Ethereum Clients

An Ethereum client is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum clients. Different Ethereum clients *interoperate* if they comply with the reference specification and the standardized communications protocols. While these different clients are implemented by different teams and in different programming languages, they all "speak" the same protocol and follow the same rules. As such, they can all be used to operate and interact with the same Ethereum network.

Ethereum is an open source project, and the source code for all the major clients is available under open source licenses (e.g., LGPL v3.0), free to download and use for any purpose. *Open source* means more than simply free to use, though. It also means that Ethereum is developed by an open community of volunteers and can be modified by anyone. More eyes means more trustworthy code.

Ethereum is defined by a formal specification called the "Yellow Paper" (see [references]).

This is in contrast to, for example, Bitcoin, which is not defined in any formal way. Where Bitcoin's "specification" is the reference implementation Bitcoin Core, Ethereum's specification is documented in a paper that combines an English and a mathematical (formal) specification. This formal specification, in addition to various Ethereum Improvement Proposals, defines the standard behavior of an Ethereum client. The Yellow Paper is periodically updated as major changes are made to Ethereum.

As a result of Ethereum's clear formal specification, there are a number of independently developed, yet interoperable, software implementations of an Ethereum client. Ethereum has a greater diversity of implementations running on the network than any other blockchain, which is generally regarded as a good thing. Indeed, it has, for example, proven itself to be an excellent way of defending against attacks on the network, because exploitation of a particular client's implementation strategy simply hassles the developers while they patch the exploit, while other clients keep the network running almost unaffected.

Ethereum Networks

There exist a variety of Ethereum-based networks that largely conform to the formal specification defined in the Ethereum Yellow Paper, but which may or may not interoperate with each other.

Among these Ethereum-based networks are Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin, and many others. While mostly compatible at the protocol level, these networks often have features or attributes that require maintainers of Ethereum client software to make small changes in order to support each network. Because of this, not every version of Ethereum client software runs every Ethereum-based blockchain.

Currently, there are six main implementations of the Ethereum protocol, written in six different languages:

- Parity, written in Rust
- Geth, written in Go
- cpp-ethereum, written in C++

- pyethereum, written in Python
- Mantis, written in Scala
- Harmony, written in Java

In this section, we will look at the two most common clients, Parity and Geth. We'll show how to set up a node using each client, and explore some of their command-line options and application programming interfaces (APIs).

Should I Run a Full Node?

The health, resilience, and censorship resistance of blockchains depend on them having many independently operated and geographically dispersed full nodes. Each full node can help other new nodes obtain the block data to bootstrap their operation, as well as offering the operator an authoritative and independent verification of all transactions and contracts.

However, running a full node will incur a cost in hardware resources and bandwidth. A full node may download up to 300 GB of data (as of March 2021, depending on the client configuration) and store it on a local hard drive. This data burden increases quite rapidly every day as new transactions and blocks are added. We discuss this topic in greater detail in Hardware Requirements for a Full Node.

A full node running on a live *mainnet* network is not necessary for Ethereum development. You can do almost everything you need to do with a *testnet* node (which connects you to one of the smaller public test blockchains), with a local private blockchain like Ganache, or with a cloud-based Ethereum client offered by a service provider like Infura.

You also have the option of running a remote client, which does not store a local copy of the blockchain or validate blocks and transactions. These clients offer the functionality of a wallet and can create and broadcast transactions. Remote clients can be used to connect to existing networks, such as your own full node, a public blockchain, a public or permissioned (proof-of-authority) testnet, or a private local blockchain. In practice, you will likely use a remote client such as MetaMask, Emerald Wallet, MyEtherWallet, or MyCrypto as a convenient way to switch between all of the different node options.

The terms "remote client" and "wallet" are used interchangeably, though there are some differences. Usually, a remote client offers an API (such as the web3.js API) in addition to the transaction functionality of a wallet.

Do not confuse the concept of a remote wallet in Ethereum with that of a *light client* (which is analogous to a Simplified Payment Verification client in Bitcoin). Light clients validate block headers and use Merkle proofs to validate the inclusion of transactions in the blockchain and determine their effects, giving them a similar level of security to a full node. Conversely, Ethereum remote clients do not validate block headers or transactions. They entirely trust a full client to give them access to the blockchain, and hence lose significant security and anonymity guarantees. You can mitigate these problems by using a full client you run yourself.

Full Node Advantages and Disadvantages

Choosing to run a full node helps with the operation of the networks you connect it to, but also incurs some mild to moderate costs for you. Let’s look at some of the advantages and disadvantages.

Advantages:

- Supports the resilience and censorship resistance of Ethereum-based networks
- Authoritatively validates all transactions
- Can interact with any contract on the public blockchain without an intermediary
- Can directly deploy contracts into the public blockchain without an intermediary
- Can query (read-only) the blockchain status (accounts, contracts, etc.) offline
- Can query the blockchain without letting a third party know the information you’re reading

Disadvantages:

- Requires significant and growing hardware and bandwidth resources
- May require several days to fully sync when first started
- Must be maintained, upgraded, and kept online to remain synced

Public Testnet Advantages and Disadvantages

Whether or not you choose to run a full node, you will probably want to run a public testnet node. Let’s look at some of the advantages and disadvantages of using a public testnet.

Advantages:

- A testnet node needs to sync and store significantly less data compared to mainnet—about 75 GB depending on the network.
- A testnet node can sync fully in much less time.
- Deploying contracts or making transactions requires test ether, which has no value and can be acquired for free from several "faucets."
- Testnets are public blockchains with many other users and contracts, running "live."

Disadvantages:

- You can’t use "real" money on a testnet; it runs on test ether. Consequently, you can’t test security against real adversaries, as there is nothing at stake.
- There are some aspects of a public blockchain that you cannot test realistically on a testnet. For example, transaction fees, although necessary to send transactions, are not a consideration on a testnet, since gas is free. Further, the testnets do not experience network congestion like the public mainnet sometimes does.

Local Blockchain Simulation Advantages and Disadvantages

For many testing purposes, the best option is to launch a single-instance private blockchain. Ganache (formerly named testrpc) is one of the most popular local blockchain simulations that you can interact with, without any other participants. It shares many of the advantages and disadvantages of the public testnet, but also has some differences.

Advantages:

- No syncing and almost no data on disk; you mine the first block yourself
- No need to obtain test ether; Ganache is initialized with accounts that already hold ether for testing
- No other users, just you
- No other contracts, just the ones you deploy after you launch it unless you use the option of forking off an existing Ethereum node

Disadvantages:

- Having no other users means that it doesn't behave the same as a public blockchain. There's no competition for transaction space or sequencing of transactions.
- No miners other than you means that mining is more predictable; therefore, you can't test some scenarios that occur on a public blockchain.
- If you are forking off an existing Ethereum node, it will need to be an archival node for you to interact with state from blocks that may have been pruned otherwise

Running an Ethereum Client

If you have the time and resources, you should attempt to run a full node, even if only to learn more about the process. In this section we cover how to download, compile, and run the Ethereum clients Parity and Geth. This requires some familiarity with using the command-line interface on your operating system. It's worth installing these clients, whether you choose to run them as full nodes, as testnet nodes, or as clients to a local private blockchain.

Hardware Requirements for a Full Node

Before we get started, you should ensure you have a computer with sufficient resources to run an Ethereum full node. You will need at least 300 GB of disk space to store a full copy of the Ethereum blockchain. If you also want to run a full node on the Ethereum testnet, you will need at least an additional 75 GB. Downloading 375 GB of blockchain data can take a long time, so it's recommended that you work on a fast internet connection.

Syncing the Ethereum blockchain is very input/output (I/O) intensive. It is best to have a solid-state drive (SSD). If you have a mechanical hard disk drive (HDD), you will need at least 8 GB of RAM to use as cache. Otherwise, you may discover that your system is too slow to keep up and sync fully.

Minimum requirements:

- CPU with 2+ cores
- At least 300 GB free storage space
- 4 GB RAM minimum with an SSD, 8 GB+ if you have an HDD
- 8 MBit/sec download internet service

These are the minimum requirements to sync a full (but pruned) copy of an Ethereum-based blockchain.

At the time of writing the Parity codebase is lighter on resources, so if you're running with limited hardware you'll likely see better results using Parity.

If you want to sync in a reasonable amount of time and store all the development tools, libraries, clients, and blockchains we discuss in this book, you will want a more capable computer.

Recommended specifications:

- Fast CPU with 4+ cores
- 16 GB+ RAM
- Fast SSD with at least 500 GB free space
- 25+ MBit/sec download internet service

It's difficult to predict how fast a blockchain's size will increase and when more disk space will be required, so it's recommended to check the blockchain's latest size before you start syncing.

NOTE The disk size requirements listed here assume you will be running a node with default settings, where the blockchain is "pruned" of old state data. If you instead run a full "archival" node, where all state is kept on disk, it will likely require more than 1 TB of disk space.

These links provide up-to-date estimates of the blockchain size:

- Ethereum
- Ethereum Classic

Software Requirements for Building and Running a Client (Node)

This section covers Parity and Geth client software. It also assumes you are using a Unix-like command-line environment. The examples show the commands and output as they appear on an Ubuntu GNU/Linux operating system running the bash shell (command-line execution environment).

Typically every blockchain will have its own version of Geth, while Parity provides support for multiple Ethereum-based blockchains (Ethereum, Ethereum Classic, Ellaism, Expanse, Musicoin) with the same client download.

TIP In many of the examples in this chapter, we will be using the operating system's command-line interface (also known as a "shell"), accessed via a "terminal" application. The shell will display a prompt; you type a command, and the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples, it is denoted by a $ symbol. In the examples, when you see text after a $ symbol, don't type the $ symbol but type the command immediately following it (shown in bold), then press Enter to execute the command. In the examples, the lines below each command are the operating system's responses to that command. When you see the next $ prefix, you'll know it's a new command and you should repeat the process.

Before we get started, you may need to install some software. If you've never done any software development on the computer you are currently using, you will probably need to install some basic tools. For the examples that follow, you will need to install git, the source-code management system; golang, the Go programming language and standard libraries; and Rust, a systems programming language.

Git can be installed by following the instructions at https://git-scm.com.

Go can be installed by following the instructions at https://golang.org, or https://github.com/golang/go/wiki/Ubuntu if you are using Ubuntu.

NOTE Geth requirements vary, but if you stick with Go version 1.13 or greater you should be able to compile most versions of Geth. Of course, you should always refer to the documentation for your chosen flavor of Geth.

The version of golang that is installed on your operating system or is available from your system's package manager may be significantly older than 1.13. If so, remove it and install the latest version from https://golang.org/.

Rust can be installed by following the instructions at https://www.rustup.rs/.

NOTE Parity requires Rust version 1.27 or greater.

Parity also requires some software libraries, such as OpenSSL and libudev. To install these on a Ubuntu or Debian GNU/Linux compatible system, use the following command:

```
<pre data-type="programlisting">
$ <strong>sudo apt-get install openssl libssl-dev libudev-dev cmake clang</strong>
</pre>
```

For other operating systems, use the package manager of your OS or follow the Wiki instructions to install the required libraries.

Now that you have git, golang, Rust, and the necessary libraries installed, let's get to work!

Parity

Parity is an implementation of a full-node Ethereum client and DApp browser. It was written "from the ground up” in Rust, a systems programming language, with the aim of building a modular, secure, and scalable Ethereum client. Parity is developed by Parity Tech, a UK company, and is released under the GPLv3 free software license.

NOTE Disclosure: One of the authors of this book, Dr. Gavin Wood, is the founder of Parity Tech and wrote much of the Parity client. Parity represents about 25% of the installed Ethereum client base.

To install Parity, you can use the Rust package manager cargo or download the source code from GitHub. The package manager also downloads the source code, so there’s not much difference between the two options. In the next section, we will show you how to download and compile Parity yourself.

Installing Parity

The Parity Wiki offers instructions for building Parity in different environments and containers. We’ll show you how to build Parity from source. This assumes you have already installed Rust using rustup (see Software Requirements for Building and Running a Client (Node)).

First, get the source code from GitHub:

```
<pre data-type="programlisting">
$ <strong>git clone https://github.com/paritytech/parity</strong>
</pre>
```

Then change to the *parity* directory and use cargo to build the executable:

```
<pre data-type="programlisting">
$ <strong>cd parity</strong>
$ <strong>cargo install --path .</strong>
</pre>
```

If all goes well, you should see something like:

```
<pre data-type="programlisting">
$ <strong>cargo install --path .</strong>
Installing parity-ethereum v2.7.0 (/root/parity)
Updating crates.io index
Updating git repository `https://github.com/paritytech/rust-ctrlc.git`
Updating git repository `https://github.com/paritytech/app-dirs-rs`   Updating git
repository

 [...]

Compiling parity-ethereum v2.7.0 (/root/parity)
Finished release [optimized] target(s) in 10m 16s
Installing /root/.cargo/bin/parity
Installed package `parity-ethereum v2.7.0 (/root/parity)` (executable `parity`)
$
</pre>
```

Try and run parity to see if it is installed, by invoking the --version option:

```
<pre data-type="programlisting">
$ <strong>parity --version</strong>
Parity Ethereum Client.
  version Parity-Ethereum/v2.7.0-unstable-b69a33b3a-20200124/x86_64-unknown-linux-
gnu/rustc1.40.0
Copyright 2015-2020 Parity Technologies (UK) Ltd.
License GPLv3+: GNU GPL version 3 or later <ulink
url="http://gnu.org/licenses/gpl.html">http://gnu.org/licenses/gpl.html</ulink>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

By Wood/Paronyan/Kotewicz/Drwięga/Volf/Greeff
   Habermeier/Czaban/Gotchac/Redman/Nikolsky
   Schoedon/Tang/Adolfsson/Silva/Palm/Hirsz et al.
$
</pre>
```

Great! Now that Parity is installed, you can sync the blockchain and get started with some basic command-line options.

Go-Ethereum (Geth)

Geth is the Go language implementation that is actively developed by the Ethereum Foundation, so is considered the "official" implementation of the Ethereum client. Typically, every Ethereum-based blockchain will have its own Geth implementation. If you're running Geth, then you'll want to make sure you grab the correct version for your blockchain using one of the following repository links:

- Ethereum (or https://geth.ethereum.org/)
- Ethereum Classic
- Ellaism
- Expanse
- Musicoin
- Ubiq

NOTE You can also skip these instructions and install a precompiled binary for your platform of choice. The precompiled releases are much easier to install and can be found in the "releases" section of any of the repositories listed here. However, you may learn more by downloading and compiling the software yourself.

Cloning the repository

The first step is to clone the Git repository, to get a copy of the source code.

To make a local clone of your chosen repository, use the git command as follows, in your home directory or under any directory you use for development:

<pre data-type="programlisting">
$ <strong>git clone &lt;Repository Link&gt;</strong>
</pre>

You should see a progress report as the repository is copied to your local system:

```
Cloning into 'go-ethereum'...
remote: Enumerating objects: 86915, done.
remote: Total 86915 (delta 0), reused 0 (delta 0), pack-reused 86915
Receiving objects: 100% (86915/86915), 134.73 MiB | 29.30 MiB/s, done.
Resolving deltas: 100% (57590/57590), done.
```

Great! Now that you have a local copy of Geth, you can compile an executable for your platform.

Building Geth from source code

To build Geth, change to the directory where the source code was downloaded and use the make command:

<pre data-type="programlisting">
$ <strong>cd go-ethereum</strong>
$ <strong>make geth</strong>
</pre>

If all goes well, you will see the Go compiler building each component until it produces the geth executable:

```
build/env.sh go run build/ci.go install ./cmd/geth
>>> /usr/local/go/bin/go install -ldflags -X main.gitCommit=58a1e13e6dd7f52a1d...
github.com/ethereum/go-ethereum/common/hexutil
github.com/ethereum/go-ethereum/common/math
github.com/ethereum/go-ethereum/crypto/sha3
github.com/ethereum/go-ethereum/rlp
github.com/ethereum/go-ethereum/crypto/secp256k1
github.com/ethereum/go-ethereum/common
[...]
github.com/ethereum/go-ethereum/cmd/utils
github.com/ethereum/go-ethereum/cmd/geth
Done building.
Run "build/bin/geth" to launch geth.
$
```

Let's make sure geth works without actually starting it running:

<pre data-type="programlisting">
$ <strong>./build/bin/geth version</strong>

Geth
Version: 1.9.11-unstable

```
Git Commit: 0b284f6c6cfc6df452ca23f9454ee16a6330cb8e
Git Commit Date: 20200123
Architecture: amd64
Protocol Versions: [64 63]
Go Version: go1.13.4
Operating System: linux
[...]
</pre>
```

Your geth version command may show slightly different information, but you should see a version report much like the one seen here.

The next sections explains the challenge with the initial synchronization of Ethereum's blockchain.

The First Synchronization of Ethereum-Based Blockchains

Traditionally, when syncing an Ethereum blockchain, your client would download and validate every block and every transaction since the very start—i.e., from the genesis block.

While it is possible to fully sync the blockchain this way, this type of sync will take a very long time and has high resource requirements (it will need much more RAM, and will take a very long time indeed if you don't have fast storage).

Many Ethereum-based blockchains were the victim of denial-of-service attacks at the end of 2016. Affected blockchains will tend to sync slowly when doing a full sync.

For example, on Ethereum, a new client will make rapid progress until it reaches block 2,283,397. This block was mined on September 18, 2016, and marks the beginning of the DoS attacks. From this block to block 2,700,031 (November 26, 2016), the validation of transactions becomes extremely slow, memory intensive, and I/O intensive. This results in validation times exceeding 1 minute per block. Ethereum implemented a series of upgrades, using hard forks, to address the underlying vulnerabilities that were exploited in the DoS attacks. These upgrades also cleaned up the blockchain by removing some 20 million empty accounts created by spam transactions.

If you are syncing with full validation, your client will slow down and may take several days, or perhaps even longer, to validate the blocks affected by the DoS attacks.

Fortunately, most Ethereum clients by default now perform a "fast" synchronization that skips the full validation of transactions until it has synced to the tip of the blockchain, then resumes full validation.

Geth performs fast synchronization by default for Ethereum. You may need to refer to the specific instructions for other chosen Ethereum chain.

Parity also does fast synchronization by default.

NOTE Geth can only operate fast synchronization when starting with an empty block database. If you have already started syncing without fast mode, Geth cannot switch. It is faster to delete the blockchain data directory and start fast syncing from the beginning than to continue syncing with full validation. Be careful to not delete any wallets when deleting the blockchain data!

Running Geth or Parity

Now that you understand the challenges of the "first sync," you're ready to start an Ethereum client and sync the blockchain. For both Geth and Parity, you can use the --help option to see all the configuration parameters. The default settings are usually sensible and appropriate for most uses. Choose how to configure any optional parameters to suit your needs, then start Geth or Parity to sync the chain. Then wait...

TIP Syncing the Ethereum blockchain will take anywhere from half a day on a very fast system with lots of RAM, to several days on a slower system.

The JSON-RPC Interface

Ethereum clients offer an application programming interface and a set of Remote Procedure Call (RPC) commands, which are encoded as JavaScript Object Notation (JSON). You will see this referred to as the *JSON-RPC API*. Essentially, the JSON-RPC API is an interface that allows us to write programs that use an Ethereum client as a *gateway* to an Ethereum network and blockchain.

Usually, the RPC interface is offered as an HTTP service on port 8545. For security reasons it is restricted, by default, to only accept connections from localhost (the IP address of your own computer, which is 127.0.0.1).

To access the JSON-RPC API, you can use a specialized library (written in the programming language of your choice) that provides "stub" function calls corresponding to each available RPC command, or you can manually construct HTTP requests and send/receive JSON-encoded requests. You can even use a generic command-line HTTP client, like curl, to call the RPC interface. Let's try that. First, ensure that you have Geth up and running, configured with --rpc to allow HTTP access to the RPC interface, then switch to a new terminal window (e.g., with Ctrl-Shift-N or Ctrl-Shift-T in an existing terminal window) as shown here:

```
<pre data-type="programlisting">
$ <strong>curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
  http://localhost:8545</strong>

{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.9.11-unstable-0b284f6c-20200123/linux-amd64/go1.13.4"}
</pre>
```

In this example, we use curl to make an HTTP connection to the address *http://localhost:8545*. We are already running geth, which offers the JSON-RPC API as an HTTP service on port 8545. We instruct curl to use the HTTP POST command and to identify the content as type application/json. Finally, we pass a JSON-encoded request as the data component of our HTTP request. Most of our command line is just setting up curl to make the HTTP connection correctly. The interesting part is the actual JSON-RPC command we issue:

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}
```

The JSON-RPC request is formatted according to the JSON-RPC 2.0 specification. Each request contains four elements:

jsonrpc
: Version of the JSON-RPC protocol. This MUST be exactly "2.0".

method
: The name of the method to be invoked.

params
: A structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id
: An identifier established by the client that MUST contain a String, Number, or NULL value if included. The server MUST reply with the same value in the response object if included. This member is used to correlate the context between the two objects.

TIP The id parameter is used primarily when you are making multiple requests in a single JSON-RPC call, a practice called *batching*. Batching is used to avoid the overhead of a new HTTP and TCP connection for every request. In the Ethereum context, for example, we would use batching if we wanted to retrieve thousands of transactions over one HTTP connection. When batching, you set a different id for each request and then match it to the id in each response from the JSON-RPC server. The easiest way to implement this is to maintain a counter and increment the value for each request.

The response we receive is:

```
{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.9.11-unstable-0b284f6c-20200123/linux-amd64/go1.13.4"}
```

This tells us that the JSON-RPC API is being served by Geth client version 1.9.11.

Let's try something a bit more interesting. In the next example, we ask the JSON-RPC API for the current price of gas in wei:

```
<pre data-type="programlisting">
$ <strong>curl -X POST -H "Content-Type: application/json" --data \
  '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":4213}' \
  http://localhost:8545</strong>

{"jsonrpc":"2.0","id":4213,"result":"0x430e23400"}
</pre>
```

The response, 0x430e23400, tells us that the current gas price is 18 gwei (gigawei or billion wei). If, like us, you don't think in hexadecimal, you can convert it to decimal on the command line with a little bash-fu:

```
<pre data-type="programlisting">
$ <strong>echo $((0x430e23400))</strong>

18000000000
</pre>
```

The full JSON-RPC API can be investigated on the Ethereum wiki.

Parity's Geth compatibility mode

Parity has a special "Geth compatibility mode," where it offers a JSON-RPC API that is identical to that offered by Geth. To run Parity in this mode, use the --geth switch:

```
<pre data-type="programlisting">
$ <strong>parity --geth</strong>
</pre>
```

Remote Ethereum Clients

Remote clients offer a subset of the functionality of a full client. They do not store the full Ethereum blockchain, so they are faster to set up and require far less data storage.

These clients typically provide the ability to do one or more of the following:

- Manage private keys and Ethereum addresses in a wallet.
- Create, sign, and broadcast transactions.
- Interact with smart contracts, using the data payload.
- Browse and interact with DApps.
- Offer links to external services such as block explorers.
- Convert ether units and retrieve exchange rates from external sources.
- Inject a web3 instance into the web browser as a JavaScript object.
- Use a web3 instance provided/injected into the browser by another client.
- Access RPC services on a local or remote Ethereum node.

Some remote clients, for example mobile (smartphone) wallets, offer only basic wallet functionality. Other remote clients are full-blown DApp browsers. Remote clients commonly offer some of the functions of a full-node Ethereum client without synchronizing a local copy of the Ethereum blockchain by connecting to a full node being run elsewhere, e.g., by you locally on your machine or on a web server, or by a third party on their servers.

Let's look at some of the most popular remote clients and the functions they offer.

Mobile (Smartphone) Wallets

All mobile wallets are remote clients, because smartphones do not have adequate resources to run a full Ethereum client. Light clients are in development and not in general use for Ethereum. In the case of Parity, the light client is marked "experimental" and can be used by running parity with the --light option.

Popular mobile wallets include the following (we list these merely as examples; this is not an endorsement or an indication of the security or functionality of these wallets):

Jaxx
: A multicurrency mobile wallet based on BIP-39 mnemonic seeds, with support for Bitcoin, Litecoin, Ethereum, Ethereum Classic, ZCash, a variety of ERC20 tokens, and many other currencies. Jaxx is available on Android and iOS, as a browser plug-in wallet, and as a desktop wallet for a variety of operating systems.

Status
: A mobile wallet and DApp browser, with support for a variety of tokens and popular DApps. Available for iOS and Android.

Trust Wallet
: A mobile multi-currency wallet that supports Ethereum and Ethereum Classic as well as ERC20 and ERC223 tokens. Trust Wallet is available for iOS and Android.

Cipher Browser
: A full-featured Ethereum-enabled mobile DApp browser and wallet that allows integration with Ethereum apps and tokens. Available for iOS and Android.

Browser Wallets

A variety of wallets and DApp browsers are available as plug-ins or extensions of web browsers such as Chrome and Firefox. These are remote clients that run inside your browser.

Some of the more popular ones are MetaMask, Jaxx, MyEtherWallet, and MyCrypto.

MetaMask

MetaMask,introduced in [intro_chapter], is a versatile browser-based wallet, RPC client, and basic contract explorer. It is available on Chrome, Firefox, Opera, and Brave Browser.

Unlike other browser wallets, MetaMask injects a web3 instance into the browser JavaScript context, acting as an RPC client that connects to a variety of Ethereum blockchains (mainnet, Ropsten testnet, Kovan testnet, local RPC node, etc.). The ability to inject a web3 instance and act as a gateway to external RPC services makes MetaMask a very powerful tool for developers and users alike. It can be combined, for example, with MyEtherWallet or MyCrypto, acting as a web3 provider and RPC gateway for those tools.

Jaxx

Jaxx,which was introduced as a mobile wallet in the previous section, is also available as a Chrome and Firefox extension and as a desktop wallet.

MyEtherWallet (MEW)

MyEtherWalletis a browser-based JavaScript remote client that offers:

- A bridge to popular hardware wallets such as the Trezor and Ledger
- A web3 interface that can connect to a web3 instance injected by another client (e.g., MetaMask)
- An RPC client that can connect to an Ethereum full client
- A basic interface that can interact with smart contracts, given a contract's address and application binary interface (ABI)
- A mobile app, MEWConnect, that enables one to use a compatible Android or iOS device to store funds, similarly to a hardware wallet.
- A software wallet running in JavaScript

WARNING You must be very careful when accessing MyEtherWallet and other browser-based JavaScript wallets, as they are frequent targets for phishing. Always use a bookmark and not a search engine or link to access the correct web URL.

MyCrypto

In early 2018, theMyEtherWallet project split into two competing implementations, guided by two independent development teams: a "fork," as it is called in open source development. The two projects are called MyEtherWallet (the original branding) and MyCrypto. MyCrypto offers almost identical functionality to MyEtherWallet, but instead of using MEWConnect, it offers a connection to the Parity Signer mobile app. Like MEWConnect, Parity Signer stores keys on the phone and interfaces with MyCrypto in a similar manner as a hardware wallet.

Mist (Deprecated)

Mistwas the first Ethereum-enabled browser, built by the Ethereum Foundation. It contained a browser-based wallet that was the first implementation of the ERC20 token standard (Fabian Vogelsteller, author of ERC20, was also the main developer of Mist). Mist was also the first wallet to introduce the camelCase checksum (EIP-55). As of March, 2019, Mist was deprecated and should no longer be used.

Conclusions

In this chapter we explored Ethereum clients. You downloaded, installed, and synchronized a client, becoming a participant in the Ethereum network, and contributing to the health and stability of the system by replicating the blockchain on your own computer.