Building Ethereum ĐApps

Decentralized applications on the Ethereum Blockchain

Roberto Infante

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Building Ethereum ĐApps
Decentralized applications on the Ethereum blockchain
Version 6

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Building Ethereum Đapps*. You will find it easier to progress through the book if you have consolidated programming skills in JavaScript or another object-oriented programming language. But don't worry if you don't know anything about Ethereum, blockchain platforms, or decentralized applications! There are many programmers who, like you, are keen to learn about it, and my objective is to make this learning curve as flat as possible.

After having experimented with Bitcoin a couple of years ago, I started to get interested in Ethereum for the potential it offers, when compared with other blockchain platforms, of providing a Dapp friendly infrastructure. Trying to learn about the technology was not easy. There were no books on Ethereum yet, and the available documentation was limited and fragmented. Most importantly, the technology was evolving quickly, and was not easy to keep up with an environment moving much faster than its documentation.

As the core Ethereum ecosystem became progressively more stable, its documentation started to reflect more closely the latest version of the APIs, and many interesting articles and blogs on Ethereum started to appear. I felt, though, that the documentation and learning resources were even more fragmented than when I started. So I decided to take up the challenge of sharing what I had learned over months of experimenting and building Dapps in one self-contained resource, to make the life easier for new Ethereum adopters.

My goal is to teach you the key technologies you need to become a competent Ethereum Dapp developer. I will do so by helping you build progressively more complex Dapps and highlighting along the way the key concepts for transitioning from conventional to decentralized applications. As you progress through the book, you will learn about the high-level architectural aspects of an Ethereum Đapp, but our focus will be on the tools, infrastructural components, languages, APIs, and development frameworks such as the Ethereum wallet, the geth client, the Solidity language, the Web3 JavaScript API, and Truffle. It is very likely the technology will keep changing while I am writing the book so I will make every effort to keep the content updated and relevant until publication.

Finally, I will try to give you some advice on best practices for designing and securing your Dapp.

Your views and feedback are essential in improving Building Ethereum Đapps, so I strongly encourage you to post any questions, comments, or suggestions you might have to the Author Online Forum. Thanks again for your interest and for purchasing the MEAP!

—Roberto Infante

brief contents

1

A first look at decentralized applications

This chapter covers

- What a decentralized application is
- What a Dapp looks like and how it works
- Dapp terminology
- Suitable and less suitable Dapps

How many times have you found yourself in the following situation? You were browsing around to buy the latest gadget and were comparing prices online, when you came across SmallWebRetailer.com that was offering it 30% cheaper than WellKnown.com. You quickly put the item in the basket, fearing the price would rise at any moment, entered your postal address and credit card details but suddenly… you got cold feet. You started to wonder: is the price too good to be true? What if this unknown SmallWebReailer.com is a scam? Will they run with my money? After a few minutes hesitating on the Buy button, you opened a new browser tab and went straight to WellKnown.com. You submitted the order, aware you might have overpaid your gadget by 30%.

Why did you panic? Perhaps you didn't trust SmallWebRetailer.com, perhaps you didn't want the hassle of having to waste your time contacting the card merchant and possibly wait for a refund, would the transaction turn sour.

What if you could have bought the gadget from the same small unknown retailer through an "alternative e-commerce application" that guaranteed the seller could not access your money until you had confirmed safe delivery of your order? What if such guarantee had not

been provided by the seller (obviously) or by a single third party but by many independent parties participating into a platform designed to process transactions according to conditions encoded in software anyone can inspect? Hold on, probably I have said it too fast. I am going to repeat it more slowly…

1. What if the money transfer was held until delivery not by the retailer or a third party but by many participants to the platform?
2. What if the rules escrowing and then releasing the money transfer where encoded in logic, not subjected to manual interaction?
3. What if, in case you were still unconvinced, you could inspect the code?

I bet you would have clicked the Buy button, confident your funds would have kept safely stored on this platform until the delivery arrived. Such kind of systems do exist and they are called **decentralized applications**. Decentralized marketplaces such as Openbazaar (https://openbazaar.org/), for example, work in this way. The mechanism by which, in our example, funds are routed to the seller only when you have confirmed safe delivery of the goods, is called a *smart contract*.

Decentralized applications, also known as *Decentralized Apps*, or simply as *Dapps* (generally pronounced as dee-apps), are part of a new wave of web applications which are meant to increase the transparency around commercial transactions, governmental processes, supply-chains and all those systems which currently require mutual trust between customer and supplier, user and provider. The objective of decentralized applications is the minimization or elimination of the need of any trust between the participants of a system interaction, with the aim of empowering users beyond what has been delivered by Web 2.0. Some claim decentralized applications could be the backbone of Web 3.0.

Assuming you have programming experience, even better if in JavaScript, and some familiarity with web applications, this book will teach you how to build decentralized applications made of one or more smart contracts controlled by a user interface. By the end of this book you will be able not only to write smart contract code, but to design, implement, test, debug, deploy and secure a full end-to-end decentralized application. Along the way you will also learn a new language, a new platform and, most of all, a new way of thinking, designing and running applications.

In this first chapter, I will give you a high-level overview of decentralized applications. I will explain in detail what they are, how they look like, what technology stack they are built on, when it makes sense to build them and best of all… I will help you start building one straight away!

Let's start our journey!

1.1 What is a Dapp?

Before talking about decentralized applications, let us refresh a concept you are already familiar with, most likely without realizing it: that of a *centralized application*. Probably you

have never heard this expression before because conventional web and enterprise applications are implicitly centralized with respect to their users. I can hear you asking: what does "centralized" mean exactly?

A **centralized application** or system is controlled by a single or *central* entity: an individual, a company, an institution, a governmental agency. The entity hosts the system directly on their premises or through a service or cloud provider and has full control on all the components and layers of the system architecture. The user trusts the good faith of the *central* entity and decides whether to access their system depending on their reputation. In other words, from the point of view of the user, the system is either *trusted* or not. This is how most web and enterprise applications are designed today.

Figure 1.1 illustrates a typical interaction between a user and a *centralized trusted* system. You should not find anything surprising about it.

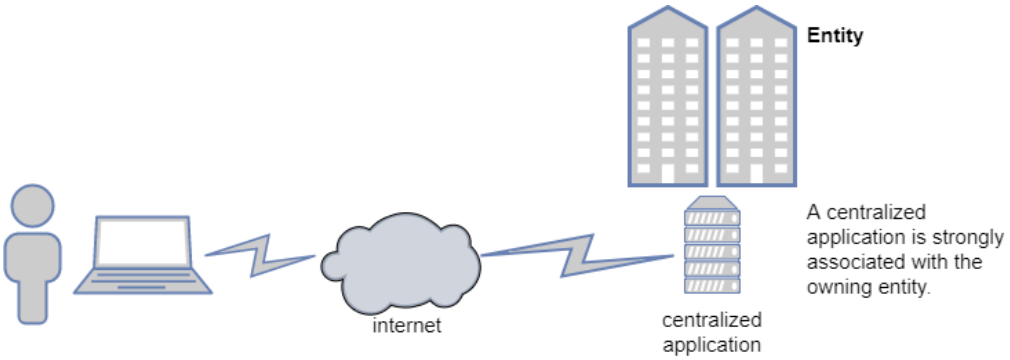


Figure 1.1. A centralized application is strongly associated with the single entity controlling it. Consequently, users decide whether to access it depending on their trust on the entity.

Let us now move on to decentralized applications.

If you think for one moment back at the "alternative e-commerce application" I introduced in the opening paragraph, you would agree it has various advantages with respect to SmallWebRetailer.com:

- *Favorable transaction conditions*. The transaction would be completed, and the money would be fully transferred to the retailer, only when it had complied with all the conditions associated with it, such as your confirmation of safe delivery. This would remove one of the biggest reserves you had with SmallWebRetailer.com: the uncertainty about whether you would get the delivery and what would happen with your money otherwise.
- *Independent transaction execution and verification*. The transaction would not be processed by the retailer or a single third party, but by one of many participants to the platform supporting the e-commerce application and it would then be independently

verified by all the participants of the platform. The mechanism by which all parties would agree on the verification of a transaction is called "**consensus**" (see definition in the callout below). The consensus mechanism would reassure you that the promised transaction conditions are not enforced and verified by an unknown retailer, but by many independent parties.

DEFINITION Consensus is a *distributed* and *trustless* form of agreement on the verification of a transaction. *Distributed* means that the verification of a transaction is not performed by an independent central authority; instead all parties contribute and agree on its verification. *Trustless* means that parties do no need to trust each other to agree on the verification outcome. Consensus is reached when a qualified majority of the participants have agreed on the outcome of the transaction.

- *Transparency*. You would be able to check the code processing the transaction and verify by yourself it would be indeed observing the specified conditions before transferring the money to the retailer. This would give you a further level of reassurance the application will execute under the promised terms.

All these requirements can be delivered by building the "alternative e-commerce application" as a network of processing nodes of equal importance and functionality, each owned by a different party. Each node would:

- Be able to process a transaction in the same other nodes would
- Verify all transactions in the same way other nodes do
- Contribute in equal way to the outcome of a transaction

The consequence of this architecture would be that the processing would be decentralized to a network of independent nodes rather than being centralized to a specific set of servers owned by a specific entity. Such decentralization would relieve the user from having to trust a specific entity: the user would have only to trust the design of the network as a whole.

Applications built on this architecture are known as *decentralized applications*. I am now going to make the concept clearer with another example.

1.1.1 Dapps vs conventional centralized applications

To explain more clearly what the benefit of building a Dapp would be, as opposed to developing a conventional centralized application, I am going to illustrate you a typical use case: an electronic voting application.

CENTRALIZED VOTING APPLICATION

Traditional centralized voting applications are generally provided by a company to facilitate shareholder voting or by a local administration or government to facilitate the approval or selection of law proposals. They are owned directly or indirectly by the institution running it, at least during the voting session.

A centralized voting application runs on one or more application servers connected to a central database. The system is exposed to the voters through one or more web servers hosting the voting website. Web, application and database servers can be hosted directly on the premises of the institution or on the cloud, through a cloud computing provider offering Infrastructure as a Service (IaaS), if the voting system has been implemented in-house by the institution, or through a cloud application provider offering Software as a Service (SaaS), if the voting system is simply leased or rented from an external provider during the voting session. This architecture might not be ideal, from the point of view of the voter, because of potential worries about *trust* and *security*.

TRUST IN CENTRALIZED VOTING

Given all the financial and accounting scandals happened both at corporate and governmental level in the last few years, it is understandable you might not have full trust in the organizations you are shareholder or citizen of. You might wonder whether the outcome of electronic voting might get manipulated in some way.

It is easy to imagine, for example, that a malicious developer or administrator of the voting application, colluding with some party interested in a certain outcome of the voting, could access key parts of the system, and tamper with the way votes are collected, processed and stored at various levels of the application architecture.

Depending on how the application has been designed, it could be possible for some malicious database administrators to even modify votes retroactively.

SECURITY IN CENTRALIZED VOTING

When voting through a centralized application you would not worry only about the good faith of the company or institution organizing the election but also whether the system is secured adequately against external manipulation. External parties, for example, might be interested the voting went in a certain way and might try to get their desired outcome by hacking into the system. A centralized voting system includes, as explained earlier, only a certain number of servers located within the same network. Each server provides generally only one function, and it is therefore a single point of failure not only from a processing point of view but also, and especially, from a security point of view. For example, if a hacker manages to alter code on the web server so that votes are intercepted and modified in that layer, the entire system would be compromised. The same outcome could be achieved by hacking only into the application server or, even better, into the database server. A breach of security in one part of the system is sufficient to compromise the security of the entire system.

DECENTRALIZED VOTING APPLICATION

A decentralized application is based on two key technical principles:

- Its application logic is present and executed simultaneously and independently on each server of a **peer-to-peer (P2P) network**. Each server, also known as a **node**, is

owned in theory by a different participant. Servers are not controlled or coordinated by a central node but communicate directly with each other and are consequently also known as peer nodes. The application data and state are stored on a local copy of a database on each server of the network, as shown in figure 1.2.

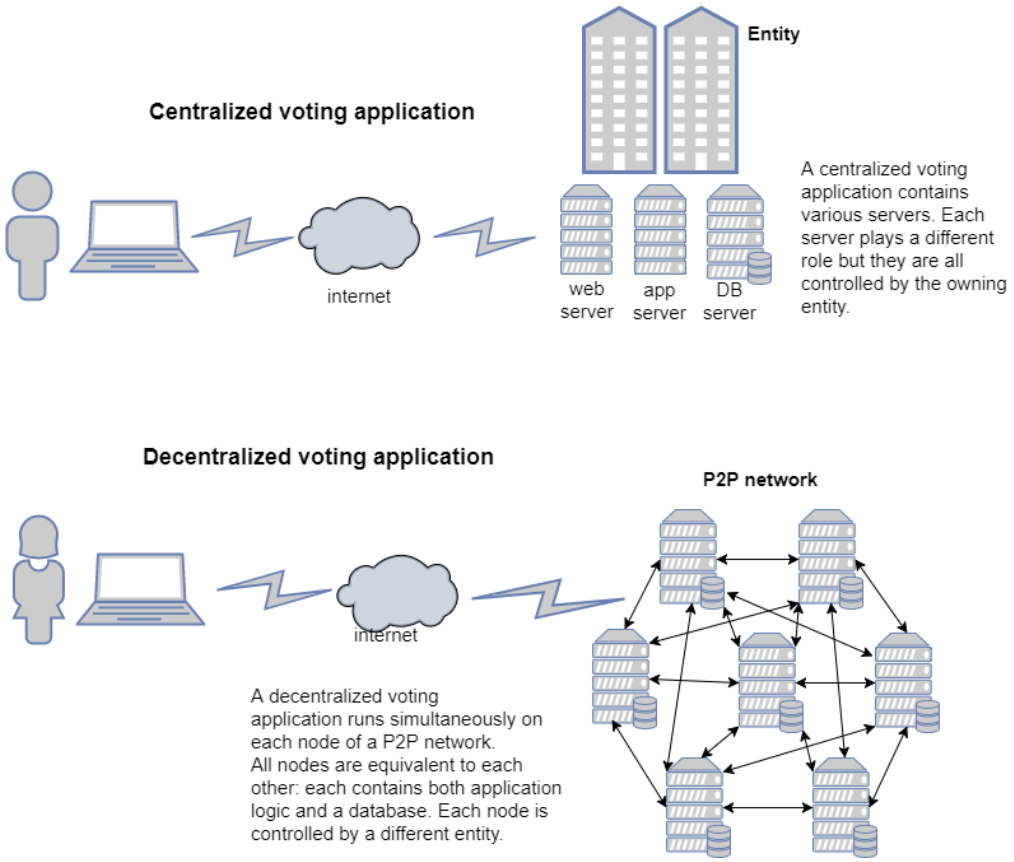


Figure 1.2. A decentralized voting application runs simultaneously on multiple nodes of network. Each node contains and executes the application logic and stores the application state on blockchain database. Different entities own each node and continuously verify each other's output, so that a voter only needs to trust the P2P network, and no individual organization.

- Its database technology, called **blockchain**, guarantees data cannot be modified retroactively.

Let's see how **trust** and **security** concerns can be addressed by decentralizing the voting application according to these two principles.

TRUST AND SECURITY IN DAPPS THANKS TO P2P NETWORK REPLICATION

A decentralized voting application makes trust and security breaches pointless by replicating its execution over a network including many servers, each owned in theory by a different party. Think about it: if votes were processed and verified not by one single server but independently by many servers owned by different parties and they were also stored not on a single database but on many databases, each one local to the processing party, both trust and security concerns would be addressed:

- *Trust* - if one of the participants tried to alter maliciously a vote and propagate the modified vote to the network, the other participants would detect the vote as modified during their validation and would reject it. They would not store it into their local copy of the database and they would not propagate the altered vote further throughout the network, therefore rendering the malicious modification pointless.
- *Security* - hackers would find trying to alter votes on a decentralized system much more difficult than trying to do so in a centralized one. Even if they managed to modify votes on one server, or they hosted themselves one server of the decentralized voting application network to do so more easily, the alteration would be spotted and rejected by other participants, as seen earlier. Successful hacking would therefore require compromising not one server of the network but at least 51% of the nodes of the network simultaneously, assuming the state of the application is what is agreed among the majority of the network nodes. As you can understand, trying to manipulate a large part of a network including thousands of servers is an incredibly challenging task, especially if each one is managed independently: each one might potentially be set up with a different way of preventing security breaches.

TRUST AND SECURITY IN DAPPS THANKS TO THE BLOCKCHAIN

A blockchain database is based on a data structure which, as its name suggests, is a chain of blocks. A block can be seen as a record containing a set of transactions, each one digitally signed, some metadata (such as block number and timestamping information) and a link to the previous block. Each transaction, each block as a whole and the links between blocks are secured with cryptographic technology which makes them *immutable*: retroactive alteration of single transactions is nearly impossible, especially as more subsequent blocks are added to the chain.

A blockchain database therefore addresses *trust* and *security* concerns by providing further protection against manipulation attempts by malicious participants and external parties.

OUTSTANDING QUESTIONS ON LOW-LEVEL ASPECTS

At this stage, you might find the decentralized voting application concept promising from a logical or a high-level point of view, but you might be still confused about physical and lower-level aspects of its architecture. You might have doubts in various areas:

- *System architecture* - is the network hosting the decentralized voting application a special kind of network? Do servers communicate with each other with a special protocol or with standard internet technology?
- *Vote processing and validation* - how does vote submission get propagated across the network so that a vote gets processed on each server of the network? How does a vote get counted and then stored on the blockchain? How does a member of the network verify the authenticity of the consolidated vote records received from other members?

I will try to answer these questions in next two sections, which cover low-level details of the decentralized voting application. I will also assume the voting Dapp has been developed for Ethereum, the blockchain and Dapp platform this book is focused on, which will allow me to start introducing Ethereum and to refer to concrete infrastructural components while presenting two complementary low-level views of the system:

- *A structural Dapp view* — I will describe the low-level architecture of both the client and the server sides of our voting Dapp.
- *A transactional Dapp view* — I will walk you, step-by-step, through the entire lifecycle of a voting transaction.

1.1.2 Structural view: anatomy of a Dapp

The *structural view* of the of the decentralized voting application includes a description of the components of both the client side, represented by the web UI through which a voter submits a vote, and of the server side, represented by a network of servers running the application logic.

DAPP CLIENT-SIDE: A WEB APPLICATION

The voting application web client, shown in figure 1.3, gets initially downloaded into the user browser from a conventional web server, generally as a web application containing HTML and JavaScript. The web user interface does not contain any server-side scripts and communicates directly with a specific server of the network through a client-side JavaScript library called Web3.js. It might also allow the user to communicate with a network node located on their premises. So far there are no major differences with a conventional web application.

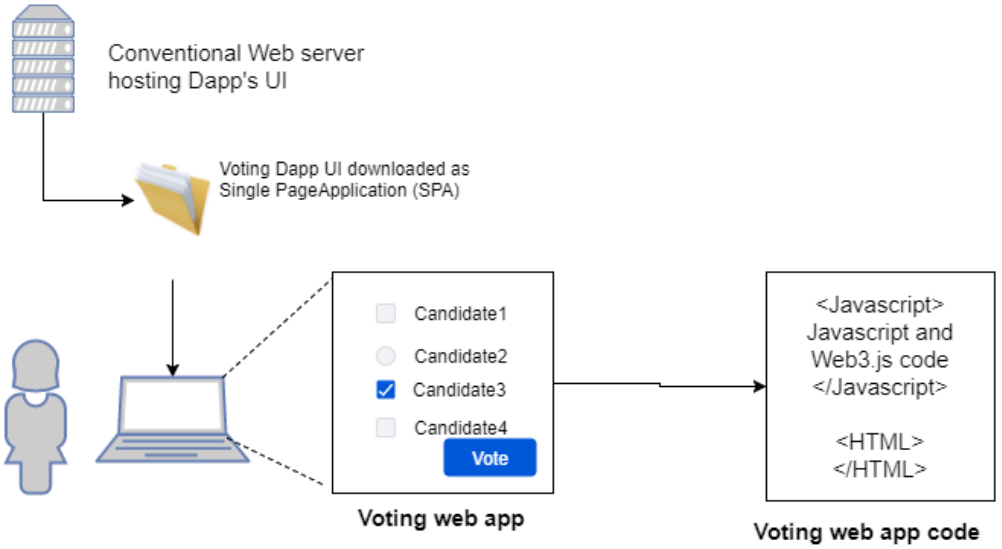


Figure 1.3. A decentralized voting application is exposed to the voter as a web application, which contains both HTML and JavaScript and is downloaded from a conventional web server. The web application, which does not contain any server-side scripts, is generally configured to communicate directly with a specific node of the network. In some cases, the web application might allow the user to point to a server node on their premises.

Let's now move to the server side.

DAPP SERVER-SIDE: A P2P NETWORK

The server-side of a decentralized application is, as repeated various times already, a P2P network of servers which run the same code and have the same copy of a blockchain database.

The interesting characteristic about this network topology is, as you already know, that there is no central coordination, but instead direct communication between each node and a number of other nodes which are considered its peers. As shown in figure 1.4, a node does not need to be connected to all the other nodes of the network.

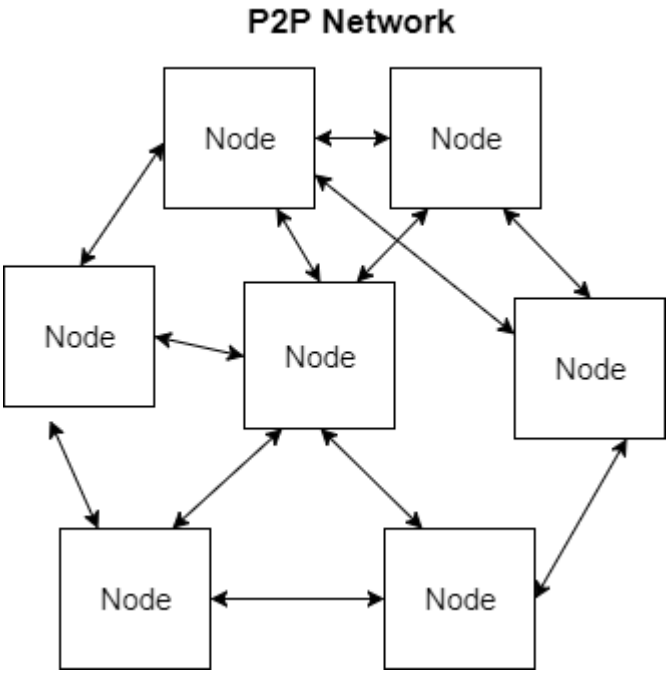


Figure 1.4. A peer-to-peer (P2P) network is made of nodes which communicate directly with each other without the coordination of a master node

THE ETHEREUM NETWORK

Various P2P networks supporting blockchain platforms exist. The most well-known one is the Bitcoin network.

In this book, I will focus on the Ethereum network whose **participant** nodes, as shown in figure 1.5, host a **blockchain database** and a piece of software called node **client** which allows a node to communicate with other nodes. Being all nodes equivalent to each other, there is no clear-cut concept of client and server within the Ethereum network: each node is a server to other nodes but at the same time it is a client of other nodes. This is the reason why the software element of an Ethereum node is called client.

Ethereum clients expose a common client interface and communicate with each other through a P2P protocol called **Wire** which enforces a standard way of sending data throughout the network, specifically a **transaction**, such as a submitted vote, and a **block**, such as a set of votes consolidated on the blockchain database. Various implementations of an Ethereum client exists: as you can see in figure 1.5, they are written in various languages, from C++ to Go, but all implement the standard client interface and the Wire protocol and they are therefore able to interact seamlessly.

The advantage of an Ethereum node client with respect to a Bitcoin node client is that it is able not only to propagate cryptocurrency transactions and blocks throughout the network, as a Bitcoin node does, but also to execute application code hosted on the blockchain database. From this point of view, platforms such as Ethereum are known as *programmable blockchains*. The code of decentralized applications is structured in **smart contracts**, which encapsulate logic and state in the same way classes do in most object-oriented languages. The voting decentralized application, for example, would be structured on various smart contracts which would be hosted on the Ethereum blockchain. I am going to explain shortly what a smart contract is, how you deploy it, how you execute it and also where a smart contract is stored and runs. Bear with me.

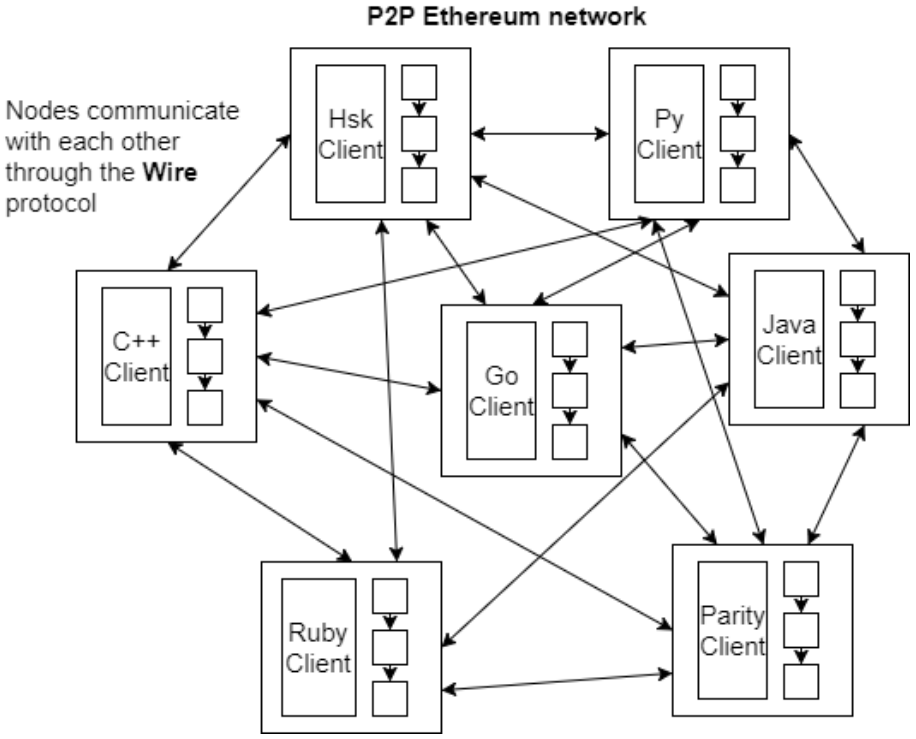


Figure 1.5. Each node of the Ethereum network hosts a blockchain database and a node client capable of executing application code stored on blockchain. Nodes communicate through Wire protocol and expose same interface but can be implemented in different languages.

THE ROLE OF NETWORK NODES

Although all network nodes communicate seamlessly through the common P2P Wire protocol, not all nodes perform the same function.

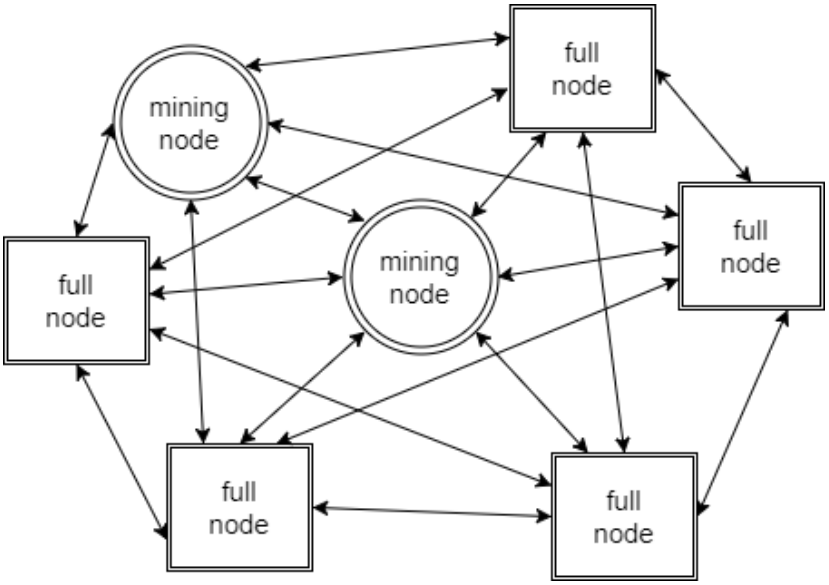


Figure 1.6. The Ethereum network includes two main types of nodes: Full nodes process transactions passively: can read but cannot write on the blockchain; mining nodes process transaction actively: they validate the correctness of transactions as full node do, but additionally they also assemble transactions into new blocks which are appended onto the blockchain.

Broadly, as shown in figure 1.6, there are two main types of nodes which are functionally different:

- *Full node* — Most nodes have a standard setup which allows them to process transactions passively: they can read from the blockchain database but cannot write on it. They do execute transactions but only to verify the correctness of the blockchain blocks they receive from peer nodes. In the case of the voting application, full nodes propagate votes received from their peers to other peers and verify the blocks received are correct and contain authentic votes by running the voting Dapp smart contracts. But full nodes do not store votes into blockchain blocks.
- *Mining node* — Some of the nodes are configured to process transactions actively: they can write into the blockchain. They group and store transactions into new blockchain blocks, and they are rewarded in ether, the cryptocurrency supported in the Ethereum platform, for performing such computationally intensive and energy demanding work. They then propagate these new blocks to the rest of the P2P network. Such nodes are called mining nodes because the process of consolidating a new block to the blockchain and being rewarded for it in cryptocurrency tokens is known as **mining**. In the case of the voting Dapp, mining nodes group votes received from peer nodes into a new block, append the block to the blockchain and propagate the block through their peers.

PUTTING EVERYTHING TOGETHER

You have so far examined the structural view of the voting Dapp. Figure 1.7 shows the entire system, including client and server sides.

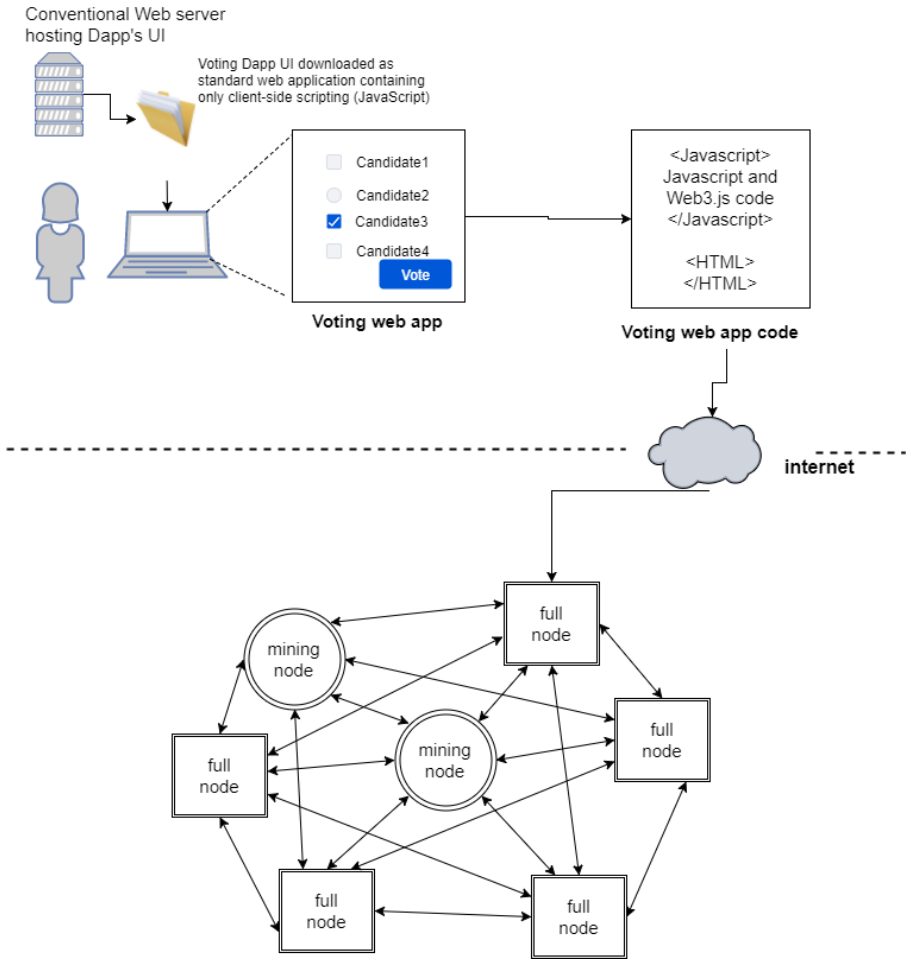


Figure 1.7. Entire static view of a decentralized voting application, including client and server sides

1.1.3 Transactional view: through the lifecycle of a transaction

The full lifecycle of a voting transaction is illustrated step-by-step in figure 1.8, which adds a temporal dimension to the static view you saw earlier in figure 1.7:

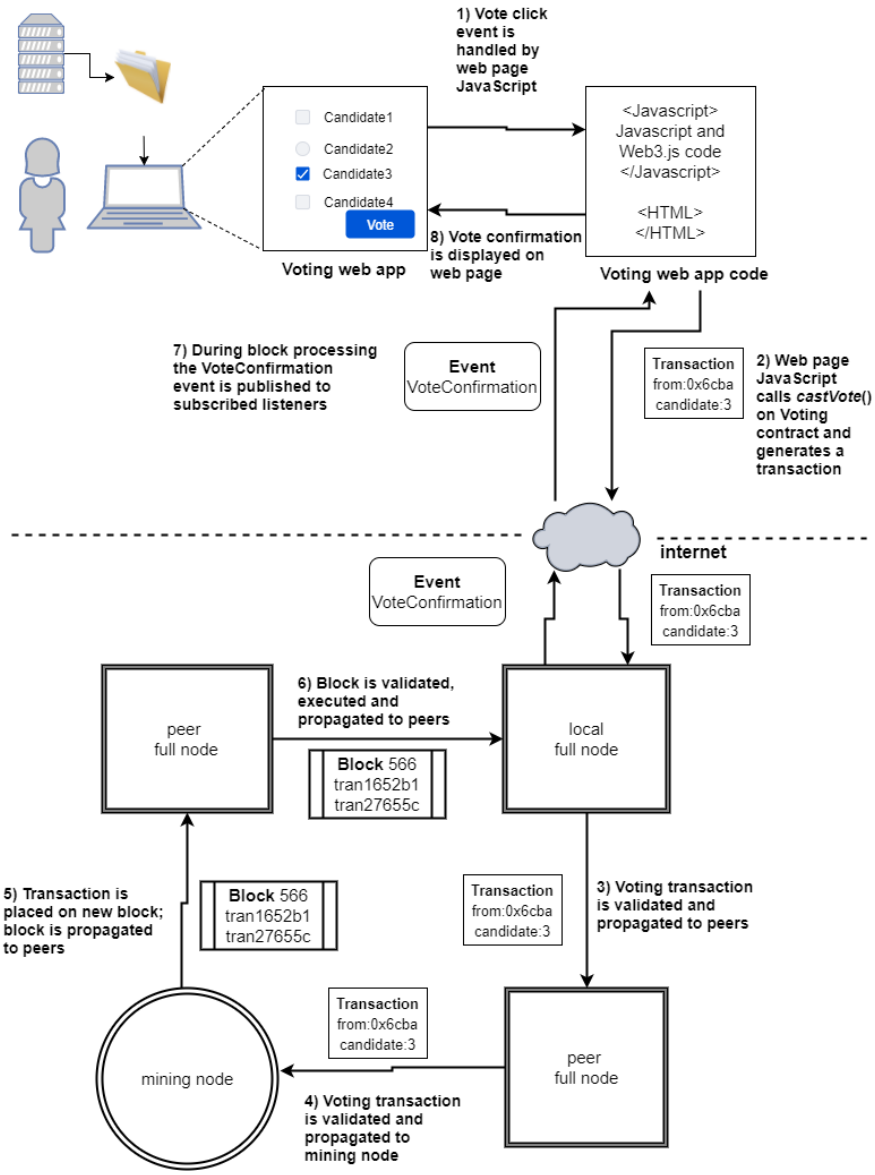


Figure 1.8. Lifecycle of a voting transaction. A voting transaction is created when a voter browser invokes the castVote() function on the Voting smart contract on a local node of the Ethereum network. This is then validated, propagated throughout the network until it is included on a new blockchain block by a mining node. The new block is then propagated throughout the network and finally it gets back to the local node.

1. The user selects one of the possible voting options from a dropdown list box on the web client and then clicks the Vote button.
2. The click event is handled by a JavaScript function which grabs the voting selection, and then, through various web3.js library functions, it sets up the communication with a configured Ethereum node, it connects with the voting smart contract and calls the `castVote()` function on it. The invocation of `castVote()` generates a transaction message which is digitally signed against the account of the user, to prove they are the genuine sender.
3. The contacted local Ethereum node handles the transaction message, it verifies it and it relays it to its peer nodes.
4. Peer nodes do the same and keep propagating the transaction, until the transaction hits mining nodes; this will happen relatively soon, depending on the ratio between the number of full nodes and mining nodes. Mining nodes perform the same steps (2 and 3 described above) of a full node. In addition, a mining node picks a transaction, such as a voting transaction. A transaction is considered profitable if it is expected to generate an acceptable transaction fee, higher than the electricity costs the mining node faces while processing transactions. If so, the mining node executes the `castVote()` function and competes with other mining nodes to store the transaction on the blockchain. The winning mining node (which is the mining node solving successfully the so-called consensus algorithm, a cryptographic problem) consolidates the voting transaction among other transactions onto a ***new block*** of the blockchain. It then relays the new block to all its peer nodes (regardless of whether they are full or mining nodes).
5. Each node which has received a new block verifies the individual transactions included in it are genuine and that the block as a whole is valid. It then processes all the transactions present on it. While doing this, it implicitly verifies the validity of the contract state. For example, the vote submission logic might include an invariant verifying that the number of votes cast to a candidate is not higher than the number of registered voters; or that the total number of votes cast to all candidates is also not higher than the number of registered voters. If the node verifies the block successfully it relays it to its peer nodes; these perform the same validation and propagation action until the whole network has acquired the new blockchain block. The verification process will become clearer to you in the next chapter, when I will present you the cryptographic techniques on which it is based.
6. The local Ethereum node with respect to the user receives the new block and it verifies it by executing all the transactions present on it, as all the other nodes have done. One of these is the voting transaction which, on successful completion has been programmed to raise a `VoteConfirmation` event. The event is published to all the clients subscribed to it, among which is one is the Dapp web UI.
7. The JavaScript code present on the voting web client contains a callback function registered against the `VoteConfirmation` event which then gets triggered.

8. Finally, the callback function shows a vote confirmation notification on the voter's screen.

1.1.4 Let's get familiar with some Dapp terminology

Although Decentralized applications are a relatively new idea, standard terminology around them started to appear relatively soon after the first Dapps were built. In this section, I am providing a summary of the key terms that were described by Vitalik Buterin, the creator of Ethereum, in a famous blog post[1] he wrote to explain key Dapp concepts. You have already come across some of these terms in the previous sections, but I am going to define them more precisely.

SMART CONTRACT

A *smart contract* is an arrangement between two or more parties which involves exchange of digital assets. One or more of these parties allocate digital assets onto the contract at its initiation. Subsequently the assets are redistributed among the parties according to a predefined protocol encoded in logic and a state which is initialized at the start of the contract.

AUTONOMOUS AGENT

An *autonomous agent* is a software entity which interacts autonomously with external software services and can reconfigure or even reprogram itself following verified changes in the external environment.

DECENTRALIZED ORGANIZATION

A traditional *centralized organization* contains assets and different classes of individuals, typically investors, employees and customers. Investors control the organization by owning a part of it through the purchase of shares. Interactions between some classes of individuals are influenced by whether they control the organization. For instance, employees can get recruited by investors or by employees authorized directly or indirectly by investors.

A *Decentralized Organization (DO)* is not controlled by anyone or by any entity. Interactions between classes of individuals involved in the organization are determined purely by predefined protocols. Such protocols, on the other hand, can be designed so that certain individuals have more powers than others, for instance depending on the number of shares owned, exactly as in the case of centralized organizations.

[1] Vitalik Buterin, "DAOs, DACs, DAs and More: An Incomplete Terminology Guide", https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide/

DECENTRALIZED AUTONOMOUS ORGANIZATION

A *Decentralized Autonomous Organization (DAO)* is both a DO and an autonomous agent: it is, like an autonomous agent, a software entity which interacts autonomously with external software services. Individuals involved with the DAO interact, like in DOs, through predefined protocols.

The main difference between a DAO and a DO is that interactions between DAOs and external parties are largely automated, and the interaction protocols are programmed in a *smart contract*, whereas interactions between the individuals who own the DO and external parties are subject only to a "manual protocol". The key point is that, from the point of view of external parties, DAOs are more trustworthy than DOs since automated interactions are predictable whereas interactions based on a manual protocol rely entirely on the reputation of the individuals following it.

According to these definitions, there are diverging opinions on whether blockchain platforms built with the main or only purpose to support a cryptocurrency can be classified as DAO or DO. Since the Bitcoin infrastructure does not allow to implement easily automated interaction protocols, some think it should be classified as a DO.

DECENTRALIZED AUTONOMOUS CORPORATION

A *Decentralized Autonomous Corporation (DAC)* is a DAO which can be partially owned through purchase of shares. As for classic (centralized) corporations, a DAC redistributes dividends periodically, depending on its financial success. A pure DAO, on the other hand, is generally a non-profit organization, and participants benefit economically exclusively by contributing to its ecosystem and increasing its internal capital.

> **NOTE** The current widely accepted definition of *decentralized application* corresponds to that of DAO described above, which is still in use among Ethereum purists. As far as we are concerned, this is what we will mean for Dapp for the rest of this book. The reason why the initial terminology used the word "*organization*" rather than "*application*" was because the Ethereum founders wanted to put emphasis on the fact a decentralized application can transact with other parties exactly like conventional organizations: by following rules and protocols and exchanging monetary value, obviously in the form of cryptocurrency rather than conventional currency.

The key aspects of each of these terms are summarized in table 1.2.

Table 1.2. Matrix summarizing key aspects of each term. DAO is what we mean for Dapp.

	Is software	Has capital	Is autonomous	Is it owned
Autonom agent	YES	NO	YES	NO
DO	NO	YES	NO	YES

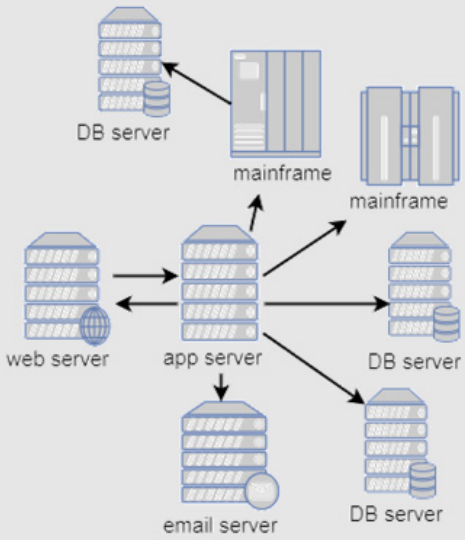
DAO	**YES**	**YES**	**YES**	**NO**
DAC	YES	YES	YES	YES

Although you have just acquired some high-level terminology, you cannot truly understand the purpose of Dapps and how they work without familiarizing with a concept you have already come across a few times in this book and it is now overdue a proper explanation: that of blockchain. Because Dapps are built on top of the blockchain and they rely heavily on it, you should learn about it and its underlying technologies. This is what we are going to cover in the next section.

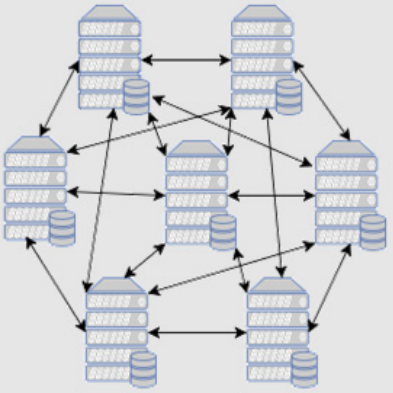
Decentralized vs distributed applications

Decentralized applications should not be confused with distributed applications. The two concepts are orthogonal to each other.

An application is distributed if it runs over multiple servers within a network. The simplest example of a distributed application is a web application, which is typically distributed over a web server, an application server and a database server, and possibly an email server and legacy mainframes. The centralized voting application seen earlier is an example of distributed application.



Distributed application



Decentralized application

A distributed application runs over multiple servers of a network. A decentralized application is replicated in its entirety over each node of a wide network.

An application is decentralized if it is replicated in its entirety over each node of a network, with each node being theoretically owned by a different entity. The higher the number of entities owning nodes of the network, the more

trustful the network in its entirety is. Obviously, networks owned only by a few owners cannot be considered trustful because they do not truly decentralize the processing.

A centralized application is generally distributed, but decentralized applications can also be distributed over multiple servers within each logical node.

1.2 Good and bad Dapps

In the last few years many Dapps have been developed. Some have received various rounds of venture capitalist funding and have been deployed successfully into production. Others have failed to convince investors and users and never passed the proof of concept stage.

1.2.1 Good use cases

Given the novelty of the technology, it is hard to predict what is going to work and what is going to fail. Nevertheless, various use cases are being widely recognized as a good fit for the blockchain and in particular for Ethereum. Ideal Dapps are those that leverage the main benefits of these technologies, specifically record immutability, decentralization, security, resilience. The main fields likely to be revolutionized by Dapps are therefore: provenance and ownership tracking, authenticity tracking, identity verification, regulatory auditing, charity expense auditing, prediction markets, customer loyalty management, crowdfunding, electronic voting, investing, gambling, lending, online games management, author royalty payment, Internet of Things, cloud computing and even freedom of speech. Let's see what innovative solutions have already been found in some of these areas.

PROVENANCE AND AUTHENTICITY TRACKING

One of the biggest problems affecting supply chain management, particularly that involving long chains of processed goods crossing several countries, is the tracking of the authenticity of materials. Here are some Dapps that are innovating in this area with blockchain based solutions:

- Dapps such as ***Provenance*** provide blockchain-based provenance tracking of materials, to ensure no information is lost or manipulated within the supplier chain and goods of expected quality reach the end customer. One of the first applications built on Provenance has been focused on the food industry, to track the supply chain of ingredients from the point of collection, though the process of food manufacturing, to the final point of consumer sale. The aim of this system is to prove the food being sold has the claimed characteristics advertised to the consumers, such location and sustainability of harvesting or breeding, whether the sources are organic or have been genetically modified, whether they are coming from fair trade, and so forth.
- Unilever, the multinational consumer goods corporate, is developing a blockchain based system in collaboration with a number of start-ups, to track the tea supply chain starting from the farmers in Malawi.

- ***Everledger*** is a Dapp which aims at replacing the paper certification process of diamonds with a blockchain-based system. A full digital record of a diamond, including its certificate id and many properties such as cut grade, clarity, color and carat are stored on the blockchain and the certificate id is then engraved with a laser on the stone. All the information related with a diamond can then be retrieved at any point of the supply chain with the help of a scanner which reads the certificate id from the stone. Almost 2 million diamonds have already been stored on Everledger.
- The pharmaceutical company Pfizer is partnering with the biotechnology company Genentech to develop ***Mediledger***, a blockchain-based drug delivery tracking system. The aim is to verify provenance and authenticity of Pfizer drug deliveries throughout the entire distribution chain, in order to prevent thefts, fraud and counterfeiting.

IDENTITY VERIFICATION

As for provenance tracking, verification of proof of identity tries to protect businesses and individuals from the consequences of fraud and identity theft. ***KYC-Chain*** is a novel platform built on the Ethereum blockchain, that allows users to manage their digital identity securely, and helps businesses and financial institutions to manage customer data in a reliable and easy manner. The system is designed so that users own the "keys" to their personal data and identity certificates. Consequently, identity owners, who can be individuals or companies, are the only ones who get to choose which part of their information is to be shared, with whom and under what terms. Such information is digitally attested by notaries and institutions before being shared by owners and registered agents.

PROVING OWNERSHIP

Traditional blockchains associated with cryptocurrencies such as Bitcoin can be seen as ledgers which implicitly proof the ownership of digital assets such as the amount of Bitcoin stored at a certain address. Only the legitimate owners of the address are able to transfer funds because they are the only ones to know the private key.

TrustToken tries to go further: it is a Dapp conceived for proving the ownership of physical assets ranging from real estate, to financial assets such as stocks and bonds, commodities such as gold and even intellectual property such as music, books and patents, through smart contracts. The idea is the ownership of physical assets can be transferred from one person to the other in the same way Bitcoins are transferred between addresses. The underlying assumption for TrustToken to be successful is that proof of ownership recorded through the system should be enforceable under law.

ECONOMY OF THINGS

The tech startup Slock.it is building the infrastructure for the 'economy of things', which lies at the intersection between the Internet of Things and blockchain technology. This infrastructure, which has been named ***Universal Sharing Network***, has the potential to be used as a

financial internet, where connected autonomous objects can not only sell and rent themselves, but also pay for each other services. The technology being developed, based on Ethereum smart contracts, aims at providing autonomous objects an identity, the ability to receive payments and enter into agreements without the need of intermediaries. Smart lockers, which enable the unlocking of physical objects upon payment of a fee, are some of the applications already created on these platforms. Because smart lockers make renting of sport equipment, hotel rooms, bicycles and offices easy, this solution is thought to provide the foundation for the "sharing economy".

DECENTRALIZED PREDICTION MARKETS

Prediction markets reward people for correctly predicting real-world events such as the winner of a presidential election, the outcome of a referendum, the level of interest rates at a specific date, the winner of a sport competition. Aside of being speculative markets, they are useful tools for economists, public administration planners and corporate strategists, who can base their decisions on the event probabilities being currently traded, which are thought to reflect the "wisdom of the crowds".

Although centralized markets such as predictit.org exist, several decentralized initiatives are starting to emerge. ***Augur*** is a decentralized market prediction platform built on Ethereum. The idea is that decentralization brings the following benefits:

- Being based on the Ethereum network, it has no central point of failure, so it is inherently highly available
- Nobody controls the definition of markets: anyone can start a new market on a new prediction and can get rewarded for having created that market.
- The official outcome of each prediction is not decided centrally but it is crowdsourced from market participants, so it is less likely to be subject to manipulation
- Funds are stored on the blockchain, which eliminates counterparty risk, makes payment to prediction winner fast, and reduces the likelihood of errors.

INTERNATIONAL TRADE FINANCE

International trade between a supplier and a manufacturer placed in different countries is a complex business because, as you can see in figure 1.9, it is generally based on a complicated workflow involving many parties such as banks that facilitate the payment, commercial intermediaries that facilitate the distribution, shipping and delivery companies that transport the goods, insurers that cover financially risks while the goods are in transit, custom officials that check the legality of the goods and the payment of import duties.

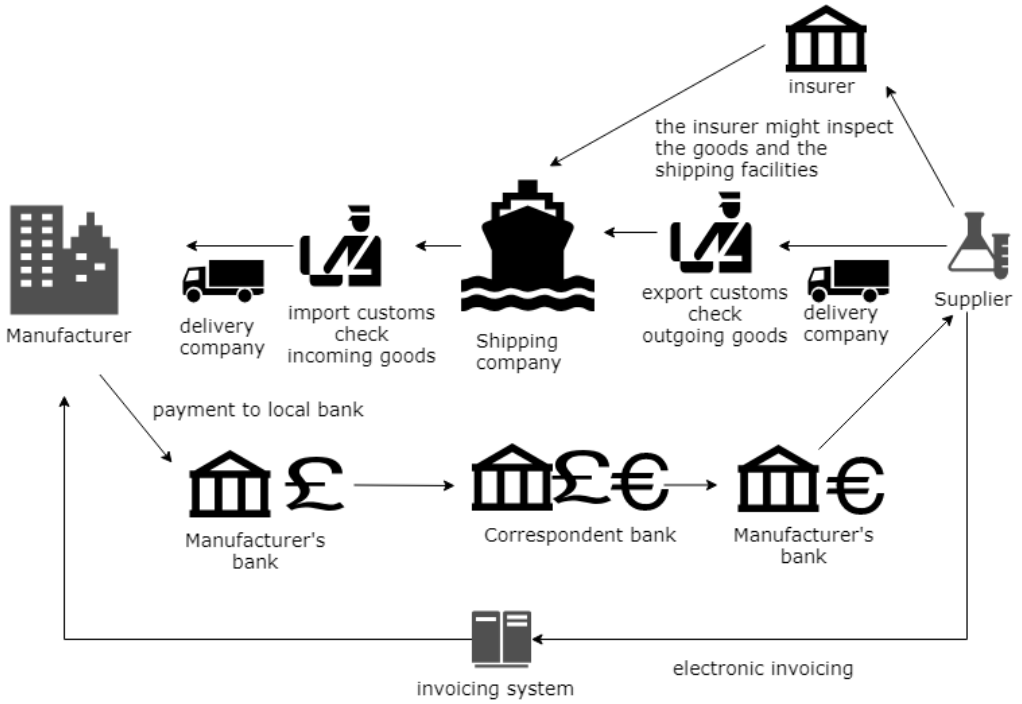


Figure 1.9 A typical international trade involves many parties: banks, commercial intermediaries, shipping companies, banks, insurers, custom officials. In fact this is a very simplified example of what happens in reality, just to give you an idea!

Parties involved in a specific transaction often have never dealt with each other previously. However, for the transaction to complete successfully, they must effectively communicate with each other, generally through established lengthy protocols generally designed to protect a party against malicious behaviour of another party. Parties cross check each other and this takes a huge amount of paperwork and time, which often cause long delays. **we.trade** is a platform sponsored by a consortium of banking partners that aims at simplifying and streamlining such processes with the help of blockchain technology. The platform tracks each step of the transaction openly and transparently so that each party is able to submit and consume the relevant documentation with the confidence that this will not be tampered with by anyone. Trades that used to take weeks can be now completed in a few days.

REGULATORY AUDITING

The blockchain is particularly suitable for ensuring records stored on it have not been altered or tampered with. ***Balanc3*** is a Dapp built on Ethereum that ensures the integrity of accountancy records for regulatory purposes.

CROWDFUNDING

WeiFund aims at providing open source modular and extensible decentralized crowdfunding utilities based on the Ethereum blockchain. Users will be able to setup and manage crowdfunding campaigns through these utilities.

The possibility of encoding funding rules based on smart contract technology will allow users to know precisely what will happen with their money in case the campaign failed or was successful.

GAMBLING

Intuitively, a natural fit for a decentralized application is a gambling platform, because users get the benefit of being reassured bets are processed fairly and predictably. *Edgeless* is an example of such a platform, and it is currently being developed after a successful crowdfunding campaign.

Now that you have learned about some successful Dapp implementations, you might be wondering whether it is always worth basing your application on blockchain technology. This is what we will explore in the next section.

1.2.2 Pointless Dapps

Deciding whether the blockchain is a suitable technology for an application you are planning to build might be difficult to determine. What you should ask yourself is whether your business requirements are going to be met by the functionality offered by a blockchain platform but also, and especially, whether the benefits of using such platform are going to be outweighed by all the technical limitations and additional complexities which come with this technology. A sobering blogpost titled *Avoiding the pointless blockchain project*[2] analyzes the requirements necessary to justify the use of a blockchain platform over more traditional technologies such a SQL or noSQL databases. It concludes that a blockchain project only makes sense if all the following questions are answered positively:

- Does your application require a shared database?
- Does the database need to support multiple writing parties?
- Have the writing parties got no trust in each other?
- Do the writing parties need to modify the state of the database directly, without recurring to a central entity trusted by all participants?
- Do transactions created by the writing parties interact collaboratively with each other?

According to these criteria, for example, an internal enterprise application which was not exposing any data to external parties, would not be a suitable choice for a Dapp.

[2] *Avoiding the pointless blockchain project, Gideon Greenspan,* http://www.multichain.com/blog/2015/11/avoiding-pointless-blockchain-project/

Other poor candidate Dapps are applications for which confidentiality around the business rules is important. A smart contract is, by definition, completely open and transparent to all interacting parties. Therefore, preventing participants from accessing and understanding the logic of the rules would defeat the purpose.

Although decentralized microblogging applications such as *EthTweet*, are considered sensible Dapps to those who value the fact messages cannot be censored and altered after having been sent, an instant messaging Dapp such as a "decentralized WhatsApp" would not be a particularly useful product for a fundamental reason. One of the technical downsides of the a blockchain platform is that processing transactions (in this case instant messages) takes roughly the 15 seconds needed to consolidate a new blockchain block. Therefore, messages could never be *instant* at all.

When building a Dapp you should also keep in mind some operational aspects, which, given the novelty of the technology, my cause some issues down the road. For instance, although a smart contract can automatically guarantee funds are routed and released subject to certain conditions, a commercial transaction might be also subject to "real-world" conditions which cannot be enforced by programming logic. A classic example for a non-fully automatically enforceable smart contract is that of an electronic loan. If the borrower had to keep the borrowed money stuck on a blockchain account, so that a smart contract could automatically give it back to the lender if the borrower missed an interest payment, the borrowing would not make any economic sense. In these cases, it is not clear yet whether a court of law would be able to enforce the non-automated elements of a smart contract or whether it would be necessary to complement the deal with a traditional legal arrangement.

1.3 A five-minute Dapp implementation

By now you should have a good understanding of what a Dapp is, of the purpose of Dapps over conventional apps, of the main architectural components of a decentralized application and of whether it makes sense to embark on a project based on blockchain technologies. It is now time to get one little step further and finally get on with some programming. In the rest of the chapter we'll start building the smart contract of custom cryptocurrency. We'll then activate it and interact with it.

1.3.1 Let's start building SimpleCoin, a basic cryptocurrency

Most Dapps are designed on functionality based on the exchange of cryptocurrency or tokens through rules encoded in one or more smart contracts. You will start to get a feel for Dapps programming by building SimpleCoin, a basic cryptocurrency which will present useful preliminary concepts about smart contracts and the Ethereum platform. You will progressively build upon it in the following chapters, where you will learn more about Dapp development. You will also use or reference SimpleCoin from other Dapps that you will build in later chapters.

Since you have not installed yet any client of the Ethereum platform on your computer, you will be writing code on the Remix Solidity IDE (previously known as Browser Solidity) for now. This is an online tool which will allow you to implement smart contracts in a high-level language called Solidity, very similar to JavaScript, and run them on a local JavaScript VM that emulates the Ethereum Virtual Machine that you will meet in the next chapter. It is also possible, through this tool, to interact with real smart contracts deployed on the Ethereum network.

Open a web browser and go to: http://remix.ethereum.org/

You should see a screen like in figure 1.10.

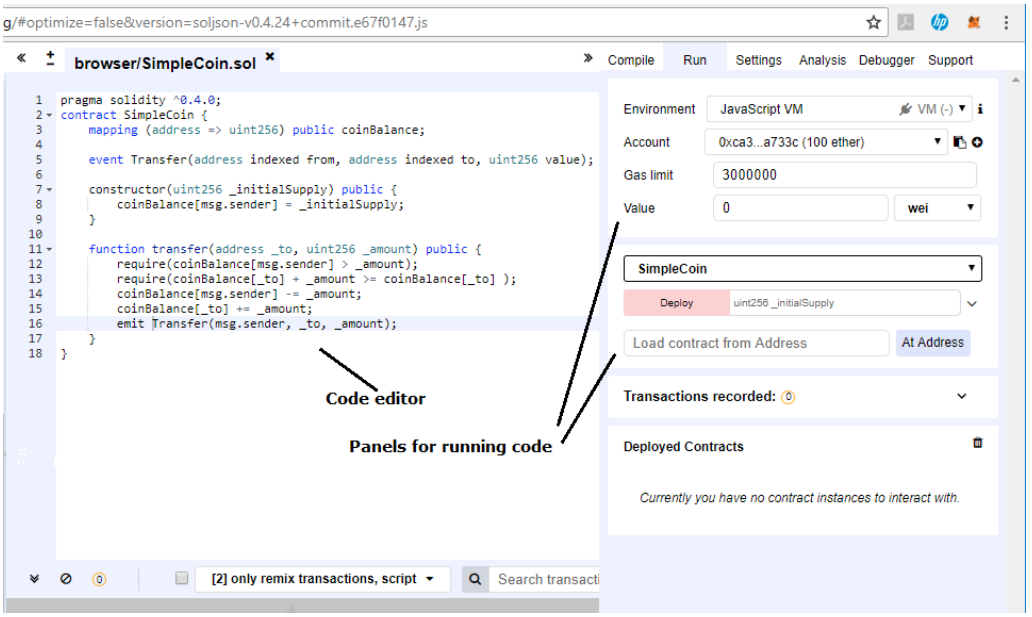


Figure 1.10 Screenshot of the Remix opening screen. You can see the file explorer on the left, the code editor in the middle and the code execution panels on the right.

The left-hand side of the IDE is a file explorer; in the middle you have the code editor; the right-hand hand side contains various panels to run the code and interact with it.

In your first encounter with Solidity you will implement the simplest possible smart contract. If you think of a smart contract as the equivalent of a class in an object-oriented language, you will write a single class with only one-member field, one constructor and one method. Then you will run it and interact with it.

Enter the code of the following listing in the Remix editor, on the left side of the screen.

Listing 1.1 First implementation of SimpleCoin, a basic cryptocurrency

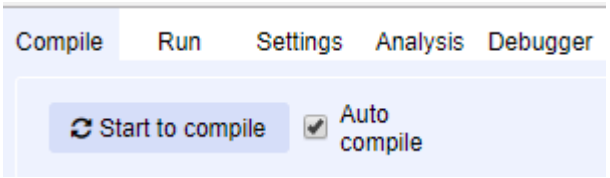
```
pragma solidity ^0.4.0;    //#A

contract SimpleCoin {    //#B

  mapping (address => uint256) public coinBalance;   //#C

  constructor() public {   //#D
    coinBalance[0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C] = 10000;   //#E
  }

  function transfer(address _to, uint256 _amount) public {   //#F
    coinBalance[msg.sender] -= _amount;   //#G
    coinBalance[_to] += _amount;   //#H
  }
}
```

#A Pragma directive indicating the supported version of the Solidity compiler. The code supports a compiler later than 0.4.0 but earlier than 0.5.0.
#B Defines a contract, which is very similar to a class in other languages.
#C Defines a state variable as a "mapping" between an address and an integer. A state variable is the equivalent of a member variable. A mapping is the equivalent to a hash table or hash map.
#D Starts defining the contract constructor.
#E Assigns 10000 SimpleCoin tokens to the coin account with address 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c at contract creation.
#F Defines a function which moves a number of SimpleCoin tokens from the coin account of the function caller to a specified coin account.
#G Decreases the coin balance of the message sender by the specified number of tokens. The special implicit property msg.sender represents the address of the sender of the transaction.
#H Increases the coin balance by the specified number of tokens.

Let's examine this code in detail. A *contract*, in Solidity, is a type very similar to a class in any other language: it has *state variables* (such as `coinBalance`), a constructor (such as function `SimpleCoin`), functions (such as `transfer`) and events.

The `coinBalance` state variable is defined as a `mapping`. A *mapping* is a hash map, equivalent to a hashMap in Java, Dictionary in C# or dict in Python. In this example, the type of the key is an `address`, whereas the value is a `uint256`, an unsigned 256-bit integer. An *address* holds a 20-byte value and it can identify a specific smart contract account or a specific user account. An account, as we will see later in detail, is the sender or the receiver of a transaction. `coinBalance` therefore represents a collection of coin accounts, each holding a number of SimpleCoin tokens.

The `transfer` function is meant to move a number of SimpleCoin tokens from the coin account of the function caller to a specified coin account. In smart contract terminology, a function caller is the *transaction sender*. `msg` is a special implicitly defined variable that represents the incoming message. It has various properties, among which `msg.sender` represents the address of the sender of the transaction, who is the caller of `transfer`. The body of the transfer function is simple to understand. It is simply subtracting the specified amount from the cash account associated with the function caller and adding the amount



specified in the `_amount` parameter to the account associated with the address specified in the _to parameter. To keep this initial code simple, this implementation is not performing yet any boundary checks on the number of SimpleCoin tokens owned by the transaction sender, who, for example, should not be allowed to send more tokens than the owned ones. You will perform such checks when we will revisit SimpleCoin in later chapters.

At this point you should understand our SimpleCoin "contract" is, in practice, a class with a constructor (`SimpleCoin` function), some state (`coinBalance` variable) and a method (`transfer` function). Table 1.3. gives a quick summary of the Solidity keywords you have just come across.

Table 1.3. A summary of Solidity keywords met in the first code sample

Keyword	Explanation
contract	Type similar to class in any other language
mapping	Data structure similar to a hash table or hash map
address	20-byte value representing an Ethereum user account or contract account
uint256	Unsigned 256-bit integer
msg	Special variable representing an incoming message object
msg.sender	Property of the msg object representing the address of the message sender

1.3.2 Let's run the contract

Move now to the right-hand side of the screen to deploy the SimpleCoin contract.

First of all, make sure the Auto Compile option in the Compile tab is ticked, as shown in figure 1.11, so that Remix will recompile the code at every change.

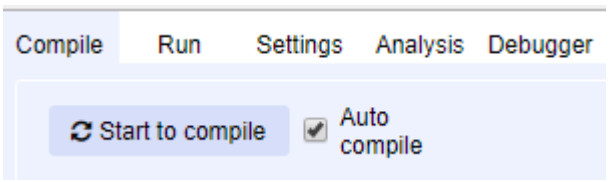


Figure 1.11 The auto-compile option in the Compile tab makes sure the code entered in the editor is recompiled at every change

If you have typed your code correctly (you can copy the code from the files provided on the book website, if you prefer!), and there are no compilation errors, you should see the following buttons in the *Run* tab: *Deploy* and *At Address*, as shown in figure 1.12. Ignore for the moment At Address. Focus your attention on *Deploy*, for the moment. By clicking this button, you will deploy the SimpleCoin contract on an emulated blockchain within Remix.

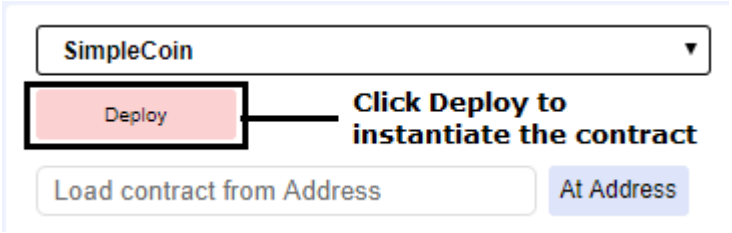


Figure 1.12 Once the code has been compiled correctly, the Run tab will show two buttons: Deploy and At Address. You can instantiate the contract by clicking *Deploy*.

The contract will be stored against an address on the emulated Ethereum blockchain and a new *Deployed Contracts* panel will appear, as shown in figure 1.13. You can read the deployment address by clicking the *copy address* icon and pasting it on Notepad, for example.

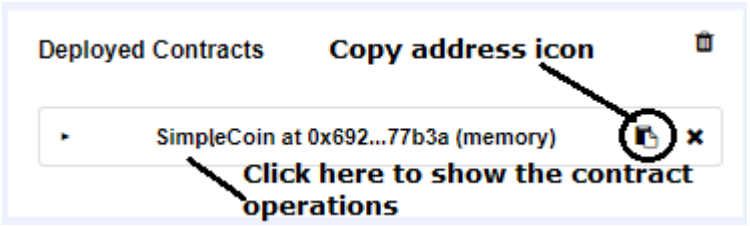


Figure 1.13 After deploying the contract the Deployed Contracts panel appears; it contains a dropdown with a SimpleCoin option; click it and you will see the contract operations

1.3.3 Let's interact with the contract

Now that the SimpleCoin contract has been deployed, you will be able to perform simple operations against it: you will check SimpleCoin token balances and move tokens across accounts.

Click on the SimpleCoin dropdown list within the Deployed contracts panel. Two new buttons will appear: *coinBalance* and *transfer*, as shown in figure 1.14.

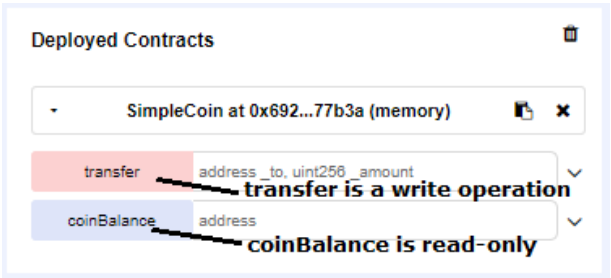


Figure 1.14. SimpleCoin operations buttons. coinBalance is a getter of the coinBalance state variable and it is a read-only operation. transfer allows to transfer the specified number of tokens to the indicated address.

The Remix IDE shows two types of buttons:

- *Blue buttons*. They perform read operations against the contract, such as to check the value of state variables or call read-only functions.
- *Red buttons*. They perform "write" operations against the contract, such as instantiating the contract through the constructor (Create) or, call functions which modify any state variables.

In our case *coinBalance* is blue because it allows to read the coin balance associated with an address. *transfer* is red because by clicking it you will alter the state of the contract, specifically by changing values contained in the coinBalance mapping state variable.

Let's check that the coinBalance associated with the address specified in the constructor has indeed the full initial supply of SimpleCoin you set at construction. Wrap the address with double quotes: `"0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C"`, enter it in the textbox and click *coinBalance*. Some output will appear. At its bottom, you should see the expected number of SimpleCoin tokens you specified in the constructor: 10000.

`0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C` is the address of one of the 5 test accounts present on the Remix IDE. You can see them in the *Transaction Origin* dropdown list box on the top right of the screen. Although they are not fully visible on the screen, their full addresses are reported in table 1.4 (I have retrieved them one by one by clicking the Copy Address icon next to the Account dropdown):

Table 1.4. Remix test accounts whose full address is hidden behind the HTML

0xca35b7d915458ef540ade6068dfe2f44e8fa733c
0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
0x583031d1113ad414f02576bd6afabfb302140225
0xdd870fa1b7c4700f2bd7f44238821c26f7392148

You can double check that the amount of SimpleCoin tokens associated with any address different from `0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C` is zero. For instance, enter the following address, wrapped with double quotes as you did earlier, in the *coinBalance* textbox: `"0x583031D1113aD414F02576BD6afaBfb302140225"`. After clicking the button, you will see, as expected, an amount equal to 0.

Transaction costs and execution transaction costs

You will have noticed that after clicking on the Create button while instantiating the contract and after clicking on the coinBalance button while checking the balance of a certain address, some text about *transaction costs* and *execution costs* appeared.

I will explain transaction and execution costs in detail in the following chapters. For now, you just need to know that the designers of the Ethereum platform decided to charge for code execution for two main reasons: in the first place to encourage and reward participants (mainly the "miners") who contribute to the platform by offering their computing power to process and validate transactions. Secondly, to discourage any malicious participant who would deploy and execute nonsensical contracts with the only intent of damaging and disrupting the network, for instance with a Denial of Service attack.

Recapping, when you instantiated the contract, an amount of 10000 SimpleCoin tokens got assigned, as initial money supply, to address starting with `0x14723A09`. No other address owns any token yet, as summarized in table 1.5.

Table 1.5. Balance of each Remix test account after contract instantiation

Account address	Account balance
0xca35b7d915458ef540ade6068dfe2f44e8fa733c	0
0x14723a09acff6d2a60dcdf7aa4aff308fddc160c	10,000
0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db	0
0x583031d1113ad414f02576bd6afabfb302140225	0
0xdd870fa1b7c4700f2bd7f44238821c26f7392148	0

Now you will call the `transfer` function to move some tokens from account with address starting with 0x14723a09 to a different test account. Since the transfer function moves tokens from the account of its caller, the function must be called from the contract creator's address starting with 0x14723a09. Pick this address from the *Transaction Origin* dropdown, then enter in the textbox of the transfer method the destination address (let's pick, for example the address starting with 0x4b0897b0) and a number of tokens to be transferred (for instance, 150 tokens). The values of these parameters should be separated by a comma:

```
"0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB", 150
```

Now click the *transfer* button. Apart from the usual information on transaction and execution costs, there is no actual return value from the function, as expected.

Check the number of tokens present in the contract creator's address by clicking on the coinBalance button after having entered again the contract creator's address ("0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C") in the related textbox: the value is now 9850, as expected.

If you perform the same check on the destination address ("0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB") you will get: 150. All other addresses still have zero tokens, as summarized in Table 1.6:

Table 1.6. Balance of each Remix test account after a transfer operation

Account address	Account balance
0xca35b7d915458ef540ade6068dfe2f44e8fa733c	0
0x14723a09acff6d2a60dcdf7aa4aff308fddc160c	9,850
0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db	150
0x583031d1113ad414f02576bd6afabfb302140225	0
0xdd870fa1b7c4700f2bd7f44238821c26f7392148	0

As an exercise, you can try to transfer coins from the address starting with 0x4b0897b05 to a different address and recheck the amounts are correct. While doing so, please do not perform yet any "crazy" transaction, such us trying to move more coins than a certain address is holding. As you know, in order to keep the code simple for the moment, you have not coded yet any boundary conditions. You will cover these from the next chapter.

Although the code you have written so far is very simple, your main objective, at this stage, was only starting to familiarize with smart contracts, with the Solidity language and with Remix. I believe you have achieved the objective. You now understand how contract instantiation works and how to interact with a contract from different accounts.

SimpleCoin is still at the stage of an embryonic Dapp. So far you have executed its code only on a JavaScript VM based simulator and, because it is lacking a UI, you have seen its output through Remix. In the next chapter, you will take one step further and you will install an Ethereum client. You will then deploy SimpleCoin to a real Ethereum network and you will interact again with it.

> **WARNING** If the compiler configured in the Setting tab is version 0.4.25, Remix will only allow you to enter addresses with a valid a checksum in the code editor. I will explain what a valid checksum is in chapter 5. However for now it means `0x14723a09acff6d2a60dcdf7aa4aff308fddc160c` (all in lowercase) and `0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C` are not interpreted as being equivalent to each other. Unfortunately addresses in the Account dropdown within the Run tab are all in lowercase and therefore not compliant. If you want to know the corresponding address with a valid checksum, you can use Etherscan,

the online blockchain viewer (https://etherscan.io/). Just enter the incorrectly formatted address (for example `0x14723a09acff6d2a60dcdf7aa4aff308fddc160c`) in the textbox at the top of the screen and you will see the correctly formatted corresponding address (`0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C`) in the Address header also at the top of the screen.

1.4 Summary

This chapter has been an introduction to Dapps, especially for those readers who are relatively unfamiliar with the concept of Decentralized application, those who do not know much about the blockchain or have never seen a smart contract in action.

The main points to take away from this chapter are the following ones:

- A Decentralized application is a novel type of application which is not owned or controlled by any entity and runs on a trustless decentralized P2P network
- The topology of a decentralized application is different from that of a conventional centralized one because both its business logic layer and its data layer (the blockchain) are fully replicated on each node of the network.
- Dapps rely on blockchain technology, which is based, in turn, on public key cryptography, cryptographic hash functions and the concept of "mining" through a consensus protocol.
- Modern blockchain platforms such as Ethereum extend the capabilities of earlier blockchain systems by introducing new technology, such as smart contracts based on a Turing complete virtual machine that makes Dapp development viable.
- There are many appropriate use cases for which it makes sense to decentralize an application, especially in the fields of provenance and authenticity tracking, identity verification, regulatory auditing, prediction markets, crowdfunding.
- Not always decentralized applications are the best solution for a business problem. For example, it does not make sense to decentralize an internal enterprise application which is not shared with any external participant.
- Smart contracts, that sit at the heart of Dapps, can be implemented, within the Ethereum platform, in a language called Solidity, very similar to JavaScript. It is possible to write simple smart contracts through the Remix Solidity IDE and simulate their activation and interaction with various mock Ethereum accounts.