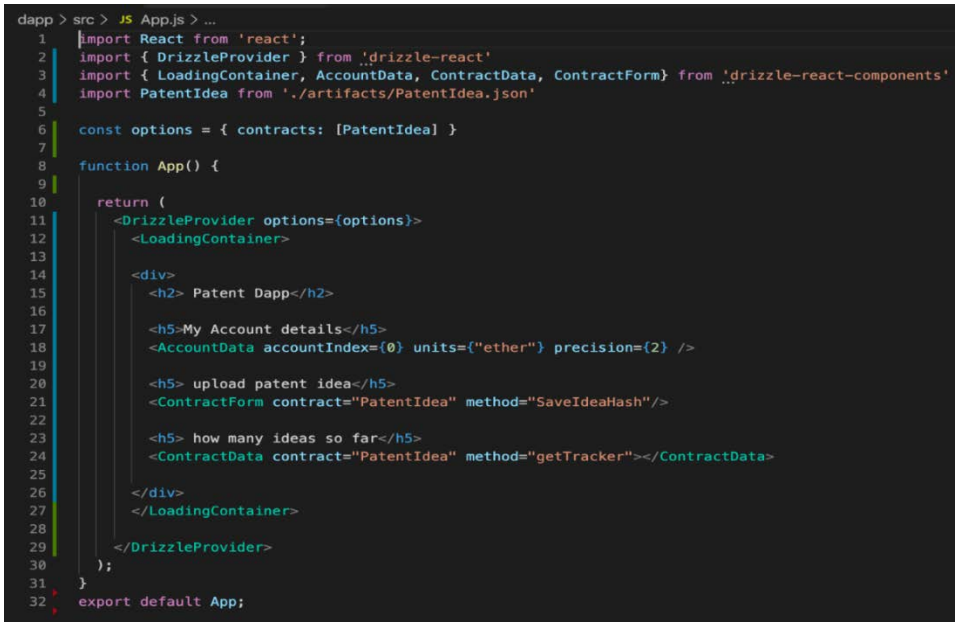


Directory structure of the DApp

We'll now build our frontend in `App.js`. Place the following code into Visual Studio:

```
import React from 'react';
import PatentIdea from './artifacts/PatentIdea.json'
import { DrizzleProvider } from 'drizzle-react'
import { LoadingContainer, AccountData, ContractData, ContractForm} from
'drizzle-react-components'

const options = { contracts: [PatentIdea] }

function App() {

  return (
    <DrizzleProvider options={options}>
      <LoadingContainer>

      <div>
        <h2> Patent Dapp</h2>

        <h5>My Account details</h5>
        <AccountData accountIndex={0} units={"ether"} precision={2} />

        <h5> upload patent idea</h5>
        <ContractForm contract="PatentIdea" method="SaveIdeaHash"/>

        <h5> how many ideas so far</h5>
        <ContractData contract="PatentIdea" method="getTracker"></ContractData>

      </div>
      </LoadingContainer>

    </DrizzleProvider>
  );
}
export default App;
```

The following screenshot demonstrates how the code is displayed in Visual Studio Code:



Code entered in Visual Studio Code

Let's run through an explanation of what the preceding code means. The first steps are to import `React`, `drizzle-react`, and `drizzle-react-components`:

```
import React from 'react';
import { DrizzleProvider } from 'drizzle-react'
import { LoadingContainer, AccountData, ContractData, ContractForm} from
'drizzle-react-components'
```

After that we import the `PatentIdea` artifact from the contract:

```
import PatentIdea from './artifacts/PatentIdea.json'
```

We then create the `options` object, with the `PatentIdea` contract artifacts for instantiation by Drizzle:

```
const options = { contracts: [PatentIdea] }
```

The app is wrapped in `DrizzleProvider` with the `options` object as shown here:

```
<DrizzleProvider options={options}>
```

After this, we load several components. First, the `LoadingContainer` component, which wraps the application and shows a loading screen until drizzle is fully initialized.

```
<LoadingContainer>
```

Next, the `AccountData` component is used to display the balance and addresses of accounts.

```
<AccountData accountIndex={0} units={"ether"} precision={2} />
```

It takes three parameters:

- `accountIndex` (`int`): This takes on an integer parameter, which is the index of the account. Index `0` is the first account.
- `units` (`string`): This takes a string parameter as a denomination of the account balance, such as ether or Wei.
- `precision` (`int`): This takes an integer as a parameter and sets the number of digits after the decimal point.

The `ContractForm` component is used to automatically generate a form with required UI elements to take the inputs and interact with the smart contract:

```
<ContractForm contract="PatentIdea" method="SaveIdeaHash"/>
```

It has the following parameters:

- `contract` (`string`): This parameter takes the name of the contract as a string. This is a mandatory parameter.
- `method` (`string`): This parameter takes the name of the method (function) in the contract from which the corresponding form fields will be created automatically. This is a mandatory parameter.
- `sendArgs` (`object`): This is an object that specifies various transaction parameters such as `from`, `gasPrice`, `gas`, and `value`. This is an optional parameter.
- `labels` (`array`): This can be used to provide custom labels instead of ABI-based names. This is helpful when creating user-friendly forms to specify custom and detailed labels. This is an optional parameter.

The `ContractData` component is used to display the output returned by a contract call:

```
<ContractData contract="PatentIdea" method="getTracker"></ContractData>
```

It takes a number of parameters, as described here:

- `contract` (`string`): This parameter takes the name of the contract to call as a string. This is a mandatory parameter.
- `method` (`string`): This parameter takes the name of the method (function) in the contract to call. This is a mandatory parameter.
- `methodArgs` (`array`): This parameter takes an array of arguments for the contract method call.

- `hideIndicator` (`Boolean`): This parameter takes a Boolean value: `true` or `false`. If it is set to `true`, it hides the loading indicator during contract state updates.
- `toUtf8` (`boolean`): This parameter takes a Boolean value: `true` or `false`, and converts the return value into a UTF-8 string before display.
- `toAscii` (`Boolean`): This parameter takes a Boolean value: `true` or `false`, and converts the return value into an ASCII string before display.

Once we have placed all the code in relevant files, we can start up the application by issuing the command shown here from within the directory `dapp`:

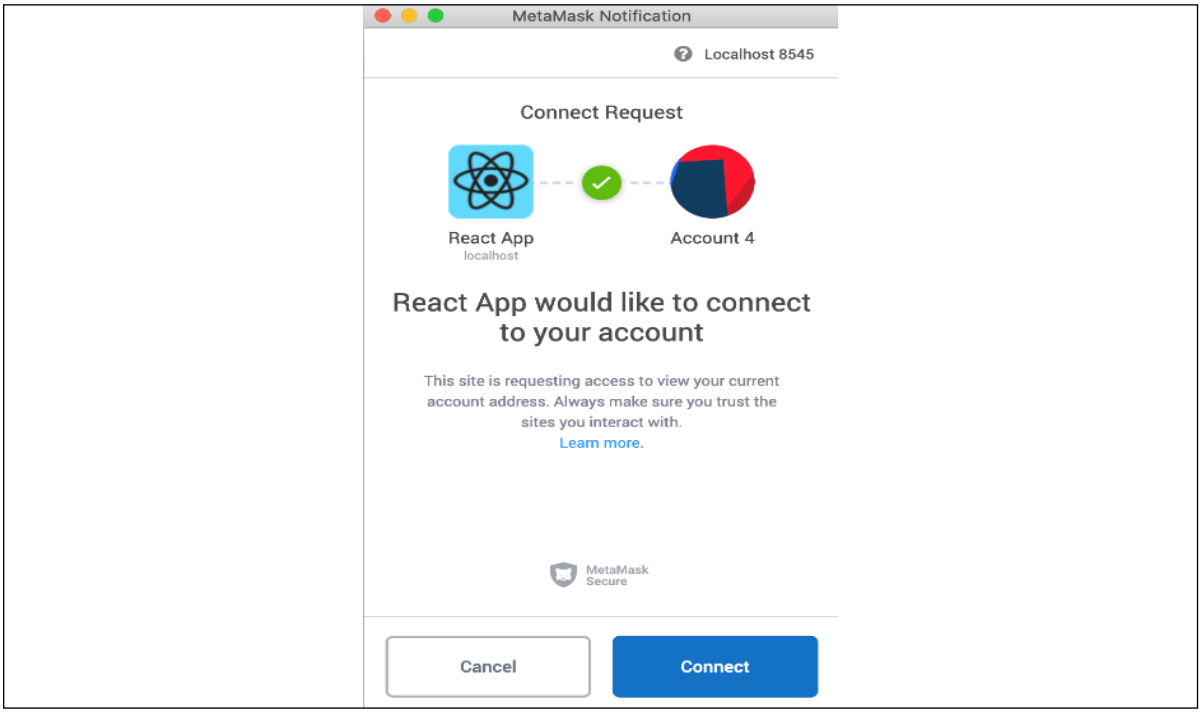
```
$ npm start
```

This will show an output similar to the one shown here:

```
Compiled successfully!

You can now view dapp in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.0.18:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

npm start output

The first time when the server starts up, it will ask to confirm the connection in MetaMask:



MetaMask asking for connection

Make sure that network ID **786** (our private network) is running in the background correctly and mining. If it's not mining the transactions will not work.

Remember to start the network if it's not running by using the following commands:

```
$ geth --miner.gastarget 90000000000 --miner.gaslimit 90000000000 --
datadir ~/etherprivate --networkid 786 --allow-insecure-unlock --rpc
--rpcapi "web3,net,eth,debug,personal" --rpccorsdomain "*" --debug --
vmdebug –nodiscover
```

Then mining is started, as shown following, by using `geth attach`:

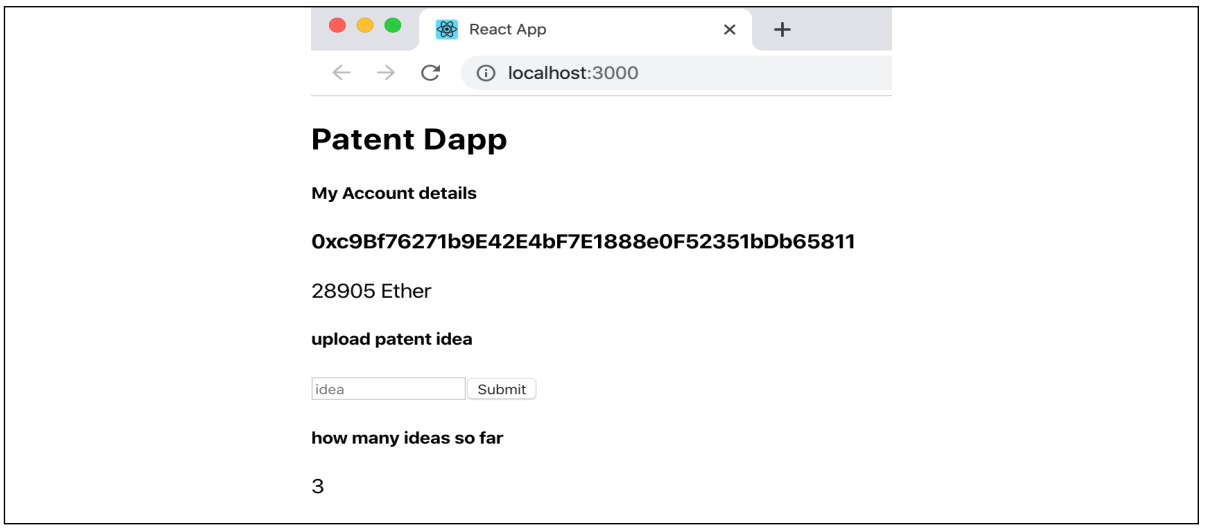
```
$ geth attach ~/etherprivate/geth.ipc
```

The output of this command will appear as follows:

```
➜  ~ geth attach ~/etherprivate/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.10-stable/darwin-amd64/go1.13.6
coinbase: 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
at block: 5666 (Sun, 09 Feb 2020 17:12:04 GMT)
 datadir: /Users/drequinox/etherprivate
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

> web3.personal.unlockAccount("0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811","Password123",0)
true
> miner.start()
null
```

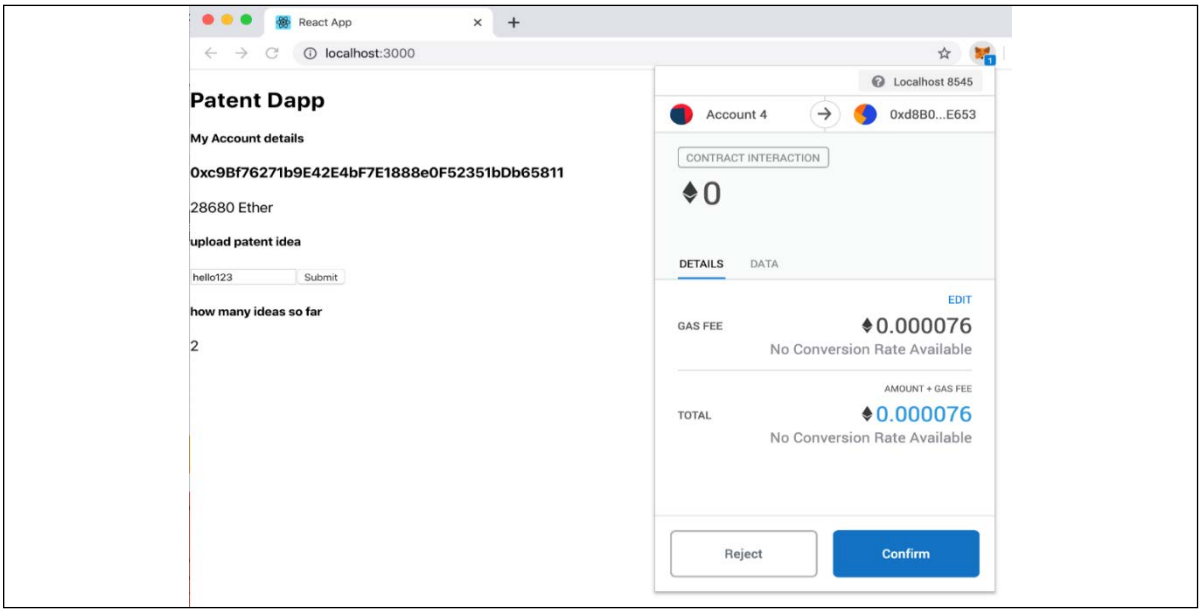
Geth attach

When the page loads up, it will show the UI of our DApp:



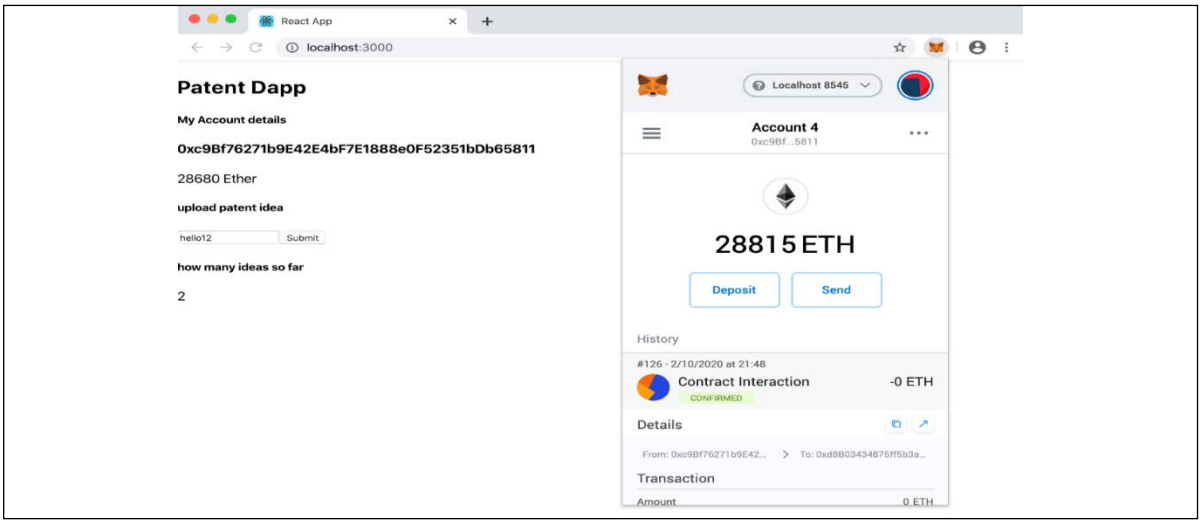
DApp web interface

Now we can interact with the blockchain using this `react` frontend built using Drizzle. Simply enter the "idea" and press **Submit**. Once the button is pressed, MetaMask will invoke and display a dialogue, as shown here, to confirm the transaction:



Interaction with Patent DApp—Confirm

Once confirmed, the contract interaction transaction can be seen in the MetaMask transaction history:



Contract interaction using Patent DApp

With this, we have completed an introduction to Drizzle and how it can be used to create a React-based DApp. In this section, we created a complete end-to-end decentralized application called Proof of Idea. We started with writing smart contract code. Then, we used Truffle to migrate it to our privatenet, interacted with it, and then created a Drizzle frontend for it.

Chapter 16

Running Ethereum 2.0 clients

Now that we have a good understanding of Ethereum 2.0's design goals, its phases, and architecture, we can now look at how Ethereum 2.0 clients can be installed and run. As we saw, there are many teams building Ethereum 2.0 clients. We will look at a client from Prysmatic Labs here.

The Core repository of Prysmatic's Ethereum 2.0 implementation is available at `https://github.com/prysmaticlabs/prysm`.

There is also a testnet available from Prysmatic named the **Onyx Testnet**. There is another testnet available called **Altona**.

We will use the Onyx testnet and the Prysmatic Ethereum 2.0 Beacon chain client to demonstrate how the beacon chain works.

In summary, the process consists of seven steps, which are as follows:

1. Obtain ether (in this case, for the Goerli testnet)
2. Install a validator and some beacon chain client software
3. Generate a validator public/private key pair
4. Start the beacon chain client
5. Send a deposit of ether to the deposit contract on the Ethereum 1.0 chain
6. Start the validator client
7. Wait for the validator assignment

After these steps have been completed, we will have a beacon node and validator node running on the beacon chain.

We now look at these steps in detail.

Obtaining ether for the Goerli testnet

We use Goerli in our example. This is a multi-client **Proof of Authority** (**PoA**) testnet for Ethereum.

If you already have enough ether (32 ETH plus gas fees) available, then this step can be skipped. This exercise assumes that we already have MetaMask running, we are able to connect to testnets, and that we know how to create testnet accounts. If this isn't the case, follow the steps mentioned in *Chapter 12*, *Further Ethereum*, to install MetaMask, then resume this step.

First, create a new account on the Goerli testnet and fund it using a faucet with at least 32 ether.

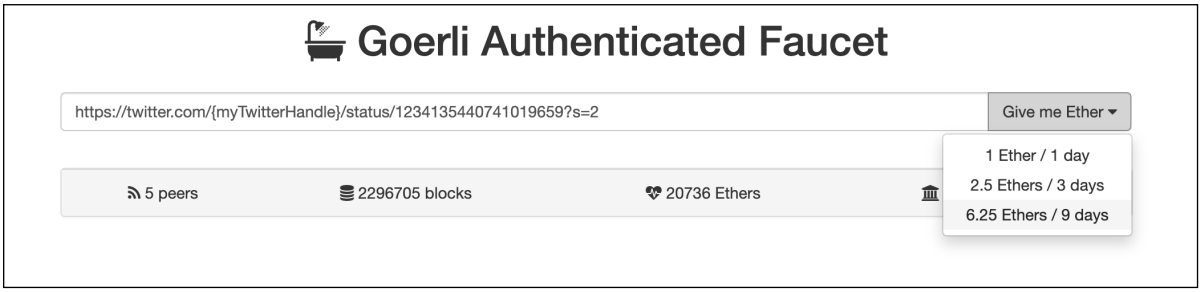
There are a few faucets available for the Goerli test network at the following URLs:

`https://faucet.goerli.mudit.blog/`

`https://goerli-faucet.slock.it/`.

We can also request test ether from `https://prylabs.net/participate`. In this example, we have used this option.

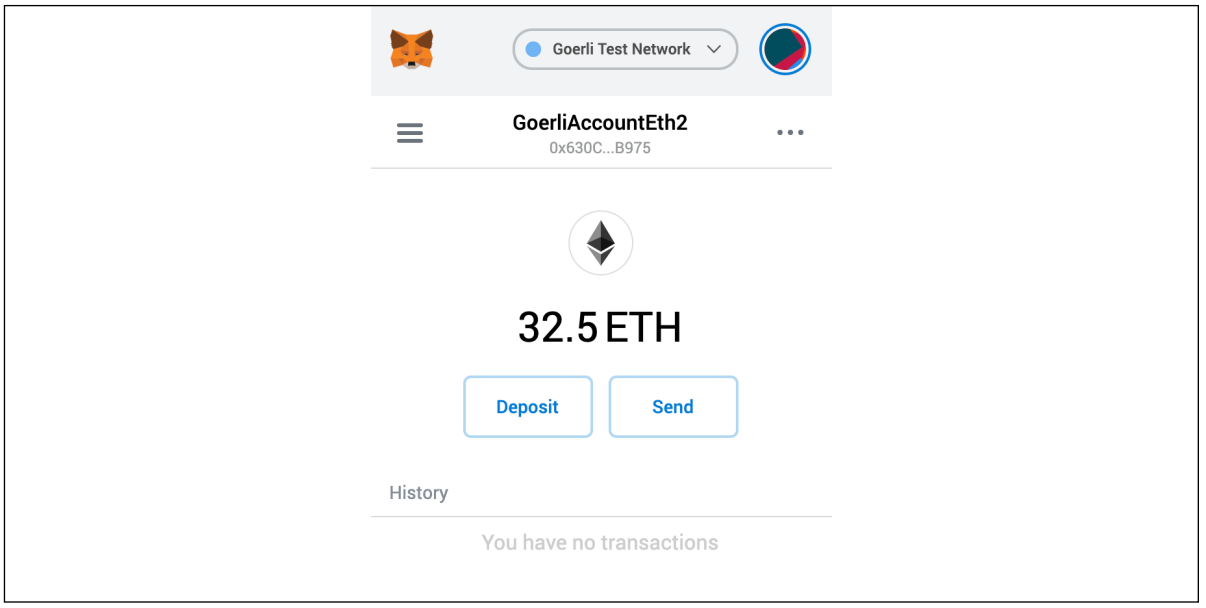
The faucet process is shown in the following screenshots. First, to request ether, the requesters are expected to make a tweet with their Ethereum address pasted in the contents of the tweet. After this, copy and paste the URL of the tweet in the faucet address box, as shown in the following screenshot (replace `{myTwitterHandle}` in the URL in the following screenshot with your Twitter handle):



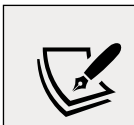
Goerli authenticated faucet

After a few moments, the account will be funded with the required ether. If you use Prysm's website, they will send 32.5 ETH to your account within a few seconds. If you use other methods, such as the **Goerli Authenticated Faucet**, you'll have to repeat this step several times over a period of few days to get the required amount of ether. Whichever way it is done (or perhaps you already have ether), the requirement is to have 32.5 ETH in your account.

Once you have achieved that, we can move to the next step:



MetaMask connected to Goerli test network



Note that the step of getting ether on the Goerli testnet can be performed independently at any time. The requirement is to ensure that we have enough funds available to fund our deposit contract to become a validator on the beacon chain.

Once we have enough ether available, we can proceed to the next step. We use MetaMask for this purpose. We covered how MetaMask can be connected to other networks in *Chapter 12*, *Further Ethereum*. We have connected to the Goerli test network in this example, using our account **GoerliAccountEth2**, `0x630CFD787280ee364c165FA0d15AFA515923B975`.

Installing the client software

The installation of the Prysmatic Labs client can be performed either using Docker or by manually downloading it, building it, and then running it. We'll use the scripts provided by *Prysmatic Labs* to install the client locally. We are using macOS for our example.

Pre-requisites

To complete the following steps in this exercise, you will need to install several packages. Details on how to install each of these are as follows.

GPG

First, we need `gpg`, which is used to verify the installation package. It's not essential but a good practice to install this so that software packages can be verified. To install the `gpg` package, enter the following command at the Terminal:

```
$ brew install gpg
```

Brew will produce a long output and once installed successfully, the installation can be checked by issuing the following command:

```
$ gpg –version
```

This will produce the following output (note: the full output is not shown for brevity):

```
gpg (GnuPG) 2.2.20
```

If there are no errors and an output similar to the preceding is shown, it means that the installation was successful.

Golang

Go is an open source programming language developed at Google. As the Ethereum 2.0 clients we are going to use in this example are written in the Go language (Golang), we need to install the language locally so that the Ethereum 2.0 clients used in this example can be compiled.

We can install Golang's latest version from `https://golang.org/doc/install`. Installation packages are available for different operating systems, but for macOS, we used the `pkg` package to install Golang.

Curl

`curl` is a command-line tool and library usually used for downloads and data transfers of scripts. We'll use that to download the files we need for our installation.

It can be downloaded from `https://curl.haxx.se`.

Git

Git is a distributed version control system. Git may already be available in the operating system you are using, however, if it's not there, it can be downloaded from `https://git-scm.com`.

After all the pre-requisites are correctly installed, we can start the process of installing the Ethereum 2.0 Prysm client.

Installing the Ethereum 2.0 Prysm client

First, we create a directory, so that all the required software can be downloaded into a separate directory:

```
$ mkdir prysm
```

Change directory to `prysm` so that we can start the download process:

```
$ cd prysm
```

Once inside the directory, use `curl` to download the required installation script:

```
$ curl https://raw.githubusercontent.com/prysmaticlabs/prysm/master/prysm.sh
--output prysm.sh && chmod +x prysm.sh
```

This will download the file called `prysm.sh` and assign the execute permission to it so that the shell script can be executed on the operating system.

The next step is to generate a key pair for our validator node so that it can be identified on the network and can provide an attestation service.

Generating a validator public/private key pair

In this step, we generate a validator private/public key pair. The public key will be used to identify our validator on the network, whereas the private key is used to sign proposed blocks and for attestation purposes.

We create the validator account by issuing the following command through the terminal:

```
✓  prysm ./prysm.sh validator accounts create
```

This command will produce an output similar to that in the following snippet. Note that we have removed unnecessary lines from the output and are only showing important elements for brevity:

```
Latest Prysm version is v1.0.0-alpha.13.

 [2020-07-07 20:00:18]  INFO accounts: Please specify the keystore
path for your private keys (default:
"/Users/drequinox/Library/Eth2Validators"):
.
.
 [2020-07-07 20:00:37]  INFO accounts: Account creation complete!
Copy and paste the raw deposit data shown below when issuing a
transaction into the ETH1.0 deposit contract to activate your
validator client

=======================Deposit Data=======================
```

```
0x228951180000000000000000000000000000000000000000000000000000000000
0008000000000000000000000000000000000000000000000000000000000000e0
000000000000000000000000000000000000000000000000000000012050c1a
9e92704848cc263a94bbcdf12e598798fe61eb8a482f589daafcd5031440000000000
0000000000000000000000000000000000000000000000000308a289d558ceecef
01d31792dfa136f2789e0026c10f6e9a5fdabf663c7ec85e704bdb60a1e284055288e
0fb99a6aa8df0000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000200067f0d123615dccf51fec37826ed4
d8470649ca2447780af26a5a8c797b062100000000000000000000000000000000000000
0000000000000000000000000060b827457640fac2482bfabf8c864e0508bcda5014
7e033c173661f9819f598783497c5332aec02c2011290db473017b5400ed20fadffcd
bd6983d2bb47eb4f599d317cebe76e756e90fee705e02b0fb17bb2d85ee13b6706e71
b889e6320ce880

================================================================
```

This output has several important elements. First, the command creates the public/private key pair after asking for the password. Once the password is supplied, it creates the keystore containing the keys. Next, it creates the deposit data, which is used in the transaction sent to the deposit contract to activate the validator client. We will send this to the deposit contract. Finally, it displays the public key of validator: `0x8a289d558ceecef01d31792dfa136f2789e0026c10f6e9a5fdabf663c7ec85e704bdb60a1e284055288e0fb99a6aa8df`. We can use this key to query the status of our validator on the beacon chain block explorer. Fundamentally, this public key is used as an identity on the network for our validator client.

After this, we start the beacon chain client, which will synchronize the beacon chain.

Starting the beacon chain client

The beacon chain client can be started using the following command:

```
✓  prysm ./prysm.sh beacon-chain
```

This will show an output similar to the following. Note that we are only showing the most important lines for brevity:

```
Latest Prysm version is v1.0.0-alpha.13.
 [2020-07-07 20:03:54]  INFO node: Starting beacon node
version=Prysm/v1.0.0-alpha.13/a9bbbae19abac0c3c8cf7c4213b949172f97cf57. Built at:
2020-06-
29 04:54:41+00:00
[2020-07-07 20:03:54]  INFO gateway: Starting JSON-HTTP API
address=127.0.0.1:3500
[2020-07-07 20:03:54]  INFO rpc: RPC-API listening on port
address=127.0.0.1:4000
 [2020-07-07 20:04:25]  INFO initial-sync: Processing block
0x79bb6a3e... 167937/169735 - estimated time remaining 9h59m20s
blocksPerSecond=0.1 peers=3
```

This output has several important elements. It shows that when the beacon node starts up it performs several actions, such as starting up the required RPC and P2P services. Also, it connects to other peers and starts synchronizing blocks received from other nodes.

Initially, don't worry about the estimated time remaining. It will soon decrease to a reasonable number.

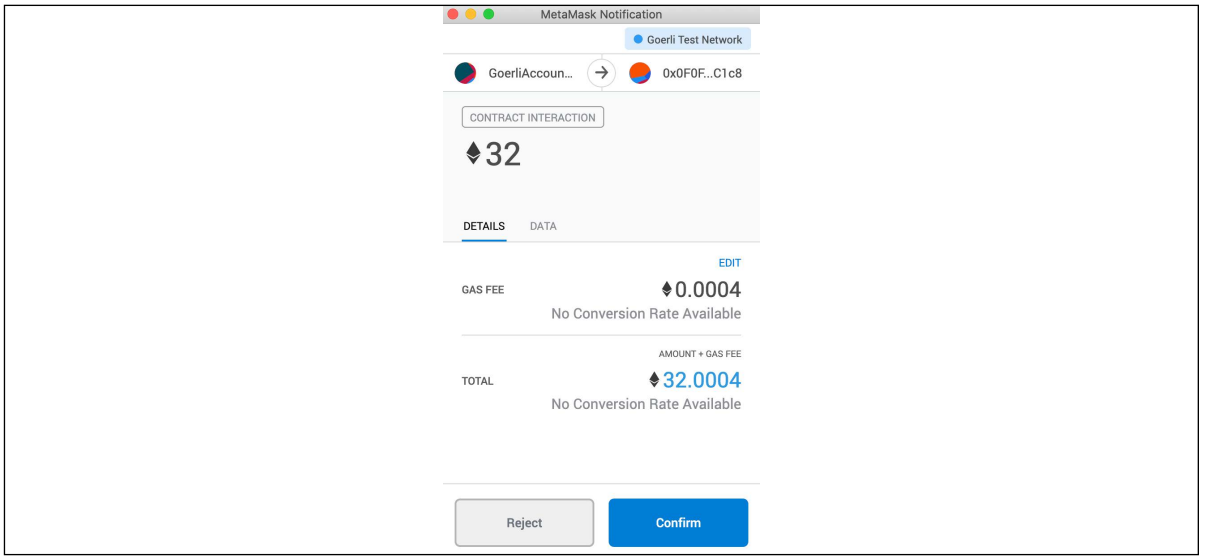
Note that the data directory where the chain will be stored locally will be under the directory at `/Users/drequinox/Library/Eth2/beaconchaindata`. A file named `beaconchain.db` will be created and used for persisting beacon chain data locally. After complete beacon chain synchronization, the size of this file will be around 2 GB in total. This is expected to increase with time, and on production, it might be even more.

In the next section, we will initiate the deposit to the deposit contract as part of the validator activation process.

Sending a deposit of ether to the deposit contract

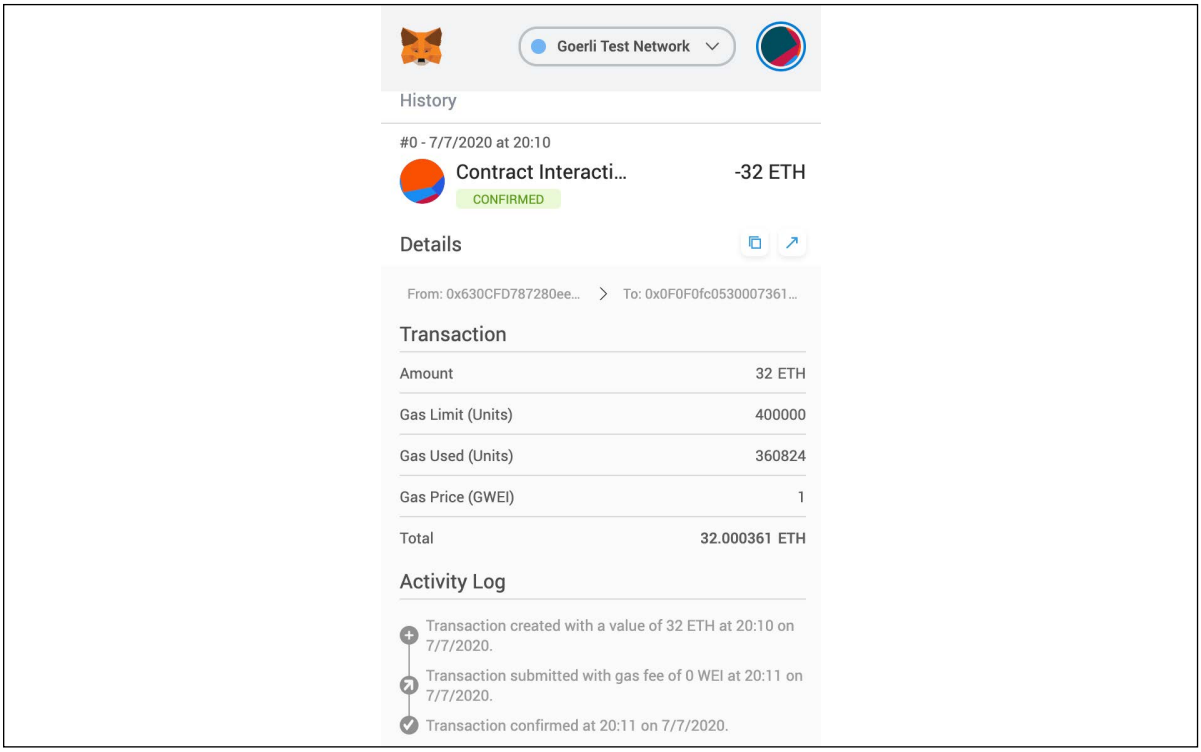
In this section, we will make a deposit of 32 ether to the deposit contract in order to activate our validator. The validator client listens for "deposit events" from the deposit contract on Ethereum 1.0 (in this case, the Ethereum Goerli testnet chain). As soon as it receives the event it will be activated, and according to the rules enforced by the beacon chain, will able to perform the proposal and attestation activities.

We can make the deposit via the Prysmatic Eth2 testnet. Simply browse to `https://prylabs.net/participate` and in the list of steps shown to make deposit, go to *step 2*, **Get GöETH—Test ether**, and choose **MetaMask** as the Web3 provider. **MetaMask** will ask for confirmation, as shown in the following screenshot:



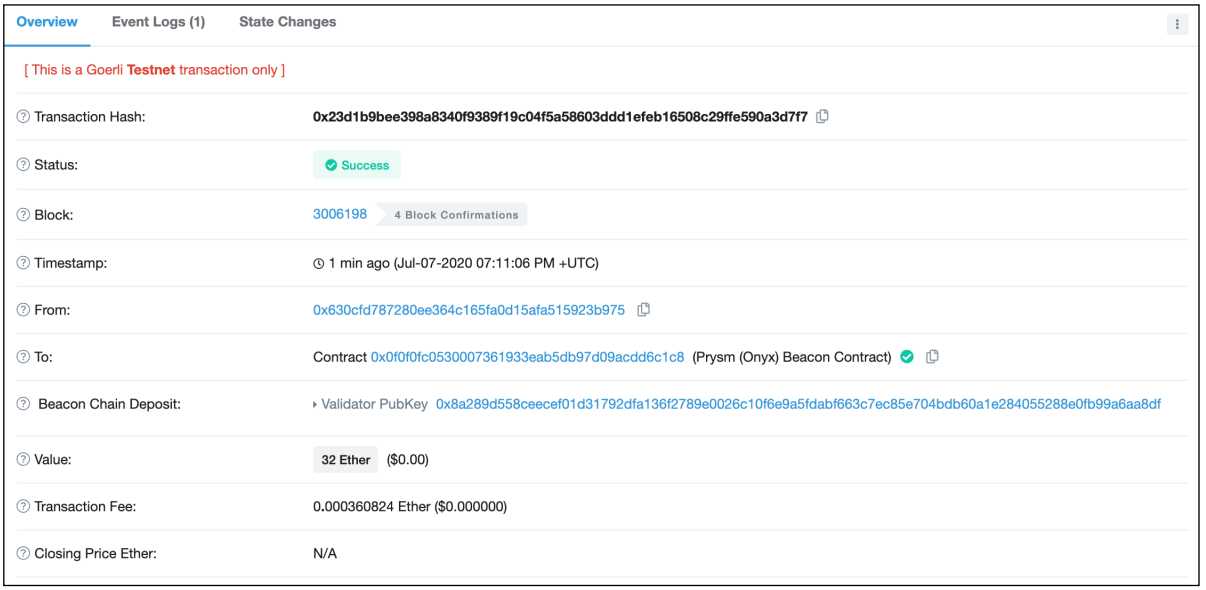
MetaMask notification

Once the transaction is confirmed, we can also view the transaction details in the history as follows:



MetaMask transaction history

The transaction can be viewed here at the Etherscan too, as shown in the following screenshot. We can use the transaction hash `0x23d1b9bee398a8340f9389f19c04f5a58603ddd1efeb16508c29ffe590a3d7f7`, which can be accessed at `https://goerli.etherscan.io/tx/0x23d1b9bee398a8340f9389f19c04f5a58603ddd1efeb16508c29ffe590a3d7f7`:

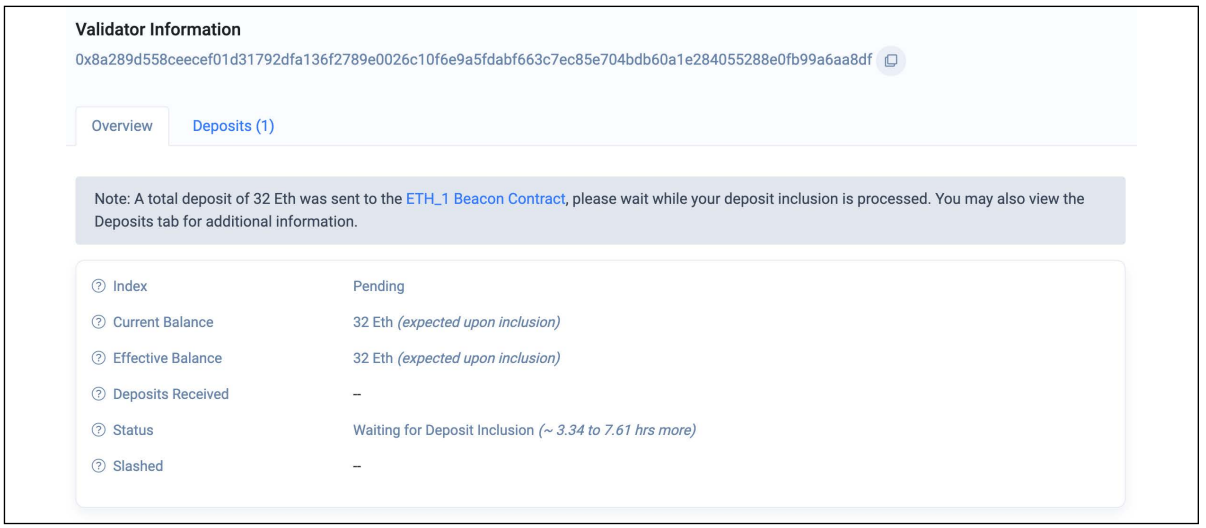


Viewing an Ethereum transaction on Goerli Etherscan

With the beacon chain explorer at the following URL, we can see the validator information:

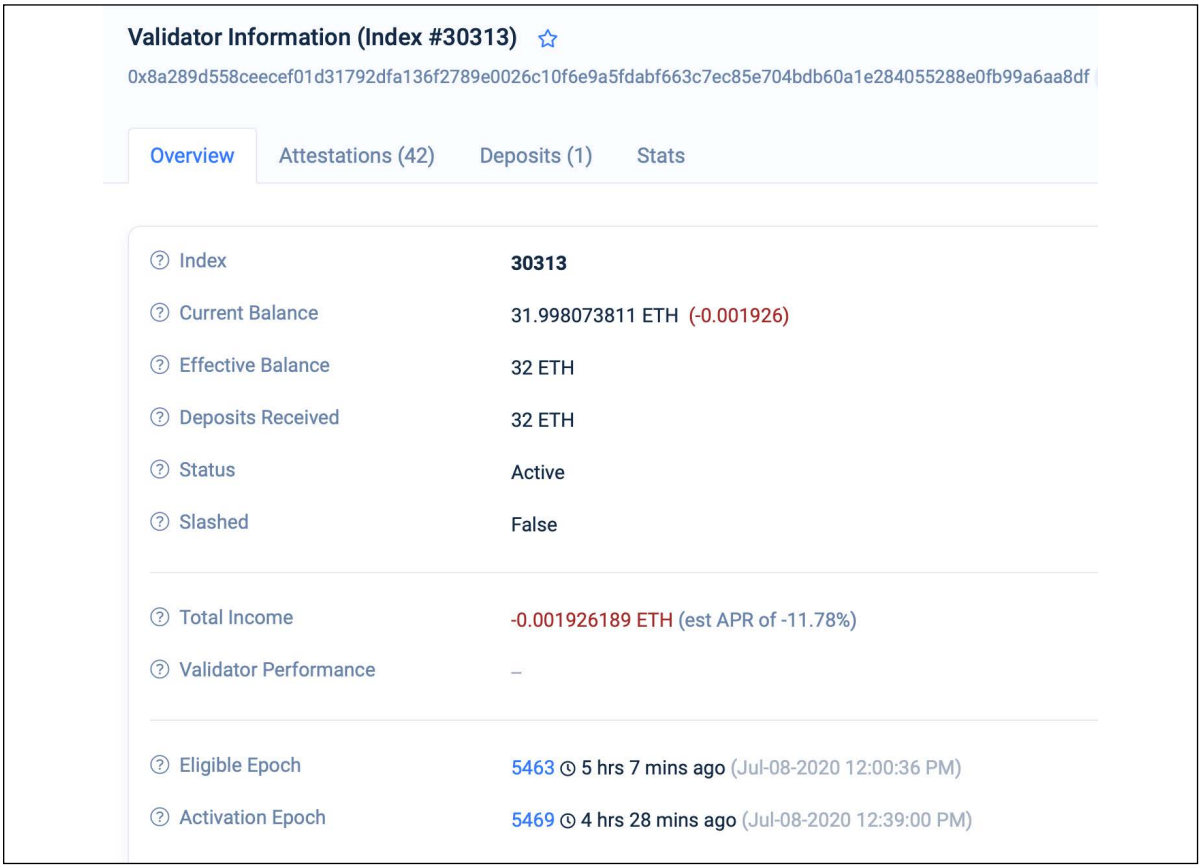
```
https://beaconscan.com/onyx/validator/0x8a289d558ceecef01d31792dfa136f2789e0026c10f6e9a5fdabf663c7ec85e704bdb60a1e284055288e0fb99a6aa8df
```

This can also be viewed in the following screenshot:



Transaction validator information

After a few hours, when the deposit has been processed, we can see the validator information with the current balance, status, and some other statistics indicating the success of the activation process:



Validator information view

Now that we have sent an ether deposit to the deposit contract, we can move on to starting up the validator client.

Starting the validator client

Note that the validator client should be started after the beacon chain synchronization is fully complete. Once the beacon chain client is synchronized, the validator client can be started using the following command:

```
✓  prysm ./prysm.sh validator
```

This will produce the output shown in the following snippet. Note that we have not shown the full output for brevity and have only included the most important elements of the output:

```
Latest Prysm version is v1.0.0-alpha.13.
.
.
[2020-07-07 20:06:29]  INFO keymanager: Checking validator keys
keystorePath=/Users/drequinox/Library/Eth2Validators
```

```
[2020-07-07 20:06:39]  INFO validator: Waiting for beacon chain start
log from the ETH 1.0 deposit contract
[2020-07-07 20:06:39]  INFO validator: Beacon chain started
genesisTime=2020-06-14 06:17:24 +0100 BST
```

Note that the same keystore will be used for starting up validators that we created earlier when generating the public/private key pair. Also, be aware of whether the beacon chain is fully synchronized or not, as if not, the validator node will wait until it is fully synchronized before proceeding further. In that case, the following message will appear with the preceding output:

```
INFO validator: Waiting for beacon node to sync to latest chain head
```

When the validator is active after the deposit is processed, a message similar to the following will appear in the output logs, indicating that the deposit has been accepted and the validator is now active. The attestation action can also be observed in the logs:

```
INFO validator: Validator activated index=30313
publicKey=0x8a289d558cee

INFO validator: Attestation schedule attesters=1
pubKeys=[0x8a289d558cee] slot=176413

INFO validator: Submitted new attestations AggregatorIndices=[]
AttesterIndices=[30313] BeaconBlockRoot=0x2bac6a8bd897
CommitteeIndex=5 Slot=176413 SourceEpoch=5511
SourceRoot=0x42377e74b49e TargetEpoch=5512 TargetRoot=0xc3b230b1832f
```

When the validator starts up, as shown in the preceding output, it validates the public key, performs several other checks, and starts up. If the deposit has not been observed by the beacon node yet, then it will wait, otherwise it will start to perform attestations as per the beacon chain rules.

Waiting for validator assignment

Once the validator assignment is complete, the validator will start performing its functions, such as block production and attestation, immediately.

We can see validator deposit inclusion data and other relevant details on Beaconscan by using the public key (`0x8a289d558ceecef01d31792dfa136f2789e0026c10f6e9a5fdabf663c7ec85e704bdb60a1e284055288e0fb99a6aa8df`) of our validator. This information can be accessed here:

`https://beaconscan.com/onyx/validator/0x8a289d558ceecef01d31792dfa136f2789e0026c10f6e9a5fdabf663c7ec85e704bdb60a1e284055288e0fb99a6aa8df`

Now we have our Ethereum 2.0 setup running with a validator client and beacon client.

As with any network, it is desirable to monitor the status of the nodes on a blockchain. For this purpose we use eth2stats. We'll see in the next section how eth2stats can be set up and how we can add our beacon chain client to the dashboard.

Monitoring using eth2stats

Monitoring is a desirable but optional step. We can monitor the beacon chain clients using eth2stats, which is available on GitHub at `https://github.com/Alethio/eth2stats-client`.

In order to run our example, we need to download eth2stats from GitHub, build it, and run it with the specific configuration of our setup. We'll see next how this is done.

Running eth2stats

First, we clone it from GitHub using the following command:

```
$ git clone https://github.com/Alethio/eth2stats-client.git
```

This will display the following output, indicating that the code is being downloaded. Note that the full output is not shown for brevity:

```
Cloning into 'eth2stats-client'...
remote: Enumerating objects: 196, done.
.
Resolving deltas: 100% (250/250), done.
```

After the code is downloaded, it will be available in the `eth2stats-client` directory.

The next step is to build the `eth2stats-client` instance. Remember that we need Golang installed for this, but Golang should be available now as we installed it as part of our pre-requisites.

Building the eth2stats-client instance

First, change the directory to `eth2stats-client`:

```
$ cd eth2stats-client
```

Once inside the directory, in order to build the `eth2stats-client`, we issue the following command:

```
$ eth2stats-client git:(master) make build
```

This will show a long output. We have only included the first few lines in the following snippet. If there are no errors reported then it is an indication that the build was successful:

```
go build -ldflags "-X main.buildVersion="v0.0.11+a1cc94b""
go: downloading github.com/gin-gonic/gin v1.5.0
```

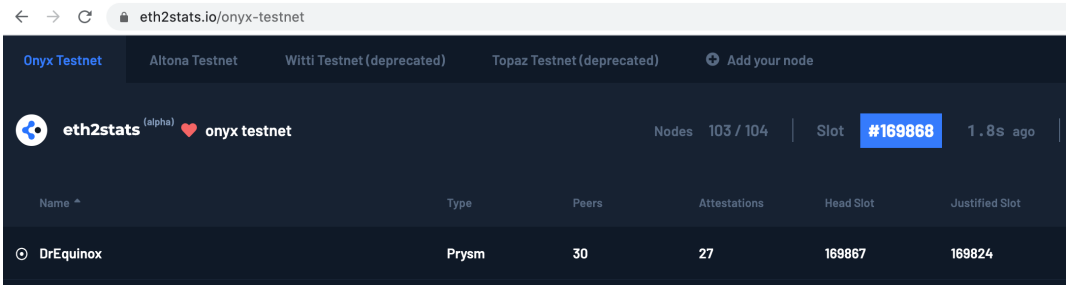
Once the build is complete, we can run `eth2stats-client` by using the following command:

```
$ eth2stats-client git:(master) ./eth2stats-client run --
eth2stats.node-name="DrEquinox" --data.folder ~/.eth2stats/data --
eth2stats.addr="grpc.onyx.eth2stats.io:443" --eth2stats.tls=true --
beacon.type="prysm" --beacon.addr="localhost:4000" --beacon.metrics-
addr="http://localhost:8080/metrics"
```

The command will produce an output similar to the one shown in the following snippet. Note that the output has been truncated for brevity:

```
INFO[0000] [prysm] setting up beacon client connection
INFO[0000] [core] setting up eth2stats server connection
INFO[0000] [core] got beacon client version              version="Prysm/
v1.0.0-alpha.13/a9bbbae19abac0c3c8cf7c4213b949172f97cf57. Built at: 2020-06-29
04:54:41+00:00"
INFO[0000] [core] getting beacon client genesis time
INFO[0000] [core] successfully connected to eth2stats server
INFO[0000] [core] setting up chain heads subscription
INFO[0000] [prysm] listening on stream
```

When `eth2stats-client` is up, we can see our node listed on the dashboard here: `https://eth2stats.io/onyx-testnet`. This is also shown in the following screenshot:



Eth2stats—onyx testnet

When the command runs successfully, the node with its name will be visible on the eth2stats website for the **onyx testnet**. As shown in the preceding screenshot, a node named `DrEquinox` is visible on the eth2stats web page.

With this, we have created a complete Ethereum 2.0 setup with a beacon node, a validator node, and the option to monitor it on eth2stats.

Chapter 22

Alternative blockchains

This is an introduction to alternative blockchain solutions. With the success of Bitcoin and the subsequent realization of the potential of blockchain technology, something of a Cambrian explosion happened in the tech world, which resulted in the development of various blockchain protocols, applications, and platforms. Some projects did not gain much traction, but many have succeeded in creating a stable place in this space.

Here, we'll introduce alternative blockchains. The success of Ethereum and Bitcoin has resulted in various projects that have been spawned into existence by leveraging the underlying technologies and concepts these earlier blockchains introduced. These new projects add value by addressing the limitations in the traditional blockchains and also by enhancing the existing solutions by providing developer tools, web interfaces, and deployment tools. In additional to better supporting tools, many blockchains have emerged that aim to address the limitations in the original protocols introduced by Bitcoin and Ethereum.

In this section, an introduction to alternative blockchain solutions will be given. For example, Kadena is a new private blockchain with novel ideas, such as scalable **Byzantine Fault Tolerance** (**BFT**). Various concepts, such as **sidechains**, **drivechains**, and **pegging** have also been introduced with this growth of blockchain technologies. This section will cover all these technologies and related concepts in detail. Of course, it's not possible to cover all **alternative chains** (**altchains**), but the platforms that have been included here are related to blockchains, were covered in insufficient detail in the core book chapters, or are expected to gain traction soon.

We will explore Kadena, EOS, and Tezos. We will also introduce blockchain-based payment networks such as Ripple and Stellar. In addition, we will also look at some other notable projects, such as BigChainDB, Storj, and MultiChain.

Kadena

Kadena is a blockchain that has successfully addressed scalability and privacy issues in blockchain systems. A new Turing incomplete language, called **Pact**, has also been introduced with Kadena that allows the development of smart contracts. A key innovation in Kadena is its scalable BFT consensus algorithm, which has the potential to scale to thousands of nodes without performance degradation.

Scalable BFT is based on the original **Raft** algorithm and is a successor of Tangaroa and Juno. Tangaroa, which is a name given to an implementation of Raft with BFT, was developed to address the availability and safety issues that arose from the behavior of **Byzantine nodes** in the Raft algorithm, and Juno was a fork of Tangaroa.

Consensus algorithms were discussed in *Chapter 5*, *Consensus Algorithms*, in more detail.

Both Tangaroa and Juno have a fundamental limitation—they cannot scale. As such, Juno and Tangaroa could not gain much traction. Tangaroa could scale up to 50 nodes with a processing speed of around 5,000 transactions per second, but it was only a **Proof of Concept** (**PoC**) and later it was discovered that it has some safety and liveness issues. Blockchains have the more desirable property of maintaining high performance as the number of nodes increases, but the aforementioned proposals lack this feature. Kadena solves this issue with its proprietary **scalable BFT** algorithm, which, according to the official documentation on Kadena, scales up to thousands of nodes without any performance degradation.

Moreover, confidentiality is another significant aspect of Kadena, and it enables privacy of transactions on the blockchain. This security service is achieved by using a combination of key rotation, symmetric on-chain encryption, incremental hashing, and the **Double Ratchet** protocol:

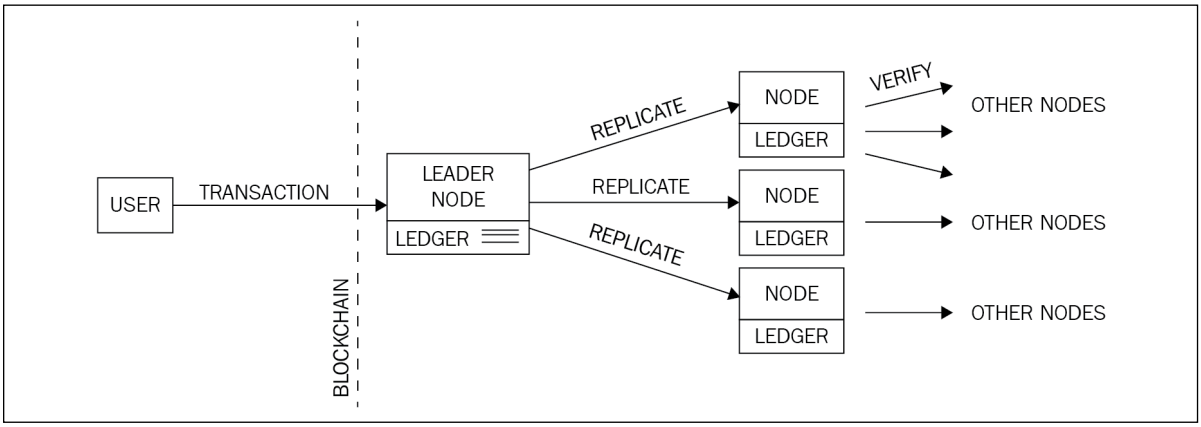
- Key rotation is used as a standard mechanism to ensure the security of the private blockchain. It is used as a best practice to thwart any attacks if the keys have been compromised by periodically changing the encryption keys. There is native support for key rotation in Pact smart contract language.
- Symmetric on-chain encryption allows the encryption of transaction data on the blockchain. These transactions can be automatically decrypted by the participants of a particular private transaction.
- Incremental hash functions are useful in situations where, if a message previously hashed changes slightly into a new message, then instead of recomputing a new hash from scratch again, the hash generated originally for the message is used to generate a new hash. This method is faster than generating a new hash altogether for the new message.
- The Double Ratchet protocol is used to provide key management and encryption functions.

The scalable BFT consensus protocol ensures that adequate replication and consensus has been achieved before smart contract execution. The consensus is achieved with the following process, which is how a transaction originates and flows in the network:

1. First, a new transaction is signed by the user and broadcast over the blockchain network, which is picked up by a leader node that adds it to its immutable log. At this point, an incremental hash is also calculated for the log. An incremental hash is a type of hash function that allows the computation of hash messages in the scenario where, if a previous original message which is already hashed is slightly changed, then the new hash message is computed from the already existing hash. This scheme is quicker and less resource-intensive compared to a conventional hash function, where an altogether new hash message is required to be generated even if the original message has only changed very slightly.

2. Once the transaction is written to the log by the leader node, it signs the replication and incremental hash and broadcasts it to other nodes.
3. Other nodes, after receiving the transaction, verify the signature of the leader node, add the transaction into their own logs, and broadcast their own calculated incremental hashes (quorum proofs) to other nodes. Finally, the transaction is committed to the ledger permanently after an adequate number of proofs are received from other nodes.

A simplified version of this process is shown in the following diagram, where the leader node is recording the new transactions and then replicating them to the follower nodes:



Consensus mechanism in Kadena

Once consensus is achieved, a smart contract execution can start and takes a number of steps, as follows:

1. The signature of the message is verified.
2. The Pact smart contract layer takes over.
3. The Pact code is compiled.
4. The transaction is initiated and executes any business logic embedded within the smart contract. In case of any failures, an immediate rollback is initiated that reverts that state back to what it was before the execution started.
5. Finally, the transaction completes and the relevant logs are updated:

Pact has been open sourced by Kadena and is available for download at `http://kadena.io/pact/downloads.html`.

Next, let's move on to an examination of Kadena's main programming language, Pact.

Pact

Pact can be downloaded as a standalone binary that provides a REPL for the language. An example is shown here, where Pact is run by issuing the `pact` command in the Linux console:

```
drequinox@drequinox-OP7010:~/Downloads$ ./pact
pact> 1234
1234
pact> (+ 1 2)
3
pact> (if (= (+ 1 2) 3 "OK" "ERROR")
(interactive):1:31: error: unexpected
    EOF, expected: ")", ";", "{",
    Boolean false, Boolean true,
    Decimal literal, Integer literal,
    String literal, Symbol literal,
    list literal, pact, sexp, space
(if (= (+ 1 2) 3 "OK" "ERROR")<EOF>
                              ^
pact> (if (= (+ 1 2) 3) "OK" "ERROR")
"OK"
pact>
```

Pact REPL, showing sample commands and error output

A smart contract in the Pact language is usually composed of three sections: keysets, modules, and tables. These sections are described here:

- **Keysets**: This section defines relevant authorization schemes for modules and tables.
- **Modules**: This section defines the smart contract code encompassing the business logic in the form of functions and pacts. Pacts within modules are composed of multiple steps and are executed sequentially.
- **Tables**: This section is an access-controlled construct defined within modules. Only administrators defined in the admin keyset have direct access to this table. Code within the module is granted full access, by default, to the tables.

Pact also allows several execution modes. These modes include contract definition, transaction execution, and querying. These execution modes are described here:

- **Contract definition**: This mode allows a contract to be created on the blockchain via a single transaction message.
- **Transaction execution**: This mode entails the execution of modules of smart contract code that represent business logic.

- **Querying**: This mode is concerned with simply probing the contract for data and is executed locally on the nodes for performance reason. Pact uses LISP-like syntax and represents in the code exactly what will be executed on the blockchain, as it is stored on the blockchain in human-readable format. This is in contrast to Ethereum's EVM, which compiles into bytecode for execution, which makes it difficult to verify what code is in execution on the blockchain. Moreover, it is Turing incomplete, supports immutable variables, and does not allow null values, which improves the overall safety of the transaction code execution.

It is not possible to cover the complete syntax and functions of Pact in this short introduction; however, a small example is shown here that shows the general structure of a smart contract written in Pact. This example shows a simple addition module that defines a function named `addition` that takes three parameters. When the code is executed, it adds all three values and displays the result:

The following example has been developed using the online Pact compiler available at `https://pact.kadena.io/`.

```
"Define keyset"
(define-keyset 'admin-keyset (read-keyset "admin-keyset"))
"Define module"
(module addition 'admin-keyset
"Define function addition that takes 3 argument of type integer"
  (defun addition ( x y z : integer)
"Run format"
    (format "Result : {}" [( + x ( + y z ) )])
)
)
"Run addition function with three arbitrary numbers to add"
(addition 432 4562 87)
```

Sample Pact code

When the code is run, it produces the output shown as follows:

The output of the code

As shown in the preceding example, the execution output exactly matches the code layout and structure, which allows greater transparency and limits the possibility of malicious code execution.

Kadena is a new class of blockchains introducing the novel concept of **pervasive determinism** where, in addition to standard public/private key-based data origin security, an additional layer of fully deterministic consensus is also provided. It provides cryptographic security at all layers of the blockchain, including transactions and the consensus layer.

Documentation and source code for Pact can be found at `https://github.com/kadena-io/pact`.

Kadena also introduced a public blockchain in January, 2018, which is another leap forward in building blockchains with massive throughput. The novel idea in this proposal is to build a **Proof of Work** (**PoW**) parallel chain architecture. This scheme works by combining individually mined chains on peers into a single network. The result is massive throughput capable of processing more than 10,000 transactions per second.

The original research paper is available at `http://kadena.io/docs/chainweb-v15.pdf`.

In the next section we introduce EOS, which aims to be a scalable blockchain and has introduced several innovative ideas.

EOS

The **EOSIO**, also known as EOS, blockchain is developed by block.one (`https://block.one/`) in the C++ programming language. It was first released on January 31, 2018. It has been built for both public and private blockchain use cases.

A new concept of system resources has been introduced with EOS. Just like the usual concept of computing resources in a computer, such as RAM, CPU, and Networking (NET), the EOS blockchain uses the same concept of resources to manage the blockchain. The amount of resources allocated is directly proportional to the amount of EOS stake—meaning the higher the stake, the more the resources you get. This resource model protects against abuse because in order to game the system, an attacker would need a large amount of stake.

Resources

During the process of staking, users (stakers) specify how much of the stake is going to be allocated for CPU and NET. RAM is not allocated as a result of staking, but is required to be bought separately from the RAM market.

We describe these resources one by one in the next sections.

CPU

CPU is required for executing transactions. It is the processing power of an account. It represents the processing time of an action in microseconds. It is referred to as **CPU bandwidth**.

RAM

This is used for data storage on the network. It is required by nodes to store account data in the blockchain state. It is required to be purchased by developers to run applications. Data such as the name of the account, relevant metadata, permissions, token balance, and public keys is stored in the RAM. It can be thought of as hard disk space. RAM can also be traded with other accounts for a fee if it's no longer required by an account.

NET

NET is the network bandwidth measured in bytes. This is the amount of bandwidth a user is allowed to use.

CPU and NET combined are also commonly referred to as bandwidth.

Components

EOS consists of a number of components, which we'll describe in the following sections.

nodeos

This is the core element that runs as a **daemon** on every EOS.IO blockchain node. It runs a blockchain node, which can also be configured with plugins for additional functionality.

It handles the blockchain persistence layer, P2P networking, and smart contract code scheduling.

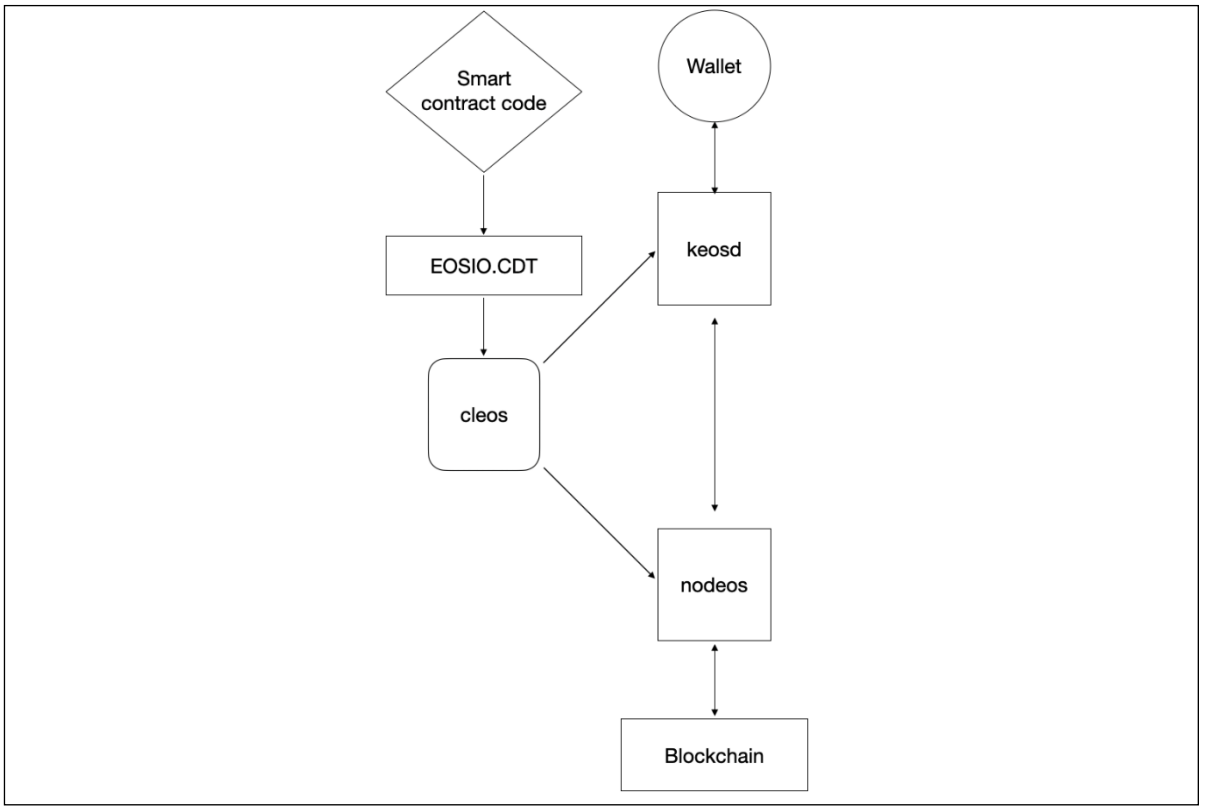
cleos

This is the primary command-line tool that is used to interact with the nodeos. Interaction is done via REST APIs exposed by `nodeos`. **cleos** is also is used to test and deploy smart contracts.

keosd

This is the key management daemon responsible for storing private keys and signing messages.

We can visualize this architecture in the following diagram:



EOS high-level architecture

Other components of EOS blockchain include accounts, transactions and actions, and wallets, which we will introduce next.

Accounts

Accounts in EOS are represented by strings of 12 characters. They act as identifiers on the blockchain. An EOS account also has permissions associated with it that define which actions are allowed to be performed by the account, and are required for actions such as staking, voting, and sending or receiving funds. Accounts are controlled by cryptographic key pairs (a pair of private and public key). An EOS account name can only contain a-z letters in lowercase, a period (.), and numbers 1-5, and it must start with a letter.

Transactions, actions, and blocks

A **transaction** can be defined as an atomic change to the blockchain, usually as a result of smart contract execution. Transactions make up the bodies of the **blocks**. A block is composed of a **block header** and transactions. One or more actions make up a transaction. An action can be described as a suggested change in the blockchain or a call to a smart contract. There are three main types of actions in EOSIO, namely, *calling actions*, *inline actions*, and *deferred actions*. Calling actions are calls made by users to a contract, inline actions are calls made between different contracts or within the same contract, and deferred actions represent deferred transactions.

Wallet

A wallet is an encrypted file created by a client such as `cleos`. It manages the private keys and transaction signing. A wallet file is unlocked (decrypted) by using a master key. It can be in a locked or unlocked state.

EOS supports the development of smart contracts and provides libraries for developers to use. We explore them in the next section.

Developing with EOS

Development on EOS is made possible by several tools and libraries, which we will describe in this section.

EOSIO.CDT

This is the contract development toolkit. It is a set of tools that enables smart contract development on EOS. It is a toolchain for creating **web assembly** (**Wasm**). It contains several elements, such as:

- `eosio-cpp`: Used for compiling contract source code
- `eos-ld`: The web assembly linker for smart contracts
- `eosio-init`: Generates the skeleton code and directory structure for a smart contract
- `eos-abidiff`: Compares two ABI files and outputs the difference

EOSJS

This is the general-purpose JavaScript library for the EOS blockchain. It is available at `https://github.com/EOSIO/eosjs`.

Now, let's see how to install EOS and perform some basic experiments.

Installation

On macOS, EOS can be installed by using the following command:

```
$ brew tap eosio/eosio
```

This command will give a large output. Eventually, it will give an output similar to the following, indicating the success of the process:

```
Tapped 2 formulae (29 files, 98.7KB).
```

After this step, installation can commence. Issue the following command to install `eosio`:

```
$ brew install eosio
```

A message similar to the following will appear, indicating the success of the installation of EOS:

```
==> Installing eosio/eosio/eosio
==> Pouring eosio-2.0.5.mojave.bottle.tar.gz
  /usr/local/Cellar/eosio/2.0.5: 19 files, 59.4MB
```

With this, we have installed `eosio`, consisting of `nodeos`, `cleos`, and `keosd`.

EOS can also be built from source, if required. Instructions to do so are available here: `https://developers.eos.io/manuals/eos/v2.0/install/build-from-source/index`

We've used macOS in this example to install EOS. However, a comprehensive list of installation methods for different operating systems is available here: `https://developers.eos.io/manuals/eos/v2.0/install/install-prebuilt-binaries`. Readers can refer to this documentation if using a different operating system.

Development

Next, we install, contract development toolkit, known as EOSIO.CDT. This component allows the development of smart contracts for EOSIO. It consists of a toolchain for Wasm and other tools for smart contract development for the EOSIO blockchain.

The CDT supports the development of **Ricardian contracts**.

Refer to *Chapter 10, Smart Contracts*, for a refresher on Ricardian contracts.

It introduces the Ricardian template toolkit, which is a set of libraries that facilitates the writing of Ricardian contracts. There's also the Ricardian specification, which serves as the working specification of the Ricardian template toolkit.

Contract development supports C++, which makes it easy for developers as C++ is a mainstream language.

To install EOSIO.CDT, the contract development toolkit, on macOSwe use `brew`:

```
$ brew tap eosio/eosio.cdt
```

We'll get the following output:

```
Updating Homebrew...
==> Tapping eosio/eosio.cdt
.
.
.
Tapped 1 formula (66 files, 44.5KB).
```

Now we'll install EOSIO.CDT:

```
$ brew install eosio.cdt
```

We'll get the following output:

```
==> Pouring eosio.cdt-1.7.0.mojave.bottle.tar.gz
  /usr/local/Cellar/eosio.cdt/1.7.0: 3,989 files, 354.8MB
```

To confirm that the installation has been successful, run the following command:

```
$ eosio-cpp -version
```

We'll get the following output:

```
eosio-cpp version 1.7.0
```

This output indicates that installation has been successful and shows the installed version. With this, we have installed the EOSIO contract development toolkit. More information on EOSIO.CDT is available at: `https://github.com/EOSIO/eosio.cdt`.

In the next section, we'll compare the relative strengths and weaknesses of Ethereum and EOS.

Comparing Ethereum and EOS

EOSIO can be thought of as a decentralized operating system, whereas Ethereum is considered to be a decentralized world computer. From the cryptocurrency supply point of view, there is no limit on the amount of ether that can be generated in Ethereum, whereas in EOS, the total supply is capped at 1 billion EOS, with inflation of 5% per year. In Ethereum, the **Ethash** PoW mechanism is used as a consensus facilitation mechanism, whereas in EOS, **Delegated Proof of Stake** (**DPOS**) is used.

The throughput of EOS is significantly better, with thousands of **transactions per second** (**TPS**), compared to 10-15 TPS for Ethereum.

Ethereum and Bitcoin utilize PoW as their consensus mechanism. This is an inherently slow mechanism because it maintains the block production frequency at a specific interval. For example, in Bitcoin it is around every 10 minutes, and in Ethereum it's around 14 seconds. Also, the requirement to solve hash puzzles to prove that a certain amount of resources has been spent naturally slows down the block production speed because solving the puzzles takes time.

Block time, or the time required to mine a block, is 10-15 seconds in Ethereum, whereas in EOS it is 0.5 seconds. The Solidity language is used for smart contract development in Ethereum, whereas C++ is used in EOS.

A comparison between EOS and Ethereum is provided in the following table, which covers different aspects of a blockchain.

Attribute	**Ethereum**	**EOS**
Block time	10-15 seconds	0.5 seconds
Smart contract development language	Solidity	C++ and others via Wasm
Total supply of cryptocurrency	Unlimited	Capped at 1 billion
Account names	String of non-user-friendly characters	User-friendly name
Data persistence layer	Key value store	Relational database
Smart contract code upgradeability	Not upgradeable, immutable code	Upgradeable
Consensus mechanism	PoW Ethash	DPOS + aBFT
Performance	10-15 TPS	1000s TPS
ICO fundraiser	18.4 million USD	4 billion USD
Fees	Gas-based mechanism that charges per operation—rent-based model	No user fee. A stake in the blockchain allows users to use resources on the blockchain—ownership-based model
Permissioning	Role-based permissioning	No permissioning mechanism
Currency	ETH	EOS

There are some further resources that can be explored. This section merely touched the basics of EOS blockchain: EOS is a vast enough system that you could barely cover the subject with a whole book.

An EOS technical white paper is available at `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`.

The GitHub repository is available at `https://github.com/EOSIO/eos`.

The official EOS website is at `https://eos.io`.

There are a number of block explorers available for EOS. A selected list is provided here:

- `https://eostracker.io`
- `https://bloks.io`
- `https://www.eosx.io`

In the next section, we introduce Tezos, which made headlines in 2017 with their ICO of 232 million USD.

Tezos

Tezos is a generic self-amending cryptographic ledger, which means that it not only allows decentralized consensus on the state of the blockchain but also allows consensus on how the protocol and nodes are evolved over time. Tezos has been developed to address limitations in the Bitcoin protocol (and other similar blockchains) such as contentious issues arising from hard forks, cost, and mining power centralization due to PoW, limited scripting ability, and security issues. It has been developed in a purely functional language called **OCaml**.

The white paper is available at `https://tezos.com/static/papers/white_paper.pdf`.

The position paper is available at `https://tezos.com/static/papers/position_paper.pdf`.

Source code is available at `https://gitlab.com/tezos/tezos`.

The architecture of a Tezos distributed ledger is divided into three layers: the **network layer**, the **consensus layer**, and the **transaction layer**. This decomposition allows the protocol to evolve in a decentralized fashion. For this purpose, a generic network shell is implemented in Tezos that is responsible for maintaining the blockchain, which is represented by a combination of the consensus and transaction layers. This shell provides an interface layer between the network and the protocol.

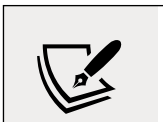
The concept of a **seed protocol** has also been introduced, which is used as a mechanism to allow stakeholders on the network to approve any changes to the protocol.

The Tezos blockchain starts from a seed protocol, in contrast with a traditional blockchain that starts from a genesis block.

This seed protocol is responsible for defining procedures for amendments in the blockchain and even the amendment protocol itself. The reward mechanism in Tezos is based on a **Proof of Stake** (**PoS**) algorithm, hence there is no mining requirement.

A domain-specific language called **Michelson** has been developed in Tezos for writing smart contracts, which is a stack-based Turing complete language. Smart contracts in Tezos are formally verifiable, which allows the code to be mathematically proven for its correctness.

Tezos completed crowdfunding via an ICO of 232 million USD in July 2017. Their public network was released in June 2018.



Tezos code is available at `https://gitlab.com/tezos/tezos`.

Tezos can be distinguished as a platform that is not only a smart contract platform like Ethereum, but it also has a built-in governance mechanism and supports formal verification of contract code. These two additional properties make it a different and arguably better protocol than existing traditional blockchain networks.

Due to the governance mechanism, there is no central control by either developers or miners on the blockchain, and formal verification allows the development of secure and formally verified smart contracts. By allowing formal verification, bugs can be avoided, which leads to better security. Tezos' cryptocurrency is called **tezzies** and is symbolized with the letters XTZ.

Now let's have a look at the Tezos architecture, and see the different elements that make up Tezos and how they fit together.

Architecture

Before discussing the components of a Tezos network, let's first see what a Tezos network is.

Network

The network is the underlying blockchain network using the **peer-to-peer** (**P2P**) protocol where all Tezos nodes exist and communicate with each other.

There are two test networks on Tezos:

- **Babylonnet**, which is the same as mainnet.
- **Zeronet**, which is used for development and future enhancements.

Tezos mainnet is the main production network used by Tezos. The mainnet block explorer of Tezos is available at `https://www.tezos.id`.

The Tezos network consists of several components, which we'll describe in the following sections.

Client

The client is a basic wallet and acts as a primary interface to the node. Clients or other third-party applications communicate with the node via RPC, which uses JSON format and the HTTP protocol.

Node

A node can be defined as a peer on the Tezos P2P network. It is an entity that is responsible for connecting with the Tezos blockchain. A node has a local state and consists of a shell and the protocol. The shell consists of a P2P network layer and validator. The P2P layer is responsible for communicating with other nodes via the **gossip protocol**. The **economic protocol** is the self-amending element that is responsible for different operations, such as transaction interpretation.

A node propagates blocks and operations such as transactions, accusation, activation, delegation, endorsement, and origination. Let's look at each of these terms:

- A **transaction** in Tezos can be defined as a transfer of funds between two accounts, or a smart contract execution.
- **Accusation** is the process whereby a node can accuse another node of deviating from the protocol or, in other words, abusing or misusing the network, such as injecting incompatible blocks into the blockchain.
- **Activation** is the process of claiming the tokens from the ICO sale and activating the address on the blockchain.
- **Delegation** is the mechanism by which a token holder delegates the rights of baking (baking is Tezos' term for validating transactions and blocks) to another party.
- **Endorsing** is a mechanism whereby a baker is asked to validate and witness a block to check that it has been created correctly and is a valid block. Endorsers are also rewarded with tezzies for their activity.
- **Origination** is the action (or operation) of creating a new smart contract.

All the aforementioned operations are actions that result in a state change in the blockchain.

Other than the peer nodes, there are other daemons, which can be endorsers, bakers, or accusers. We'll discuss these individually, as follows.

Endorser

This is a node on the Tezos blockchain network that has the endorsement right, which results in increasing the endorsed block's score. A block's score is a measure of its weight or fitness to be considered the head of the blockchain. It can be defined as a unit of comparing contexts. Context is simply defined as the state of the blockchain. If there are conflicting blocks, the context's score is measured and the highest scoring (or fittest) block becomes the head of the blockchain.

In relation to the context and operations, there is another concept called **economic protocol**, which has been introduced in Tezos. This is defined as the application that runs on top of the blockchain and defines its state (context) and actions that result in state changes (operations).

Baker

A node is a baker when its role is to add blocks in the Tezos blockchain. During the baking process, a baker picks up transactions accumulated in its memory pool that have been propagated on the network.

A set of consecutive blocks is called a cycle. A cycle represents blocks from a certain height to another height and is used as a perception of time on the blockchain. Another relevant concept is called a roll, which means an amount of tezzies as a unit, to establish a delegate's rights of baking in a cycle.

Baking in Tezos is the equivalent of mining in Bitcoin. In PoW blockchains such as Bitcoin or Ethereum, the right to add a new block to the blockchain is won by solving the PoW problem. In Tezos, this right is assigned randomly to a baker who owns tezzies. One key point to note is that it is not the baker that is selected randomly, it is the token that a baker holds. The tokens can also be delegated to someone else, and if a token out of those delegated tokens is randomly chosen for baking, then the delegated party will win the right to add a new block.

A baker is required to create a safety deposit if chosen as the next block creator, which ensures the honesty of the baker. If a baker tries to be dishonest, it would be punished and its safety deposit would be lost. If the baker honestly creates a new block that is accepted, it obtains tezzies as a reward.

A list of bakers can be found here: `https://mytezosbaker.com`

Accuser

An accuser node provides evidence to show that the accused node has attempted to perform some illegal activity. In return for this effort, the accuser is rewarded with some funds from the accused node's baking deposit. Illegal activities primarily include operations where a baker signs two different blocks at the same block height or when more than one endorsement operation is injected by an endorser within the same baking cycle.

Like any blockchain network, an account is required to perform transactions on a blockchain network. We introduce accounts in Tezos next.

Accounts

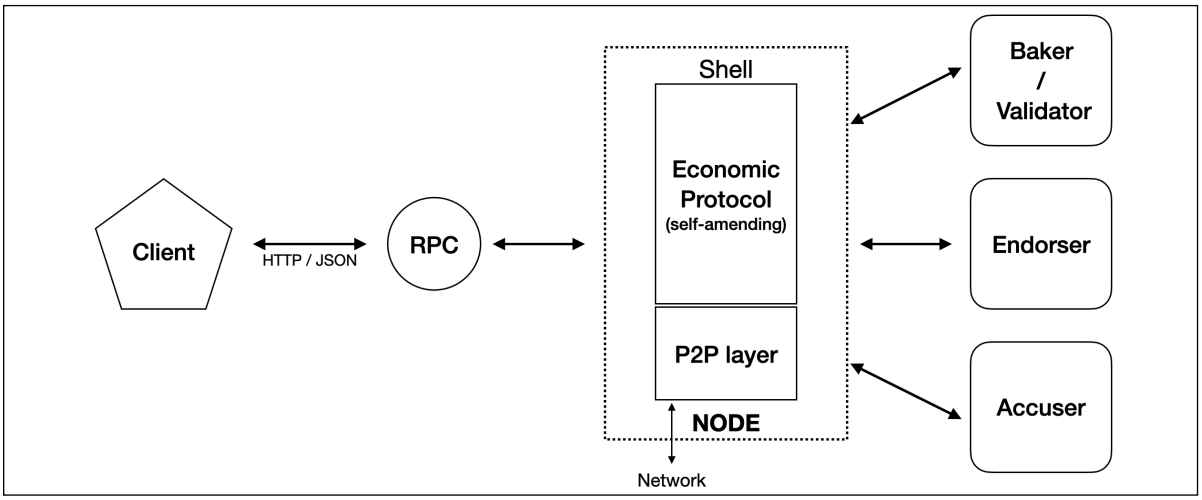
There are two types of accounts within Tezos, namely **implicit accounts**, which are identified by the prefix `tz1`, and **originated accounts**, which are identified by the `kt1` prefix.

Implicit accounts are created by using a public and private key pair. They have an account owner representing the owner of the private key and the account balance. The public address for these accounts is identified by the `tz{1,2,3}` prefix, is derived from the public key.

The other type of account is for smart contracts and are called originated accounts. These are identified by the prefix `kt1`. They are created by the origination operation from another contract. Originated accounts cannot act as a baker and can be spendable or delegatable. They are managed via an implicit account or another contract.

These accounts have four fields, namely, **manager**, **amount**, **delegatable**, and **delegate**. Manager contains the private key for the account. Amount, as the name suggests, holds the number of tezzies for this account. The delegatable field describes whether this account is capable of delegating baking operations or not. Finally, the delegate field identifies the delegated account for baking.

The high-level architecture of Tezos can be visualized as follows:



Tezos high-level architecture

Now that we understand the theoretical foundation, let's see what options are available for smart contract development in Tezos.

Development

Smart contracts in Tezos can be readily formally verified, thus increasing the reliability and security of the contracts. A domain-specific language called **Michelson** has been developed for writing smart contracts for Tezos. In contrast with Ethereum's Solidity, which needs to be compiled into bytecode first for it to be executed on the VM, Michelson can directly run normal, human-readable code on the Tezos VM. This approach helps with the formal verification of the smart contract code. Michelson is a stack-based and strongly typed language, which allows smart contracts written using Michelson to be formally verified.

Strongly typed languages have strictly defined restrictions imposed by the compiler of the language, which enforces certain rules around the data types. Usually these are restrictions on the automatic conversion of one data type to another, and variables are required to have a well-defined type. If these rules are violated, then exceptions are usually raised at compile time.

Smart contracts are identified by addresses starting with `KT1`. Smart contracts are created with an operation called origination, meaning contract registration on the blockchain network.

Some languages other than Michelson that have been developed for smart contract development for Tezos are as follows:

- **LIGO**: This is an easy-to-use smart contract language for Tezos. There are three flavors available for LIGO, namely PascaLIGO, CameLIGO and ReasonLIGO. More information can be found at `https://ligolang.org`.

- **Fi**: A high-level language for Michelson. More information is available at `https://fi-code.com`.
- **Liquidity**: A high-level typed smart contract language. More information is available at `https://www.liquidity-lang.org`.

There are also various libraries that can be used to integrate Tezos with an application. A selection of these libraries is as follows:

- **ConseilJS**: `https://cryptonomic.github.io/ConseilJS/#/`
- **Taquito**: `https://tezostaquito.io`
- **TezBridge**: `https://docs.tezbridge.com`
- **eztz**: `https://github.com/TezTech/eztz`

Wallets

There are quite a few wallets available for Tezos. Software wallets include Galleon (`https://galleon-wallet.tech/`), Air Gap (`https://airgap.it/`), Kukai (`https://kukai.app/`), and ZenGo (`https://zengo.com/`). Hardware wallets include Ledger (`https://www.ledger.com/`) and Trezor (`https://trezor.io/`).

Moreover, the Tezos client **command-line interface** (**CLI**) can also be used to support basic wallet functionality. Let's see how we can set up a Tezos client and perform some basic experiments.

Setting up a Tezos client

As Tezos is built using OCaml, we need the OPAM utility to build the Tezos code. In this example, we'll download and compile Tezos code on macOS:

1. First, we run the following command to download and install the OPAM package manager:

```
$ sh <(curl -sL https://raw.githubusercontent.com/ocaml/opam/master/
shell/install.sh)
```

We should get the following output:

```
## Downloading opam 2.0.7 for macos on x86_64...
## Downloaded.
## Where should it be installed ? [/usr/local/bin]
## opam 2.0.7 installed to /usr/local/bin
## Converting the opam root format & updating
```

When the installation is successful, we should get the following output, which means that the synchronization has been successful:

```
[default] synchronised from https://opam.ocaml.org
```

2. When the OPAM installation is successful, initialize it using the following command:

```
$ opam init –bare
```

This command will update the profile.

3. Once it is all completed, run this:

```
$ eval $(opam env).
```

This command will update the local environment to use OPAM packages and the compiler.

4. Now clone the Tezos repository, which will download the latest available source code from the Tezos repository:

```
$ git clone -b mainnet https://gitlab.com/tezos/tezos.git
```

You should get the following output:

```
Cloning into 'tezos'...
remote: Enumerating objects: 1635, done.
remote: Counting objects: 100% (1635/1635), done.
remote: Compressing objects: 100% (629/629), done.
remote: Total 68280 (delta 1190), reused 1326 (delta 1004), pack-reused
66645
Receiving objects: 100% (68280/68280), 18.76 MiB | 357.00 KiB/s, done.
Resolving deltas: 100% (56240/56240), done
```

5. Change the directory into the newly created `tezos` directory from the previous step:

```
$ cd tezos
```

6. Install OCaml dependencies using this command:

```
$ make build-deps
```

7. Once the dependencies are built, update the environment again using the following command:

```
$ eval $(opam env)
```

8. Run `make` to build binaries:

```
$ make
```

This will take some time. Once all the compilation is successful, it will create binaries in the `tezos` directory. These binaries are used to provide several services, as shown in the following table:

Binary name	Description
`tezos-node`	The core Tezos daemon. It is an entry point for initializing, configuring, and running a Tezos node.
`tezos-validator`	A binary for the external validation of blocks.
`tezos-client`	Tezos command-line interface and wallet.
`tezos-admin-client`	Node administration tools.
`tezos-signer`	A client for the remote signing of operations or blocks.
`tezos-codec`	A binary for encoding and decoding values.
`tezos-protocol-compiler`	The compiler for economic protocols.
`tezos-baker-005-PsBabyM1`	Baker daemon with the PsBabyM1 Babylon protocol.
`tezos-endorser-005-PsBabyM1`	Endorser daemon with the PsBabyM1 Babylon protocol.
`tezos-accuser-005-PsBabyM1`	Accuser daemon with the PsBabyM1 Babylon protocol.
`tezos-baker-006-PsCARTHA`	Baker daemon with the PsCARTHA Carthage protocol.
`tezos-endorser-006-PsCARTHA`	Endorser daemon with the PsCARTHA Carthage protocol.
`tezos-accuser-006-PsCARTHA`	Accuser daemon with the PsCARTHA Carthage protocol.

This completes our installation. Now we can create the node identity, which will identify the node on the Tezos network:

```
$ ./tezos-node identity generate 26
```

We should get the following output, and an `identity.json` file will be generated in the Tezos node's storage location:

```
Generating a new identity... (level: 26.00)
Stored the new identity (idrv1iPeKNGhHRFrCe47wEYWjdARVY) into '/
Users/<username>/.tezos-node/identity.json'.
```

This may take some time depending on the difficulty chosen (in this case, specified by the `26` in the preceding command), but eventually the ID will be generated. `<username>` will be the username under which Tezos has been installed. The identity consists of a cryptographic key pair and a PoW stamp. PoW in identity generation serves as an antispam protection mechanism.

The value of `26` is the level of difficulty value for PoW. It can be a number between 0 (no PoW) and 256 (the maximum difficulty level) and reflects the number of expected zeros (roughly a difficulty target) in the hash of the identity data structure. The difficulty doubles with each number increment. The PoW mechanism ensures that enough CPU time has been consumed to produce the identity, which helps to thwart Sybil attacks.

We can view the contents of the identity file, which contains elements such as peer ID, public key, private key, and PoW stamp, as shown here:

```
{ "peer_id": "idrv1iPeKNGhHRFrCe47wEYWjdARVY",
  "public_key":
    "8060e898b9e2a357cbe9e81935bb043027c2ee27b4bf9263738b29c2f7d2d715",
  "secret_key":
    "896f500aea0e0e2acd033f838093a485b473e71956f780d684b40241120718e5",
  "proof_of_work_stamp": "5b131c5471a0c0941c5744e3375c6970f943cddc4be423cb"}
```

Now the client can be run with the following command:

```
$ nohup ./tezos-node run --rpc-addr 127.0.0.1:8732 --connections 5 &
```

This will run the client in the background with logs being written to the `nohup` file.

Interacting with the Tezos client

Now we can use the Tezos client to interact with the blockchain:

- For example, to list all understood protocols, enter this command, which will result in the output that follows it:

```
$ ./tezos-client list understood protocols

Disclaimer:
  The  Tezos  network  is  a  new  blockchain technology.
  Users are  solely responsible  for any risks associated
  with usage of the Tezos network.  Users should do their
  own  research to determine  if Tezos is the appropriate
  platform for their needs and should apply judgement and
  care in their network interactions.

Ps9mPmXaRzmz
PsddFKi32cMJ
PsCARTHAGazK
PsYLVpVvgbLh
PtCJ7pwoxe8J
Pt24m4xiPbLD
PsBabyM1eUXZ
```

Note that the disclaimer message will always appear. For brevity, we will not show this message in the subsequent examples.

This command lists all protocols understood by the Tezos client.

- We can also check block synching progress by using the following command:

```
$ ./tezos-client bootstrapped
```

We should get the following output, indicating the synchronization of blocks:

```
Current head: BLGD9HqcQiyK (timestamp: 2018-06-30T18:11:27-00:00,
validation: 2020-05-17T19:29:40-00:00)
Current head: BLmKwFTgjX9A (timestamp: 2018-06-30T18:12:27-00:00,
validation: 2020-05-17T19:29:41-00:00)
Current head: BKoZmRbRzQYc (timestamp: 2018-06-30T18:13:27-00:00,
validation: 2020-05-17T19:29:41-00:00)
Current head: BLNATcNy6owb (timestamp: 2018-06-30T18:14:27-00:00,
validation: 2020-05-17T19:29:41-00:00)
Current head: BM6Ftt9ugHR7 (timestamp: 2018-06-30T18:15:27-00:00,
validation: 2020-05-17T19:29:41-00:00)
Current head: BMRCQjtQSXmW (timestamp: 2018-06-30T18:16:27-00:00,
validation: 2020-05-17T19:29:42-00:00)
```

- In order to find the list of known addresses, we can use the following command:

```
$./tezos-client list known addresses
```

The following output shows the list of known accounts to the Tezos client:

```
myT3account: tz1cKXwHAZksRokEwbvUTWL3FcRZezesPjeX (encrypted sk known)
mytestkeys: tz1Z1YLy6TUCKXaqbDZPXAN5VcUVuughMN7e (encrypted sk known)
```

- We can also find the current balance for our accounts. For example, to get the balance of `myT3account`, we can use the following command:

```
$./tezos-client get balance for myT3account
```

In the output, we see `0` tezzies. Of course, this is just an example; if you have tezzies available, it will show the actual balance:

```
0
```

- To access Tezos `shell` services via `rpc`, the following command can be used:

```
$./tezos-client rpc get /chains/main/blocks/head/header/shell
```

The output of the preceding command shows the `timestamp` and other relevant protocol information:

```
{ "level": 8604, "proto": 1,
  "predecessor": "BME4j19HxuAwjSY5P6AyJvjaeb6XjBALm7H1cZgWwcbzGAfVuUL",
  "timestamp": "2018-07-06T17:44:12Z", "validation_pass": 4,
  "operations_hash":
"LLoa2CJeKKh3wkDa9gAEpgCQj67gZ4HfYTYSgb5R1JSQJ8AfuqDCR",
  "fitness": [ "00", "0000000000044da0" ],
  "context": "CoUwooRMyn9jGAjZ2TzvmsuQhfEbwz5J5xfsm2ShLUc8mDawfkHp" }
```

- To create a new account in Tezos, issue the following command:

```
$./tezos-client gen keys mynewaccount
```

This command will ask for the password, and once the password is provided, it will generate a new account:

```
Enter password to encrypt your key:
Confirm password:
```

- We can see the new account (`mynewaccount`) created by the preceding command by using the following command:

```
$./tezos-client list known addresses
```

The output will list all the addresses of our accounts and their descriptive names:

```
mynewaccount: tz1MW6zMZGK5BJXpaKSjofU45crKDx51oWGs (encrypted sk known)
myT3account: tz1cKXwHAZksRokEwbvUTWL3FcRZezesPjeX (encrypted sk known)
mytestkeys: tz1Z1YLy6TUCKXaqbDZPXAN5VcUVuughMN7e (encrypted sk known)
```

- To stop the node, simply list the Tezos processes and kill it using the standard Unix command:

```
$ ps -eaf | grep Tezos
```

Running this command will list the running Tezos processes:

```
  501 28417 74800   0  8:05pm ttys001   12:19.47 ./tezos-node run
--rpc-addr 127.0.0.1:8732 --connections 5
```

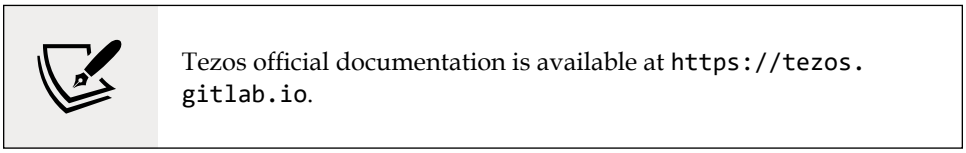
We note the process ID from the output of the preceding command and then issue the `kill` command with the process ID as the argument to kill the process:

```
kill -15 28417
```

Once the process is killed, in the logs we can see this output, which indicates a successful shutdown:

```
May 17 20:43:31 - shell.validation_process.external: The process
terminated normally
May 17 20:43:31 - node.main: Shutting down the RPC server...
May 17 20:43:31 - node.main: BYE (-11)
```

This completes our basic introduction to Tezos. It is a vast subject, and these few pages cannot do justice to the vast and complex Tezos ecosystem. However, this basic introduction should serve as a solid foundation to explore further.



In the next section, we discuss Ripple, which offers a blockchain platform for global payments.

Ripple

Introduced in 2012, **Ripple** is a currency exchange and real-time gross settlement system. In Ripple, the payments are settled without any waiting, as opposed to traditional settlement networks, where settlement can take days.

It has a native currency called **Ripples** (**XRP**), but it also supports non-XRP payments. This system is considered similar to a traditional money transfer mechanism known as **Hawala**. This system works by making use of agents who take money and a password, from the sender, then contact the payee's agent and instruct them to release funds to the person who can provide the password. The payee then contacts the local agent, tells them the password and collects the funds. An analogy to the agent is a **gateway** in Ripple. This is just a very simple analogy; the actual protocol is rather complex, but it is the same in principle.

The Ripple network is composed of various nodes that can perform different functions based on their type:

- **User nodes**: These nodes are used in payment transactions and can send or receive payments.
- **Validator nodes**: These nodes participate in the consensus mechanism. Each server maintains a set of unique nodes, which it needs to query while achieving consensus. Nodes in the **Unique Node List** (**UNL**) are trusted by the server involved in the consensus mechanism, which will accept votes only from this list of unique nodes.

Ripple is sometimes not considered a truly decentralized network as there are network operators and regulators involved. However, it can be considered decentralized due to the fact that anyone can become part of the network by running a validator node. Moreover, the consensus process is also decentralized because any proposed changes to the ledger have to be decided by following a scheme of **super majority voting**. However, this is a hot topic among researchers and enthusiasts and there are arguments against and in favor of each school of thought. There are some discussions online that readers can refer to for further exploration of these ideas. You can find a couple of these online discussions at the following addresses:

- `https://www.quora.com/Why-is-Ripple-centralized`
- `https://thenextweb.com/hardfork/2018/02/06/ripple-report-bitmex-centralized/`

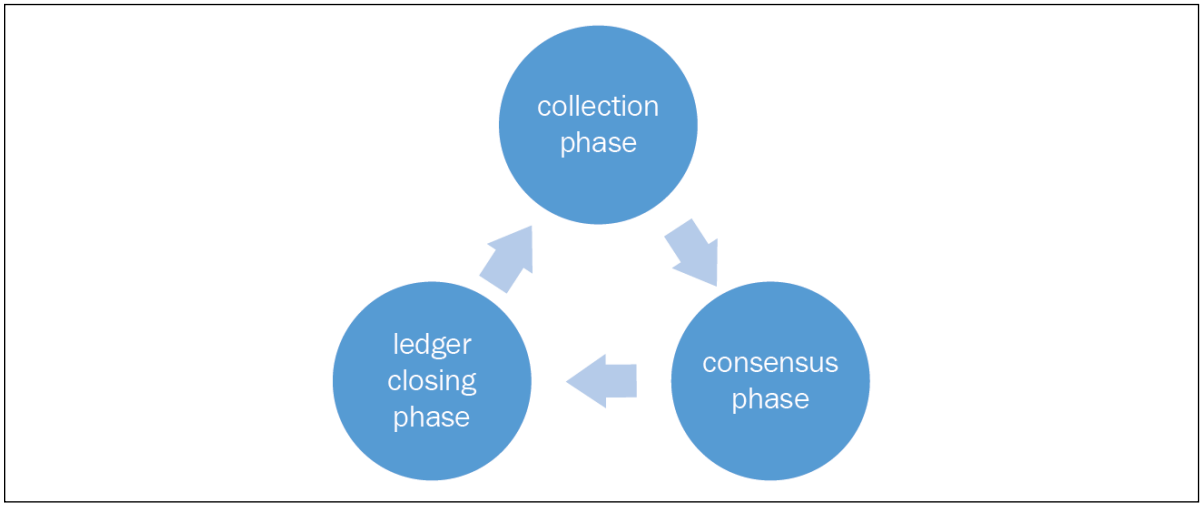
Ripple maintains a globally distributed ledger of all transactions that are governed by a novel low-latency consensus algorithm called **Ripple Protocol Consensus Algorithm** (**RPCA**). The consensus process works by achieving agreement on the state of an open ledger containing transactions by seeking verification and acceptance from validating servers in an iterative manner until an adequate number of votes is collected. Once enough votes are received (a super majority, initially 50% and gradually increasing with each iteration up to at least 80%), the changes are validated and the ledger is closed. At this point, an alert is sent to the whole network indicating that the ledger is closed.

The original research paper for RPCA is available at `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.

Another updated version is available at `https://arxiv.org/pdf/1802.07242.pdf`.

In summary, the consensus protocol is a three-phase process:

1. Collection phase: In this phase, validating nodes gather all transactions broadcast on the network by account owners and validate them. Transactions, once they are accepted, are called *candidate transactions* and can be accepted or rejected based on the validation criteria.
2. Consensus phase: After the collection phase, the consensus process starts, and after achieving it, the ledger is closed.
3. Ledger closing phase: This process runs asynchronously every few seconds in rounds and when it completes, the ledger is opened and closed (updated) accordingly:



Ripple consensus protocol phases

In a Ripple network, there are a number of components that work together in order to achieve consensus and form a payment network. These components are discussed individually here:

- **Server**: This component serves as a participant in the consensus protocol. Ripple server software is required in order to be able to participate in the consensus protocol.
- **Ledger**: This is the main record of balances of all accounts on the network. A ledger contains various elements, such as the ledger number, account settings, transactions, timestamp, and a flag that indicates the validity of the ledger.
- **Last closed ledger**: A ledger is closed once consensus is achieved by validating nodes.

- **Open ledger**: This is a ledger that has not been validated yet and no consensus has been reached about its state. Each node has its own open ledger, which contains proposed transactions.
- **Unique Node List**: This is a list of unique trusted nodes that a validating server uses in order to seek votes and subsequent consensus.
- **Proposer**: As the name suggests, this component proposes new transactions to be included in the consensus process. It is usually a subset of nodes (UNL defined in the previous point) that can propose transactions to the validating server.

Like any other blockchain, a fundamental activity in Ripple is a transaction. We introduce the design and architecture of transactions in Ripple in the next section.

Transactions

Transactions are created by the network users in order to update the ledger. A transaction is expected to be digitally signed and valid in order for it to be considered as a candidate in the consensus process. Each transaction costs a small amount of XRP, which serves as a protection mechanism against **denial of service** (**DoS**) attacks caused by spamming.

There are different types of transaction in the Ripple network. A single field within the Ripple transaction data structure called `TransactionType` is used to represent the type of the transaction. Transactions are executed by using a four-step process:

1. First, transactions are prepared whereby an unsigned transaction is created by following the standards.
2. The second step is signing, where the transaction is digitally signed by the sender to authorize it.
3. After this, the actual submission to the network occurs via the connected server.
4. Finally, the verification is performed to ensure that the transaction is validated successfully.

A transaction in Ripple is composed of various fields that are common to all transaction types. These fields are listed as follows with a description:

- `Account`: This is the address of the initiator of the transaction.
- `AccountTxnID`: This is an optional field that contains the hash of another transaction. It is used to chain the transactions together.
- `Fee`: This is the amount of XRP.
- `Flags`: This is an optional field specifying the flags for the transaction.
- `LastLedgerSequence`: This is the highest sequence number of the ledger in which the transaction can appear.
- `Memos`: This represents optional arbitrary information.
- `SigningPubKey`: This represents the public key.
- `Signers`: This represent signers in a `multisig` transaction.
- `SourceTag`: This represents either the sender of, or the reason for, the transaction.
- `TxnSignature`: This is the verification digital signature for the transaction.

Roughly, transactions can be categorized into three types, namely **payments-related**, **order-related**, and **account-and-security related**. These types and their unique fields are described in the following sections.

Payments-related

There are several fields in this category that result in certain actions. These fields are described as follows:

- `Payment`: This transaction is most commonly used and allows one user to send funds to another.
- `PaymentChannelClaim`: This is used to claim XRP from a payment channel. A payment channel is a mechanism that allows recurring and unidirectional payments between parties. This can also be used to set the expiration time of the payment channel.
- `PaymentChannelCreate`: This transaction creates a new payment channel and adds XRP to it in **drops**. A single drop is equivalent to 0.000001 of an XRP.
- `PaymentChannelFund`: This transaction is used to add more funds to an existing channel. Similar to the `PaymentChannelClaim` transaction, this can also be used to modify the expiration time of the payment channel.

Order-related

This type of transaction includes the following two fields:

- `OfferCreate`: This transaction represents a limit order, which represents an intent for the exchange of currency. It results in creating an offer node in the consensus ledger if it cannot be completely fulfilled.
- `OfferCancel`: This is used to remove a previously created offer node from the consensus ledger, indicating withdrawal of the order.

Account and security-related

This type of transaction includes the fields listed as follows. Each field is responsible for performing a certain function:

- `AccountSet`: This transaction is used to modify the attributes of an account in the Ripple consensus ledger.
- `SetRegularKey`: This is used to change or set the transaction signing key for an account. An account is identified using a base58 Ripple address derived from the account's master public key.
- `SignerListSet`: This can be used to create a set of signers for use in multi-signature transactions.
- `TrustSet`: This is used to create or modify a trust line between accounts.

Interledger

Original work on the **Interledger** protocol was started by Ryan Fugger in 2004. With the introduction of Bitcoin in 2009, this work generated even more interest, and since then many contributions have been made to this project.

A protocol for Interledger payments was invented by Stefan Thomas and Evan Schwartz from Ripple. The research paper is available at `https://interledger.org/interledger.pdf`.

Interledger is an open protocol suite for cross-ledger payments. It is composed of four layers: **Application**, **Transport**, **Interledger**, and **Ledger**. Each layer is responsible for performing various functions under certain protocols. These functions and protocols are described in the following section.

The official website is `https://interledger.org`.

The RFCs of this protocol are available at `https://github.com/interledger/rfcs`.

Application layer

Protocols running on this layer govern the key attributes of a payment transaction. Examples of application layer protocols include **Simple Payment Setup Protocol** (**SPSP**) and **Open Web Payment Scheme** (**OWPS**). SPSP is an Interledger protocol that allows secure payment across different ledgers by creating connectors between them. OWPS is another scheme that allows consumer payments across different networks.

Once the protocols on this layer have run successfully, protocols from the transport layer will be invoked in order to start the payment process.

Transport layer

This layer is responsible for managing transactions. Protocols such as **Optimistic Transport Protocol** (**OTP**), **Universal Transport Protocol** (**UTP**), and **Atomic Transport Protocol** (**ATP**) are available currently for this layer. OTP is the simplest protocol, which manages payment transfers without any escrow protection, whereas UTP provides escrow protection. ATP is the most advanced protocol, which not only provides an escrowed transfer mechanism, but also makes use of trusted notaries to further secure transactions.

Interledger layer

This layer provides interoperability and routing services. This layer contains protocols such as **Interledger Protocol** (**ILP**), **Interledger Quoting Protocol** (**ILQP**), and **Interledger Control Protocol** (**ILCP**). ILP packet provides the final target (destination) of the transaction in a transfer. ILQP is used in making quote requests by the senders before the actual transfer. ILCP is used to exchange data related to routing information and payment errors between connectors on the payment network.

Ledger layer

This layer contains protocols that enable the communication and execution of payment transactions between connectors. **Connectors** are objects that implement the protocol for forwarding payments between different ledgers. The ledger layer can support various protocols such as **Simple Ledger Protocol** (**SLP**), various blockchain protocols, legacy protocols, and different proprietary protocols.

Ripple Connect consists of various **plug and play** modules that allow connectivity between ledgers by using the ILP.

Plug and play is a feature of software or electronic devices that allows them to work the first time they are used without requiring any configuration by the user.

It enables the exchange of required data between parties before the transaction, visibility, fee management, delivery confirmation, and secure communication using transport layer security. A third-party application can connect to the Ripple network via various connectors that forward payments between different ledgers.

All layers described in the preceding sections make up the architecture of the ILP. Overall, Ripple is a solution that targets the financial industry and makes real-time payments possible without any settlement risk.

As this is a very feature-rich platform, covering all aspects of it is not possible in this brief introduction. However, Ripple documentation is available at `https://ripple.com/`.

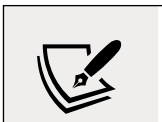
In the next section, we'll discuss another payment network called Stellar, which emerged about two years after the introduction of Ripple.

Stellar

Stellar is a payment network based on blockchain technology and a novel consensus model called **Federated Byzantine Agreement** (**FBA**). FBA works by creating quorums of trusted parties. **Stellar Consensus Protocol** (**SCP**) is an implementation of FBA.

Stellar Consensus Protocol

Some of the key issues identified in the Stellar white paper are the cost and complexity of the current financial infrastructure. This limitation warrants the need for a global financial network that addresses these issues without compromising the integrity and security of the financial transaction. This requirement resulted in the invention of SCP, which is a demonstrably safe consensus mechanism.



The original research paper for SCP is available at `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`.

SCP has four main properties, which are described here:

- **Decentralized control**: This allows participation by anyone without any central party.
- **Low latency**: This addresses the much-desired requirement of fast transaction processing.
- **Flexible trust**: This allows users to choose which parties they trust for a specific purpose.
- **Asymptotic security**: This makes use of digital signatures and hash functions for providing the required level of security on the network.

The Stellar network allows the transfer and representation of the value of an asset by its native digital currency, called **lumens,** abbreviated as XLM. Lumens are consumed when a transaction is broadcast on the network, which also serves as a deterrent against DoS attacks.

At its core, the Stellar network maintains a distributed ledger that records every transaction and is replicated on each Stellar server (node). The consensus is achieved by verifying transactions between servers and updating the ledger with updates. The Stellar ledger can also act as a distributed exchange order book by allowing users to store their offers to buy or sell currencies.

The core software is available at `https://github.com/stellar/stellar-core`.

Rootstock

Before discussing **Rootstock** (**RSK**) in detail, it's important to define and introduce some concepts that are fundamental to its design. These concepts include **two-way pegging**, **sidechains**, and **drivechains**.

Two-way pegging

This is a mechanism by which value (coins) can be transferred between one blockchain and another. There is no real transfer of coin between chains. The idea revolves around the concept of locking the same amount and value of coins in a Bitcoin blockchain (the main chain) and unlocking the equivalent number of tokens in the secondary chain.

Bearing this definition in mind, sidechains will be defined next.

Sidechain

This is a blockchain that runs in parallel with a main blockchain and allows the transfer of value between them. This means that tokens from one blockchain can be used in the sidechain and vice versa. This is also called a pegged sidechain because it supports two-way pegged assets. The concept of the sidechain was originally developed by Blockstream.

Blockstream's online presence is located at `https://blockstream.com`.

Drivechain

This is a relatively new concept, where control on unlocking the locked bitcoins (in the main chain) is given to the miners who can vote when to unlock them. This is in contrast with sidechains, where consensus is validated though a simple payment verification mechanism in order to transfer the coins back to the main chain.

For more details regarding drivechains, sidechains, and two-way peg designs, refer to the original research paper at `https://docs.rsk.co/Drivechains_Sidechains_and_Hybrid_2-way_peg_Designs_R9.pdf`.

Now that we understand the basic ideas behind these core concepts, let's now discuss RSK in more detail.

RSK is a smart contract platform that has a two-way peg into the Bitcoin blockchain. The core idea is to increase the scalability and performance of the Bitcoin system and enable it to work with smart contracts. RSK runs a Turing complete deterministic VM called the **Rootstock Virtual Machine** (**RVM**).

It is also compatible with the EVM and allows Solidity-compiled contracts to run on RSK. Smart contracts can also run under the time-tested security of Bitcoin. The RSK blockchain works by merge-mining (merge-mining is the process of mining more than one blockchain at once) with Bitcoin. This allows RSK to achieve the same level of security as Bitcoin. This is especially true for preventing double-spends and achieving settlement finality. It allows scalability, up to 400 transactions per second due to faster block times and other design considerations.

The research paper is available at `https://uploads.strikinglycdn.com/files/ec5278f8-218c-407a-af3c-ab71a910246d/RSK%20White%20Paper%20-%20Overview.pdf` should you want to explore it further.

RSK has released a mainnet called Bamboo, which is currently in beta.

Further information on Rootstock is available at `http://www.rsk.co/`.

In the next section, we introduce some projects related to decentralized storage.

Storage layer blockchain projects

In this section, we introduce some projects that use the principles of blockchain technology to provide decentralized storage systems that can be used by blockchains or by any other project that wishes to use decentralized storage. These projects include Storj, Maidsafe, and BigchainDB.

Storj

Existing models for cloud-based storage are all centralized solutions, which may or may not be as secure as users expect them to be. We need cloud storage systems that are secure, highly available, and above all decentralized. **Storj** aims to provide blockchain-based, decentralized, and distributed storage. It is a cloud shared by the community instead of a central organization. It allows the execution of storage contracts between nodes that act as autonomous agents. These agents (nodes) execute various functions such as data transfer, validation, and data integrity checks.

Storj's core concept is based on a **Distributed Hash Table** (**DHT**) called **Kademlia**. However, Storj's protocol has been enhanced by adding new message types and functionalities. It also implements a P2P publish/subscribe (**pub/sub**) mechanism known as **Quasar**, which ensures that messages successfully reach the nodes that are interested in storage contracts. This is achieved via a storage contract parameter selection mechanism based on Bloom filters.

Storj stores files in an encrypted format spread across the network. Before the file is stored on the network, it is encrypted using AES-256-CTR symmetric encryption and is then stored piece by piece in a distributed manner on the network. This process of dissecting the file into pieces is called **sharding** and results in increased availability, security, performance, and privacy of the network. Also, if a node fails, the shard is still available because, by default, a single shard is stored at three different locations on the network.

It maintains a blockchain, which serves as a shared ledger and implements standard security features such as public/private key cryptography and hash functions, similar to any other blockchain. As the system is based on hard drive sharing between peers, anyone can contribute by sharing the extra space on their drive and get paid with Storj's own cryptocurrency, called **Storjcoin X** (**SJCX**). SJCX was developed as a **Counterparty** asset and makes use of the Bitcoin blockchain-based Counterparty platform for transactions. This has now been migrated to Ethereum.

A detailed discussion is available at `https://blog.storj.io/post/158740607128/migration-from-counterparty-to-ethereum`.

Storj code is available at `https://github.com/Storj/`.

MaidSafe

This is another distributed storage system similar to Storj. Users are paid in **SafeCoin** for their storage space contribution to the network. This mechanism of payment is governed by **Proof of Resource**, which ensures that the disk space committed by a user to the network is available; if not, then the payment of SafeCoin will drop accordingly. The files are encrypted and divided into small portions before being transmitted on to the network for storage.

Another concept of **opportunistic caching** has been introduced with **MaidSafe**, which is a mechanism to create copies of frequently accessed data physically closer to where the access requests are coming from, which results in high performance of the network. Another novel feature of Maidsafe's network (the SAFE network) is that it automatically removes any duplicate data on the network, thus resulting in reduced storage requirements.

Moreover, the concept of **churning** has also been introduced, which basically means that data is constantly moved across the network so that the data cannot be targeted by malicious adversaries. It also keeps multiple copies of data across the network to provide redundancy in case a node goes offline or fails.

BigchainDB

BigchainDB is a scalable blockchain database. It is not strictly a blockchain itself but complements blockchain technology by providing a decentralized database. At its core, it's a distributed database, but with the added attributes of a blockchain such as decentralization, immutability, and handling of digital assets. It also allows the use of NoSQL for querying the database.

It is intended to provide a database in a decentralized ecosystem where not only processing is decentralized (a blockchain) or the filesystem is decentralized (for example, IPFS), but the database is also decentralized. This makes the whole application ecosystem decentralized.

This is available at `https://www.bigchaindb.com/`.

After this exploration of some notable storage layer blockchain projects, let's introduce some other platforms that have introduced some new ideas into the blockchain world.

Other platforms

This section will briefly consider some novel platforms that have introduced some innovative ideas to blockchain technology.

MultiChain

MultiChain has been developed as a platform for the development and deployment of private blockchains. It is based on Bitcoin code and addresses security, scalability, and privacy issues. It is a highly configurable blockchain platform that allows users to set different blockchain parameters. It supports control and privacy via a granular permissioning layer. Installation of MultiChain is very quick.

To install MultiChain, this link can be followed: `http://www.multichain.com/download-install/`.

Tendermint

Tendermint is a software that provides a BFT consensus mechanism and state machine replication functionality to an application. Its main motivation is to develop a general-purpose, secure, and high-performance replicated state machine.

There are two components of Tendermint, which are described in the following sections.

Tendermint Core

This is a consensus engine that enables secure replication of transactions on each node in the network.

Tendermint Socket Protocol

Tendermint Socket Protocol (**TMSP**) is an application interface protocol that allows interfacing with any programming language to process transactions. Tendermint allows decoupling of the application consensus processes, which allows any application to benefit from the consensus mechanism.

Tendermint consensus algorithm

The Tendermint consensus algorithm is a round-based mechanism where validator nodes propose new blocks in each round. A locking mechanism is used to ensure protection against a scenario where two different blocks are selected to be committed at the same height of the blockchain. Each validator node maintains a full local replicated ledger of blocks that contain transactions. Each block contains a header, which consists of the previous block hash, the timestamp of the proposal of the block, the current block height, and the Merkle root hash of all transactions present in the block.

Cosmos

Tendermint has recently been used in **Cosmos** (`https://cosmos.network`), which is a network of blockchains that allows interoperability between different chains running on the BFT consensus algorithm. Blockchains on this network are called zones. The first zone in Cosmos is called Cosmos hub, which is, in fact, a public blockchain and is responsible for providing connectivity services to other blockchains. For this purpose, the hub makes use of the **Inter Blockchain Communication** (**IBC**) protocol. The IBC protocol supports two types of transactions called `IBCBlockCimmitTx` and `IBCPacketTx`. The first type is used to provide proof of the most recent block hash in a blockchain to any party, whereas the latter type is used to provide data origin authentication. A packet from one blockchain to another is published by first posting a proof to the target chain. The receiving (target) chain checks this proof in order to verify that the sending chain has indeed published the packet. In addition, Cosmos has its own native currency called Atom. This scheme addresses scalability and interoperability issues by allowing multiple blockchains to connect to the hub.

Tendermint is available at `https://tendermint.com/`.

In this vast ecosystem of blockchains and networks, several initiatives have been taken to introduce more feature-rich platforms with better security, interoperability, developer tools and toolkits. One of these efforts is Eris.

Eris

Eris is not a single blockchain, it is an open modular platform developed by **Monax** for the development of blockchain-based ecosystem applications. It offers various frameworks, SDKs, and tools that allow accelerated development and deployment of blockchain applications.

The core idea behind the Eris application platform is to enable the development and management of ecosystem applications with a blockchain backend. It allows integration with multiple blockchains and enables various third-party systems to interact with various other systems.

This platform makes use of smart contracts written in Solidity. It can interact with blockchains such as Ethereum and Bitcoin. The interaction can include connectivity commands, starting, stopping, disconnecting, and creating new blockchains. Complexity related to setup and interaction with blockchains has been abstracted away in Eris. All commands are standardized for different blockchains, and the same commands can be used across the platform regardless of the blockchain type being targeted.

An ecosystem application can consist of the Eris platform, enabling the API gateway to allow legacy applications to connect to key management systems, consensus engines, and application engines. The Eris platform provides various toolkits that are used to provide various services to the developers. These modules are described as follows:

- **Chains**: this allows the creation of and interaction with blockchains.
- **Packages**: this allows the development of smart contracts.
- **Keys**: this is used for key management and signing operations.
- **Files**: this allows working with distributed data management systems. It can be used to interact with filesystems such as IPFS and data lakes.
- **Services**: this exposes a set of services that allows the management and integration of ecosystem applications.

Several SDKs have also been developed by Eris that allow the development and management of ecosystem applications. These SDKs contain smart contracts that have been fully tested and address specific needs and requirements of business, for example, a finance SDK, insurance SDK, and logistics SDK. There is also a base SDK that serves as a basic development kit to manage the life cycle of an ecosystem application.

Monax has developed its own permissioned blockchain client called `eris:db`.

`erisdb` became Hyperledger Burrow under Hyperledger. For more information, visit: `https://www.hyperledger.org/wp-content/uploads/2017/06/HIP_Burrowv2.pdf`.

It is a **Proof of Stake** (**PoS**)-based blockchain system that allows integration with a number of different blockchain networks. The `eris:db` client consists of four components:

- **Consensus**: this is based on the Tendermint consensus mechanism, (discussed in the *Tendermint* section).
- **Virtual machine**: Eris uses the Ethereum Virtual Machine (EVM), and as such it supports Solidity-compiled contracts.
- **Permissions layer**: being a permissioned ledger, Eris provides an access control mechanism that can be used to assign specific roles to different entities on the network.
- **Interface**: this provides various command-line tools and RPC interfaces to enable interaction with the backend blockchain network.

The key difference between the Ethereum blockchain and `eris:db` is that `eris:db` makes use of a **Practical Byzantine Fault-Tolerance** (**PBFT**) algorithm, which is implemented as a **deposit-based Proof of Stake** (**DPoS**) system, whereas Ethereum uses PoW. Moreover, `eris:db` uses the ECDSA `ed22519` curve scheme, whereas Ethereum uses the `secp256k1` algorithm. Finally, it is permissioned with an access control layer on top, whereas Ethereum is a public blockchain.

Eris is a feature-rich application platform that offers a large selection of toolkits and services to develop custom blockchain solutions. More information on this platform is available at `https://erisindustries.com`.

This brings our introduction to some innovative blockchain-based platform services to an end. We started with an introduction to alternative blockchains and was divided into two main sections discussing blockchains and other asserted platforms and tools. Blockchain technology is a thriving area; as such, changes are quite rapid in existing solutions and new technologies or tools are being introduced almost every day. Here, a careful selection of platforms and blockchains was introduced. New blockchains such as Kadena, various new protocols such as Ripple, and concepts such as sidechains and drivechains were also discussed.

The material covered is intended to provide a strong foundation for more in-depth research into areas that readers are interested in. As said before, blockchain is a very fast-moving field, and there are many other blockchain projects, such as Libra, Tau-Chain, HydraChain, Elements, CREDITS, Qtum, and many more that have not been introduced here. Readers are encouraged to keep an eye on any developments in this field to keep themselves up to date with advancement in this rapidly growing area, but a few are discussed in the following sections to give an idea of current projects.

Notable projects

In this section, we'll look at some notable blockchain projects that are currently in progress. In addition to these projects, there is also a myriad of start-ups and companies working with blockchain technology and offering blockchain-related products.

Zcash on Ethereum

A recent project by the Ethereum research and development team is the implementation of Zcash on Ethereum. This is an exciting project in which developers are trying to create a privacy layer for Ethereum using zk-SNARKs that are already used in Zcash.

With a Zcash implementation on Ethereum, the aim is to create a platform that allows for applications such as voting, where privacy is of paramount importance. It will also allow the creation of anonymous tokens on the Ethereum chain that can be used in a number of applications.

Zether

Zether is a new mechanism introduced to provide privacy on Ethereum blockchain. It is implemented using smart contracts, and provides transaction confidentiality. Zether also proposes an approach to achieving greater anonymity, which has been developed as a complete protocol and implemented in **Anonymous Zether**.

More information about Zether and Anonymous Zether protocol is available at `https://crypto.stanford.edu/~buenz/papers/zether.pdf` and `https://ia.cr/2020/293`.

Libra

Libra is a permissioned blockchain cryptocurrency proposed by Facebook. It is in the experimental stage, and is expected to go in production in late 2020.

More information about this initiative is available here: `https://libra.org/en-US/`.

CollCo

Collateralized Coin (**CollCo**) is a project developed by Deutsche Börse, which is based on the Hyperledger code base and is used for managing commercial bank cash transfers. CollCo provides a blockchain-based platform that allows the real-time transfer of commercial banks' money while still relying on traditional capabilities provided by Eurex Clearing CCP. This is a major project that can be used to address inefficiencies in the post-trade settlement processes.

A study is available here: `https://www.hyperledger.org/wp-content/uploads/2018/03/Hyperledger_CaseStudy_DeutscheBorse_FINAL.pdf`.

Solidus

This is a new cryptocurrency, which provides a solution for selfish mining (groups of miners colluding to increase their revenue) while addressing scaling, performance, and confidentiality issues. It is based on a permissionless Byzantine consensus mechanism.

The original research paper is available at: `https://eprint.iacr.org/2017/317.pdf`.

Hawk

Hawk is a project that aims to address the privacy issues facing smart contracts in blockchains. It is a smart contract system that allows a programmer to write a private smart contract without the need to manually program the cryptographic protocol.

The paper on this topic is available here: `https://eprint.iacr.org/2015/675.pdf`.

Town Crier

This project aims to provide real-world, authentic feeds into smart contracts. This system is based on Intel's SGX trusted hardware technology. This is a step further in **oracle** design, whereby smart contracts can request data from online sources while simultaneously preserving confidentiality.

More information on this project can be found on its website: `https://www.town-crier.org`.

TEEChan

TEEChan proposed the novel idea of using **Trusted Execution Environments** (**TEEs**) to provide a scalable and efficient approach to scaling the Bitcoin blockchain. This is similar to the concept of payment channels, whereby off-chain channels are used for faster transfers of transactions. The principal attraction of this idea is that it is implementable on the Bitcoin blockchain without the need for any changes in the Bitcoin network (because it is an off-the-chain solution).

There is, however, a small caveat, in that this solution does require users to trust Intel for remote attestation (verification) as Intel's SGX CPUs are used to provide TEEs. This is not a desirable property in decentralized blockchains; however, it should be noted that the confidentiality of transactions is still preserved even if remote attestation is used, as the remote attester (Intel) cannot see the contents of the communication between users. This limitation makes the question of whether it is an entirely decentralized and trustless solution or not a matter of debate.

More details can be found here: `https://arxiv.org/abs/1612.07766`.

Falcon

Falcon is a project that helps Bitcoin scalability, by providing a fast relay network for the purpose of broadcasting Bitcoin blocks. The core idea revolves around a technique to reduce orphan blocks and speed up block transmission, thus helping with the overall scalability of the Bitcoin network. The technique used for this purpose has been called application-level cut-through routing.

More information is available at the Falcon website: `https://www.falcon-net.org`.

Bletchley

This project was introduced by Microsoft, indicating Microsoft's commitment to blockchain technology. **Bletchley** allows the use of Azure cloud services to build blockchains in a user-friendly manner. A major concept introduced by Bletchley is called **cryptlets**, which can be thought of as an advanced version of oracles, which reside outside the blockchain and can be called by smart contracts using secure channels. These can be written in any language and execute within a secure container.

There are two types of cryptlets: **utility cryptlets** and **contract cryptlets**. The first type is used to provide basic services, such as encryption and basic data fetching from external sources. On the other hand, the latter is a more intelligent version that is created automatically when a smart contract is created on the chain, and which resides off-chain while still being linked to the on-chain contract. Contract cryptlets execute in a secure off-chain container and are communicated with using secure channels. They also are signed by trusted and known counterparties, thus eliminating the need to execute code on all blockchain nodes and resulting in increased blockchain performance.

The Bletchley whitepaper is available at: `https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/bletchley-whitepaper.md`.

Now we've covered some interesting projects that would be useful points for further research, let's explore a few tools that weren't covered in the main book, but are useful to consider when beginning your own blockchain projects!

Miscellaneous tools

Some tools that have not been previously discussed are listed in the following subsections, as a brief introduction to make readers aware of the myriad of development options available for blockchains. This list includes platforms, utilities, and tools that can be used for blockchain development.

Solidity extension for Microsoft Visual Studio

This extension provides IntelliSense, autocompletion, and DApp templates. It works within the familiar Visual Studio IDE, making it easier for developers to familiarize themselves with Ethereum development.

You can download this extension from: `https://marketplace.visualstudio.com/items?itemName=ConsenSys.Solidity`.

Stratis

This is a blockchain development platform that works in conjunction with the main **Stratis** blockchain for security reasons, and allows the creation of custom private blockchains.

It allows development of blockchain applications using C# .NET technologies. It is also available via Microsoft Azure as part of BaaS services.

This is available at: `https://stratisplatform.com/`.

Meteor

This is a full-stack development framework for single-page applications. It can be used for Ethereum DApp development. There is a development environment available in **Meteor** which allows easier development of complex DApps.

It is available at `https://www.meteor.com/` and Ethereum-specific DApp building information is available at `https://github.com/ethereum/wiki/wiki/Dapp-using-Meteor`.

uPort

This platform is built on Ethereum and provides a decentralized identity management system. This allows users to have full control over their identity and personal information on the blockchain. It is based on the idea of reputation systems, which enable users to attest one another and build trust. In this way, users can be confident that the parties they are interacting with are reputable, and have the authority to operate on the network. It also allows secure and scalable data exchanges between individuals and organizations.

This is available at `https://www.uport.me/`.

INFURA

This project aims to provide enterprise-level Ethereum and IPFS access which can help with mainstream adoption of blockchain technology. INFURA consists of Ethereum nodes, IPFS nodes, and a service layer named Ferryman which provides routing and load balancing services.

More information and resources available on the website `https://infura.io`.

It is not possible to cover all projects here that are being developed in the blockchain space. However, with the examples provided here, you should start to understand some of the types of projects out there. The information provided in this section should enable you to further explore these, and similar, projects and tools further, and participate in this exciting and rapidly evolving discipline!