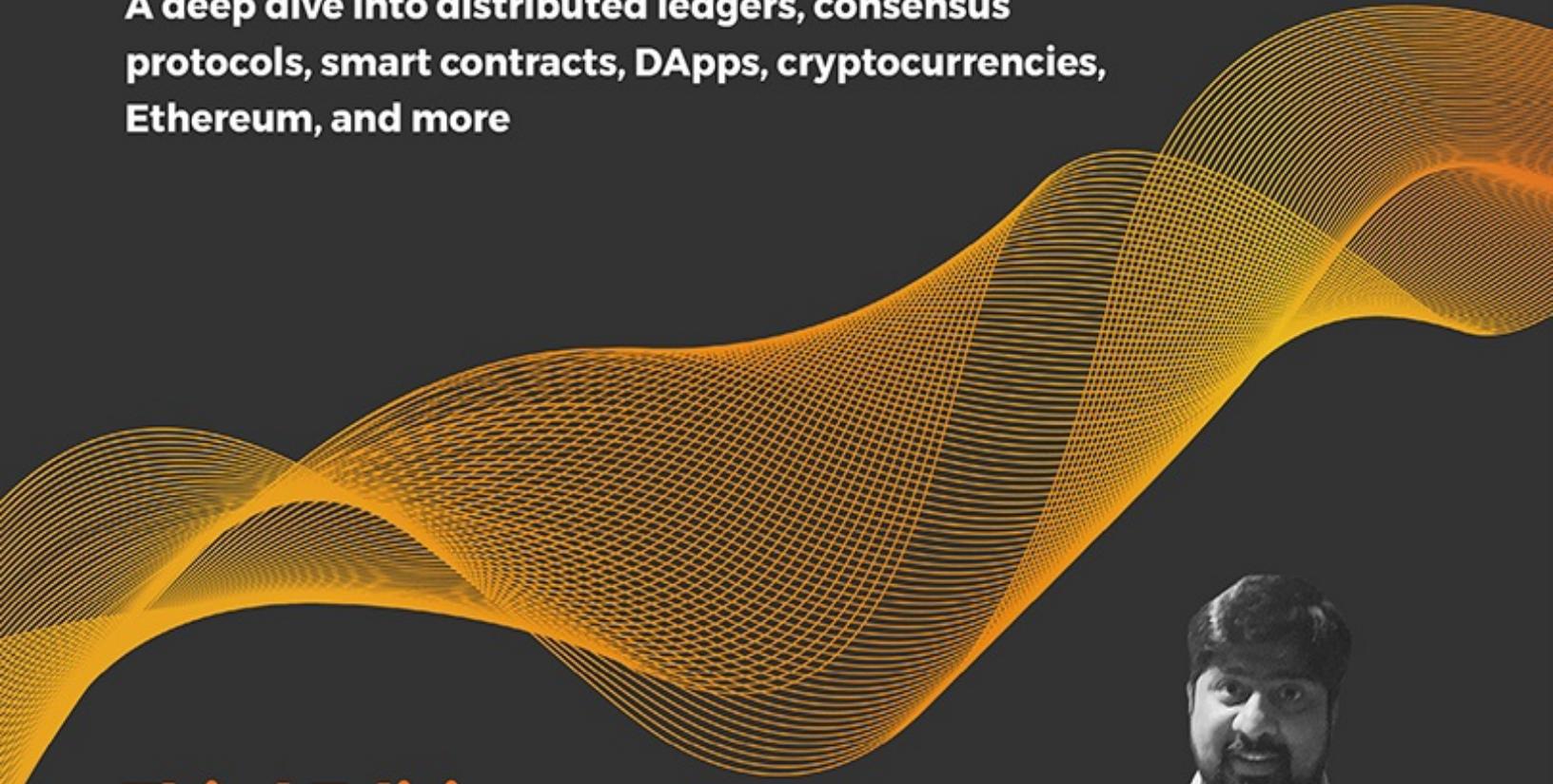


EXPERT INSIGHT

Mastering Blockchain

A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more



Third Edition



Imran Bashir

Packt

Mastering Blockchain

Third Edition

A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more

Imran Bashir

Packt

BIRMINGHAM - MUMBAI

Mastering Blockchain

Third Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Producer: Ben Renow-Clarke

Acquisition Editor – Peer Reviews: Suresh Jain

Content Development Editor: Edward Doxey

Technical Editor: Aniket Shetty

Project Editor: Carol Lewis

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Manju Arasan

Presentation Designer: Pranit Padwal

First published: March 2017

Second edition: March 2018

Third edition: August 2020

Production reference: 1290820

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-83921-319-9

www.packtpub.com



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.Packt.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Imran Bashir has an M.Sc. in Information Security from Royal Holloway, University of London, and has a background in software development, solution architecture, infrastructure management, and IT service management. He is also a member of the Institute of Electrical and Electronics Engineers (IEEE) and the British Computer Society (BCS). Imran has extensive experience in both the public and financial sectors, having worked on large-scale IT projects in the public sector before moving to the financial services industry. Since then, he has worked in various technical roles for different financial companies in Europe's financial capital, London.

I would like to thank the talented team at Packt, including Ben Renow-Clarke, Edward Doxey, Aniket Shetty, and Carol Lewis, who provided prompt guidance and extremely valuable feedback throughout this project. Edward Doxey provided extremely valuable advice that helped immensely in being able to properly structure the content of the book. I am also very thankful to the reviewer, Pranav Burnwal, who provided valuable feedback that helped to improve the material in this book.

I thank my wife and children for putting up with my all-night and weekend-long writing sessions. Above all, I would like to thank my parents, whose blessings on me have made everything possible.

Disclaimer: The information and viewpoints expressed in this book are those of the author and not necessarily those of any of the author's employers or their affiliates.

About the reviewer

Pranav Burnwal is currently serving as co-founder and CTO for Edufied, a Blockchain-based SaaS product. Pranav has a background in research and development, and has been working with cutting-edge technologies for a good number of years. These technologies include blockchain, big data, analytics (log and data), the cloud, and message queues.

With over 5 years' experience in blockchain, Pranav has established in excess of 12 pilot solutions on blockchain platforms, four live exchanges, six crypto wealth management systems, and many more awesome projects that are currently under development. Pranav has worked in a variety of domains, including BFSI, HLS, FMCG, and the automobile sector.

Pranav has been an active community member in multiple communities, offline as well as online. He was regional head for the Blockchain Education Network (BEN), and a volunteer at TechStars. He also organizes and attends hackathons.

Pranav has also served as a technical reviewer for multiple books on blockchain, including *Mastering Blockchain – 2nd Edition*, and *Foundations of Blockchain*.

Contents

Preface

Who this book is for

What this book covers

To get the most out of this book

Get in touch

1. Blockchain 101

The growth of blockchain technology

Progress toward maturity

Increasing interest

Distributed systems

The history of blockchain and Bitcoin

The events that led to blockchain

Electronic cash

Blockchain

Blockchain defined

Peer-to-peer

Distributed ledger

Cryptographically secure

Append-only

Updatable via consensus

Blockchain architecture

Blockchain by layers

Blockchain in business

Generic elements of a blockchain

How blockchain works

Benefits, features, and limitations of blockchain

Types of blockchain

Distributed ledgers

Distributed Ledger Technology

Public blockchains

Private blockchains

Semi-private blockchains

Sidechains

Permissioned ledger

Shared ledger

Fully private and proprietary blockchains

Tokenized blockchains

Tokenless blockchains

Consensus

Consensus mechanism

Types of consensus mechanisms

Consensus in blockchain

CAP theorem and blockchain

Summary

2. Decentralization

Decentralization using blockchain

Methods of decentralization

Disintermediation

Contest-driven decentralization

Routes to decentralization

How to decentralize

Decentralization framework example

Blockchain and full ecosystem decentralization

Storage

Communication

Computing power and decentralization

Pertinent terminology

Smart contracts

Autonomous agents

Decentralized organizations

Decentralized autonomous organizations

Decentralized autonomous corporations

Decentralized autonomous societies

Decentralized applications

Requirements of a DApp

Operations of a DApp

Design of a DApp

DApp examples

Platforms for decentralization

Ethereum

MaidSafe

[Lisk](#)
[EOS](#)

[Innovative trends](#)

[Decentralized web](#)

[Web 1](#)
[Web 2](#)
[Web 3](#)

[Decentralized identity](#)

[Decentralized finance \(DeFi\)](#)

[Summary](#)

[3. Symmetric Cryptography](#)

[Working with the OpenSSL command line](#)

[Introduction](#)

[Cryptography](#)

[Confidentiality](#)
[Integrity](#)
[Authentication](#)
[Non-repudiation](#)
[Accountability](#)

[Cryptographic primitives](#)

[Keyless primitives](#)

[Random numbers](#)
[Hash functions](#)
[Secure Hash Algorithms](#)

[Symmetric cryptography](#)

[Message authentication codes \(MACs\)](#)

[Hash-based MACs \(HMACs\)](#)

[Stream ciphers](#)

[Block ciphers](#)

[Data Encryption Standard \(DES\)](#)

[Advanced Encryption Standard \(AES\)](#)

[How AES works](#)

[An OpenSSL example of how to encrypt and decrypt using AES](#)

[Summary](#)

[4. Public Key Cryptography](#)

[Mathematics](#)

[Modular arithmetic](#)

[Sets](#)

[Fields](#)

[Finite fields](#)

[Prime fields](#)

[Groups](#)

[Abelian groups](#)

[Rings](#)

[Cyclic groups](#)

[Order](#)

[Asymmetric cryptography](#)

[Integer factorization](#)

[Discrete logarithm](#)

[Elliptic curves](#)

[Public and private keys](#)

[RSA](#)

[Elliptic curve cryptography](#)

[Mathematics behind ECC](#)

[The discrete logarithm problem in ECC](#)

[RSA using OpenSSL](#)

[Encryption and decryption using RSA](#)

[ECC using OpenSSL](#)

[Digital signatures](#)

[RSA digital signature algorithms](#)

[The elliptic curve digital signature algorithm](#)

[How to generate a digital signature using OpenSSL](#)

[Cryptographic constructs and blockchain technology](#)

[Homomorphic encryption](#)

[Signcryption](#)

[Secret sharing](#)

[Commitment schemes](#)

[Zero-knowledge proofs](#)

[zk-SNARKs](#)

[zk-STARKs](#)

[Zero-knowledge range proofs—ZKRPs](#)

[Different types of digital signatures](#)

[Blind signatures](#)

[Multisignatures](#)
[Threshold signatures](#)
[Aggregate signatures](#)
[Ring signatures](#)
[Encoding schemes](#)
[Base64](#)
[Base58](#)
[Applications of cryptographic hash functions](#)
[Merkle trees](#)
[Patricia trees](#)
[Distributed hash tables](#)

[Summary](#)

5. [Consensus Algorithms](#)

[Introducing the consensus problem](#)
[The Byzantine generals problem](#)
[Fault tolerance](#)
[Types of fault-tolerant consensus](#)
[State machine replication](#)
[FLP impossibility](#)
[Lower bounds on the number of processors to solve consensus](#)
[Analysis and design](#)
[Model](#)
[Processes](#)
[Timing assumptions](#)
[Synchrony](#)
[Asynchrony](#)
[Partial synchrony](#)
[Classification](#)
[Algorithms](#)
[CFT algorithms](#)
[Paxos](#)
[Raft](#)
[BFT algorithms](#)
[Practical Byzantine Fault Tolerance](#)
[Istanbul Byzantine Fault Tolerance](#)
[Tendermint](#)

Nakamoto consensus

Proof of stake (PoS)

HotStuff

Choosing an algorithm

Finality

Speed, performance, and scalability

Summary

6. Introducing Bitcoin

Bitcoin—an overview

The beginnings of Bitcoin

Egalitarianism versus authoritarianism

Bitcoin definition

Bitcoin—A user's perspective

Sending a payment

Cryptographic keys

Private keys in Bitcoin

Public keys in Bitcoin

Addresses in Bitcoin

Base58Check encoding

Vanity addresses

Transactions

The transaction lifecycle

Transaction pool

Transaction fees

The transaction data structure

Metadata

Inputs

Outputs

Verification

The Script language

Types of scripts

Contracts

Coinbase transactions

Transaction validation

Transaction bugs

Transaction malleability

Value overflow

Blockchain

The genesis block

Stale and orphan blocks

Size of the blockchain

Network difficulty

Mining

Tasks of the miners

Mining rewards

Proof of Work

The mining algorithm

The hash rate

Mining systems

CPU

GPU

FPGA

ASICs

Mining pools

Summary

7. The Bitcoin Network and Payments

The Bitcoin network

Full client and SPV client

Bloom filters

Wallets

Non-deterministic wallets

Deterministic wallets

Hierarchical deterministic wallets

Brain wallets

Paper wallets

Hardware wallets

Online wallets

Mobile wallets

Bitcoin payments

Innovation in Bitcoin

Bitcoin Improvement Proposals

Advanced protocols

Segregated Witness

Bitcoin Cash

[Bitcoin Unlimited](#)

[Bitcoin Gold](#)

[Bitcoin investment and buying and selling Bitcoin](#)
[Summary](#)

8. [Bitcoin Clients and APIs](#)

[Bitcoin client installation](#)

[Types of clients and tools](#)

[bitcoind](#)

[bitcoin-cli](#)

[bitcoin-qt](#)

[Setting up a Bitcoin node](#)

[Setting up the source code](#)

[Setting up bitcoin.conf](#)

[Starting up a node in the testnet](#)

[Starting up a node in regtest](#)

[Experimenting further with bitcoin-cli](#)

[Using the Bitcoin command-line tool – bitcoin-cli](#)

[Using the JSON RPC interface](#)

[Using the HTTP REST interface](#)

[Bitcoin programming](#)

[Summary](#)

9. [Alternative Coins](#)

[Introducing altcoins](#)

[Theoretical foundations](#)

[Alternatives to Proof of Work](#)

[Proof of Storage](#)

[Proof of Stake \(PoS\)](#)

[Various types of stake](#)

[Proof of Activity \(PoA\)](#)

[Non-outsourcable puzzles](#)

[Difficulty adjustment and retargeting algorithms](#)

[Kimoto Gravity Well](#)

[Dark Gravity Wave](#)

[DigiShield](#)

[MIDAS](#)

[Bitcoin limitations](#)

[Privacy and anonymity](#)

[Mixing protocols](#)
[Third-party mixing protocols](#)
[Inherent anonymity](#)

[Extended protocols on top of Bitcoin](#)

[Colored coins](#)
[Counterparty](#)

[Development of altcoins](#)

[Consensus algorithms](#)

[Hashing algorithms](#)

[Difficulty adjustment algorithms](#)

[Inter-block time](#)

[Block rewards](#)

[Reward halving rate](#)

[Block size and transaction size](#)

[Interest rate](#)

[Coinage](#)

[Total supply of coins](#)

[Token versus cryptocurrency](#)

[Initial Coin Offerings \(ICOs\)](#)

[ERC20 standard](#)

[Summary](#)

10. [Smart Contracts](#)

[History](#)

[Definition](#)

[Ricardian contracts](#)

[Smart contract templates](#)

[Oracles](#)

[Software and network-assisted proofs](#)

[TLSNotary](#)

[TLS-N based mechanism](#)

[Hardware device-assisted proofs](#)

[Android proof](#)

[Ledger proof](#)

[Trusted hardware-assisted proofs](#)

[Types of blockchain oracles](#)

[Inbound oracles](#)

[Outbound oracles](#)

[Blockchain oracle services](#)
[Deploying smart contracts](#)
[The DAO](#)
[Summary](#)

11. [Ethereum 101](#)

[Ethereum – an overview](#)
 [The yellow paper](#)
 [Useful mathematical symbols](#)
 [The Ethereum blockchain](#)
 [Ethereum – a user's perspective](#)
[The Ethereum network](#)
 [The mainnet](#)
 [Testnets](#)
 [Private nets](#)
[Components of the Ethereum ecosystem](#)
 [Keys and addresses](#)
 [Accounts](#)
 [Types of accounts](#)
 [Transactions and messages](#)
 [RLP](#)
 [Contract creation transactions](#)
 [Message call transactions](#)
 [Messages](#)
 [Transaction validation and execution](#)
 [The transaction substate](#)
 [State storage in the Ethereum blockchain](#)
 [Transaction receipts](#)
 [Ether cryptocurrency/tokens \(ETC and ETH\)](#)
[The Ethereum Virtual Machine \(EVM\)](#)
 [Execution environment](#)
 [The machine state](#)
 [The iterator function](#)
[Smart contracts](#)
 [Native contracts](#)
 [The elliptic curve public key recovery function](#)
 [The SHA-256-bit hash function](#)
 [The RIPEMD-160-bit hash function](#)

[The identity/datacopy function](#)
[Big mod exponentiation function](#)
[Elliptic curve point addition function](#)
[Elliptic curve scalar multiplication](#)
[Elliptic curve pairing](#)
[Blake2 compression function 'F'](#)

[Summary](#)

[12. Further Ethereum](#)

[Blocks and blockchain](#)

[The genesis block](#)
[The block validation mechanism](#)
[Block finalization](#)
[Block difficulty mechanism](#)
[Gas](#)
[Fee schedule](#)

[Wallets and client software](#)

[Wallets](#)
[Geth](#)
[Eth](#)
[Parity](#)
[Trinity](#)
[Light clients](#)

[Installation and usage](#)

[Geth](#)
[Ethereum account management using Geth](#)
[How to query the blockchain using Geth](#)
[Ethereum keystore](#)
[Eth installation](#)
[OpenEthereum installation](#)

[MetaMask](#)

[Installation](#)
[Creating and funding an account using MetaMask](#)

[Nodes and miners](#)

[The consensus mechanism](#)
[Forks in the blockchain](#)
[Ethash](#)
[CPU mining](#)

[GPU mining](#)
[Benchmarking](#)
[Mining rigs](#)
[Mining pools](#)
[ASICs](#)
[APIs, tools, and DApps](#)
 [Applications \(DApps and DAOs\) developed on Ethereum](#)
 [Tools](#)
 [Geth JSON RPC API](#)
 [Examples](#)
 [Supporting protocols](#)
 [Whisper](#)
 [Swarm](#)
[Programming languages](#)
 [Runtime bytecode](#)
 [Opcodes](#)
[Summary](#)

13. [Ethereum Development Environment](#)

[Overview](#)
 [Test networks](#)
 [Components of a private network](#)
 [Network ID](#)
 [The genesis file](#)
 [Data directory](#)
 [Flags and their meaning](#)
 [Static nodes](#)
 [Starting up the private network](#)
 [Mining on the private network](#)
 [Remix IDE](#)
 [MetaMask](#)
 [Using MetaMask and Remix IDE to deploy a smart contract](#)
 [Adding a custom network to MetaMask and connecting](#)
 [Remix IDE with MetaMask](#)
 [Importing accounts into MetaMask using keystore files](#)
 [Deploying a contract with MetaMask](#)
 [Interacting with a contract through MetaMask using Remix IDE](#)

Summary

14. Development Tools and Frameworks

Languages

Compilers

The Solidity compiler

Installation

Functions

Tools and libraries

Node.js

Ganache CLI

Ganache

Frameworks

Truffle

Drizzle

Embark

Brownie

Waffle

Etherlime

OpenZeppelin

Contract development and deployment

Writing smart contracts

Testing smart contracts

Deploying smart contracts

The layout of a Solidity source code file

Version pragma

Import

Comments

The Solidity language

Variables

Local variables

Global variables

State variables

Data types

Value types

Reference types

Control structures

Events

[Inheritance](#)
[Libraries](#)
[Functions](#)
[Error handling](#)

[Summary](#)

15. [Introducing Web3](#)

[Exploring Web3 with Geth](#)
[Contract deployment](#)
[POST requests](#)
[Retrieving the list of accounts](#)
[Interacting with contracts via frontends](#)
[The HTML and JavaScript frontend](#)
[Installing Web3.js JavaScript library](#)
[Interacting with contracts via a web frontend](#)
[Creating an app.js JavaScript file](#)
[Creating a Web3 object](#)
[Calling contract functions](#)
[Development frameworks](#)
[Using Truffle to develop a decentralized application](#)
[Installing and initializing Truffle](#)
[Compiling, testing, and migrating using Truffle](#)
[Interacting with the contract](#)
[Using Truffle to test and deploy smart contracts](#)
[Deployment on decentralized storage using IPFS](#)

[Summary](#)

16. [Serenity](#)

[Ethereum 2.0—an overview](#)
[Goals](#)
[Main features](#)
[Roadmap of Ethereum](#)
[Development phases](#)
[Phase 0](#)
[The beacon chain](#)
[Beacon nodes](#)
[Validator nodes](#)
[Beacon and validator node comparison](#)
[Deposit contracts](#)

[Fork choice](#)
[P2P interface \(networking\)](#)
[ETH 2](#)
[Simple Serialize](#)
[BLS cryptography](#)

[Phase 1](#)

[Shard chains](#)

[Transitioning from Ethereum 1 to Ethereum 2](#)

[Phase 2](#)

[Phase 3](#)

[Architecture](#)

[Summary](#)

17. [Hyperledger](#)

[Projects under Hyperledger](#)

[Distributed ledgers](#)

[Fabric](#)

[Sawtooth](#)

[Iroha](#)

[Indy](#)

[Besu](#)

[Burrow](#)

[Libraries](#)

[Aries](#)

[Transact](#)

[Quilt](#)

[Ursa](#)

[Tools](#)

[Avalon](#)

[Cello](#)

[Caliper](#)

[Explorer](#)

[Domain-specific](#)

[Grid](#)

[Labs](#)

[Hyperledger reference architecture](#)

[Hyperledger design principles](#)

[Modular structure](#)

[Privacy and confidentiality](#)

[Identity](#)

[Scalability](#)

[Deterministic transactions](#)

[Auditability](#)

[Interoperability](#)

[Portability](#)

[Rich data queries](#)

[Hyperledger Fabric](#)

[Membership services](#)

[Blockchain services](#)

[Consensus services](#)

[Distributed ledger](#)

[The peer-to-peer protocol](#)

[Ledger storage](#)

[Smart contract services](#)

[APIs and CLIs](#)

[Components](#)

[Peers](#)

[Clients](#)

[Channels](#)

[World state database](#)

[Transactions](#)

[Membership Service Provider](#)

[Smart contracts](#)

[Crypto service provider](#)

[Applications on blockchain](#)

[Chaincode implementation](#)

[The application model](#)

[Consensus in Hyperledger Fabric](#)

[The transaction lifecycle in Hyperledger Fabric](#)

[Fabric 2.0](#)

[New chaincode lifecycle management](#)

[New chaincode application patterns](#)

[Enhanced data privacy](#)

[External chaincode launcher](#)

[Raft consensus](#)

Better performance

Hyperledger Sawtooth

Core features

Modular design

Parallel transaction execution

Global state agreement

Dynamic and pluggable consensus algorithms

Multi-language support

Enhanced event mechanism

On-chain governance

Interoperability

Consensus in Sawtooth

PoET

PBFT

Raft

Transaction lifecycle

Components

Validator

REST API

Client

State

Transaction processors

Transaction families

REST API

Setting up a Sawtooth development environment

Prerequisites

Using PoET

Using PBFT

Setting up a Sawtooth network

Summary

18. Tokenization

Tokenization on a blockchain

Advantages of tokenization

Disadvantages of tokenization

Types of tokens

Fungible tokens

Non-fungible tokens

Stable tokens

- Fiat collateralized
- Commodity collateralized
- Crypto collateralized
- Algorithmically stable

Security tokens

Process of tokenization

Token offerings

- Initial coin offerings
- Security token offerings
- Initial exchange offerings
- Equity token offerings
- Decentralized autonomous initial coin offering
- Other token offerings

Token standards

- ERC-20
- ERC-223
- ERC-777
- ERC-721
- ERC-884
- ERC-1400
- ERC-1404

Trading and finance

- Financial markets
- Trading
- Exchanges
- Orders and order properties
 - Order management and routing systems

Components of a trade

- The underlying instrument

Trade lifecycle

- Order anticipators
- Market manipulation

DeFi

- Trading tokens

- Regulation

Building an ERC-20 token

Pre requisites

Building the Solidity contract

Solidity contract source code

Deploying the contract on the Remix JavaScript virtual machine

Adding tokens in MetaMask

Emerging concepts

Tokenomics/token economics

Token engineering

Token taxonomy

Summary

19. Blockchain – Outside of Currencies

The Internet of Things

Internet of Things architecture

Physical object layer

Device layer

Network layer

Management layer

Application layer

Benefits of IoT and blockchain convergence

Implementing blockchain-based IoT in practice

Setting up Raspberry Pi

Setting up the first node

Setting up the Raspberry Pi node

Building the electronic circuit

Government

Border control

Voting

Citizen identification (ID cards)

Health

Finance

Insurance

Post-trade settlement

Financial crime prevention

Payments

Decentralization

Faster settlement

Better resilience
Cross-border payments
Peer-to-peer loans

Media
Summary

20. Enterprise Blockchain

Enterprise solutions and blockchain

Success factors

Limiting factors

Slow performance

Lack of access governance

Lack of privacy

Probabilistic consensus

Transaction fees

Requirements

Privacy

Confidentiality

Anonymity

Performance

Scalability/speed

Access governance

Further requirements

Compliance

Interoperable

Integration

Ease of use

Monitoring

Secure off-chain computation

Better tools

Enterprise blockchain versus public blockchain

Use cases of enterprise blockchains

Enterprise blockchain architecture

Network layer

Protocol layer

Privacy layer

Governance layer

Integration layer

What is Apache Camel?

Application layer

Security, performance, scalability, monitoring

Designing enterprise blockchain solutions

TOGAF

Business architecture domain

Data architecture domain

Application architecture domain

Technology architecture domain

Architecture development method

Preliminary phase

Architecture vision

Business architecture

Information systems architecture

Technology architecture

Opportunities and solutions

Migration planning

Implementation governance

Architecture change management

Blockchain in the cloud

Currently available enterprise blockchains

Corda

Quorum

Fabric

Autonity

Comparison of main platforms

Enterprise blockchain challenges

Interoperability

Lack of standardization

Compliance

Business challenges

Corda

Architecture

Corda network

State objects

Transactions

Consensus

[Flows](#)

[CorDapps](#)

[Components](#)

[Nodes](#)

[The permissioning service](#)

[Network map service](#)

[Notary service](#)

[Oracle service](#)

[Transactions](#)

[Vaults](#)

[Other tools](#)

[Transaction flow](#)

[Corda development environment](#)

[Quorum](#)

[Architecture](#)

[Enhanced P2P](#)

[Enhanced state \(private and public\)](#)

[Pluggable consensus](#)

[No transaction fees](#)

[Private transactions](#)

[Modified block generation mechanism](#)

[Modified block validation mechanism](#)

[Enhanced RPC API](#)

[Privacy manager](#)

[Transaction manager](#)

[Enclave](#)

[Cryptography used in Quorum](#)

[Privacy](#)

[Enclave encryption](#)

[Transaction propagation to transaction managers](#)

[Enclave decryption](#)

[Access control with permissioning](#)

[Performance](#)

[Pluggable consensus](#)

[Setting up Quorum with IBFT](#)

[Quorum Wizard](#)

[Installing Quorum Wizard](#)

[Running Quorum Wizard to create a new network](#)
[Cakeshop](#)

[Running a private transaction](#)

[Node 1](#)

[Node 2](#)

[Node 3](#)

[Node 4](#)

[Viewing the transaction in Cakeshop](#)

[Further investigation](#)

[Node 1](#)

[Node 2, which is privy to the transaction](#)

[Node 3, which is not privy to the transaction](#)

[Other Quorum projects](#)

[Remix plugin](#)

[Pluggable architecture](#)

[Summary](#)

21. [Scalability and Other Challenges](#)

[Scalability](#)

[Blockchain planes](#)

[Network plane](#)

[Consensus plane](#)

[Storage plane](#)

[View plane](#)

[Side plane](#)

[Methods for improving scalability](#)

[Layer 0 – network solutions](#)

[Layer 1 – on-chain solutions](#)

[Layer 2 – off-chain and multichain solutions](#)

[Privacy](#)

[Anonymity](#)

[Confidentiality](#)

[Techniques to achieve privacy](#)

[Layer 0](#)

[Layers 1 and 2](#)

[Security](#)

[Formal verification](#)

[Model checking](#)

Verifying consensus mechanisms
Smart contract security
 Static analysis in Remix IDE
 Why3
 Oyente
 Other tools
 Formal verification of smart contracts
Other challenges
 Interoperability
 Polkadot
 Lack of standardization
 Post-quantum resistance
 Compliance and regulation
Summary

22. Current Landscape and What's Next

Emerging trends
 New implementations of blockchain technology
 Application-specific blockchains
 Start-ups
 Technology improvements
 Standardization
 Consortia
 Enhancements
 Ongoing research and study
 Cryptography
 Cryptoeconomics
 Hardware development
 Formal methods and security
 New programming languages
 Education and employment within blockchain
 Innovative blockchain applications
 Blockchain as a Service
 Convergence with other technologies
 Alternatives to blockchains
 Some debatable ideas
 Public versus private on the blockchain
 Central bank digital currency

[Areas to address](#)

[Regulation](#)

[Illegal activity](#)

[Privacy or transparency](#)

[Blockchain research topics](#)

[Smart contracts](#)

[Cryptographic function limitations](#)

[Consensus algorithms](#)

[Scalability](#)

[Code obfuscation](#)

[Blockchain and AI](#)

[The future of blockchain](#)

[Summary](#)

[Index](#)

Preface

This book has one goal. To teach the theory and practice of distributed ledger technology to anyone interested in learning this fascinating new subject. Whether a seasoned technologist, student, business executive or merely an enthusiast; anyone with an interest in blockchain technology can benefit from this book. To this end, I aim to provide a comprehensive and in-depth reference of distributed ledger technology that not only serves the expert, but is also accessible for beginners. I especially focus on describing the core characteristics of blockchain, so that readers can establish a strong foundation on which to build further knowledge and expertise. I also envisage that with this approach, the book will remain useful in the future despite constant advancements. Some of these core topics include core blockchain principles, cryptography, consensus algorithms, distributed systems theory, and smart contracts. In addition, practical topics such as programming smart contracts in Solidity, building blockchain networks, using blockchain development frameworks such as Truffle, and writing decentralized applications, also constitute a significant part of this book. Moreover, many different types of blockchain, related use cases and cross-industry applications of blockchain technology have been discussed in detail.

This approach makes this book a unique blend of theoretical principles and hands-on applications. After reading this book, readers will not only be able to understand the technical underpinnings of this technology, but will also be empowered to write smart contract code and build blockchain networks themselves. Practitioners can use this book as a reference, and it can also serve as a textbook for students wishing to learn this technology. Indeed, some institutions have adopted the second edition of this book as a primary textbook for their courses on blockchain technology.

Soon after the advent of blockchain technology, several texts were written on the subject. However, they mostly concentrated on a specific aspect of blockchain technology, such as cryptocurrencies. Especially between 2015

and 2017, no single text was available that could provide a detailed account of the whole picture of blockchain technology.

To fill this gap, the first edition of this book was written in 2017. In 2018, due to rapid development in the area of blockchain technology, the need was felt to update the book. As a result, the second edition of the book was published in early 2018 with some important updates. Inevitably, since the publication of the second edition, blockchain technology has evolved quite significantly. Some of the ideas that contributed to the evolution of blockchain technology include novel cryptographic protocols, new consensus algorithms, new privacy techniques, and extensive work on scalability. Moreover, innovative types of blockchain, developed to address limitations in the underlying technology, have emerged, along with a deep interest in the adoption of blockchain technology in enterprise settings. Each of these changes inspired us to update the book again with the latest industry developments.

This book has four new chapters on some of the latest topics, including consensus algorithms, Ethereum 2.0, tokenization, and enterprise blockchains.

Moreover, all chapters have been thoroughly revised, and various new topics, including a discussion on new blockchains or cryptocurrencies, such as Tezos, EOS, and Quorum, are now part of the book package. Also, many additional details on oracles, cryptography, scalability, and privacy techniques have been added. Some of the older material that is not quite so relevant today has been removed. We have also created a bonus online content pack, which contains academic slides and review questions and answers, which will enable teachers to use this book readily as an academic textbook. In addition, the bonus content includes advanced material on alternative coins, full end to end decentralized application development using Solidity, Truffle and Drizzle, and an exploration of various alternative blockchain platforms. I encourage the readers to read this book in conjunction with the online content, which can be found here:

https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_M

[ore_Bonus_Content.pdf](#). This will enable them to fully benefit from this book.

I hope that this book will serve technologists, teachers, students, scientists, developers, business executives, and indeed anyone who wants to learn about this fascinating technology well for many years to come.

Who this book is for

This book is for anyone who wants to understand blockchain technology in depth. It can also be used as a reference resource by developers who are developing applications for blockchain. Also, this book can be used as a textbook or learning resource for courses, examinations, and certifications related to blockchain technology and cryptocurrencies.

What this book covers

Chapter 1, Blockchain 101, introduces the basic concepts of distributed computing, on which blockchain technology is based. It also covers the history, definitions, features, types, and benefits of blockchains, along with various consensus mechanisms that are at the core of the blockchain technology.

Chapter 2, Decentralization, covers the concept of decentralization and its relationship with blockchain technology. Various methods and platforms that can be used to decentralize a process or a system are also introduced.

Chapter 3, Symmetric Cryptography, introduces the theoretical foundations of symmetric cryptography, which is necessary in order to understand how various security services such as confidentiality and integrity are provided.

Chapter 4, Public Key Cryptography, introduces concepts such as public and private keys, digital signatures, and hash functions, and includes a number of practical examples.

Chapter 5, Consensus Algorithms, covers the fundamentals of consensus algorithms and describes the design and inner workings of several such algorithms. It covers both traditional consensus protocols and blockchain consensus protocols.

Chapter 6, Introducing Bitcoin, covers Bitcoin, the first and largest blockchain. It introduces technical concepts related to the Bitcoin cryptocurrency in detail.

Chapter 7, Bitcoin Network and Payments, covers the Bitcoin network, relevant protocols, and various Bitcoin wallets. Moreover, advanced protocols, Bitcoin trading and payments are also introduced.

Chapter 8, Bitcoin Clients and APIs, introduces various Bitcoin clients and programming APIs that can be used to build Bitcoin applications.

Chapter 9, Alternative Coins, introduces alternative cryptocurrencies that were introduced after the invention of Bitcoin, along with their theoretical foundations.

Chapter 10, Smart Contracts, provides an in-depth discussion on smart contracts. Topics such as their history and definition, Ricardian contracts, oracles, and the theoretical aspects of smart contracts are presented in this chapter.

Chapter 11, Ethereum 101, introduces the design and architecture of the Ethereum blockchain in detail. It covers various technical concepts related to the Ethereum blockchain that explain the underlying principles, features, and components of this platform in depth.

Chapter 12, Further Ethereum, continues the introduction of Ethereum from the previous chapter, and covers topics related to the Ethereum Virtual Machine, mining, and supporting protocols for the Ethereum chain.

Chapter 13, Ethereum Development Environment, covers topics related to setting up private networks for Ethereum smart contract development and programming.

Chapter 14, Development Tools and Frameworks, provides a detailed practical introduction to the Solidity programming language, and different relevant tools and frameworks that are used for development with Ethereum.

Chapter 15, Introducing Web3, covers the development of decentralized applications and smart contracts using the Ethereum blockchain. A detailed introduction to the Web3 API is provided, along with multiple practical examples.

Chapter 16, Serenity, introduces the latest developments in Serenity, the update that will introduce Ethereum 2.0. It covers the theory relating to the

beacon chain, validator nodes, and explores various phases of Ethereum 2 development.

Chapter 17, Hyperledger, presents a discussion about the Hyperledger project from the Linux Foundation, which includes different blockchain projects introduced by its members.

Chapter 18, Tokenization, introduces the topic of tokenization, stable coins, and other relevant ideas such as initial coin offerings, decentralized exchanges, decentralized finance, and token development standards.

Chapter 19, Blockchain – Outside of Currencies, provides a practical and detailed introduction to applications of blockchain technology in fields others than cryptocurrencies, including the Internet of Things, government, media, and finance.

Chapter 20, Enterprise Blockchains, explains the use and application of blockchain technology in enterprise settings, and covers distributed ledger platforms such as Quorum and Corda.

Chapter 21, Scalability and Other Challenges, is dedicated to a discussion of the challenges faced by blockchain technology and how to address them.

Chapter 22, Current Landscape and What's Next, is aimed at providing information about the current landscape, projects, and research efforts related to blockchain technology, and includes a number of predictions based on the current state of blockchain technology.

To get the most out of this book

In order to gain the most from this book, some familiarity with computer science and basic knowledge of a programming language is desirable. However, beginner-level knowledge of computing in general and enthusiasm to learn should suffice!

Download the example code files

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at <http://www.packtpub.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Blockchain-Third-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

https://static.packt-cdn.com/downloads/9781839213199_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example, "The `call`, `callcode`, and `delegatecall` functions are provided in order to interact with functions that do not have an ABI."

A block of code is set as follows:

```
pragma solidity >5.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

Any command-line input or output is written as follows:

```
$ sudo apt-get install solc
```

Bold: Indicates a new term, an important word, or words that you see on screen, for example, in menus or dialog boxes, which also appear in the text like this. For example: "Two notable schemes that originate from ECC are **Elliptic Curve Diffie-Hellman (ECDH)** for key exchange and **Elliptic Curve Digital Signature Algorithm (ECDSA)** for digital signatures."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Blockchain 101

It is very likely that if you are reading this book, you already have heard about blockchain and have some fundamental appreciation of its enormous potential. If not, then let me tell you that this is a technology that has promised to positively alter the existing paradigms of nearly all industries including, but not limited to, the IT, finance, government, media, medical, and law sectors.

This chapter is an introduction to blockchain technology, its technical foundations, the theory behind it, and various techniques that have been combined together to build what is known today as **blockchain**.

In this chapter, the theoretical foundations of distributed systems are described first. Next, the precursors of Bitcoin are presented. Finally, blockchain technology is introduced. This approach is a logical way of understanding blockchain technology, as the roots of blockchain are in distributed systems and cryptography. We will be covering a lot of ground quickly here, but don't worry—we will go over a great deal of this material in much greater detail as you move throughout the book.

The growth of blockchain technology

With the invention of Bitcoin in 2008, the world was introduced to a new concept, which revolutionized the whole of society. It was something that

promised to have an impact upon every industry. This new concept was blockchain; the underlying technology that underpins Bitcoin.

Some describe blockchain as a revolution, whereas another school of thought believes that it is going to be more evolutionary, and it will take many years before any practical benefits of blockchain reach fruition. This thinking is correct to some extent, but, in my opinion, the revolution has already begun. It is a technology that has an impact on current technologies too and possesses the ability to change them at a fundamental level.

Many prominent organizations all around the world have moved on from the proof-of-concept stage and are already writing production systems utilizing blockchain. The disruptive potential of blockchain has now been fully recognized. For example, **Decentralized Finance (DeFi)** has recently emerged as a new disruptive idea that aims to decentralize the existing financial system. DeFi offers individuals more control over their assets, allowing them to fully own and operate their financial strategy on blockchain without requiring any intermediaries. This is already challenging traditional finance by creating decentralized platforms for trading, investment, insurance, payments, and lending.



A glimpse of the potential of DeFi can be seen at <https://defipulse.com>. A noticeable fact is the amount of total value locked in the system, which is almost 5 billion US Dollars.

However, some organizations are still in the preliminary exploration stage, though they are expected to progress more quickly as the technology matures.

Progress toward maturity

If we look at the last few years, we notice that, in 2013, some ideas started to emerge that suggested that blockchain may have the potential for application in areas other than cryptocurrencies. Around that time, the

primary usage of blockchain was in cryptocurrency space such as Bitcoin and Litecoin, and many new coins emerged during that period.



Cryptocurrency can be defined as a digital currency that is secured by cryptography.

The following graph shows a broad-spectrum outline of the year-wise progression and adoption trends of blockchain technology. The years shown on the x axis indicate the range of time in which a specific phase of blockchain technology falls. Each phase has a name that represents the stage at which the technology was reached, and this is shown on the x axis starting from the period of **IDEAS AND THOUGHTS** in 2013 to eventually **MATURITY AND FURTHER PROGRESS**, expected in 2025. The y axis shows the level of activity, involvement, and adoption of blockchain technology. The graph shows that, by roughly 2025, blockchain technology is expected to become mature and have a high number of users:

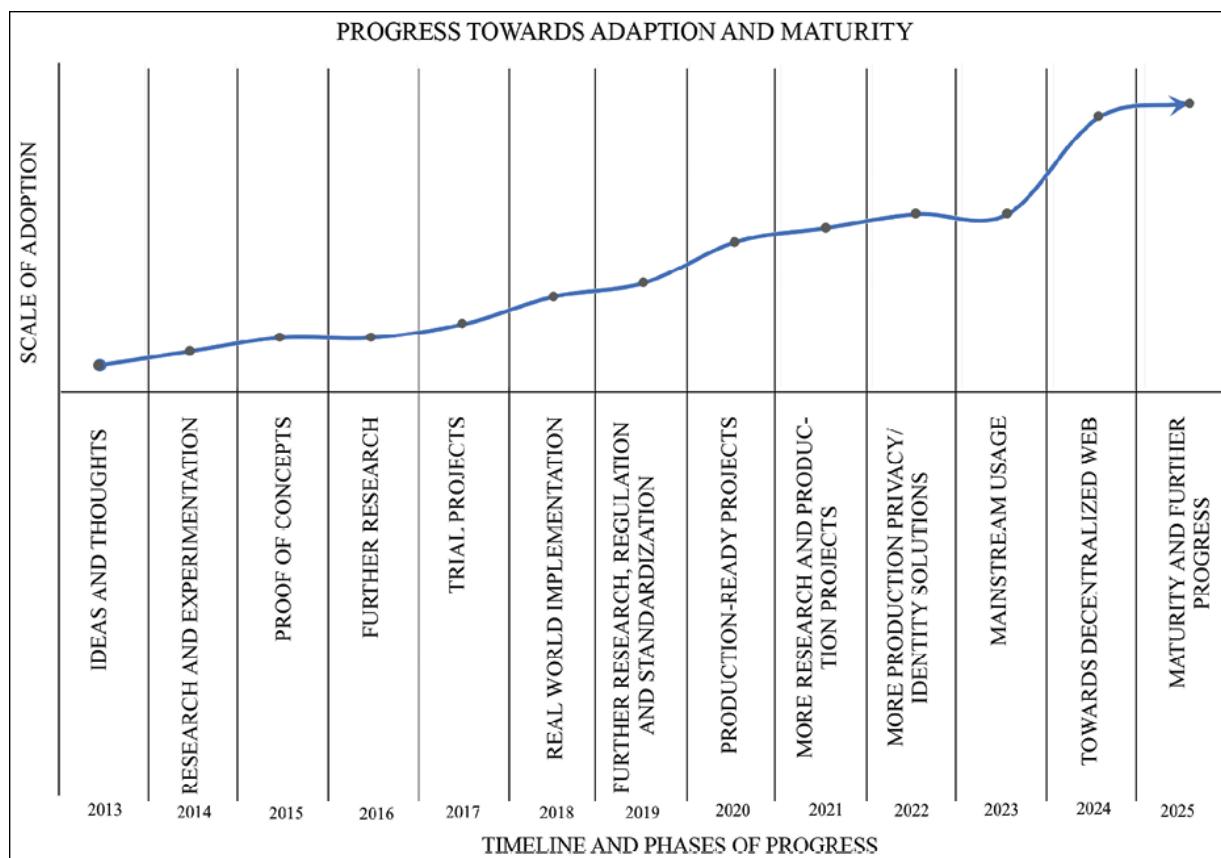


Figure 1.1: Blockchain technology adoption and maturity

The preceding graph shows that, in 2013, ideas and thoughts emerged regarding other usages of blockchain technology apart from cryptocurrencies. Then, in 2014, some research and experimentation began, which led to proofs of concept, further research, and full-scale trial projects between 2015 and 2017. In 2018, we saw real-world implementations. Already many projects are underway and set to replace existing systems; for example, the **Australian Securities Exchange (ASX)** is soon to become the first organization to replace its legacy clearing and settlement system with blockchain technology.



More information on this topic can be found at
<https://www.asx.com.au/services/chess-replacement.htm>.

Another recent prominent example is that of a production-ready project implemented by Santander, where the first end-to-end blockchain bond has been issued. This is a significant step toward the mainstream adoption of blockchain.



You can read more about this at
http://www.santander.com/csgs/Satellite/CFWCSancomQP01/en_GB/Corporate/Press-room/2019/09/12/Santander-launches-the-first-end-to-end-blockchain-bond.html.

It is expected that, during 2020, more research will be carried out, along with an increasing interest toward the regulation and standardization of blockchain technology. After this, production-ready projects and off-the-shelf projects will be available utilizing blockchain technology from 2020, and by 2021 mainstream production-level usage of blockchain technology is expected.



Progress in blockchain technology almost feels like the internet dot-com boom of the late 1990s.

In the next few years, research in the scalability of blockchains, where blockchains will be able to handle a large number of transactions similar to traditional financial networks, is expected to become more mature. Also, during the first few years of the 2020s, we will see more production-level usage of blockchain addressing issues such as privacy, decentralized identity, and some progress toward the decentralized web (or internet decentralization). Note that such solutions already exist but are not in mainstream use. The mainstream usage of such technologies is expected from the start of 2020.

It is expected that, at the start of the next decade, research in such areas will continue along with the adoption and further maturity of blockchain technology. Finally, in 2025, it is expected that the technology will be mature enough to be used on a day-to-day basis by, less tech-savvy people. For example, blockchain networks can be used as easily and naturally as consumers using the internet now. Further research is expected to continue even beyond this point. Please note that the timelines provided in the chart are not strict and may vary, as it is quite difficult to predict when exactly blockchain technology will become mature. This graph is based on the progress made in recent years and the current climate of research, interest, and enthusiasm regarding this technology, which can be extrapolated to predict that blockchain will progress to become a mature technology by 2025.

Increasing interest

Interest in blockchain technology has risen quite significantly over the last few years. Once dismissed simply as "geek money" from a cryptocurrency point of view, or as something that was just not considered worth pursuing, blockchain is now being researched by the largest companies and organizations around the world. Millions of dollars are being spent to adopt and experiment with this technology. This is evident from recent actions taken by the European Union, where they have announced plans to increase funding for blockchain research to almost 340 million Euros by 2020.



Interested readers can read more about this at
<https://www.irishtimes.com/business/technology/boost-for-blockchain-research-as-eu-increases-funding-four-fold-1.3383340>.

Another report suggests that global spending on blockchain technology research could reach 9.2 billion US Dollars by 2021.



More information regarding this can be found at
<https://bitcoinmagazine.com/articles/report-suggests-global-spending-blockchain-tech-could-reach-92-billion-2021/>.

Also, the interest in blockchain within academia is astounding, and many educational establishments—including prestigious universities around the world—are conducting research and development on blockchain technology. There are not only educational courses being offered by many institutions, but academics are also conducting high-quality research and producing a number of insightful research papers on the topic. There are also a number of research groups and conferences around the world that specifically focus on blockchain research. This is extremely useful for the growth of the entire blockchain ecosystem. A simple online search of "blockchain research groups" would reveal hundreds, if not thousands, of these research groups.

There are also various consortiums such as **Enterprise Ethereum Alliance (EEA)** at <https://entethalliance.org> and Hyperledger at <https://www.hyperledger.org>, which have been established for research, development, and the standardization of blockchain technology.

Moreover, a large number of start-ups are providing blockchain-based solutions already. A simple trend search on Google reveals the immense scale of interest in blockchain technology over the last few years.

Especially since early 2017, the increase in the search term "blockchain" is quite significant, as shown in the following graph:

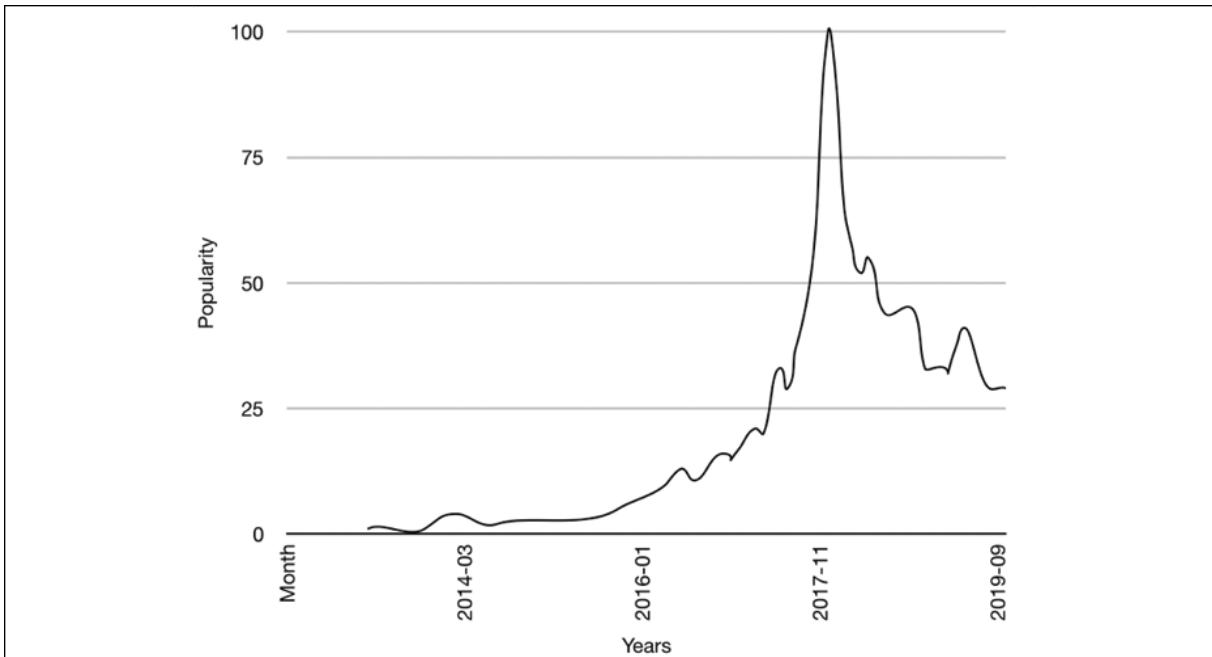


Figure 1.2: A popularity graph for the search term "blockchain," based on data from Google trends

It should be noted that the decrease shown at the end of the graph does not necessarily directly translate into the decreased interest in blockchain technology, but it simply captures the fact that the search term was searched fewer times as compared to late 2017 and early 2018. This could be simply due to the fact that people interested in blockchain technology have already understood enough about blockchain to no longer warrant searching for it on Google. We do see, however, a significant and continued interest in blockchain technology.

Various benefits of this technology have already been envisioned, such as decentralized trust, cost savings, transparency, and efficiency. However, there are multiple challenges too that are actively being researched on blockchain, such as scalability and privacy.

In this book, we are going to see how blockchain technology can help to bring about the benefits mentioned previously. You are going to learn what exactly blockchain technology is and how it can reshape businesses, multiple industries, and indeed everyday life by bringing about a plenitude of benefits such as efficiency, cost savings, transparency, and security. We will also explore what **distributed ledger technology (DLT)** is,

decentralization, and smart contracts, and how technology solutions can be developed and implemented using mainstream blockchain platforms such as Ethereum and Hyperledger. We will also investigate what challenges need to be addressed before blockchain can become a mainstream technology.

We'll also take a critical look at blockchain; *Chapter 21, Scalability and Other Challenges*, is dedicated to a discussion of the limitations and challenges of blockchain technology.

We shall begin our exploration of blockchain by looking at distributed systems in the following section. This is a foundational paradigm used within blockchain, and we must have a firm grasp on what distributed systems are before we can meaningfully discuss blockchain in detail.

Distributed systems

Understanding distributed systems is essential to our understanding of blockchain, as blockchain was a distributed system at its core. It is a distributed ledger that can be centralized or decentralized. A blockchain is originally intended to be and is usually used as a decentralized platform. It can be thought of as a system that has properties of both decentralized and distributed paradigms. It is a decentralized-distributed system.

Distributed systems are a computing paradigm whereby two or more nodes work with each other in a coordinated fashion to achieve a common outcome. It is modeled in such a way that end users see it as a single logical platform. For example, Google's search engine is based on a large distributed system; however, to a user, it looks like a single, coherent platform.

A **node** can be defined as an individual player in a distributed system. All nodes are capable of sending and receiving messages to and from each other. Nodes can be honest, faulty, or malicious, and they have memory and a processor. A node that exhibits irrational behavior is also known as a *Byzantine node* after the **Byzantine Generals** problem.



The Byzantine Generals problem

In 1982, a thought experiment was proposed by Lamport et al. in their research paper, *The Byzantine Generals Problem*, which is available here:

<https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/>

In this problem, a group of army generals who lead different parts of the Byzantine army is planning to attack or retreat from a city. The only way of communicating among them is via a messenger. They need to agree to strike at the same time in order to win. The issue is that one or more generals might be traitors who could send a misleading message. Therefore, there is a need for a viable mechanism that allows for agreement among the generals, even in the presence of the treacherous ones, so that the attack can still take place at the same time. As an analogy for distributed systems, the generals can be considered honest nodes, the traitors as Byzantine nodes (that is, nodes with arbitrary behavior), and the messenger can be thought of as a channel of communication among the generals.

This problem was solved in 1999 by Castro and Liskov who presented the **Practical Byzantine Fault Tolerance (PBFT)** algorithm, which solves the consensus problem in the presence of Byzantine faults in asynchronous networks by utilizing the state machine replication protocol. PBFT goes through a number of rounds to eventually reach an agreement between nodes on the proposed value. PBFT and other consensus protocols will be discussed in greater detail in *Chapter 5, Consensus Algorithms*.

This type of inconsistent behavior of Byzantine nodes can be intentionally malicious, which is detrimental to the operation of the network. Any unexpected behavior by a node on the network, whether malicious or not, can be categorized as Byzantine.

A small-scale example of a distributed system is shown in the following diagram. This distributed system has six nodes, out of which one (**N4**) is a Byzantine node leading to possible data inconsistency. **L2** is a link that is broken or slow, and this can lead to a partition in the network:

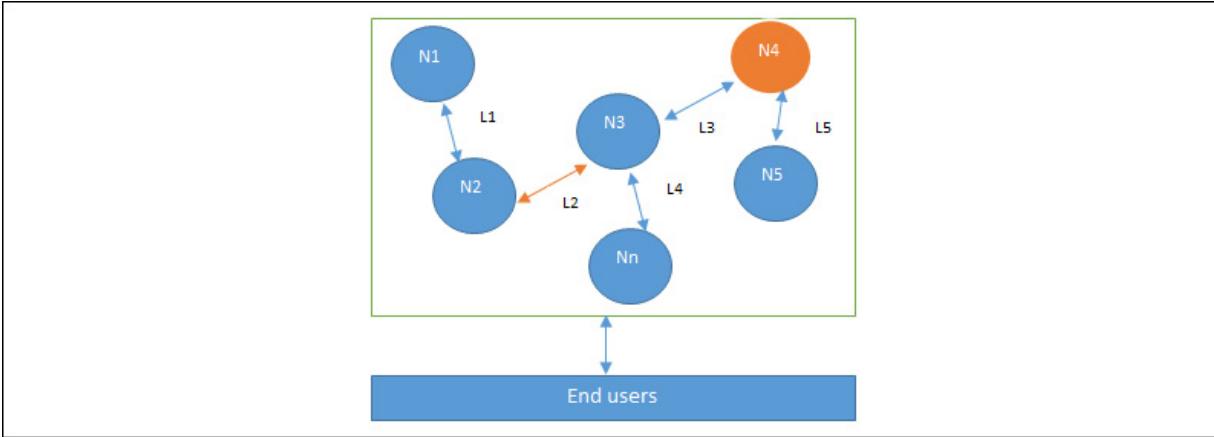


Figure 1.3: Design of a distributed system: N4 is a Byzantine node and L2 is broken or a slow network link

The primary challenge of a distributed system design is the coordination between nodes and fault tolerance. Even if some (a certain threshold dictated by the consensus protocol) of the nodes become faulty or network links break, the distributed system should be able to tolerate this and continue to work to achieve the desired result. This problem has been an active area of distributed system design research for many years, and several algorithms and mechanisms have been proposed to overcome these issues.

Distributed systems are so challenging to design that a theory known as the **CAP theorem** has been proven, which states that a distributed system cannot have all three of the much-desired properties simultaneously; that is, consistency, availability, and partition tolerance. We will dive into the CAP theorem in more detail later in this chapter.

Even though blockchain can be considered to be both a distributed and decentralized system, there are, however, critical differences between distributed systems and decentralized systems that make both of these systems architecturally different. We will discuss these differences in detail in *Chapter 2, Decentralization*.

With a better understanding of distributed systems, let's now move on to talking about blockchain itself. We'll begin with a brief rundown of the history of blockchain and Bitcoin.

The history of blockchain and Bitcoin

Blockchain was introduced with the invention of Bitcoin in 2008. Its practical implementation then occurred in 2009. For the purposes of this chapter, it is sufficient to review Bitcoin very briefly, as it will be explored in great depth in *Chapter 6, Introducing Bitcoin*. However, it is essential to refer to Bitcoin because, without it, the history of blockchain is not complete.

The events that led to blockchain

Now we will look at the early history of computing and computer networks and will discuss how these technologies evolved and contributed to the development of Bitcoin in 2008. We can view this in chronological order:

- 1960s – Invention of computer networks
- 1969 – Development of ARPANET
- 1970s – Early work on secure network communication including public key cryptography
- 1970s – Cryptographic hash functions
- 1973 – Extension of ARPANET to other geographic locations
- 1974 – First internet service provider, Telenet
- 1976 – Diffie–Hellman work on securely exchanging cryptographic keys
- 1978 – Invention of public key cryptography
- 1979 – Invention of Merkle Trees (hashes in a tree structure) by Ralph C. Merkle
- 1980s – Development of TCP/IP
- 1980 – Protocols for public key cryptosystems, Ralph C. Merkle
- 1982 – Blind signatures proposed by David Chaum

- 1982 – The Byzantine Generals Problem (Bitcoin can be considered a solution to the Byzantine Generals Problem; however, the original intention of the Bitcoin network was to address the previously unsolved double-spending problem)
- 1985 – Work on elliptic curve cryptography by Neal Koblitz and Victor Miller
- 1991 – Haber and Stornetta work on tamper proofing document timestamps. This can be considered the earliest idea of a chain of blocks or hash chains
- 1992 – Cynthia Dwork and Moni Naor publish *Pricing via Processing or Combatting Junk Mail*. This is considered the first use of **Proof of Work (PoW)**
- 1993 – Haber, Bayer, and Stornetta upgraded the tamper-proofing of document timestamps system with Merkle trees
- 1995 – David Chaum's DigiCash system (an anonymous electronic cash system) started to be used in some banks
- 1998 – Bit Gold, a mechanism for decentralized digital currency, invented by Nick Szabo. It used hash chaining and Byzantine Quorums
- 1999 – Emergence of a file-sharing application mainly used for music sharing, Napster, which is a P2P network, but was centralized with the use of indexing servers
- 1999 – Development of a secure timestamping service for the Belgian project TIMESEC
- 2000 – Gnutella file-sharing network, which introduced decentralization
- 2001 – Emergence of BitTorrent and **Distributed Hash Tables (DHTs)**
- 2002 – Hashcash by Adam Back
- 2004 – Development of B-Money by Wei Dei using hashcash
- 2004 – Hal Finney, the invention of the reusable PoW system
- 2005 – Prevention of Sybil attacks by using computation puzzles, due to James Aspnes et al.
- 2009 – Bitcoin (first blockchain)

The aforementioned technologies contributed in some way to the development of Bitcoin, even if not directly; the work is relevant to the problem that Bitcoin solved. All previous attempts to create anonymous and decentralized digital currency were successful to some extent, but they could not solve the problem of preventing double spending in a completely trustless or permissionless environment. This problem was finally addressed by the Bitcoin blockchain, which introduced the Bitcoin cryptocurrency.

It should be noted that other concepts such as **state machine replication** (the SMR problem), introduced in 1978 by Leslie Lamport and formalized in 1980 by Fred Schneider, are also solvable by Bitcoin. Bitcoin solves the SMR problem (probabilistically) by allowing the replication of blocks and ensuring consistency via its PoW consensus mechanism.



The SMR, or state machine replication problem, is a technique used to provide fault-tolerant replication in distributed systems. We will explore this in greater detail in *Chapter 5, Consensus Algorithms*.

Now we will discuss some of the major milestones in the history of blockchain in more detail.

Electronic cash

The concept of **electronic cash (e-cash)**, or digital currency, is not new. Since the 1980s, e-cash protocols have existed that are based on a model proposed by David Chaum.

Just as understanding the concept of distributed systems is necessary to comprehend blockchain technology, the idea of e-cash is also essential in order to appreciate the first, and astonishingly successful, application of blockchain, Bitcoin, and more broadly, cryptocurrencies in general.

Two fundamental e-cash system issues need to be addressed: *accountability* and *anonymity*.

Accountability is required to ensure that cash is spendable only once (addressing the double-spending problem) and that it can only be spent by its rightful owner. The double-spending problem arises when the same money can be spent twice. As it is quite easy to make copies of digital data, this becomes a big issue in digital currencies as you can make many copies of the same amount of digital cash.

Anonymity is required to protect users' privacy. With physical cash, it is almost impossible to trace back spending to the individual who actually paid the money, which provides adequate privacy should the consumer choose to hide their identity. In the digital world, however, providing such a level of privacy is difficult due to inherent personalization, tracing, and logging mechanisms in digital payment systems such as credit card payments. This is indeed a required feature for ensuring the security and safety of the financial network, but it is also often seen as a breach of privacy.

This is due to the fact that end users do not have any control over who their data might be shared with, even without their consent. Nevertheless, this is a solvable problem and cryptography is used to address such issues. Especially in blockchain networks, the privacy and anonymity of the participants on the blockchain are sought-after features. We will learn more about this in *Chapter 4, Public Key Cryptography*.

David Chaum solved both of these problems during his work in the 1980s by using two cryptographic operations, namely, **blind signatures** and **secret sharing**. These terminologies and related concepts will be discussed in detail in *Chapter 4, Public Key Cryptography*. For the moment, it is sufficient to say that *blind signatures* allow for signing a document without actually seeing it, and *secret sharing* is a concept that enables the detection of double-spending, that is, using the same e-cash token twice.

In 2009, the first practical implementation of an e-cash system named Bitcoin appeared. The term cryptocurrency emerged later. For the very first time, it solved the problem of distributed consensus in a trustless network. It used **public key cryptography** with a PoW mechanism to provide a secure, controlled, and decentralized method of minting digital currency. The key innovation was the idea of an ordered list of blocks composed of

transactions which is cryptographically secured by the PoW mechanism to prevent double-spending in a trustless environment. This concept will be explained in greater detail in *Chapter 6, Introducing Bitcoin*.

Other technologies used in Bitcoin, but which existed before its invention, include Merkle trees, hash functions, and hash chains. All these concepts are explained in appropriate depth in *Chapter 4, Public Key Cryptography*.

Looking at all the technologies mentioned previously and their relevant history, it is easy to see how concepts from e-cash schemes and distributed systems were combined to create Bitcoin and what now is known as blockchain. This concept can also be visualized with the help of the following diagram:

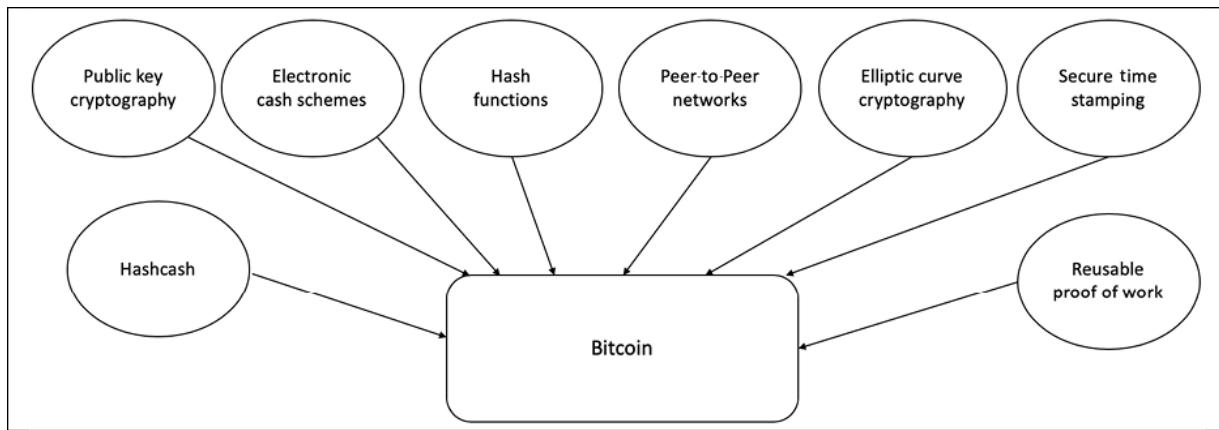


Figure 1.4: The various ideas that supported the invention of Bitcoin and blockchain

With the emergence of e-cash covered, along with the ideas that led to the formation of Bitcoin and blockchain, we can now begin to discuss blockchain itself.

Blockchain

In 2008, a groundbreaking paper, entitled *Bitcoin: A Peer-to-Peer Electronic Cash System*, was written on the topic of peer-to-peer e-cash under the pseudonym of *Satoshi Nakamoto*.

This paper is available at <https://bitcoin.org/bitcoin.pdf>.

It introduced the term **chain of blocks**. No one knows the actual identity of Satoshi Nakamoto. After introducing Bitcoin in 2009, he remained active in the Bitcoin developer community until 2011. He then handed over Bitcoin development to its core developers and simply disappeared. Since then, there has been no communication from him whatsoever, and his existence and identity are shrouded in mystery. The term "chain of blocks" evolved over the years into the word "blockchain."

As stated previously, blockchain technology incorporates a multitude of applications that can be implemented in various economic sectors. Particularly in the finance sector, significant improvement in the performance of financial transactions and settlements manifests as highly desirable time-and-cost reductions. Additional light will be shed on these aspects of blockchain in *Chapter 19, Blockchain – Outside of Currencies*, where practical use cases will be discussed in detail for various industries. For now, it is sufficient to say that parts of nearly all economic sectors have already realized the potential and promise of blockchain, and have embarked, or will do so soon, on the journey to capitalize on the benefits of blockchain technology.

Blockchain defined

A good place to start learning what blockchain is would be to see its definition. There are some different ways that blockchain may be defined; following are two of the most widely accepted definitions:



Layman's definition: Blockchain is an ever-growing, secure, shared recordkeeping system in which each user of the data holds a copy of the records, which can only be updated if all parties involved in a transaction agree to update.

Technical definition: Blockchain is a peer-to-peer, distributed ledger that is cryptographically secure, append-only, immutable (extremely hard to change), and updateable only via consensus or agreement among peers.

Now, let's examine things in some more detail. We will look at the keywords from the technical definition one by one.

Peer-to-peer

The first keyword in the technical definition is **peer-to-peer**, or **P2P**. This means that there is no central controller in the network, and all participants (nodes) talk to each other directly. This property allows for transactions to be conducted directly among the peers without third-party involvement, such as by a bank.

Distributed ledger

Dissecting the technical definition further reveals that blockchain is a "distributed ledger," which means that a ledger is spread across the network among all peers in the network, and each peer holds a copy of the complete ledger.

Cryptographically secure

Next, we see that this ledger is "cryptographically secure," which means that cryptography has been used to provide security services that make this ledger secure against tampering and misuse. These services include non-repudiation, data integrity, and data origin authentication. You will see how this is achieved later in *Chapter 4, Public Key Cryptography*, which introduces the fascinating world of cryptography.

Append-only

Another property that we encounter is that blockchain is "append-only," which means that data can only be added to the blockchain in *time-sequential order*. This property implies that once data is added to the blockchain, it is almost impossible to change that data and it can be considered practically immutable. In other words, blocks added to the blockchain cannot be changed, which allows blockchain to become an immutable and tamper-proof ledger of transactions.

However, remember that it can be changed in rare scenarios wherein collusion against the blockchain network by bad actors succeeds in gaining more than 51 percent of the power. Otherwise, the blockchain is practically immutable.



There may be some legitimate reasons to change data in the blockchain once it has been added, such as the "right to be forgotten" or "right to erasure" (also defined in the GDPR ruling: <https://gdpr-info.eu/art-17-gdpr/>).

However, those are individual cases that need to be handled separately and that require an elegant technical solution. For all practical purposes, blockchain is indeed immutable and cannot be changed.

Updatable via consensus

The most critical attribute of a blockchain is that it is updateable only via consensus. This is what gives it the power of decentralization. In this scenario, no central authority is in control of updating the ledger. Instead, any update made to the blockchain is validated against strict criteria defined by the blockchain protocol and added to the blockchain only after a consensus has been reached among all participating peers/nodes on the network. To achieve consensus, there are various consensus facilitation algorithms that ensure all parties agree on the final state of the data on the blockchain network and resolutely agree upon it to be true. Consensus algorithms are introduced later in this chapter, and then in more detail in *Chapter 5, Consensus Algorithms*.

Blockchain architecture

Having detailed the primary features of blockchain, we are now in a position to begin to look at its actual architecture. We'll begin by looking at how blockchain acts as a layer within a distributed peer-to-peer network.

Blockchain by layers

Blockchain can be thought of as a layer of a distributed peer-to-peer network running on top of the internet, as can be seen in the following diagram. It is analogous to SMTP, HTTP, or FTP running on top of TCP/IP:

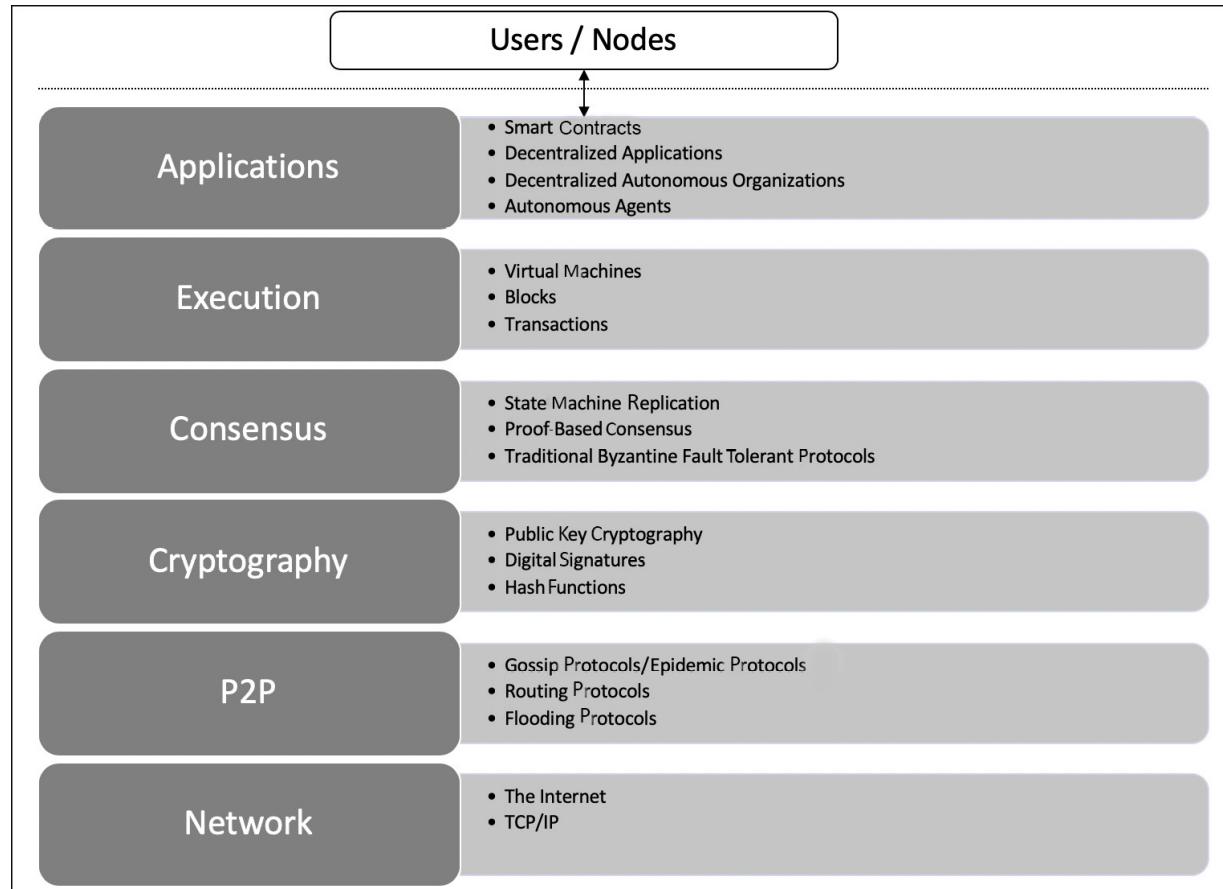


Figure 1.5: The architectural view of a generic blockchain

Now we'll discuss all these elements one by one:

- In the preceding diagram, the lowest layer is the **Network**, which is usually the internet and provides a base communication layer for any blockchain.
- A peer-to-peer network runs on top of the **Network** layer, which consists of information propagation protocols such as gossip or flooding protocols.
- After this comes the **Cryptography** layer, which contains crucial cryptographic protocols that ensure the security of the blockchain. These cryptographic protocols play a vital role in the integrity of

blockchain processes, secure information dissemination, and blockchain consensus mechanisms. This layer consists of public key cryptography and relevant components such as digital signatures and cryptographic hash functions. Sometimes, this layer is abstracted away, but it has been included in the diagram because it plays a fundamental role in blockchain operations.

- Next comes the **Consensus** layer, which is concerned with the usage of various consensus mechanisms to ensure agreement among different participants of the blockchain. This is another crucial part of the blockchain architecture, which consists of various techniques such as SMR, proof-based consensus mechanisms, or traditional (from traditional distributed systems research) Byzantine fault-tolerant consensus protocols.
- Further to this, we have the **Execution** layer, which can consist of virtual machines, blocks, transaction, and smart contracts. This layer, as the name suggests, provides executions services on the blockchain and performs operations such as value transfer, smart contract execution, and block generation. Virtual machines such as **Ethereum Virtual Machine (EVM)** provide an execution environment for smart contracts to execute.
- Finally, we have the **Applications** layer, which is composed of smart contracts, decentralized applications, DAOs, and autonomous agents. This layer can effectively contain all sorts of various user level agents and programs that operate on the blockchain. Users interact with the blockchain via decentralized applications. We will discuss more about decentralized applications in *Chapter 2, Decentralization*.

All these concepts will be discussed in detail later in this book in various chapters. Next, we'll look at blockchain from more of a business-oriented perspective.

Blockchain in business

From a business standpoint, a blockchain can be defined as a platform where peers can exchange value/e-cash using transactions without the need for a centrally trusted arbitrator. For example, for cash transfers, banks act

as a trusted third party. In financial trading, a central clearing house acts as a trusted third party between two or more trading parties. This concept is compelling, and, once you absorb it, you will realize the enormous potential of blockchain technology. This disintermediation allows blockchain to be a decentralized consensus mechanism where no single authority is in charge of the database. Immediately, you'll see a significant benefit of decentralization here, because if no banks or central clearing houses are required, then it immediately leads to cost savings, faster transaction speeds, and more trust.

We've now looked at what blockchain is at a fundamental level. Next, we'll go a little deeper and look at some of the elements that comprise a blockchain.

Generic elements of a blockchain

Now, let's walk through the generic elements of a blockchain. You can use this as a handy reference section if you ever need a reminder about the different parts of a blockchain. More precise elements will be discussed in the context of their respective blockchains in later chapters, for example, the Ethereum blockchain. The structure of a generic blockchain can be visualized with the help of the following diagram:

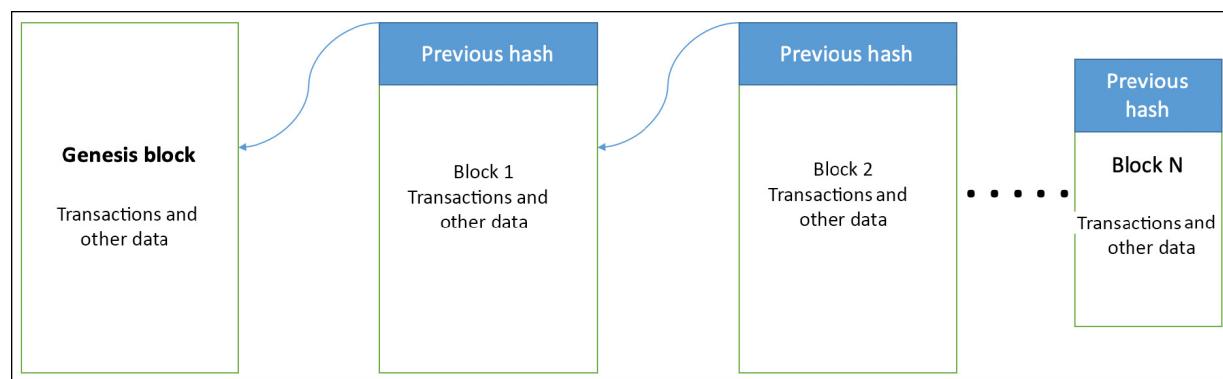


Figure 1.6: Generic structure of a blockchain

Elements of a generic blockchain are described here one by one. These are the elements that you will come across in relation to blockchain:

- **Address:** Addresses are unique identifiers used in a blockchain transaction to denote senders and recipients. An address is usually a public key or derived from a public key.
- **Transaction:** A transaction is the fundamental unit of a blockchain. A transaction represents a transfer of value from one address to another.
- **Block:** A block is composed of multiple transactions and other elements, such as the previous block hash (hash pointer), timestamp, and nonce. A block is composed of a block header and a selection of transactions bundled together and organized logically. A block contains several elements, which we introduce as follows:
 - A reference to a previous block is also included in the block unless it is a genesis block. This reference is the hash of the header of the previous block. A **genesis block** is the first block in the blockchain that is hardcoded at the time the blockchain was first started. The structure of a block is also dependent on the type and design of a blockchain.
 - A **nonce** is a number that is generated and used only once. A nonce is used extensively in many cryptographic operations to provide replay protection, authentication, and encryption. In blockchain, it's used in PoW consensus algorithms and for transaction replay protection. A block also includes the nonce value.
 - A **timestamp** is the creation time of the block.
 - A **Merkle root** is a hash of all of the nodes of a Merkle tree. In a blockchain block, it is the combined hash of the transactions in the block. Merkle trees are widely used to validate large data structures securely and efficiently. In the blockchain world, Merkle trees are commonly used to allow efficient verification of transactions. Merkle root in a blockchain is present in the block header section of a block, which is the hash of all transactions in a block. This means that verifying only the Merkle root is required to verify all transactions present in the Merkle tree instead of verifying all transactions one by one. We will elaborate further on these concepts in *Chapter 4, Public Key Cryptography*.

- In addition to the block header, the block contains transactions that make up the block body. A **transaction** is a record of an event, for example, the event of transferring cash from a sender's account to a beneficiary's account. A block contains transactions and its size varies depending on the type and design of the blockchain.

The following structure is a simple block diagram that depicts a block. Specific block structures relative to their blockchain technologies will be discussed later in the book with greater in-depth technical detail:

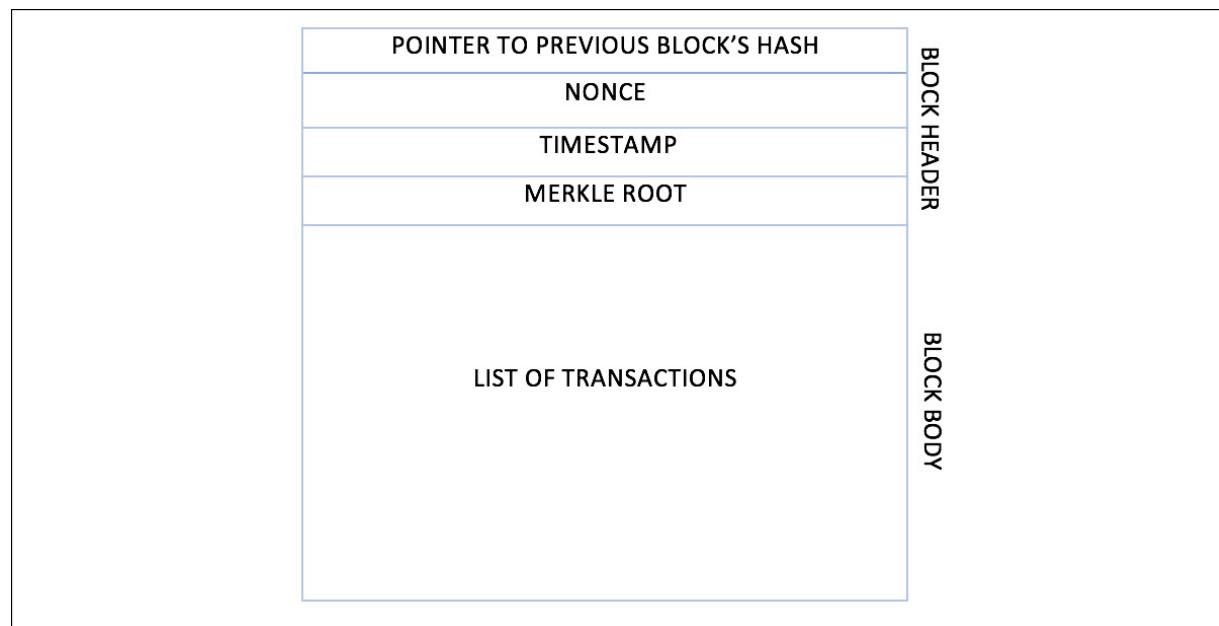


Figure 1.7: The generic structure of a block



Generally, however, there are just a few attributes that are essential to the functionality of a block: the block header, which is composed of the hash of the previous block's header, the timestamp, nonce, Merkle root, and the block body that contains the transactions. There are also other attributes in a block, but generally, the components introduced in this section are usually available in a block.

- **Peer-to-peer network:** As the name implies, a *peer-to-peer network* is a network topology wherein all peers can communicate with each other and send and receive messages.

- **The scripting or programming language:** *Scripts or programs* perform various operations on a transaction in order to facilitate various functions. For example, in Bitcoin, transaction scripts are predefined in a language called **Script**, which consists of sets of commands that allow nodes to transfer bitcoins from one address to another. Script is a limited language, in the sense that it only allows essential operations that are necessary for executing transactions, but it does not allow for arbitrary program development.



Think of the scripting language as a calculator that only supports standard preprogrammed arithmetic operations. As such, the Bitcoin Script language cannot be called "Turing complete." In simple words, a Turing complete language means that it can perform any computation. It is named after Alan Turing, who developed the idea of a Turing machine that can run any algorithm however complex. Turing complete languages need loops and branching capability to perform complex computations. Therefore, Bitcoin's scripting language is not Turing complete, whereas Ethereum's Solidity language is.

To facilitate arbitrary program development on a blockchain, a Turing complete programming language is needed, and it is now a very desirable feature to have for blockchains. Think of this as a computer that allows the development of any program using programming languages. Nevertheless, the security of such languages is a crucial question and an essential and ongoing research area. We will discuss this in greater detail in *Chapter 6, Introducing Bitcoin*, *Chapter 10, Smart Contracts*, and the chapters on *Ethereum Development*, later in this book.

- **Virtual machine:** This is an extension of the transaction script introduced previously. A *virtual machine* allows Turing complete code to be run on a blockchain (as smart contracts); whereas a transaction script is limited in its operation. However, virtual machines are not available on all blockchains. Various blockchains use virtual machines to run programs such as **Ethereum Virtual Machine (EVM)** and **Chain Virtual Machine (CVM)**. EVM is used in the Ethereum blockchain, while CVM is a virtual machine developed for and used in an enterprise-grade blockchain called "Chain Core."

- **State machine:** A blockchain can be viewed as a state transition mechanism whereby a state is modified from its initial form to the next one by nodes on the blockchain network as a result of transaction execution.
- **Smart contracts:** These programs run on top of the blockchain and encapsulate the business logic to be executed when certain conditions are met. These programs are enforceable and automatically executable. The *smart contract* feature is not available on all blockchain platforms, but it is now becoming a very desirable feature due to the flexibility and power that it provides to blockchain applications. Smart contracts have many use cases, including but not limited to identity management, capital markets, trade finance, record management, insurance, and e-governance. Smart contracts will be discussed in more detail in *Chapter 10, Smart Contracts*.
- **Node:** A *node* in a blockchain network performs various functions depending on the role that it takes on. A node can propose and validate transactions and perform mining to facilitate consensus and secure the blockchain. This goal is achieved by following a **consensus protocol** (most commonly PoW). Nodes can also perform other functions such as simple payment verification (lightweight nodes), validation, and many other functions depending on the type of the blockchain used and the role assigned to the node. Nodes also perform a transaction signing function. Transactions are first created by nodes and then also digitally signed by nodes using private keys as proof that they are the legitimate owner of the asset that they wish to transfer to someone else on the blockchain network. This asset is usually a token or virtual currency, such as Bitcoin, but it can also be any real-world asset represented on the blockchain by using tokens. There are also now standards related to tokens; for example, on Ethereum, there are ERC20, ERC721, and a few others that define the interfaces and semantics of tokenization. We will cover these in *Chapter 12, Further Ethereum*.

A high-level diagram of blockchain architecture highlighting the key elements mentioned previously is shown as follows:

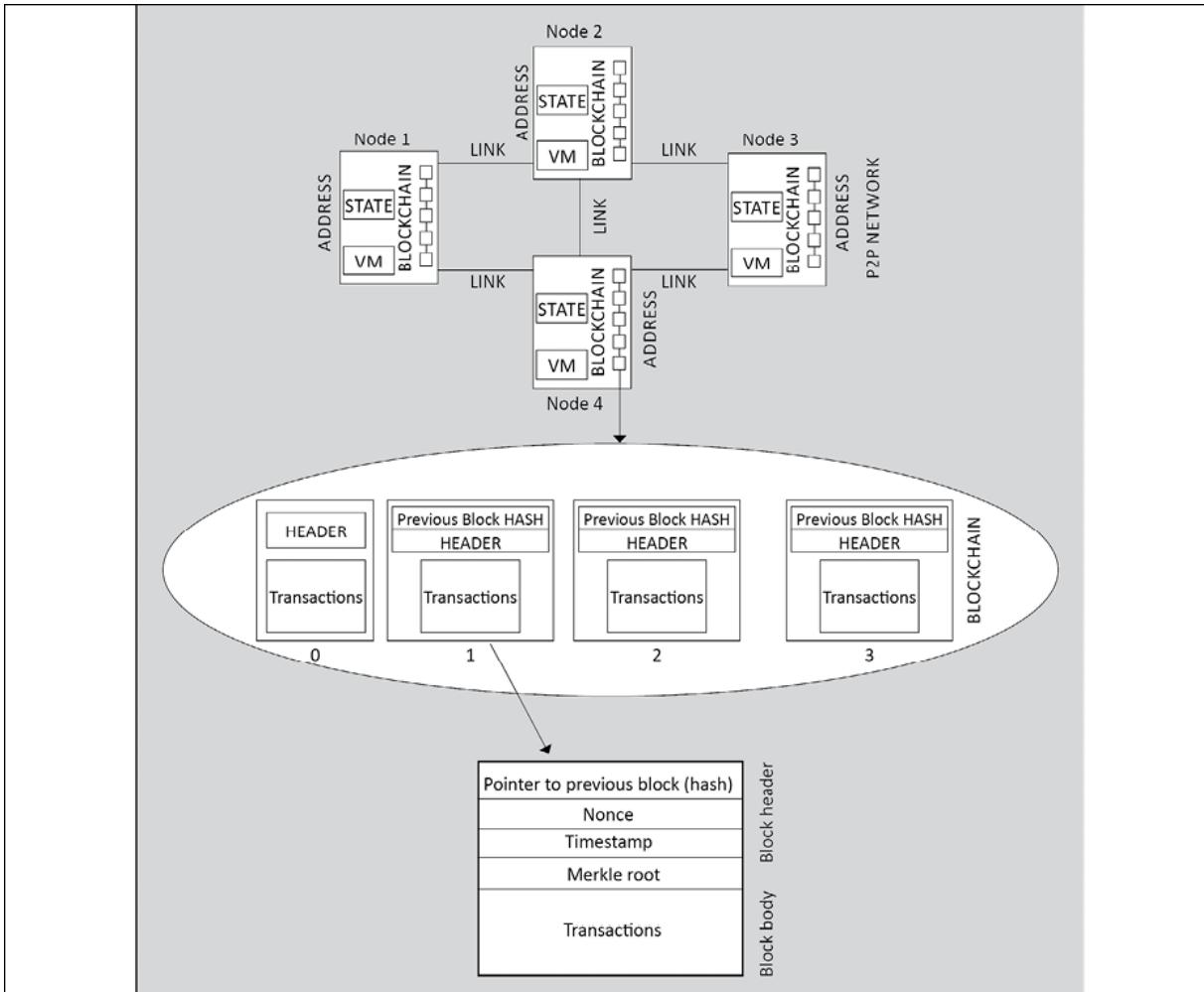


Figure 1.8: Generic structure of a blockchain network

The preceding diagram shows a four-node blockchain network (top), each maintaining a chain of blocks, virtual machine, state machine, and address. The blockchain is then further magnified (middle) to show the structure of the chain of blocks, which is again magnified (bottom) to show the structure of a transaction. Note that this is a generic structure of a blockchain; we will see specific blockchain structures in detail in the context of Ethereum and Bitcoin blockchains later in this book.

How blockchain works

We have now defined and described blockchain. Now, let's see how a blockchain actually works. Nodes are either *miners* who create new blocks

and mint cryptocurrency (coins) or *block signers* who validate and digitally sign the transactions. A critical decision that every blockchain network has to make is to figure out which node will append the next block to the blockchain. This decision is made using a *consensus mechanism*. The consensus mechanism will be described later in this chapter. For now, we will look at how a blockchain validates transactions and creates and adds blocks to grow the blockchain.

We will look at a general scheme for creating blocks. This scheme is presented here to give you a general idea of how blocks are generated and what the relationship is between transactions and blocks:

1. **Transaction is initiated:** A node starts a transaction by first creating it and then digitally signing it with its private key. A transaction can represent various actions in a blockchain. Most commonly, this is a data structure that represents the transfer of value between users on the blockchain network. The transaction data structure usually consists of some logic of transfer of value, relevant rules, source and destination addresses, and other validation information. Transactions are usually either a cryptocurrency transfer (transfer of value) or smart contract invocation that can perform any desired operation. A transaction occurs between two or more parties. This will be covered in more detail in specific chapters on Bitcoin and Ethereum later in the book.
2. **Transaction is validated and broadcast:** A transaction is propagated (broadcast) usually by using data-dissemination protocols, such as *Gossip protocol*, to other peers that validate the transaction based on preset validity criteria. Before a transaction is propagated, it is also verified to ensure that it is valid.
3. **Find new block:** When the transaction is received and validated by special participants called miners on the blockchain network, it is included in a block, and the process of mining starts. This process is also sometimes referred to as "finding a new block." Here, nodes called miners race to finalize the block they've created by a process known as mining.
4. **New block found:** Once a miner solves a mathematical puzzle (or fulfills the requirements of the consensus mechanism implemented in a

blockchain), the block is considered "found" and finalized. At this point, the transaction is considered confirmed. Usually, in cryptocurrency blockchains such as Bitcoin, the miner who solves the mathematical puzzle is also rewarded with a certain number of coins as an incentive for their effort and the resources they spent in the mining process.

5. **Add new block to the blockchain:** The newly created block is validated, transactions or smart contracts within it are executed, and it is propagated to other peers. Peers also validate and execute the block. It now becomes part of the blockchain (ledger), and the next block links itself cryptographically back to this block. This link is called a hash pointer.

This process can be visualized in the diagram as follows:

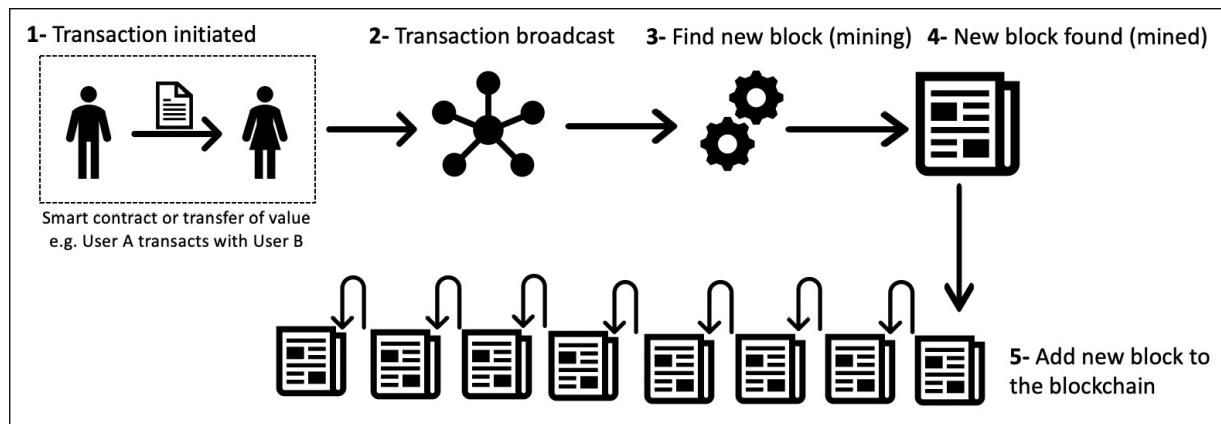


Figure 1.9: How a block is generated

This completes the basic introduction to blockchain. In the next section, you will learn about the benefits and limitations of this technology.

Benefits, features, and limitations of blockchain

Numerous advantages of blockchain technology have been discussed in many industries and proposed by thought leaders around the world who are

participating in the blockchain space. The notable benefits of blockchain technology are as follows:

1. **Decentralization:** This is a core concept and benefit of blockchain. There is no need for a trusted third party or intermediary to validate transactions; instead, a consensus mechanism is used to agree on the validity of transactions.
2. **Transparency and trust:** As blockchains are shared and everyone can see what is on the blockchain, this allows the system to be transparent. As a result, trust is established. This is more relevant in scenarios such as the disbursement of funds or benefits where personal discretion in relation to selecting beneficiaries needs to be restricted.
3. **Immutability:** Once the data has been written to the blockchain, it is extremely difficult to change it back. It is not genuinely immutable, but because changing data is so challenging and nearly impossible, this is seen as a benefit to maintaining an immutable ledger of transactions.
4. **High availability:** As the system is based on thousands of nodes in a peer-to-peer network, and the data is replicated and updated on every node, the system becomes highly available. Even if some nodes leave the network or become inaccessible, the network as a whole continues to work, thus making it highly available. This redundancy results in high availability.
5. **Highly secure:** All transactions on a blockchain are cryptographically secured and thus provide network integrity. Any transactions posted from the nodes on the blockchain are verified based on a predetermined set of rules. Only valid transactions are selected for inclusion in a block. The blockchain is based on proven cryptographic technology that ensures the integrity and availability of data.
Generally, confidentiality is not provided due to the requirements of transparency. This limitation is the leading barrier to its adoption by financial institutions and other industries that require the privacy and confidentiality of transactions. As such, the privacy and confidentiality of transactions on the blockchain are being researched very actively, and advancements are already being made. It could be argued that, in many situations, confidentiality is not needed and transparency is preferred. For example, with Bitcoin, confidentiality is not an absolute

requirement; however, it is desirable in some scenarios. A more recent example is Zcash (<https://z.cash>), which provides a platform for conducting anonymous transactions. Other security services, such as non-repudiation and authentication, are also provided by blockchain, as all actions are secured using private keys and digital signatures.

6. **Simplification of current paradigms:** The current blockchain model in many industries, such as finance or health, is somewhat disorganized. In this model, multiple entities maintain their own databases and data sharing can become very difficult due to the disparate nature of the systems. However, as a blockchain can serve as a single shared ledger among many interested parties, this can result in simplifying the model by reducing the complexity of managing the separate systems maintained by each entity.
7. **Faster dealings:** In the financial industry, especially in post-trade settlement functions, blockchain can play a vital role by enabling the quick settlement of trades. Blockchain does not require a lengthy process of verification, reconciliation, and clearance because a single version of agreed-upon data is already available on a shared ledger between financial organizations.
8. **Cost-saving:** As no trusted third party or clearing house is required in the blockchain model, this can massively eliminate overhead costs in the form of the fees, which are paid to such parties.
9. **Platform for smart contracts:** A blockchain is a platform on which programs can run that execute business logic on behalf of the users. This is a very useful feature but not all blockchains have a mechanism to execute *smart contracts*; however, this is a very desirable feature. It is available on newer blockchain platforms such as Ethereum and MultiChain, but not on Bitcoin.

Smart contracts

Blockchain technology provides a platform for running smart contracts. These are automated, autonomous programs that reside on the blockchain network and encapsulate the business logic and code needed to execute a required function when certain conditions are met. For example, think about an insurance contract where a claim is paid to the traveler if the flight is canceled. In the real world, this



process normally takes a significant amount of time to make the claim, verify it, and pay the insurance amount to the claimant (traveler). What if this whole process were automated with cryptographically-enforced trust, transparency, and execution so that as soon as the smart contract received a feed that the flight in question has been canceled, it automatically triggers the insurance payment to the claimant? If the flight is on time, the smart contract pays itself.

This is indeed a revolutionary feature of blockchain, as it provides flexibility, speed, security, and automation for real-world scenarios that can lead to a completely trustworthy system with significant cost reductions. Smart contracts can be programmed to perform any actions that blockchain users need and according to their specific business requirements.

10. **Smart property:** It is possible to link a digital or physical asset to the blockchain in such a secure and precise manner that it cannot be claimed by anyone else. You are in full control of your asset, and it cannot be double-spent or double-owned. Compare this with a digital music file, for example, which can be copied many times without any controls. While it is true that many **Digital Rights Management (DRM)**schemes are being used currently along with copyright laws, none of them are enforceable in the way a blockchain-based DRM can be. Blockchain can provide digital rights management functionality in such a way that it can be enforced fully. There are famously broken DRM schemes that looked great in theory but were hacked due to one limitation or another. One example is the Oculus hack:
<http://www.wired.co.uk/article/oculus-rift-drm-hacked>. Another example is the PS3 hack; also, copyrighted digital music, films, and e-books are routinely shared on the internet without any limitations. We have had copyright protection in place for many years, but digital piracy refutes all attempts to fully enforce the law. On a blockchain, however, if you own an asset, no one else can claim it unless you decide to transfer it. This feature has far-reaching implications, especially in DRM and e-cash systems where double-spend detection is a crucial requirement. The double-spend problem was first solved without the requirement of a trusted third party in Bitcoin.

As with any technology, some challenges need to be addressed in order to make a system more robust, useful, and accessible. Blockchain technology is no exception. In fact, much effort is being made in both academia and industry to overcome the challenges posed by blockchain technology. The most sensitive blockchain problems are as follows:

- **Scalability:** Currently, blockchain networks are not as scalable as, for example, current financial networks. This is a known area of concern and a very ripe area for research.
- **Adoption:** Often, blockchain is seen as a nascent technology. Even though this perspective is rapidly changing, there is still a long way to go before the mass adoption of this technology. The challenge here is to allow blockchain networks to be easier to use so that adoption can increase. In addition, several other challenges such as scalability (introduced previously) exist, which must be solved in order to increase adoption.
- **Regulation:** Due to its decentralized nature, regulation is almost impossible on blockchain. This is sometimes seen as a barrier toward adoption because, traditionally, due to the existence of regulatory authorities, consumers have a certain level of confidence that if something goes wrong they can hold someone accountable. However, in blockchain networks, no such regulatory authority and control exists, which is an inhibiting factor for many consumers.
- **Relatively immature technology:** As compared to traditional IT systems that have benefited from decades of research, blockchain is still a new technology and requires a lot of research to achieve maturity.
- **Privacy and confidentiality:** Privacy is a concern on public blockchains such as Bitcoin where everyone can see every single transaction. This transparency is not desirable in many industries such as the financial, law, or medical sectors. This is also a known concern and a lot of valuable research with some impeccable solutions has already been developed. However, further research is still required to drive the mass adoption of blockchain.

All of these issues and possible solutions will be discussed in detail in *Chapter 21, Scalability and Other Challenges*.

You now know the basics of blockchain and its benefits and limitations. Now, let's take a look at the various types of blockchain that exist.

Types of blockchain

Based on the way that blockchain has evolved over the last few years, it can be divided into multiple categories with distinct, though sometimes partially overlapping attributes. You should note that the tiers described earlier in the chapter are a different concept, whereby the logical categorization of blockchain, based upon its evolution and usage, is presented.

In this section, we will examine the different types of blockchains from a technical and business use perspective. These blockchain types can occur on any blockchain tier, as there is no direct relationship between those tiers mentioned earlier and the various types of blockchain.

In this section, we'll examine:

- Distributed ledgers
- Distributed Ledger Technology (DLT)
- Blockchains
- Ledgers

Distributed ledgers

First, I need to clarify an ambiguity. It should be noted that a *distributed ledger* is a broad term describing shared databases; hence, all blockchains technically fall under the umbrella of shared databases or distributed ledgers. Although all blockchains are fundamentally distributed ledgers, all distributed ledgers are not necessarily blockchains.

A critical difference between a distributed ledger and a blockchain is that a distributed ledger does not necessarily consist of blocks of transactions to keep the ledger growing. Rather, a blockchain is a special type of shared database that is comprised of blocks of transactions. An example of a distributed ledger that does not use blocks of transactions is R3's Corda (<https://www.corda.net>). Corda is a distributed ledger that is developed to record and manage agreements and is especially focused on the financial services industry. On the other hand, more widely known blockchains like Bitcoin and Ethereum make use of blocks to update the shared database.

As the name suggests, a distributed ledger is distributed among its participants and spread across multiple sites or organizations. This type of ledger can be either private or public. The fundamental idea here is that, unlike many other blockchains, the records are stored contiguously instead of being sorted into blocks. This concept is used in Ripple, which is a blockchain- and cryptocurrency-based global payment network.

Distributed Ledger Technology

It should be noted that over the last few years, the terms distributed ledger or DLT have grown to be commonly used to describe blockchain in the finance industry. Sometimes, blockchain and DLT are used interchangeably. Though this is not entirely accurate, it is how the term has evolved recently, especially in the finance sector. In fact, DLT is now a very active and thriving area of research in the financial sector. From a financial sector point of view, DLTs are permissioned blockchains that are used by consortiums. DLTs usually serve as a shared database, with all participants known and verified. They do not have a cryptocurrency and do not require mining to secure the ledger.



At a broader level, DLT is an umbrella term that represents Distributed Ledger Technology as a whole, comprising of blockchains and distributed ledgers of different types.

Public blockchains

As the name suggests, public blockchains are not owned by anyone. They are open to the public, and anyone can participate as a node in the decision-making process. Users may or may not be rewarded for their participation. All users of these "permissionless" or "un-permissioned" ledgers maintain a copy of the ledger on their local nodes and use a distributed consensus mechanism to decide the eventual state of the ledger. Bitcoin and Ethereum are both considered public blockchains.

Private blockchains

As the name implies, private blockchains are just that—private. That is, they are open only to a consortium or group of individuals or organizations who have decided to share the ledger among themselves. There are various blockchains now available in this category, such as Kadena and Quorum. Optionally, both of these blockchains can also run in public mode if required, but their primary purpose is to provide a private blockchain.

Semi-private blockchains

With semi-private blockchains, part of the blockchain is private and part of it is public. Note that this is still just a concept today, and no real-world proofs of concept have yet been developed. With a semi-private blockchain, the private part is controlled by a group of individuals, while the public part is open for participation by anyone.

This hybrid model can be used in scenarios where the private part of the blockchain remains internal and shared among known participants, while the public part of the blockchain can still be used by anyone, optionally allowing mining to secure the blockchain. This way, the blockchain as a whole can be secured using PoW, thus providing consistency and validity for both the private and public parts. This type of blockchain can also be called a "semi-decentralized" model, where it is controlled by a single entity but still allows for multiple users to join the network by following appropriate procedures.

Sidechains

More precisely known as "pegged sidechains," this is a concept whereby coins can be moved from one blockchain to another and then back again. Typical uses include the creation of new *altcoins* (alternative cryptocurrencies) whereby coins are burnt as a proof of an adequate stake. "Burnt" or "burning the coins" in this context means that the coins are sent to an address that is un-spendable, and this process makes the "burnt" coins irrecoverable. This mechanism is used to bootstrap a new currency or introduce scarcity, which results in the increased value of the coin.

This mechanism is also called "Proof of Burn" and is used as an alternative method for distributed consensus to PoW and **Proof of Stake (PoS)**. The example provided previously for burning coins applies to a **one-way pegged sidechain**. The second type is called a **two-way pegged sidechain**, which allows the movement of coins from the main chain to the sidechain and back to the main chain when required.

This process enables the building of smart contracts for the Bitcoin network. Rootstock is one of the leading examples of a sidechain, which enables smart contract development for Bitcoin using this paradigm. It works by allowing a two-way peg for the Bitcoin blockchain, and this results in much faster throughput.

Permissioned ledger

A *permissioned ledger* is a blockchain where participants of the network are already known and trusted. Permissioned ledgers do not need to use a distributed consensus mechanism; instead, an agreement protocol is used to maintain a shared version of the truth about the state of the records on the blockchain. In this case, for verification of transactions on the chain, all verifiers are already preselected by a central authority and, typically, there is no need for a mining mechanism.

By definition, there is also no requirement for a permissioned blockchain to be private, as it can be a public blockchain but with regulated access control. For example, Bitcoin can become a permissioned ledger if an access control layer is introduced on top of it that verifies the identity of a user and then allows access to the blockchain.

Shared ledger

This is a generic term that is used to describe any application or database that is shared by the public or a consortium. Generally, all blockchains fall into the category of a shared ledger.

Fully private and proprietary blockchains

There is no mainstream application of these types of blockchains, as they deviate from the core concept of decentralization in blockchain technology. Nonetheless, in specific private settings within an organization, there could be a need to share data and provide some level of guarantee of the authenticity of the data.

An example of this type of blockchain might be to allow for collaboration and the sharing of data between various government departments. In that case, no complex consensus mechanism is required, apart from simple SMR and an agreement protocol with known central validators. Even in private blockchains, tokens are not really required, but they can be used as a means of transferring value or representing some real-world assets.

Tokenized blockchains

These blockchains are standard blockchains that generate cryptocurrency as a result of a consensus process via mining or initial distribution. Bitcoin and Ethereum are prime examples of this type of blockchain.

Tokenless blockchains

These blockchains are designed in such a way that they do not have the basic unit for the transfer of value. However, they are still valuable in situations where there is no need to transfer value between nodes and only the sharing of data among various trusted parties is required. This is similar to fully private blockchains, the only difference being that the use of tokens is not required. This can also be thought of as a shared distributed ledger used for storing and sharing data between the participants. It does have its

benefits when it comes to immutability, tamper proofing, security, and consensus-driven updates but is not used for a common blockchain application of value transfer or cryptocurrency. Most of the permissioned blockchains can be seen as an example of tokenless blockchains, for example, Hyperledger Fabric or Quorum. Tokens can be built on these chains as an application, but intrinsically these blockchains do not have a token associated with them.

All the aforementioned terminologies are used in literature, but fundamentally all these blockchains are distributed ledgers and fall under the top-level category of DLTs. We can view these different types in the simple chart as follows:

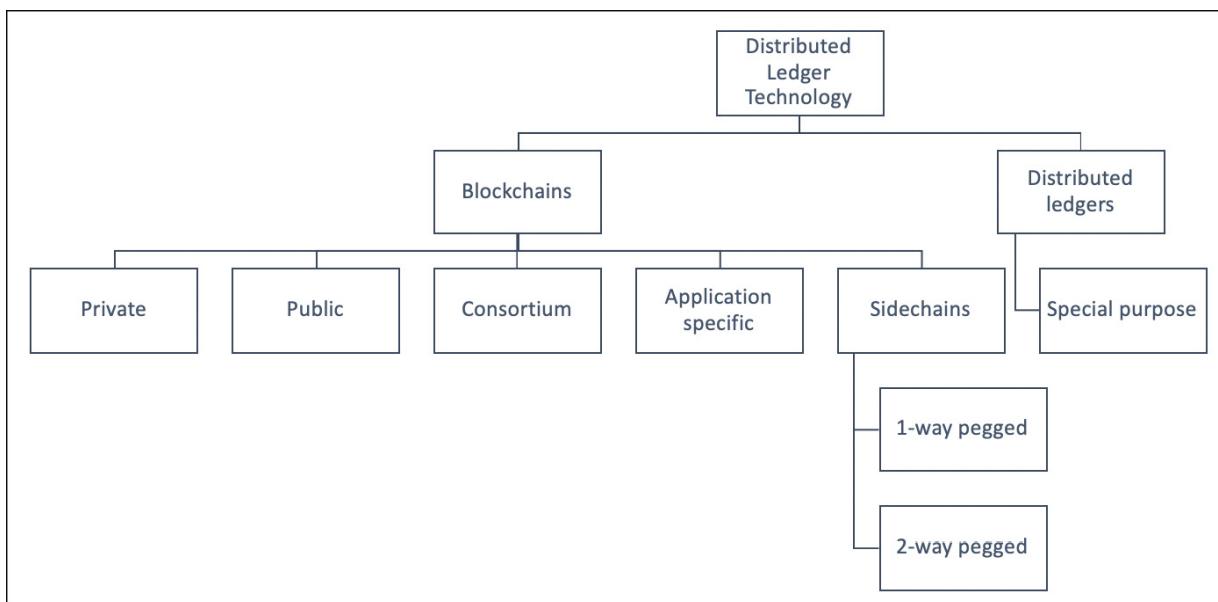


Figure 1.10: DLT hierarchy

This ends our examination of the various types of blockchain. We'll now move on to the next section to discuss the concept of consensus.

Consensus

Consensus is the backbone of a blockchain, as it provides the decentralization of control through an optional process known as **mining**. The choice of the **consensus algorithm** to utilize is governed by the type of blockchain in use; that is, not all consensus mechanisms are suitable for all types of blockchains. For example, in public permissionless blockchains, it would make sense to use PoW instead of mechanisms that are more suitable for permissioned blockchains, such as **Proof of Authority (PoA)** or traditional Byzantine fault-tolerant consensus mechanisms. Therefore, it is essential to choose an appropriate consensus algorithm for a particular blockchain project.

Consensus is a process of achieving agreement between distrusting nodes on the final state of data. To achieve consensus, different algorithms are used. It is easy to reach an agreement between two nodes (in client-server systems, for example), but when multiple nodes are participating in a distributed system and they need to agree on a single value, it becomes quite a challenge to achieve consensus. This process of attaining agreement on a common state or value among multiple nodes despite the failure of some nodes is known as **distributed consensus**.

Consensus mechanism

A **consensus mechanism** is a set of steps that are taken by most or all nodes in a blockchain to agree on a proposed state or value. For more than three decades, this concept has been researched by computer scientists in industry and academia. With the advent of blockchain and Bitcoin, consensus mechanisms have come into the limelight again and gained considerable popularity.

There are various requirements for a consensus mechanism. The following describes these requirements:

- **Agreement:** All honest nodes decide on the same value.
- **Integrity:** This is a requirement that no node can make the decision more than once in a single consensus cycle.

- **Validity:** The value agreed upon by all honest nodes must be the same as the initial value proposed by at least one honest node.
- **Fault tolerant:** The consensus algorithm should be able to run correctly in the presence of faulty or malicious nodes (Byzantine nodes).
- **Termination:** All honest nodes terminate the execution of the consensus process and eventually reach a decision.

Having seen these general requirements, we'll now look at the different types of consensus mechanisms.

Types of consensus mechanisms

All consensus mechanisms are developed to deal with faults in a distributed system and to allow distributed systems to reach a final state of agreement. There are two general categories of consensus mechanisms. These categories deal with all types of faults (fail-stop types or arbitrary). These common types of consensus mechanisms are as follows:

- **Proof-based consensus mechanisms:** This arrangement requires nodes to compete in a leader-election lottery, and the node that wins proposes the final value. The algorithm works on the principle of providing proof of some work and the possession of some authority or tokens to win the right of proposing the next block. For example, the PoW mechanism used in Bitcoin falls into this category, where a miner who solves the computational puzzle as proof of computational effort expended wins the right to add the next block to the blockchain.
- **Traditional fault tolerance-based:** With no compute-intensive operations, such as partial hash inversion (as in Bitcoin PoW), this type of consensus mechanism relies on a simple scheme of nodes that publish and verify signed messages in a number of phases. Eventually, when a certain number of messages are received over a period of rounds (phases), then an agreement is reached.

To achieve fault tolerance, replication is used. This is a standard and widely used method to achieve fault tolerance. In general, there are two types of faults that a node can experience:

- **Fail-stop faults:** This type of fault occurs when a node merely has crashed. Fail-stop faults are the easier ones to deal with of the two fault types. Paxos or the RAFT protocol, introduced earlier in this chapter, are normally used to deal with this type of fault. These faults are simpler to deal with.
- **Byzantine faults:** The second type of fault is one where the faulty node exhibits malicious or inconsistent behavior arbitrarily. This type is difficult to handle since it can create confusion due to misleading information. This can be a result of an attack by adversaries, a software bug, or data corruption. SMR protocols such as **Practical Byzantine Fault Tolerance (PBFT)** was developed to address this second type of faults.

Many other implementations of consensus protocols have been proposed in traditional distributed systems. **Paxos** is the most famous of these protocols. It was introduced by **Leslie Lamport** in 1989. With Paxos, nodes are assigned various roles such as Proposer, Acceptor, and Learner. Nodes or processes are named replicas, and consensus is achieved in the presence of faulty nodes by an agreement among a majority of nodes.

An alternative to Paxos is RAFT, which works by assigning any of three states; that is, *Follower*, *Candidate*, or *Leader* to the nodes. A Leader is elected after a Candidate node receives enough votes, and all changes then have to go through the Leader. The Leader commits the proposed changes once replication on the majority of the follower nodes is completed.

We will briefly touch on some aspects of consensus in blockchain now, but more detail on the theory of consensus mechanisms from a distributed system point of view and also from the blockchain perspective will be presented in *Chapter 5, Consensus Algorithms*.

Consensus in blockchain

Consensus is a distributed computing concept that has been used in blockchain in order to provide a means of agreeing to a single version of the truth by all peers on the blockchain network. This concept was previously discussed in the *distributed systems* section of this chapter. In this section, we will address consensus in the context of blockchain technology. Some concepts presented following are still relevant to the distributed systems theory, but they are explained from a blockchain perspective.

Roughly, the following describes the two main categories of consensus mechanisms:

1. **Proof-based, leader-election lottery-based, or the Nakamoto consensus** whereby a leader is elected at random (using an algorithm) and proposes a final value. This category is also referred to as the fully decentralized or permissionless type of consensus mechanism. This type is well used in the Bitcoin and Ethereum blockchain in the form of a PoW mechanism.
2. **Byzantine fault tolerance (BFT)-based** is a more traditional approach based on rounds of votes. This class of consensus is also known as the consortium or permissioned type of consensus mechanism.

BFT-based consensus mechanisms perform well when there are a limited number of nodes, but they do not scale well. On the other hand, leader-election lottery-based (PoW) consensus mechanisms scale very well but perform very slowly. As there is significant research being conducted in this area, new types of consensus mechanisms are also emerging, such as the semi-decentralized type, which is used in the Ripple network. The Ripple network will be discussed in detail in this book's online content pages, here:

https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf. There are also various other proposals out there, which are trying to find the right balance between scalability and performance. Some notable projects include PBFT, Hybrid BFT, BlockDAG, Tezos, Stellar, and GHOST.

The consensus algorithms available today, or that are being researched in the context of blockchain, are presented as follows. The following is not an exhaustive list, but it includes all notable algorithms:

- **Proof of Work (PoW)**: This type of consensus mechanism relies on proof that adequate computational resources have been spent before proposing a value for acceptance by the network. This scheme is used in Bitcoin, Litecoin, and other cryptocurrency blockchains. Currently, it is the only algorithm that has proven to be astonishingly successful against any collusion attacks on a blockchain network, such as the Sybil attack. The Sybil attack will be discussed in *Chapter 6, Introducing Bitcoin*.
- **Proof of Stake (PoS)**: This algorithm works on the idea that a node or user has an adequate stake in the system; that is, the user has invested enough in the system so that any malicious attempt by that user would outweigh the benefits of performing such an attack on the network. This idea was first introduced by Peercoin, and it is going to be used in the Ethereum blockchain version called *Serenity*. Another important concept in PoS is **coin age**, which is a criterion derived from the amount of time and number of coins that have not been spent. In this model, the chances of proposing and signing the next block increase with the coin age.
- **Delegated Proof of Stake (DPoS)**: This is an innovation over standard PoS, whereby each node that has a stake in the system can delegate the validation of a transaction to other nodes by voting. It is used in the BitShares blockchain.
- **Proof of Elapsed Time (PoET)**: Introduced by Intel in 2016, PoET uses a **Trusted Execution Environment (TEE)** to provide randomness and safety in the leader-election process via a guaranteed wait time. It requires the Intel **SGX (Software Guard Extensions)** processor to provide the security guarantee for it to be secure. This concept is discussed in more detail in *Chapter 17, Hyperledger*, in the context of the Intel Sawtooth Lake blockchain project.
- **Proof of Deposit (PoD)**: In this case, nodes that wish to participate in the network have to make a security deposit before they can mine and propose blocks. This mechanism is used in the Tendermint blockchain.

- **Proof of Importance (PoI):** This idea is significant and different from PoS. PoI not only relies on how large a stake a user has in the system, but it also monitors the usage and movement of tokens by the user in order to establish a level of trust and importance. It is used in the NEM coin blockchain. More information about this coin is available from NEM's website (<https://nem.io>).
- **Federated consensus or federated Byzantine consensus:** This mechanism is used in the stellar consensus protocol. Nodes in this protocol retain a group of publicly-trusted peers and propagate only those transactions that have been validated by the majority of trusted nodes.
- **Reputation-based mechanisms:** As the name suggests, a leader is elected by the reputation it has built over time on the network. It is based on the votes of other members.
- **Practical Byzantine Fault Tolerance (PBFT):** This mechanism achieves SMR, which provides tolerance against Byzantine nodes. Various other protocols, including PBFT, PAXOS, RAFT, and **Federated Byzantine Agreement (FBA)**, are also being used or have been proposed for use in many different implementations of distributed systems and blockchains.
- **Proof of Activity (PoA):** This scheme is a combination of PoS and PoW, which ensures that a stakeholder is selected in a pseudorandom but uniform fashion. This is a comparatively more energy-efficient mechanism as compared to PoW. It utilizes a new concept called "Follow the Satoshi." In this scheme, PoW and PoS are combined together to achieve consensus and a good level of security. This scheme is more energy efficient as PoW is used only in the first stage of the mechanism; after the first stage, it switches to PoS, which consumes negligible energy. We will discuss these ideas further in *Chapter 7, Bitcoin Network and Payments*, where protocols are reviewed in the context of advanced Bitcoin protocols.
- **Proof of Capacity (PoC):** This scheme uses hard disk space as a resource to mine the blocks. This is different from PoW, where CPU resources are used. In PoC, hard disk space is utilized for mining and, as such, is also known as *hard drive mining*. This concept was first introduced in the BurstCoin cryptocurrency.

- **Proof of Storage:** This scheme allows for the outsourcing of storage capacity. This scheme is based on the concept that a particular piece of data is probably stored by a node, which serves as a means to participate in the consensus mechanism. Several variations of this scheme have been proposed, such as Proof of Replication, Proof of Data Possession, Proof of Space, and Proof of Space-time.
- **Proof of Authority (PoA):** This scheme utilizes the identity of the participants called validators as a stake on the network. Validators are known and have the authority to propose new blocks. Validators propose the new blocks and validate them as per blockchain rules. Commonly used PoA algorithms are Clique and Aura.

Some prominent protocols in blockchain will be discussed in detail in *Chapter 5, Consensus Algorithms*. In this chapter, a light introduction is presented only.

Remember, earlier in this chapter we said that distributed systems are difficult to build and a distributed system cannot have consistency, availability, and partition tolerance at the same time. This is a proven result; however, in blockchain, it seems that this theorem is somehow violated. In the next section, we will introduce the CAP theorem formally and discuss why blockchain appears to achieve all three properties simultaneously.

CAP theorem and blockchain

The **CAP theorem**, also known as Brewer's theorem, was introduced by Eric Brewer in 1998 as a conjecture. In 2002, it was proven as a theorem by Seth Gilbert and Nancy Lynch. The theorem states that any distributed system cannot have consistency, availability, and partition tolerance simultaneously:

- **Consistency** is a property that ensures that all nodes in a distributed system have a single, current, and identical copy of the data.



Consistency is achieved using consensus algorithms in order to ensure that all nodes have the same copy of the data. This is also called **state machine replication**. The blockchain is a means for achieving state machine replication.

- **Availability** means that the nodes in the system are up, accessible for use, and are accepting incoming requests and responding with data without any failures as and when required. In other words, data is available at each node and the nodes are responding to requests.
- **Partition tolerance** ensures that if a group of nodes is unable to communicate with other nodes due to network failures, the distributed system continues to operate correctly. This can occur due to network and node failures.

A Venn diagram is commonly used to visualize the CAP theorem:

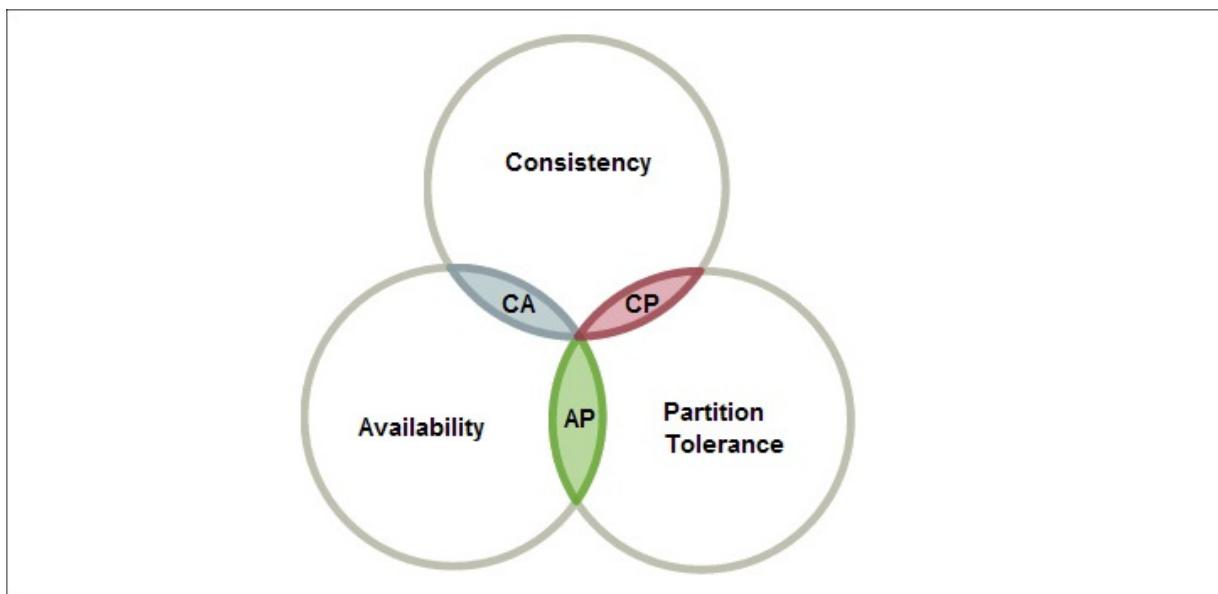


Figure 1.11: CAP theorem

The preceding diagram shows that only two properties at a time can be achieved. Either AP, CA, or CP.

In summary:

1. If we opt for **CP** (consistency and partition tolerance), we sacrifice availability.
2. If we opt for **AP** (availability and partition tolerance), we sacrifice consistency.
3. If we opt for **AC** (availability and consistency), we sacrifice partition tolerance.

Usually, a network partition cannot be ignored; therefore, the choice mostly becomes either consistency or availability in the case of a network partition.

As shown previously, a distributed system cannot have consistency, availability, and partition tolerance simultaneously. This can be explained with the following example.

Let's imagine that there is a distributed system with two nodes. Now, let's apply the three theorem properties on this smallest of possible distributed systems only with two nodes:

- **Consistency** is achieved if both nodes have the same shared state; that is, they have the same up-to-date copy of the data.
- **Availability** is achieved if both nodes are up and running and responding with the latest copy of data.
- **Partition tolerance** is achieved if, despite communication failure or delay between nodes, the network (distributed system) continues to operate.

Now think of a scenario where a partition occurs, and nodes can no longer communicate with each other. If new updated data comes in now, it can only be updated on one node only. In that case, if the node accepts the update, then only that one node in the network is updated and therefore consistency is lost. Now, if the update is rejected by the node, that would result in loss of availability. In that case, due to partition tolerance, both availability and consistency are unachievable.

This is strange because somehow blockchain manages to achieve all of these properties—or does it?

It seems that the CAP theorem is violated by blockchain, especially in its most successful implementation, Bitcoin. However, this is not the case. In blockchains, consistency is sacrificed in favor of availability and partition tolerance. In this scenario, **Consistency (C)** on the blockchain is not achieved simultaneously with **Partition tolerance (P)** and **Availability (A)**, but it is achieved over time. This is called **eventual consistency**, where consistency is achieved as a result of validation from multiple nodes over time. It means that there can be a temporary disagreement between nodes on the final state, but it is eventually agreed upon. For example, in Bitcoin, multiple transaction confirmations are required to achieve a good level of confidence that transactions may not be rolled back in the future and eventually a consistent view of transaction history is available to all nodes. Multiple confirmations of a transaction over time provide eventual consistency in Bitcoin. For this purpose, the process of mining was introduced in Bitcoin. **Mining** is a process that facilitates the achievement of consensus by using the PoW algorithm. At a higher level, mining can be defined as a process that is used to add more blocks to the blockchain. We will cover more on this later in *Chapter 6, Introducing Bitcoin*.

Summary

This chapter introduced blockchain technology at an advanced level. First, we discussed blockchain's progress toward becoming a mature technology, followed by some basic concepts about distributed systems, and then the history of blockchain was reviewed. Concepts such as e-cash were also discussed.

Various definitions of blockchain from different points of view were presented. Some applications of blockchain technology were also introduced. Next, different types of blockchain were explored. Finally, the benefits and limitations of this new technology were also examined. Some topics such as blockchain scalability and adaptability issues were intentionally introduced only lightly, as they will be discussed in depth in later chapters.

In the next chapter, we will introduce the concept of decentralization, which is central to the idea behind blockchains and their vast number of applications.

2

Decentralization

Decentralization is not a new concept. It has been in use in strategy, management, and government for a long time. The basic idea of decentralization is to distribute control and authority to the peripheries of an organization instead of one central body being in full control of the organization. This configuration produces several benefits for organizations, such as increased efficiency, expedited decision making, better motivation, and a reduced burden on top management.

In this chapter, the concept of decentralization will be discussed in the context of blockchain. The fundamental basis of blockchain is that no single central authority is in control of the network. This chapter will present examples of various methods of decentralization and ways to achieve it. Furthermore, the decentralization of the blockchain ecosystem, decentralized applications, and platforms for achieving decentralization will be discussed in detail. Many exciting applications and ideas emerge from the decentralized blockchain technology, such as decentralized finance and decentralized identity, which will be introduced in this chapter.

Decentralization using blockchain

Decentralization is a core benefit and service provided by blockchain technology. By design, blockchain is a perfect vehicle for providing a platform that does not need any intermediaries and that can function with many different leaders chosen via consensus mechanisms. This model

allows anyone to compete to become the decision-making authority. A consensus mechanism governs this competition, and the most famous method is known as **Proof of Work (PoW)**.

Decentralization is applied in varying degrees from a semi-decentralized model to a fully decentralized one depending on the requirements and circumstances. Decentralization can be viewed from a blockchain perspective as a mechanism that provides a way to remodel existing applications and paradigms, or to build new applications, to give full control to users.

Information and communication technology (ICT) has conventionally been based on a centralized paradigm whereby database or application servers are under the control of a central authority, such as a system administrator. With Bitcoin and the advent of blockchain technology, this model has changed, and now the technology exists to allow anyone to start a decentralized system and operate it with no single point of failure or single trusted authority. It can either be run autonomously or by requiring some human intervention, depending on the type and model of governance used in the decentralized application running on the blockchain.

The following diagram shows the different types of systems that currently exist: central, distributed, and decentralized. This concept was first published by Paul Baran in *On Distributed Communications: I. Introduction to Distributed Communications Networks* (Rand Corporation, 1964):

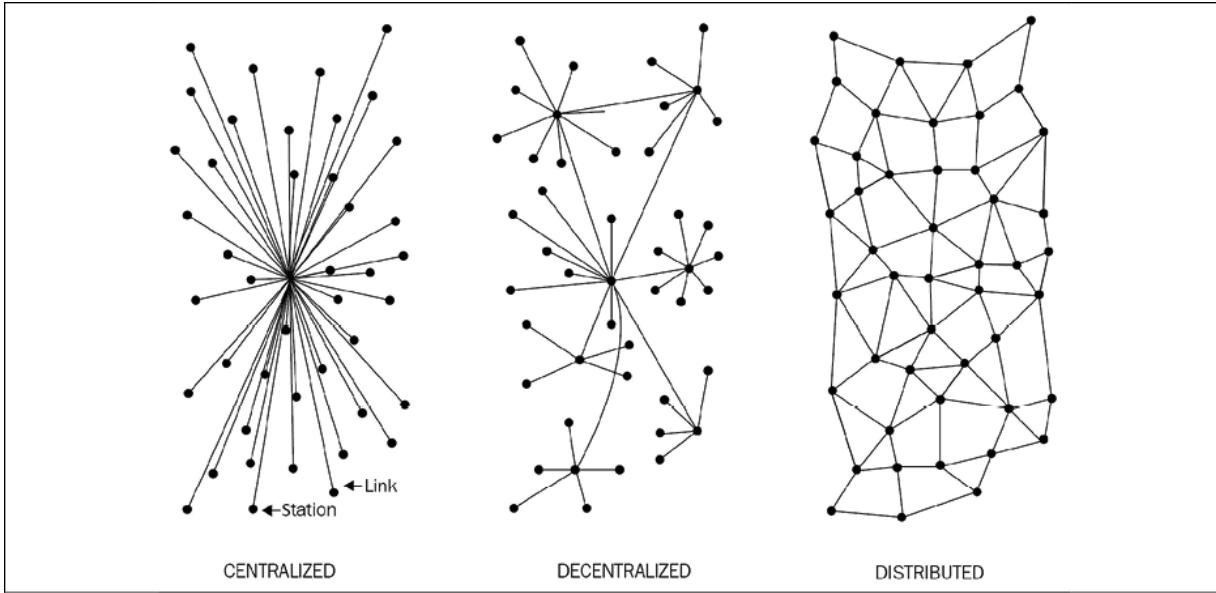


Figure 2.1: Different types of networks/systems

Centralized systems are conventional (client-server) IT systems in which there is a single authority that controls the system, and who is solely in charge of all operations on the system. All users of a centralized system are dependent on a single source of service. The majority of online service providers, including Google, Amazon, eBay, and Apple's App Store, use this conventional model to deliver services.

In a **distributed system**, data and computation are spread across multiple nodes in the network. Sometimes, this term is confused with *parallel computing*. While there is some overlap in the definition, the main difference between these systems is that in a parallel computing system, computation is performed by all nodes simultaneously in order to achieve the result; for example, parallel computing platforms are used in weather research and forecasting, simulation, and financial modeling. On the other hand, in a distributed system, computation may not happen in parallel and data is replicated across multiple nodes that users view as a single, coherent system. Variations of both of these models are used to achieve fault tolerance and speed. In the parallel system model, there is still a central authority that has control over all nodes and governs processing. This means that the system is still centralized in nature.

The critical difference between a decentralized system and distributed system is that in a distributed system, there is still a central authority that governs the entire system, whereas in a decentralized system, no such authority exists.

A **decentralized system** is a type of network where nodes are not dependent on a single master node; instead, control is distributed among many nodes. This is analogous to a model where each department in an organization is in charge of its own database server, thus taking away the power from the central server and distributing it to the sub-departments, who manage their own databases.

A significant innovation in the decentralized paradigm that has given rise to this new era of decentralization of applications is **decentralized consensus**. This mechanism came into play with Bitcoin, and it enables a user to agree on something via a consensus algorithm without the need for a central, trusted third party, intermediary, or service provider.

We can also now view the different types of networks shown earlier from a different perspective, where we highlight the controlling authority of these networks as a symbolic hand, as shown in the following diagram. This model provides a clearer understanding of the differences between these networks from a decentralization point of view:

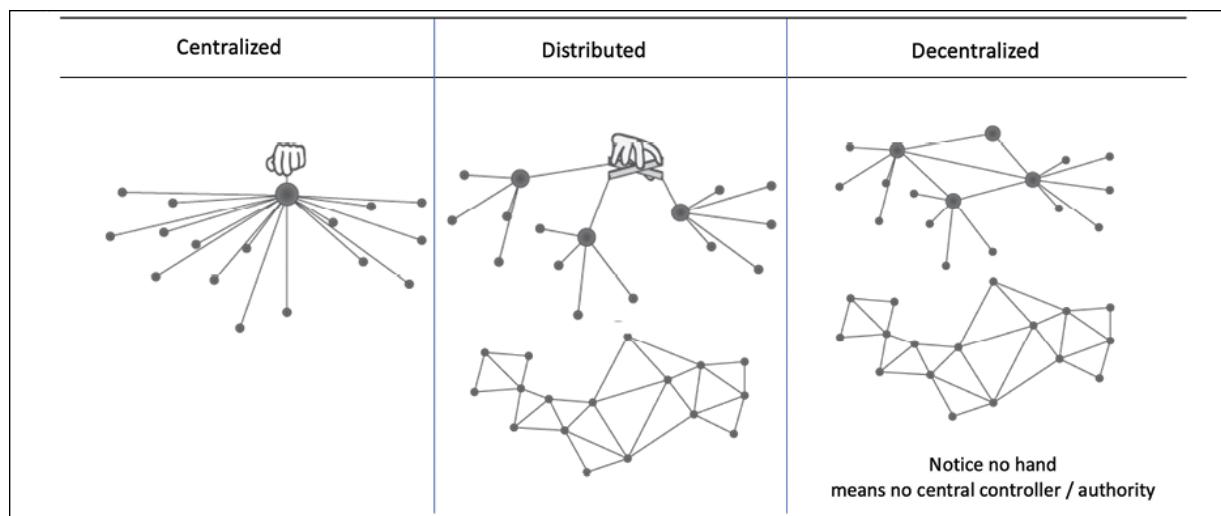


Figure 2.2: Different types of networks/systems depicting decentralization from a modern perspective

The preceding diagram shows that the centralized model is the traditional one in which a central controller exists, and it can be viewed as a depiction of the usual client/server model. In the middle we have distributed systems, where we still have a central controller but the system comprises many dispersed nodes. On the right-hand side, notice that there is no hand/controller controlling the networks.

This is the key difference between decentralized and distributed networks. A decentralized system may look like a distributed system from a topological point of view, but it doesn't have a central authority that controls the network.

The differences between distributed and decentralized systems can also be viewed at a practical level in the following diagrams:

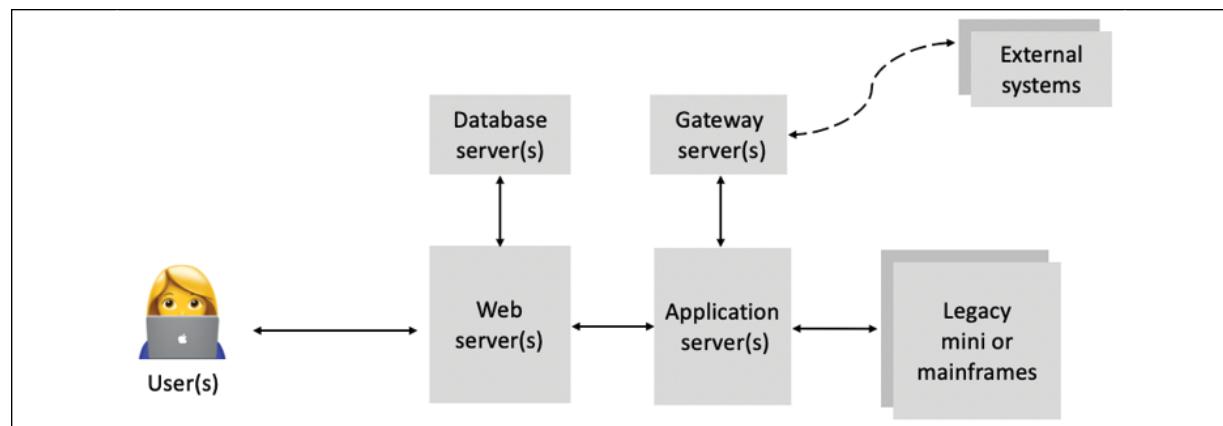


Figure 2.3: A traditional distributed system comprises many servers performing different roles

The following diagram shows a decentralized system (based on blockchain) where an exact replica of the applications and data is maintained across the entire network on each participating node:

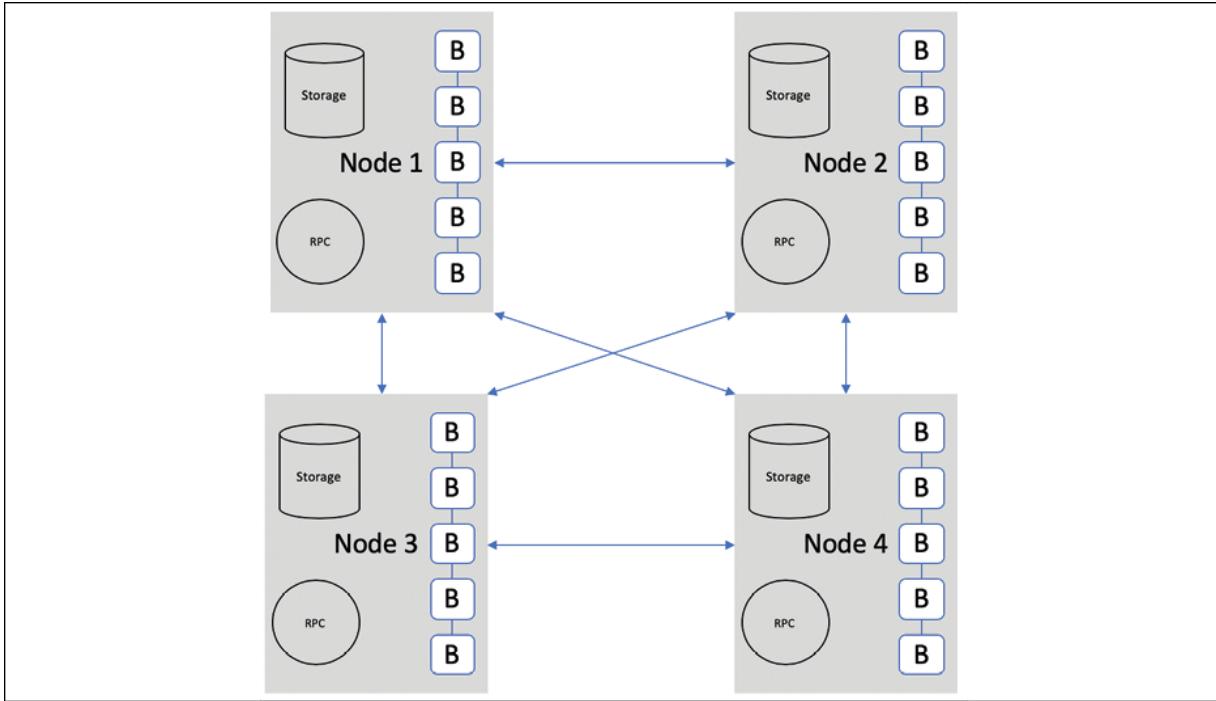


Figure 2.4: A blockchain-based decentralized system (notice the direct P2P connections and the exact replicas of blocks)

A comparison between centralized and decentralized systems (networks/applications) is shown in the following table:

Feature	Centralized	Decentralized
Ownership	Service provider	All users
Architecture	Client/server	Distributed, different topologies
Security	Basic	More secure
High availability	No	Yes
Fault tolerance	Basic, single point of failure	Highly tolerant, as service is replicated

Collusion resistance	Basic, because it's under the control of a group or even single individual	Highly resistant, as consensus algorithms ensure defense against adversaries
Application architecture	Single application	Application replicated across all nodes on the network
Trust	Consumers have to trust the service provider	No mutual trust required
Cost for consumer	Higher	Lower

The comparison in the table only covers some main features and is not an exhaustive list of all features. There may be other features of interest that can be compared too, but this list should provide a good level of comparison.

Now we will discuss what methods can be used to achieve decentralization.

Methods of decentralization

Two methods can be used to achieve decentralization: disintermediation and competition. These methods will be discussed in detail in the sections that follow.

Disintermediation

The concept of **disintermediation** can be explained with the aid of an example. Imagine that you want to send money to a friend in another country. You go to a bank, which, for a fee, will transfer your money to the bank in that country. In this case, the bank maintains a central database that is updated, confirming that you have sent the money. With blockchain technology, it is possible to send this money directly to your friend without

the need for a bank. All you need is the address of your friend on the blockchain. This way, the intermediary (that is, the bank) is no longer required, and decentralization is achieved by disintermediation. It is debatable, however, how practical decentralization through disintermediation is in the financial sector due to the massive regulatory and compliance requirements. Nevertheless, this model can be used not only in finance but in many other industries as well, such as health, law, and the public sector. In the health industry, where patients, instead of relying on a trusted third party (such as the hospital record system) can be in full control of their own identity and their data that they can share directly with only those entities that they trust. As a general solution, blockchain can serve as a decentralized health record management system where health records can be exchanged securely and directly between different entities (hospitals, pharmaceutical companies, patients) globally without any central authority.

Contest-driven decentralization

In the method involving **competition**, different service providers compete with each other in order to be selected for the provision of services by the system. This paradigm does not achieve complete decentralization. However, to a certain degree, it ensures that an intermediary or service provider is not monopolizing the service. In the context of blockchain technology, a system can be envisioned in which smart contracts can choose an external data provider from a large number of providers based on their reputation, previous score, reviews, and quality of service.

This method will not result in full decentralization, but it allows smart contracts to make a free choice based on the criteria just mentioned. This way, an environment of competition is cultivated among service providers where they compete with each other to become the data provider of choice.

In the following diagram, varying levels of decentralization are shown. On the left side, the conventional approach is shown where a central system is in control; on the right side, complete disintermediation is achieved, as intermediaries are entirely removed. Competing intermediaries or service providers are shown in the center. At that level, intermediaries or service

providers are selected based on reputation or voting, thus achieving partial decentralization:

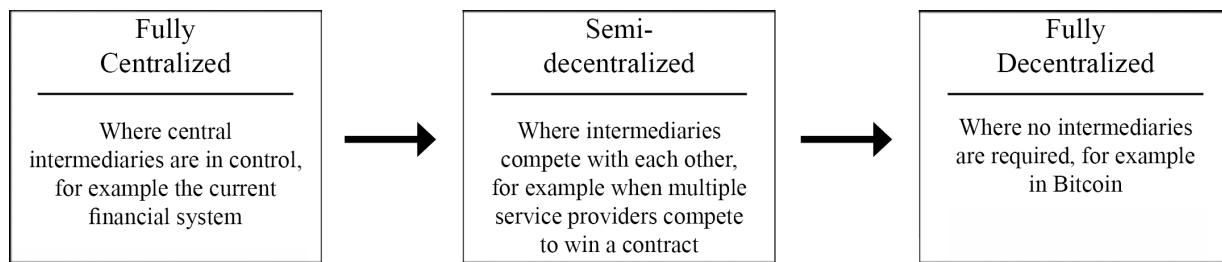


Figure 2.5: Scale of decentralization

There are many benefits of decentralization, including transparency, efficiency, cost saving, development of trusted ecosystems, and in some cases privacy and anonymity. Some challenges, such as security requirements, software bugs, and human error, need to be examined thoroughly.

For example, in a decentralized system such as Bitcoin or Ethereum where security is normally provided by private keys, how can we ensure that an asset or a token associated with these private keys cannot be rendered useless due to negligence or bugs in the code? What if the private keys are lost due to user negligence? What if due to a bug in the smart contract code the decentralized application becomes vulnerable to attack?



Before embarking on a journey to decentralize everything using blockchain and decentralized applications, it is essential that we understand that not everything can or needs to be decentralized.

This view raises some fundamental questions. Is a blockchain really needed? When is a blockchain required? In what circumstances is blockchain preferable to traditional databases? To answer these questions, go through the simple set of questions presented below:

Question	Yes/No	Recommended solution

Is high data throughput required?	Yes	Use a traditional database.
	No	A central database might still be useful if other requirements are met. For example, if users trust each other, then perhaps there is no need for a blockchain. However, if they don't or trust cannot be established for any reason, blockchain can be helpful.
Are updates centrally controlled?	Yes	Use a traditional database.
	No	You may investigate how a public/private blockchain can help.
Do users trust each other?	Yes	Use a traditional database.
	No	Use a public blockchain.
Are users anonymous?	Yes	Use a public blockchain.
	No	Use a private blockchain.
Is consensus required to be maintained within a consortium?	Yes	Use a private blockchain.
	No	Use a public blockchain.
Is strict data immutability required?	Yes	Use a blockchain.

No	Use a central/traditional database.
----	-------------------------------------

Answering all of these questions can help you decide whether or not a blockchain is required or suitable for solving the problem. Beyond the questions posed in this model, there are many other issues to consider, such as latency, choice of consensus mechanisms, whether consensus is required or not, and where consensus is going to be achieved. If consensus is maintained internally by a consortium, then a private blockchain should be used; otherwise, if consensus is required publicly among multiple entities, then a public blockchain solution should be considered. Other aspects, such as immutability, should also be considered when deciding whether to use a blockchain or a traditional database. If strict data immutability is required, then a public blockchain should be used; otherwise, a central database may be an option.

As blockchain technology matures, there will be more questions raised regarding this selection model. For now, however, this set of questions is sufficient for deciding whether a blockchain-based solution is suitable or not.

Now we understand different methods of decentralization and have looked at how to decide whether a blockchain is required or not in a particular scenario. Let's now look at the process of decentralization, that is, how we can take an existing system and decentralize it. First, we'll briefly look at the different ways to achieve decentralization.

Routes to decentralization

There are systems that pre-date blockchain and Bitcoin, including BitTorrent and the Gnutella file-sharing system, which to a certain degree could be classified as decentralized, but due to a lack of any incentivization mechanism, participation from the community gradually decreased. There wasn't any incentive to keep the users interested in participating in the growth of the network. With the advent of blockchain technology, many

initiatives are being taken to leverage this new technology to achieve decentralization. The Bitcoin blockchain is typically the first choice for many, as it has proven to be the most resilient and secure blockchain and has a market cap of nearly \$166 billion at the time of writing. Alternatively, other blockchains, such as Ethereum, serve as the tool of choice for many developers for building decentralized applications. Compared to Bitcoin, Ethereum has become a more prominent choice because of the flexibility it allows for programming any business logic into the blockchain by using **smart contracts**.

How to decentralize

Arvind Narayanan and others have proposed a framework in their book *Bitcoin and Cryptocurrency Technologies* that can be used to evaluate the decentralization requirements of a variety of issues in the context of blockchain technology. The framework raises four questions whose answers provide a clear understanding of how a system can be decentralized:

1. What is being decentralized?
2. What level of decentralization is required?
3. What blockchain is used?
4. What security mechanism is used?

The first question simply asks you to identify what system is being decentralized. This can be any system, such as an identity system or a trading system.

The second question asks you to specify the level of decentralization required by examining the scale of decentralization, as discussed earlier. It can be full disintermediation or partial disintermediation.

The third question asks developers to determine which blockchain is suitable for a particular application. It can be Bitcoin blockchain, Ethereum blockchain, or any other blockchain that is deemed fit for the specific application.

Finally, a fundamental question that needs to be addressed is how the security of a decentralized system will be guaranteed. For example, the security mechanism can be atomicity-based, where either the transaction executes in full or does not execute at all. This deterministic approach ensures the integrity of the system. Other mechanisms may include one based on reputation, which allows for varying degrees of trust in a system.

In the following section, let's evaluate a money transfer system as an example of an application selected to be decentralized.

Decentralization framework example

The four questions discussed previously are used to evaluate the decentralization requirements of this application. The answers to these questions are as follows:

1. Money transfer system
2. Disintermediation
3. Bitcoin
4. Atomicity

The responses indicate that the money transfer system can be decentralized by removing the intermediary, implemented on the Bitcoin blockchain, and that a security guarantee will be provided via atomicity. Atomicity will ensure that transactions execute successfully in full or do not execute at all. We have chosen the Bitcoin blockchain because it is the longest established blockchain and has stood the test of time.

Similarly, this framework can be used for any other system that needs to be evaluated in terms of decentralization. The answers to these four simple questions help clarify what approach to take to decentralize the system.

To achieve complete decentralization, it is necessary that the environment around the blockchain also be decentralized. We'll look at the full ecosystem of decentralization next.

Blockchain and full ecosystem decentralization

The blockchain is a distributed ledger that runs on top of conventional systems. These elements include storage, communication, and computation.



There are other factors, such as identity and wealth, which are traditionally based on centralized paradigms, and there's a need to decentralize these aspects as well in order to achieve a sufficiently decentralized ecosystem.

Storage

Data can be stored directly in a blockchain, and with this fact it achieves decentralization. However, a significant disadvantage of this approach is that a blockchain is not suitable for storing large amounts of data by design. It can store simple transactions and some arbitrary data, but it is certainly not suitable for storing images or large blobs of data, as is the case with traditional database systems.

A better alternative for storing data is to use **distributed hash tables (DHTs)**. DHTs were used initially in peer-to-peer file sharing software, such as BitTorrent, Napster, Kazaa, and Gnutella. DHT research was made popular by the CAN, Chord, Pastry, and Tapestry projects. BitTorrent is the most scalable and fastest network, but the issue with BitTorrent and the others is that there is no incentive for users to keep the files indefinitely. Users generally don't keep files permanently, and if nodes that have data still required by someone leave the network, there is no way to retrieve it except by having the required nodes rejoin the network so that the files once again become available.

Two primary requirements here are high availability and link stability, which means that data should be available when required and network links also should always be accessible. **Inter-Planetary File System (IPFS)** by

Juan Benet possesses both of these properties, and its vision is to provide a decentralized World Wide Web by replacing the HTTP protocol. IPFS uses Kademlia DHT and Merkle **Directed Acyclic Graphs (DAGs)** to provide storage and searching functionality, respectively. The concept of DHTs and DAGs will be introduced in detail in *Chapter 4, Public Key Cryptography*.

The incentive mechanism for storing data is based on a protocol known as Filecoin, which pays incentives to nodes that store data using the Bitswap mechanism. The Bitswap mechanism lets nodes keep a simple ledger of bytes sent or bytes received in a one-to-one relationship. Also, a Git-based version control mechanism is used in IPFS to provide structure and control over the versioning of data.

There are other alternatives for data storage, such as Ethereum Swarm, Storj, and MaidSafe. Ethereum has its own decentralized and distributed ecosystem that uses Swarm for storage and the Whisper protocol for communication. MaidSafe aims to provide a decentralized World Wide Web. All of these projects are discussed later in this book in greater detail.

BigChainDB is another storage layer decentralization project aimed at providing a scalable, fast, and linearly scalable decentralized database as opposed to a traditional filesystem. BigChainDB complements decentralized processing platforms and filesystems such as Ethereum and IPFS.

Communication

The Internet (the communication layer in blockchain) is considered to be decentralized. This belief is correct to some extent, as the original vision of the Internet was to develop a decentralized communications system. Services such as email and online storage are now all based on a paradigm where the service provider is in control, and users trust such providers to grant them access to the service as requested. This model is based on the unconditional trust of a central authority (the service provider) where users are not in control of their data. Even user passwords are stored on trusted third-party systems.

Thus, there is a need to provide control to individual users in such a way that access to their data is guaranteed and is not dependent on a single third party. Access to the Internet (the communication layer) is based on **Internet Service Providers (ISPs)** who act as a central hub for Internet users. If the ISP is shut down for any reason, then no communication is possible with this model.

An alternative is to use **mesh networks**. Even though they are limited in functionality when compared to the Internet, they still provide a decentralized alternative where nodes can talk directly to each other without a central hub such as an ISP.



An example of a mesh network is Firechat, which allows iPhone users to communicate with each other directly in a peer-to-peer fashion without an Internet connection. More information is available at <https://www.opengarden.com/firechat/>.

Now imagine a network that allows users to be in control of their communication; no one can shut it down for any reason. This could be the next step toward decentralizing communication networks in the blockchain ecosystem. It must be noted that this model may only be vital in a jurisdiction where the Internet is censored and controlled by the government.

As mentioned earlier, the original vision of the Internet was to build a decentralized network; however, over the years, with the advent of large-scale service providers such as Google, Amazon, and eBay, control is shifting toward these big players. For example, email is a decentralized system at its core; that is, anyone can run an email server with minimal effort and can start sending and receiving emails. There are better alternatives available. For example, Gmail and Outlook already provide managed services for end users, so there is a natural inclination toward selecting one of these large centralized services as they are more convenient and free to use. This is one example that shows how the Internet has moved toward centralization.

Free services, however, are offered at the cost of exposing valuable personal data, and many users are unaware of this fact. Blockchain has revived the vision of decentralization across the world, and now concerted efforts are being made to harness this technology and take advantage of the benefits that it can provide.

Computing power and decentralization

Decentralization of computing or processing power is achieved by a blockchain technology such as Ethereum, where smart contracts with embedded business logic can run on the blockchain network. Other blockchain technologies also provide similar processing-layer platforms, where business logic can run over the network in a decentralized manner.

The following diagram shows an overview of a decentralized ecosystem. In the bottom layer, the Internet or mesh networks provide a decentralized communication layer. In the next layer up, a storage layer uses technologies such as IPFS and BigChainDB to enable decentralization. Finally, in the next level up, you can see that the blockchain serves as a decentralized processing (computation) layer. Blockchain can, in a limited way, provide a storage layer too, but that severely hampers the speed and capacity of the system. Therefore, other solutions such as IPFS and BigChainDB are more suitable for storing large amounts of data in a decentralized way. The Identity and Wealth layers are shown at the top level. Identity on the Internet is a vast topic, and systems such as bitAuth and OpenID provide authentication and identification services with varying degrees of decentralization and security assumptions:

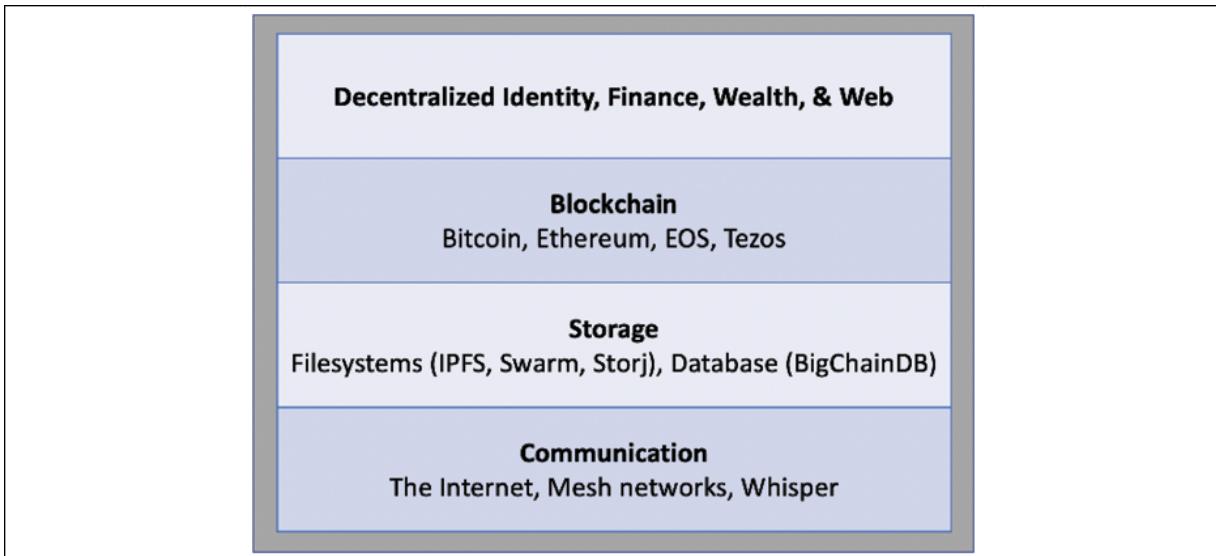


Figure 2.6: Decentralized ecosystem

The blockchain is capable of providing solutions to various issues relating to decentralization. A concept relevant to identity known as **Zooko's Triangle** requires that the naming system in a network protocol is secure, decentralized, and able to provide human-meaningful and memorable names to the users. Conjecture has it that a system can have only two of these properties simultaneously.

Nevertheless, with the advent of blockchain in the form of **Namecoin**, this problem was resolved. It is now possible to achieve security, decentralization, and human-meaningful names with the Namecoin blockchain. However, this is not a panacea, and it comes with many challenges, such as reliance on users to store and maintain private keys securely. This opens up other general questions about the suitability of decentralization to a particular problem.

Decentralization may not be appropriate for every scenario. Centralized systems with well-established reputations tend to work better in many cases. For example, email platforms from reputable companies such as Google or Microsoft would provide a better service than a scenario where individual email servers are hosted by users on the Internet.

There are many projects underway that are developing solutions for a more comprehensive distributed blockchain system. For example, Swarm and

Whisper are developed to provide decentralized storage and communication for Ethereum. We will discuss Swarm and Ethereum in more detail in *Chapter 13, Ethereum Development Environment*.

With the advent of blockchain technology, it is now possible to build software versions of traditional physical organizations in the form of **Decentralized Organizations (DOs)** and other similar constructs, which we will examine in detail shortly.

Moreover, with the emergence of the decentralization paradigm, different terminology and buzzwords are now appearing in the media and academic literature, which we will explore in the next section.

Pertinent terminology

The following concepts are worth citing in the context of decentralization. The terminology introduced here is often used in the literature concerning decentralization and its applications.

Smart contracts

A **smart contract** is a software program that usually runs on a blockchain. Smart contracts do not necessarily need a blockchain to run; however, due to the security benefits that blockchain technology provides, blockchain has become a standard decentralized execution platform for smart contracts.

A smart contract usually contains some business logic and a limited amount of data. The business logic is executed if specific criteria are met. Actors or participants in the blockchain use these smart contracts, or they run autonomously on behalf of the network participants.

More information on smart contracts will be provided in *Chapter 10, Smart Contracts*.

Autonomous agents

An **Autonomous Agent (AA)** is an artificially intelligent software entity that acts on the behalf of its owner to achieve some desirable goals without requiring any or minimal intervention from its owner.

Decentralized organizations

DOs are software programs that run on a blockchain and are based on the idea of actual organizations with people and protocols. Once a DO is added to the blockchain in the form of a smart contract or a set of smart contracts, it becomes decentralized and parties interact with each other based on the code defined within the DO software.

Decentralized autonomous organizations

Just like DOs, a **decentralized autonomous organization (DAO)** is also a computer program that runs on top of a blockchain, and embedded within it are governance and business logic rules. DAOs and DOs are fundamentally the same thing. The main difference, however, is that DAOs are autonomous, which means that they are fully automated and contain artificially intelligent logic. DOs, on the other hand, lack this feature and rely on human input to execute business logic.

Ethereum blockchain led the way with the introduction of DAOs. In a DAO, the code is considered the governing entity rather than people or paper contracts. However, a human curator maintains this code and acts as a proposal evaluator for the community. DAOs are capable of hiring external contractors if enough input is received from the token holders (participants).

The most famous DAO project is **The DAO**, which raised \$168 million in its crowdfunding phase. The DAO project was designed to be a venture capital fund aimed at providing a decentralized business model with no single entity as owner. Unfortunately, this project was hacked due to a bug in the DAO code, and millions of dollars' worth of **ether** currency (**ETH**)

was siphoned out of the project and into a child DAO created by hackers. A major network change (hard fork) was required on the Ethereum blockchain to reverse the impact of the hack and initiate the recovery of the funds. This incident opened up the debate on the security, quality, and need for thorough testing of the code in smart contracts in order to ensure their integrity and adequate control. There are other projects underway, especially in academia, that are seeking to formalize smart contract coding and testing.

Currently, DAOs do not have any legal status, even though they may contain some intelligent code that enforces certain protocols and conditions. However, these rules have no value in the real-world legal system at present. One day, perhaps an AA (that is, a piece of code that runs without human intervention) commissioned by a law enforcement agency or regulator will contain rules and regulations that could be embedded in a DAO for the purpose of ensuring its integrity from a legalistic and compliance perspective. The fact that DAOs are purely decentralized entities enables them to run in any jurisdiction. Thus, they raise a big question as to how the current legal system could be applied to such a varied mix of jurisdictions and geographies.

Decentralized autonomous corporations

Decentralized autonomous corporations (DAsCs) are similar to DAOs in concept, though considered to be a subset of them. The definitions of DACs and DAOs may sometimes overlap, but the general distinction is that DAOs are usually considered to be nonprofit, whereas DACs can earn a profit via shares offered to the participants and to whom they can pay dividends. DACs can run a business automatically without human intervention based on the logic programmed into them.

Decentralized autonomous societies

Decentralized autonomous societies (DASes) are a concept whereby an entire society can function on a blockchain with the help of multiple,

complex smart contracts and a combination of DAOs and **decentralized applications (DApps)** running autonomously. This model does not necessarily translate to a free-for-all approach, nor is it based on an entirely libertarian ideology; instead, many services that a government commonly offers can be delivered via blockchains, such as government identity card systems, passports, and records of deeds, marriages, and births. Another theory is that, if a government is corrupt and central systems do not provide the levels of trust that a society needs, then that society can start its own virtual one on a blockchain that is driven by decentralized consensus and transparency. This concept might look like a libertarian's or cypherpunk's dream, but it is entirely possible on a blockchain.

Decentralized applications

All the ideas mentioned up to this point come under the broader umbrella of decentralized applications, abbreviated to DApps. DAOs, DACs, and DOs are DApps that run on top of a blockchain in a peer-to-peer network. They represent the latest advancement in decentralization technology.

DApps at a fundamental level are software programs that execute using either of the following methods. They are categorized as Type 1, Type 2, or Type 3 DApps:

1. **Type 1:** Run on their own dedicated blockchain, for example, standard smart contract based DApps running on Ethereum. If required, they make use of a native token, for example, ETH on Ethereum blockchain.



For example, Ethlance is a DApp that makes use of ETH to provide a job market. More information about Ethlance can be found at <https://ethlance.com>.

2. **Type 2:** Use an existing established blockchain. that is, make use of Type 1 blockchain and bear custom protocols and tokens, for example, smart contract based tokenization DApps running Ethereum blockchain. An example is DAI, which is built on top of Ethereum

blockchain, but contains its own stable coins and mechanism of distribution and control. Another example is Golem, which has its own token GNT and a transaction framework built on top of Ethereum blockchain to provide a **decentralized marketplace** for computing power where users share their computing power with each other in a peer-to-peer network.



A prime example of Type 2 DApps is the OMNI network, which is a software layer built on top of Bitcoin to support trading of custom digital assets and digital currencies. More information on the OMNI network can be found at <https://www.omnilayer.org>. More information on the Golem network is available at <https://golem.network>. More information on DAI is available at <https://makerdao.com/en/>.

3. Type 3: Use the protocols of Type 2 DApps; for example, the SAFE Network uses the OMNI network protocol.



More information on the SAFE Network can be found at <https://safenetwork.tech>.

Another example to understand the difference between different types of DApps is the USDT token (Tethers). The original USDT uses the OMNI layer (a Type 2 DApp) on top of the Bitcoin network. USDT is also available on Ethereum using ERC20 tokens. This example shows that a USDT can be considered a Type 3 DApp, where the OMNI layer protocol (a Type 2 DApp) is used, which is itself built on Bitcoin (a Type 1 DApp). Also, from an Ethereum point of view USDT can also be considered a Type 3 DApp in that it makes use of the Type 1 DApp Ethereum blockchain using the ERC 20 standard, which was built to operate on Ethereum.

More information can be found about Tether at <https://tether.to>.

In the last few years, the expression DApp has been increasingly used to refer to any end-to-end decentralized blockchain application, including a



user interface (usually a web interface), smart contract(s), and the host blockchain. The clear distinction between different types of DApps is now not commonly referred to, but it does exist. Often, DApps are now considered just as apps (blockchain apps) running on a blockchain such as Ethereum, Tezos, NEO, or EOS without any particular reference to their type.

There are thousands of different DApps running on various platforms (blockchains) now. There are various categories of these DApps covering media, social, finance, games, insurance, and health. There are various decentralized platforms (or blockchains) running, such as Ethereum, EOS, NEO, Loom, and Steem. The highest number of DApps currently is on Ethereum.

Requirements of a DApp

For an application to be considered decentralized, it must meet the following criteria. This definition was provided in a whitepaper by Johnston et al. in 2015, *The General Theory of Decentralized Applications, DApps*:

1. The DApp should be fully open source and autonomous, and no single entity should be in control of a majority of its tokens. All changes to the application must be consensus-driven based on the feedback given by the community.
2. Data and records of operations of the application must be cryptographically secured and stored on a public, decentralized blockchain to avoid any central points of failure.
3. A cryptographic token must be used by the application to provide access for and incentivize those who contribute value to the applications, for example, miners in Bitcoin.
4. The tokens (if applicable) must be generated by the decentralized application using consensus and an applicable cryptographic algorithm. This generation of tokens acts as a proof of the value to contributors (for example, miners).

Generally, DApps now provide all sorts of different services, including but not limited to financial applications, gaming, social media, and health.

Operations of a DApp

Establishment of consensus by a DApp can be achieved using consensus algorithms such as PoW and **Proof of Stake (PoS)**. So far, only PoW has been found to be incredibly resistant to attacks, as is evident from the success of and trust people have put in the Bitcoin network. Furthermore, a DApp can distribute tokens (coins) via **mining**, **fundraising**, and **development**.

Design of a DApp

A DApp—pronounced Dee-App, or now more commonly rhyming with app—is a software application that runs on a decentralized network such as a distributed ledger. They have recently become very popular due to the development of various decentralized platforms such as Ethereum, EOS, and Tezos.

Traditional apps commonly consist of a user interface and usually a web server or an application server and a backend database. This is a common client/server architecture. This is visualized in the following diagram:

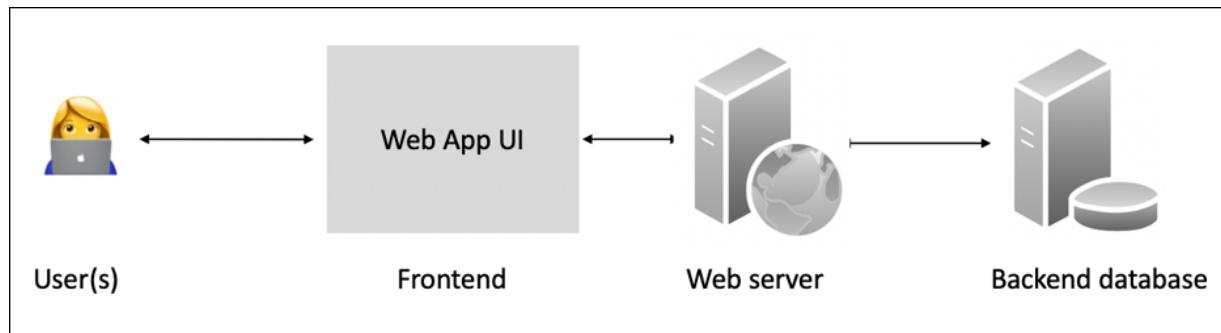


Figure 2.7: Traditional application architecture (generic client/server)

A DApp on the other hand has a blockchain as a backend and can be visualized as depicted in the following diagram. The key element that plays a vital role in the creation of a DApp is a smart contract that runs on the blockchain and has business logic embedded within it:

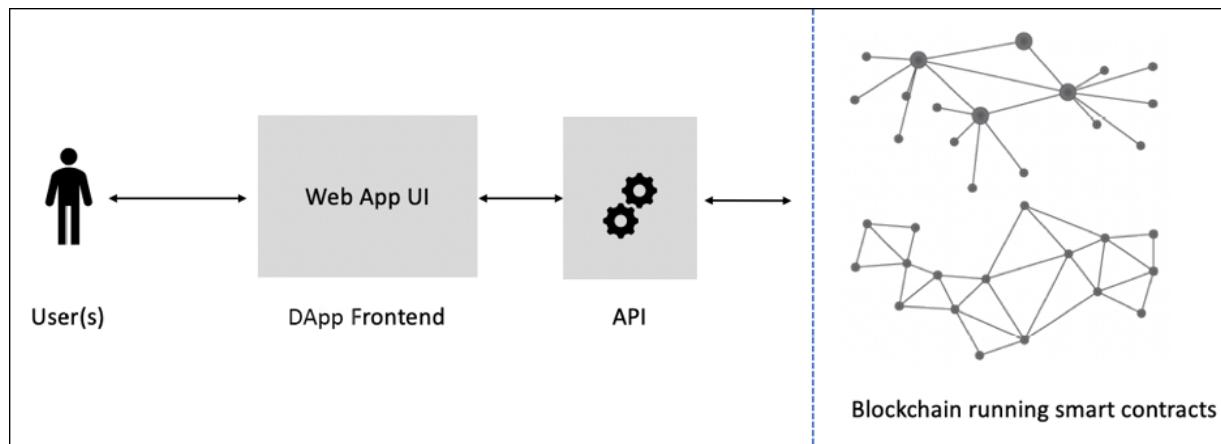


Figure 2.8: Generic DApp architecture

Note that the frontend in either a DApp or app architecture can either be a thick client, a mobile app, or a web frontend (a web user interface). However, it is usually a web frontend commonly written using a JavaScript framework such as React or Angular.

The following comparison table highlights the key properties of and differences between these different types of decentralized entities:

Entity	Autonomous?	Software?	Owned?	Capital?	Legal status	Cost
DO	No	No	Yes	Yes	Yes	High
DAO	Yes	Yes	No	Yes	Unsettled	Low
DAC	Yes	Yes	Yes	Yes	Unsettled	Low
DAS	Yes	Yes	No	Possible	Unsettled	Low
DApp	Yes	Yes	Yes	Optional tokens	Unsettled	Use case dependent

Having covered the main concepts of DApps, it will be useful to explore some specific examples.

DApp examples

Examples of some DApps are provided here.

KYC-Chain

This application provides the facility to manage **Know Your Customer (KYC)** data securely and conveniently based on smart contracts.

OpenBazaar

This is a decentralized peer-to-peer network that enables commercial activities directly between sellers and buyers instead of relying on a central party, such as eBay or Amazon. It should be noted that this system is not built on top of a blockchain; instead, distributed hash tables are used in a peer-to-peer network to enable direct communication and data sharing among peers. It makes use of Bitcoin and various other cryptocurrencies as a payment method.



More information regarding Open Bazaar is available at
<https://openbazaar.org>.

Lazooz

This is the decentralized equivalent of Uber. It allows peer-to-peer ride sharing and users to be incentivized by proof of movement, and they can earn Zooz coins.



More information on Lazooz is available at <http://lazooz.org>.

Many other DApps have been built on the Ethereum blockchain and are showcased at <http://dapps.ethercasts.com/>.

Now that we have covered the pertinent terminology, DApps, and relevant examples, let's now look at what platforms can be used to build and host DApps.

Platforms for decentralization

Today, there are many platforms available for decentralization. In fact, the fundamental feature of blockchain networks is to provide decentralization. Therefore, any blockchain network, such as Bitcoin, Ethereum, Hyperledger Fabric, or Quorum, can be used to provide a decentralization service. Many organizations around the world have introduced platforms that promise to make distributed application development easy, accessible, and secure. Some of these platforms are described as follows.

Ethereum

Ethereum tops the list as being the first blockchain to introduce a Turing-complete language and the concept of a virtual machine. This is in stark contrast to the limited scripting language in Bitcoin and many other cryptocurrencies. With the availability of its Turing-complete language, Solidity, endless possibilities have opened for the development of decentralized applications. This blockchain was first proposed in 2013 by Vitalik Buterin, and it provides a public blockchain to develop smart contracts and decentralized applications. Currency tokens on Ethereum are called **ethers**.

MaidSafe

This is a project for the decentralized Internet introduced in 2006. This is not a blockchain, but a decentralized and autonomous network.

MaidSafe provides a **SAFE (Secure Access for Everyone)** network that is made up of unused computing resources, such as storage, processing power, and the data connections of its users. The files on the network are divided into small chunks of data, which are encrypted and distributed randomly throughout the network. This data can only be retrieved by its respective owner. One key innovation of MaidSafe is that duplicate files are automatically rejected on the network, which helps reduce the need for additional computing resources needed to manage the load. It uses *Safecoin* as a token to incentivize its contributors.



More information on MaidSafe is available at
<https://maidsafe.net>.

Lisk

Lisk is a blockchain application development and cryptocurrency platform. It allows developers to use JavaScript to build decentralized applications and host them in their respective sidechains. Lisk uses the **Delegated Proof of Stake (DPOS)** mechanism for consensus, whereby 101 nodes can be elected to secure the network and propose blocks. It uses the Node.js and JavaScript backend, while the frontend allows the use of standard technologies, such as CSS3, HTML5, and JavaScript. Lisk uses **LSK** coin as a currency on the blockchain. Another derivative of Lisk is Rise, which is a Lisk-based DApp and digital currency platform. It offers greater focus on the security of the system.

EOS

This is a blockchain protocol launched in January 2018, with its own cryptocurrency called EOS. EOS raised an incredible 4 billion USD in 2018 through its **Initial Coin Offering (ICO)**. The key purpose behind EOS is, as stated by its founders, to build a decentralized operating system. Its throughput is significantly higher (approx. 3,996 **transactions per second**)

(TPS)) than other common blockchain platforms, such as Bitcoin (approx. 7 TPS) and Ethereum (approx. 15 TPS).

A practical introduction to these platforms and some others are provided in this book's bonus content pages, which can be found here:

https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

Due to fast-paced innovation and the natural evolution of these platforms and blockchain in general, many innovative trends have emerged, which we explore in the next section.

Innovative trends

With the growth of blockchain, several ideas have emerged that make use of the decentralized property of blockchain to provide more user centric and fully decentralized services. Some of the key ideas in this space are decentralized web, decentralized identity, and decentralized finance. We will explore these ideas one by one as follows.

Decentralized web

Decentralized web is a term that's used to describe a vision of the web where no central authority or set of authorities will be in control. The original intention of the Internet was indeed decentralized, and the development of open protocols such HTTP, SMTP, and DNS meant that any individual could use these protocols freely, and immediately become part of the Internet. This is still true; however, with the emergence of a layer above these protocols called the **Web layer** introduced a more service-oriented infrastructure, which inevitably led to large profit-seeking companies taking over. This is evident from the rise of Facebook, Google, Twitter, and

Amazon, which of course provide excellent user services but at the cost of a more controlled, centralized, and closed system.

Once intended and developed as decentralized, open and free protocols are now being dominated by powerful commercial entities around the world, which has resulted in major concerns around privacy and data protection. These types of business models do work well and are quite popular due to the high level of standardization and services provided, but they pose a threat to privacy and decentralization due to the dominance of only a handful of entities on the entire Internet.

With blockchain, it is envisioned that this situation will change as it will allow development of the decentralized Internet, or the decentralized web, or Web 3 for short, which was the original intention of the Internet.

We can review the evolution of the Web over the last few decades by dividing the major developments into three key stages, Web 1, Web 2, and Web 3.

Web 1

This is the original World Wide Web, which was developed in 1989. This was the era when static web pages were hosted on servers and usually only allowed read actions from a user's point of view.

Web 2

This is the era when more service-oriented and web-hosted applications started to emerge. E-commerce websites, social networking, social media, blogs, multimedia sharing, mash-ups, and web applications are the main features of this period. The current Web is Web 2, and even though we have a richer and more interactive Internet, all of these services are still centralized. Web 2 has generated massive economic value and provides services that are essential for day-to-day business, personal use, social interactions, and almost every walk of life, but privacy concerns, the need for trusted third parties, and data breaches are genuine issues that need to be addressed. Common examples of centralized Web 2 services include

Twitter, Facebook, Google Docs, and email services such Gmail and Hotmail.

Web 3

This is the vision of the decentralized internet or web that will revolutionize the way we use the internet today. This is the era that will be fully user-centric and decentralized without any single authority or a large organization or internet company in control. Some examples of Web 3 are as follows:

- **Steemit:** This is a social media platform based on the Steem blockchain and STEEM cryptocurrency. This cryptocurrency is awarded to contributors for the content they have shared, and the more votes they get, the more tokens they earn. More information is available at <https://steemit.com>.
- **Status:** This is a decentralized multipurpose communication platform providing secure and private communication. More information is available at <https://status.im>.
- **IPFS:** This is a peer-to-peer hypermedia/storage protocol that allows storage and sharing of data in a decentralized fashion across a peer-to-peer network. More information is available at <https://ipfs.io>.

Other fast-growing trends include decentralized identity and decentralized finance, which we introduce next.

Decentralized identity

Another trend that has gained massive popularity recently is decentralized identity. Identity is a sensitive and difficult problem to solve. Currently, due to the dominance of large Internet companies and other similar organizations, the identity of a user is not in control of the identity holder and this often leads to privacy issues. Decentralized identity gives control of identity credentials back to identity holders and enables them to control when and how they share their credentials and with whom.

A prime example of such an initiative is that Microsoft has built a decentralized identity network called **Identity Overlay Network (ION)** on top of Bitcoin blockchain. This infrastructure is based on work done for decentralized identity at W3C and the Decentralized Identity Foundation. Similar initiatives have also been taken by IBM and other organizations around the world.



More information regarding ION is available at
<https://github.com/decentralized-identity/ion>.

Information regarding the Decentralized Identity Foundation is available at
<https://identity.foundation>.

Information regarding **decentralized identifiers (DIDs)** is available at
<https://www.w3.org/TR/did-core/#dfn-decentralized-identifiers>.

Decentralized finance (DeFi)

DeFi could be the killer app of the decentralized revolution that everyone has been waiting for. Traditionally, finance is a business that is almost impossible to do without the involvement of a trusted third party. It can either be a bank or some other financial firm; consumers have to trust a central authority to do business on their behalf and provide the services.

There are rules, policies, procedures, and strict regulations that govern this ecosystem. This control and management is of paramount importance for the integrity of the entire financial ecosystem. Still, since there is always a central party that is required in every single transaction consumers do, this approach has some disadvantages.

Some main disadvantages are listed as follows:

- **Access barrier:** Access to financial services and banking requires a rigorous onboarding process involving KYC and other relevant checks and procedures. Even though this is extremely important for the integrity of the existing financial system, it can become a barrier at

times for millions of unbanked people all around the world, especially in third world countries.

- **High cost:** Financial services can be costly in some scenarios, especially investment-related activities, and could be seen as a barrier toward entering the financial services industry.
- **Transparency issues:** There are concerns around transparency and trust due to the proprietary nature of the financial industry.
- **Siloed:** Most current financial industry solutions are proprietary and are owned by their respective organization. This results in interoperability issues where systems from one organization are unable to talk with another.

DeFi comes with a number of advantages, such as inclusion, easy access, and cheaper services; however, it has its own challenges. These drawbacks must be addressed for further adoption of this novel paradigm of financial services:

- **Underdeveloped ecosystem:** This ecosystem is still developing and requires more effort to improve usability and adoption.
- **Too technical:** As some users may find understanding and handling all financial transactions and jargon a bit daunting, adopting a DeFi platform may take some time and educational effort.
- **Lack of regulation:** This is a genuine and clear concern because without the existence of any regulatory framework, this ecosystem can be used for illegal activities. Moreover, trusting the code alone, instead of traditional paper contracts with established financial institutions, is seen as a major problem, especially by consumers who are used to traditional financial systems.
- **Human error:** On a blockchain, human errors can result in serious implications. Especially on DeFi systems or cryptocurrency blockchains, any human negligence can result in serious implications, such as financial loss. Moreover, due to the lack of regulation (relative to traditional finance), such an issue can have an even greater detrimental impact.

Note that human error can occur in any system, and blockchain is not immune either. Generally, a blockchain is regarded as a tamper-proof, all-secure, and decentralized platform due to its security promises.

However, this perception can give the wrong impression. For example, for a general user who perhaps doesn't understand the underlying technology and associated limitations, human errors may go unnoticed, and users may innocently accept everything as accurate, due to *blockchain being the source of ultimate truth!*

Therefore, human errors must be minimized as much as possible. It is essential to perform validation checks in DApps, especially on user interfaces where users might be entering different types of data. For example, it should be ensured at the user interface level that account details (addresses) are correct (at least the format of the address) to reduce the possibility of loss of funds. Not much can be done afterward, for example, if funds are sent to the wrong address.

Other types of human errors might include, but are not limited to, sending funds to incompatible wallets, not appropriately protecting account information, and not understanding how to use DApps' frontends correctly.



DeFi applications are used in a broad spectrum of use cases, including loans, **decentralized exchanges (DEXs)**, derivatives, payments, insurance, and assets. All of these use cases have one thing in common, which is that there is no central authority in control and participants (that is, users) on DeFi platforms conduct business directly with each other without the involvement of any intermediaries. This may sound like a utopian fantasy, but the DeFi revolution is happening right now and there is almost 5 billion USD (at the time of writing) of value locked in the DeFi system. This large amount of locked value is a clear indication that the usage of DeFi applications is quite significant and is expected to only increase.



You can find the latest rankings and analytics of the DeFi protocol on this excellent website: <https://defipulse.com>.

Uses cases of all these developments and further details will be introduced later in *Chapter 19, Blockchain—Outside of Currencies*.

In this section, we discussed some innovative trends, including decentralized web and DeFi; however, these are not the only applications of blockchain technology. Some other emerging trends include the

convergence of different technologies with blockchain, which can open new vistas for further innovation such as artificial intelligence and the **Internet of Things (IoT)**. We will cover these topics in *Chapter 22, Current Landscape and What's Next*.

Summary

This chapter introduced the concept of decentralization, which is the core service offered by blockchain technology. Although the concept of decentralization is not new, it has gained renewed significance in the world of blockchain. As such, various applications based on a decentralized architecture have recently been introduced.

The chapter began with an introduction to the concept of decentralization. Next, decentralization from the blockchain perspective was discussed. Moreover, ideas relating to the different layers of decentralization in the blockchain ecosystem were introduced. Several new concepts and terms have emerged with the advent of blockchain technology and decentralization from the blockchain perspective, including DAOs, DACs, and DAPPs. Finally, some innovative trends and examples of DApps were presented.

In the next chapter, fundamental concepts necessary to understanding the blockchain ecosystem will be presented. Principally cryptography, which provides a crucial foundation for blockchain technology.

3

Symmetric Cryptography

In this chapter, you will be introduced to the concepts, theory, and practical aspects of **symmetric cryptography**. More focus will be given to the elements that are specifically relevant in the context of blockchain technology. This chapter will provide the concepts required to understand the material covered in later chapters.

You will also be introduced to applications of cryptographic algorithms so that you can gain hands-on experience in the practical implementation of cryptographic functions. For this, the OpenSSL command-line tool is used. Before starting with the theoretical foundations, the installation of OpenSSL is presented in the following section so that you can do some practical work as you read through the conceptual material.

Working with the OpenSSL command line

On the Ubuntu Linux distribution, OpenSSL is usually already available. However, it can be installed using the following commands:

```
$ sudo apt-get install openssl
```

Examples in this chapter have been developed using OpenSSL version 1.0.2t. It is available at

<https://packages.ubuntu.com/xenial/openssl>. You are

encouraged to use this specific version, as all examples in the chapter have been developed and tested with it. The OpenSSL version can be checked using the following command:

```
$ openssl version
```

If you see the following output,

```
OpenSSL 1.0.2t 10 Sep 2019
```

then you are all set to run the examples provided in this chapter. If you are running a version other than 1.0.2t, the examples may still work but that is not guaranteed, especially as older versions lack the features used in the examples and newer versions may not be backward compatible with version 1.0.2t.

In the sections that follow, the theoretical foundations of cryptography are first discussed and then a series of relevant practical experiments will be presented.

Introduction

Cryptography is the science of making information secure in the presence of adversaries. It does so under the assumption that limitless resources are available to adversaries. **Ciphers** are algorithms used to encrypt or decrypt data so that if intercepted by an adversary, the data is meaningless to them without **decryption**, which requires a secret key.

Cryptography is primarily used to provide a confidentiality service. On its own, it cannot be considered a complete solution, rather it serves as a crucial building block within a more extensive security system to address a security problem. For example, securing a blockchain ecosystem requires many different cryptographic primitives, such as hash functions, symmetric key cryptography, digital signatures, and public key cryptography.

In addition to a confidentiality service, cryptography also provides other security services such as integrity, authentication (entity authentication and data origin authentication), and non-repudiation. Additionally, accountability is also provided, which is a requirement in many security systems.

Cryptography

A generic cryptographic model is shown in the following diagram:

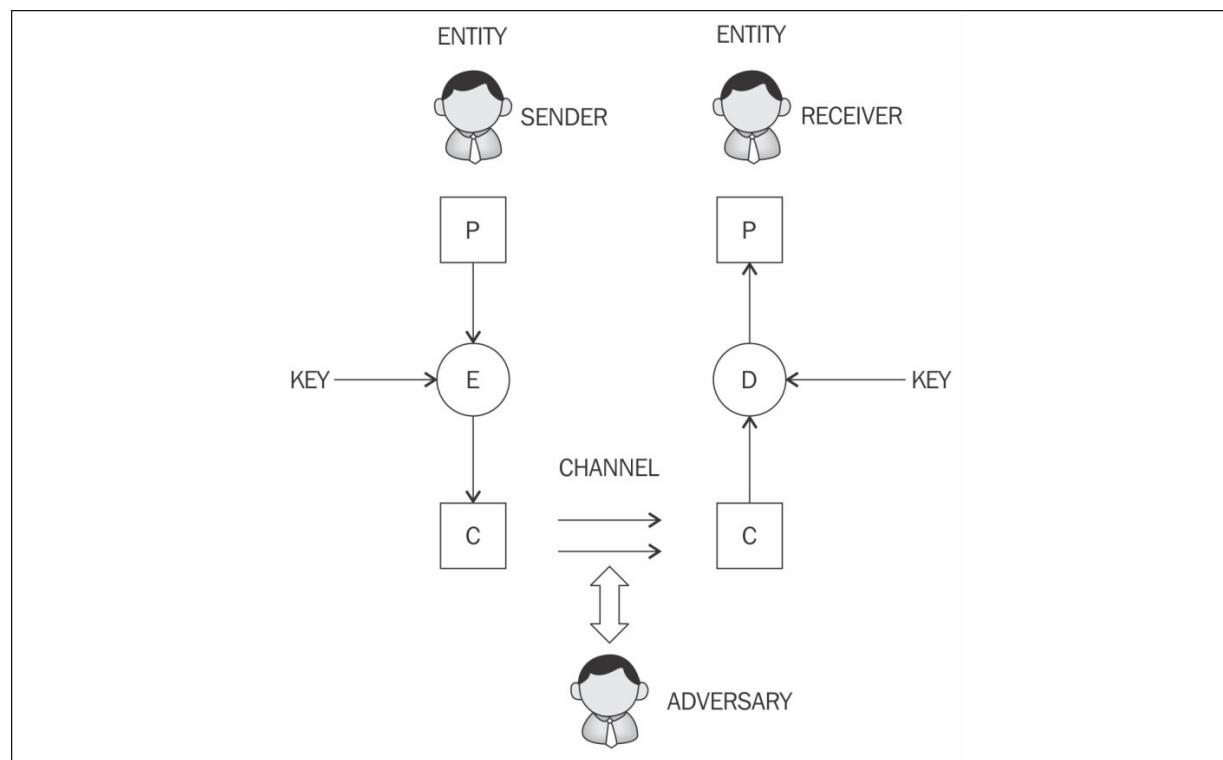


Figure 3.1: A model of the generic encryption and decryption model

In the preceding diagram, **P**, **E**, **C**, and **D** represent plaintext, encryption, ciphertext, and decryption, respectively. This model utilizes concepts such as entities, senders, receivers, adversaries, keys, and channels. These concepts are explained as follows:

- **Entity:** Either a person or system that sends, receives, or performs operations on data

- **Sender**: This is an entity that transmits the data
- **Receiver**: This is an entity that takes delivery of the data
- **Adversary**: This is an entity that tries to circumvent the security service
- **Key**: A key is data that is used to encrypt the plaintext and also to decrypt the ciphertext
- **Channel**: A channel provides a medium of communication between entities

We mentioned the fact that cryptography provides various services. In the following sections, we'll discuss these services in greater detail.

Confidentiality

Confidentiality is the assurance that information is only available to authorized entities.

Integrity

Integrity is the assurance that information is modifiable only by authorized entities.

Authentication

Authentication provides assurance about the identity of an entity or the validity of a message.

There are two types of authentication mechanisms, namely, entity authentication and data origin authentication, which are discussed in the following sections.

Entity authentication

Entity authentication is the assurance that an entity is currently involved and active in a communication session. Traditionally, users are issued a

username and password that is used to gain access to the various platforms with which they are working. This practice is known as **single-factor authentication**, as there is only one factor involved, namely, something you know; that is, the password and username.

This type of authentication is not very secure for a variety of reasons, for example, password leakage; therefore, additional factors are now commonly used to provide better security. The use of additional techniques for user identification is known as **multi-factor authentication** (or two-factor authentication if only two methods are used).

Various authentication methods are described here:

The first method uses *something you have*, such as a hardware token or a smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. This mechanism protects the user by requiring two factors of authentication. A user who has access to the hardware token and knows the login credentials will be able to access the system. Both factors should be available to gain access to the system, thus making this method a two-factor authentication mechanism. If the hardware token is inaccessible (for example, lost or stolen) or the login password is forgotten by the user, access to the system will be denied. The hardware token won't be of any use on its own unless the login password (*something you know*) is also known and used in conjunction with the hardware token.

The second method uses *something you are*, which uses biometric features to identify the user. With this method, a user's fingerprint, retina, iris, or hand geometry is used to provide an additional factor for authentication. This way, it can be ensured that the user was indeed present during the authentication process, as biometric features are unique to every individual. However, careful implementation is required to guarantee a high level of security, as some research has suggested that biometric systems can be circumvented under specific conditions.

Data origin authentication

Also known as **message authentication**, **data origin authentication** is an assurance that the source of the information is indeed verified. Data origin

authentication guarantees data integrity because, if a source is corroborated, then the data must not have been altered. Various methods, such as **Message Authentication Codes (MACs)** and digital signatures, are most commonly used. These terms will be explained in detail later in the chapter.

Non-repudiation

Non-repudiation is the assurance that an entity cannot deny a previous commitment or action by providing incontrovertible evidence. This is a security service that offers definitive proof that a particular activity has occurred. This property is essential in debatable situations whereby an entity has denied the actions performed, for example, the placement of an order on an e-commerce system. This service produces cryptographic evidence in electronic transactions so that in case of disputes, it can be used as confirmation of an action.

Non-repudiation has been an active research area for many years. Disputes in electronic transactions are a common issue, and there is a need to address them to increase consumers' confidence levels in such services.

The non-repudiation protocol usually runs in a communications network, and it is used to provide evidence that an action has been taken by an entity (originator or recipient) on the network. In this context, two communications models can be used to transfer messages from originator *A* to recipient *B*:

- A message is sent directly from originator *A* to recipient *B*; and
- A message is sent to a delivery agent from originator *A*, which then delivers the message to recipient *B*

The primary requirements of a non-repudiation protocol are fairness, effectiveness, and timeliness. In many scenarios, there are multiple participants involved in a transaction, as opposed to only two parties. For example, in electronic trading systems, there can be many entities, including clearing agents, brokers, and traders, who can be involved in a single transaction. In this case, two-party non-repudiation protocols are not

appropriate. To address this problem, **multi-party non-repudiation (MPNR)** protocols have been developed.

MPNR refers to non-repudiation protocols that run between multiple parties instead of the traditional two parties.



An excellent text on MPNR is *Onieva, J.A. and Zhou, J., 2008. Secure multi-party non-repudiation protocols and applications (Vol. 43). Springer Science & Business Media.*

Accountability

Accountability is the assurance that actions affecting security can be traced back to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are vital to trace an entity's actions, such as when a trade is placed in an audit record with the date and timestamp and the entity's identity is generated and saved in the log file. This log file can optionally be encrypted and be part of the database or a standalone ASCII text log file on a system.

In order to provide all of the services discussed in this section, different cryptographic primitives are used, which are presented in the next section.

Cryptographic primitives

Cryptographic primitives are the basic building blocks of a security protocol or system. In the following section, you are introduced to cryptographic algorithms that are essential for building secure protocols and systems. A **security protocol** is a set of steps taken to achieve the required security goals by utilizing appropriate security mechanisms. Various types of security protocols are in use, such as authentication protocols, non-repudiation protocols, and key management protocols.

The taxonomy of cryptographic primitives can be visualized as shown here:

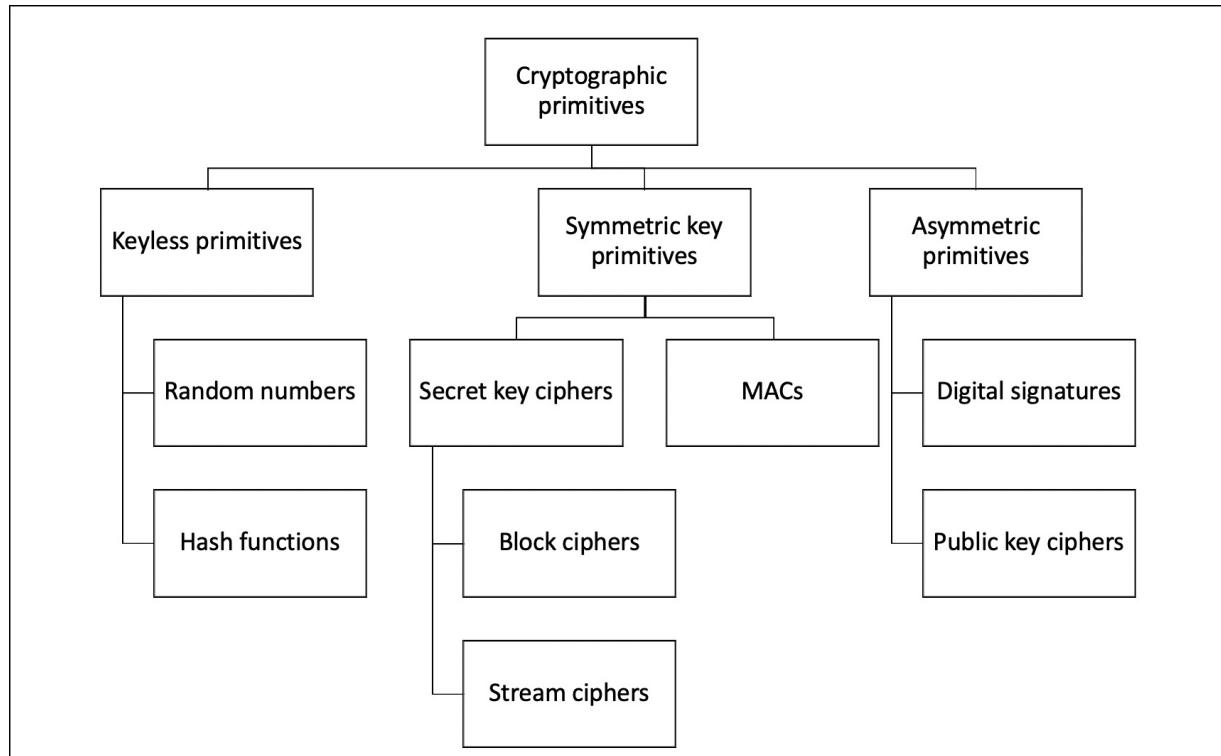


Figure 3.2: Cryptographic primitives

As shown in the preceding cryptographic primitive taxonomy diagram, cryptography is mainly divided into three categories: **keyless primitives**, **symmetric key primitives**, or **cryptography**, and **asymmetric key primitives**, or **cryptography**.

Keyless primitives and symmetric cryptography are discussed further in the next section, and we will cover asymmetric cryptography, or public key cryptography, in *Chapter 4, Public Key Cryptography*.

Keyless primitives

In this section, we will introduce two keyless primitives, namely, random numbers and hash functions.

Random numbers

Randomness provides an indispensable element for the security of the cryptographic protocols. It is used for the generation of keys and in encryption algorithms.



Private keys must be generated randomly so that they are unpredictable and not so easy to guess.

Randomness ensures that operations of a cryptographic algorithm do not become predictable enough to allow cryptanalysts to predict the outputs and operations of the algorithm, which will make the algorithm insecure. It is quite a feat to generate suitable randomness with a high degree of uncertainty, but there are methods that ensure an adequate level of randomness is generated for use in cryptographic algorithms.

There are two categories of source of randomness, namely, **Random Number Generators (RNGs)** and **Pseudorandom Number Generators (PRNGs)**.

RNGs

RNGs are software or hardware systems that make use of the randomness available in the real world, that is, the analog world, where uncertainty in the environment produces randomness. This can be temperature variations, thermal noises from various electronic components, or acoustic noise. This is called **real randomness**. Other sources are based on the fact that a running computer system generates some randomness from the running processes, such as keystrokes or disk movements. These types of sources of randomness are not very practical due to the difficulty of acquiring this data or not having enough entropy. Also, these sources are not always available and could be available only for a limited time.



The measure of randomness is called **entropy**.

PRNGs

PRNGs are deterministic functions that work on the principle of using a random initial value called a **seed** to produce a random looking set of elements. PRNGs are commonly used to generate keys for encryption algorithms. A common example of a PRNG is the **Blum-Blum-Shub (BBS)**. PRNGs are a better alternative to RNGs due to their reliability and deterministic nature.



More information on BBS is available in the following original research paper, *Blum, L., Blum, M. and Shub, M., 1986. A simple unpredictable pseudo-random number generator. SIAM Journal on computing, 15(2), pp.364-383:*

https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf

With this, our discussion on random numbers is complete. This is a deep and vast subject, but we have covered enough to understand their need in cryptographic algorithms.

In the next section, we will look at a category of keyless cryptographic primitives that are not used to encrypt data; instead, they produce a fixed-length digest of the data that is provided as input. These constructions are called hash functions and play a very crucial role in the development of blockchain.

Hash functions

Hash functions are used to create fixed-length digests of arbitrarily long input strings. Hash functions are **keyless**, and they provide a **data integrity service**. They are usually built using iterated and dedicated hash function construction techniques.

Various families of hash functions are available, such as MD, SHA1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used for digital signatures and message authentication codes, such as HMACs. They have three security properties, namely, pre-image resistance, second pre-

image resistance, and collision resistance. These properties are explained later in this section.

Hash functions are also typically used to provide data integrity services. These can be used both as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications use hash functions as a means for generating PRNGs. There are two practical and three security properties of hash functions that must be met depending on the level of integrity required. These properties are discussed next.

Compression of arbitrary messages into fixed-length digests

This property relates to the fact that a hash function must be able to take an input text of any length and output a fixed-length compressed message. Hash functions produce a compressed output in various bit sizes, usually between 128-bits and 512-bits.

Easy to compute

Hash functions are efficient and fast one-way functions. It is required that hash functions be very quick to compute regardless of the message size. The efficiency may decrease if the message is too big, but the function should still be fast enough for practical use.

In the following section, the security properties of hash functions are discussed.

Pre-image resistance

This property can be explained by using the simple equation:

$$h(x) = y$$

Here, h is the hash function, x is the input, and y is the hash. The first security property requires that y cannot be reverse-computed to x . x is considered a pre-image of y , hence the name **pre-image resistance**. This is also called a one-way property.

Second pre-image resistance

The **second pre-image resistance** property requires that given x and $h(x)$, it is almost impossible to find any other message m , where $m \neq x$ and $hash\ of\ m = hash\ of\ x$ or $h(m) = h(x)$. This property is also known as **weak collision resistance**.

Collision resistance

The **collision resistance** property requires that two different input messages should not hash to the same output. In other words, $h(x) \neq h(z)$. This property is also known as **strong collision resistance**.

All these properties are shown in the following diagrams:

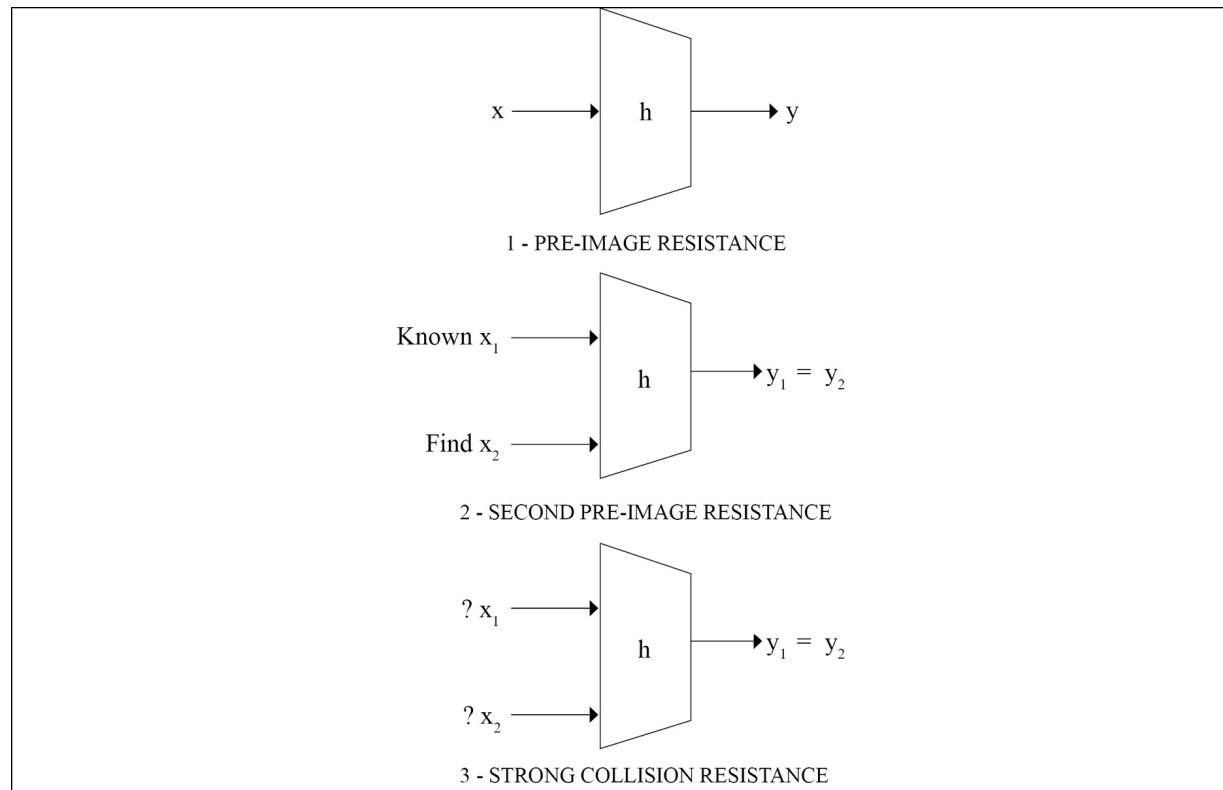


Figure 3.3: Three security properties of hash functions

Due to their very nature, hash functions will always have some collisions. This is a situation where two different messages hash to the same output. However, they should be computationally impractical to find. A concept

known as the **avalanche effect** is desirable in all cryptographic hash functions. The avalanche effect specifies that a small change, even a single character change in the input text, will result in an entirely different hash output.

Hash functions are usually designed by following an iterated hash functions approach. With this method, the input message is compressed in multiple rounds on a block-by-block basis in order to produce the compressed output. A popular type of iterated hash function is the **Merkle-Damgard construction**. This construction is based on the idea of dividing the input data into equal block sizes and then feeding them through the compression functions in an iterative manner. The collision resistance of the property of compression functions ensures that the hash output is also collision-resistant. Compression functions can be built using block ciphers. In addition to Merkle-Damgard, there are various other constructions of compression functions proposed by researchers, for example, Miyaguchi-Preneel and Davies-Meyer.



Details of these schemes are available in the excellent book, *Katz, J., Menezes, A.J., Van Oorschot, P.C. and Vanstone, S.A., 1996, Handbook of applied cryptography, CRC press.*

Refer to *Chapter 9, Hash Functions and Data Integrity:*
<http://cacr.uwaterloo.ca/hac/about/chap9.pdf>

Multiple categories of hash functions are introduced in the following section.

Message Digest

Message Digest (MD) functions were prevalent in the early 1990s. MD4 and MD5 fall into this category. Both MD functions were found to be insecure and are not recommended for use anymore. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms

The following list describes the most common **Secure Hash Algorithms (SHAs)**:

SHA-0: This is a 160-bit function introduced by the U.S. **National Institute of Standards and Technology (NIST)** in 1993.

SHA-1: SHA-1 was introduced in 1995 by NIST as a replacement for SHA-0. This is also a 160-bit hash function. SHA-1 is used commonly in SSL and TLS implementations. It should be noted that SHA-1 is now considered insecure, and it is being deprecated by certificate authorities. Its usage is discouraged in any new implementations.

SHA-2: This category includes four functions defined by the number of bits of the hash: SHA-224, SHA-256, SHA-384, and SHA-512.

SHA-3: This is the latest family of SHA functions. SHA3-224, SHA3-256, SHA3-384, and SHA3-512 are members of this family. SHA3 is a NIST-standardized version of Keccak. Keccak uses a new approach called **sponge construction** instead of the commonly used Merkle-Damgard transformation.

RIPEMD: RIPEMD is the acronym for **RACE Integrity Primitives Evaluation Message Digest**. It is based on the design ideas used to build MD4. There are multiple versions of RIPEMD, including 128-bit, 160-bit, 256-bit, and 320-bit.

Whirlpool: This is based on a modified version of the Rijndael cipher known as *W*. It uses the Miyaguchi-Preneel compression function, which is a type of one-way function used for the compression of two fixed-length inputs into a single fixed-length output. It is a single block length compression function.

Hash functions have many practical applications ranging from simple file integrity checks and password storage to use in cryptographic protocols and algorithms. They are used in hash tables, distributed hash tables, bloom filters, virus fingerprinting, **peer-to-peer (P2P)** file-sharing, and many other applications.

Hash functions play a vital role in blockchain. The **Proof-of-Work (PoW)** function in particular uses SHA-256 twice in order to verify the computational effort spent by miners. RIPEMD 160 is used to produce Bitcoin addresses. This will be discussed further in later chapters.

In the next section, the design of the SHA algorithm is introduced.

Design of Secure Hash Algorithms (SHA)

In the following section, you will be introduced to the design of SHA-256 and SHA-3. Both of these are used in Bitcoin and Ethereum, respectively. Ethereum uses Keccak, which is the original algorithm presented to NIST, rather than NIST standard SHA-3. NIST, after some modifications, such as an increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

Design of SHA-256

SHA-256 has an input message size limit of $2^{64} - 1$ bits. The block size is 512 bits, and it has a word size of 32 bits. The output is a 256-bit digest.

The compression function processes a 512-bit message block and a 256-bit intermediate hash value. There are two main components of this function: the compression function and a message schedule.

The algorithm works as follows, in nine steps:

Pre-processing

1. **Padding** of the message is used to adjust the length of a block to 512 bits if it is smaller than the required block size of 512 bits.
2. **Parsing** the message into message blocks, which ensures that the message and its padding is divided into equal blocks of 512 bits.
3. Setting up the initial hash value, which consists of the eight 32-bit words obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. These initial values are

fixed and chosen to initialize the process. They provide a level of confidence that no backdoor exists in the algorithm.

Hash computation

4. Each message block is then processed in a sequence, and it requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
5. The message schedule is prepared.
6. Eight working variables are initialized.
7. The compression function runs 64 times.
8. The intermediate hash value is calculated.
9. Finally, after repeating steps 5 through 8 until all blocks (chunks of data) in the input message are processed, the output hash is produced by concatenating intermediate hash values.

At a high level, SHA-256 can be visualized in the following diagram:

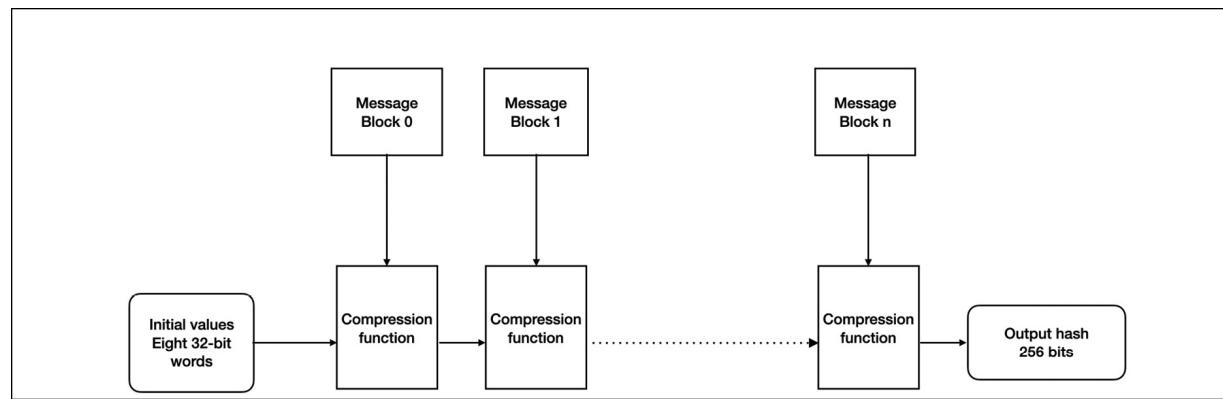


Figure 3.4: SHA-256 high level overview

As shown in the preceding diagram, SHA-256 is a Merkle Damgard construction that takes the input message and divides it into equal blocks (chunks of data) of 512 bits. Initial values (or initial hash values) or the initialization vector are composed of eight 32 bit words (256 bits) that are fed into the compression function with the first message. Subsequent blocks are fed into the compression function until all blocks are processed and finally, the output hash is produced.

The compression function of SHA-256 is shown in the following diagram:

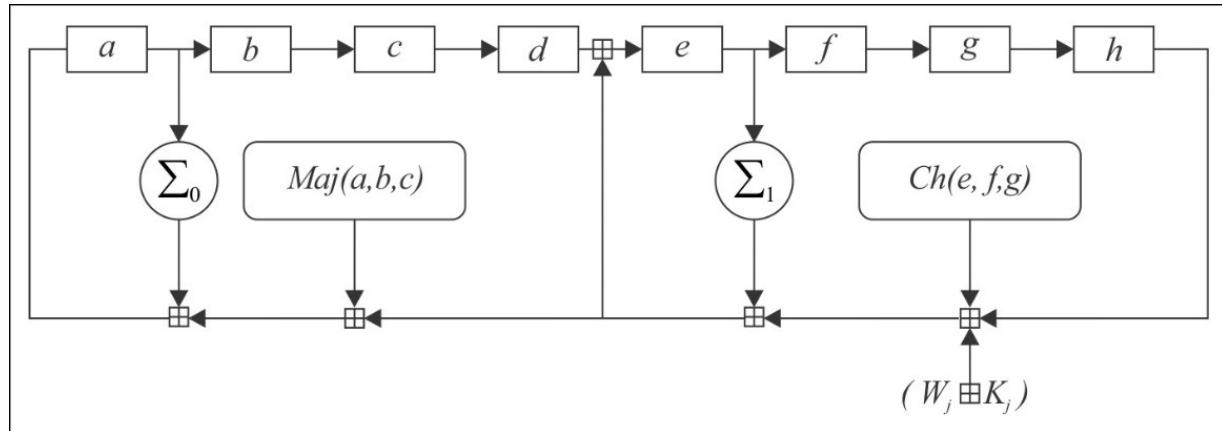


Figure 3.5: One round of an SHA-256 compression function

In the preceding diagram, a, b, c, d, e, f, g , and h are the registers for 8 working variables. **Maj** and **Ch** functions are applied bitwise. Σ_0 and Σ_1 perform bitwise rotation. The round constants are W_j and K_j , which are added in the main loop (compressor function) of the hash function, which runs 64 times.



The SHA-256 algorithm is used in Bitcoin's PoW algorithm.

With this, our introduction to SHA-256 is complete and next we explore a newer class of hash functions known as the SHA-3 algorithm.

Design of SHA-3 (Keccak)

The structure of **SHA-3** is very different from that of SHA-1 and SHA-2. The key idea behind SHA-3 is based on unkeyed permutations, as opposed to other typical hash function constructions that used keyed permutations. **Keccak** also does not make use of the Merkle-Damgard transformation that is commonly used to handle arbitrary-length input messages in hash functions. A newer approach, called **sponge and squeeze construction**, is used in Keccak. It is a random permutation model. Different variants of SHA-3 have been standardized, such as SHA3-224, SHA3-256, SHA3-384,

SHA3-512, SHAKE128, and SHAKE256. SHAKE128 and SHAKE256 are **extendable-output functions (XOFs)**, which allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model, which is the basis of SHA-3 or Keccak. Analogous to a sponge, the data (m input data) is first absorbed into the sponge after applying padding. It is then changed into a subset of permutation state using **XOR (exclusive OR)**, and then the output is squeezed out of the sponge function that represents the transformed state. The rate r is the input block size of the sponge function, while capacity c determines the security level:

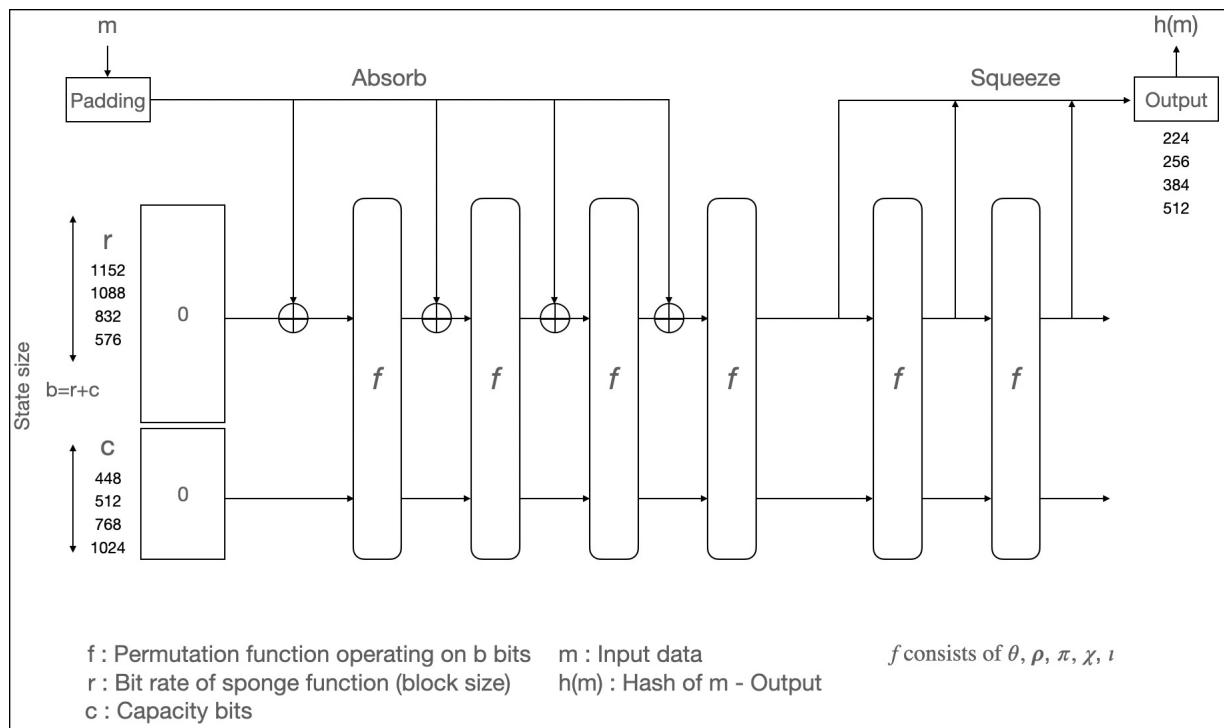


Figure 3.6: The SHA-3 absorbing and squeezing function in SHA3

In the preceding diagram, state size b is calculated by adding bit rate r and capacity bits c . r and c can be any values as long as sizes of $r + c$ are 25, 50, 100, 200, 400, 800, or 1600. The state is a 3-dimensional bit matrix. The initial state is set to 0. The data m is entered into the absorb phase block by block via XOR \oplus after applying padding.

The following table shows the value of bit rate r (block size) and capacity c required to achieve the desired output hash size under the most efficient setting of $r + c = 1600$:

r (block size)	c (capacity)	Output hash size
1152	448	224
1088	512	256
832	768	384
576	1024	512

The function f is a permutation function. It contains five transformation operations named Theta, Rho, Pi, Chi, and Iota described as follows.

θ – Theta: XOR bits in the state, used for mixing

ρ – Rho: Diffusion function performing rotation of bits

π – Pi: Diffusion function

χ – Chi: XOR each bit, bitwise combine

ι – Iota: Combination with round constants

The details of transformation operations is beyond the scope of this book. However, a high level function of each transformation is described previously. The key idea is to apply these transformations to achieve the **avalanche effect**, which we introduced earlier in this chapter. These five operations combined form a **round**. In the SHA-3 standard, the number of rounds is 24 to achieve the desired level of security.

We will see some applications and examples of constructions built using hash functions such as Merkle trees in *Chapter 11, Ethereum 101*.

OpenSSL example of hash functions

The following command will produce a hash of 256 bits of `Hello` messages using the SHA-256 algorithm:

```
$ echo -n 'Hello' | openssl dgst -sha256  
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764
```

Note that even a small change in the text, such as changing the case of the letter `H`, results in a big change in the output hash.



This phenomenon is known as the avalanche effect, where if the input is slightly changed, the output changes drastically so that cryptanalysts cannot make predictions on the input text by analyzing the output.

Now we run the following command to see the avalanche effect in action:

```
$ echo -n 'hello' | openssl dgst -sha256  
(stdin)= 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e730433e
```

Note that both outputs are completely different:

```
Hello: 18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:  
hello: 2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:  
04:33:62:93:8b:98:24
```

Usually, hash functions do not use a key. Nevertheless, if they are used with a key, then they can be used to create another cryptographic construct called **message authentication codes (MACs)**.

We will see how MACs and HMACs work after we've introduced symmetric key cryptography, because these constructions make use of keys for encryption.

Symmetric cryptography

Symmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is the same one that is used for decrypting the data. Thus, it is also known as **shared key cryptography**. The key must be established or agreed upon before the data exchange occurs between the communicating parties. This is the reason it is also called **secret key cryptography**.

A key, more precisely, a **cryptographic key**, is required for the encryption and decryption process and must be kept secret, hence it is called a secret key. There are many different types of keys depending upon the protocol. In symmetric key systems, only one key is required, which is shared between the sender and the receiver. It is used for both **encryption** and **decryption**, hence the name **symmetric key cryptography** or **shared key cryptography**, or just **symmetric cryptography**.

Other types of keys are **public keys** and **private keys**, which are generated in pairs for public key cryptography or asymmetric cryptography. Public keys are used for encrypting the plaintext, whereas private keys are used for decryption and are expected to be kept secret by the receiver. We will discuss asymmetric cryptography in the next chapter, *Chapter 4, Public Key Cryptography*.

Keys can also be **ephemeral** (temporary) or **static**. Ephemeral keys are intended to be used only for a short period of time, such as in a single session between the participants, whereas static keys are intended for long-term usage.

Another type of key is called the **master key**, which is used for the protection, encryption, decryption, and generation of other keys.

There are different methods to generate keys. These methods are listed as follows:

1. Random, where a random number generator is used to generate a random set of bytes that can be used as a key.
2. Key derivation-based, where a single key or multiple keys are derived from a password. A **key derivation function (KDF)** is used for this purpose, taking the password as input, and converting that into a key. Commonly used key derivation functions are **Password-Based Key Derivation Function 1 (PBKDF1)**, PBKDF2, Argon 2, and Scrypt.



More information about PBKDFs can be found at
<https://tools.ietf.org/html/rfc8018>.

3. Using a key agreement protocol, two or more participants run a protocol that produces a key and that key is then shared between participants. In key agreement schemes, all participants contribute equally in the effort to generate the shared secret key. The most commonly used key agreement protocol is the Diffie-Hellman key exchange protocol.



KDFs are used in Ethereum wallets (keystore files) to generate an AES symmetric key that is used to encrypt the wallets. The KDF function used in Ethereum wallets is Scrypt. We will explain all this in great detail in *Chapter 11, Ethereum 101*.

4. In encryption schemes, there are also some random numbers that play a vital role in the operation of the encryption process. These concepts are presented as follows:

- The **nonce**: This is a number that can be used only once in a cryptographic protocol. It must not be reused. Nonces can be generated from a large pool of random numbers or they can also be sequential. The most common use of nonces is to prevent replay attacks in cryptographic protocols.

- The **initial value** or **initialization vector (IV)** is a random number, which is basically a nonce, but it must be chosen in an unpredictable manner. This means that it cannot be sequential. IVs are used extensively in encryption algorithms to provide increased security.
- The **salt**: Salt is a cryptographically strong random value that is typically used in hash functions to provide defense against dictionary or rainbow attacks. Using dictionary attacks, hashing-based password schemes can be broken by trying hashes of millions of words from a dictionary in a brute-force manner and matching it with the hashed password. If a salt is used, then a dictionary attack becomes difficult to run because a random salt makes each password unique, and secondly, the attacker will then have to run a separate dictionary attack for random salts, which is quite unfeasible.

Now that we've introduced symmetric cryptography, we're ready to take a look at MACs and HMACs.

Message authentication codes (MACs)

Message authentication codes (MACs) are sometimes called **keyed hash functions**, and they can be used to provide message integrity and authentication. More specifically, they are used to provide data origin authentication. These are symmetric cryptographic primitives that use a shared key between the sender and the receiver. MACs can be constructed using block ciphers or hash functions.

Hash-based MACs (HMACs)

Similar to the hash function, **hash-based MACs (HMACs)** produce a fixed-length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using the MAC and the receiver verifies it using the shared key. The key is hashed with the message using either of the two methods known as **secret prefix** or **secret suffix**. With the secret prefix method, the key is concatenated with the message; that is, the key

comes first and the message comes afterward, whereas with the secret suffix method, the key comes after the message, as shown in the following equations:

$$\text{Secret prefix: } M = \text{MAC}_k(x) = h(k||x)$$

$$\text{Secret suffix: } M = \text{MAC}_k(x) = h(x||k)$$

There are pros and cons to both methods. Some attacks on both schemes have occurred. There are HMAC constructions schemes that use various techniques, such as **ipad (inner padding)** and **opad (outer padding)** that have been proposed by cryptographic researchers. These are considered secure with some assumptions:

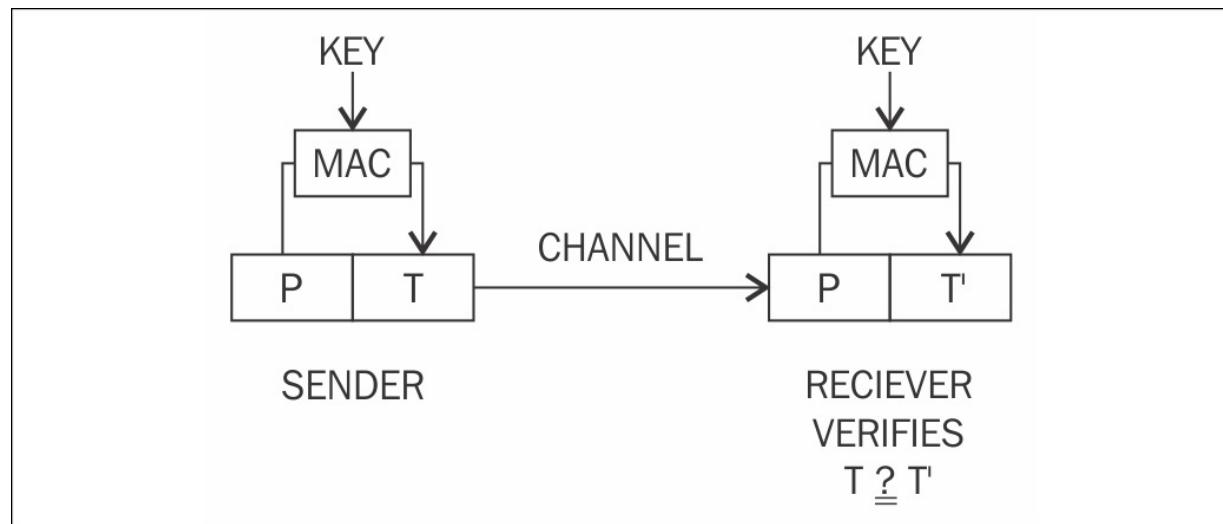


Figure 3.7: Operation of a MAC function

There are various powerful applications of hash functions used in peer-to-peer networks and blockchain technologies, such as Merkle trees, Patricia trees, and distributed hash tables.

There are two types of symmetric ciphers: **stream ciphers** and **block ciphers**. **Data Encryption Standard (DES)** and **Advanced Encryption Standard (AES)** are typical examples of block ciphers, whereas RC4 and A5 are commonly used stream ciphers.

Stream ciphers

Stream ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis (one bit at a time) to plaintext using a keystream. There are two types of stream ciphers: **synchronous stream ciphers** and **asynchronous stream ciphers**.

Synchronous stream ciphers are those where the keystream is dependent only on the key. Asynchronous stream ciphers have a keystream that is also dependent on the encrypted data.

In stream ciphers, encryption and decryption are the same function because they are simple modulo 2 additions or **XOR** operations. The fundamental requirement in stream ciphers is the security and randomness of keystreams. Various techniques ranging from PRNGs to true hardware RNGs have been developed to generate random numbers, and it is vital that all key generators be cryptographically secure:

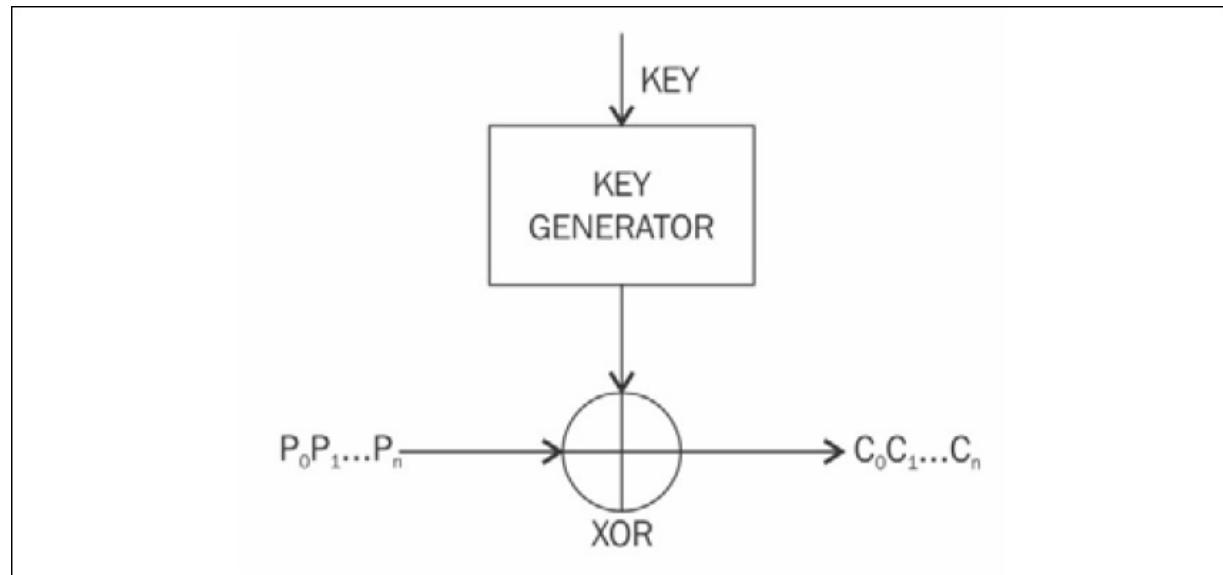


Figure 3.8: Operation of a stream cipher

Block ciphers

Block ciphers are encryption algorithms that break up the text to be encrypted (plaintext) into blocks of a fixed length and apply the encryption

block by block. Block ciphers are generally built using a design strategy known as a **Feistel cipher**. Recent block ciphers such as AES (Rijndael) have been built using a combination of substitution and permutation called a **Substitution-Permutation Network (SPN)**.

Feistel ciphers are based on the Feistel network, which is a structure developed by Horst Feistel. This structure is based on the idea of combining multiple rounds of repeated operations to achieve desirable cryptographic properties known as **confusion** and **diffusion**. Feistel networks operate by dividing data into two blocks (left and right) and processing these blocks via keyed **round functions** in iterations to provide sufficient pseudorandom permutations.

Confusion adds complexity to the relationship between the encrypted text and plaintext. This is achieved by substitution. In practice, A in plaintext is replaced by X in encrypted text. In modern cryptographic algorithms, substitution is performed using lookup tables called **S-boxes**. The **diffusion** property spreads the plaintext statistically over the encrypted data. This ensures that even if a single bit is changed in the input text, it results in changing at least half (on average) of the bits in the ciphertext. Confusion is required to make finding the encryption key very difficult, even if many encrypted and decrypted data pairs are created using the same key. In practice, this is achieved by transposition or permutation.

A key advantage of using a Feistel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process to achieve decryption. DES is a prime example of Feistel-based ciphers:

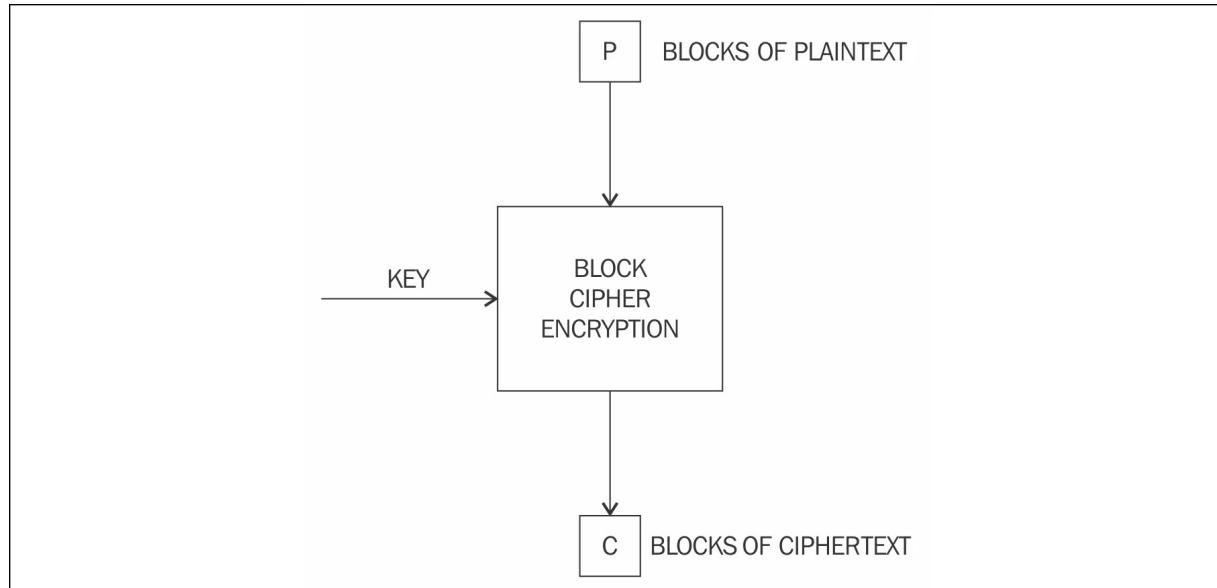


Figure 3.9: Simplified operation of a block cipher

Various modes of operation for block ciphers are **Electronic Code Book (ECB)**, **Cipher Block Chaining (CBC)**, **Output Feedback (OFB)** mode, and **Counter (CTR)** mode. These modes are used to specify the way in which an encryption function is applied to the plaintext. Some of these modes of block cipher encryption are introduced here.

Block encryption mode

In **block encryption mode**, the plaintext is divided into blocks of fixed length depending on the type of cipher used. Then the encryption function is applied to each block.

The most common block encryption modes are briefly discussed in the following sections.

Electronic codebook

Electronic codebook (ECB) is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm to each block of plaintext, one by one. This is the most straightforward mode, but it should not be used in practice as it is insecure and can reveal information:

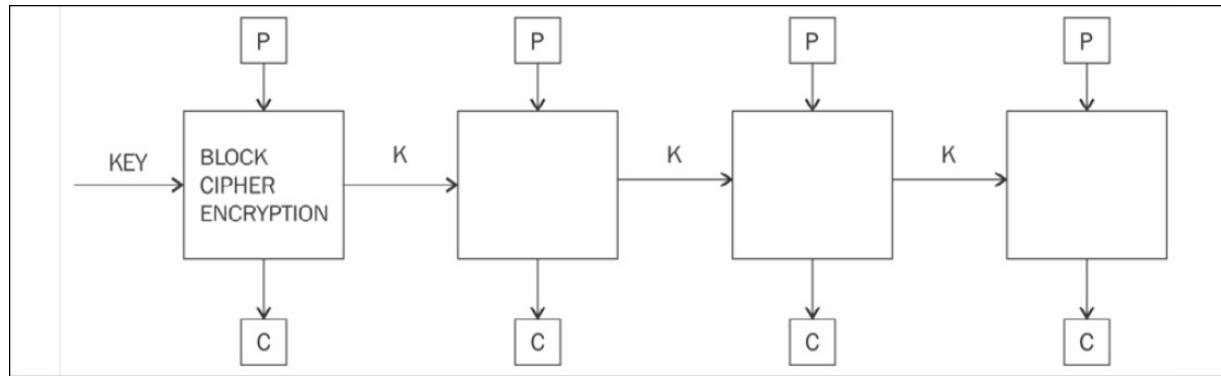


Figure 3.10: Electronic codebook mode for block ciphers

The preceding diagram shows that we have plain text P provided as an input to the block cipher encryption function along with a key, and ciphertext C is produced as output.

Cipher block chaining

In **cipher block chaining (CBC)** mode, each block of plaintext is XORed with the previously encrypted block. CBC mode uses the IV to encrypt the first block. It is recommended that the IV be randomly chosen:

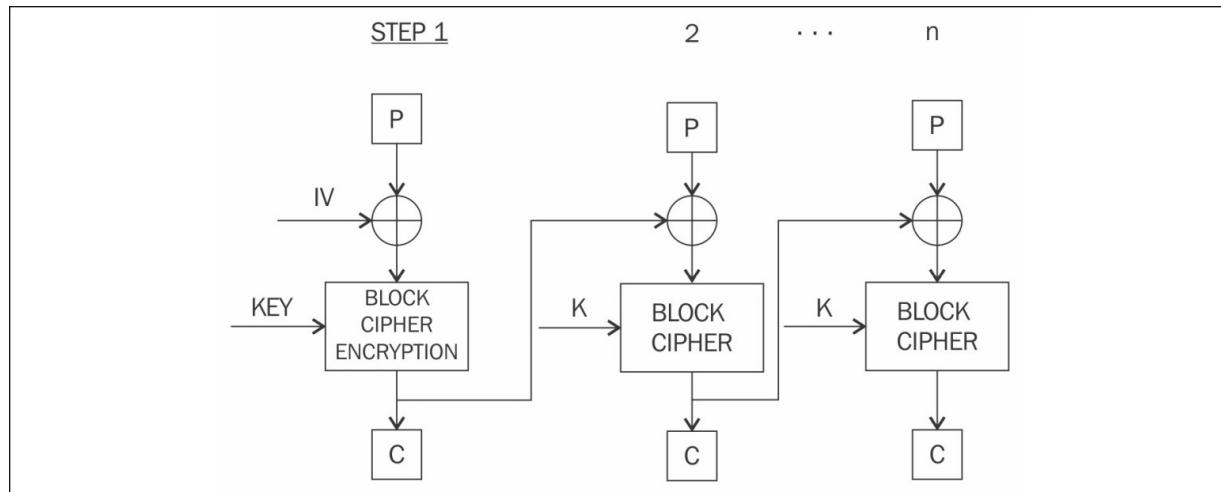


Figure 3.11: Cipher block chaining mode

Counter mode

The **counter (CTR) mode** effectively uses a block cipher as a stream cipher. In this case, a unique nonce is supplied that is concatenated with the counter value to produce a **keystream**:

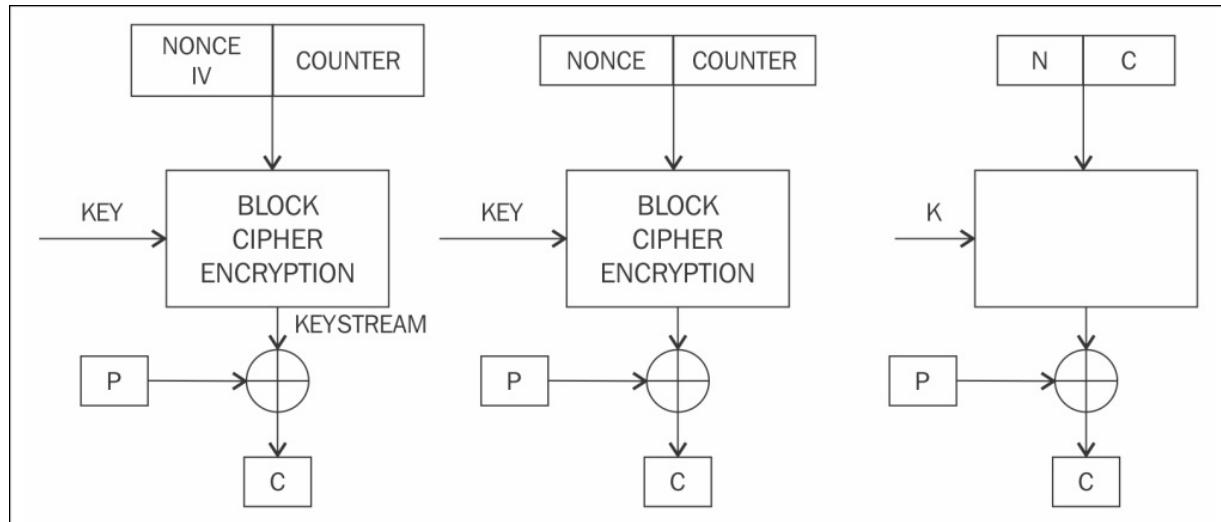


Figure 3.12: Counter mode

CTR mode, as shown in the preceding diagram, works by utilizing a **nonce (N)** and a **counter (C)** that feed into the **block cipher encryption** function. The block cipher encryption function takes the **secret key (KEY)** as input and produces a **keystream** (a stream of pseudorandom or random characters), which, when XORed with the **plaintext (P)**, produces the **ciphertext (C)**.

So far, we have discussed modes that are used to produce ciphertexts (encrypted data). However, there are other modes that can be used for different purposes. We discuss some of these in the following section.

Keystream generation mode

In **keystream generation mode**, the encryption function generates a keystream that is used in stream ciphers. Specifically, the keystream is usually XORed with the plaintext stream to produce encrypted text.

Message authentication mode

In **message authentication mode**, a **message authentication code (MAC)** is produced from an encryption function. A MAC is a cryptographic checksum that provides an *integrity service*. The most common method to generate a MAC using block ciphers is CBC-MAC, where a part of the last block of the chain is used as a MAC. In other words, block ciphers are used in the **cipher block chaining mode (CBC mode)** to generate a MAC. A MAC can be used to check if a message has been modified by an unauthorized entity. This can be achieved by encrypting the message with a key using the MAC function. The resulting message and the MAC of the message, once received by the receiver, are checked by encrypting the message received, again with the key, and comparing it with the MAC received from the sender. If they both match, then it means that the message has not been modified by some unauthorized entity, thus an integrity service is provided. If they don't match, then it means that the message has been altered by some unauthorized entity during transmission.

Any block cipher, for example, AES in CBC mode, can be used to generate a MAC. The MAC of the message is, in fact, the output of the last round of the CBC operation. The length of the MAC output is the same as the block length of the block cipher used to generate the MAC. It should also be noted that MACs work like **digital signatures**. However, they cannot provide a non-repudiation service due to their symmetric nature.

Cryptographic hash mode

Hash functions are primarily used to compress a message to a fixed-length digest. In **cryptographic hash mode**, block ciphers are used as a compression function to produce a hash of plaintext.

There are also other modes, such as **Cipher Feedback (CFB) mode**, **Galois Counter (GCM) mode**, and **Output Feedback (OFB) mode**, which are also used in various scenarios. Discussion of all these different modes is beyond the scope of this book. Interested readers may refer to any standard textbook on cryptography for further details.

With this, we have now concluded the introduction to block ciphers. We now introduce some concepts that are relevant to cryptography and are

extensively used in many applications. The primitives introduced next are also used in blockchain.

In the following section, we will introduce the design and mechanism of a currently market-dominant block cipher known as AES.

Before discussing AES, let's review some history about **DES** that led to the development of the new AES standard.

Data Encryption Standard (DES)

DES was introduced by **NIST** as a standard algorithm for encryption, and it was in widespread use during the 1980s and 1990s. However, it did not prove to be very resistant to brute force attacks, due to advances in technology and cryptography research. In July 1998, for example, the **Electronic Frontier Foundation (EFF)** broke DES using a special-purpose machine called EFF DES cracker (or **Deep Crack**).

DES uses a key of only 56 bits, which raised some concerns. This problem was addressed with the introduction of **Triple DES (3DES)**, which proposed the use of a 168-bit key by means of three 56-bit keys and the same number of executions of the DES algorithm, thus making brute-force attacks almost impossible. However, other limitations, such as slow performance and 64-bit block size, were not desirable.

Advanced Encryption Standard (AES)

In 2001, after an open competition, an encryption algorithm named Rijndael invented by cryptographers Joan Daemen and Vincent Rijmen was standardized as **AES** with minor modifications by NIST. So far, no attack has been found against AES that is more effective than the brute-force method. The original version of Rijndael permits different key and block

sizes of 128 bits, 192 bits, and 256 bits. In the AES standard, however, only a 128-bit block size is allowed. However, key sizes of 128 bits, 192 bits, and 256 bits are permissible.

How AES works

During AES algorithm processing, a 4×4 array of bytes known as the **state** is modified using multiple rounds. Full encryption requires 10 to 14 rounds, depending on the size of the key. The following table shows the key sizes and the required number of rounds:

Key size	Number of rounds required
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

Once the state is initialized with the input to the cipher, the following four operations are performed sequentially step by step to encrypt the input:

1. **AddRoundKey**: In this step, the state array is XORed with a **subkey**, which is derived from the master key.
2. **SubBytes**: This is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array.
3. **ShiftRows**: This step is used to shift each row to the left, except for the first one, in the state array in a cyclic and incremental manner.
4. **MixColumns**: Finally, all bytes are mixed in a linear fashion (linear transformation), column-wise.

This is one round of AES. In the final round (either the 10th, 12th, or 14th round, depending on the key size), stage 4 is replaced with **AddRoundKey**.

to ensure that the first three steps cannot be simply reversed, as shown in the following diagram:

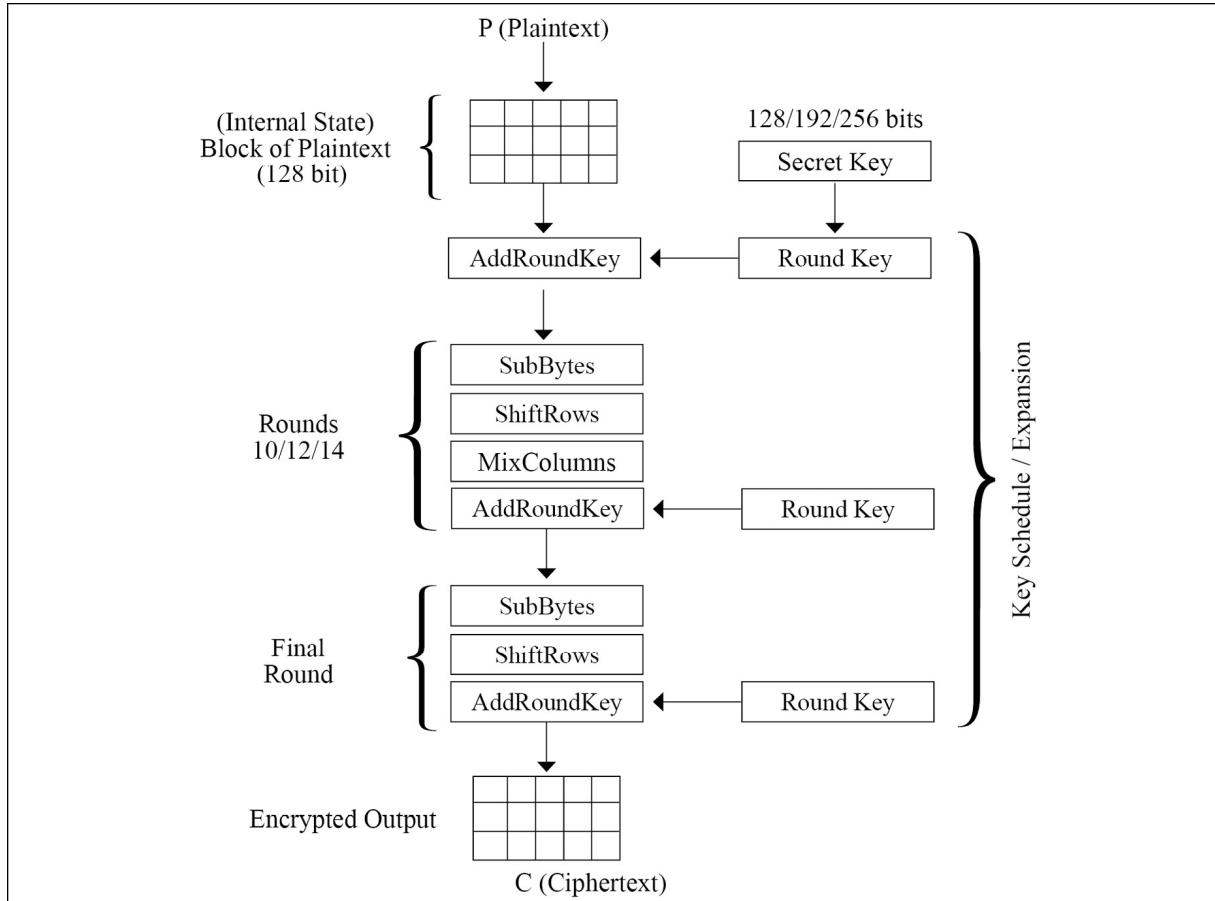


Figure 3.13: AES block diagram, showing the first round of AES encryption. In the final round, the mixing step is not performed



Various cryptocurrency wallets use AES encryption to encrypt locally-stored data. Bitcoin wallets use AES-256 in CBC mode to encrypt the private keys. In Ethereum wallets, AES-128-CTR is used; that is, AES 128-bit in counter mode is used to encrypt the private key. Peers in Ethereum also use AES in counter mode (AES CTR) to encrypt their **Peer to Peer (P2P)** communication.

An OpenSSL example of how to encrypt and decrypt using AES

We can use the OpenSSL command-line tool to perform encryption and decryption operations. An example is given here.

First, we create a plain text file to be encrypted:

```
$ echo Data to encrypt > message.txt
```

Now the file is created, we can run the OpenSSL tool with appropriate parameters to encrypt the file `message.txt` using 256-bit AES in CBC mode:

```
$ openssl enc -aes-256-cbc -in message.txt -out message.bin  
enter aes-256-cbc encryption password:  
Verifying - enter aes-256-cbc encryption password:
```

Once the operation completes, it will produce a `message.bin` file containing the encrypted data from the `message.txt` file. We can view this file, which shows encrypted contents of the `message.txt` file:

```
$ cat message.bin
```

```
Salted__w[s_y]h~?
```

Note that `message.bin` is a binary file. Sometimes, it is desirable to encode this binary file in a text format for compatibility/interoperability reasons. A common text encoding format is `base64`. The following commands can be used to create a base64-encoded message:

```
$ openssl enc -base64 -in message.bin -out message.b64
```

```
$ cat message.b64
```

```
U2FsdGVkX193uByIcwZf0Z7J1at+4L+Fj8/uzeDATJE=
```

In order to decrypt an AES-encrypted file, the following commands can be used. An example of `message.b64` from a previous example is used:

```
$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec  
enter aes-256-cbc decryption password:
```

```
$ cat message.dec  
Datatoencrypt
```

Readers may have noticed that no IV has been provided, even though it's required in all block encryption modes of operation except ECB. The reason for this is that OpenSSL automatically derives the IV from the given password. Users can specify the IV using the following switch:

```
-K/-iv , (Initialization Vector) should be provided in Hex.
```

In order to decode from base64, the following commands are used. Follow the `message.b64` file from the previous example:

```
$ openssl enc -d -base64 -in message.b64 -out message.ptx  
:~/Crypt$ cat message.ptx  
Salted_w s_ÿ h~? :~/Crypt$
```

There are many types of ciphers that are supported in OpenSSL. You can explore these options based on the examples provided thus far. A list of supported cipher types is shown as follows:

```

Cipher Types
-aes-128-cbc           -aes-128-ccm          -aes-128-cfb
-aes-128-cfb1          -aes-128-cfb8         -aes-128-ctr
-aes-128-ecb           -aes-128-ofb          -aes-192-cbc
-aes-192-ccm           -aes-192-cfb          -aes-192-cfb1
-aes-192-cfb1          -aes-192-ctr          -aes-192-ecb
-aes-192-ofb           -aes-256-cbc          -aes-256-ccm
-aes-256-cfb           -aes-256-cfb1         -aes-256-cfb8
-aes-256-ctr           -aes-256-ecb          -aes-256-ofb
-aes128                -aes192              -aes256
-bf                     -bf-cbc              -bf-cfb
-bf-ecb                -bf-ofb              -blowfish
-camellia-128-cbc     -camellia-128-cfb        -camellia-128-cfb1
-camellia-128-cfb8    -camellia-128-ecb        -camellia-128-ofb
-camellia-192-cbc     -camellia-192-cfb        -camellia-192-cfb1
-camellia-192-cfb8    -camellia-192-ecb        -camellia-192-ofb
-camellia-256-cbc     -camellia-256-cfb        -camellia-256-cfb1
-camellia-256-cfb8    -camellia-256-ecb        -camellia-256-ofb
-camellia128           -camellia192           -camellia256
-cast                  -cast-cbc            -cast5-cbc
-cast5-cfb             -cast5-ecb           -cast5-ofb
-des                   -des-cbc              -des-cfb
-des-cfb1              -des-cfb8             -des-ecb
-des-edc               -des-edc-cbc          -des-edc-cfb
-des-edc-ofb            -des-ede3              -des-ede3-cbc
-des-edc3-cfb           -des-ede3-cfb1          -des-ede3-cfb8
-des-edc3-ofb            -des-ofb              -des3
-desx                  -desx-cbc            -id-aes128-CCM
-id-aes128-wrap         -id-aes192-CCM          -id-aes192-wrap
-id-aes256-CCM          -id-aes256-wrap         -id-smime-alg-CMS3DESwrap
-idea                  -idea-cbc             -idea-cfb
-idea-ecb              -idea-ofb              -rc2
-rc2-40-cbc             -rc2-64-cbc           -rc2-cbc
-rc2-cfb                -rc2-ecb              -rc2-ofb
-rc4                   -rc4-40               -seed
-seed-cbc              -seed-cfb             -seed-ecb
-seed-ofb

```

Figure 3.14: Some of the cipher types available in OpenSSL



Readers might see a different number of supported ciphers depending on the version of the OpenSSL tool being used.

The OpenSSL tool can be used to experiment with all the ciphers shown in the preceding screenshot. We will also use OpenSSL in the next chapter to demonstrate various public key cryptographic primitives and protocols.

Summary

This chapter introduced symmetric key cryptography. We started with basic mathematical definitions and cryptographic primitives. After this, we introduced the concepts of stream and block ciphers along with the working modes of block ciphers. Moreover, we introduced some practical exercises using OpenSSL to complement the theoretical concepts covered.

In the next chapter, we will introduce public key cryptography, which is used extensively in blockchain technology and has very interesting properties.

4

Public Key Cryptography

In this chapter, you will be introduced to the concepts and practical aspects of **public key cryptography**, also called **asymmetric cryptography** or **asymmetric key cryptography**. We will continue to use OpenSSL, as we did in the previous chapter, to experiment with some applications of cryptographic algorithms so that you can gain hands-on experience. We will start with the theoretical foundations of public key cryptography and will gradually build on the concepts with relevant practical exercises. After this, we will introduce some new and advanced cryptography constructs.

Before discussing cryptography further, some mathematical terms and concepts need to be explained in order to build a foundation for fully understanding the material provided later in this chapter. The next section serves as a basic introduction to these concepts. An explanation with proofs and relevant background for all of these terms would require somewhat complex mathematics, which is beyond the scope of this book. More details on these topics can be found in any standard number theory, algebra, or cryptography book.

Mathematics

As the subject of cryptography is based on mathematics, this section will introduce some basic concepts that will help you understand the concepts presented later.

Modular arithmetic

Also known as clock arithmetic, numbers in modular arithmetic wrap around when they reach a certain fixed number. This fixed number is a positive number called **modulus** (sometimes abbreviated to **mod**), and all operations are performed concerning this fixed number.

Modular arithmetic is analogous to a 12-hour clock; there are numbers from 1 to 12. When 12 is reached, the numbers start from 1 again. Imagine that the time is 9:00 now; 4 hours from now, it will be 1:00 because the numbers *wrap around* at 12 and start from 1 again. In normal addition, this would be $9 + 4 = 13$, but that is not the case on a 12-hour clock; it is 1:00.

In other words, this type of arithmetic deals with the remainders after the division operation. For example, $50 \bmod 11$ is 6 because $50 / 11$ leaves a remainder of 6.

Sets

These are collections of distinct objects, for example, $X = \{1, 2, 3, 4, 5\}$.

Fields

A field is a set in which all its elements form an additive and multiplicative group. It satisfies specific axioms for addition and multiplication. For all group operations, the **distributive law** is also applied.

The law dictates that the same sum or product will be produced, even if any of the terms or factors are reordered.

Finite fields

A **finite field** is one with a finite set of elements. Also known as **Galois fields**, these structures are of particular importance in cryptography as they can be used to produce accurate and error-free results of arithmetic

operations. For example, prime finite fields are used in **Elliptic Curve Cryptography (ECC)** to construct discrete logarithm problems.

Prime fields

A **prime field** is a finite one with a prime number of elements. It has specific rules for addition and multiplication, and each non-zero element in the field has an inverse. Addition and multiplication operations are performed modulo **p**, that is, modulo a prime number.

Groups

A group is a commutative set with an operation that combines two elements of the set. The group operation is closed and associated with a defined identity element. Additionally, each element in the set has an inverse.

Closure (closed) means that if, for example, elements **A** and **B** are in the set, then the resultant element after performing an operation on the elements is also in the set. **Associative** means that the grouping of elements does not affect the result of the operation.

Four group axioms must be satisfied for a set to qualify as a group. These group axioms include closure, associativity, an identity element, and an inverse element.

Abelian groups

An **abelian group** is formed when the operation on the elements of a set is commutative. The commutative law means that changing the order of the elements does not affect the result of the operation, for example, **A X B = B X A**.

The key difference is that in an abelian group, closure, associativity, identity element, inverse element, and commutativity abelian group axioms are satisfied, whereas in a group, only the first four axioms are required to be

satisfied; that is, closure, associativity, an identity element, and an inverse element.

Rings

If more than one operation can be defined over an abelian group, that group becomes a **ring**. There are also specific properties that need to be satisfied. A ring must have closure and associative and distributive properties.

Cyclic groups

A **cyclic group** is a type of group that can be generated by a single element called the group generator.

Order

This is the number of elements in a field. It is also known as the **cardinality** of the field.

This completes a basic introduction to some mathematical concepts involved in cryptography. In the next section, you will be introduced to cryptography concepts.

Asymmetric cryptography

Asymmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is different from the key that is used to decrypt the data. These keys are called private and public keys, respectively, which why asymmetric cryptography is also known as **public key cryptography**. It uses both public and private keys to encrypt and decrypt data, respectively. Various asymmetric cryptography schemes are in use,

including **RSA** (named after its founders, Rivest, Shamir, and Adelman), **DSA (Digital Signature Algorithm)**, and **ElGamal** encryption.

An overview of public-key cryptography is shown in the following diagram:

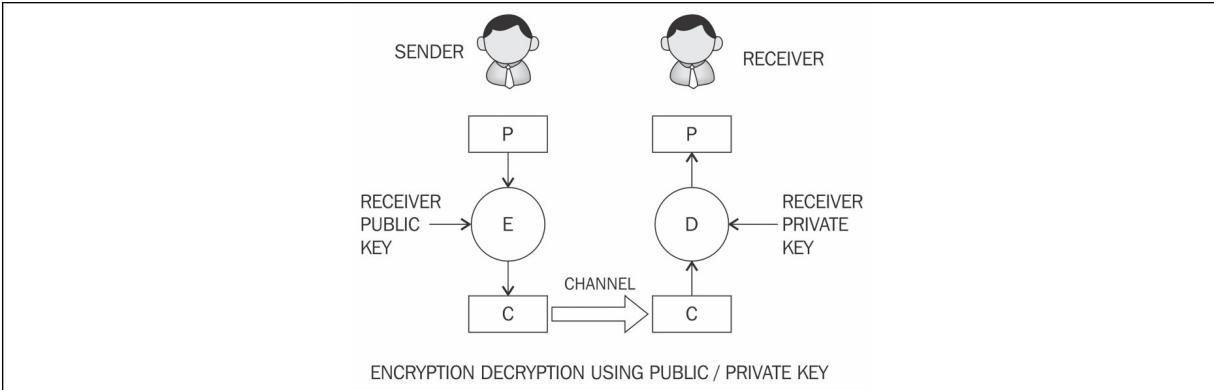


Figure 4.1: Encryption/decryption using public/private keys

The preceding diagram illustrates how a sender encrypts data **P** using the recipient's public key and encryption function **E**, and produces an output encrypted data **C**, which is then transmitted over the network to the receiver. Once it reaches the receiver, it can be decrypted using the receiver's private key by feeding the **C** encrypted data into function **D**, which will output plaintext **P**. This way, the private key remains on the receiver's side, and there is no need to share keys in order to perform encryption and decryption, which is the case with symmetric encryption.

The following diagram shows how the receiver uses public key cryptography to verify the integrity of the received message. In this model, the sender signs the data using their private key and transmits the message across to the receiver. Once the message is received, it is verified for integrity by the sender's public key.

It's worth noting that there is no encryption being performed in this model. It is simply presented here to help you thoroughly understand the material covering message authentication and validation that will be provided later in this chapter:

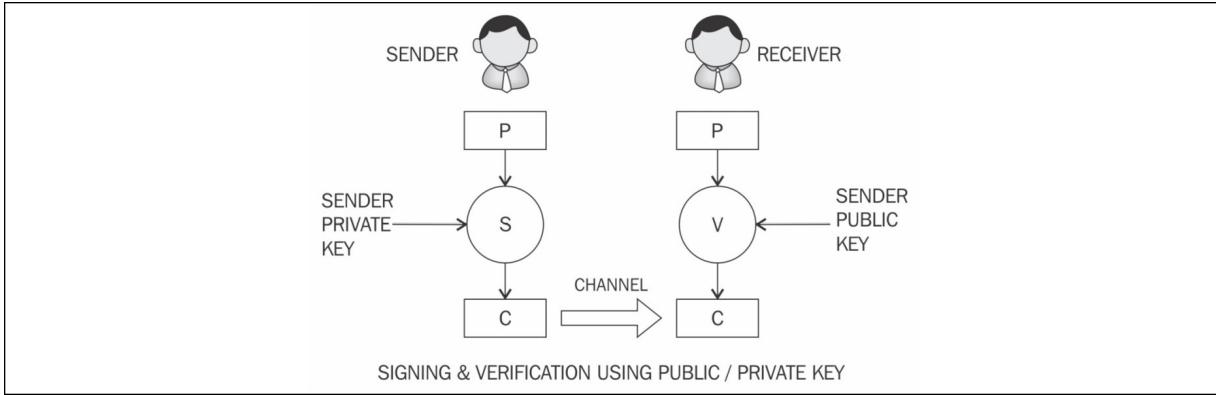


Figure 4.2: Model of a public-key cryptography signature scheme

The preceding diagram shows that the sender digitally signs the plaintext **P** with their private key using signing function **S**, and produces data **C**, which is sent to the receiver, who verifies **C** using the sender's public key and function **V** to ensure the message has indeed come from the sender.

Security mechanisms offered by public key cryptosystems include key establishment, digital signatures, identification, encryption, and decryption.

Key establishment mechanisms are concerned with the design of protocols that allow the setting up of keys over an insecure channel. Non-repudiation (defined in *Chapter 3, Symmetric Cryptography*, as the assurance that an action, once taken by someone, cannot be denied later) services, a very desirable property in many scenarios, can be provided using **digital signatures**. Sometimes, it is important not only to authenticate a user but also to identify the entity involved in a transaction. This can also be achieved by a combination of digital signatures and **challenge-response protocols**. Finally, the encryption mechanism to provide confidentiality can also be obtained using public key cryptosystems, such as RSA, ECC, and ElGamal.

Public key algorithms are slower in terms of computation than symmetric key algorithms. Therefore, they are not commonly used in the encryption of large files or the actual data that requires encryption. They are usually used to exchange keys for symmetric algorithms. Once the keys are established securely, symmetric key algorithms can be used to encrypt the data.

Public key cryptography algorithms are based on various underlying mathematical functions. The three main **categories of asymmetric algorithms** are described here.

Integer factorization

Integer factorization schemes are based on the fact that large integers are very hard to factor. RSA is a prime example of this type of algorithm.

Discrete logarithm

A **discrete logarithm scheme** is based on a problem in modular arithmetic. It is easy to calculate the result of a modulo function, but it is computationally impractical to find the exponent of the generator. In other words, it is extremely difficult to find the input from the result (output). This is called a one-way function.

For example, consider the following equation:

$$3^2 \bmod 10 = 9$$

Now, given 9, the result of the preceding equation, finding 2, which is the exponent of the generator 3 in the equation, is extremely hard to determine. This difficult problem is commonly used in the **Diffie-Hellman** key exchange and digital signature algorithms.

Elliptic curves

The **elliptic curves algorithm** is based on the discrete logarithm problem discussed previously, but in the context of elliptic curves. An **elliptic curve** is an algebraic cubic curve over a field, which can be defined by an equation, as shown here. The curve is non-singular, which means that it has no cusps or self-intersections. It has two variables a and b , as well as a point of infinity:

$$y^2 = x^3 + ax + b$$

Here, a and b are integers whose values are elements of the field on which the elliptic curve is defined. Elliptic curves can be defined over reals, rational numbers, complex numbers, or finite fields. For cryptographic purposes, an elliptic curve over prime finite fields is used instead of real numbers. Additionally, the prime should be greater than 3. Different curves can be generated by varying the values of a and/or b .

The most prominently used cryptosystems based on elliptic curves are the **Elliptic Curve Digital Signatures Algorithm (ECDSA)** and the **Elliptic Curve Diffie-Hellman (ECDH)** key exchange.

In this section, we discovered the basic principle of public key cryptography and relevant categories of asymmetric algorithms. This understanding will serve as a foundation for the concepts introduced next in this chapter. One of the key concepts in asymmetric cryptography is of public and private keys, which we'll describe next.

Public and private keys

A **private key**, as the name suggests, is a randomly generated number that is kept secret and held privately by its users. Private keys need to be protected and no unauthorized access should be granted to that key; otherwise, the whole scheme of public key cryptography is jeopardized, as this is the key that is used to decrypt messages. Private keys can be of various lengths, depending on the type and class of algorithms used. For example, in RSA, typically, a key of 1,024 bits or 2,048 bits is used. The 1,024-bit key size is no longer considered secure, and at least a 2,048-bit key size is recommended.

A **public key** is freely available and published by the private key owner. Anyone who would then like to send the publisher of the public key an encrypted message can do so, by encrypting the message using the published public key and sending it to the holder of the private key. No one else is able to decrypt the message because the corresponding private key is

held securely by the intended recipient. Once the public key-encrypted message is received, the recipient can decrypt the message using the private key. There are a few concerns, however, regarding public keys. These include authenticity and identification of the publisher of the public keys.

In the following section, we will introduce two examples of asymmetric key cryptography: **RSA** and **ECC**. RSA is the first implementation of public key cryptography where ECC is used extensively in blockchain technology.

RSA

RSA was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adelman, hence the name RSA. This type of public key cryptography is based on the integer factorization problem, where the multiplication of two large prime numbers is easy, but it is difficult to factor the product (the result of the multiplication) back to the two original numbers.

The crux of the work involved with the RSA algorithm happens during the key generation process. An RSA key pair is generated by performing the following steps:

1. Modulus generation:

- Select p and q , which are very large prime numbers
- Multiply p and q , $n=p \cdot q$ to generate modulus n

2. Generate the co-prime:

- Assume a number called e .
- e should satisfy a certain condition; that is, it should be greater than 1 and less than $(p-1)(q-1)$. In other words, e must be a number such that no number other than 1 can be divided into e and $(p-1)(q-1)$. This is called a **co-prime**, that is, e is the co-prime of $(p-1)(q-1)$.

3. Generate the public key:

- The modulus generated in *step 1* and co-prime e generated in *step 2* is a pair that is a public key. This part is the public part that can

be shared with anyone; however, p and q need to be kept secret.

4. Generate the private key:

- The private key is called d here and is calculated from p , q , and e . The private key is basically the inverse of e modulo $(p-1)(q-1)$. As an equation, it is as follows:

$$ed = 1 \bmod (p-1)(q-1)$$

Usually, an extended Euclidean algorithm is used to calculate d . This algorithm takes p , q , and e and calculates d . The key idea in this scheme is that anyone who knows p and q can easily calculate private key d by applying the extended Euclidean algorithm. However, someone who does not know the value of p and q cannot generate d . This also implies that p and q should be large enough for the modulus n to become extremely difficult (computationally impractical) to factor.



The original RSA paper is available here:

<http://web.mit.edu/6.857/OldStuff/Fall03/ref/rivest78method.pdf>.

Rivest, R.L., Shamir, A. and Adleman, L., 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), pp.120-126.

Now, let's see how encryption and decryption operations are performed using RSA. RSA uses the following equation to produce ciphertext:

$$C = P^e \bmod n$$

This means that plaintext P is raised to the power of e and then reduced to modulo n . Decryption in RSA is provided in the following equation:

$$P = C^d \bmod n$$

This means that the receiver who has a public key pair (n, e) can decipher the data by raising C to the value of the private key d , and then reducing to modulo n .

We'll look at a practical example of encryption and decryption with RSA later in the chapter. For now, let's explore the fundamentals of ECC.

Elliptic curve cryptography

ECC is based on the discrete logarithm problem founded upon elliptic curves over finite fields (Galois fields). The main benefit of ECC over other types of public key algorithms is that it requires a smaller key size, while providing the same level of security as, for example, RSA. Two notable schemes that originate from ECC are ECDH for key exchange and ECDSA for digital signatures.

ECC can also be used for encryption, but it is not usually used for this purpose in practice. Instead, key exchange and digital signatures are used more commonly. As ECC needs less space to operate, it is becoming very popular on embedded platforms and in systems where storage resources are limited. By comparison, the same level of security can be achieved with ECC when only using 256-bit operands as compared to 3,072 bits in RSA.

Mathematics behind ECC

To understand ECC, a basic introduction to the underlying mathematics is necessary. An elliptic curve is basically a type of polynomial equation known as the **Weierstrass equation**, which generates a curve over a finite field. The most commonly used field is where all arithmetic operations are performed modulo a prime number p . Elliptic curve groups consist of points on the curve over a finite field.

An elliptic curve is defined in the following equation:

$$y^2 = x^3 + Ax + B \text{ mod } P$$

Here, A and B belong to a finite field Zp or Fp (prime finite field), along with a special value called the point of infinity. The point of infinity ∞ is used to provide identity operations for points on the curve.

Furthermore, a condition also needs to be met that ensures that the equation mentioned earlier has no repeated roots. This means that the curve is non-singular.

The condition is described in the following equation, which is a standard requirement that needs to be met. More precisely, this ensures that the curve is non-singular:

$$4a^3 + 27b^2 \neq 0 \text{ mod } p$$

To construct the discrete logarithm problem based on elliptic curves, a large enough cyclic group is required. First, the group elements are identified as a set of points that satisfy the previous equation. After this, group operations need to be defined on these points.

Basic group operations on elliptic curves are point addition and point doubling. **Point addition** is a process where two different points are added, and **point doubling** means that the same point is added to itself.

Point addition

Point addition is shown in the following diagram. This is a geometric representation of point addition on elliptic curves. In this method, a diagonal line is drawn through the curve that intersects the curve at two points shown below **P** and **Q**, which yields a third point between the curve and the line.

This point is mirrored as **P+Q**, which represents the result of the addition as **R**. This is shown as **P+Q** in the following diagram:

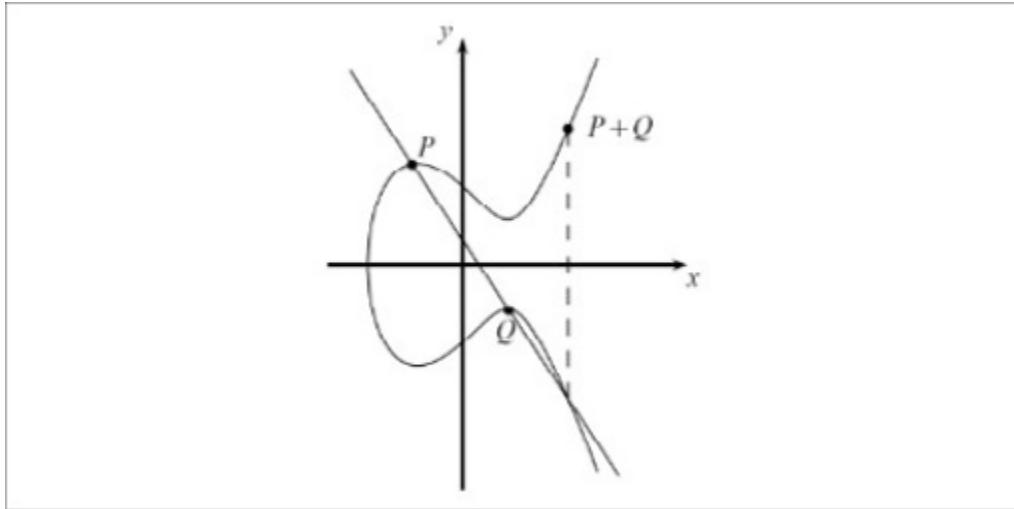


Figure 4.3: Point addition over \mathbb{R}

The group operation denoted by the + sign for addition yields the following equation:

$$P + Q = R$$

In this case, two points are added to compute the coordinates of the third point on the curve:

$$P + Q = R$$

More precisely, this means that coordinates are added, as shown in the following equation:

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

The equation of point addition is as follows:

$$X_3 = s^2 - x_1 - x_2 \bmod p$$

$$y_3 = s(x_1 - x_3) \cdot y_1 \bmod p$$

Here, we can see the result of the preceding equation:

$$S = \frac{(y_2 - y_1)}{(x_2 - x_1)} \text{ mod } p$$

S in the preceding equation depicts the line going through P and Q .

An example of point addition is shown here. It was produced using Certicom's online calculator. This example shows the addition and solutions for the equation over finite field F_{23} . This is in contrast to the example shown earlier, which is over real numbers and only shows the curve but provides no solutions to the equation:

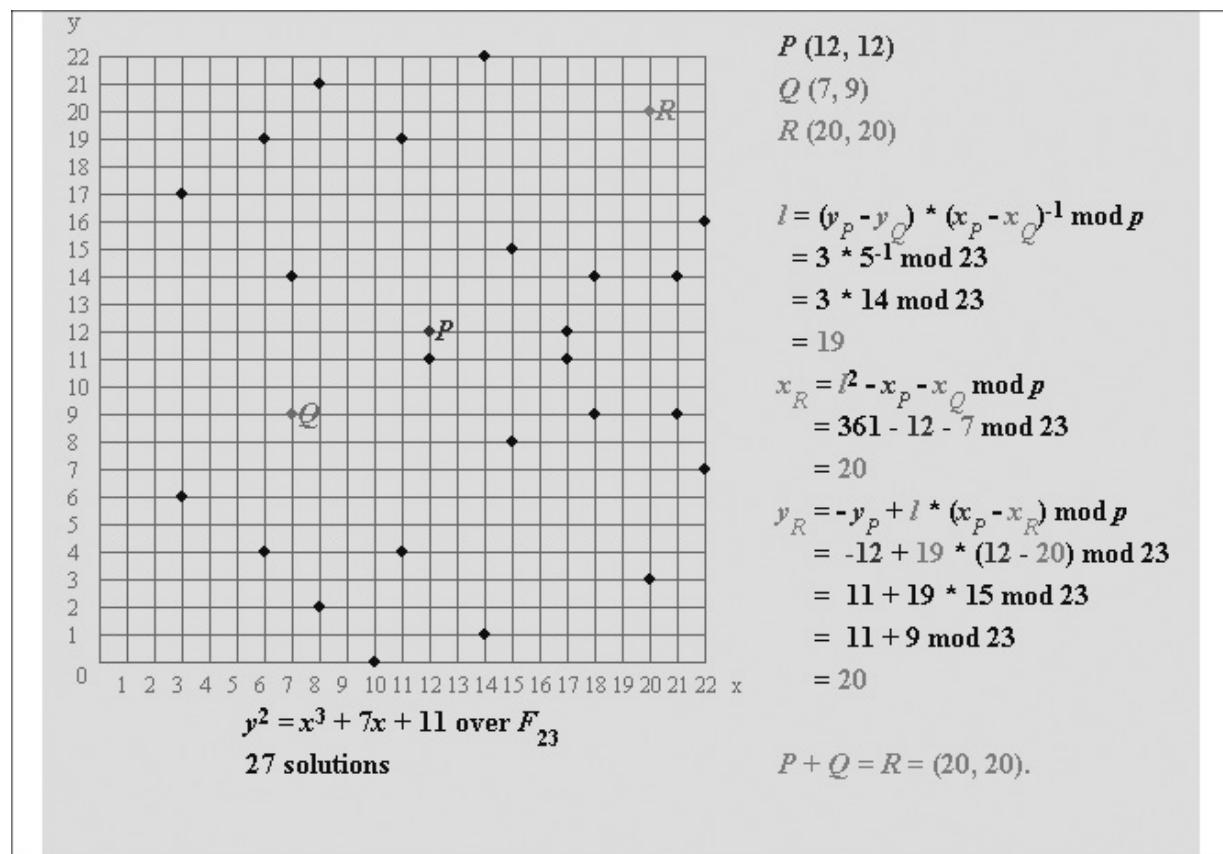


Figure 4.4: Example of point addition

In the preceding example, the graph on the left-hand side shows the points that satisfy this equation:

$$y^2 = x^3 + 7x + 11$$

There are 27 solutions to the equation shown earlier over finite field F_{23} . P and Q are chosen to be added to produce point R . Calculations are shown on the right-hand side, which calculates the third point R . Note that in the preceding graph, in the calculation shown on the right-hand side, l is used to depict the line going through P and Q .

As an example, to show how the equation is satisfied by the points shown in the graph, a point (x, y) is picked up where $x = 3$ and $y = 6$. This point can be visualized in the graph shown here. Notice the point at coordinate $(3, 6)$, indicated by an arrow:

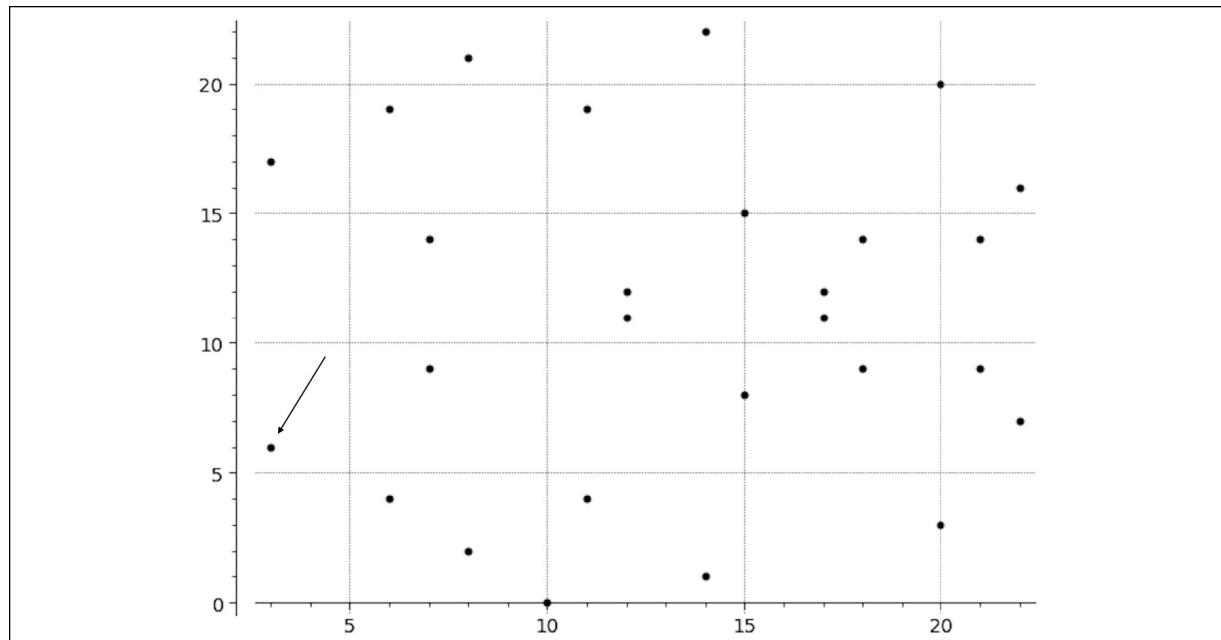


Figure 4.5: Point $(3, 6)$ shown with an arrow (this graph was generated using sageMath)

Using these values shows that the equation is indeed satisfied:

$$y^2 \bmod 23 = x^3 + 7x + 11 \bmod 23$$

$$6^2 \bmod 23 = 3^3 + 7(3) + 11 \bmod 23$$

$$36 \bmod 23 = 59 \bmod 23$$

$$13 = 13$$

The next section introduces the concept of point doubling, which is another operation that can be performed on elliptic curves.

Point doubling

The other group operation on elliptic curves is called **point doubling**. This is a process where P is added to itself. This is the case when P and Q are at the same point, so effectively the operation becomes adding the point to itself, and is therefore called point doubling. In this method, a tangent line is drawn through the curve, as shown in the following graph. The second point is obtained, which is at the intersection of the tangent line drawn and the curve. This point is then mirrored to yield the result, which is shown as $2P = P + P$:

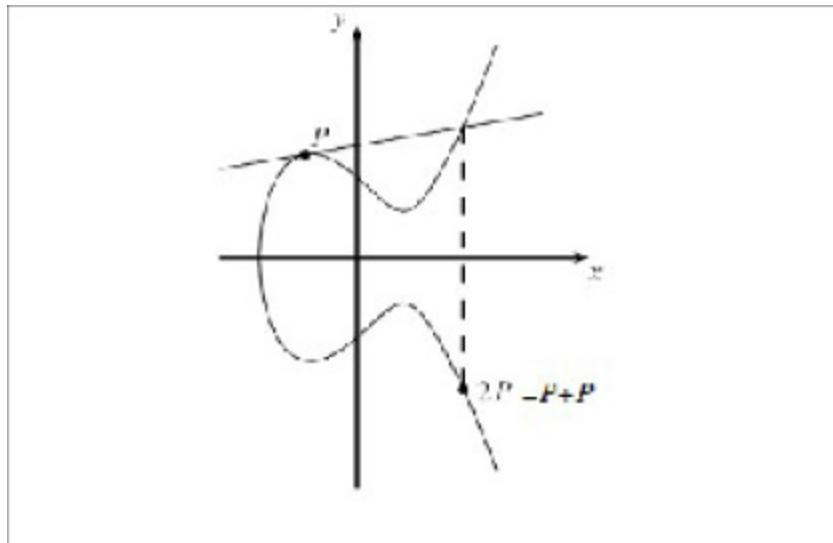


Figure 4.6: Graph representing point doubling

In the case of point doubling, the equation becomes:

$$x_3 = s^2 - x_1 - x_2 \bmod p$$

$$y_3 = s(x_1 - x_3) - y_1 \bmod p$$

$$S = \frac{(y_2 - y_1)}{(x_2 - x_3)} \bmod p$$

Here, S is the slope of the tangent line going through P , which is the line shown at the top in the preceding graph. In the preceding example, the curve is plotted as a simple example, and no solution to the equation is shown.

The following example shows the solutions and point doubling of elliptic curves over finite field F_{23} . The graph on the left-hand side shows the points that satisfy the equation:

$$y^2 = x^3 + 7x + 11$$

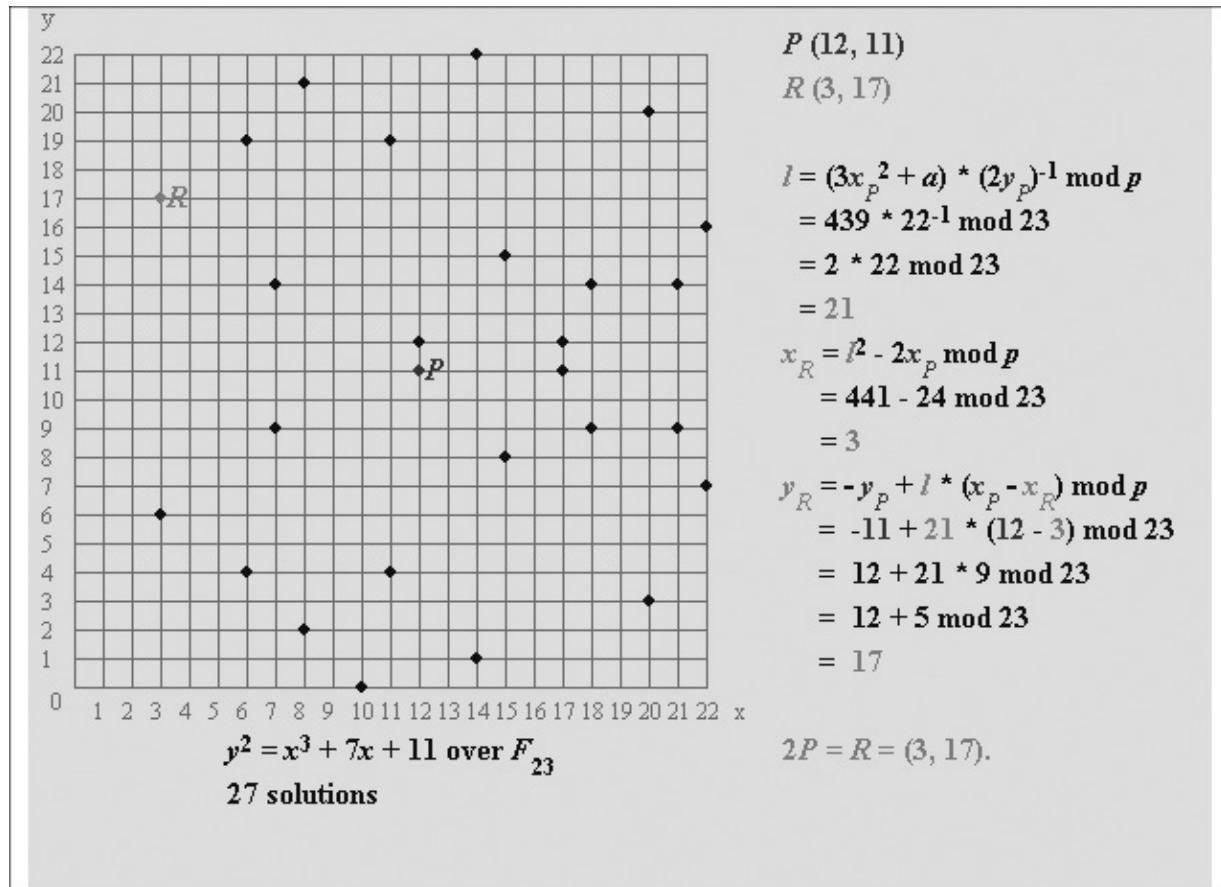


Figure 4.7: Example of point doubling over real numbers

As shown on the right-hand side of the preceding graph, the calculation that finds R after P is added to itself (point doubling). There is no Q shown here because the same point P is used for doubling. Note that in the calculation shown previously on the right-hand side of the graph, l is used to depict the tangent line going through P .

With this, we have covered basics of ECC. In the next section, an introduction to the discrete logarithm problem based on ECC will be presented. The discrete log problem was introduced earlier in this chapter and can be depicted with a simple equation; that is, given y , find x such that $g^x = y$. Fundamentally, it is a one-way function, which is easy to evaluate but extremely hard (computationally) to invert.

Using addition and doubling we can construct another operation, called point multiplication, which we introduce next.

Point multiplication

This operation is also called scalar point multiplication. It is used to multiply points on elliptic curves by a given integer. Let's call this integer d and the point P . We get dP by repeatedly adding P , d times. This operation can be described as follows:

$$P + P + \dots + P = dP, \text{ where } P \text{ is added } d \text{ times.}$$

Any point on the curve can be added multiple times to itself. The result of this operation is always another point on the curve.

The addition operation is not efficient for large values of d . However, point multiplication operation can be made efficient by utilizing **double and add algorithm**, which combine point addition and doubling operations to achieve exponential performance gain.

For example, if using addition only, to get $9P$ we have to do $P + P + P + P + P + P + P + P + P$, which can become infeasible pretty quickly if the number of P s increases. To address this we can use the double and add mechanism where we first convert nine into binary, then starting from the most significant bit, for each bit that is 1 (high) perform double and

addition operation and for each 0 perform only the double operation. We do not perform any operation on the most significant bit. As nine is 1001 in binary, we get for each bit, starting from left to right, P , $2P$, $4P$, $8P+P$. This process produces $9P$ only with three double operations and one addition operation, instead of 9 addition operations.



Further discussion on double and add algorithm is not in the scope of this book. More details on double and add algorithm is available here
https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication.

In this example, point doubling and addition has been used to construct an efficient operation of scalar multiplication.

Now consider that dP results in producing another point on the curve; let's call that point T . We can say that with all these doublings and additions we have computed a point called T . We multiplied a point P on the curve with a number d to compute another point T .

Here's the key idea now: even if we know points P and T , it is computationally infeasible to reconstruct the sequence of all the double and addition operations that we did to figure out the number d . In other words, even if someone knows P and T , it is almost impossible for them to find d . This means that it is a one-way function, and it is the basis of the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**. We describe the discrete logarithm problem in more detail next.

The discrete logarithm problem in ECC

The discrete logarithm problem in ECC is based on the idea that, under certain conditions, all points on an elliptic curve form a cyclic group.

On an elliptic curve, the public key is a random multiple of the generator point, whereas the private key is a randomly chosen integer used to generate the multiple. In other words, a private key is a randomly selected integer, whereas the public key is a point on the curve. The discrete logarithm

problem is used to find the private key (an integer) where that integer falls within all points on the elliptic curve. The following equation shows this concept more precisely.

Consider an elliptic curve E , with two elements P and T . The discrete logarithmic problem is to find the integer d , where $1 \leq d \leq \#E$, such that:

$$P + P + \dots + P = dP = T$$

Here, T is the public key (a point on the curve, (x, y)), and d is the private key. In other words, the public key is a random multiple of the generator P , whereas the private key is the integer that is used to generate the multiple. $\#E$ represents the order of the elliptic curve, which means the number of points that are present in the cyclic group of the elliptic curve. A **cyclic group** is formed by a combination of points on the elliptic curve and the point of infinity.

The initial starting point P is a public parameter, and the public key T is also published, whereas d , the private key, is kept secret. If d is not known, it is impractical to calculate it by only having the knowledge of T and P , thus creating the hard problem on which ECDLP is built.

A key pair is linked with the specific domain parameters of an elliptic curve. Domain parameters include field size, field representation, two elements from the field a and b , two field elements X_g and Y_g , order n of point G , which is calculated as $G = (X_g, Y_g)$, and the cofactor $h = \#E(F_q)/n$. A practical example using OpenSSL will be described later in this section.

Various parameters are recommended and standardized so they can be used as curves with ECC. An example of the SECP256K1 specification is shown here. The following figure is the specification that is used in Bitcoin, which can be accessed here: <http://www.secg.org/sec2-v2.pdf>.

The elliptic curve domain parameters over \mathbb{F}_p associated with a Koblitz curve `secp256k1` are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field \mathbb{F}_p is defined by:

$$\begin{aligned} p &= \text{FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE} \\ &\quad \text{FFFFFFC2F} \\ &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \end{aligned}$$

The curve E : $y^2 = x^3 + ax + b$ over \mathbb{F}_p is defined by:

$$\begin{aligned} a &= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \\ &\quad 00000000 \\ b &= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 \\ &\quad 00000007 \end{aligned}$$

The base point G in compressed form is:

$$\begin{aligned} G &= 02 \text{ 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9} \\ &\quad 59F2815B 16F81798 \end{aligned}$$

and in uncompressed form is:

$$\begin{aligned} G &= 04 \text{ 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9} \\ &\quad 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 \\ &\quad A6855419 9C47D08F FB10D4B8 \end{aligned}$$

Finally the order n of G and the cofactor are:

$$\begin{aligned} n &= \text{FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C} \\ &\quad D0364141 \\ h &= 01 \end{aligned}$$

Figure 4.8: Specification of SECP256K1

An explanation of all of these values in the sextuple follows:

- P is the prime p that specifies the size of the finite field.
- a and b are the coefficients of the elliptic curve equation.
- G is the base point that generates the required subgroup, also known as the generator. The base point can be represented in either compressed or uncompressed form. There is no need to store all points on the curve in a practical implementation. The compressed generator works because the points on the curve can be identified using only the x coordinate and the least significant bit of the y coordinate.
- n is the order of the subgroup.
- h is the cofactor of the subgroup.

We can also use the OpenSSL command line to view these parameters of SECP256K1. This can be seen here:

```
$ openssl ecparam -param_enc explicit -text -noout -name secp256
```

This command will show the output as follows:

```
Field Type: prime-field
Prime:
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
ff:fc:2f
A: 0
B: 7 (0x7)
Generator (uncompressed):
04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
8f:fb:10:d4:b8
Order:
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
36:41:41
Cofactor: 1 (0x1)
```

This output can be readily compared and verified using the SECP256K1 specification shown earlier.



Note that there are different types of curves and they must be chosen carefully in order to ensure cryptographic security guarantees. There are also curves that have been standardized by NIST in the US and published in the FIPS 186 document.

You can find more about safe curves and their selection criteria at <https://safecurves.cr.yp.to>.

In the following section, two examples of using OpenSSL will be shown to help you understand the practical aspects of RSA and ECC.

RSA using OpenSSL

The following example illustrates how RSA public and private key pairs can be generated using the OpenSSL command line.

First, the RSA private key can be generated using OpenSSL as follows:

Private key

```
$ openssl genpkey -algorithm RSA -out privatekey.pem -pkeyopt rs
.....+++++
.....+++++
```

After executing the command, a file named `privatekey.pem` is produced, which contains the generated private key, as follows:

```
$ cat privatekey.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKJOFBzPy2vOd6en
-----END PRIVATE KEY-----
```



It is OK to reveal the private key here, because we are simply using it for our example. However, in production systems, safeguarding the private key is of utmost importance. Make sure that the private key is kept secret. Also, remember that the preceding key is just being used here as an example; do not reuse it.

Public key

As the private key is mathematically linked to the public key, it is also possible to generate or derive the public key from the private key. Using the example of the preceding private key, the public key can be generated as follows:

```
$ openssl rsa -pubout -in privatekey.pem -out publickey.pem
```

```
writing RSA key
```

The public key can be viewed using a file reader or any text viewer:

```
$ cat publickey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznengZP1Bq8w8uC
-----END PUBLIC KEY-----
```

In order to see more details of the various components, such as the modulus, prime numbers that are used in the encryption process, or exponents and coefficients of the generated private key, the following command can be used (only part of the output is shown here as the actual output is very long):

```
$ openssl rsa -text -in privatekey.pem
Private-Key: (1024 bit)
modulus:
00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
ad:71:93:82:18:2a:21:b1:d1
publicExponent: 65537 (0x10001) privateExponent:
00:94:1d:ec:fc:de:25:26:1d:25:d5:39:0e:49:8b:
a7:5a:dc:d3:ea:2b:27:36:44:7b:83:55:ab:63:d1:
fe:ac:6f:71:c7:89:0f:e5:bd:43:c8:4d:d2:bf:10:
4b:24:0e:b1:80:19:5e:f0:95:50:33:39:b7:b0:a2:
6b:24:f1:59:cf:34:f9:34:d3:67:ba:00:b1:6a:2a:
f1:70:5d:66:3f:32:40:3f:76:aa:a5:c4:a3:c4:aa:
53:bf:8a:a0:a9:87:af:c2:05:b5:03:44:77:c4:a4:
19:cc:12:94:f9:b4:ce:da:15:d5:0f:5a:07:c6:ee:
f9:98:36:37:c4:2c:2a:48:01
prime1:
00:d4:61:5d:38:c0:d4:84:8c:4c:bd:b5:82:74:cd:
aa:f2:0c:07:2f:77:f5:46:db:f5:3f:ea:a8:90:19:
a8:c1:79:2a:52:aa:81:04:4a:28:1b:ba:c2:a4:bb:
ec:15:b5:16:20:59:61:28:98:bf:d7:e7:cb:7e:2b:
```

```
4d:5e:30:c6:59
prime2:
00:c3:a3:d4:47:d6:0b:79:bb:26:fd:28:e2:88:e4:
a1:57:80:0a:b8:57:62:ee:de:c3:61:61:39:52:56:
f5:ba:a3:ff:8b:6f:33:37:fc:ad:b7:80:57:28:8b:
3f:29:c1:72:91:01:9e:00:2e:8b:0c:76:72:c9:cc:
9c:9e:d0:c8:39
```

Exploring the public key

Similarly, the public key can be explored using the following commands. Public and private keys are base64-encoded:

```
$ openssl pkey -in publickey.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
-----END PUBLIC KEY-----
Public-Key: (1024 bit) Modulus:
00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
ad:71:93:82:18:2a:21:b1:d1
Exponent: 65537 (0x10001)
```

Now, the public key can be shared openly, and anyone who wants to send you a message can use the public key to encrypt the message and send it to you. You can then use the corresponding private key to decrypt the file.

Encryption and decryption using RSA

In this section, an example will be presented that demonstrates how encryption and decryption operations can be performed using RSA with OpenSSL.

Encryption

Taking the private key we generated in the previous example, the command to encrypt a text file `message.txt` can be constructed, as shown here:

```
$ echo datatoencrypt > message.txt  
$ openssl rsautl -encrypt -inkey pubkey.pem -pubin -in message.txt -out message.rsa
```

This will produce a file named `message.rsa`, which is in binary format. If you open `message.rsa` in the `nano` editor or any other text editor of your choice, it will show some scrambled data, as shown in the following screenshot:

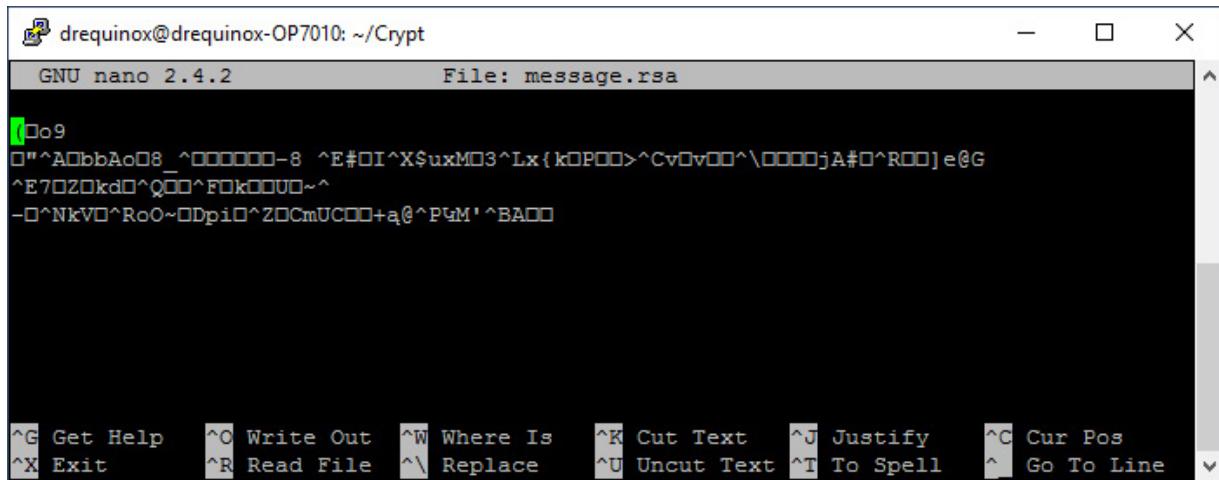


Figure 4.9: `message.rsa` showing scrambled data

Decryption

In order to decrypt the RSA-encrypted file, the following command can be used:

```
$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa -out decrypted.txt
```

Now, if the file is read using `cat`, decrypted plaintext can be seen, as shown here:

```
$ cat message.dec  
datatoencrypt
```

ECC using OpenSSL

OpenSSL provides a very rich library of functions to perform ECC. The following section shows how to use ECC functions in a practical manner in OpenSSL.

ECC private and public key pairs

In this section, first, an example will be presented that demonstrates the creation of a private key using the ECC functions available in the OpenSSL library.

Private key

ECC is based on domain parameters defined by various standards. You can see the list of all available standards defined and the recommended curves available in OpenSSL using the following command (once again, only part of the output is shown here, and it is truncated in the middle):

```
$ openssl ecparam -list_curves  
secp112r1 : SECG/WTLS curve over a 112 bit prime field secp112r2  
prime192v1: NIST/X9.62/SECG curve over a 192 bit prime field  
. .  
. .  
. .  
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field brain
```

In the following example, SECP256k1 is used to demonstrate ECC usage.

Private key generation

In this section, a private key based on SECP256k1 will be generated using the OpenSSL command-line tool.

```
$ openssl ecparam -name secp256k1 -genkey -noout -out ec-private
```

This command will produce a file named `ec-privatekey.pem`, which we can view using the command shown here:

```
$ cat ec-privatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIJHUIm9NZAgfpUrSxUk/iINq1ghM/ewn/RLNreuR52h/oAcGBSuBAAK
-----END EC PRIVATE KEY-----
```

The file named `ec-privatekey.pem` now contains the elliptic curve private key that is generated based on the SECP256K1 curve. In order to generate a public key from a private key, issue the following command:

```
$ openssl ec -in ec-privatekey.pem -pubout -out ec-pubkey.pem
read EC key
writing EC key
```

Reading the file produces the following output, displaying the generated public key:

```
$ cat ec-pubkey.pem
-----BEGIN PUBLIC KEY----- MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE0G33n
dcGXYGxHdzc0Jt1NIaYE0GGChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END PUBLIC KEY-----
```

Now, the `ec-pubkey.pem` file contains the public key derived from `ec-privatekey.pem`. The private key can be further explored using the following command:

```
$ openssl ec -in ec-privatekey.pem -text -noout
read EC key
Private-Key: (256 bit) priv:
00:91:d4:22:6f:4d:64:08:1f:a5:4a:d2:c5:49:3f:
```

```
88:83:6a:d6:08:4c:fd:ec:27:fd:12:cd:ad:eb:91: e7:68:7f
pub:

04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
33:c1:62:a3:9e
ASN1 OID: secp256k1
```

Similarly, the public key can be further explored with the following command:

```
$ openssl ec -in ec-pubkey.pem -pubin -text -noout
read EC key
Private-Key: (256 bit) pub:
04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
33:c1:62:a3:9e
ASN1 OID: secp256k1
```

It is also possible to generate a file with the required parameters—in this case, SECP256K1—and then explore it further to understand the underlying parameters:

```
$ openssl ecparam -name secp256k1 -out secp256k1.pem
$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
```

The file now contains all the SECP256K1 parameters, and it can be analyzed using the following command:

```
$ openssl ecparam -in secp256k1.pem -text -param_enc explicit -r
```

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

```
Field Type: prime-field Prime:  
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  
ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff: ff:fc:2f  
A: 0  
B: 7 (0x7)  
Generator (uncompressed): 04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95  
0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:  
f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:  
0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:  
8f:fb:10:d4:b8 Order:  
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:  
ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:  
36:41:41  
Cofactor: 1 (0x1)
```

The preceding example shows the prime number used and the values of A and B, along with the generator, order, and cofactor of the SECP256K1 curve domain parameters.

With the preceding example, our introduction to public key cryptography from an encryption and decryption perspective is complete.

Another application of hash functions is in digital signatures, where they can be used in combination with asymmetric cryptography. This concept will be explored in detail in the following section.

Digital signatures

Digital signatures provide a means of associating a message with an entity from which the message has originated. Digital signatures are used to provide data origin authentication and non-repudiation.

Digital signatures are used in blockchains, where transactions are digitally signed by senders using their private key, before the sender broadcasts the transaction to the network. This digital signing proves that the sender is the rightful owner of the asset; for example, bitcoins. These transactions are verified again by other nodes on the network to ensure that the funds indeed belong to the node (user) who claims to be the owner. We will discuss these

concepts in more detail in the chapters dedicated to Bitcoin and Ethereum in this book.

Digital signatures are calculated in two steps. As an example, the high-level steps of the RSA digital signature scheme are as follows.

RSA digital signature algorithms

RSA-based digital signature algorithms are calculated using the two steps listed here. Fundamentally, the idea is to first compute the hash of the data and then sign it with the private key:

1. **Calculate the hash value of the data packet.** This will provide the data integrity guarantee, as the hash can be computed at the receiver's end again and matched with the original hash to check whether the data has been modified in transit. Technically, message signing can work without hashing the data first, but that is not considered secure.
2. **Sign the hash value with the signer's private key.** As only the signer has the private key, the authenticity of the signature and the signed data is ensured.

Digital signatures have some important properties, such as authenticity, unforgeability, and non-reusability:

- **Authenticity** means that the digital signatures are verifiable by a receiving party.
- The **unforgeability** property ensures that only the sender of the message can use the signing functionality using the private key. Digital signatures must provide protection against **forgery**. Forgery means an adversary fabricating a valid signature for a message without any access to the legitimate signer's private key. In other words, unforgeability means that no one else can produce the signed message produced by a legitimate sender. This is also called the property of **non-repudiation**.
- **Non-reusability** means that the digital signature cannot be separated from a message and used again for another message. In other words,

the digital signature is firmly bound to the corresponding message and cannot be simply cut from its original message and attached to another.

The operation of a generic digital signature function is shown in the following diagram:

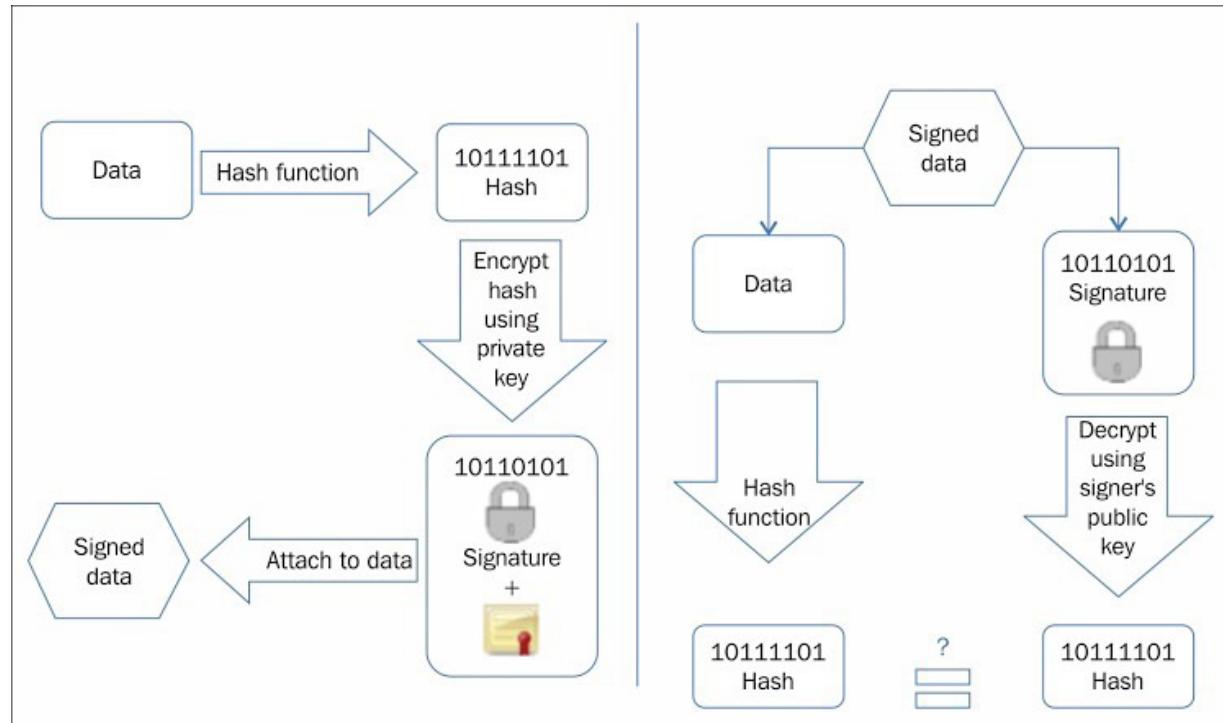


Figure 4.10: Digital signing (left) and verification process (right) (example of RSA digital signatures)

If a sender wants to send an authenticated message to a receiver, there are two methods that can be used: sign then encrypt and encrypt then sign. These two approaches of using digital signatures with encryption are as follows.

Sign then encrypt

With this approach, the sender digitally signs the data using the private key, appends the signature to the data, and then encrypts the data and the digital signature using the receiver's public key. This is considered a more secure scheme compared to the **encrypt then sign** scheme described next.

Encrypt then sign

With this method, the sender encrypts the data using the receiver's public key and then digitally signs the encrypted data.



In practice, a digital certificate that contains the digital signature is issued by a **Certificate Authority (CA)** that associates a public key with an identity.

Various schemes, such as RSA-, DSA-, and ECDSA-based digital signature schemes, are used in practice. RSA is the most commonly used; however, with the traction of ECC, ECDSA-based schemes are also becoming quite popular. This is beneficial in blockchains because ECC provides the same level of security that RSA does, but it uses less space. Also, the generation of keys is much faster in ECC compared to RSA, so it helps with the overall performance of the system. The following table shows that ECC is able to provide the same level of cryptographic strength as an RSA-based system with smaller key sizes:

RSA key size (bits)	Elliptic curve key size (bits)
1,024	160
2,048	224
3,072	256
7,680	384
15,360	521

Now that we understand how the RSA digital signature scheme works, in the next section, we'll introduce ECDSA, which is another popular digital signature scheme.

The elliptic curve digital signature algorithm

ECDSA is a DSA based on elliptic curves. The DSA is a standard for digital signatures. It is based on modular exponentiation and the discrete logarithm problem. It is used on the Bitcoin and Ethereum blockchain platforms to validate messages and provide data integrity services. Now, we'll describe how ECDSA works.

In order to sign and verify using the ECDSA scheme, the first key pair needs to be generated:

1. First, define an elliptic curve E with the following:
 - Modulus P
 - Coefficients a and b
 - Generator point A that forms a cyclic group of prime order q
2. An integer d is chosen randomly so that $0 < d < q$.
3. Calculate public key \mathbf{B} so that $\mathbf{B} = \mathbf{d} \mathbf{A}$.
 - The public key is a sextuple in the form shown here:

$$Kpb = (p, a, b, q, A, B)$$
 - The private key is a randomly chosen integer d in step 2:

$$Kpr = d$$

Now, the signature can be generated using the private and public key.

4. An ephemeral key K_e is chosen, where $0 < K_e < q$. It should be ensured that K_e is truly random and that no two signatures have the same key; otherwise, the private key can be calculated.
5. Another value R is calculated using $R = K_e A$; that is, by multiplying A (the generator point) and the random ephemeral key.
6. Initialize a variable r with the x coordinate value of point R so that $r = xR$.
7. The signature can be calculated as follows:

$$S = (h(m) + dr)K_e^{-1} \bmod q$$

Here, m is the message for which the signature is being computed, and $h(m)$ is the hash of the message m .

8. Signature verification is carried out by following this process:

- Auxiliary value w is calculated as $w = s^{-1} \bmod q$
- Auxiliary value $u1 = w \cdot h(m) \bmod q$
- Auxiliary value $u2 = w \cdot r \bmod q$
- Calculate point $P, P = u1A + u2B$

9. Verification is carried out as follows:

- r, s is accepted as a valid signature if the x -coordinate of point P calculated in *Step 4* has the same value as the signature parameter $r \bmod q$; that is:

$X_p = r \bmod q$ means valid signature

$X_p \neq r \bmod q$ means invalid signature

Various practical examples will now be shown that demonstrate how the RSA digital signature can be generated, used, and verified using OpenSSL.

How to generate a digital signature using OpenSSL

The first step is to generate a hash of the message file:

```
$ openssl dgst -sha256 message.txt
SHA256(message.txt)=
eb96d1f89812bf4967d9fb4ead128c3b787272b7be21dd2529278db1128d559c
```

Both hash generation and signing can be done in a single step, as shown here. Note that `privatekey.pem` was generated in the steps provided previously:

```
$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin
```

Now, let's display the directory showing the relevant files:

```
$ cat signature.bin
```

In order to verify the signature, the following operation can be performed:

```
$ openssl dgst -sha256 -verify publickey.pem -signature signature  
Verified OK
```

Similarly, if some other signature file that is not valid is used, the verification will fail, as shown here:

```
$ openssl dgst -sha256 -verify publickey.pem -signature someother  
Verification Failure
```

Next, an example will be presented that shows how OpenSSL can be used to perform ECDSA-related operations.

ECDSA using OpenSSL

First, the private key is generated using the following commands:

```
$ openssl ecparam -genkey -name secp256k1 -noout -out eccprivatekey.pem  
-----BEGIN EC PRIVATE KEY-----  
MHQCAQEEIMVmyrnEDOs7SYxS/AbXoIwqZqJ+gND9Z2/nQyzcpaPBoAcGBSuBBAK  
-----END EC PRIVATE KEY-----
```

Next, the public key is generated from the private key:

```
$ openssl ec -in eccprivatekey.pem -pubout -out eccpublickey.pem  
read EC key  
writing EC key  
$ cat eccpublickey.pem  
-----BEGIN PUBLIC KEY----- MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEEKKS4  
NRo/k4Y/CZA4pXvlyTgH5LYmQbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==  
-----END PUBLIC KEY-----
```

Now, suppose a file named `testsign.txt` needs to be signed and verified. This can be achieved as follows:

1. Create a test file:

```
$ echo testing > testsign.txt  
  
$ cat testsign.txt  
testing
```

2. Run the following command to generate a signature using a private key for the `testsign.txt` file:

```
$ openssl dgst -ecdsa-with-SHA1 -sign eccprivatekey.pem tes...
```

3. Finally, the command for verification can be run, as shown here:

```
$ openssl dgst -ecdsa-with-SHA1 -verify eccpublickey.pem -s...  
Verified OK
```

A certificate can also be produced by using the private key generated earlier by using the command shown here:

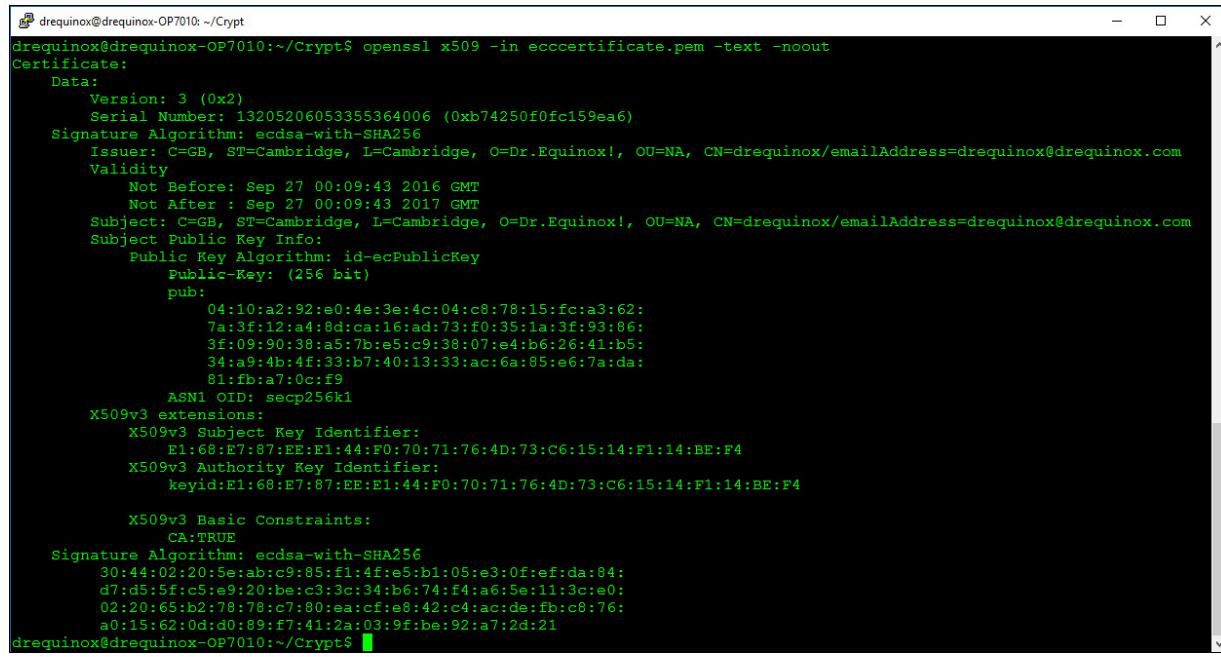
```
$ openssl req -new -key eccprivatekey.pem -x509 -nodes -days 365
```

This command will produce an output similar to the one shown here. Enter the appropriate parameters to generate the certificate:

```
You are about to be asked to enter information that will be included.  
What you are about to enter is what is called a Distinguished Name.  
For some fields there will be a default value. If you enter '.',  
-----  
Country Name (2 letter code) [AU]:GB  
State or Province Name (full name) [Some-State]:Cambridge Locali  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dr.Ec  
Common Name (e.g. server FQDN or YOUR name) []:drequinox Email A
```

The certificate can be explored using the following command:

```
$ openssl x509 -in ecccertificate.pem -text -noout
```



```
drequinox@drequinox-OP7010:~/Crypt$ openssl x509 -in ecccertificate.pem -text -noout
Certificate:
Data:
    Version: 3 (0x2)
    Serial Number: 13205206053355364006 (0xb74250f0fc159ea6)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=GB, ST=Cambridge, L=Cambridge, O=Dr.Equinox!, OU=NA, CN=drequinox/emailAddress=drequinox@drequinox.com
    Validity
        Not Before: Sep 27 00:09:43 2016 GMT
        Not After : Sep 27 00:09:43 2017 GMT
    Subject: C=GB, ST=Cambridge, L=Cambridge, O=Dr.Equinox!, OU=NA, CN=drequinox/emailAddress=drequinox@drequinox.com
    Subject Public Key Info:
        Public Key Algorithm: id-ecPublicKey
            Public-Key: (256 bit)
                pub:
                    04:10:a2:92:e0:4e:3e:4c:04:c8:78:15:fc:a3:62:
                    7a:3f:12:a4:8d:ca:16:ad:73:f0:35:1a:3f:93:86:
                    3f:09:90:38:a5:7b:e5:c9:38:07:e4:b6:26:41:b5:
                    34:a9:4b:4f:33:b7:40:13:33:ac:6a:85:e6:7a:da:
                    81:fb:a7:0c:f9
                    ASN1 OID: secp256k1
    X509v3 extensions:
        X509v3 Subject Key Identifier:
            E1:68:E7:87:EE:E1:44:F0:70:71:76:4D:73:C6:15:14:F1:14:BE:F4
        X509v3 Authority Key Identifier:
            keyid:E1:68:E7:87:EE:E1:44:F0:70:71:76:4D:73:C6:15:14:F1:14:BE:F4
        X509v3 Basic Constraints:
            CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:5e:ab:c9:85:f1:4f:e5:b1:05:e3:0f:ef:da:84:
        d7:d5:5f:c5:e9:20:be:c3:3c:34:b6:74:f4:a6:5e:11:3c:e0:
        02:20:65:b2:78:78:c7:80:ea:cf:e8:42:c4:ac:de:fb:c8:76:
        a0:15:62:0d:d0:89:f7:41:2a:03:9f:be:92:a7:2d:21
drequinox@drequinox-OP7010:~/Crypt$
```

Figure 4.11: An X509 certificate that uses the ECDSA algorithm with SHA-256

In this section, we covered digital signature schemes and some practical examples. Next, we'll present some more advanced cryptography topics.

Cryptographic constructs and blockchain technology

Now, we'll present some advanced topics in cryptography that not only are important on their own, but are also relevant to blockchain technology due to their various applications in this space.

Homomorphic encryption

Usually, public key cryptosystems, such as RSA, are multiplicative homomorphic or additive homomorphic, such as the Paillier cryptosystem, and are called **partially homomorphic** systems. Additive **Partially Homomorphic Encryptions (PHEs)** are suitable for e-voting and banking applications.

Until recently, there has been no system that supported both operations, but in 2009, a **fully homomorphic** system was discovered by Craig Gentry. As these schemes enable the processing of encrypted data without the need for decryption, they have many different potential applications, especially in scenarios where maintaining privacy is required, but data is also mandated to be processed by potentially untrusted parties, for example, cloud computing and online search engines.

Recent development in homomorphic encryption have been very promising, and researchers are actively working to make it efficient and more practical. This is of particular interest in blockchain technology, as described later in this book, as it can solve the problem of confidentiality and privacy in the blockchain.

Signcryption

Signcryption is a public key cryptography primitive that provides all of the functions of a digital signature and encryption. Yuliang Zheng invented signcryption, and it is now an ISO standard, ISO/IEC 29150:2011.

Traditionally, *sign then encrypt* or *encrypt then sign* schemes are used to provide unforgeability, authentication, and non-repudiation, but with signcryption, all the services of digital signatures and encryption such as confidentiality are provided at a cost that is less than that of the *sign then encrypt* scheme.

Signcryption enables $\text{Cost}(\text{signature \& encryption}) \ll \text{Cost}(\text{signature}) + \text{Cost}(\text{Encryption})$ in a single logical step. This means that the cost of applying a digital signature and encrypting a message in the same logical step is lower, compared to a scheme where a message is first signed and

then encrypted in two independent steps. This property makes signcryption an efficient scheme.

Secret sharing

Secret sharing is the mechanism of distributing a secret among a set of entities. All entities within a set get a unique part of the secret after it is split into multiple parts. The secret can be reconstructed by combining all or some parts (a certain number or threshold) of the secret. The individual secret shares/parts, on their own, do not reveal anything about the secret.

This concept can be visualized through the following diagram:

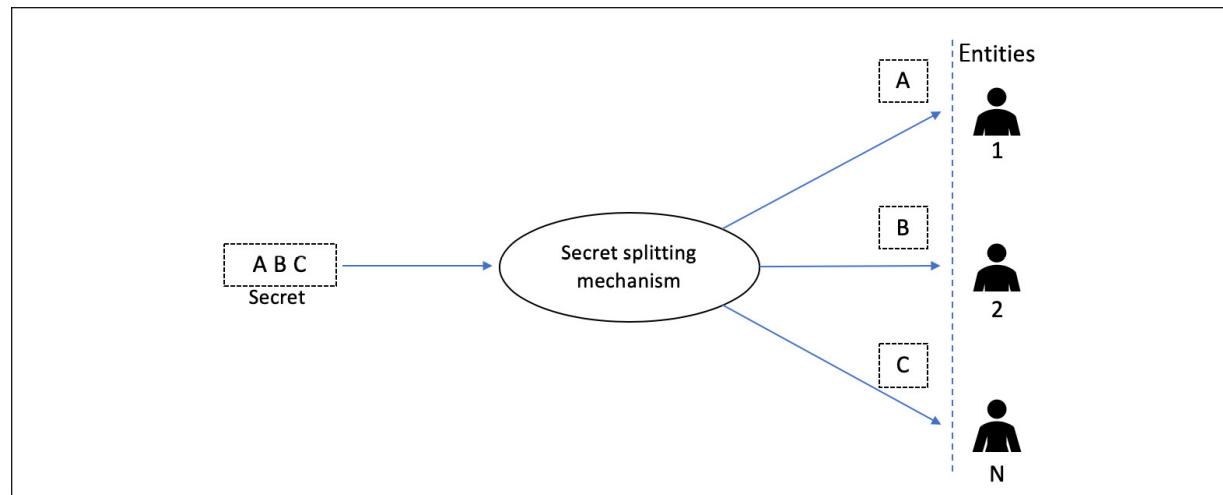


Figure 4.12: Secret sharing scheme

Commitment schemes

Commitment schemes are usually described as a digital cryptographic equivalent of a sealed envelope. A commitment itself does not reveal any information about the actual value inside it. This scheme runs in two phases, namely:

- Commit phase

- Open phase

The commit phase provides two security features; that is, **hiding** and **binding**:

- The hiding property is required to ensure that the recipient party cannot find any information or reveal the committed value before the open phase.
- The binding property ensures that once the sender has committed to a value, that value or message cannot be changed.

The open phase (also called the unveil phase) is where the receiver receives some additional information from the sender. This allows the receiver to establish the value hidden (concealed) by the commitment with a proof that the sender has not modified the value after the commitment, that is, that the sender has not deceived during the commit phase.

Commitment schemes are used for building zero-knowledge protocols and range proof protocols. The key idea behind commitment schemes is that they allow someone to secretly commit to a value with the ability to reveal it later. These schemes prevent parties from changing their committed value later, after they've committed to a value.

A prime example of a commitment scheme is the **Pedersen commitment scheme**.



The original paper on the Pedersen commitment scheme is available here:

https://link.springer.com/content/pdf/10.1007/3-540-46766-1_9.pdf

Pedersen, T.P., 1991, August. Non-interactive and information-theoretic secure verifiable secret sharing. In Annual international cryptology conference (pp. 129-140). Springer, Berlin, Heidelberg.

In the next section, we'll introduce zero-knowledge proofs, which is an exciting subject and a very ripe area for research. Already, various major

blockchains and cryptocurrencies makes use of zero-knowledge proofs to provide privacy. A prime example is **Zcash** (<https://z.cash>).

Zero-knowledge proofs

Zero-Knowledge Proofs (ZKPs) were introduced by Goldwasser, Micali, and Rackoff in 1985. These proofs are used to prove the validity of an assertion without revealing any information whatsoever about the assertion. There are three properties of ZKPs that are required: completeness, soundness, and the zero-knowledge property.

Completeness ensures that if a certain assertion is true, then the verifier will be convinced of this claim by the prover.

The **soundness** property makes sure that if an assertion is false, then no dishonest prover can convince the verifier otherwise.

The **zero-knowledge property**, as the name implies, is the key property of ZKPs, whereby it is ensured that absolutely nothing is revealed about the assertion except whether it is true or false.



Zero-knowledge proofs are probabilistic instead of deterministic. The probabilistic nature of ZKPs is because these protocols require several rounds to achieve a higher level of assurance gradually, with each round, which allows the verifier to accept that the prover indeed knows the secret.

In literature, usually, an analogy known as "Ali Baba's Cave" is used to explain zero-knowledge proofs.



This analogy was first introduced by Jean-Jacques Quisquater et al. in their paper, which is available at:

https://link.springer.com/content/pdf/10.1007/0-387-34805-0_60.pdf

Quisquater, J.J., Quisquater, M., Quisquater, M., Quisquater, M., Guillou, L., Guillou, M.A., Guillou, G., Guillou, A., Guillou, G. and Guillou, S.,

1989, August. How to explain zero-knowledge protocols to your children. In Conference on the Theory and Application of Cryptology (pp. 628-631). Springer, New York, NY.

The following diagram shows a variant of this analogy of how the zero-knowledge protocol works. It shows two characters called **Peggy** and **Victor** whose roles are Prover and Verifier, respectively. Peggy knows a secret magic word to open the door in a cave, which is shaped like a ring. It has a split entrance and has a **magic door** in the middle of the ring that blocks the other side. We have also labeled the left and right entrances as **A** and **B**. Victor wants to know the secret, but Peggy does not want to reveal it. However, she would like to prove to Victor that she does know the secret:

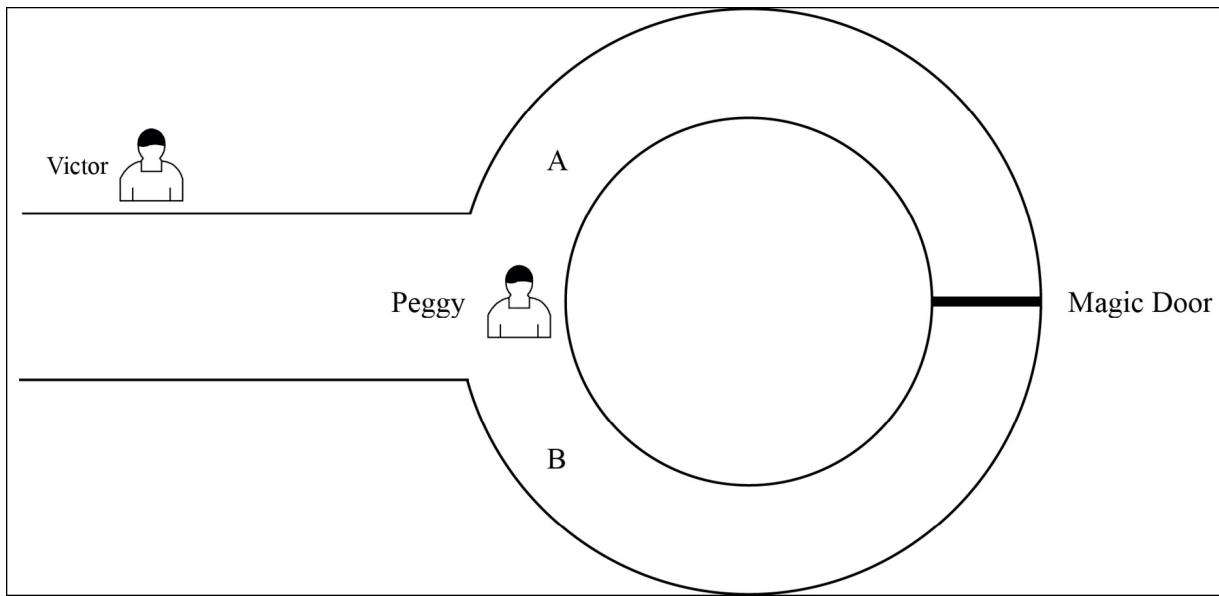


Figure 4.13: Analogy of the zero-knowledge mechanism

Now, there are several steps that are taken by both Peggy and Victor to reach a conclusion regarding whether Peggy knows the secret or not:

1. First, Victor waits outside the main cave entrance and Peggy goes in the cave.
2. Peggy randomly chooses either the **A** or **B** entrance to the cave.

3. Now, Victor enters the cave and shouts either **A** or **B** randomly, asking Peggy to come out of the exit he named.
4. Victor records which exit Peggy comes out from.

Now, suppose Victor asked Peggy to come out from exit A and she came out from exit B. Victor then knows that Peggy does not know the secret. If Peggy comes out of exit A, then there is a 50% chance that she does know the secret, but this also means that she may have got lucky and chose A to enter in the first place, and now has just returned without needing to go through the magic door at all. Now, if this routine is performed several times, and given that Victor is choosing A or B at random, with each run (round) of this routine (protocol), the chances of Peggy getting lucky diminish. If Peggy repeatedly manages to emerge from the entrance that Victor has named, then it is highly probable that Peggy does know the secret to open the magic door.



Remember, we said earlier that zero-knowledge protocols are probabilistic. Peggy and Victor repeatedly performing this routine proves knowledge of the secret with high probability, as the probability of guessing or getting lucky reduces to the point of being negligible after n number of rounds.

Note that during this process, Peggy has not revealed the secret at all, but still managed to convince Victor with high probability that she does know the secret.

A ZKP comprises the following phases:

1. **Witness phase:** In this phase, the prover sends proof of the statement and sends it to the verifier.
2. **Challenge phase:** In this phase, the verifier chooses a question (challenge) and sends it to the prover.
3. **Response phase:** In this phase, the prover generates an answer and sends it as a response to the verifier. The verifier then checks the answer to ascertain whether the prover really knows the statement.

This scheme can also be visualized using the following diagram, which shows how the zero-knowledge protocol generally works and what steps the prover and verifier take in order to successfully run the protocol:

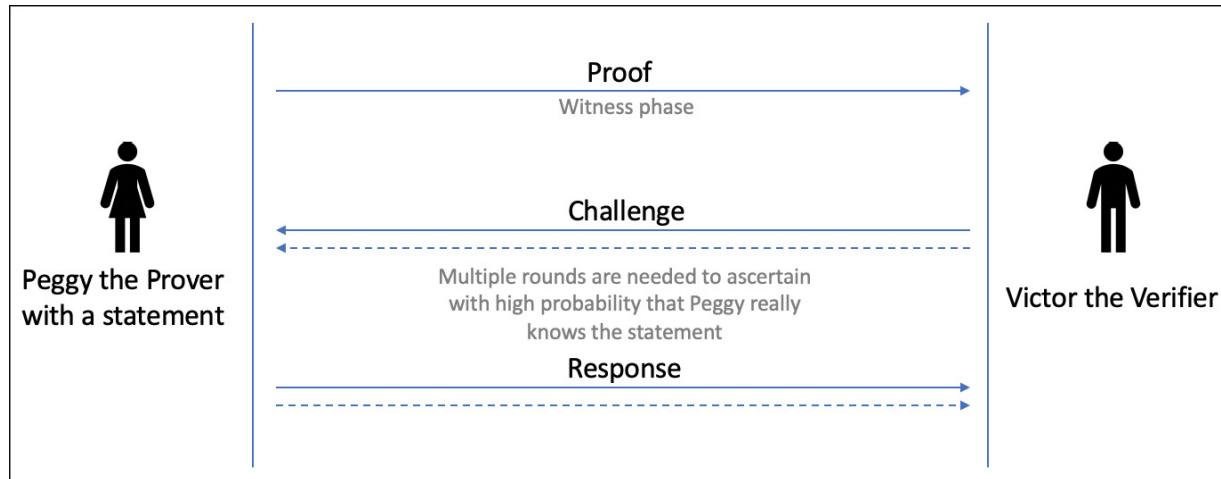


Figure 4.14: A zero-knowledge proof scheme

Even though zero-knowledge is not a new concept, these protocols have sparked a special interest among researchers in the blockchain space due to their privacy properties, which are very much desirable in finance and many other fields, including law and medicine. A prime example of the successful implementation of a zero-knowledge proof mechanism is the Zcash cryptocurrency. In Zcash, the **zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK)** is implemented to provide anonymity and confidentiality.

Applications of ZKP include proof of ownership, that is, proof that the prover is the owner of some secret or a private key without revealing the nature of it. ZKPs can also be used to prove that the prover is a member of some organization or group, without revealing their identity. Another use case could be proof of age, where the prover proves that they are, for example, older than 25 years, but do not want to reveal their exact age. Another application could be where citizens can pay taxes without revealing their income.

Zero-knowledge protocols are usually interactive as they require repeated interactions between the prover and the verifier, but there are protocols that

do not require any interaction between a prover and a verifier. These type of ZKPs are called non-interactive types of proofs. A prominent example is the zk-SNARK. Before the introduction of zk-SNARKs, ZKPs were considered not very practical because of the complexity that arises from the requirements of repeated interaction between the prover and the verifier and a large proof size.

zk-SNARKs

The **zk-SNARK** is a variant of ZKPs that is more efficient and simpler to use and implement.

There has been some influential work published on non-interactive zero-knowledge proofs over the years. The most prominent of these, and the one that proposed the first universal design, is provided by the paper by Eli Ben-Sasson et al., which is available at:

<https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-ben-sasson.pdf>

Ben-Sasson, E., Chiesa, A., Tromer, E. and Virza, M., 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In 23rd {USENIX} Security Symposium ({USENIX} Security 14) (pp. 781-796).

Other previous notable works that contributed to the development of zk-SNARKs include:



Gennaro, R., Gentry, C., Parno, B. and Raykova, M., 2013, May. Quadratic span programs and succinct NIZKs without PCPs. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 626-645). Springer, Berlin, Heidelberg.

https://link.springer.com/content/pdf/10.1007/978-3-642-38348-9_37.pdf

and:

Groth, J. and Sahai, A., 2008, April. Efficient non-interactive proof systems for bilinear groups. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 415-432). Springer, Berlin, Heidelberg.

https://link.springer.com/content/pdf/10.1007/978-3-540-78967-3_24.pdf

Zk-SNARKs have several properties, which are described here:

- **Zero knowledge (zk):** This property allows a prover to convince a verifier that an assertion (statement) is true without revealing any information about it.



A statement in this context is any computer program that terminates and does not take too long to run, that is, not too many cycles.

- **Succinct (S):** This property allows for a small proof that is very quick to verify and is succinct.
- **Non-interactive (N):** This property allows the prover to prove a statement without any interaction with the verifier.
- **ARguments of Knowledge (ARKs):** These are the arguments to convince the verifier that the prover's assertion is true. Remember that we discussed soundness and completeness earlier. In the case of ARKs, the prover has a computational soundness property, meaning that the prover is computationally bounded, and it is computationally infeasible for it to cheat the verifier.

Zk-SNARKs have been implemented in different blockchains. The most prominent example is Zcash, which uses it for its shielded transactions feature to provide confidentiality and anonymity. Support for cryptographic primitives required for zk-SNARKs has also been introduced in the Ethereum blockchain with the Byzantium release.

Zk-SNARK proofs are generated by following a number of different steps. Generating SNARKs for a program (statement) is not a simple process as it requires the program to be converted into a circuit with a very specific format. This specific form is called the **Quadratic Arithmetic Program (QAP)**. The process of proof generation is as follows:

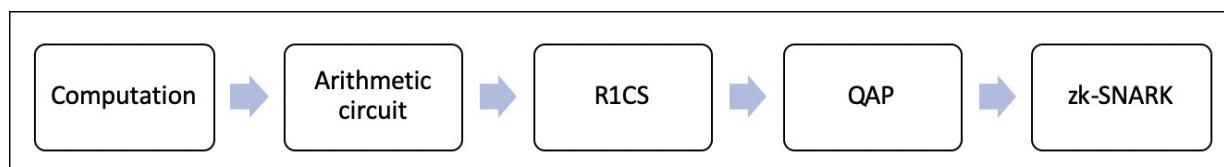


Figure 4.15: zk-SNARK construction

- **Arithmetic circuit:** The first step in zk-SNARK construction is to convert a logical step of a **computation** into the smallest possible units comprised of basic arithmetic operations. Arithmetic operations include addition, subtraction, multiplication, and division. In the arithmetic circuit, the computation is presented as wires and gates, representing the flow of inputs and arithmetic operation performed on the inputs.
- **R1CS:** R1CS is the abbreviation of **Rank 1 Constraint System (R1CS)**. Basically, this system is a set of constraints that allows all the steps in the arithmetic circuit to be verified, and also confirms that, at the end of the process, the output is as expected.
- **QAP:** The prover makes use of QAPs to construct proof for the statement. In R1CS, the verifier has to check many different constraints, but with a QAP representation of the circuit, all the different constraints can be bundled into only a single constraint. Finally, QAP is used in the zk-SNARK protocol to prove the assertion through the prover and verifier.

Zk-SNARKs are not a silver bullet to privacy problems. Instead, like many other technologies, there are pros and cons.

The biggest criticism of zk-SNARKs is the initial trusted setup. This trusted setup can be influenced and compromised. However, if done correctly, it does work, but there is, however, always a chance that the initial setup was compromised and no one will ever be able to find out. This is the issue that has been addressed in zk-STARKs.

zk-STARKs

Zero-knowledge Scalable Transparent ARguments of Knowledge (zk-STARKs) are a new type of ZKP that has addressed several limitations in zk-SNARKs.

This scheme was designed by Eli-Ben Sasson et al.



The original paper on the subject is available here:

<https://eprint.iacr.org/2018/046.pdf>

Ben-Sasson, E., Bentov, I., Horesh, Y. and Riabzev, M., 2018. Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptology ePrint Archive, 2018, p.46.

The key differences between the zk-STARK and zk-SNARK schemes are listed in the following table:

Properties	zk-SNARKs	zk-STARKs
Scalability	Less scalable.	More scalable.
Initial trusted setup	Required.	Not required, has a publicly verifiable mechanism.
Post-quantum resistant	Not resistant to attacks from quantum computers.	Yes, resistant to attacks from quantum computers.
Construction techniques	Zk-SNARKs rely on elliptic curves and pairings.	Zk-STARKs are based on hash functions and concepts from information theory.



What is "post-quantum resistant"?

Quantum computing is a concept that uses quantum mechanics to solve problems that take an extremely long time to solve on conventional computers. When large-scale quantum computers are built they can easily compromise the security of most existing cryptosystems. **Post-quantum resistance** refers to the idea of building cryptographic algorithms that are resistant to attacks by powerful quantum computers.

Further discussion on this topic is not in the scope of this book, and more information on the subject can be found at the following Wikipedia page:

https://en.wikipedia.org/wiki/Quantum_computing#Cryptography

Even though this scheme is more efficient, post-quantum resistant, scalable, and transparent, the biggest limitation is its size of proof, which is a few hundred kilobytes compared to zk-SNARKs' 288 bytes. This is seen as a problem in public blockchains where large proofs may pose a problem to scalability and performance.

Zero-knowledge range proofs—ZKRPs

ZKRPs are used to prove that a number is between a certain range. This can be useful when, for example, someone does not want to reveal their salary but is willing to only reveal the range between which their salary lies. Range proofs can also be used for age checks without requiring the prover to reveal their exact age.

In this section, we have covered the very interesting topic of ZKPs and their relevant techniques and developments. ZKPs is a very active area of research, especially in the context of blockchains, where this technology can provide much needed privacy and confidentiality services on public blockchain networks. Next, we'll introduce different types of digital signatures.

Different types of digital signatures

We previously covered digital signatures, but that was an introduction to standard digital signatures. Now, we'll provide an overview of different types of digital signatures and their relevance and applications in blockchain technology.

Blind signatures

Blind signatures were introduced by David Chaum in 1982. They are based on public key digital signature schemes, such as RSA. The key idea behind blind signatures is to get the message signed by the signer, without actually revealing the message. This is achieved by disguising or blinding the message before signing it, hence the name **blind signatures**.

This blind signature can then be verified against the original message, just like a normal digital signature. Blind signatures were introduced as a mechanism to allow the development of digital cash schemes. Specifically, a blind signature can provide unlinkability and anonymity services in a distributed system, such as a blockchain.



The original paper from David Chaum on blind signatures is available at <http://blog.koehtopp.de/uploads/Chaum.BlindSigForPayment.1982.PDF>.

Chaum, David. "Blind signatures for untraceable payments." In Advances in cryptology, pp. 199-203. Springer, Boston, MA, 1983.

The blind signature scheme is also used in building Blindcoin, which is used in the Mixcoin protocol for Bitcoin to hide user addresses from the mixing service. Mixcoin protocol allows anonymous payments in the Bitcoin network, but the user addresses are still visible to the mixing service. Blindcoin is a protocol that addresses this limitation.



More information on this scheme is available in the following paper:

Valenta, L. and Rowan, B., 2015, January. Blindcoin: Blinded, accountable mixes for bitcoin. In International Conference on Financial Cryptography and Data Security (pp. 112-126). Springer, Berlin, Heidelberg.

Multisignatures

In this scheme, a group of entities signs a single message. In other words, multiple unique keys held by their respective owners are used to sign a single message. This scheme was introduced in 1983 by Itakura et al. in their paper *A Public-key Cryptosystem Suitable for Digital Multisignatures*, vol. 71, *Nec Research & Development* (1983), pp. 474-480. Multisignatures are also sometimes called multiparty signatures in literature.

In blockchain implementations, multisignatures provide the ability to allow transactions to be signed by multiple users, which results in increased

security. This is also called multi-sig and has been implemented in Bitcoin. These schemes can be used in such a way that the requirement of a number of signatures can be set in order to authorize transactions. For example, a 1-of-2 multisignature can represent a joint account where either one of the joint account holders is required to authorize a transaction by signing it. As another variation, for example, a 2-of-2 multisignature can be used in a scenario where both joint account holders' signatures are required to sign the transaction. This concept is also generalized as m of n signatures, where m is the minimum number of required signatures and n is the total number of signatures. This concept can be visualized with the following diagram:

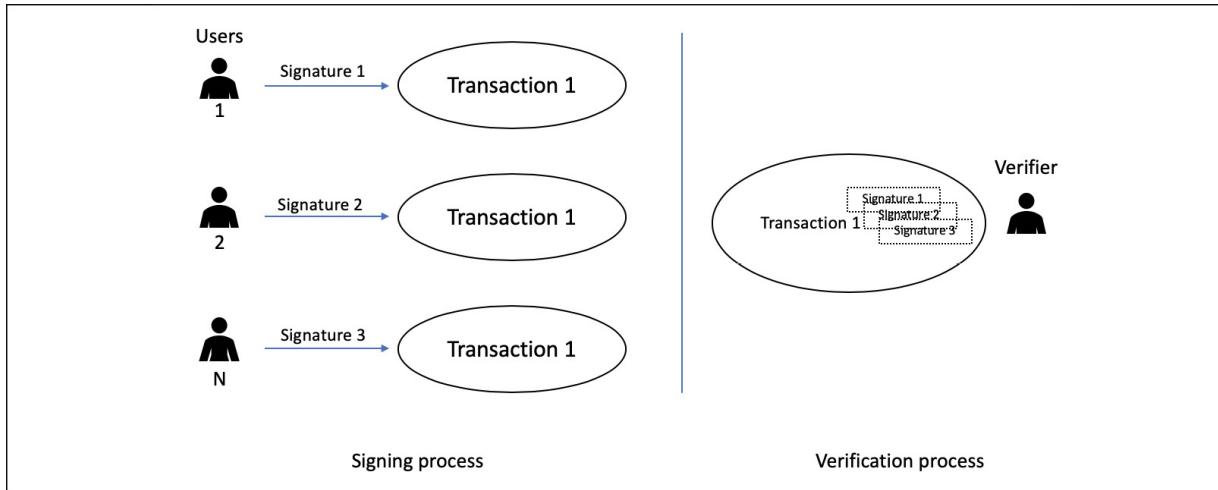


Figure 4.16: Multi signature scheme

The preceding diagram shows the signing process on the left-hand side, where m is the number of different users, and holding m unique signatures signs a single transaction. When the validator or verifier receives it, all the signatures in it need to be individually verified.

The Openchain and Multichain blockchains make use of multisignature schemes.



More information on Openchain is available at
<https://docs.openchain.org/en/latest/general/overview.html>.

More information regarding Multichain's multisignature scheme is available at:

<https://www.multichain.com/developers/multisignature-transactions/>.

Threshold signatures

This scheme does not rely on users to sign the message with their own unique keys; instead, it requires only one public key and one private key, and results in only one digital signature. In multisignature, the resultant message contains digital signatures from all signers and requires verification individually by the verification party, but in threshold signatures, the verifier has to verify only one digital signature. The key idea in the scheme is to split the private key into multiple parts, and each signer keeps their own share of the private key. The signing process requires each user to use their respective share of the private key to sign the message. The communication between signers is governed by a specific communication protocol. In contrast with multisignatures, the threshold signatures result in a smaller transaction and are quicker to verify. A disadvantage, however, is that in order for threshold signatures to work, all signers must remain online, whereas in multisignatures, the signature can be provided asynchronously; that is, users can provide signatures whenever they are available. From another angle, there could be a scenario in multisignatures where a user can withhold their signature maliciously, which could be disadvantageous. Threshold signatures can also be used to provide anonymity services in a blockchain network:

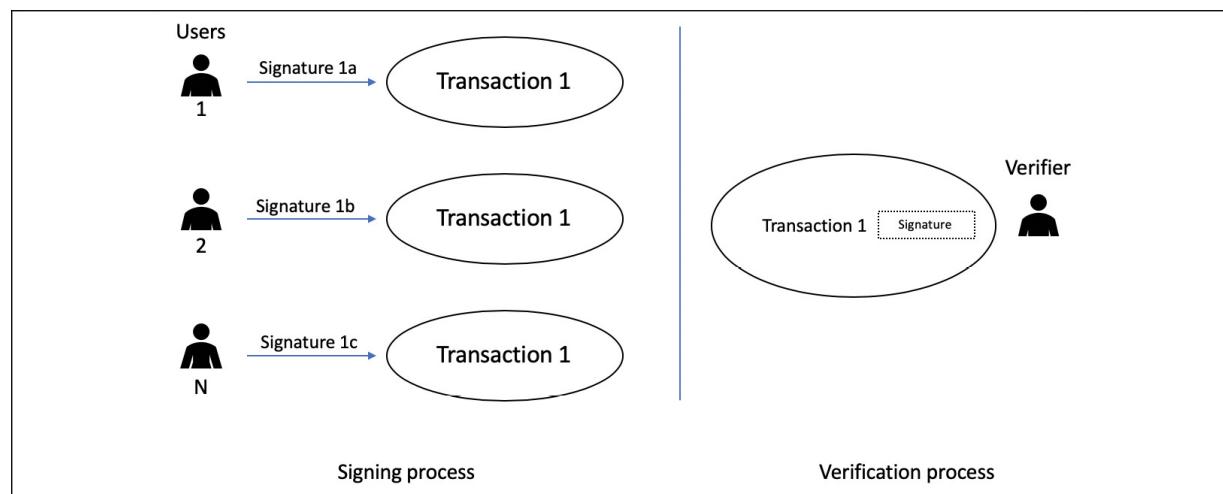


Figure 4.17: Threshold signature scheme

The preceding diagram shows the signing process on the left-hand side, where m number of different users, holding different parts (shares) of a digital signature, sign a single transaction. When the validator or verifier receives it, only one signature needs to be verified.

Aggregate signatures

Aggregate signatures are used to reduce the size of digital signatures. This scheme is particularly useful in scenarios where multiple digital signatures are in use. The core idea is to aggregate multiple signatures into a single signature, without increasing the size of the signature of a single message. It is simply a type of digital signature that supports aggregation. The small aggregate signature is enough to provide verification to the verifier that all users did sign their original messages. Aggregate signatures are commonly used to reduce the size of messages in network and security protocols. For example, the size of digital certificate chains in **Public Key Infrastructure (PKI)** can be reduced significantly by compressing all signatures in the chain into a single signature. **Boneh–Lynn–Shacham (BLS)** aggregate signatures is a popular example of the aggregate signature. BLS has also been used in various blockchains, and especially in Ethereum 2.0.

More information regarding aggregate BLS signatures is available in the paper here:

<https://crypto.stanford.edu/~dabo/papers/aggrep.pdf>

Boneh, D., Gentry, C., Lynn, B. and Shacham, H., Aggregate and Verifiably Encrypted Signatures from Bilinear Maps.



More information on BLS and specifically its usage in blockchains to reduce the size of the blockchain is available in an excellent paper here:

<https://eprint.iacr.org/2018/483.pdf>.

Boneh, D., Drijvers, M. and Neven, G., 2018, December. Compact multi-signatures for smaller blockchains. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 435-464). Springer, Cham.

Ring signatures

Ring signatures were introduced in 2001 by Ron Rivest, Adi Shamir, and Yael Tauman.



The original paper is available at:

https://link.springer.com/content/pdf/10.1007/3-540-45682-1_32.pdf

Rivest, R.L., Shamir, A. and Tauman, Y., 2001, December. How to leak a secret. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 552-565). Springer, Berlin, Heidelberg.

Ring signature schemes allow a mechanism where any member of a group of signers can sign a message on behalf of the entire group. The key requirement here is that the identity of the actual signer who signed the message must remain unknown (computationally infeasible to determine) to an outside observer. It looks equally likely that anyone of the trusted group of signers could have signed the message, but it is not possible to figure out who actually signed the message. Each member of the ring group keeps a public key and a private key. Ring signatures can be used to provide anonymity services. This scheme is used in CryptoNote and Monero.

In this section, we covered different types of digital signatures, their operation, and their relevance to blockchains. In the next section, we will introduce encoding schemes and some relevant examples.

Encoding schemes

Other than cryptographic primitives, binary-to-text **encoding schemes** are also used in various scenarios. The most common use is to convert binary data into text so that it can either be processed, saved, or transmitted via a protocol that does not support the processing of binary data. For example, images can be stored in a database encoded in base64, which allows a text field to be able to store a picture. Another encoding, named base58, was

popularized by its use in Bitcoin and is addressed in detail in *Chapter 6, Introducing Bitcoin*.

Base64

The base64 encoding scheme is used to encode binary data into printable characters.

We can do a quick experiment using the OpenSSL command line:

```
$ openssl rand 16 -base64  
4ULD5sJtGeoSnogIniHp7g==
```

The preceding command has generated a random sequence of 16 bits and then, using the `-base64` switch, it has converted that into a base64 text string. Base64 converts from 8 bits to a 6-bit ASCII representation. This is useful for storage and transmission, especially in cases where binary data handling could lead to incompatibility between systems. This mechanism is a flexible way to represent binary data as ASCII, which can be easily and universally stored and transmitted.

Base58

The base58 scheme was first introduced with Bitcoin and is used to encode integers into alphanumeric strings. The key idea behind this encoding scheme is to avoid non-alphanumeric characters and also those characters that look similar and could lead to ambiguity; for example, a lower-case L (l) may look like the number one (1). This feature is especially useful because Bitcoin addresses must not have any confusion about the character representation; otherwise, it could lead to wrongly sending bitcoins to some non-existent or incorrect address, which is clearly a financial loss. This ingenious encoding scheme avoids this type of situation by ignoring similar looking characters.

We will explore base58 and its role in generating Bitcoin addresses in detail in *Chapter 6, Introducing Bitcoin*.



More information on the specifics of base58 in the context of Bitcoin can be found at:

https://en.bitcoin.it/wiki/Base58Check_encoding

Cryptography is a vast field, and this section has introduced some of the main concepts that are essential to understanding cryptography in general, and specifically from a blockchain and cryptocurrency point of view.

In the next section, we'll introduce some applications of hash functions, which we introduced in *Chapter 3, Symmetric Cryptography*. Hash functions are extremely important and we will explore how these constructs are used, in blockchains and generally, to build useful constructs that provide the foundation for building blockchain networks.

Applications of cryptographic hash functions

There are various constructs that have been built using basic cryptographic parameters to solve different problems in computing. These constructs are also used in blockchains to provide various protocol-specific services. For example, hash functions are used to build Merkle trees, which are used to efficiently and securely verify large amounts of data in distributed systems. Some other applications of hash functions in blockchains are to provide a number of security services.

These services are listed here:

- Hash functions are used in cryptographic puzzles such as the **Proof of Work (PoW)** mechanism in Bitcoin. Bitcoin's PoW makes use of the SHA-256 cryptographic hash function.
- The generation of addresses in blockchains. For example, in Ethereum, blockchain accounts are represented as addresses. These addresses are obtained by hashing the public key with the Keccak-256 hash algorithm and then using the last 20 bytes of this hashed value.
- Message digests in digital signatures.

- The creation of Merkle trees to guarantee the integrity of transaction structure in the blockchain. Specifically, this structure is used to quickly verify whether a transaction is included in a block or not. However, note that Merkle trees on their own is not a new idea; it has just been made more popular with the advent of blockchain technology.

Merkle trees are the core building blocks of all blockchains; for example, Bitcoin and Ethereum. We will explore Merkle trees in detail now.

Merkle trees

The concept of Merkle trees was introduced by Ralph Merkle. A diagram of a Merkle tree is shown here. **Merkle trees** enable the secure and efficient verification of large datasets:

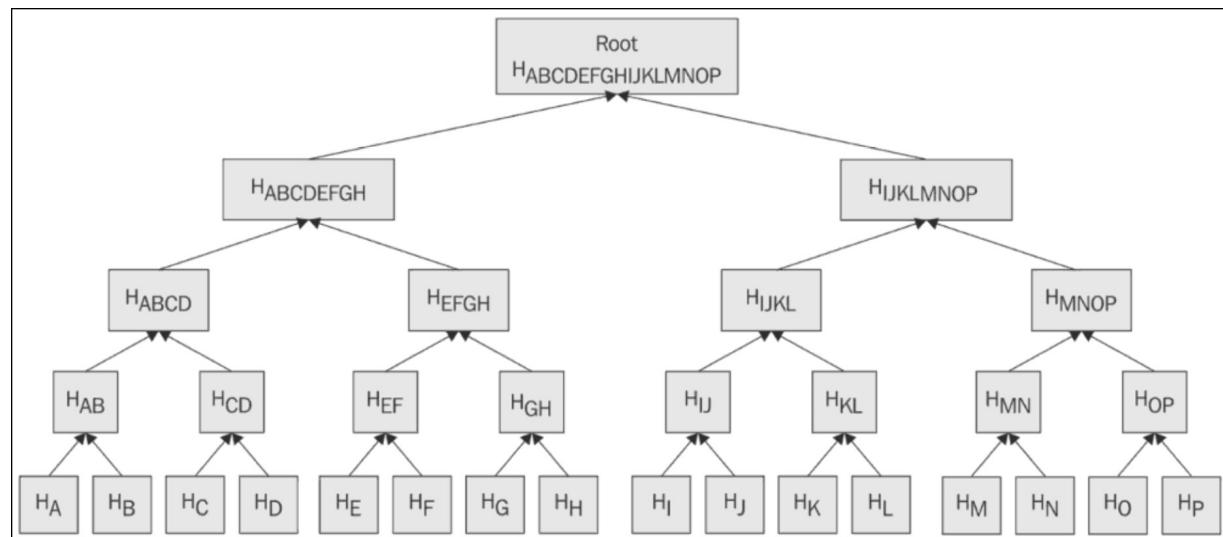


Figure 4.18: A Merkle tree

A Merkle tree is a binary tree in which the inputs are first placed at the leaves (nodes with no children), and then the values of pairs of child nodes are hashed together to produce a value for the parent node (internal node), until a single hash value known as a **Merkle root** is achieved. This structure helps to quickly verify the integrity of the entire tree (entire dataset), but just by verifying the Merkle root on top of the Merkle tree, because if any change occurs in any of the hashes in the tree, the Merkle root will also

change. This is the reason why the integrity of the system can be verified quickly by just looking at the Merkle root. Another advantage of Merkle trees is that there is no requirement of storing large amounts of data, only the hashes of the data, which are fixed-length digests of the large dataset. Due to this property, the storage and management of Merkle trees is easy and efficient as it takes a very small amount of space for storage. Also, due to the fact that the tree is storage efficient, the relevant proofs for integrity are also smaller in size and quick to transmit over the network, thus making them bandwidth efficient over the network.

We will explain in detail how Merkle trees are fundamental for the consistency and integrity of a blockchain and its transactions in the chapters on Ethereum and Bitcoin.

Patricia trees

To understand Patricia trees, you will need to be introduced to the concept of a **trie**.

A trie, or a **digital tree**, is an ordered tree data structure used to store a dataset.

The **Practical Algorithm to Retrieve Information Coded in Alphanumeric (Patricia)** tree, also known as a **Radix tree**, is a compact representation of a trie in which a node that is the only child of a parent is merged with its parent.

A **Merkle-Patricia tree**, based on the definitions of Patricia and Merkle, is a tree that has a root node that contains the hash value of the entire data structure. Merkle-Patricia trees are used in the Ethereum blockchain.

Distributed hash tables

A hash table is a data structure that is used to map keys to values. Internally, a hash function is used to calculate an index into an array of buckets from which the required value can be found. Buckets have records stored in them using a hash key and are organized into a particular order.

With the definition provided earlier in mind, we can think of a **Distributed Hash Table (DHT)** as a data structure where data is spread across various nodes, and nodes are equivalent to buckets in a peer-to-peer network.

The following diagram shows how a DHT works:

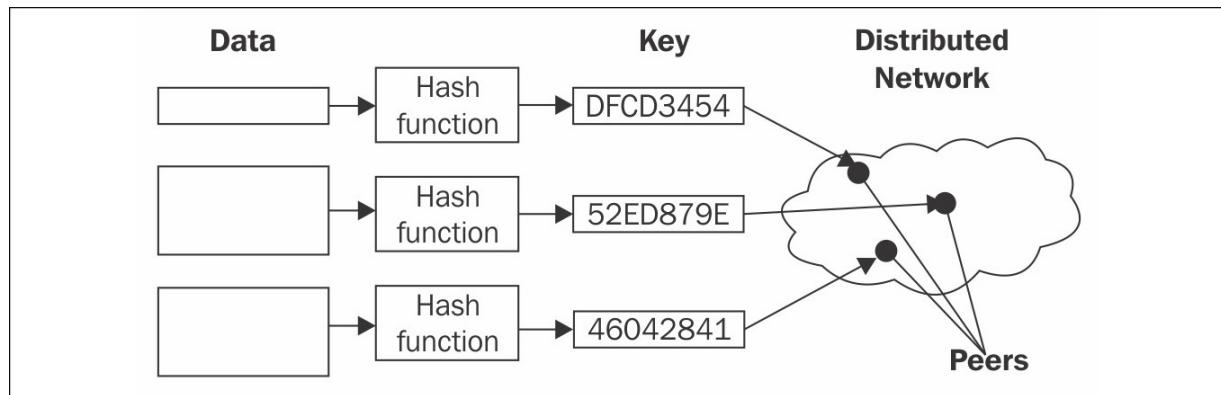


Figure 4.19: DHT

In the preceding diagram, data is passed through a hash function, which then generates a compact key. This key is then linked with the data (values) on the peer-to-peer network. When users on the network request the data (via the filename), the filename can be hashed again to produce the same key, and any node on the network can then be requested to find the corresponding data. A DHT provides decentralization, fault tolerance, and scalability.

In this section, we covered various applications of cryptographic hash functions. Hash functions are of particular importance in blockchains, as they are key to constructing Merkle trees, which are used in blockchains for the efficient and fast verification of large datasets. With this section, we have completed our introduction to the applications of hash functions.

Summary

This chapter started with an introduction to some basic mathematics concepts and asymmetric key cryptography. We discussed various

constructs such as RSA and ECC. We also performed some experiments using OpenSSL to see how theoretical concepts can be implemented practically. After this, hash functions were discussed in detail, along with their properties and usage. Next, we covered concepts, such as Merkle trees, that are used extensively in blockchains and in fact are at its core. Other concepts such as Patricia trees and hash tables were also introduced. We also looked at some advanced and modern concepts such as zero-knowledge proofs and relevant constructions, along with different types of digital signatures.

In the next chapter, we will explore the captivating world of distributed consensus, which is central to the integrity of any blockchain and is a very hot area of research.

5

Consensus Algorithms

Consensus is a fundamental problem in distributed systems. Since the 1970s this problem has been researched in the context of distributed systems, but recently, with the advent of blockchain technology, a renewed interest has arisen in developing distributed consensus algorithms that are suitable for blockchain networks. In this chapter, we will explore the underlying techniques behind distributed consensus algorithms, their inner workings, and new algorithms that have been specifically developed for blockchain networks.

In addition, we will introduce various well-known algorithms in a traditional distributed systems arena that can also be implemented in blockchain networks with some modifications, such as Paxos, Raft, and PBFT. We will also explore other mechanisms that have been introduced specifically for blockchain networks such as **Proof of Work (PoW)**, **Proof of Stake (PoS)**, and modified versions of traditional consensus such as **Istanbul Byzantine Fault Tolerant (IBFT)**, which is a modified, **blockchained** version of the **Practical Byzantine Fault Tolerant (PBFT)** algorithm, suitable for a blockchain network. Along the way, we'll cover the following topics:

- Introducing the consensus problem
- Analysis and design
- Classification
- Algorithms
- Choosing an algorithm

Before we delve into specific algorithms, we first need to understand some fundamental concepts and an overview of the consensus problem.

Introducing the consensus problem

The distributed consensus problem has been studied extensively in distributed systems research since the late 1970s. Distributed systems are classified into two main categories, namely **message passing** and **shared memory**. In the context of blockchain, we are concerned with the message passing type of distributed systems, where participants on the network communicate with each other via passing messages to each other.

Blockchain is a distributed system that relies upon a consensus mechanism, which ensures the safety and liveness of the blockchain network.

In the past decade, the rapid evolution of blockchain technology has been observed. Also, with this tremendous growth, research regarding distributed consensus has grown significantly. Researchers from industry and academia are especially interested in researching novel methods of consensus. A common research area is to convert traditional (classical) distributed consensus mechanisms into their blockchain variants that are suitable for blockchain networks. Another area of interest is to analyze existing and new consensus protocols.

As we saw in *Chapter 1, Blockchain 101*, there are different types of blockchain networks. In particular, two types, permissioned and public (non-permissioned) were discussed. The consensus is also classified based on these two paradigms. For example, Bitcoin is a public blockchain. It runs PoW, sometimes called **Nakamoto consensus**.

In contrast, many permissioned blockchains tend to run variants of traditional or classical distributed consensus. A prime example is IBFT, which is a **blockchained** version of PBFT. Other examples include Tendermint, Casper FFG, and many variants of PBFT. We will discuss more on that later in this chapter.

First, we will look at traditional consensus mechanisms, which are also known as fault-tolerant distributed consensus, classical distributed

consensus, or pre-Bitcoin distributed consensus. Distributed consensus is a highly researched problem and many fundamental ideas to describe and elaborate on the problem have been developed. One of them, and arguably the most famous, is the Byzantine generals problem, which we'll describe next. In addition to the Byzantine generals problem, we will look at relevant and important impossibility results, which will help to build a general understanding of the consensus and relevant limitations. An understanding of these concepts is vital to understand what problem exactly is being solved and how.

The Byzantine generals problem

The problem of reaching agreement in the presence of faults or Byzantine consensus was first formulated by M. Pease, R. Shostak, and L. Lamport. In distributed systems, a common goal is to achieve consensus (agreement) among nodes on the network even in the presence of faults. In order to explain the problem, Lamport came up with an allegorical representation of the problem and named it the **Byzantine generals problem**.

The Byzantine generals problem metaphorically depicts a situation where a Byzantine army, divided into different units, is spread around a city. A general commands each unit, and they can only communicate with each other using a messenger. To be successful, the generals must coordinate their plan and decide whether to attack or retreat. The problem, however, is that any generals could potentially be disloyal and act maliciously to obstruct agreement upon a united plan. The requirement now becomes that every honest general must somehow agree on the same decision even in the presence of treacherous generals.

In order to address this issue, honest (loyal) generals must reach a majority agreement on their plan.

Leslie Lamport introduced numerous fundamental ideas and techniques for distributed consensus.

The famous Byzantine generals problem was formulated by Lamport et al. in their paper: Lamport, L., Shostak, R. and Pease, M., 1982. The Byzantine



Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), pp.382-401.

The paper is available here: <https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/>.

In the digital world, generals are represented by computers (nodes) and communication links are messengers carrying messages. Disloyal generals are faulty nodes. Later in this chapter, we'll see how agreement can be achieved using consensus algorithms even in the presence of faults.

Fault tolerance

A fundamental requirement in a consensus mechanism is that it must be fault-tolerant. In other words, it must be able to tolerate a number of failures in a network and should continue to work even in the presence of faults. This naturally means that there has to be some limit to the number of faults a network can handle, since no network can operate correctly if a large majority of its nodes are failing. Based on the requirement of fault tolerance, consensus algorithms are also called fault-tolerant algorithms, and there are two types of fault-tolerant algorithms.

Types of fault-tolerant consensus

Fault-tolerant algorithms can be divided into two types of fault-tolerance. The first is **Crash fault-tolerance (CFT)** and the other is **Byzantine fault-tolerance (BFT)**. CFT covers only crash faults or, in other words, benign faults. In contrast, BFT deals with the type of faults that are arbitrary and can even be malicious.

Replication is a standard approach to make a system fault-tolerant. Replication results in a synchronized copy of data across all nodes in a network. This technique improves the fault tolerance and availability of the network. This means that even if some of the nodes become faulty, the

overall system/network remains available due to the data being available on multiple nodes.

There are two main types of replication techniques:

- **Active replication**, which is a type where each replica becomes a copy of the original state machine replica.
- **Passive replication**, which is a type where there is only a single copy of the state machine in the system kept by the primary node, and the rest of the nodes/replicas only maintain the state.

In this section, we've briefly looked at what replication is and its types. In the context of fault-tolerant consensus mechanisms, replication plays a vital role by introducing resiliency into the system. We'll now introduce another relevant concept, known as **state machine replication**, which is a standard technique used to achieve fault tolerance in distributed systems.

State machine replication

State machine replication (SMR) is a de facto technique that is used to provide deterministic replication services in order to achieve fault tolerance in a distributed system. State machine replication was first proposed by Lamport in 1978 in his paper:



Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), pp.558-565.

The paper is available here:

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf>

Later, in 1990, Schneider formalized the state machine replication approach and published the results in a paper titled:



Schneider, F.B., 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4), pp.299-319.

The paper is available here:

<https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf>

Now, just before we discuss SMR further, let's understand what a state machine is. At an abstract level, it is a mathematical model that is used to describe a machine that can be in different states. It is important to understand that a state machine can only have one state at a time. A state machine stores a state of the system and transitions it to the next state as a result of input received. As a result of state transition, an output is produced along with an updated state.

The fundamental idea behind SMR can be summarized as follows:

1. All servers always start with the same initial state.
2. All servers receive requests in a **totally ordered** fashion (sequenced as generated from clients).
3. All servers produce the same deterministic output for the same input.

State machine replication is implemented under a primary/backup paradigm, where a primary node is responsible for receiving and broadcasting client requests. This broadcast mechanism is called **total order broadcast** or **atomic broadcast**, which ensures that backup or replica nodes receive and execute the same requests in the same sequence as the primary.

Consequently, this means that all replicas will eventually have the same state as the primary, thus resulting in achieving consensus. In other words, this means that total order broadcast and distributed consensus are equivalent problems; if you solve one, the other is solved too.

Now that we understand the basics of replication and fault tolerance, it is important to understand that fault tolerance works up to a certain threshold.

For example, if a network has a vast majority of constantly failing nodes and communication links, it is not hard to understand that this type of network may not be as fault-tolerant as we might like it to be. In other words, even in the presence of fault-tolerant measures, if there is a lack of resources on a network, the network may still not be able to provide the required level of fault tolerance. In some scenarios, it might be impossible to provide the required services due to a lack of resources in a system. In distributed computing, such impossible scenarios are researched and reported as impossibility results.

In distributed computing, impossibility results provide an understanding of whether a problem is solvable and the minimum resources required to do so. If the problem is unsolvable, then these results give a clear understanding that a specific task cannot be accomplished and no further research is necessary. From another angle, we can say that impossibility results (sometimes called unsolvability results) show that certain problems are not computable under insufficient resources. Impossibility results unfold deep aspects of distributed computing and enable us to understand why certain problems are difficult to solve and under what conditions a previously unsolved problem might be solved.

The requirement of minimum available resources is known as **lower bound results**. The problems that are not solvable under any conditions are known as **unsolvability results**. For example, it has been proven that asynchronous deterministic consensus is impossible. This result is known as the FLP impossibility result, which we'll introduce next.

FLP impossibility

FLP impossibility is a fundamental unsolvability result in distributed computing theory that states that in an asynchronous environment, the deterministic consensus is impossible, even if only one process is faulty.

FLP is named after the authors' names, Fischer, Lynch, and Patterson, who in 1985 introduced this result. This result was presented in their paper:



Fischer, M.J., Lynch, N.A. and Paterson, M.S., 1982. Impossibility of distributed consensus with one faulty process (No. MIT/LCS/TR-282). *Massachusetts Inst of Tech Cambridge lab for Computer Science*.

The paper is available at:

<https://apps.dtic.mil/dtic/tr/fulltext/u2/a132503.pdf>

To circumvent **FLP impossibility**, several techniques have been introduced in the literature. These techniques include:

- **Failure detectors**, which can be seen as **oracles** associated with processors to detect failures.
- **Randomized algorithms** have been introduced to provide a probabilistic termination guarantee. The core idea behind the randomized protocols is that the processors in such protocols can make a random choice of decision value if the processor does not receive the required quorum of trusted messages.
- **Synchrony assumptions**, where additional synchrony and timing assumptions are made to ensure that the consensus algorithm terminates and makes progress.

Now that we understand a fundamental impossibility result, let's look at another relevant result that highlights the unsolvability of consensus due to a lack of resources: that is, a lower bound result. We can think of lower bound as a minimum amount of resources, for example, the number of processors or communication links required to solve a problem. In other words, if a minimum required number of resources is not available in a system, then the problem cannot be solved. In the context of a consensus problem, a fundamental proven result is *lower bounds on the number of processors*, which we describe next.

Lower bounds on the number of processors to solve consensus

As we described previously, there are proven results in distributed computing that state several lower bounds, for example, the minimum number of processors required for consensus or the minimum number of rounds required to achieve consensus. The most common and fundamental of these results is the minimum number of processors required for consensus. These results are listed below:

- In the case of CFT, at least $2F + 1$ number of nodes is required to achieve consensus.
- In the case of BFT, at least $3F + 1$ number of nodes is required to achieve consensus.

F represents the number of failures.

These lower bounds are discussed in several papers with relevant proofs. The most fundamental one is by Lamport et. al:

Pease, M., Shostak, R. and Lamport, L., 1980. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2), pp.228-234.

The paper is available here:



<https://lamport.azurewebsites.net/pubs/reaching.pdf>

Another paper that provides a number of impossibility proofs is:

Fischer, M.J., Lynch, N.A. and Merritt, M., 1986. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1), pp.26-39.

The paper is available here:

<https://apps.dtic.mil/dtic/tr/fulltext/u2/a157411.pdf>

We have now covered the fundamentals of distributed consensus theory. Now, we'll delve a little bit deeper into the analysis and design of consensus algorithms.

Analysis and design

In order to analyze and understand a consensus algorithm, we need to define a model under which our algorithm will run. This model provides some assumptions about the operating environment of the algorithm and provides a way to intuitively study and reason about the various properties of the algorithm.

In the following sections, we'll describe a model that is useful for describing and analyzing consensus mechanisms.

Model

Distributed computing systems represent different entities in the system under a computational model. This computational model is a beneficial way of describing the system under some system assumptions. A computational model represents processes, network conditions, timing assumptions, and how all these entities interact and work together. We will now look at this model in detail and introduce all objects one by one.

Processes

Processes communicate with each other by passing messages to each other. This is why these systems are called message-passing distributed systems. There is another class, called shared memory, which we will not discuss here as we are only dealing with message-passing systems.

Timing assumptions

There are also some timing assumptions that are made when designing consensus algorithms.

Synchrony

In synchronous systems, there is a known upper bound on the communication and processor delays. Synchronous algorithms are designed to be run on synchronous networks. At a fundamental level, in a synchronous system, a message sent by a processor to another is received by the receiver in the same communication round as it is sent.

Asynchrony

In asynchronous systems, there is no upper bound on the communication and processor delays. In other words, it is impossible to define an upper bound for communication and processor delays in asynchronous systems. Asynchronous algorithms are designed to run on asynchronous networks without any timing assumptions. These systems are characterized by the unpredictability of message transfer (communication) delays and processing delays. This scenario is common in large-scale geographically dispersed distributed systems and systems where the input load is unpredictable.

Partial synchrony

In this model, there is an upper bound on the communication and processor delays, however, this upper bound is not known to the processors.

Eventually, synchronous systems are a type of partial synchrony, which means that the system becomes synchronous after an instance of time called global stabilization time or GST. GST is not known to the processors. Generally, partial synchrony captures the fact that, usually, the systems are synchronous, but there are arbitrary but bounded asynchronous periods. Also, the system at some point is synchronous for long enough that processors can decide (achieve agreement) and terminate during that period.

Now that we understand the fundamentals of the distributed consensus theory, let's look at two main classes of consensus algorithms. This categorization emerged after the invention of Bitcoin. Prior to Bitcoin, there's a long history of research in distributed consensus protocols.

Classification

The consensus algorithms can be classified into two broad categories:

- **Traditional**—voting-based consensus
- **Lottery-based**—Nakamoto and post-Nakamoto consensus

Traditional voting-based consensus has been researched in distributed systems for many decades. Many fundamental results and a lot of groundbreaking work have already been produced in this space. Algorithms like **Paxos** and **PBFT** are prime examples of such types of algorithms.

Traditional consensus can also be called fault-tolerant distributed consensus. In other words, this is a class of consensus algorithms that existed before Bitcoin and has been part of distributed system research for almost three decades.

Lottery-based or Nakamoto-type consensus was first introduced with Bitcoin. This class can also be simply called blockchain consensus.

The fundamental requirements of consensus algorithms boil down to safety and liveness conditions. A consensus algorithm must be able to satisfy the safety and liveness properties. Safety is usually based on some safety requirements of the algorithms, such as agreement, validity, and integrity. Liveness means that the protocol can make progress even if the network conditions are not ideal.

Now we'll define these terms separately:

Safety: This requirement generally means that nothing bad happens. There are usually three properties within this class of requirements, which are listed as follows:

- **Agreement.** The agreement property requires that no two processes decide on different values.
- **Validity.** Validity states that if a process has decided a value, that value must have been proposed by a process. In other words, the decided

value is always proposed by an honest process and has not been created out of thin air.

- **Integrity.** A process must decide only once.

Liveness: This requirement generally means that something good eventually happens.

- **Termination.** This liveness property states that each honest node must eventually decide on a value.

With this, we have covered the classification and requirements of consensus algorithms. In the next section, we'll introduce various consensus algorithms, which we can evaluate using the consensus models covered in this section.

Algorithms

In this section, we will discuss the key algorithms in detail. We'll be looking at the two main types of fault-tolerant algorithms, CFT and BFT.

CFT algorithms

We'll begin by looking at some algorithms that solve the consensus problem with crash fault tolerance. One of the most fundamental algorithms in this space is Paxos.

Paxos

Leslie Lamport developed Paxos. It is the most fundamental distributed consensus algorithm, allowing consensus over a value under unreliable communications. In other words, Paxos is used to build a reliable system that works correctly, even in the presence of faults.



Paxos was proposed first in 1989 and then later, more formally, in 1998 in the following paper:

Lamport, L., 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), pp.133-169.

The paper is available here:

<https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf>.



Note that Paxos works under an asynchronous network model and supports the handling of only benign failures. This is not a Byzantine fault-tolerant protocol. However, later, a variant of Paxos was developed that provides Byzantine fault tolerance. The original paper in which a Byzantine fault-tolerant version of Paxos is described is available here:

Lamport, L., 2011, September. Byzantizing Paxos by refinement. *International Symposium on Distributed Computing* (pp. 211-224). Springer, Berlin, Heidelberg.

A link to the paper is available here:

<http://lamport.azurewebsites.net/pubs/web-byzpaxos.pdf>

Paxos makes use of $2F + 1$ processes to ensure fault tolerance in a network where processes can crash fault, that is, experience benign failures. Benign failure means either the loss of a message or a process stops. In other words, Paxos can tolerate one crash failure in a three-node network.

Paxos is a two-phase protocol. The first phase is called the *prepare* phase, and the next phase is called the *accept* phase. Paxos has proposer and acceptors as participants, where the proposer is the replicas or nodes that propose the values and acceptors are the nodes that accept the value.

How Paxos works

The Paxos protocol assumes an asynchronous message-passing network with less than 50% of crash faults. As usual, the critical properties of the Paxos consensus algorithm are safety and liveness. Under safety, we have:

- **Agreement**, which specifies that no two different values are agreed on. In other words, no two different learners learn different values.
- **Validity**, which means that only the proposed values are decided. In other words, the values chosen or learned must have been proposed by a processor.

Under liveness, we have:

- **Termination**, which means that, eventually, the protocol is able to decide and terminate. In other words, if a value has been chosen, then eventually learners will learn it.

Processes can assume different roles, which are listed as follows:

- **Proposers**, elected leader(s) that can propose a new value to be decided.
- **Acceptors**, which participate in the protocol as a means to provide a majority decision.
- **Learners**, which are nodes that just observe the decision process and value.



A single process in a Paxos network can assume all three roles.

The key idea behind Paxos is that the proposer node proposes a value, which is considered final only if a majority of the acceptor nodes accept it. The learner nodes also learn this final decision.

Paxos can be seen as a protocol that is quite similar to the two-phase commit protocol. **Two-phase commit (2PC)** is a standard atomic commitment protocol to ensure that transactions are committed in distributed databases only if all participants agree to commit. Even if a single node cannot agree to commit the transaction, it is fully rolled back.

Similarly, in Paxos, in the first phase, the proposer sends a proposal to the acceptors, if and when they accept the proposal, the proposer broadcasts a request to commit to the acceptors. Once the acceptors commit and report

back to the proposer, the proposal is considered final, and the protocol concludes. In contrast with the two-phase commit, Paxos introduced ordering (sequencing to achieve total order) of the proposals and majority-based acceptance of the proposals instead of expecting all nodes to agree (to allow progress even if some nodes fail). Both of these improvements contribute toward ensuring the safety and liveness of the Paxos algorithm.



An excellent explanation of the two-phase commit is available here:

https://en.wikipedia.org/wiki/Two-phase_commit_protocol

We'll now describe how the Paxos protocol works step by step:

1. The proposer proposes a value by broadcasting a message, $\langle \text{prepare}(n) \rangle$, to all acceptors.
2. Acceptors respond with an acknowledgment message if proposal n is the highest that the acceptor has responded to so far. The acknowledgment message $\langle \text{ack}(n, v, s) \rangle$ consists of three variables where n is the proposal number, v is the proposal value of the highest numbered proposal the acceptor has accepted so far, and s is the sequence number of the highest proposal accepted by the acceptor so far. This is where acceptors agree to commit the proposed value. The proposer now waits to receive acknowledgment messages from the majority of the acceptors indicating the **chosen** value.
3. If the majority is received, the proposer sends out the "accept" message $\langle \text{accept}(n, v) \rangle$ to the acceptors.
4. If the majority of the acceptors accept the proposed value (now the "accept" message), then it is decided: that is, agreement is achieved.
5. Finally, in the learning phase, acceptors broadcast the "accepted" message $\langle \text{accepted}(n, v) \rangle$ to the proposer. This phase is necessary to disseminate which proposal has been finally accepted. The proposer then informs all other learners of the decided value. Alternatively, learners can learn the decided value via a message that contains the accepted value (decision value) multicast by acceptors.

We can visualize this process in the following diagram:

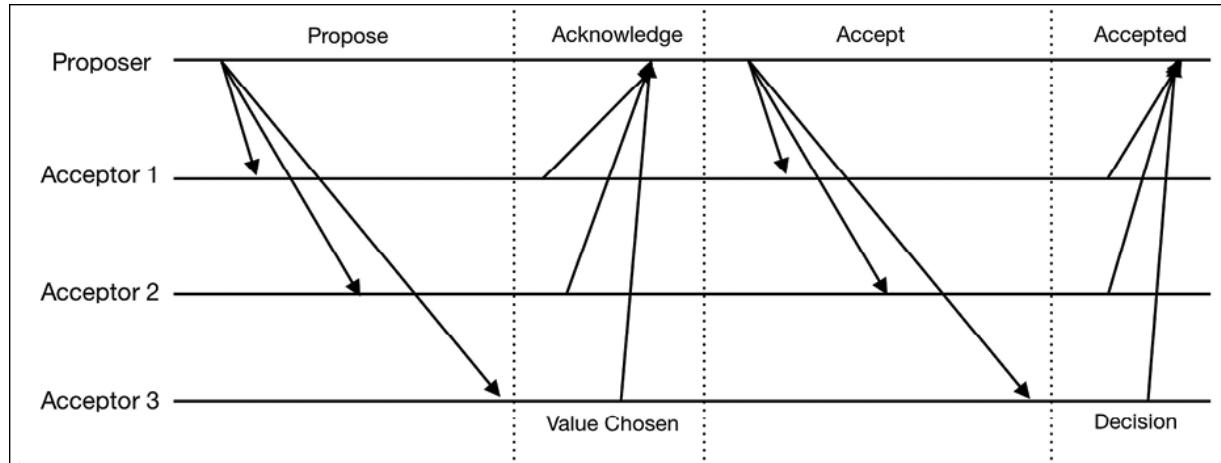


Figure 5.1: How Paxos works

Let's move on to see how Paxos achieves the much-desired properties of safety and liveness.

How Paxos achieves safety and liveness

A natural question arises about how Paxos ensures its safety and liveness guarantees. Paxos, at its core, is quite simple, yet it achieves all these properties efficiently. The actual proofs for the correctness of Paxos are quite in-depth and are not the subject of this chapter. However, the intuition behind each property is presented as follows:

- **Agreement** is ensured by enforcing that only one proposal can win votes from a majority of the acceptors.
- **Validity** is ensured by enforcing that only the genuine proposals are decided. In other words, no value is committed unless it is proposed in the proposal message first.
- **Liveness** or termination is guaranteed by ensuring that at some point during the protocol execution, eventually there is a period during which there is only one fault-free proposer.

For detailed proofs and correctness analyses, refer to the following papers:



Lamport, L., 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), pp.133-169.

<https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf>

Lamport, L., 2006. Fast Paxos. *Distributed Computing*, 19(2), pp.79-103.

<https://www.microsoft.com/en-us/research/publication/fast-paxos/>

In summary, the key points to remember about Paxos are that:

- First, a proposer suggests a value with the aim that acceptors achieve agreement on it.
- The decided value is a result of majority consensus among the acceptors and is finally learned by the learners.

There are many other protocols that have emerged from basic Paxos, such as Multi-Paxos, Fast Paxos, and Cheap Paxos.

Even though the Paxos algorithm is quite simple at its core, it is seen as challenging to understand, and many academic papers have been written to explain it. This slight problem, however, has not prevented it being implemented in many production networks, such as Google's Spanner, as it has proven to be the most efficient protocol to solve the consensus problem.

Nevertheless, there have been attempts to create alternative easy-to-understand algorithms. Raft is such an attempt to create an easy-to-understand CFT algorithm.

Raft

The Raft protocol is a CFT consensus mechanism developed by Diego Ongaro and John Ousterhout at Stanford University. In Raft, the leader is always assumed to be honest.

At a conceptual level, it is a replicated log for a **replicated state machine (RSM)** where a unique leader is elected every "term" (time division) whose

log is replicated to all follower nodes.

Raft is composed of three sub-problems:

- **Leader election** (a new leader election in case the existing one fails)
- **Log replication** (leader to follower log synch)
- **Safety** (no conflicting log entries (index) between servers)

The Raft protocol ensures election safety, leader append only, log matching, leader completeness, and state machine safety.

Each server in Raft can have either a **follower**, **leader**, or **candidate** state.

The protocol ensures election **safety** (that is, only one winner each election term) and **liveness** (that is, some candidate must eventually win).

How Raft works

The following steps will describe how the Raft protocol functions. At a fundamental level, the protocol is quite simple and can be described simply by the following sequence:

Node starts up → Leader election → Log replication

1. First, the node starts up.
2. After this, the leader election process starts. Once a node is elected as leader, all changes go through that leader.
3. Each change is entered into the node's log.
4. Log entry remains uncommitted until the entry is replicated to follower nodes and the leader receives write confirmation votes from a majority of the nodes, then it is committed locally.
5. The leader notifies the followers regarding the committed entry.
6. Once this process ends, agreement is achieved.

The state transition of the Raft algorithm can be visualized in the following diagram:

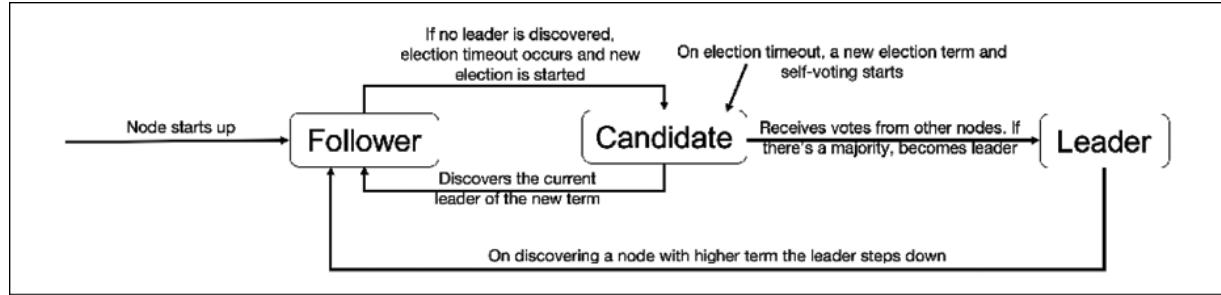


Figure 5.2: Raft state transition

We saw earlier that data is eventually replicated across all nodes in a consensus mechanism. In Raft, the log (data) is eventually replicated across all nodes. We describe this process of log replication next.

Log replication

Log replication logic can be visualized in the following diagram. The aim of log replication is to synchronize nodes with each other.

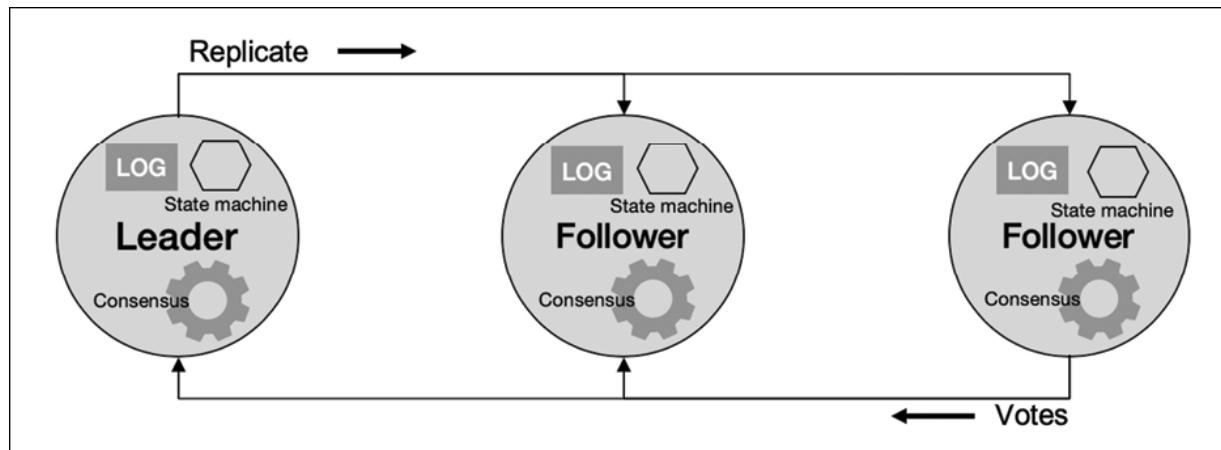


Figure 5.3: Log replication mechanism

Log replication is a simple mechanism. As shown in the preceding diagram, the leader is responsible for log replication. Once the leader has a new entry in its log, it sends out the requests to replicate to the follower nodes. When the leader receives enough confirmation votes back from the follower nodes indicating that the replicate request has been accepted and processed by the followers, the leader commits that entry to its local state machine. At this stage, the entry is considered committed.

With this, our discussion on CFT algorithms is complete. Now we'll introduce Byzantine fault-tolerant algorithms, which have been a research area for many years in distributed computing.

BFT algorithms

We described the formulation of the Byzantine generals problem at the start of this chapter. In this section, we'll introduce the mechanisms that were developed to solve the Byzantine generals (consensus in the presence of faults) problem.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) was developed in 1999 by Miguel Castro and Barbara Liskov. PBFT, as the name suggests, is a protocol developed to provide consensus in the presence of Byzantine faults. Before PBFT, Byzantine fault tolerance was considered impractical. With PBFT, it was demonstrated for the first time that practical Byzantine fault tolerance is possible.

PBFT comprises three sub-protocols called **normal operation**, **view change**, and **checkpointing**.

Normal operation sub-protocol refers to a scheme that is executed when everything is running normally and no errors are in the system. View change is a sub-protocol that runs when a faulty leader node is detected in the system. Checkpointing is another sub-protocol, which is used to discard the old data from the system.

The PBFT protocol comprises three phases or steps. These phases run in a sequence to achieve consensus. These phases are pre-prepare, prepare, and commit, which we will cover in detail shortly.

The protocol runs in rounds where, in each round, an elected leader node, called the primary node, handles the communication with the client. In each round, the protocol progresses through the three previously mentioned

phases. The participants in the PBFT protocol are called replicas, where one of the replicas becomes primary as a leader in each round, and the rest of the nodes acts as backups. PBFT is based on the SMR protocol introduced earlier. Here, each node maintains a local log, and the logs are kept in sync with each other via the consensus protocol: that is, PBFT.

As we saw earlier, in order to tolerate Byzantine faults, the minimum number of nodes required is $N = 3F + 1$, where N is the number of nodes and F is the number of faulty nodes. PBFT ensures Byzantine fault tolerance as long as the number of nodes in a system stays $N \geq 3F + 1$.

We will now look at how the PBFT protocol works.

In summary, when a client sends a request to a primary, the protocol initiates a sequence of operations between replicas, which eventually leads to consensus and a reply back to the client. These sequences of operations are divided into different phases:

- Pre-prepare
- Prepare
- Commit

In addition, each replica maintains a local state comprising three main elements:

- Service state
- A message log
- A number representing that replica's current view

Now we'll discuss the phases mentioned above one by one, starting with pre-prepare.

Pre-prepare:

This is the first phase in the protocol, where the primary node, or **primary**, receives a request from the client. The primary node assigns a sequence

number to the request. It then sends the pre-prepare message with the request to all backup replicas.

When the pre-prepare message is received by the backup replicas, it checks a number of things to ensure the validity of the message:

- First, whether the digital signature is valid.
- After this, whether the current view number is valid.
- Then, that the sequence number of the operation's request message is valid.
- Finally, if the digest/hash of the operation's request message is valid.

If all of these elements are valid, then the backup replica accepts the message. After accepting the message, it updates its local state and progresses toward the prepare phase.

Prepare:

A prepare message is sent by each backup to all other replicas in the system. Each backup waits for at least $2F + 1$ prepare messages to be received from other replicas. They also check whether the prepare message contains the same view number, sequence number, and message digest values. If all these checks pass, then the replica updates its local state and progresses toward the commit phase.

Commit:

In the commit phase, each replica sends a commit message to all other replicas in the network. The same as the prepare phase, replicas wait for $2F + 1$ commit messages to arrive from other replicas. The replicas also check the view number, sequence number, and message digest values. If they are valid for $2F + 1$ commit messages received from other replicas, then the replica executes the request, produces a result, and finally, updates its state to reflect a commit. If there are already some messages queued up, the replica will execute those requests first before processing the latest sequence numbers. Finally, the replica sends the result to the client in a reply message.

The client accepts the result only after receiving $2F + 1$ reply messages containing the same result.

Now, let's move on and look at how PBFT works in some more detail.

How PBFT works

At a high level, the protocol works as follows:

1. A client sends a request to invoke a service operation in the primary.
2. The primary multicasts the request to the backups.
3. Replicas execute the request and send a reply to the client.
4. The client waits for replies from different replicas with the same result; this is the result of the operation.

Now we'll describe each phase of the protocol (**pre-prepare**, **prepare**, and **commit**) in more formal terms.

The pre-prepare sub-protocol algorithm:

1. Accepts a request from the client.
2. Assigns the next sequence number.
3. Sends the pre-prepare message to all backup replicas.

The prepare sub-protocol algorithm:

1. Accepts the pre-prepare message. If the backup has not accepted any pre-prepare messages for the same view or sequence number, then it accepts the message.
2. Sends the prepare message to all replicas.

The commit sub-protocol algorithm:

1. The replica waits for $2F$ prepare messages with the same view, sequence, and request.
2. Sends a commit message to all replicas.

3. Waits until a $2F + 1$ valid commit message arrives and is accepted.
4. Executes the received request.
5. Sends a reply containing the execution result to the client.

In summary, the primary purpose of these phases is to achieve consensus, where each phase is responsible for a critical part of the consensus mechanism, which after passing through all phases, eventually ends up achieving consensus.

One key point to remember about each phase is listed as follows:

Pre-prepare: This phase assigns a unique sequence number to the request. We can think of it as an orderer.



Prepare: This phase ensures that honest replicas/nodes in the network agree on the total order of requests within a view. Note that the pre-prepare and prepare phases together provide the total order to the messages.

Commit: This phase ensures that honest replicas/nodes in the network agree on the total order of requests across views.

The normal view of the PBFT protocol can be visualized as shown as follows:

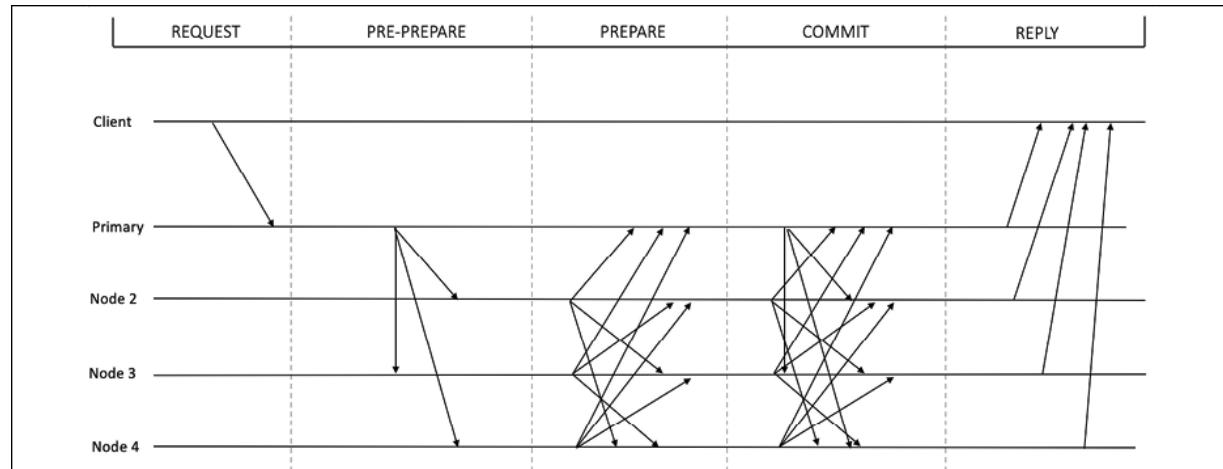


Figure 5.4: PBFT protocol

During the execution of the protocol, the integrity of the messages and protocol operations must be maintained to provide an adequate level of security and assurance. This is maintained by the use of digital signatures. In addition, certificates are used to ensure the adequate majority of participants (nodes).



Note that these certificates are not usual digital certificates commonly used in PKI and IT infrastructures to secure assets such as servers.

We'll describe the concept of certificates next.

Certificates in PBFT:

Certificates in PBFT protocols are used to demonstrate that at least $2F + 1$ nodes have stored the required information. In other words, the collection of $2F + 1$ messages of a particular type is considered a certificate. For example, if a node has collected $2F + 1$ messages of type prepare, then combining it with the corresponding pre-prepare message with the same view, sequence, and request represents a certificate, called a prepared certificate. Similarly, a collection of $2F + 1$ commit messages is called a commit certificate.

There are also a number of variables that the PBFT protocol maintains in order to execute the algorithm. These variables and the meanings of these are listed as follows:

State variable	Explanation
v	View number
m	Latest request message
n	Sequence number of the message

h	Hash of the message
i	Index number
C	Set of all checkpoints
P	Set of all pre-prepare and corresponding prepare messages
O	Set of pre-prepare messages without corresponding request messages

We can now look at the types of messages and their formats, which becomes quite easy to understand if we refer to the preceding variables table shown.

Types of messages:

The PBFT protocol works by exchanging several messages. A list of these messages is presented as follows with their format and direction.

The following table contains message types and relevant details:

Message	From	To	Format	Signed by
Request	Client	Primary	<REQUEST, m>	Client
Pre-Prepare	Primary	Backups	<PRE-PREPARE, v, n, h>	Client
Prepare	Replica	Replicas	<PREPARE, v, n, h, i>	Replica
Commit	Replica	Replicas	<COMMIT, v, n, h, i>	Replica
Reply	Replicas	Client	<REPLY, r, i>	Replica

View change	Replica	Replicas	<VIEWCHANGE, v+1, n, C, P, i>	Replica
New view	Primary replica	Replicas	<NEWVIEW, v + 1, v, O>	Replica
Checkpoint	Replica	Replicas	<CHECKPOINT, n, h, i>	Replica

Let's look at some specific message types that are exchanged during the PBFT protocol.

View-change:

View-change occurs when a primary is suspected faulty. This phase is required to ensure protocol progress. With the view change sub-protocol, a new primary is selected, which then starts normal mode operation again. The new primary is selected in a round-robin fashion.

When a backup replica receives a request, it tries to execute it after validating the message, but for some reason, if it does not execute it for a while, the replica times out and initiates the view change sub-protocol.

In the view change protocol, the replica stops accepting messages related to the current view and updates its state to view-change. The only messages it can receive in this state are checkpoint messages, view-change messages, and new-view messages. After that, it sends a view-change message with the next view number to all replicas.

When this message arrives at the new primary, the primary waits for at least $2F$ view-change messages for the next view. If at least $2F$ view-change messages are received it broadcasts a new view message to all replicas and progresses toward running normal operation mode once again.

When other replicas receive a new-view message, they update their local state accordingly and start normal operation mode.

The algorithm for the view-change protocol is shown as follows:

1. Stop accepting **pre-prepare**, **prepare**, and **commit** messages for the current view.
2. Create a set of all the certificates prepared so far.
3. Broadcast a view-change message with the next view number and a set of all the prepared certificates to all replicas.

The view change protocol can be visualized in the following diagram:

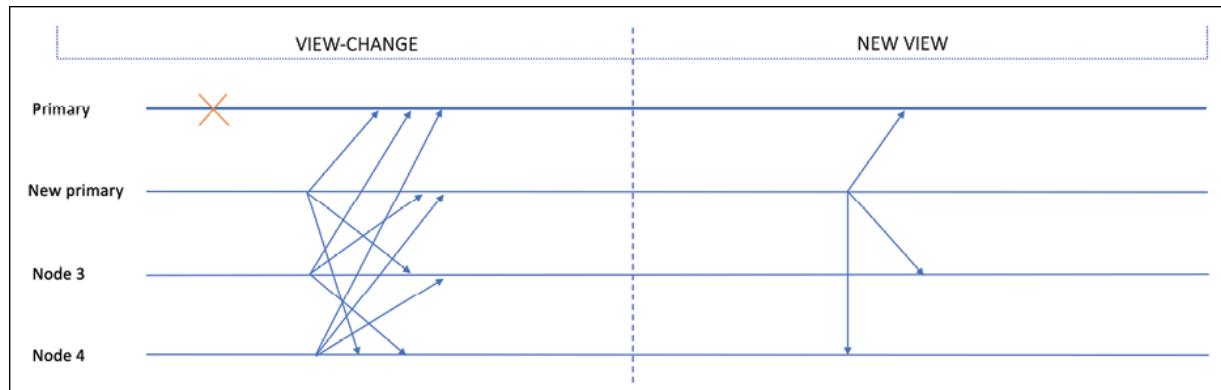


Figure 5.5: View-change sub-protocol

The view-change sub-protocol is a mechanism to achieve liveness. Three smart techniques are used in this sub-protocol to ensure that, eventually, there is a time when the requested operation executes:

1. A replica that has broadcast the view-change message waits for $2F+1$ view-change messages and then starts its timer. If the timer expires before the node receives a new-view message for the next view, the node will start the view change for the next sequence but will increase its timeout value. This will also occur if the replica times out before executing the new unique request in the new view.
2. As soon as the replica receives $F+1$ view-change messages for a view number greater than its current view, the replica will send the view-change message for the smallest view it knows of in the set so that the next view change does not occur too late. This is also the case even if the timer has not expired; it will still send the view change for the smallest view.

3. As the view change will only occur if at least $F+1$ replicas have sent the view-change message, this mechanism ensures that a faulty primary cannot indefinitely stop progress by successively requesting view changes.

Next, let's look in more detail at the checkpoint sub-protocol, another important PBFT process.

The checkpoint sub-protocol:

Checkpointing is a crucial sub-protocol. It is used to discard old messages in the log of all replicas. With this, the replicas agree on a stable checkpoint that provides a snapshot of the global state at a certain point in time. This is a periodic process carried out by each replica after executing the request and marking that as a checkpoint in its log. A variable called **low watermark** (in PBFT terminology) is used to record the sequence number of the last stable checkpoint. This checkpoint is then broadcast to other nodes. As soon as a replica has at least $2F+1$ checkpoint messages, it saves these messages as proof of a stable checkpoint. It discards all previous **pre-prepare**, **prepare**, and **commit** messages from its logs.

PBFT advantages and disadvantages:

PBFT is indeed a revolutionary protocol that has opened up a new research field of practical Byzantine fault-tolerant protocols. The original PBFT does have some strengths, but it does have some limitations. We'll discuss most of the commonly cited strengths and limitations in the following section.

Strengths:

PBFT provides immediate and deterministic transaction finality. This is in contrast with the PoW protocol, where a number of confirmations are required to finalize a transaction with high probability.

PBFT is also energy efficient as compared to PoW, which consumes a tremendous amount of electricity.

Weaknesses:

PBFT is not very scalable. This is the reason it is more suitable for consortium networks, instead of public blockchains. It is, however, much faster than PoW protocols.

Sybil attacks can be carried out on a PBFT network, where a single entity can control many identities to influence the voting and subsequently the decision. However, the fix is trivial and, in fact, this is not very practical in consortium networks where all identities are known on the network. This problem can be addressed simply by increasing the number of nodes in the network.

PBFT in blockchain:

In the traditional client-server model, PBFT works well; however, in the case of blockchain, directly implementing PBFT in its original state may not work correctly. This is because PBFT's original design was not developed for blockchain. In the following section, we present a few changes that are required to create a blockchain version of PBFT. This is not an exhaustive list; however, it does provide a baseline of requirements.

This research resulted in IBFT and PBFT implementation in Sawtooth and other blockchains. In all these scenarios, some changes have been made in the core protocol to ensure that they're compatible with the blockchain environment.



PBFT has been implemented in Hyperledger Sawtooth. More details on this implementation can be found here:

<https://github.com/hyperledger/sawtooth-rfcs/blob/master/text/0019-pbft-consensus.md>

We will now introduce an algorithm called IBFT, which was inspired by PBFT. Another algorithm that has been inspired by PBFT and DLS is Tendermint, which we will also present shortly.

Istanbul Byzantine Fault Tolerance

IBFT was developed by AMIS Technologies as a variant of PBFT suitable for blockchain networks. It was presented in EIP 650 for the Ethereum blockchain.

The differences between PBFT and IBFT

Let's first discuss the primary differences between the PBFT and IBFT protocols. They are as follows:

- There is no distinctive concept of a client in IBFT. Instead, the proposer can be seen as a client, and in fact, all validators can be considered clients.
- There is a concept of dynamic validators, which is in contrast with the original PBFT, where the nodes are static. However, in IBFT, the validators can be voted in and out as required.
- There are two types of nodes in an IBFT network, nodes and validators. Nodes are synchronized with the blockchain without participating in the IBFT consensus process. In contrast, validators are the nodes that participate in the IBFT consensus process.
- IBFT relies on a more straightforward structure of view-change (round change) messages as compared to PBFT.
- In contrast with PBFT, in IBFT there is no concrete concept of checkpoints. However, each block can be considered an indicator of the progress so far (the chain height).
- There is no concept of garbage collection in IBFT.

Now we've covered the main points of comparison between the two BFT algorithms, we'll examine how the IBFT protocol runs, and its various phases.

How IBFT works

IBFT assumes a network model under which it is supposed to run. The model is composed of at least $3F+1$ processes (standard BFT assumption), a partially synchronous message-passing network, and sound cryptography.

By sound cryptography, it is assumed that digital signatures and relevant cryptographic protocols such as cryptographic hash functions are secure.

The IBFT protocol runs in rounds. It has three phases: *pre-prepare*, *prepare*, and *commit*. In each round, usually, a new leader is elected based on a round-robin mechanism. The following flowchart shows how the IBFT protocol works:

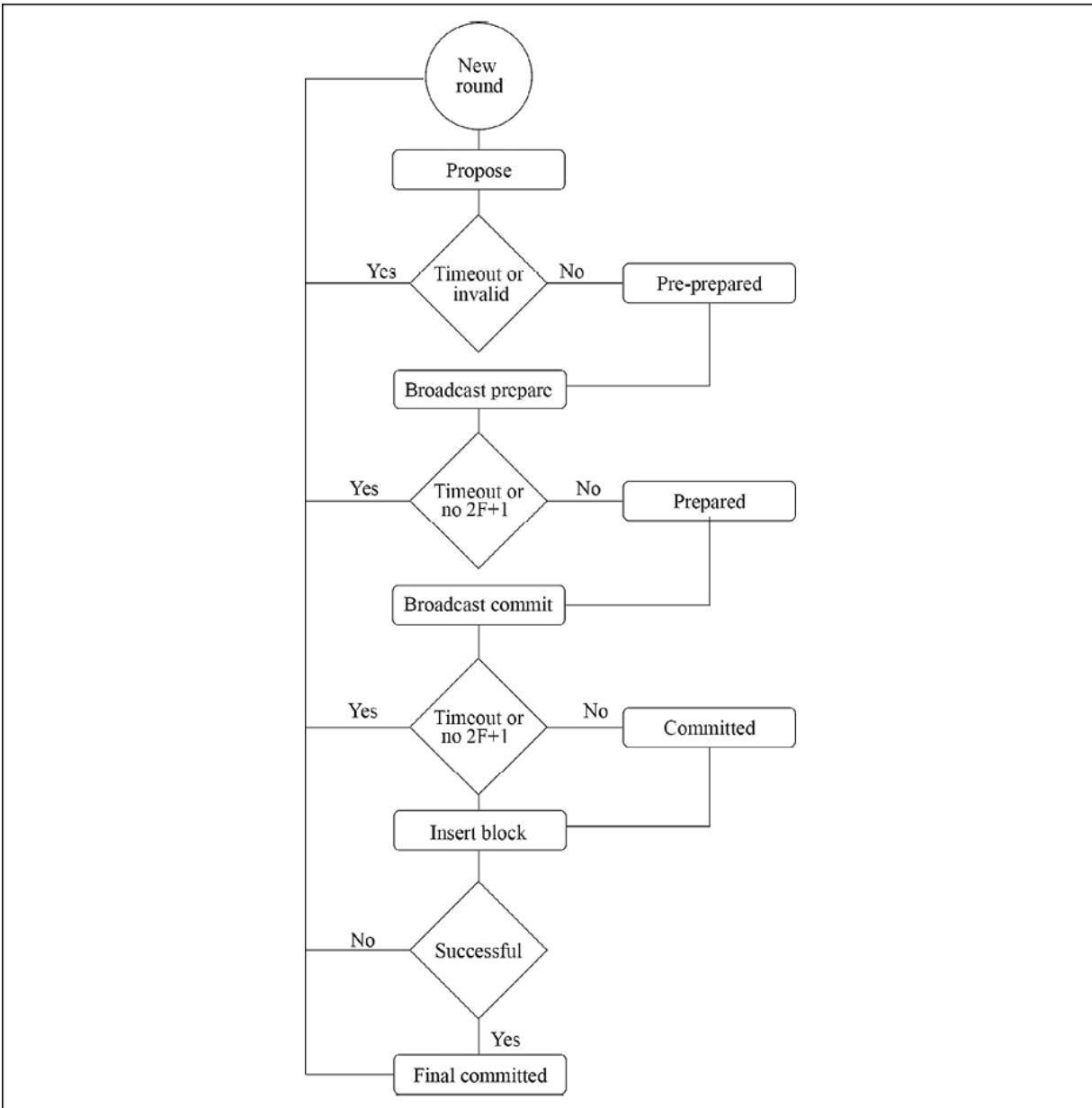


Figure 5.6: IBFT flow chart

In the preceding diagram, the flow of IBFT is shown. We'll discuss this process step by step in the following section:

1. The protocol starts with a new round. In the new round, the selected proposer broadcasts a proposal (block) as a pre-prepare message.
2. The nodes that receive this pre-prepare message validate the message and accept it if it is a valid message. The nodes also then set their state to pre-prepared.
3. At this stage, if a timeout occurs, or a proposal is seen as invalid by the nodes, they will initiate a round change. The normal process then begins again with a proposer, proposing a block.
4. Nodes then broadcast the prepare message and wait for $2F+1$ prepare messages to be received from other nodes. If the nodes do not receive $2F+1$ messages in time, then they time out, and the round change process starts. The nodes then set their state to prepared after receiving $2F+1$ messages from other nodes.
5. Finally, the nodes broadcast a commit message and wait for $2F+1$ messages to arrive from other nodes. If they are received, then the state is set to committed, otherwise, timeout occurs and the round change process starts.
6. Once committed, block insertion is tried. If it succeeds, the protocol proceeds to the final committed state and, eventually, a new round starts. If insertion fails for some reason, the round change process triggers. Again, nodes wait for $2F+1$ round change messages, and if the threshold of the messages is received, then round change occurs.

Now that we've understood the flow of IBFT, let's now further explore which states it maintains and how.

Consensus states

IBFT is an SMR algorithm. Each validator maintains a state machine replica in order to reach block consensus, that is, agreement. These states are listed as follows with an explanation:

- **New round:** In this state, a new round of the consensus mechanism starts, and the selected proposer sends a new block proposal to other validators. In this state, all other validators wait for the `PRE-PREPARE` message.
- **Pre-prepared:** A validator transitions to this state when it has received a `PRE-PREPARE` message and broadcasts a `PREPARE` message to other validators. The validator then waits for $2F + 1$ `PREPARE` or `COMMIT` messages.
- **Prepared:** This state is achieved by a validator when it has received $2F+1$ prepare messages and has broadcast the commit messages. The validator then awaits $2F+1$ commit messages to arrive from other validators.
- **Committed:** The state indicates that a validator has received $2F+1$ `COMMIT` messages. The validator at this stage can insert the proposed block into the blockchain.
- **Final committed:** This state is achieved by a validator when the newly committed block is inserted successfully into the blockchain. At this state, the validator is also ready for the next round of consensus.
- **Round change:** This state indicates that the validators are waiting for $2F+1$ round change messages to arrive for the newly proposed new round number.

The IBFT protocol can be visualized using a diagram, similar to PBFT, as follows:

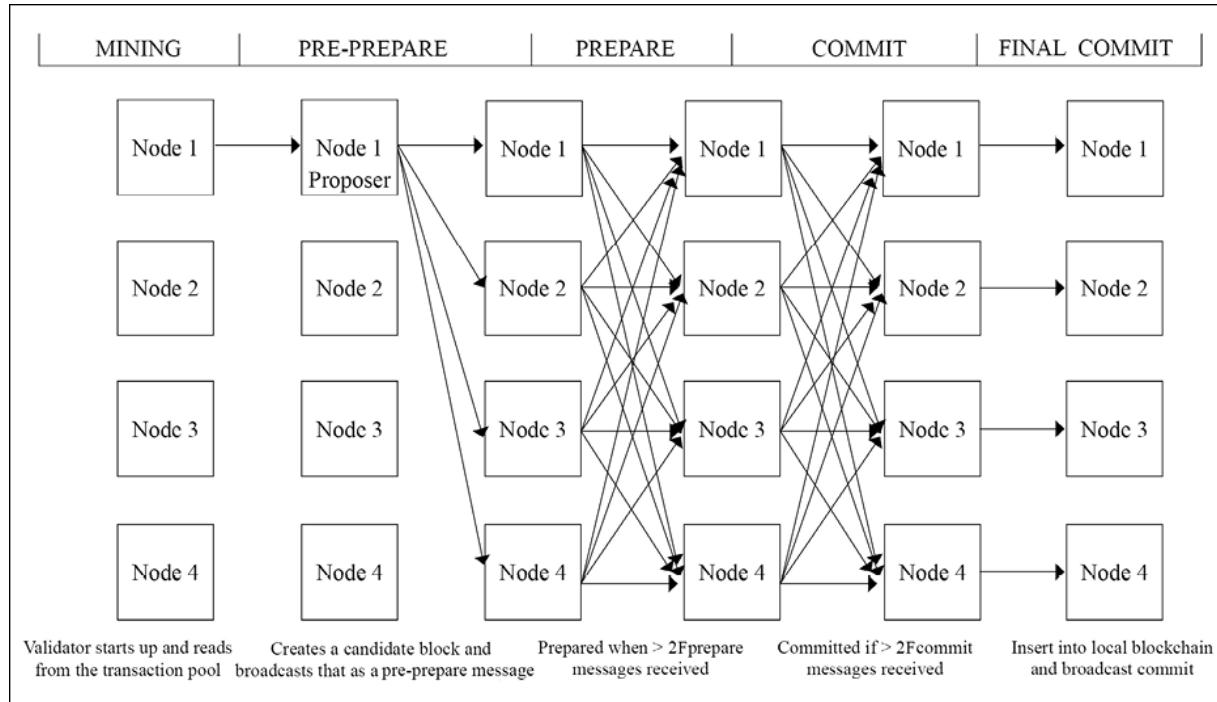


Figure 5.7: IBFT, PBFT-like flow

An additional mechanism that makes IBFT quite appealing is its validator management mechanism. By using this mechanism, validators can be added or removed by voting between members of the network. This is quite a useful feature and provides the right level of flexibility when it comes to managing validators efficiently, instead of manually adding or removing validators from the validator set.

IBFT has been implemented in several blockchains. Sample implementations include Quorum and Celo.



A Quorum implementation is available at the following link:

<https://github.com/jpmorganchase/quorum>

IBFT, with some variations, has also been implemented in the Celo blockchain, which is available at:

<https://github.com/celo-org/celo-blockchain>

Several other algorithms have been inspired by PBFT and have emerged as a result of deep interest in blockchain research. One such algorithm is

Tendermint.

Tendermint

Tendermint is another variant of PBFT. It was inspired by both the DLS and PBFT protocols. Tendermint also makes use of the SMR approach to providing consensus. As we saw before, state machine replication is a mechanism that allows synchronization between replicas/nodes of the network.

Traditionally, a consensus mechanism used to run with a small number of participants and thus performance and scalability was not a big concern. However, with the advent of blockchain, there is a need to develop algorithms that can work on wide area networks and in asynchronous environments. Research into these areas of distributed computing is not new and especially now, due to the rise of cryptocurrencies and blockchain, the interest in these research topics has grown significantly in the last few years.



More details on the DLS algorithm can be found in the original paper:

<https://groups.csail.mit.edu/tds/papers/Lynch/jacm88.pdf>

Dwork, C., Lynch, N. and Stockmeyer, L., 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), pp.288-323



The preceding paper proposes a consensus mechanism named after its authors (DLS: Dwork, Lynch, Stockmeyer). This protocol consists of two initial rounds. This process of rounds with appropriate message exchange ensures that agreement is eventually achieved on a proposed or default value. As long as the number of nodes in the system is more than $3F$, this protocol achieves consensus.

The Tendermint protocol also works by running rounds. In each round, a leader is elected, which proposes the next block. Also note that in Tendermint, the round change or view-change process is part of the normal

operation, as opposed to PBFT, where view-change only occurs in the event of errors, that is, a suspected faulty leader. Tendermint works similarly to PBFT, where three phases are required to achieve a decision. Once a round is complete, a new round starts with three phases and terminates when a decision is reached. A key innovation in Tendermint is the design of a new termination mechanism. As opposed to other PBFT-like protocols, Tendermint has developed a more straightforward mechanism, which is similar to PBFT-style normal operation. Instead of having two sub-protocols for normal mode and view-change mode (recovery in event of errors), Tendermint terminates without any additional communication costs.

The model under which Tendermint is supposed to run will now be presented. We saw earlier, in the introduction, that each consensus model is studied and developed under a system model with some assumptions about the system. Tendermint is also designed with a system model in mind. Now we'll define and introduce each element of the system model.

- **Processes:** A process is the fundamental key participant of the protocol. It is also called a replica (in PBFT traditional literature), a node, or merely a process. Processes can be correct or honest. Processes can also be faulty or Byzantine. Each process possesses some voting power. Also, note that processes are not necessarily connected directly; they are only required to connect loosely or just with their immediate subset of processes/nodes. Processes have a local timer that they use to measure timeout.
- **Network model:** The network model is a network of processes that communicate using messages. In other words, the network model is a set of processes that communicate using message passing. Particularly, the gossip protocol is used for communication between processes. The standard assumption of $N \geq 3F + 1$ BFT is also taken into consideration. This means that the protocol operates correctly as long as the number of nodes in the network is more than $3F$, where F is the number of faulty nodes. This implies that, at a minimum, there have to be four nodes in a network to tolerate Byzantine faults.
- **Timing assumptions:** Under the network model, Tendermint assumes a partially synchronous network. This means that there is an unknown

bound on the communication delay, but it only applies after an unknown instance of time called global stabilization time or GST.

- **Security and cryptography:** It is assumed that the public key cryptography used in the system is secure and the impersonation or spoofing of accounts/identities is not possible. The messages on the network are authenticated and verified via digital signatures. The protocol ignores any messages with an invalid digital signature.
- **State machine replication:** To achieve replication among the nodes, the standard SMR mechanism is used. One key observation that is fundamental to the protocol is that in SMR, it is ensured that all replicas on the network receive and process the same sequence of requests. As noted in the Tendermint paper, agreement and order are two properties that ensure that all requests are received by replicas and order ensures that the sequence in which the replicas have received requests is the same. Both of these requirements ensure total order in the system. Also, Tendermint ensures that requests themselves are valid and have been proposed by the clients. In other words, only valid transactions are accepted and executed on the network.

There are three fundamental consensus properties that Tendermint solved. As we discussed earlier in the chapter, generally in a consensus problem, there are safety and liveness properties that are required to be met. Similarly, in Tendermint, these safety and liveness properties consist of agreement, termination, and validity.

These properties are defined as follows:

- **Agreement:** No two correct processes decide on different values.
- **Termination:** All correct processes eventually decide on a value.
- **Validity:** A decided upon value is valid, that is, it satisfies the predefined predicate denoted `valid()`.

Now we'll describe how the algorithm works.

State transition in Tendermint is dependent on the messages received and timeouts. In other words, the state is changed in response to messages received by a processor or in the event of timeouts. The timeout mechanism

ensures liveness and prevents endless waiting. It is assumed that, eventually, after a period of asynchrony, there will be a round or communication period during which all processes can communicate in a timely fashion, which will ensure that processes eventually decide on a value.

The following diagram depicts the Tendermint protocol at a high level:

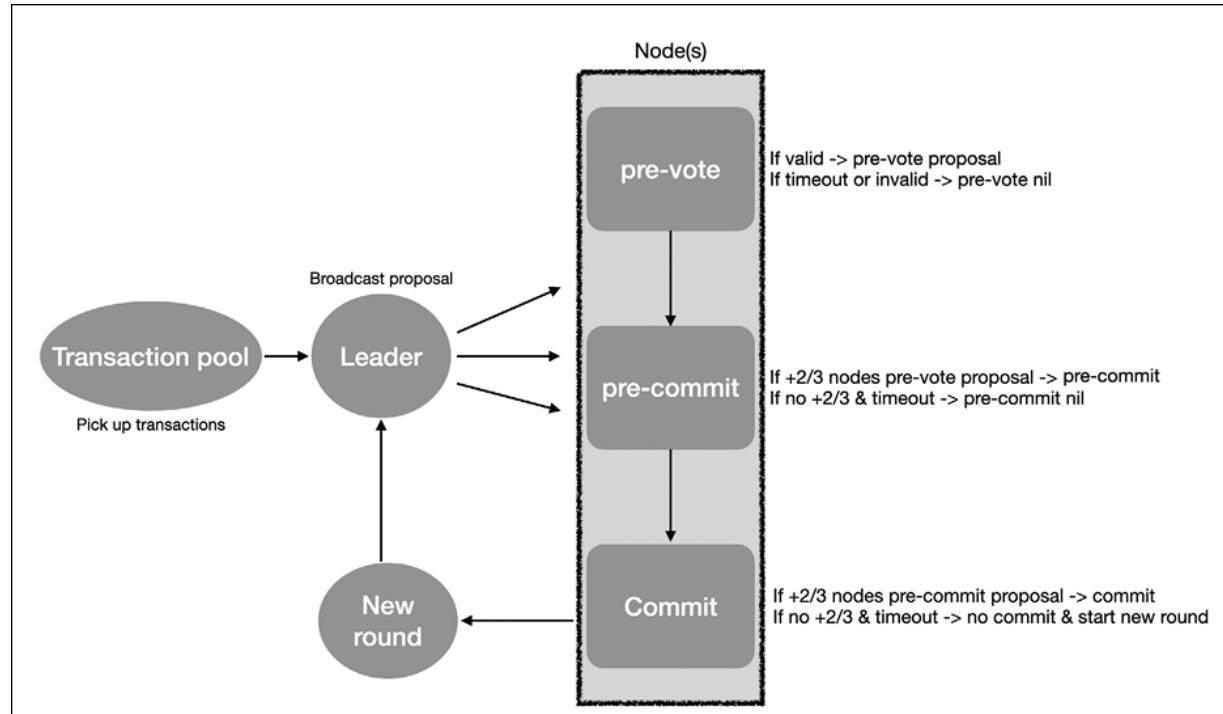


Figure 5.8: Tendermint high-level overview

Types of messages in Tendermint

There are three types of messages in Tendermint. We'll discuss each type individually here:

1. **Proposal**: As the name suggests, this is used by the leader of the current round to propose a value or block.
2. **Pre-vote**: This message is used to vote on a proposed value.
3. **Pre-commit**: This message is also used to vote on a proposed value.

These messages can be considered somewhat equivalent to PBFT's PRE-PREPARE, PREPARE, and COMMIT messages.



Note that in Tendermint, only the proposal message carries the original value and the other two messages, pre-vote and pre-commit, operate on a value identifier, representing the original proposal.

All of the aforementioned messages also have a corresponding timeout mechanism, which ensures that processes do not end up waiting indefinitely for some conditions to meet. If a processor cannot decide in an expected amount of time, it will time out and trigger a round change.

Each type of message has an associated timeout. As such, there are three timeouts in Tendermint, corresponding to each message type:

1. Timeout-propose
2. Timeout-prevote
3. Timeout-precommit

These timeout mechanisms prevent the algorithm from waiting infinitely for a condition to be met. They also ensure that processes progress through the rounds. A clever mechanism to increase timeout with every new round ensures that after reaching GST, eventually the communication between correct processes becomes reliable and a decision can be reached.

State variables

All processes in Tendermint maintain a set of variables, which helps with the execution of the algorithm. Each of these variables holds critical values, which ensure the correct execution of the algorithm.

These variables are listed and discussed as follows:

- **Step:** The step variable holds information about the current state of the algorithm, that is, the current state of the Tendermint state machine in the current round.

- **lockedValue**: The `lockedValue` variable stores the most recent value (with respect to a round number) for which a pre-commit message has been sent.
- **lockedRound**: The `lockedRound` variable contains information about the last round in which the process sent a non-nil `pre-commit` message. This is the round where a possible decision value has been locked. This means that if a proposal message and corresponding $2F + 1$ messages have been received for a value in a round, then, due to the reason that $2F + 1$ prevotes have already been received for this value, this is a possible decision value.
- **validValue**: The role of the `validValue` variable is to store the most recent possible decision value.
- **validRound**: The `validRound` variable is the last round in which `validValue` was updated.



`lockedValue`, `lockedRound`, `validValue`, and `validRound` are reset to the initial values every time a decision is reached.

Apart from the preceding variables, a process also stores the current consensus instance (called height in Tendermint), and the current round number. These variables are attached to every message. A process also stores an array of decisions. Tendermint assumes a sequence of consensus instances, one for each height.

How Tendermint works

Tendermint works in rounds and each round comprises phases: **propose**, **pre-vote**, **pre-commit**, and **commit**. In this section, we'll explore how Tendermint works:

1. Every round starts with a proposal value being proposed by a proposer. The proposer can propose any new value at the start of the first round for each height.

2. After the first round, any subsequent rounds will have the proposer, which proposes a new value only if there is no valid value present, that is, null. Otherwise the `validvalue`, that is, the possible decision value, is proposed, which has already been locked in a previous round. The proposal message also includes a value of valid round, which denotes the last round in which there was a valid value updated.
3. The proposal is accepted by a correct process only if:
 1. The proposed value is valid
 2. The process has not locked on a round
 3. Or, the process has a value locked
4. If the preceding conditions are met, then the correct process will accept the proposal and send a pre-vote message.
5. If the preceding conditions are not met, then the process will send a pre-vote message with a `nil` value.
6. In addition, there is also a timeout mechanism associated with the proposal phase, which initiates timeout if a process has not sent a pre-vote message in the current round or the timer expires in the proposal stage.
7. If a correct process receives a proposal message with a value and $2F + 1$ pre-vote messages, then it sends the pre-commit message.
8. Otherwise, it sends out a `nil` pre-commit.
9. A timeout mechanism associated with the pre-commit will initialize if the associated timer expires or if the process has not sent a pre-commit message after receiving a proposal message and $2F + 1$ pre-commit messages.
10. A correct process decides on a value if it has received the proposal message in some round and $2F + 1$ pre-commit messages for the ID of the proposed value.
11. There is also an associated timeout mechanism with this step, which ensures that the processor does not wait indefinitely to receive $2F + 1$ messages. If the timer expires before the processor can decide, the processor starts the next round.

- When a processor eventually decides, it triggers the next consensus instance for the next block proposal and the entire process of proposal, pre-vote, and pre-commit starts again.

The process is as simple as the flow shown here:

Proposal —> Pre-vote —> Pre-commit

We can visualize the protocol flow with the diagram shown here:

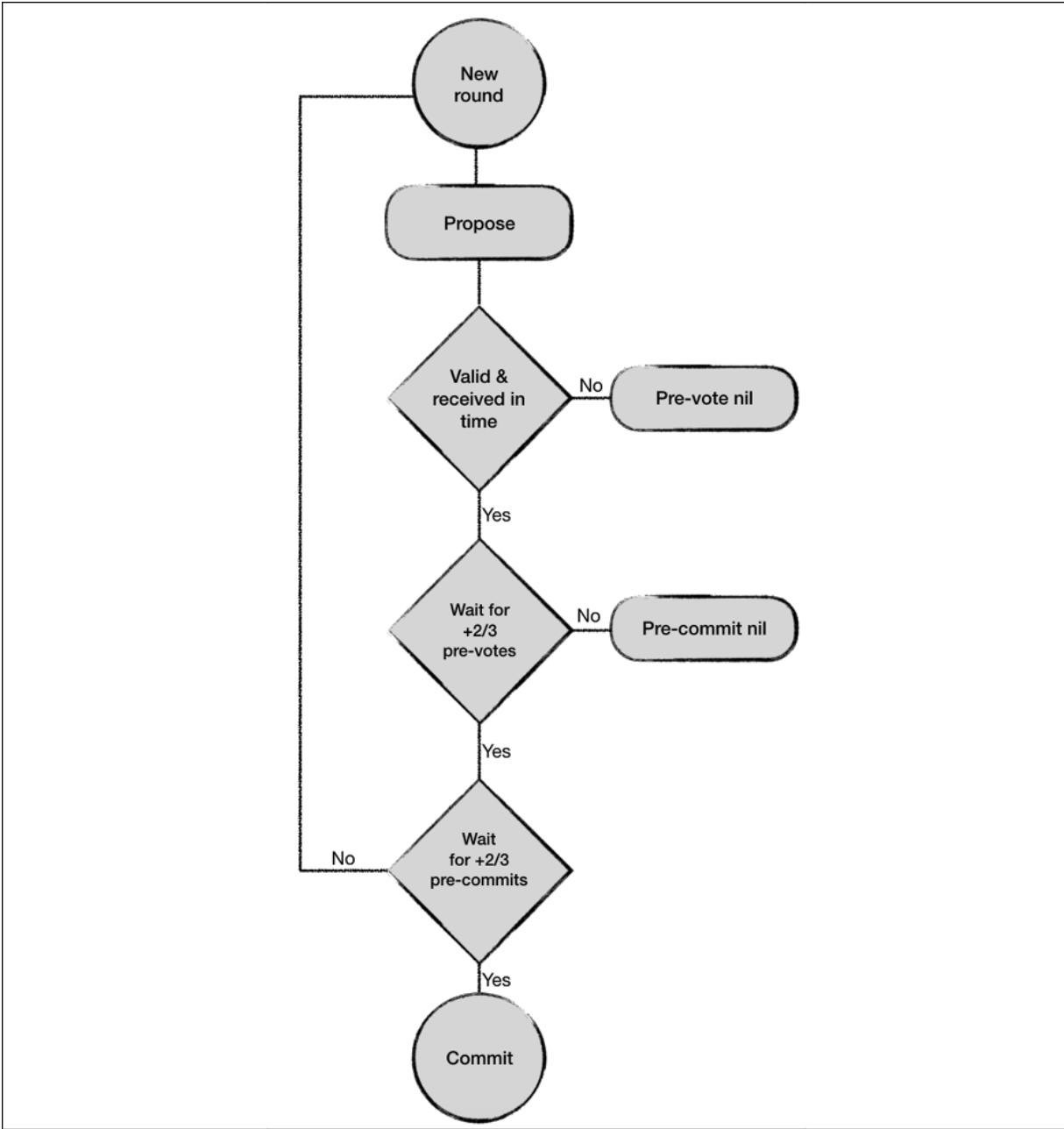


Figure 5.9: Tendermint flowchart

Now we'll discuss how the **termination** mechanism works in Tendermint. A new termination mechanism has been introduced in Tendermint. For this purpose, there are two variables, namely `validValue` and `validRound`, which are used by the proposal message. Both of these variables are updated by a correct process when the process receives a valid proposal message and subsequent/corresponding $2F + 1$ pre-vote messages.

This process works by utilizing the gossip protocol, which ensures that if a correct process has locked a value in a round, all correct processes will then update their `validValue` and `validRound` variables with the locked values by the end of the round during which they have been locked. The key idea is that once these values have been locked by a correct processor, they will be propagated to other nodes within the same round and each processor will know the locked value and round, that is, the valid values. Now, when the next proposal is made, the locked values will be picked up by the proposer, which have already been locked as a result of the valid proposal and corresponding $2F + 1$ pre-vote messages. This way, it can be ensured that the value that processes eventually decide upon is acceptable as specified by the validity conditions described above.

Tendermint implementation

Tendermint is developed in Go. It can be implemented in various different scenarios. The source code is available at

<https://github.com/tendermint/tendermint>.

It is released as Tendermint Core, which is a language-agnostic programming middleware that takes a state transition machine and replicates it on many machines.

Tendermint Core is available here:

<https://tendermint.com/core/>

The algorithms discussed so far are variants of traditional Byzantine fault-tolerant algorithms. Now, we'll introduce the protocols that are specifically developed for blockchain protocols. A prime example is, of course, PoW, which was first introduced with Bitcoin.

Nakamoto consensus

Nakamoto consensus, or **PoW**, was first introduced with Bitcoin in 2009. Since then, it has stood the test of time and is the longest-running blockchain network. This test of time is a testament to the efficacy of the PoW consensus mechanism. Now, we will explore how PoW works.

At a fundamental level, the PoW mechanism is designed to mitigate Sybil attacks, which facilitates consensus and the security of the network.



A Sybil attack is a type of attack that aims to gain a majority influence on the network to control the network. Once a network is under the control of an adversary, any malicious activity could occur. A Sybil attack is usually conducted by a node generating and using multiple identities on the network. If there are enough multiple identities held by an entity, then that entity can influence the network by skewing majority-based network decisions. The majority in this case is held by the adversary.

It is quite easy to obtain multiple identities and try to influence the network. However, in Bitcoin, due to the hashing power requirements, this attack is mitigated.

How PoW works

In a nutshell, PoW works as follows:

- PoW makes use of hash puzzles.
- A node proposes a block has to find a nonce such that $H(\text{nonce} \parallel \text{previous hash} \parallel \text{Tx} \parallel \text{Tx} \parallel \dots \parallel \text{Tx}) < \text{Threshold value.}$

The process can be summarized as follows:

- New transactions are broadcast to all nodes on the network.
- Each node collects the transactions into a candidate block.
- Miners propose new blocks.
- Miners concatenate and hash with the header of the previous block.
- The resultant hash is checked against the target value, that is, the network difficulty target value.
- If the resultant hash is less than the threshold value, then PoW is solved, otherwise, the nonce is incremented and the node tries again. This process continues until a resultant hash is found that is less than the threshold value.

We can visualize how PoW works with the diagram shown here:

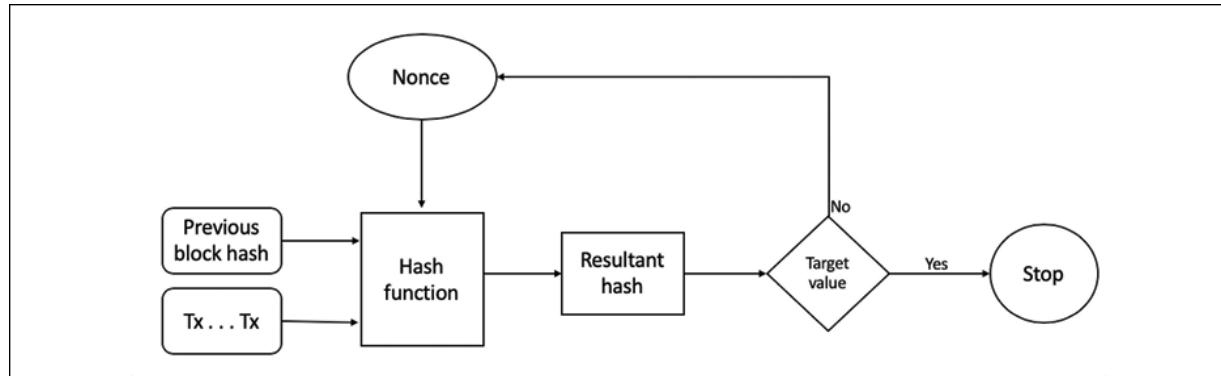


Figure 5.10: PoW diagram

PoW as a solution to the Byzantine generals problem

The original posting on this by Satoshi Nakamoto can be found here:

<https://satoshi.nakamotoinstitute.org/emails/cryptography/11/>

The key idea behind PoW as a solution to the Byzantine generals problem is that all honest generals (miners in the Bitcoin world) achieve agreement on the same state (decision value). As long as honest participants control the majority of the network, PoW solves the Byzantine generals problem. Note that this is a probabilistic solution and not deterministic.

Nakamoto versus traditional consensus

Nakamoto consensus was the first of its kind. It solved the consensus problem, which had been traditionally solved using pre-Bitcoin protocols like PBFT. A natural question that arises is, are there any similarities between the Nakamoto world and traditional distributed consensus? The short answer is yes, because we can map the properties of PoW consensus to traditional Byzantine consensus. This mapping is useful to understand what properties of traditional consensus can be applied to the Nakamoto world and vice versa.

In traditional consensus algorithms, we have **agreement**, **validity**, and **liveness** properties, which can be mapped to Nakamoto-specific properties of the **common prefix**, **chain quality**, and **chain growth** properties respectively.

The **common prefix** property means that the blockchain hosted by honest nodes will share the same large common prefix. If that is not the case, then the **agreement** property of the protocol cannot be guaranteed, meaning that the processors will not be able to decide and agree on the same value.

The **chain quality** property means that the blockchain contains a certain required level of correct blocks created by honest nodes (miners). If chain quality is compromised, then the **validity** property of the protocol cannot be guaranteed. This means that there is a possibility that a value will be decided that is not proposed by a correct process, resulting in safety violation.

The **chain growth** property simply means that new correct blocks are continuously added to the blockchain. If chain growth is impacted, then the **liveness** property of the protocol cannot be guaranteed. This means that the system can deadlock or fail to decide on a value.



PoW chain quality and common prefix properties are introduced and discussed in the following paper:

Garay, J., Kiayias, A. and Leonardos, N., 2015, April. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 281-310). Springer, Berlin, Heidelberg.

<https://eprint.iacr.org/2014/765.pdf>

Variants of PoW

There are two main variants of PoW algorithms, based on the type of hardware used for their processing.

CPU-bound PoW

CPU-bound PoW refers to a type of PoW where the processing required to find the solution to the cryptographic hash puzzle is directly proportional to the calculation speed of the CPU or hardware such as ASICs. Because ASICs have dominated the Bitcoin PoW and provide somewhat undue advantage to the miners who can afford to use ASICs, this CPU-bound PoW is seen as shifting toward centralization. Moreover, mining pools with extraordinary hashing power can shift the balance of power towards them. Therefore, memory-bound PoW algorithms have been introduced, which are ASIC-resistant and are based on memory-oriented design instead of CPU.

Memory-bound PoW

Memory-bound PoW algorithms rely on system RAM to provide PoW. Here, the performance is bound by the access speed of the memory or the size of the memory. This reliance on memory also makes these PoW algorithms ASIC-resistant. Equihash is one of the most prominent memory-bound PoW algorithms.

PoW consumes tremendous amounts of energy; as such, there are several alternatives suggested by researchers. One of the first alternatives proposed is PoS.

Proof of stake (PoS)

PoS is an energy-efficient alternative to the PoW algorithm, which consumes an enormous amount of energy. PoS was first used in Peercoin, and now, prominent blockchains such as EOS, NxT, Steem, and Tezos are using PoS algorithms. Ethereum, with its Serenity release, will soon transition to a PoS-based consensus mechanism.

The stake represents the number of coins (money) in the consensus protocol staked by a blockchain participant. The key idea is that if someone has a stake in the system, then they will not try to sabotage the system. Generally speaking, the chance of proposing the next block is directly proportional to the value staked by the participant. However, there are a few intricate variations, which we will discuss shortly. There are different variations of

PoS, such as chain-based PoS, committee-based PoS, BFT-based PoS, delegated PoS, leased PoS, and master node PoS.

In PoS systems, there is no concept of mining in the traditional Nakamoto consensus sense. However, the process related to earning revenue is sometimes called virtual mining. A PoS miner is called either a validator, minter, or stakeholder.

The right to win the next proposer role is usually assigned randomly. Proposers are rewarded either with transaction fees or block rewards. Similar to PoW, control over the majority of the network in the form of the control of a large portion of the stake is required to attack and control the network.

PoS mechanisms generally select a stakeholder and grant appropriate rights to it based on their staked assets. The stake calculation is application-specific, but generally, is based on balance, deposit value, or voting among the validators. Once the stake is calculated, and a stakeholder is selected to propose a block, the block proposed by the proposer is readily accepted. The probability of selection increases with a higher stake. In other words, the higher the stake, the better the chances of winning the right to propose the next block.

How PoS works

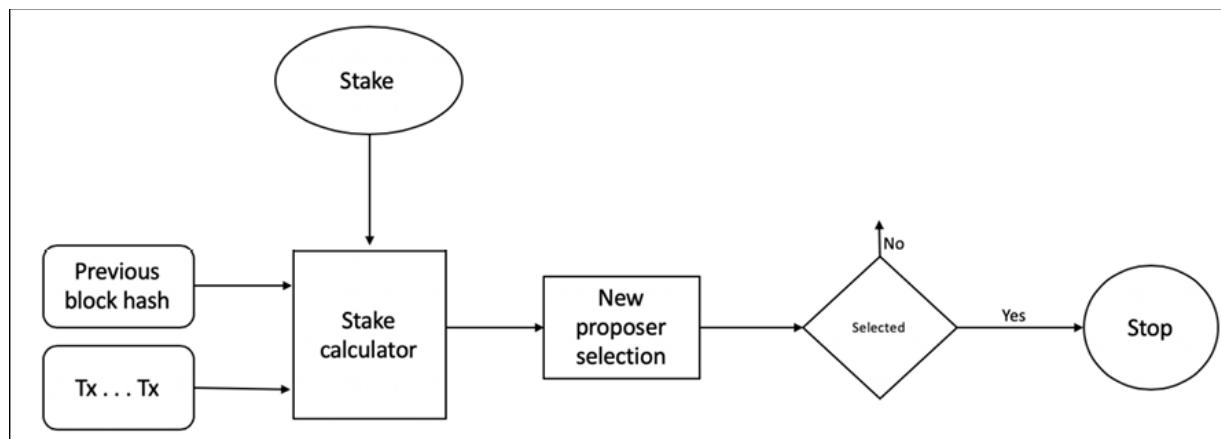


Figure 5.11: PoS diagram

In the preceding diagram, a generic PoS mechanism is shown where a stake calculator function is used to calculate the amount of staked funds and, based on that, select a new proposer.

Types of PoS

There are several types of PoS algorithm.

Chain-based PoS

This mechanism is very similar to PoW. The only change from the PoW mechanism is the block generation method. A block is generated in two steps by following a simple protocol:

- Transactions are picked up from the memory pool and a candidate block is created.
 - A clock is set up with a constant tick interval, and at each clock tick, whether the hash of the block header concatenated with the clock time is less than the product of the target value and stake value is checked.
- This process can be shown in a simple formula:

$$\text{Hash}(B \parallel \text{clock time}) < \text{target} \times \text{stake value}$$

The stake value is dependent on the way the algorithm is designed. In some systems, it is directly proportional to the amount of stake, and in others, it is based on the amount of time the stake has been held by the participant (also called coinage). The target is the mining difficulty per unit of the value of stake.

This mechanism still uses hashing puzzles, as in PoW. But, instead of competing to solve the hashing puzzle by consuming a high amount of electricity and specialized hardware, the hashing puzzle in PoS is solved at regular intervals based on the clock tick. A hashing puzzle becomes easier to solve if the stake value of the minter is high.

Peercoin was the first blockchain to implement PoS.





Nxt (<https://www.jelurida.com/nxt>) and Peercoin (<https://www.peercoin.net>) are two examples of blockchains where chain-based PoS is implemented.

Committee-based PoS

In this scheme, a group of stakeholders is chosen randomly, usually by using a **verifiable random function (VRF)**. This VRF, once invoked, produces a random set of stakeholders based on their stake and the current state of the blockchain. The chosen group of stakeholders becomes responsible for proposing blocks in sequential order.

This mechanism is used in the Ouroboros PoS consensus mechanism, which is used in Cardano. More details on this are available here:



<https://www.cardano.org/en/ouroboros/>

VRFs were introduced in *Chapter 4, Public Key Cryptography*. More information on VRF can be found here:

<https://tools.ietf.org/id/draft-goldbe-vrf-01.html>.

Delegated PoS

DPoS is very similar to committee-based PoS, but with one crucial difference. Instead of using a random function to derive the group of stakeholders, the group is chosen by stake delegation. The group selected is a fixed number of minters that create blocks in a round-robin fashion. Delegates are chosen via voting by network users. Votes are proportional to the amount of the stake that participants have on the network. This technique is used in Lisk, Cosmos, and EOS. DPoS is not decentralized as a small number of known users are made responsible for proposing and generating blocks.

HotStuff

HotStuff is the latest class of BFT protocol with a number of optimizations. There are several changes in HotStuff that make it a different and, in some

ways, better protocol than traditional PBFT. HotStuff was introduced by VMware Research in 2018. Later, it was presented at the Symposium on Principles of Distributed Computing.



The paper reference is as follows:

Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G. and Abraham, I., 2019, July. HotStuff: BFT consensus with linearity and responsiveness. *In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (pp. 347-356). ACM.

The paper can be downloaded from:

https://dl.acm.org/ft_gateway.cfm?id=3331591&type=pdf

There are three key properties that HotStuff has addressed. These properties are listed as follows:

Linear view change

Linear view change results in reduced communication complexity. It is achieved by the algorithm where after GST is reached, a correct designated leader will send only **O(n) authenticators (either a partial signature or signature)** to reach consensus. In the worst case, where leaders fail successively, the communication cost is $\mathcal{O}(n^2)$ —a quadratic. In simpler words, quadratic complexity means that the performance of the algorithm is proportional to the squared size of the input.

Optimistic responsiveness

Optimistic responsiveness ensures that any correct leader after GST is reached only requires the first $N - F$ responses to ensure progress.

Chain quality

This property ensures fairness and liveness in the system by allowing fast and frequent leader rotation.



All these properties together have been addressed for the first time in the HotStuff protocol.

Another innovation in HotStuff is the separation of safety and liveness mechanisms. The separation of concerns allows better modularity, cleaner architecture, and control over the development of these features independently. Safety is ensured through voting and commit rules for participant nodes in the network. Liveness, on the other hand, is the responsibility of a separate module, called **Pacemaker**, which ensures a new, correct, and unique leader is elected.

In comparison with traditional PBFT, HotStuff has introduced several changes, which result in improved performance:

- PBFT-style protocols work using a mesh communication topology, where each message is required to be broadcast to other nodes on the network. In HotStuff, the communication has been changed to the star topology, which means that nodes do not communicate with each other directly, but all consensus messages are collected by a leader and then broadcast to other nodes. This immediately results in reduced communication complexity.

A question arises here: what happens if the leader somehow is corrupt or compromised? This issue is solved by the same BFT tolerance rules where, if a leader proposes a malicious block, it will be rejected by other honest validators and a new leader will be chosen. This scenario can slow down the network for a limited time (until a new honest leader is chosen), but eventually (as long as a majority of the network is honest), an honest leader will be chosen, which will propose a valid block. Also, for further protection, usually, the leader role is frequently (usually, with each block) rotated between validators, which can neutralize any malicious attacks targeting the network. This property ensures **fairness**, which helps to achieve **chain quality**, introduced previously.



However, note that there is one problem in this scheme where, if the leader becomes too overloaded, then the processing may become slow, impacting the whole network.

Mesh and star topologies can be visualized in the following diagram:

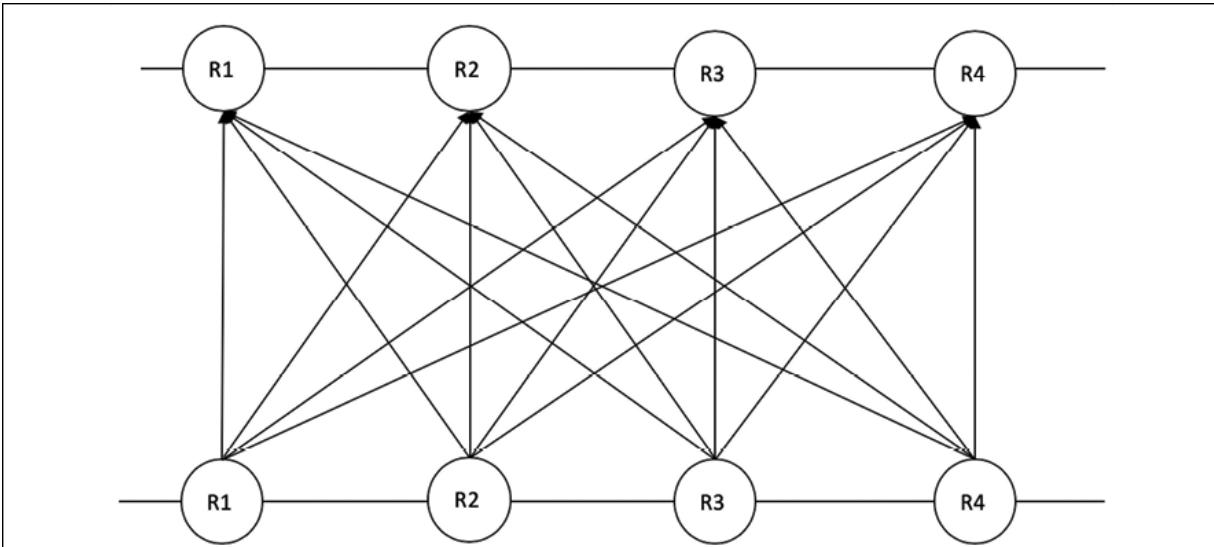


Figure 5.12: Mesh topology

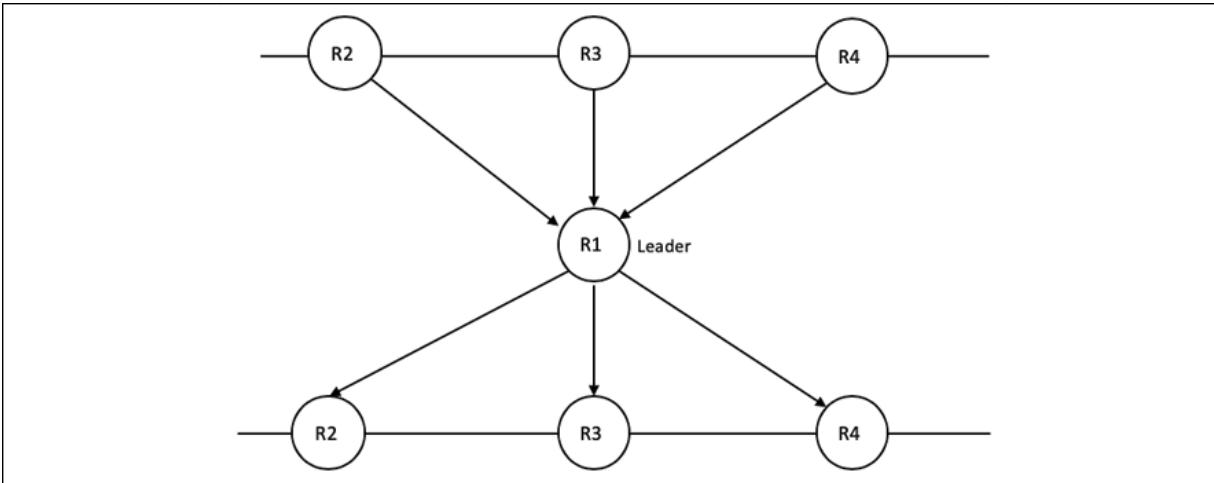


Figure 5.13: Star topology

- PBFT has two main sub-protocols, namely normal mode and view-change mode. View-change mode is triggered when a leader is suspected of being faulty. This approach does work to provide a liveness guarantee to PBFT but increases communication complexity significantly. HotStuff addresses this by merging the view-change process with normal mode. This means that nodes can switch to a new

view directly without waiting for a threshold of view-change messages to be received by other nodes. In PBFT, nodes wait for $2F+1$ messages before the view change can occur, but in HotStuff, view change can occur directly without requiring a new sub-protocol. Instead, the checking of the threshold of the messages to change the view becomes part of the normal view.

Just like PBFT, HotStuff also solves the SMR problem. Now, we'll describe how this protocol works. As we saw earlier, consensus algorithms are described and work under a system model. The model and relevant assumptions under which HotStuff works is introduced in the next section.

Model

The system is based on a standard BFT assumption of $N=3F+1$ nodes in the system, where F is a faulty node and N is the number of nodes in the network. Nodes communicate with each other via point-to-point message-passing, utilizing reliable and authenticated communication links. The network is supposed to be partially synchronous.



If you are unfamiliar with this terminology, we discussed these topics in the *Analysis and design* section of this chapter.

HotStuff makes use of threshold signatures where a single public key is used by all nodes, but a unique private key is used by each replica. The use of threshold signatures results in the decreased communication complexity of the protocol. In addition, cryptographic hash functions are used to provide unique identifiers for messages.



We discussed threshold signatures in *Chapter 4, Public Key Cryptography*. You can refer to the details there if required.

How HotStuff works

HotStuff works in phases, namely the **prepare** phase, **pre-commit** phase, **commit** phase, and **decide** phase.

We'll now introduce these phases:



A common term that we will see in the following section is quorum certificate. It is a data structure that represents a collection of signatures produced by $N - F$ replicas to demonstrate that the required threshold of messages has been achieved. In other words, it is simply a set of votes from $N - F$ nodes.

Prepare:

Once a new leader has collected **new-view** messages from $N - F$ nodes, the protocol for the new leader starts. The leader collects and processes these messages to figure out the latest branch in which the highest quorum certificate of prepare messages was formed.

Pre-commit:

As soon as a leader receives $N - F$ prepare votes, it creates a quorum certificate called "prepare quorum certificate." This "prepare quorum certificate" is broadcast to other nodes as a **PRE-COMMIT** message. When a replica receives the **PRE-COMMIT** message, it responds with a pre-commit vote. The quorum certificate is the indication that the required threshold of nodes has confirmed the request.

Commit:

When the leader receives $N - F$ pre-commit votes, it creates a **PRE-COMMIT** quorum certificate and broadcasts it to other nodes as the **COMMIT** message. When replicas receive this **COMMIT** message, they respond with their commit vote. At this stage, replicas lock the **PRE-COMMIT** quorum certificate to ensure the safety of the algorithm even if view change occurs.

Decide:

When the leader receives $N - F$ commit votes, it creates a **COMMIT** quorum certificate. This **COMMIT** quorum certificate is broadcast to other nodes in the **DECIDE** message. When replicas receive this **DECIDE** message, replicas execute the request, because this message contains an already committed certificate/value. Once the state transition occurs as a result of the **DECIDE** message being processed by a replica, the new view starts.

This algorithm can be visualized in the following diagram:

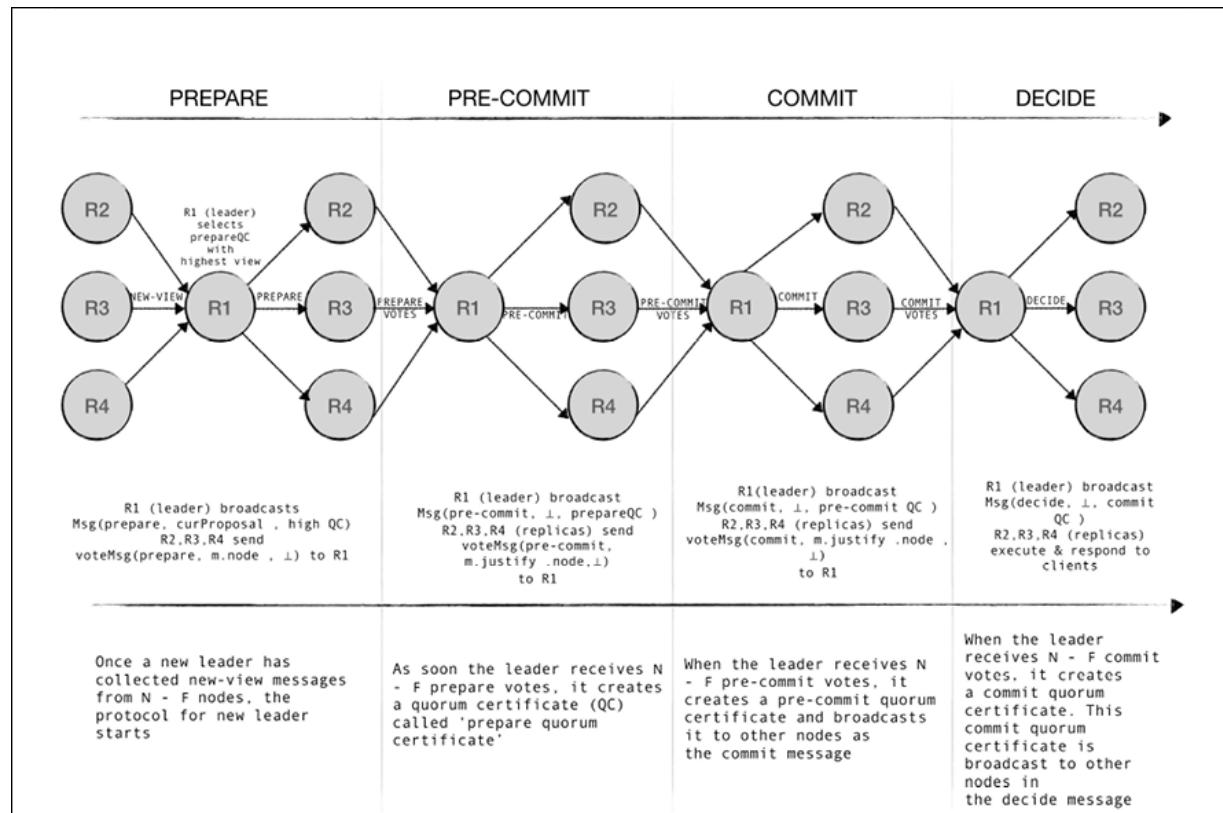


Figure 5.14: HotStuff protocol



HotStuff, with some variations, is also used in LibraBFT, which is a distributed database designed to support a global currency proposed by Facebook.

More details on this protocol are available here:

<https://libra.org/>

HotStuff guarantees **liveness** (progress) by using **Pacemaker**, which ensures progress after GST within a bounded time interval. This component has two elements:

- There are time intervals during which all replicas stay at a height for sufficiently long periods. This property can be achieved by progressively increasing the time until progress is made (a decision is made).
- A unique and correct leader is elected for the height. New leaders can be elected deterministically by using a rotating leader scheme or pseudo-random functions.

Safety in HotStuff is guaranteed by voting and relevant commit rules. **Liveness** and **safety** are separate mechanisms that allow independent development, modularity, and separation of concerns.

HotStuff is a simple yet powerful protocol that provides linearity and responsiveness properties. It allows consensus without any additional latency, at the actual speed of the network. Moreover, it is a protocol that manifests linear communication complexity, thus reducing the cost of communication compared to PBFT-style protocols. It is also a framework in which other protocols, such as DLS, PBFT, and Tendermint can be expressed.

In this section, we have explored various consensus algorithms in detail. We discussed **pre-Bitcoin** distributed consensus algorithms and **post-Bitcoin**, Proof of Work (PoW)-style algorithms. Now, a question arises naturally: with the availability of all these different algorithms, which algorithm is best, and which one should you choose for a particular use case? We'll try to answer this question in the next section.

Choosing an algorithm

Choosing a consensus algorithm depends on several factors. It is not only use case-dependent, but some trade-offs may also have to be made to create

a system that meets all the requirements without compromising the core safety and liveness properties of the system.

We will discuss some of the main factors that can influence the choice of consensus algorithm. Note that these factors are different from the core safety and liveness properties discussed earlier, which are the fundamental requirements needed to be fulfilled by any consensus mechanism. Other factors that we are going to introduce here are use case-specific and impact the choice of consensus algorithm.

These factors include, but are not limited to finality, speed, performance, and scalability.

Finality

Finality refers to a concept where once a transaction has been completed, it cannot be reverted. In other words, if a transaction has been committed to the blockchain, it won't be revoked or rolled back. This feature is especially important in financial networks, where once a transaction has gone through, the consumer can be confident that the transaction is irrevocable and final. There are two types of finality, **probabilistic** and **deterministic**.

Probabilistic finality, as the name suggests, provides a probabilistic guarantee of finality. The assurance that a transaction, once committed to the blockchain, cannot be rolled back builds over time instead of immediately. For example, in Nakamoto consensus, the probability that a transaction will not be rolled back increases as the number of blocks after the transaction commits increases. As the chain grows, the block containing the transaction goes deeper, which increasingly ensures that the transaction won't be rolled back. While this method worked and stood the test of time for many years, it is quite slow. For example, in the Bitcoin network, users usually have to wait for six blocks, which is equivalent to an hour, to get the right level of confidence that a transaction is final. This type of finality might be acceptable in public blockchains. Still, such a delay is not acceptable in financial transactions in a consortium blockchain. In consortium networks, we need immediate finality.

Deterministic finality or immediate finality provides an absolute finality guarantee for a transaction as soon as it is committed in a block. There are no forks or rollbacks, which could result in a transaction rollback. The confidence level of transaction finality is 100 percent as soon as the block that contains the transaction is finalized. This type of finality is provided by fault-tolerant algorithms such as PBFT.

Speed, performance, and scalability

Performance is a significant factor that impacts the choice of consensus algorithms. PoW chains are slower than BFT-based chains. If performance is a crucial requirement, then it is advisable to use voting-based algorithms for permissioned blockchains such as PBFT, which will provide better performance in terms of quicker transaction processing. There is, however, a caveat that needs to be kept in mind here—PBFT-type blockchains do not scale well but provide better performance. On the other hand, PoW-type chains can scale well but are quite slow and do not meet the enterprise-grade performance requirements.

Performance, scalability, and other challenges will be discussed in detail in *Chapter 21, Scalability and Other Challenges*.

Summary

In this chapter, we explained some of the most prominent protocols in blockchain and traditional distributed system consensus. We covered several algorithms, including Proof of Work, Proof of Stake, traditional BFT protocols, and the latest protocols, such as HotStuff. Distributed consensus is a very ripe area of research and academics and industry researchers are participating in this exciting subject. It is also a deep and vast area of research, therefore, it is impossible to cover every protocol and all dimensions of this gripping discipline in a single chapter. Nevertheless,

the protocols and ideas presented in this chapter provide solid ground for further research and more in-depth exploration.

In the next chapter, we'll introduce Bitcoin, which was the first blockchain and cryptocurrency, invented in 2009.

6

Introducing Bitcoin

Bitcoin was the first application of blockchain technology, and has started a revolution with the introduction of the very first fully decentralized digital currency. It has proven to be remarkably secure and valuable as a digital currency; however, it is quite unstable and highly volatile. The invention of Bitcoin has also sparked a great interest in academic and industrial research and opened up many new research areas. In this chapter, we shall introduce Bitcoin in detail.

Specifically, in this chapter, we will focus our attention upon the fundamentals of Bitcoin, how transactions are constructed and used, transaction structures, addresses, accounts, and mining, over the following topic areas:

- Bitcoin—an overview
- Cryptographic keys
- Transactions
- Blockchain
- Mining

Let's begin with an overview of Bitcoin.

Bitcoin—an overview

Since its introduction in 2008 by Satoshi Nakamoto, **Bitcoin** has gained immense popularity and is currently the most successful digital currency in

the world, with billions of dollars invested in it. The current market cap at the time of writing for this currency is 168,923,803,898 USD.

Its popularity is also evident from the high number of users and investors, the increasing price of Bitcoin, daily news related to Bitcoin, and the many start-ups and companies that are offering Bitcoin-based online exchanges. It is now also traded as Bitcoin futures on the **Chicago Mercantile Exchange (CME)**.



Interested readers can read more about Bitcoin futures at
<http://www.cmegroup.com/trading/bitcoin-futures.html>.

It is built on decades of research in the field of cryptography, digital cash, and distributed computing. In the following section, a brief history is presented to provide the background required to understand the foundations behind the invention of Bitcoin.

Digital currencies have always been an active area of research for many decades. Early proposals to create digital cash go as far back as the early 1980s. In 1982, David Chaum, a computer scientist and cryptographer, proposed a scheme that used blind signatures to build an untraceable digital currency. This research was published in a research paper entitled *Blind signatures for untraceable payments*. We covered this history in *Chapter 1, Blockchain 101*, which readers can review if required.



Interested readers can read the original research paper in which David Chaum invented the cryptographic primitive of blind signatures at
<http://www.hit.bme.hu/~buttyan/courses/BMEVIHIM219/2009/Chaum.BlindSigForPayment.1982.PDF>.

In this scheme, a bank would issue digital money by signing a blind and random serial number presented to it by the user. The user could then use the digital token signed by the bank as currency. The limitation of this scheme was that the bank had to keep track of all used serial numbers. This

was a centralized system by design and required a trusted party such as a bank to operate.

Later on, in 1988, David Chaum et al. proposed a refined version named eCash that not only used blind signatures, but also some private identification data to craft a message that was then sent to the bank.



The original research paper for this is available at
[http://citeseerx.ist.psu.edu/viewdoc/summary?
doi=10.1.1.26.5759.](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.5759)

This scheme allowed the detection of double-spending, but did not prevent it. If the same token were used at two different locations, then the identity of the double-spender would be revealed. Also, this eCash scheme could only represent a fixed amount of money.

Adam Back, a cryptographer and current CEO of **Blockstream**, who is involved in Blockchain development, introduced **hashcash** in 1997. It was initially proposed to thwart email spam. The idea behind hashcash was to solve a computational puzzle that was easy to verify but comparatively difficult to compute. The idea was that for a single user and a single email, the extra computational effort was negligible, but someone sending a large number of spam emails would be discouraged as the time and resources required to run the spam campaign would increase prohibitively.

In 1998, Wei Dai, a computer engineer who used to work for Microsoft, proposed **b-money**, which introduced the idea of using **Proof of Work (PoW)** to create money. The term PoW emerged and got popular later with Bitcoin, but in Wei Dai's b-money, a scheme of creating money was introduced by providing a solution to a previously unsolved computational problem. This concept is similar to PoW, where the money is created by broadcasting the solution to a previously unsolved computational problem.



The original paper on b-money is available here:
<http://www.weidai.com/bmoney.txt>.

A significant weakness in the system was that an adversary with higher computational power could generate unsolicited money without allowing the network to adjust to an appropriate difficulty level. The system lacked details on the consensus mechanism between nodes and some security issues such as Sybil attacks were also not addressed. At the same time, Nick Szabo, a computer scientist, introduced the concept of BitGold, which was also based on the PoW mechanism but had the same problems as b-money, with the exception that the network difficulty level was adjustable. Tomas Sander and Amnon Ta-Shma at the International Computer Science Institute, Berkley, introduced an e-cash scheme in a research paper named *Auditable, Anonymous Electronic Cash* in 1999. This scheme, for the first time, used Merkle trees to represent coins and zero-knowledge proofs to prove the possession of coins.



The original research paper, *Auditable, Anonymous Electronic Cash*, is available at
<http://www.cs.tau.ac.il/~amnon/Papers/ST.crypto99.pdf>.

In this scheme, a central bank was required that kept a record of all used serial numbers. This scheme allowed users to be fully anonymous. This was a theoretical design that was not practical to implement due to inefficient proof mechanisms.

RPOW (Reusable Proof of Work) was introduced in 2004 by Hal Finney, a computer scientist, developer, and the first person to receive Bitcoin from Satoshi Nakamoto. It used the hashcash scheme by Adam Back as a proof of computational resources spent to create the money. This was also a central system that kept a central database to keep track of all used PoW tokens. This was an online system that used remote attestation, made possible by a trusted computing platform (TPM hardware).

All the previously mentioned schemes are intelligently designed but had weaknesses in one aspect or another. Specifically, all these schemes rely on a central server that is required to be trusted by the users.

Having covered some of the fundamentals of Bitcoin, let's talk about how Bitcoin started.

The beginnings of Bitcoin

In 2008, Bitcoin was introduced in a paper called *Bitcoin: A Peer-to-Peer Electronic Cash System*.



This paper is available at <https://Bitcoin.org/Bitcoin.pdf>.

Satoshi Nakamoto wrote the Bitcoin paper. The name of the author is believed to be a pseudonym, as the true identity of the inventor of Bitcoin is unknown and is the subject of much speculation. The first key idea introduced in the paper was of a purely peer-to-peer electronic cash that does not need an intermediary bank to transfer payments between peers.

Bitcoin is built on decades of research. Various ideas and techniques from cryptography and distributed computing such as Merkle trees, hash functions, and digital signatures were used to design Bitcoin. Other ideas such as BitGold, b-money, hashcash, and cryptographic time-stamping also provided some groundwork for the invention of Bitcoin. Ideas from many of these developments were ingeniously used in Bitcoin to create the first ever truly decentralized currency. Bitcoin solves a number of historically difficult problems related to electronic cash and distributed systems, including:

- The Byzantine generals problem
- The double-spending problem
- Sybil attacks

Bitcoin is an elegant solution to the **Byzantine generals problem** and the **double-spending problem**. We examined both of these concepts in *Chapter 1, Blockchain 101*.

A **Sybil attack** is a type of attack where a single adversary creates a large number of nodes with fake identities on the network, which are used to gain influence over the network. This attack is also prevented in Bitcoin by using PoW, where miners are required to consume a considerable amount of computing power to earn rewards. If fake nodes try to add fake blocks to the Bitcoin blockchain, they will be rejected because those blocks will not have the required amount of work, that is, the PoW, associated with them. For an adversary to add a fake block, they will have to work at the same difficulty level as the other honest miners and compete honestly; otherwise, their work (as long as they do not control more than 51% of the network) will always be rejected by the network due to the insufficient amount of work performed to create blocks.

Bitcoin is currently the most valuable cryptocurrency. The value of Bitcoin fluctuates quite heavily but has increased quite significantly over time, as shown in the following graph:

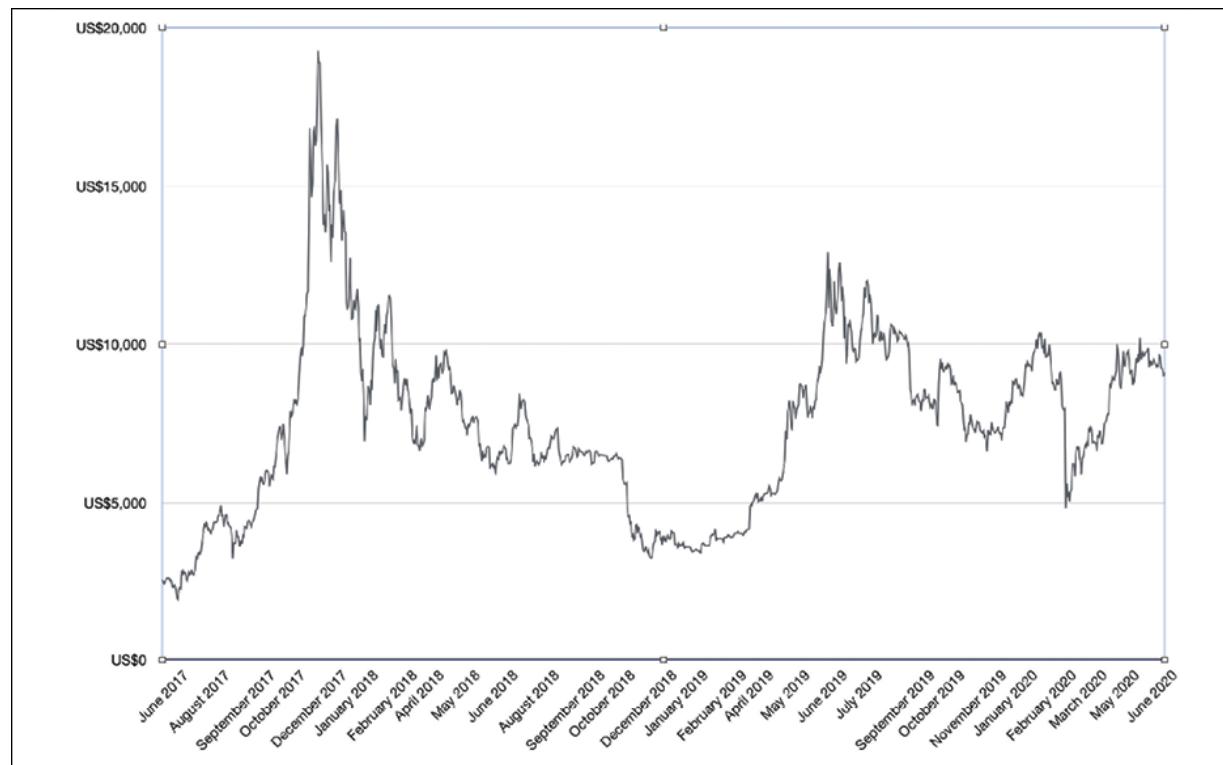


Figure 6.1: Bitcoin price trend since mid-2017

Before we talk about the specifics of Bitcoin, let's briefly discuss the philosophy behind it, go over the official definition of Bitcoin, and consider Bitcoin from a user's perspective before delving deeper into the topic in later sections.

Egalitarianism versus authoritarianism

For people with a libertarian ideology, Bitcoin is a platform that can be used instead of banks. However, some think that due to regulations, Bitcoin may become another institution that cannot be trusted. The original idea behind Bitcoin was to develop an e-cash system, which requires no trusted third party and where users can be anonymous. If regulations require checks like **Know Your Customer (KYC)** and detailed information about business transactions to facilitate the regulatory process, then it might be too much information to share. As a result, Bitcoin may not remain attractive to some entities.

The regulation of Bitcoin is a controversial subject. As much as it is a libertarian's dream, law enforcement agencies, governments, and banks are proposing various regulations to control it. One prime example is the **BitLicense**, issued by New York State's Department of Financial Services, which is a permit issued to businesses that perform activities related to virtual currencies. Due to the high cost and stringent regulatory requirements to obtain a BitLicense, many companies have withdrawn their services from New York.

There are now many initiatives being taken to regulate Bitcoin, cryptocurrencies, and related activities such as **Initial Coin Offerings (ICOs)**. The **Securities and Exchange Commission (SEC)** has recently announced that digital tokens, coins, and relevant activities such as ICOs fall under the category of securities. This announcement means that any digital currency trading platforms will now need to be registered with the SEC and all relevant securities' laws and regulations will be applicable to them. This situation impacted the Bitcoin price directly, and it fell almost 10% on the day this announcement was made. Bitcoin has even been made illegal in some countries.



Interested readers can read more about the regulation of Bitcoin and other relevant activities at
<https://cointelegraph.com/tags/Bitcoin-regulation>.

At this point, the question arises that if Bitcoin is under so much pressure from regulatory bodies, then how it has managed to grow so significantly? The simple answer is due to its decentralized and trustless nature. In this context, the term *trustless* refers to the distribution of trust between users, rather than a central entity. No single entity can control this network, and even if some entities try to enforce some regulations, they can only go so far because the network is owned collectively by its users instead of a single entity. It is also protected by its PoW mechanism, which thwarts any adversarial attacks on the network.

The growth of Bitcoin is also due to the so-called **network effect**. Also called **demand-side economies of scale**, it is a concept that means that the more users use the network, the more valuable it becomes. Over time, an exponential increase has been seen in Bitcoin network growth. This increase in the number of users is mostly driven by financial incentives. Also, the scarcity of Bitcoin and its built-in inflation control mechanism gives it value, as there are only 21 million Bitcoins that can ever be mined. Also, the miner reward halves every four years, which increases scarcity, and consequently, the demand increases even more.

Bitcoin definition

Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of a peer-to-peer network, protocols, and software that facilitates the creation and usage of the digital currency. Nodes in this peer-to-peer network talk to each other using the Bitcoin protocol.

Decentralization of currency was made possible for the first time with the invention of Bitcoin. Moreover, the double-spending problem was solved in an elegant and ingenious way in Bitcoin. The double-spending problem

arises when, for example, a user sends coins to two different users at the same time and they are verified independently as valid transactions. The double-spending problem is resolved in Bitcoin by using a distributed ledger (the blockchain) where every transaction is recorded permanently, and by implementing a transaction validation and confirmation mechanism. This process will be explained later in the chapter, in the *Mining* section.

Bitcoin—A user's perspective

In this section, we will see how the Bitcoin network looks from a user's point of view — how a transaction is made, how it propagates from the user to the network, and how transactions are verified and finally accumulated in blocks. We will look at the various actors and components of the Bitcoin network. Finally, some discussion on how all actors and components interact with each other to form the Bitcoin network will also be provided.

First, let's see that what the main components of a Bitcoin network are. Bitcoin is composed of the elements in the following list. We will further expand on these elements as we progress through the chapter:

- Digital keys
- Addresses
- Transactions
- Blockchain
- Miners
- The Bitcoin network
- Wallets (client software)

Now, we will see how a user utilizes the Bitcoin network. The following example will help you to understand how the Bitcoin network looks from the end user's perspective. We will see what actors and components are involved in a Bitcoin transaction. One of the most common operations is sending funds to someone else; therefore, in the following example, we will

see how a payment transaction can be sent from one user to another on the Bitcoin network.

Sending a payment

This example will demonstrate how money can be sent using the Bitcoin network from one user to another. There are several steps that are involved in this process. In this example, we are using the **Blockchain wallet** for mobile devices.

The steps are described as follows:

1. First, either the payment is requested from a user who sends their Bitcoin address to the sender via email or some other means such as SMS, chat applications, or in fact any appropriate communication mechanism. The sender can also initiate a transfer to send money to another user. In both cases, the address of the beneficiary is required. As an example, the Blockchain wallet is shown in the following screenshot, where a payment request is being created:

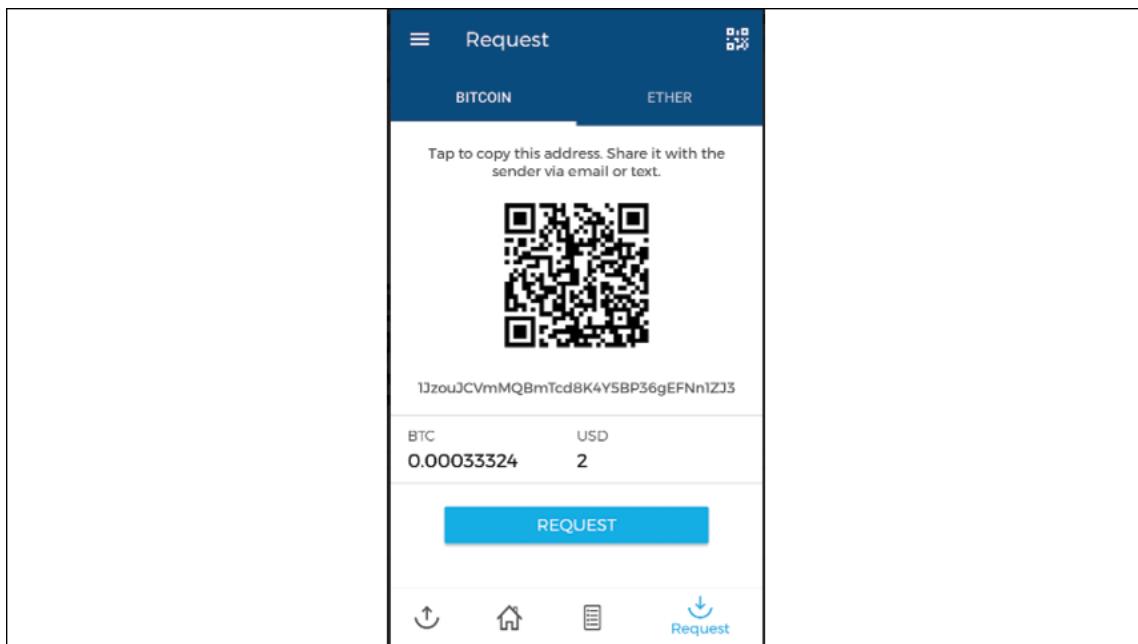


Figure 6.2: Bitcoin payment request (using the Blockchain wallet)

2. The sender either enters the receiver's address or scans the QR code that has the Bitcoin address, amount, and an optional description encoded in it. The wallet application recognizes this QR code and decodes it into something like:

```
"Please send <amount> BTC to address <receiver's Bitcoin ad
```

3. With actual values, this will look like the following:

```
"Please send 0.00033324 BTC to address 1JzouJCVmMQBmTcd8K4Y.
```

4. This is also shown in the following screenshot:



Figure 6.3: Bitcoin payment QR code

The QR code shown in the preceding screenshot is decoded to:

`bitcoin://1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3?`

`amount=0.00033324`, which can be opened as a URL in a Bitcoin wallet.

5. In the wallet application of the sender, this transaction is constructed by following some rules and is then broadcast to the Bitcoin network. This transaction is digitally signed using the private key of the sender before broadcasting it. How the transaction is created, digitally signed, broadcasted, validated, and added to the block will become clear in the following sections. From the user's point of view, once the QR code is

decoded, the transaction will appear, similar to what is shown in the following screenshot:



Figure 6.4: Sending BTC using the Blockchain wallet

Note that in the preceding screenshot, there are a number of fields such as **From**, **To**, **BTC**, and **Fee**. While other fields are self-explanatory, it's worth noting that the fee is calculated based on size of the transaction and a fee rate, which is a value that depends on the volume of the transactions in the network at that time. This is measured in *Satoshis per byte*. Bitcoin network fees ensure that your transaction will be included by miners in the block.

At times in the past, Bitcoin fees were so high that even for smaller transactions, a high fee was charged. This was due to the fact that miners are free to choose which transactions they pick to verify and add to a block, and they naturally select the ones with higher fees. A high number of users creating thousands of transactions also played a role in causing this situation of high fees because transactions were competing with each other to be picked up first and miners picked up the ones with the highest fees. This fee is also usually estimated and calculated by the Bitcoin wallet software automatically before sending the transaction. The higher the transaction fee, the greater the chances are that your transaction will be picked up as a priority and included in the block. This task is performed by the miners. Mining and miners are concepts that we will look at a bit later in this chapter in the section about *Mining*.

Once the transaction is sent, it will appear in the wallet software as shown in the following screenshot:

	SENT	0.00043946 BTC
		Value when sent: £2.00 Transaction fee: 0.00010622 BTC
	Description	What's this for?
To	1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1Zj3	
From	My Bitcoin Wallet	
Date	October 29, 2017 @ 4:47pm	
Status	Pending (0/3 Confirmations)	

Figure 6.5: Transaction sent

6. At this stage, the transaction has been constructed, signed, and sent out to the Bitcoin network. This transaction will be picked up by miners to be verified and included in the block. Also note that in the preceding screenshot, confirmation is pending for this transaction. These confirmations will start to appear as soon as the transaction is verified, included in the block, and mined. Also, the appropriate fee will be deducted from the original value to be transferred and will be paid to the miner who has included it in the block for mining.

The transaction flow described in the preceding list is illustrated in the following diagram, where a payment of 0.001267 BTC (approximately 11 USD) originated from the sender's address and has been paid to the receiver's address (starting with *1Jz*). The fee of 0.00010622 (approximately 95 cents) has also been deducted from the transaction as a mining fee:

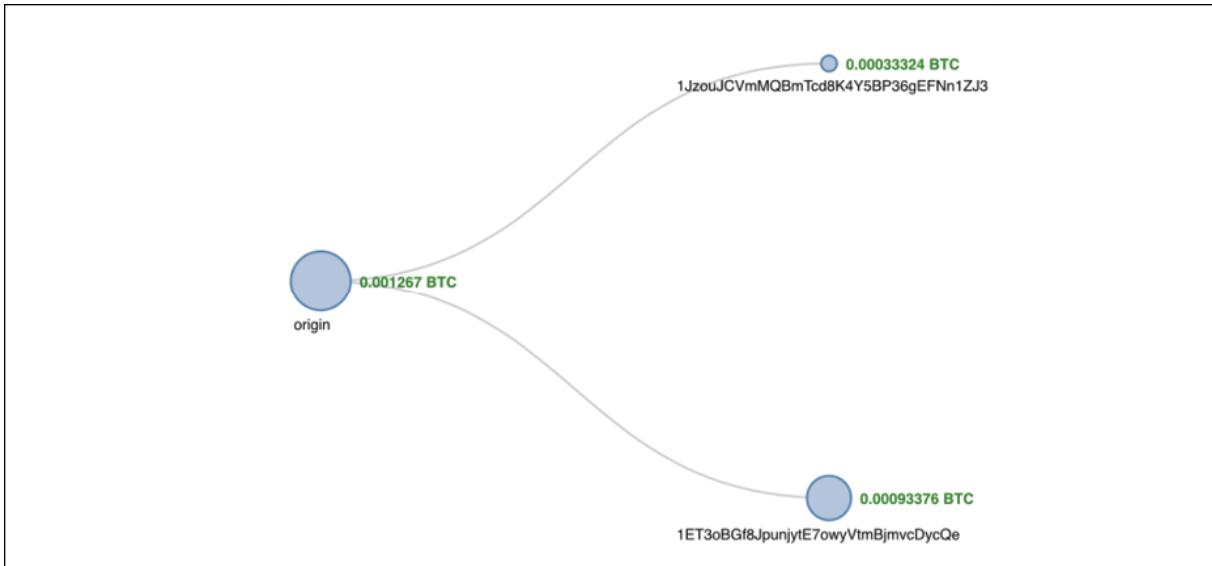


Figure 6.6: Transaction flow visualization (blockchain.info)

The preceding diagram visually shows how the transaction flowed on the network from the origin (the sender) to the receivers on the right-hand side.

A summary view of various attributes of the transaction is shown in the following screenshot:

1PL6gsm49xCFMvrXqgGcee5cdrG119GoWN (0.00137322 BTC - Output)		1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3 - (Unspent) 1ET3oBGf8JpunjyE7owyVtmBjmvcDycQe - (Unspent)	0.00033324 BTC 0.00093376 BTC
			0.001267 BTC
Summary		Inputs and Outputs	
Size	226 (bytes)	Total Input	0.00137322 BTC
Weight	904	Total Output	0.001267 BTC
Received Time	2017-10-29 16:47:58	Fees	0.00010622 BTC
Included In Blocks	492229 (2017-10-29 16:51:42 + 4 minutes)	Fee per byte	47 sat/B
Confirmations	731 Confirmations	Fee per weight unit	11.75 sat/WU
Visualize	View Tree Chart	Estimated BTC Transacted	0.00033324 BTC
		Scripts	Hide scripts & coinbase

Figure 6.7: Snapshot of the transaction taken from blockchain.info

Looking at the preceding screenshot, there are a number of fields that contain various values. Important fields are listed as follows with their purpose and explanation:

- **Size:** This is the size of the transaction in bytes.

- **Weight**: This is the new metric given for the block and transaction sizes since the introduction of the **segregated witness (SegWit)** soft fork version of Bitcoin. Please see *Chapter 7, Bitcoin Network and Payments*, for more information on SegWit.
- **Received time**: This is the time when the transaction was received.
- **Included in blocks**: This shows the block number on the blockchain in which the transaction is included.
- **Confirmations**: This is the number of confirmations completed by miners for this transaction.
- **Total input**: This is the number of total inputs in the transaction.
- **Total output**: This is the number of total outputs from the transaction.
- **Fees**: This is the total fee charged.
- **Fee per byte**: This field represents the total fee divided by the number of bytes in the transaction; for example, 10 Satoshis per byte.
- **Fee per weight unit**: For legacy transactions, this is calculated using the total number of bytes * 4. For SegWit transactions, it is calculated by combining a SegWit marker, flag, and witness field as one weight unit, and each byte of the other fields as four weight units.

The transaction ID of this transaction on the Bitcoin network is

`d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d`.

This can be further explored using the following link via the services provided by blockchain.info. This transaction ID is available in the wallet software after the transaction has been sent to the network. From there, it can be further explored using one of many Bitcoin blockchain explorers available online. We are using blockchain.info as an example, which can be found here:

<https://blockchain.info/tx/d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d>

Bitcoin transactions are serialized for transmission over the network and encoded in hex format. As an example, the transaction shown at the preceding URL is also shown in hex format as follows. We will see later, in

the *Transactions* section, how this hex-encoded transaction can be decoded and which fields make up a transaction:

```
01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef90f
```

In summary, a payment transaction in the Bitcoin network can be divided into the following steps:

1. The transaction starts with a sender signing the transaction with their private key.
2. The transaction is serialized so that it can be transmitted over the network.
3. The transaction is broadcast to the network.
4. Miners listening for transactions pick up the transaction.
5. The transaction is verified for its legitimacy by the miners.
6. The transaction is added to the candidate/proposed block for mining.
7. Once mined, the result is broadcast to all nodes on the Bitcoin network.
8. Usually, at this point, users wait for up to six confirmations to be received before a transaction is considered final; however, a transaction can be considered final at the previous step. Confirmations serve as an additional mechanism to ensure that there is probabilistically a very low chance for a transaction to be reverted, but otherwise, once a mined block is finalized and announced, the transactions within that block are final at that point.

Mining, transactions, and other relevant concepts will become clearer in the following sections in the chapter.

Bitcoin, being a digital currency, has various denominations, as shown in the following table. A sender or receiver can request any amount. The smallest Bitcoin denomination is the Satoshi. Bitcoin currency units are described as follows:

DENOMINATION	ABBREVIATION	FAMILIAR NAME	VALUE IN BTC
Satoshi	SAT	Satoshi	0.0000001 BTC
Microbit	µBTC (uBTC)	Microbitcoin or Bit	0.000001 BTC
Millibit	mBTC	Millibitcoin	0.001 BTC
Centibit	cBTC	Centibitcoin	0.01 BTC
Decibit	dBTC	Decibitcoin	0.1 BTC
Bitcoin	BTC	Bitcoin	1 BTC
DecaBit	daBTC	Decabitcoin	10 BTC
Hectobit	hBTC	Hectobitcoin	100 BTC
Kilobit	kBTC	Kilobitcoin	1000 BTC
Megabit	MBTC	Megabitcoin	1000000 BTC

Figure 6.8: Bitcoin denominations

Now, we will introduce the building blocks of Bitcoin. First, we will look at the keys and addresses that are used to represent the ownership and transfer of value on the Bitcoin network.

Cryptographic keys

On the Bitcoin network, possession of Bitcoins and the transfer of value via transactions are reliant upon private keys, public keys, and addresses.

Elliptic Curve Cryptography (ECC) is used to generate public and private key pairs in the Bitcoin network. We have already covered these concepts in *Chapter 4, Public Key Cryptography*, and here we will see how private and public keys are used in the Bitcoin network.

Private keys in Bitcoin

Private keys are required to be kept safe and normally reside only on the owner's side. Private keys are used to digitally sign the transactions, proving ownership of the bitcoins.

Private keys are fundamentally 256-bit numbers randomly chosen in the range specified by the *SECP256K1 ECDSA* curve recommendation. Any randomly chosen 256-bit number from `0x1` to `0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140` is a valid private key.

Private keys are usually encoded using **Wallet Import Format (WIF)** in order to make them easier to copy and use. It is a way to represent the full-size private key in a different format. WIF can be converted into a private key and vice versa. The steps are described here.

For example, consider the following private key:

```
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA57141952
```

When converted into WIF format, it looks as shown here:

```
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP
```

Interested readers can do some experimentation using the tool available at <http://gobittest.appspot.com/PrivateKey>.

Also, **mini private key format** is sometimes used to create the private key with a maximum of 30 characters in order to allow storage where physical space is limited; for example, etching on physical coins or encoding in damage-resistant QR codes. The QR code is more damage resistant because more dots can be used for error correction and less for encoding the private key.



QR codes use Reed-Solomon error correction. Discussion on the error correction mechanism and its underlying details is out of the scope of this book, but interested readers can refer to

https://en.wikipedia.org/wiki/QR_code#Error_correction for further information.

A private key encoded using mini private key format is also sometimes called a **minikey**. The first character of the mini private key is always the uppercase letter `s`. A mini private key can be converted into a normal-sized private key, but an existing normal-sized private key cannot be converted into a mini private key. This format was used in **Casascius** physical bitcoins.



Interested readers can find more information at
https://en.Bitcoin.t/wiki/Casascius_physical_bitcoins.



Figure 6.9: A Casascius physical Bitcoin's security hologram paper with mini key and QR code

The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

As we learned in *Chapter 4, Public Key Cryptography*, private keys have their own corresponding public keys. Public keys on their own are useless and private keys on their own are equally of no use. Pairs of public and private keys are required for the normal functioning of any public key

cryptography-based systems such as the Bitcoin blockchain. In the next section, we introduce the usage of public keys in Bitcoin.

Public keys in Bitcoin

Public keys exist on the blockchain and all network participants can see them. Public keys are derived from private keys due to their special mathematical relationship with those private keys. Once a transaction signed with the private key is broadcast on the Bitcoin network, public keys are used by the nodes to verify that the transaction has indeed been signed with the corresponding private key. This process of verification proves the ownership of the bitcoin.

Bitcoin uses ECC based on the SECP256K1 standard. More specifically, it makes use of an **Elliptic Curve Digital Signature Algorithm (ECDSA)** to ensure that funds remain secure and can only be spent by the legitimate owner. If you need to refresh the relevant cryptography concepts, you can refer to *Chapter 4, Public Key Cryptography*, where ECC was explained. Public keys can be represented in uncompressed or compressed format, and are fundamentally x and y coordinates on an elliptic curve. In uncompressed format, public keys are presented with a prefix of `0x4` in hexadecimal format. x and y coordinates are both 32 bytes in length. In total, a compressed public key is 33 bytes long, compared to the 65-byte uncompressed format. The compressed version of public keys include only the x part, since the y part can be derived from it.

The reason why the compressed version of public keys works is that if the ECC graph is visualized, it reveals that the y coordinate can be either below the x axis or above the x axis, and as the curve is symmetric, only the location in the prime field is required to be stored. If y is even then its value lies above the x axis, and if it is odd then it is below the x axis. This means that instead of storing both x and y as the public key, only x needs to be stored with the information about whether y is even or odd.

Initially, the Bitcoin client used uncompressed keys, but starting from Bitcoin Core client 0.6, compressed keys are used as standard. This resulted

in an almost 50% reduction of space used to store public keys in the blockchain.

Keys are identified by various prefixes, described as follows:

- Uncompressed public keys use `0x04` as the prefix. Uncompressed public keys are 65 bytes long. They are encoded as 256-bit unsigned big-endian integers (32 bytes), which are concatenated together and finally prefixed with a byte `0x04`. This means 1 byte for the `0x04` prefix, 32 bytes for the x integer, and 32 bytes for y integer, which makes it 65 bytes in total.
- Compressed public keys start with `0x03` if the y 32-byte (256-bit) part of the public key is odd. It is 33 bytes in length as 1 byte is used by the `0x03` prefix (depicting an odd y) and 32 bytes for storing the x coordinate.
- Compressed public keys start with `0x02` if the y 32-byte (256-bit) part of the public key is even. It is 33 bytes in length as 1 byte is used by the `0x02` prefix (depicting an even y) and 32 bytes for storing the x coordinate.

Having talked about private and public keys, let's now move on to another important aspect of Bitcoin: addresses derived from public keys.

Addresses in Bitcoin

A Bitcoin address is created by taking the corresponding public key of a private key and hashing it twice, first with the SHA256 algorithm and then with RIPEMD160. The resultant 160-bit hash is then prefixed with a version number and finally encoded with a Base58Check encoding scheme. The Bitcoin addresses are 26-35 characters long and begin with digits 1 or 3.

A typical Bitcoin address looks like the string shown here:

```
1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt
```

Addresses are also commonly encoded in a QR code for easy distribution. The QR code of the preceding Bitcoin address is shown in the following image:



Figure 6.10: QR code of the Bitcoin address 1ANAguGG8bikEv2fYsTBnRUmx7QUcK58wt

Currently, there are two types of addresses, the commonly used P2PKH and another P2SH type, starting with numbers 1 and 3, respectively. In the early days, Bitcoin used direct Pay-to-Pubkey, which is now superseded by P2PKH. These types will be explained later in the chapter. However, direct Pay-to-Pubkey is still used in Bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise. Avoiding address reuse circumvents anonymity issues to an extent, but Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks, and selfish mining, all of which require different approaches to resolve.



Transaction malleability has been resolved with the so-called SegWit soft-fork upgrade of the Bitcoin protocol. This concept will be explained later in the chapter.



Figure 6.11: From bitaddress.org, a private key and Bitcoin address in a paper wallet

Base58Check encoding

Bitcoin addresses are encoded using the Base58Check encoding. This encoding is used to limit the confusion between various characters, such as 001l, as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utilizing a set of 58 alphanumeric symbols. More explanation and logic can be found in the `base58.h` source file in the Bitcoin source code:

```
/**  
 * Why base-58 instead of standard base-64 encoding?  
 * - Don't want 001l characters that look the same in some fonts  
 *   could be used to create visually identical looking data.  
 * - A string with non-alphanumeric characters is not as easily  
 * - E-mail usually won't line-break if there's no punctuation to  
 * - Double-clicking selects the whole string as one word if it's  
 */
```



This file is present in the Bitcoin source code and can be viewed at <https://github.com/bitcoin/bitcoin/blob/c8971547d9c9460fcbec6f54888df83f002c3dfd/src/base58.h>.

Now that we have covered private keys, public keys, Base58 encoding, and addresses, with that knowledge we can now examine how a Bitcoin address is generated by using all these elements. We briefly touched on it when we introduced addresses in Bitcoin earlier, but now we will see this in more detail with a diagram.

The following diagram shows how an address is generated, from generating the private key to the final output of the Bitcoin address:

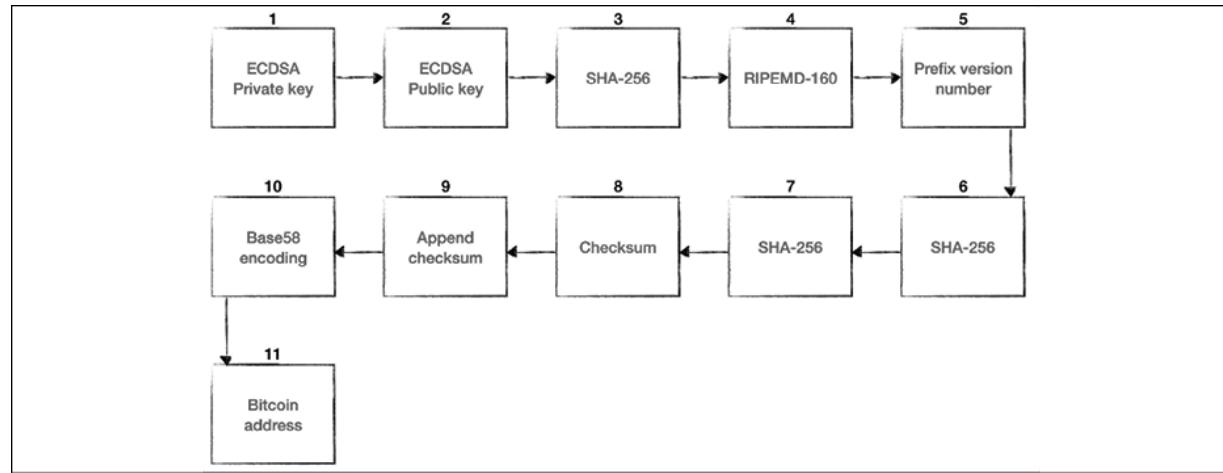


Figure 6.12: Address generation in Bitcoin

In the preceding diagram, there are several steps that we will now explain:

1. In the first step, we have a randomly generated ECDSA private key.
2. The public key is derived from the ECDSA private key.
3. The public key is hashed using the SHA-256 cryptographic hash function.
4. The hash generated in *step 3* is hashed using the RIPEMD-160 hash function.
5. The version number is prefixed to the RIPEMD-160 hash generated in *step 4*.
6. The result produced in *step 5* is hashed using the SHA-256 cryptographic hash function.
7. SHA-256 is applied again.

8. The first 4 bytes of the result produced from *step 7* is the address `checksum`.
9. This `checksum` is appended to the RIPEMD-160 hash generated in *step 4*.
10. The resultant byte string is encoded into a Base58-encoded string by applying the Base58 encoding function.
11. Finally, the result is a typical Bitcoin address.

In addition to common types of addresses in Bitcoin, there are some advanced types of addresses available in Bitcoin too. We discuss these next.

Vanity addresses

As Bitcoin addresses are based on Base58 encoding, it is possible to generate addresses that contain human-readable messages. An example is shown as follows—note that in the first line, the name `BasHir` appears:



Figure 6.13: Vanity public address encoded in QR

Vanity addresses are generated using a purely brute-force method. An example of a paper wallet with a vanity address is shown in the following screenshot:



Figure 6.14: Vanity address generated from <https://bitcoinvanitygen.com/>

In the preceding screenshot, on the right-hand bottom corner, the public vanity address is displayed with a QR code. The paper wallets can be stored physically as an alternative to the electronic storage of private keys.

Multi-signature addresses

As the name implies, these addresses require multiple private keys. In practical terms, this means that in order to release the coins, a certain set number of signatures is required. This is also known as *M of N multisig*. Here, *M* represents the threshold or minimum number of signatures required from *N* number of keys to release the Bitcoins.

Remember that we discussed this concept in *Chapter 4, Public Key Cryptography*.

With this section, we've finished our introduction to addresses in Bitcoin. In the next section, we will introduce Bitcoin transactions, which are the most fundamental and important aspect of Bitcoin.

Transactions

Transactions are at the core of the Bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a Bitcoin address, or can be quite complex, depending on the requirements. Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction, and outputs as coins being created. If a transaction is minting new coins, then there is no input, and therefore no signature is needed. If a transaction should send coins to some other user (a Bitcoin address), then it needs to be signed by the sender with their private key. In this case, a reference is also required to the previous transaction to show the origin of the coins. Coins are unspent transaction outputs represented in Satoshis.

Transactions are not encrypted and are publicly visible on the blockchain. Blocks are made up of transactions, and these can be viewed using any online blockchain explorer.

The transaction lifecycle

Now, let's look at the lifecycle of a Bitcoin **transaction**. The steps of the process are as follows:

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcasted to the Bitcoin network using a flooding algorithm.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transactions are placed in the block, they are placed in a special memory buffer called the transaction pool. The purpose of the transaction pool is explained in the next section.

5. Next, the mining starts, which is the process through which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources. Once a miner solves the PoW problem, it broadcasts the newly mined block to the network. PoW is explained in detail in the *Mining* section. The nodes verify the block and propagate the block further, and confirmations start to generate.
6. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed. However, three to six is just a recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after six confirmations.

When a transaction is created by a user and sent to the network, it ends up in a special area on each Bitcoin software client. This special area is called the transaction pool or memory pool.

Transaction pool

Also known as memory pools, these pools are basically created in local memory (computer RAM) by nodes (Bitcoin clients) in order to maintain a temporary list of transactions that have not yet been added to a block.

Miners pick up transactions from these memory pools to create candidate blocks. Miners select transactions from the pool after they pass the verification and validity checks. The selection of which transactions to choose is based on the fee and their place in the order of transactions in the pool. Miners prefer to pick up transactions with higher fees.

To send transactions on the Bitcoin network, the sender needs to pay a fee to the miners. This fee is an incentive mechanism for the miners.

Transaction fees

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs from the sum of the outputs.

A simple formula can be used:

$$fee = \text{sum(inputs)} - \text{sum(outputs)}$$

The fees are used as an incentive for miners to encourage them to include users' transactions in the block the miners are creating. All transactions end up in the memory pool, from where miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners. There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes.

Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course, but may take a very long time. This is, however, no longer practical due to the high volume of transactions and competing investors on the Bitcoin network, therefore it is advisable to always provide a fee. The time for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. Transaction time is dependent on transaction fees and network activity. If the network is very busy, then naturally, transactions will take longer to process, and if you pay a higher fee then your transaction is more likely to be picked by miners first due to the additional incentive of the higher fee.

A transaction on the Bitcoin network is represented by a data structure that consists of several fields. We will now introduce the transaction data structure.

The transaction data structure

A transaction at a high level contains metadata, inputs, and outputs. Transactions are combined to create a block's body.

The transaction data structure is shown in the following table:

Field	Size	Description
Version number	4 bytes	Specifies the rules to be used by the miners and nodes for transaction processing. There are two versions of transactions, that is, 1 and 2.
Input counter	1–9 bytes	The number (a positive integer) of inputs included in the transaction.
List of inputs	Variable	<p>Each input is composed of several fields. These include:</p> <ul style="list-style-type: none"> • The previous transaction hash • The index of the previous transaction • Transaction script length • Transaction script • Sequence number <p>The first transaction in a block is also called a coinbase transaction. It specifies one or more transaction inputs. In summary, this field describes which Bitcoins are going to be spent.</p>
Output counter	1–9 bytes	A positive integer representing the number of outputs.
List of outputs	Variable	Outputs included in the transaction. This field depicts the target recipient(s) of the Bitcoins.
Lock time	4 bytes	This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or the block height.

The following sample transaction is the decoded transaction from the first example of a payment provided at the start of this chapter:

```
{
  "lock_time":0,
  "size":226,
```

```

    "inputs": [
        {
            "prev_out": {
                "index": 139,
                "hash": "40120e43f00ff96e098a9173f14f1371655b3478bc0e"
            },
            "script": "483045022100de6fd8120d9f142a82d5da9389e271caa3a757b0",
            }
        ],
        "version": 1,
        "vin_sz": 1,
        "hash": "d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae",
        "vout_sz": 2,
        "out": [
            {
                "script_string": "OP_DUP OP_HASH160 c568ffeb46c6a9362e44",
                "address": "1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3",
                "value": 33324,
                "script": "76a914c568ffeb46c6a9362e44a5a49deaa6eab05a619"
            },
            {
                "script_string": "OP_DUP OP_HASH160 9386c8c880488e80a6ce",
                "address": "1ET3oBGf8JpunjytE7owyVtmBjmvcDycQe",
                "value": 93376,
                "script": "76a9149386c8c880488e80a6ce8f186f788f3585f74ae"
            }
        ]
    }
}

```

As shown in the preceding decoded transaction, there are a number of structures that make up the transaction. All these elements will be described now.

Metadata

This part of the transaction contains values such as the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a `lock_time` field. Every transaction has a prefix specifying the version number. These fields are shown in the preceding example as `lock_time`, `size`, and `version`.

Inputs

Generally, each input spends a previous output. Each output is considered an **Unspent Transaction Output (UTXO)** until an input consumes it. A UTXO is an unspent transaction output that can be spent as an input to a new transaction.

The transaction input data structure is explained in the following table:

Field	Size	Description
Transaction hash	32 bytes	The hash of the previous transaction with UTXO
Output index	4 bytes	This is the previous transaction's output index, such as UTXO to be spent
Script length	1-9 bytes	Size of the unlocking script
Unlocking script	Variable	Input script (<code>ScriptSig</code>), which satisfies the requirements of the locking script
Sequence number	4 bytes	Usually disabled or contains lock time — disabled is represented by <code>0xFFFFFFFF</code>

In the previous sample decoded transaction, the inputs are defined under the `"inputs" : []` section.

Outputs

Outputs have three fields, and they contain instructions for sending bitcoins. The first field contains the amount of Satoshis, whereas the second field contains the size of the locking script. Finally, the third field contains a locking script that holds the conditions that need to be met in order for the output to be spent. More information on transaction spending using locking

and unlocking scripts and producing outputs is discussed later in this section.

The transaction output data structure is explained in the following table:

Field	Size	Description
Value	8 bytes	The total number (in positive integers) of Satoshis to be transferred
Script size	1 – 9 bytes	Size of the locking script
Locking script	Variable	Output script (<code>ScriptPubKey</code>)

In the previous sample decoded transaction, two outputs are shown under the `"out": [` section.

Verification

Verification is performed using Bitcoin's scripting language, which we will now describe in the next section in detail.

The Script language

Bitcoin uses a simple stack-based language called **Script** to describe how bitcoins can be spent and transferred. It is not Turing complete and has no loops to avoid any undesirable effects of long-running/hung scripts on the Bitcoin network. This scripting language is based on a Forth programming language-like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from left to right using a **Last in, First Out (LIFO)** stack.

Scripts are composed of two components, namely elements and operations. Scripts use various operations (**opcodes**) or instructions to define their operations. Elements simply represent data such as digital signatures. Opcodes are also known as words, commands, or functions. Earlier versions

of the Bitcoin node software had a few opcodes that are no longer used due to bugs discovered in their design.

The various categories of the scripting opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography, and lock time.

A transaction script is evaluated by combining `ScriptSig` and `ScriptPubKey`. `ScriptSig` is the unlocking script, whereas `ScriptPubKey` is the locking script. We will now describe how a transaction is unlocked and spent:

- `ScriptSig` is provided by the user who wishes to unlock the transaction
- `ScriptPubKey` is part of the transaction output and specifies the conditions that need to be fulfilled in order to spend the output

In other words, outputs are locked by the `ScriptPubKey` (the locking script), which contains the conditions that when met, will unlock the output, and the coins can then be redeemed. We will see the script execution in detail shortly.

Commonly used opcodes

In a computer, an opcode is an instruction to perform some operation. For example, `ADD` is an opcode, which is used for integer addition in Intel CPUs and various other architectures. Similarly, in Bitcoin design, opcodes are introduced that perform several operations related to Bitcoin transaction verification.



All opcodes are declared in the `script.h` file in the Bitcoin reference client source code, available at
<https://github.com/Bitcoin/Bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/script/script.h#L53>.

A description of some of the most commonly used opcodes is listed in the following table, extracted from the Bitcoin Developer's Guide:

Opcode	Description
<code>OP_CHECKSIG</code>	This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then <code>TRUE</code> is pushed onto the stack; otherwise, <code>FALSE</code> is pushed.
<code>OP_EQUAL</code>	This returns <code>1</code> if the inputs are exactly equal; otherwise, <code>0</code> is returned.
<code>OP_DUP</code>	This duplicates the top item in the stack.
<code>OP_HASH160</code>	The input is hashed twice, first with SHA-256 and then with RIPEMD-160.
<code>OP_VERIFY</code>	This marks the transaction as invalid if the top stack value is not true.
<code>OP_EQUALVERIFY</code>	This is the same as <code>OP_EQUAL</code> , but it runs <code>OP_VERIFY</code> afterward.
<code>OP_CHECKMULTISIG</code>	This instruction takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of <code>1</code> is returned as a result; otherwise, <code>0</code> is returned.
<code>OP_HASH256</code>	The input is hashed twice with SHA-256.
<code>OP_MAX</code>	This returns the larger value of two inputs.

There are many opcodes in the Bitcoin scripting language, and covering all of them is out of scope of this book. Interested readers can refer to the complete list and relevant details at

<https://en.bitcoin.it/wiki/Script#Opcodes>.

Now that we've covered the transaction life cycle and data structure, let's move on to talk about the types of scripts used to undertake these transactions.

Types of scripts

There are various standard scripts available in Bitcoin to handle the verification and value transfer from the source to the destination. These scripts range from very simple to quite complex depending upon the requirements of the transaction. Standard transaction types are discussed here. Standard transactions are evaluated using the `IsStandard()` and `IsStandardTx()` tests and only those transactions that pass the test are allowed to be broadcasted or mined on the Bitcoin network. However, nonstandard transactions are also allowed on the network, as long as they pass the validity checks:

- **Pay-to-Public-Key Hash (P2PKH):** P2PKH is the most commonly used transaction type and is used to send transactions to Bitcoin addresses. The format of this type of transaction is shown as follows:

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY  
ScriptSig: <sig> <pubKey>
```

The `ScriptPubKey` and `ScriptSig` parameters are concatenated together and executed. An example will follow shortly in this section, where this is explained in more detail.

- **Pay-to-Script Hash (P2SH):** P2SH is used in order to send transactions to a script hash (that is, addresses starting with 3) and was standardized in BIP16. In addition to passing the script, the redeem script is also evaluated and must be valid. The template is shown as follows:

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL  
ScriptSig: [<sig>...<sign>] <redeemScript>
```

- **MultiSig (Pay to MultiSig):** The M of N multisignature transaction script is a complex type of script where it is possible to construct a

script that requires multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig> . . . <sign>]
```

Raw `multisig` is obsolete, and `multisig` is now usually part of the P2SH redeem script, mentioned in the previous bullet point.

- **Pay to Pubkey:** This script is a very simple script that is commonly used in coinbase transactions. It is now obsolete and was used in an old version of Bitcoin. The public key is stored within the script in this case, and the unlocking script is required to sign the transaction with the private key. The template is shown as follows:

```
<PubKey> OP_CHECKSIG
```

- **Null data/OP_RETURN:** This script is used to store arbitrary data on the blockchain for a fee. The limit of the message is 40 bytes. The output of this script is unredeemable because `OP_RETURN` will fail the validation in any case. `ScriptSig` is not required in this case. The template is very simple and is shown as follows:

```
OP_RETURN <data>
```

The P2PKH script execution is shown in the following diagram:

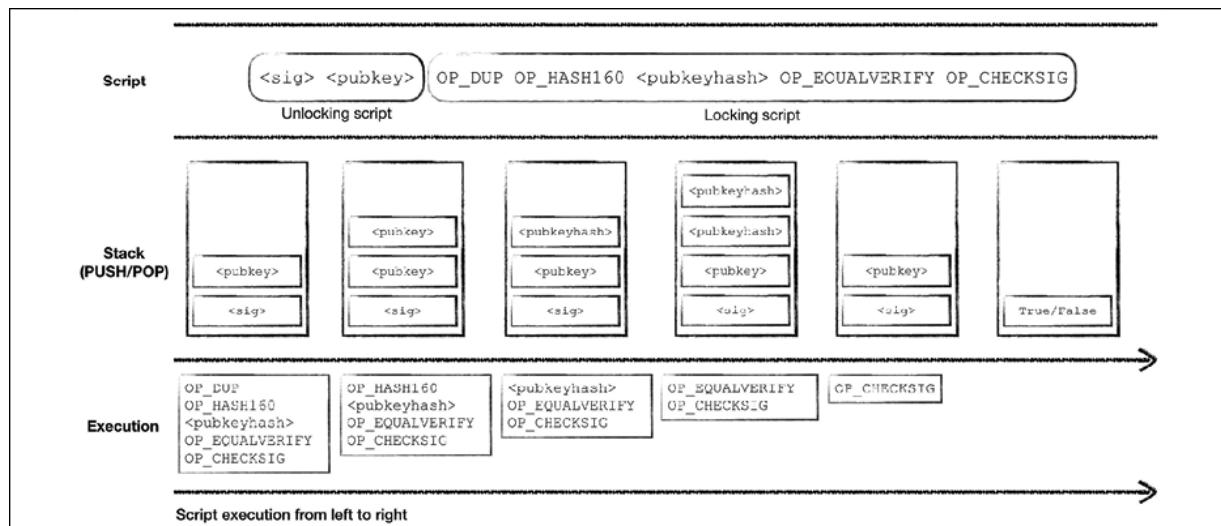


Figure 6.15: P2PKH script execution

In the preceding diagram, we have a standard P2PKH script presented on top, which shows both the unlocking (`ScriptSig`) and locking (`ScriptPubKey`) parts of the script.

Recall that in the introduction of the Script language; we mentioned that a Bitcoin script is composed of two parts, an unlocking script and a locking script. The unlocking script is composed of `<sig>` and `<pubkey>` elements, which are part of all transaction inputs. The unlocking script satisfies the conditions that are required to consume the output. The locking script defines the conditions that are required to be met to spend the bitcoins. Transactions are authorized by executing both of these parts collectively.



In simple words, locking means providing Bitcoins to somebody, whereas unlocking means consuming the acquired Bitcoins.

Now focusing again on the previous diagram, we see that in the middle we have a visualization of the stack where the data elements are pushed and popped from. In the bottom section, we show the execution of the script. This diagram shows the step-by-step execution of the script with its outcome on the stack.

Now let's examine how this script is executed:

1. In the first step of the data elements, `<sig>` and `<pubkey>` are placed on the stack.
2. The stack item at the top `<pubkey>` is duplicated due to the `OP_DUP` instruction, which duplicates the top stack item.
3. After this, the instruction `OP_HASH160` executes, which produces the hash of `<pubkey>`, which is the top element in the stack.
4. `<pubkeyhash>` is then pushed on to the stack. At this stage, we have two hashes on the stack: the one that is produced as a result of executing `OP_HASH160` on `<pubkey>` from the unlocking script, and the other one provided by the locking script.

5. Now the `OP_EQUALVERIFY` opcode instruction executes and checks that whether the top two elements (that is, the hashes) are equal or not. If they are equal, the script continues, otherwise it fails.
6. Finally, `OP_CHECKSIG` executes to check the validity of the signatures of the top two elements of the stack. If the signature is valid then the stack will contain the value `True` that is, 1, otherwise `False`, that is, 0.

All transactions are encoded into hex format before being transmitted over the Bitcoin network. The following sample transaction is shown in hex format, and is retrieved using `bitcoin-cli` on the Bitcoin node running on mainnet by using the following command:

```
$ bitcoin-cli getrawtransaction "d28ca5a59b2239864eac1c96d3fd1c2
{
    "result": "01000000017d3876b14a7ac16d8d550abc78345b6571134ff173
```



Note that this is the same transaction that was presented as an example at the start of this chapter.

Scripting is quite limited and can only be used to program one thing—the transfer of bitcoins from one address to other addresses. However, there is some flexibility possible when creating these scripts, which allows for certain conditions to be put on the spending of the bitcoins. This set of conditions can be considered a basic form of a financial contract. In other words, we can say that there is a basic support for building contracts in Bitcoin. However, it is not same as **smart contracts**, which allow the writing of arbitrary programs on the blockchain.

We will discuss smart contracts in detail later in *Chapter 10, Smart Contracts*, but for now, we will introduce the concept of contracts in Bitcoin.

Contracts

Contracts are Bitcoin scripts that use the Bitcoin blockchain to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to programmatically create complex contracts that can be used in many real-world scenarios. Contracts allow the development of completely decentralized, independent, and reduced-risk platforms by programmatically enforcing different conditions for unlocking the money (bitcoins). With the security guarantees provided by the Bitcoin blockchain, it is almost impossible to circumvent these conditions.

Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language. The current implementation of the script language is minimal, but it is still possible to develop various types of complex contracts. For example, we could set the release of funds to be allowed only when multiple parties sign the transaction, or release the funds only after a specific amount of time has elapsed. Both of these scenarios can be realized using the `multisig` and transaction lock time options.

Even though the scripting language in Bitcoin can be used to create complex contracts, it is quite limited and is not at all on par with the smart contracts, which are Turing complete constructs and allow arbitrary program development.

Recently, however, there has been some more advancement in this area. A new smart contract language for Bitcoin has been announced, called *Miniscript*, which will be discussed next.

Miniscript

This language allows for a more structured approach to writing Bitcoin scripts. Even though Bitcoin Script supports combinations of different spending conditions, such as time and hash locks, it is not easy to analyze existing scripts or build new complex scripts. Miniscript makes it easier to write complex spending rules. It also makes it easier to ascertain the correctness of the script.

Currently, Miniscript supports P2WSH and P2SH-P2WSH, and offers limited support for P2SH.



More information on Miniscript is available at
<http://bitcoin.sipa.be/miniscript/>.

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called the coinbase, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data storage.

A coinbase transaction input has the same number of fields as a usual transaction input, but the structure contains the coinbase data size and fields instead of the unlocking script size and fields. Also, it does not have a reference pointer to the previous transaction. This structure is shown in the following table:

Field	Size	Description
Transaction hash	32 bytes	Set to all zeroes as no hash reference is used
Output index	4 bytes	Set to <code>0xFFFFFFFF</code>
Coinbase data length	1-9 bytes	2-100 bytes
Data	Variable	Any data
Sequence number	4 bytes	Set to <code>0xFFFFFFFF</code>

All transactions in the Bitcoin system go under a validation mechanism. We introduce how Bitcoin transactions are validated in the next section.

Transaction validation

This verification process is performed by Bitcoin nodes. There are three main things that nodes check when verifying a transaction:

1. That transaction inputs are previously unspent. This validation step prevents double spending by verifying that the transaction inputs have not already been spent by someone else.
2. That the sum of the transaction outputs is not more than the total sum of the transaction inputs. However, both input and output sums can be the same, or the sum of the input (total value) could be more than the total value of the outputs. This check ensures that no new bitcoins are created out of thin air.
3. That the digital signatures are valid, which ensures that the script is valid.

Even though transaction construction and validation are generally a secure and sound process, some vulnerabilities exist in Bitcoin. We will now introduce some Bitcoin's infamous shortcomings.

Transaction bugs

The following are two major Bitcoin vulnerabilities that have been infamously exploited.

Transaction malleability

Transaction malleability is a Bitcoin attack that was introduced due to a bug in the Bitcoin implementation. Due to this bug, it became possible for an adversary to change the transaction ID of a transaction, thus resulting in a scenario where it would appear that a certain transaction has not been executed.

This can allow scenarios where double deposits or withdrawals can occur. In other words, this bug allows the changing of the unique ID of a Bitcoin

transaction before it is confirmed. If the ID is changed before confirmation without making the transaction invalid, it would seem that the transaction did not occur at all, which can then give the false impression that the transaction has not been executed, thus allowing double-deposit or withdrawal attacks.

Value overflow

This incident is one of the most well-known events in Bitcoin history. On 15 August 2010, a transaction was discovered that created roughly 184 billion bitcoins. This problem occurred due to the integer overflow bug where the amount field in the Bitcoin code was defined as a signed integer instead of an unsigned integer. This bug means that the amount can also be negative, and resulted in a situation where the outputs were so large that the total value resulted in an overflow. To the validation logic in Bitcoin code, all appeared to be correct, and it looked like the fee was also positive (after the overflow). This bug was fixed via a soft fork (more on this in the *Blockchain* section) quickly after its discovery.



More information on this vulnerability is available at
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-5139>.



Security research in general, and specifically in the world of Bitcoin (cryptocurrency/blockchain), is a fascinating subject; perhaps some readers will look at the vulnerability examples in the following links for inspiration and embark on a journey to discover some more vulnerabilities.

An example of a recently resolved critical problem in Bitcoin, which remained undiscovered for quite some time, can be found at
<https://Bitcoincore.org/en/2019/11/08/CVE-2017-18350/>.

Another example of a critical inflation and denial-of-service bug, which was discovered on September 17, 2018 and fixed rapidly, is detailed at
<https://Bitcoincore.org/en/2018/09/20/notice/>.

Perhaps there are still some bugs that are yet to be discovered!

Now that we understand the fundamentals of how Bitcoin transactions work, and we appreciate some of the security-related aspects of Bitcoin, let's now look at the Bitcoin blockchain, which is the foundation of the Bitcoin cryptocurrency.

Blockchain

As we discussed in *Chapter 1, Blockchain 101*, a blockchain is a distributed ledger of transactions. Specifically, from Bitcoin's perspective, the blockchain can be defined as a public, distributed ledger holding a timestamped, ordered, and immutable record of all transactions on the Bitcoin network. Transactions are picked up by miners and bundled into blocks for mining. Each block is identified by a hash and is linked to its previous block by referencing the previous block's hash in its header.

The data structure of a Bitcoin block is shown in the following table:

Field	Size	Description
Block size	4 bytes	The size of the block.
Block header	80 bytes	This includes fields from the block header described in the next section.
Transaction counter	Variable	The field contains the total number of transactions in the block, including the coinbase transaction. Size ranges from 1-9 bytes.
Transactions	Variable	All transactions in the block.

The block header mentioned in the previous table is a data structure that contains several fields. This is shown in the following table:

--	--	--

Field	Size	Description
Version	4 bytes	The block version number that dictates the block validation rules to follow.
Previous block's header hash	32 bytes	This is a double SHA256 hash of the previous block's header.
Merkle root hash	32 bytes	This is a double SHA256 hash of the Merkle tree of all transactions included in the block.
Timestamp	4 bytes	This field contains the approximate creation time of the block in Unix-epoch time format. More precisely, this is the time when the miner started hashing the header (the time from the miner's location).
Difficulty target	4 bytes	This is the current difficulty target of the network/block.
Nonce	4 bytes	This is an arbitrary number that miners change repeatedly to produce a hash that is lower than the difficulty target.

The following diagram provides a detailed view of the blockchain structure. As shown in the following diagram, blockchain is a chain of blocks where each block is linked to its previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block:

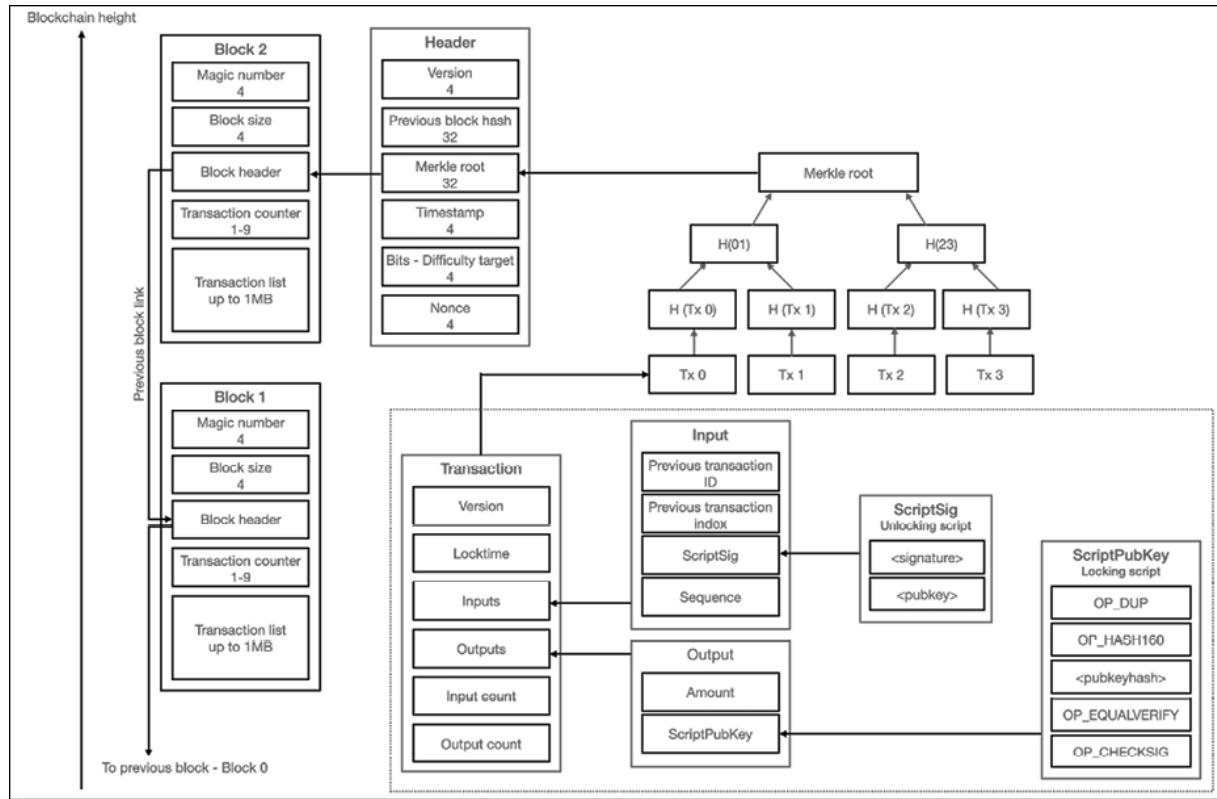


Figure 6.16: A visualization of blockchain, block, block header, transactions, and scripts

The preceding diagram shows a high-level overview of the Bitcoin blockchain. On the left-hand side, blocks are shown starting from bottom to top. Each block contains transactions and block headers, which are further magnified on the right-hand side. At the top, first, the block header is enlarged to show various elements within the block header. Then on the right-hand side, the Merkle root element of the block header is shown in magnified view, which shows how Merkle root is constructed.

We have discussed Merkle trees in detail previously. You can refer to *Chapter 4, Public Key Cryptography*, if you need to revise the concept.

Further down the diagram, transactions are also magnified to show the structure of a transaction and the elements that it contains. Also, note that transactions are then further elaborated to show what locking and unlocking scripts look like. The size (in bytes) of each field of block, header and transaction is also shown as a number under the name of the field.

Let's move on to discuss the first block in the Bitcoin blockchain: the genesis block.

The genesis block

This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software. In the genesis block, the coinbase transaction included a comment taken from The Times newspaper:



"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"

This message is a proof that the first Bitcoin block (genesis block) was not mined earlier than January 3rd, 2009. This is because the genesis block was created on January 3rd, 2009 and this news excerpt was taken from that day's newspaper.

The following representation of the genesis block code can be found in the `chainparams.cpp` file available at

<https://github.com/Bitcoin/Bitcoin/blob/master/src/chainparams.cpp>:

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce
{
    const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor
    const CScript genesisOutputScript = CScript() << ParseHex("C
    return CreateGenesisBlock(pszTimestamp, genesisOutputScript,
})
```

Bitcoin provides protection against double-spending by enforcing strict rules on transaction verification and via mining. Transactions and blocks are added to the blockchain only after the strict rule-checking explained in the *Transaction validation* section on successful PoW solutions.

Block height is the number of blocks before a particular block in the blockchain. PoW is used to secure the blockchain. Each block contains one or more transactions, out of which the first transaction is the coinbase transaction. There is a special condition for coinbase transactions that prevent them from being spent until at least 100 blocks have passed in order to avoid a situation where the block may be declared stale later on.

Stale and orphan blocks

Stale blocks are old blocks that have already been mined. Miners who keep working on these blocks due to a fork, where the longest chain (main chain) has already progressed beyond those blocks, are said to be working on a stale block. In other words, these blocks exist on a shorter chain, and will not provide any reward to their miners.

Orphan blocks are a slightly different concept. Their parent blocks are unknown. As their parents are unknown, they cannot be validated. This problem occurs when two or more miners discover a block at almost the same time. These are valid blocks and were correctly discovered at some point in the past but now they are no longer part of the main chain. The reason why this occurs is that if there are two blocks discovered at almost the same time, the one with a larger amount of PoW will be accepted and the one with the lower amount of work will be rejected. Similar to stale blocks, they do not provide any reward to their miners.

We can see this concept visually in the following diagram:

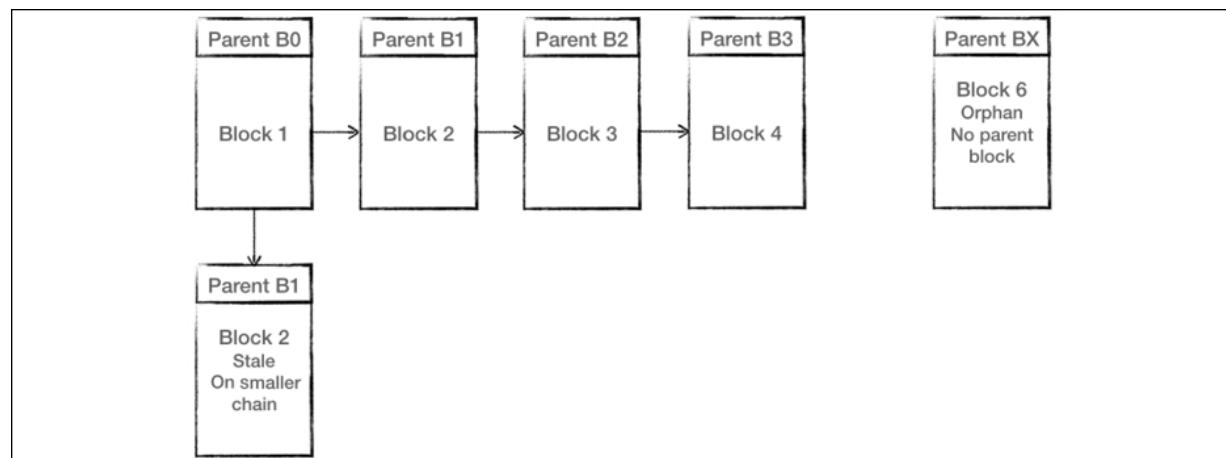


Figure 6.17: Orphan and stale blocks

In the preceding introduction to stale blocks, we introduced a new term, a **fork**. A fork is a condition that occurs when two different versions of the blockchain exist. It is acceptable in some conditions, and detrimental in a few others. There are different types of forks that can occur in a blockchain:

- Temporary forks
- Soft forks
- Hard forks

Because of the distributed nature of Bitcoin, network forks can occur inherently. In cases where two nodes simultaneously announce a valid block, it can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the Bitcoin network only by accepting the longest chain. In this case, the smaller chain will be considered orphaned. If an adversary manages to gain control of 51% of the network hash rate (computational power), then they can impose their own version of the transaction history.

Forks in a blockchain can also occur with the introduction of changes to the Bitcoin protocol. In the case of a **soft fork**, a client that chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, new and previous blocks are both acceptable, thus making a soft fork backward compatible. Miners are only required to upgrade to the new soft fork client software in order to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated already.

A **hard fork**, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure changes or major protocol changes result in a hard fork.

As Bitcoin evolves and new upgrades and innovations are introduced in it, the version associated with blocks also changes. These versions introduce various security parameters and new features. The latest Bitcoin block version is 4, which was proposed with BIP65 and has been used since

Bitcoin Core client 0.11.2. Since the implementation of BIP9, bits in the **nVersion** field are used to indicate soft-fork changes.



Details of BIP0065 are available at
<https://github.com/Bitcoin/bips/blob/master/bip-0065.mediawiki>.

Size of the blockchain

Bitcoin is an ever-growing chain of blocks and is increasing in size. The current size of the Bitcoin blockchain stands at approximately 269 GB. The following figure shows the increase in size of the blockchain over time:

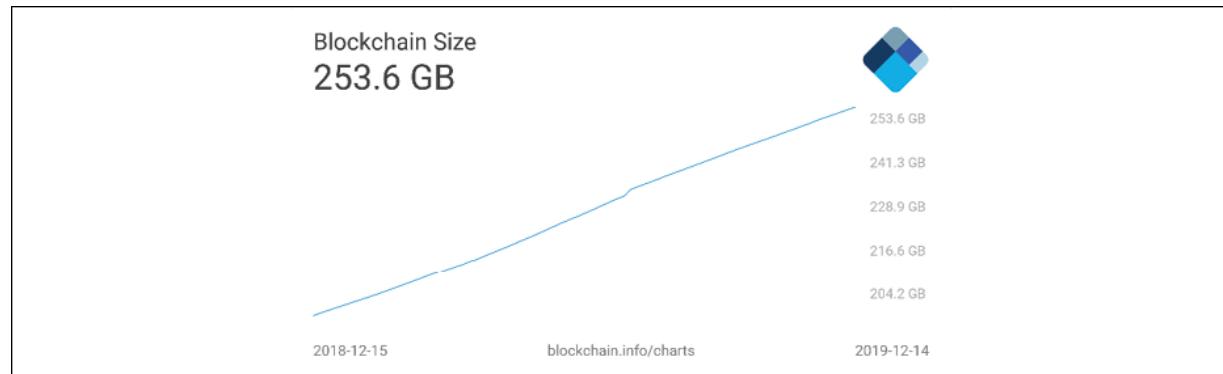


Figure 6.18: Size of Bitcoin blockchain over time

As the chain grows and more miners are added to the network, the network difficulty also increases.

Network difficulty

Network difficulty refers to a measure of how difficult it is to find a new block, or in other words, how difficult it is to find a hash below the given target.

New blocks are added to the blockchain approximately every 10 minutes, and the network difficulty is adjusted dynamically every 2,016 blocks in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following equation:

$$\text{Target} = \text{Previous target} * \text{Time}/2016 * 10 \text{ minutes}$$

Difficulty and target are interchangeable and represent the same thing. The *Previous target* represents the old target value, and *Time* is the time spent to generate the previous 2,016 blocks. Network difficulty essentially means how hard it is for miners to find a new block; that is, how difficult the hashing puzzle is now.

In the next section, mining is discussed, which will explain how the hashing puzzle is solved.

Mining

Mining is a process by which new blocks are added to the blockchain. Blocks contain transactions that are validated via the mining process by mining nodes on the Bitcoin network. Blocks, once mined and verified, are added to the blockchain, which keeps the blockchain growing. This process is resource-intensive due to the requirements of PoW, where miners compete to find a number less than the difficulty target of the network. This difficulty in finding the correct value (also called sometimes the **mathematical puzzle**) is there to ensure that miners have spent the required resources before a new proposed block can be accepted. The miners mint new coins by solving the PoW problem, also known as the partial hash inversion problem. This process consumes a high amount of resources, including computing power and electricity. This process also secures the system against fraud and double-spending attacks while adding more virtual currency to the Bitcoin ecosystem.

Roughly one new block is created (mined) every 10 minutes to control the frequency of generation of bitcoins. This frequency needs to be maintained by the Bitcoin network. It is encoded in the Bitcoin Core client to control the "money supply."

Approximately 144 blocks, that is, 1,728 bitcoins, are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at an average of 144 per day. Bitcoin supply is also limited. In 2140, all 21 million bitcoins will be finally created, and no new bitcoins can be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

Tasks of the miners

Once a node connects to the Bitcoin network, there are several tasks that a Bitcoin miner performs:

1. **Synching up with the network:** Once a new node joins the Bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the Bitcoin miner; however, this not necessarily a task that only concerns miners.
2. **Transaction validation:** Transactions broadcast on the network are validated by full nodes by verifying and validating signatures and outputs.
3. **Block validation:** Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with verification of the nonce value.
4. **Create a new block:** Miners propose a new block by combining transactions broadcast on the network after validating them.
5. **Perform PoW:** This task is the core of the mining process and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.
6. **Fetch reward:** Once a node solves the hash puzzle (PoW), it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners on the network due to a clash with another block found at roughly the same time, but once accepted,

the miner is rewarded with 12.5 bitcoins and any associated transaction fees.

We've discussed the tasks—let's now consider the rewards for performing them.

Mining rewards

Miners are rewarded with new coins if and when they discover new blocks by solving the PoW. Miners are paid transaction fees in return, for the transactions in their proposed blocks. New blocks are created at an approximate fixed rate of every 10 minutes. The rate of creation of new bitcoins decreases by 50% every 210,000 blocks, which is roughly every 4 years. When Bitcoin started in 2009, the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012 it halved down to 25 bitcoins. Currently, since May 2020, it is 6.25 bitcoins per block. This mechanism is hardcoded in Bitcoin to regulate and control inflation and limit the supply of bitcoins.

In order for miners to earn the reward, they have to show that they have solved the computational puzzle. This is called the PoW.

Proof of Work

This is a proof that enough computational resources have been spent in order to build a valid block. PoW is based on the idea that a random node is selected every time to create a new block. In this model, nodes compete with each other in proportion to their computing capacity, in order to be selected. The following equation sums up the PoW requirement in Bitcoin:

$$H(N \parallel P_hash \parallel Tx \parallel Tx \parallel \dots \parallel Tx) < Target$$

Here, N is a nonce, P_hash is a hash of the previous block, Tx represents transactions in the block, and $Target$ is the target network difficulty value.

This means that the hash of the previously mentioned concatenated fields should be less than the target hash value.

The only way to find this nonce is the brute force method. Once a certain pattern of a certain number of zeroes is met by a miner, the block is immediately broadcast and accepted by other miners.

The mining algorithm

The mining algorithm consists of the following steps:

1. The previous block's header is retrieved from the Bitcoin network.
2. Assemble a set of transactions broadcast on the network into a block to be proposed.
3. Compute the double hash of the previous block's header, combined with a nonce and the newly proposed block, using the SHA256 algorithm.
4. Check if the resulting hash is lower than the current difficulty level (the target), then PoW is solved. As a result of successful PoW, the discovered block is broadcasted to the network and miners fetch the reward.
5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce.

As the hash rate of the Bitcoin network increased, the total amount of the 32-bit nonce was exhausted too quickly. In order to address this issue, the extra nonce solution was implemented, whereby the coinbase transaction is used to provide a larger range of nonces to be searched by the miners.

This process is visualized in the following flowchart:

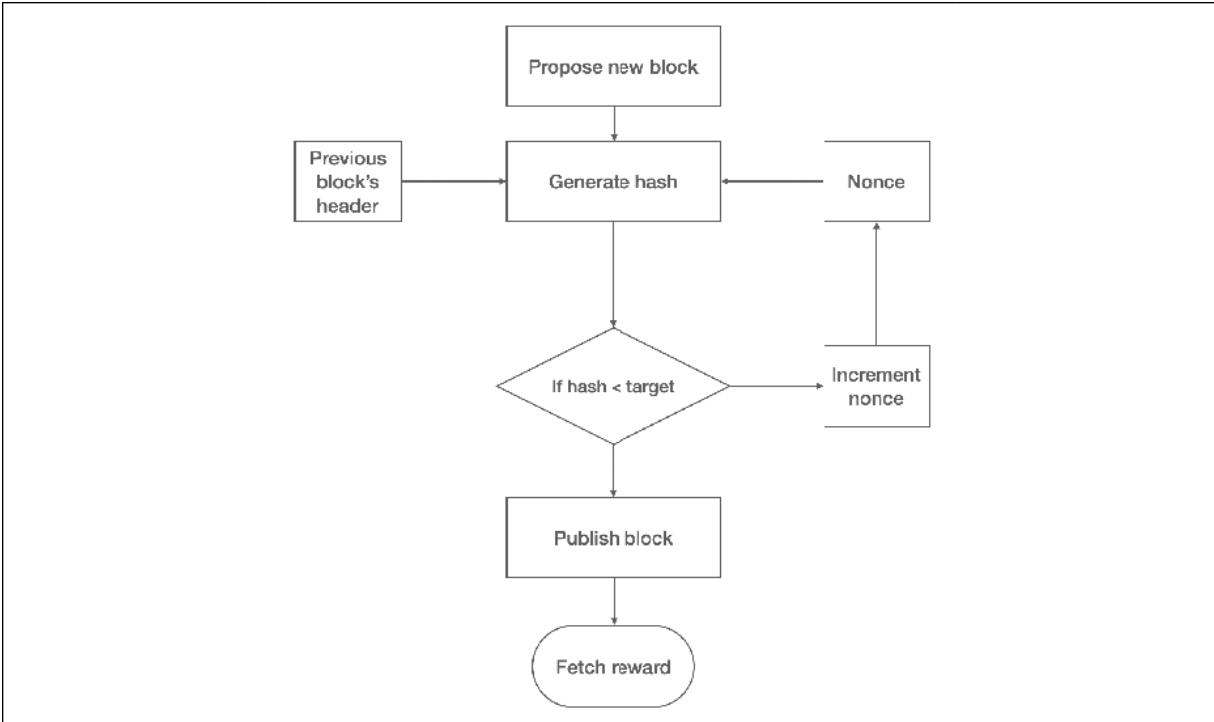


Figure 6.19: Mining process

Mining difficulty increases over time and bitcoins that could once be mined by a single-CPU laptop computer now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried through the Bitcoin command-line interface using the following command:

```
$ bitcoin-cli getdifficulty
```

This generates something like the following:

```
12876842089683.48
```

This number represents the difficulty level of the Bitcoin network. Recall from previous sections that miners compete to find a solution to a problem. This number, in fact, shows how difficult it is to find a hash lower than the network difficulty target. All successfully mined blocks must contain a hash that is less than this target number. This number is updated every 2 weeks or 2,016 blocks to ensure that on average, the 10-minute block generation

time is maintained. Bitcoin network difficulty has increased in a roughly exponential fashion. The following graph shows this difficulty level over a period of one year:

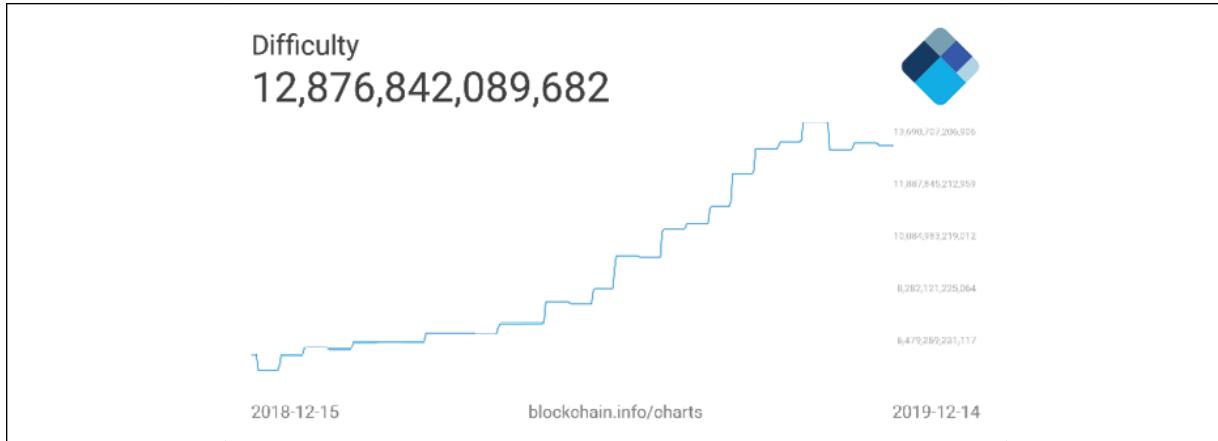


Figure 6.20: Mining difficulty since late 2018

The preceding graph shows the difficulty of the Bitcoin network over last year, and it has increased quite significantly. The reason why mining difficulty increases is because, in Bitcoin, the block generation time always has to be around 10 minutes. This means that if blocks are being mined too quickly because of faster hardware, the difficulty increases accordingly so that the block generation time can remain at roughly 10 minutes per block. This phenomenon is also true in reverse. If blocks take longer than 10 minutes on average to mine, then the difficulty is decreased. The difficulty is calculated every 2,016 blocks (around two weeks) and adjusted accordingly. If the previous set of 2,016 blocks were mined in a period of less than two weeks, then the difficulty increases. Similarly, if 2,016 blocks were found in more than two weeks (bearing in mind that if blocks are mined every 10 minutes, then 2,016 blocks take 2 weeks to be mined), then the difficulty is decreased.

The Bitcoin miners have to calculate hashes to solve the PoW algorithm. If the difficulty goes up then a higher hash rate is required to find the blocks. The difficulty increases accordingly if more hashing power is added due to more miners joining the network. Now let's explain the hash rate in a bit more detail.

The hash rate

The hash rate basically represents the rate of hash calculation per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block. In the early days of Bitcoin, it used to be quite small, as CPUs were used, which are relatively weak in mining terms. However, with dedicated mining pools and **Application Specific Integrated Circuits (ASICs)** now, this has gone up exponentially in the last few years. This has resulted in increased difficulty in the Bitcoin network.

The following hash rate graph shows the hash rate increases over time and is currently measured in exa-hashes. This means that in 1 second, Bitcoin network miners are computing more than 24,000,000,000,000,000,000 hashes per second:

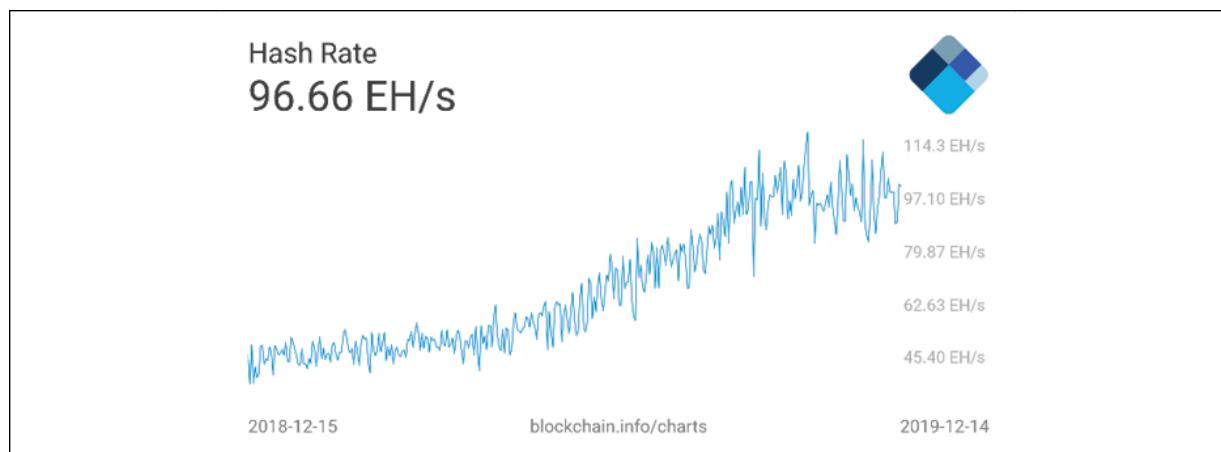


Figure 6.21: Hashing rate over time (measured in exa-hashes), shown over a period of 1 year

Mining systems

Over time, Bitcoin miners have used various methods to mine bitcoins. As the core principle behind mining is based on the double SHA256 algorithm, over time, experts have developed sophisticated systems to calculate the hash faster and faster. The following is a review of the different types of mining methods used in Bitcoin and how they evolved with time.

CPU

CPU mining was the first type of mining available in the original Bitcoin client. Users could even use laptop or desktop computers to mine bitcoins. CPU mining is no longer profitable and now more advanced mining methods such as ASIC-based mining are used. CPU mining only lasted for around a year from the introduction of Bitcoin, and soon other methods were explored and tried by the miners.

GPU

Due to the increased difficulty of the Bitcoin network and the general tendency of finding faster methods to mine, miners started to use the GPUs or graphics cards available in PCs to perform mining. GPUs support faster and parallelized calculations that are usually programmed using the OpenCL language.

This turned out to be a faster option as compared to CPUs. Users also used techniques such as overclocking to gain maximum benefit of the GPU power. Also, the possibility of using multiple graphics cards in parallel increased the popularity of graphics cards' usage for Bitcoin mining. GPU mining, however, has some limitations, such as overheating and the requirement for specialized motherboards and extra hardware to house multiple graphics cards. From another angle, graphics cards have become quite expensive due to increased demand and this has impacted gamers and graphics software users.

FPGA

Even GPU mining did not last long, and soon miners found another way to perform mining using **Field Programmable Gate Arrays (FPGAs)**. An FPGA is basically an integrated circuit that can be programmed to perform specific operations. FPGAs are usually programmed in **hardware description languages (HDLs)**, such as Verilog and VHDL. Double SHA256 quickly became an attractive programming task for FPGA programmers and several open source projects were started too. FPGA offered much better performance compared to GPUs; however, issues such

as accessibility, programming difficulty, and the requirement for specialized knowledge to program and configure FPGAs resulted in a short life for the FPGA era of Bitcoin mining.

Mining hardware such as the X6500 miner, Ztex, and Icarus were developed during the time when FPGA mining was profitable. Various FPGA manufacturers, such as Xilinx and Altera, produce FPGA hardware and development boards that can be used to program mining algorithms. The arrival of ASICs resulted in the quick phasing out of FPGA-based systems for mining. It should be noted that GPU mining is still profitable for some other cryptocurrencies to some extent, such as Zcoin (<https://zcoin.io/guide-on-how-to-mine-zcoin-xzc/>), but not Bitcoin, because the network difficulty of Bitcoin is so high that only ASICs (specialized hardware) running in parallel can produce reasonable profit.

ASICs

ASICs were designed to perform SHA-256 operations. These special chips were sold by various manufacturers and offered a very high hashing rate. This worked for some time, but due to the quickly increasing mining difficulty level, single-unit ASICs are no longer profitable.

Currently, mining is out of the reach of individuals due to the vast amounts of energy and money needed to be spent in order to build a profitable mining platform. Now, professional mining centers using thousands of ASIC units in parallel are offering mining contracts to users to perform mining on their behalf. There is no technical limitation—a single user can run thousands of ASICs in parallel—but it will require dedicated data centers and hardware, therefore the cost for a single individual can become prohibitive.

Examples of these four types of hardware are shown in the following photographs:

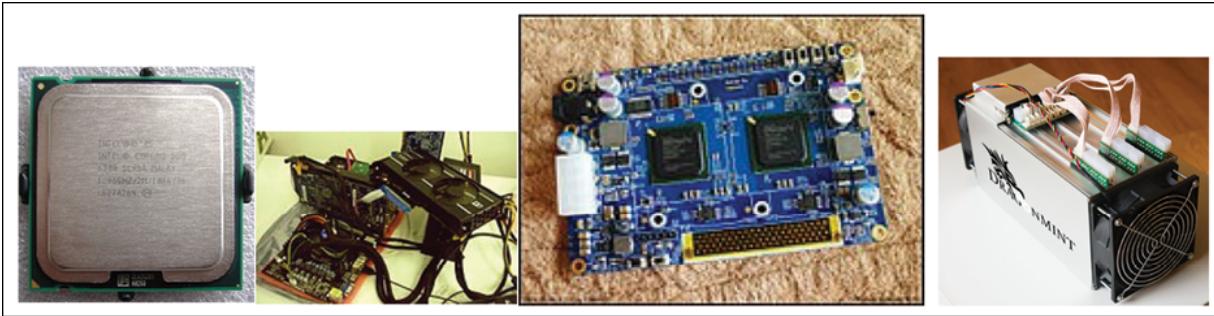


Figure 6.22: Four types of mining hardware (from left to right: a CPU, GPU, FPGA, and an ASIC)

Mining pools

A mining pool forms when a group of miners work together to mine a block. The pool manager receives the coinbase transaction if the block is successfully mined, and is then responsible for distributing the reward to the group of miners who invested resources to mine the block. This is more profitable than solo mining, where only one sole miner is trying to solve the partial hash inversion function (hash puzzle) because, in mining pools, the reward is paid to each member of the pool regardless of whether they (or more specifically, their individual node) solved the puzzle or not.

There are various models that a mining pool manager can use to pay to the miners, such as the pay-per-share model and the proportional model. In the pay-per-share model, the mining pool manager pays a flat fee to all miners who participated in the mining exercise, whereas in the proportional model, the share is calculated based on the amount of computing resources spent to solve the hash puzzle.

Many commercial pools now exist and provide mining service contracts via the cloud and easy-to-use web interfaces. The most commonly used ones are AntPool (<https://www.antpool.com>), BTC (<https://btc.com>), and BTC.TOP (<http://www.btc.top>). A comparison of the hashing power of all major mining pools is shown in the following pie chart:

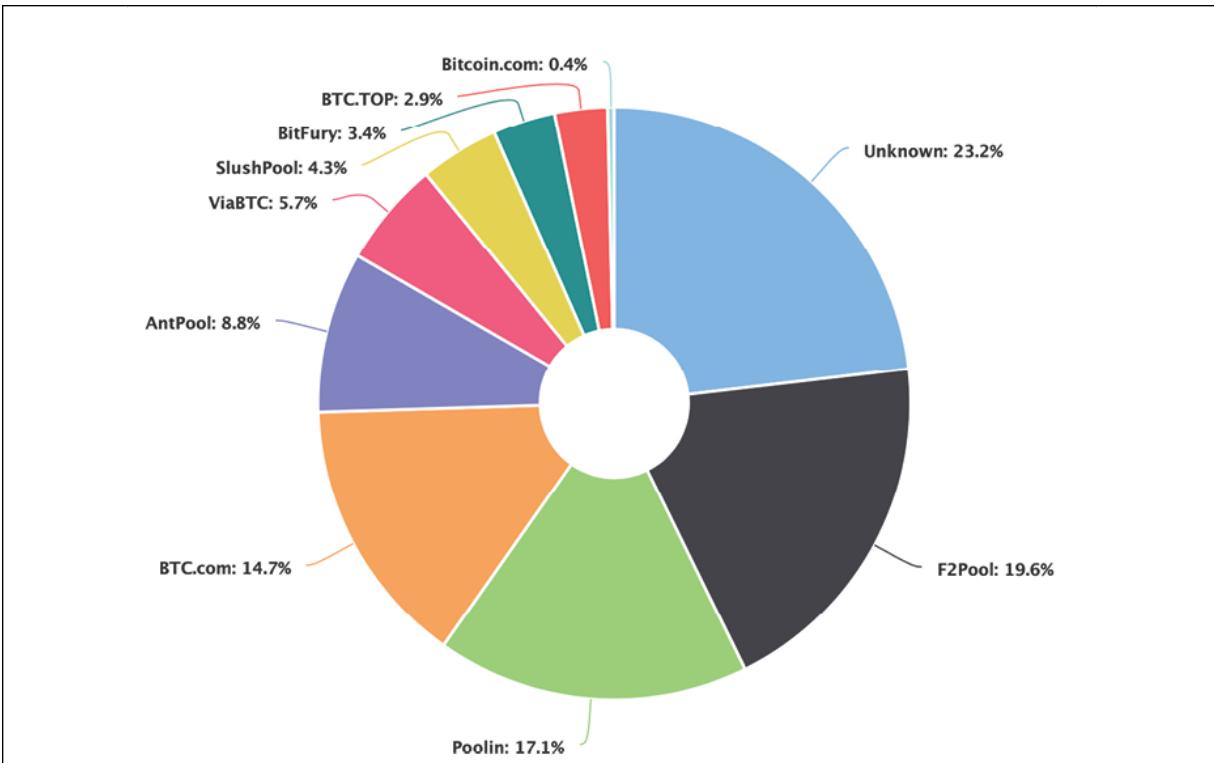


Figure 6.23: Mining pools and their hashing power (hash rate) as of late 2019. Source: <https://blockchain.info/pools>

Mining centralization can occur if a pool manages to control more than 51% of the network by generating more than 51% of the hash rate of the Bitcoin network.

As discussed earlier in the introduction section, a 51% attack can result in successful double-spending attacks, and it can impact consensus and in fact, even impose another version of the transaction history on the Bitcoin network.

This event has happened once in Bitcoin history when [GHash.IO](#), a large mining pool, managed to acquire more than 51% of the network capacity. Theoretical solutions, such as two-phase PoW, have been proposed in academia to disincentivize large mining pools.



The following article describes a two-phase PoW proposal, in order to disincentivize large Bitcoin mining pools:

<http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/>.

This scheme introduces a second cryptographic puzzle that results in mining pools either revealing their private keys or providing a considerable portion of the hash rate of their mining pool, thus reducing the overall hash rate of the pool.

Various types of hardware are commercially available for mining purposes. Currently, the most profitable one is ASIC mining, and specialized hardware is available from a number of vendors such as Antminer, AvalonMiner, and Whatsminer. Solo mining is not very profitable now, unless a vast amount of money and energy is spent to build your own mining rig or even a data center. With the current difficulty factor (as of June 2020), if a miner manages to produce a hash rate of 60 trillion hashes per second (TH/s), they can hope to make 0.000478 BTC (around \$0.01) per day, and around \$4 a year, which is extremely low compared to the investment required to source the equipment that can produce 60 TH/s. Including running costs such as electricity, this turns out to be not very profitable at all.

For example, the Antminer S9 is an efficient ASIC miner that produces a hash power of 13.5 TH/s and seems like it can produce some profit per day, which is true. However, a single Antminer S9 costs around 1,700 GBP, and when combined with the cost of electricity, the return on investment is only realized after almost a year's mining, when it produces around 0.3 BTC. It may still seem an OK investment, but also think about the fact that the Bitcoin network difficulty keeps going up with time, so over a year, it will become more difficult to mine and the mining hardware will expend its utility in a few months.

The race to build efficient miners is on and is only expected to grow. A recent addition to the ASIC miners' market is the DragonMint T1 miner (<https://halongmining.com/shop/dragonmint-16t-miner/>), which operates at 16 TH/s.

Summary

This chapter started with an introduction to Bitcoin and explained how a transaction works from the user's point of view. Then, an introduction to transactions from a technical point of view was presented. After this, the public and private keys used in Bitcoin were discussed. In the next section, we introduced addresses and their different types, followed by a discussion on transactions and their types and usage.

Following on from this, we looked at blockchain, with a detailed explanation of how blockchain works and the various components included in the Bitcoin blockchain. In the last few sections of the chapter, mining processes and relevant concepts such as hardware systems, their limitations, and bitcoin rewards were introduced.

In the next chapter, we will examine concepts related to the Bitcoin network, its elements, and client software tools.

The Bitcoin Network and Payments

In this chapter, we'll present the Bitcoin network, relevant network protocols, and wallets. We will explore different types of wallets that are available for Bitcoin. Moreover, we will examine how the Bitcoin protocol works, as well as the types of messages that are exchanged between nodes during various node and network operations. Also, we will explore some advanced and modern Bitcoin protocols that have been developed to address limitations in the original Bitcoin. Finally, we'll give you an introduction to Bitcoin trading and investment.

We will start with a detailed introduction to the Bitcoin network.

The Bitcoin network

The Bitcoin network is a **peer-to-peer (P2P)** network where nodes perform transactions. They verify and propagate transactions and blocks. Nodes called miners also produce blocks. There are different types of nodes on the network. The two main types of nodes are full nodes and **simple payment verification (SPV)** nodes. Full nodes, as the name implies, are implementations of Bitcoin Core clients performing the wallet, miner, full blockchain storage, and network routing functions. However, it is not necessary for all nodes in a Bitcoin network to perform all these functions. SPV nodes or lightweight clients perform only wallet and network routing functionality.



Versioning information is coded in the Bitcoin client in the `version.h` file, which is available here:

<https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/version.h#L9>.

Some nodes are full blockchain nodes with a complete blockchain as they are more secure and play a vital role in block propagation, while some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function). Another type of node is solo miner nodes, which can perform mining, store full blockchains, and act as Bitcoin network routing nodes.

There are a few nonstandard but heavily used nodes. These are called pool protocol servers. These nodes make use of alternative protocols, such as the stratum protocol. These nodes are used in mining pools. Nodes that only compute hashes use the stratum protocol to submit their solutions to the mining pool. Some nodes perform only mining functions and are called mining nodes. It is possible to run SPV software that runs a wallet and network routing function without a blockchain. SPV clients only download the headers of the blocks while syncing with the network. When required, they can request transactions from full nodes. Verifying transactions is possible by using a Merkle root in the block header with a Merkle branch to prove that the transaction is present in a block in the blockchain.

There are also different protocols that have been developed to facilitate communication between Bitcoin nodes. One such protocol is called **Stratum**. It is a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools.



Most protocols on the internet are line-based, which means that each line is delimited by a carriage return and newline `\r \n` character. More detail on this protocol are available at this link:

https://en.bitcoin.it/wiki/Stratum_mining_protocol.

A Bitcoin network is identified by its magic value. Magic values are used to indicate the message's origin network.

A list of these values is shown in the following table:

Network	Magic value (in hex)
main	0xD9B4BEF9
testnet	0xDAB5BFFA
testnet3	0x0709110B

A full Bitcoin node performs four functions. These are wallet, miner, blockchain, and network routing. We discussed mining and blockchains in the previous chapter, *Chapter 6, Introducing Bitcoin*. We will focus on the Bitcoin network protocol and wallets in this chapter.

Before we examine how the Bitcoin discovery protocol and block synchronization work, we need to understand the different types of messages that the Bitcoin protocol uses. Also, note that the latest version of the Bitcoin protocol is 70015, which was introduced with Bitcoin Core client 0.19.0.1.

There are 27 types of protocol messages in total, but they're likely to increase over time as the protocol grows. The most commonly used protocol messages and an explanation of them are listed as follows:

- **Version:** This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.
- **Verack:** This is the response of the version message accepting the connection request.
- **Inv:** This is used by nodes to advertise their knowledge of blocks and transactions.
- **Getdata:** This is a response to `inv`, requesting a single block or transaction identified by its hash.

- **Getblocks**: This returns an `inv` packet containing the list of all blocks starting after the last known hash or 500 blocks.
- **Getheaders**: This is used to request block headers in a specified range.
- **Tx**: This is used to send a transaction as a response to the `getdata` protocol message.
- **Block**: This sends a block in response to the `getdata` protocol message.
- **Headers**: This packet returns up to 2,000 block headers as a reply to the `getheaders` request.
- **Getaddr**: This is sent as a request to get information about known peers.
- **Addr**: This provides information about nodes on the network. It contains the number of addresses and address list in the form of an IP address and port number.
- **Ping**: This message is used to confirm if the TCP/IP network connection is active.
- **Pong**: This message is the response to a `ping` message confirming that the network connection is live.

When a Bitcoin Core node starts up, first, it initiates the discovery of all peers. This is achieved by querying DNS seeds that are hardcoded into the Bitcoin Core client and are maintained by Bitcoin community members. This lookup returns a number of DNS A records. The Bitcoin protocol works on TCP port `8333` by default for the main network and TCP `18333` for testnet.



DNS seeds are declared in the `chainparams.cpp` file in the Bitcoin source code, which can be viewed on GitHub at the following link:
<https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/chainparams.cpp#L116>.

First, the client sends a protocol message, `version`, which contains various fields, such as version, services, timestamp, network address, nonce, and

some other fields. The remote node responds with its own `version` message, followed by a `verack` message exchange between both nodes, indicating that the connection has been established.

After this, `getaddr` and `addr` messages are exchanged to find the peers that the client does not know. Meanwhile, either of the nodes can send a `ping` message to see whether the connection is still active. `getaddr` and `addr` are message types defined in the Bitcoin protocol.

This process is shown in the following diagram of the protocol:

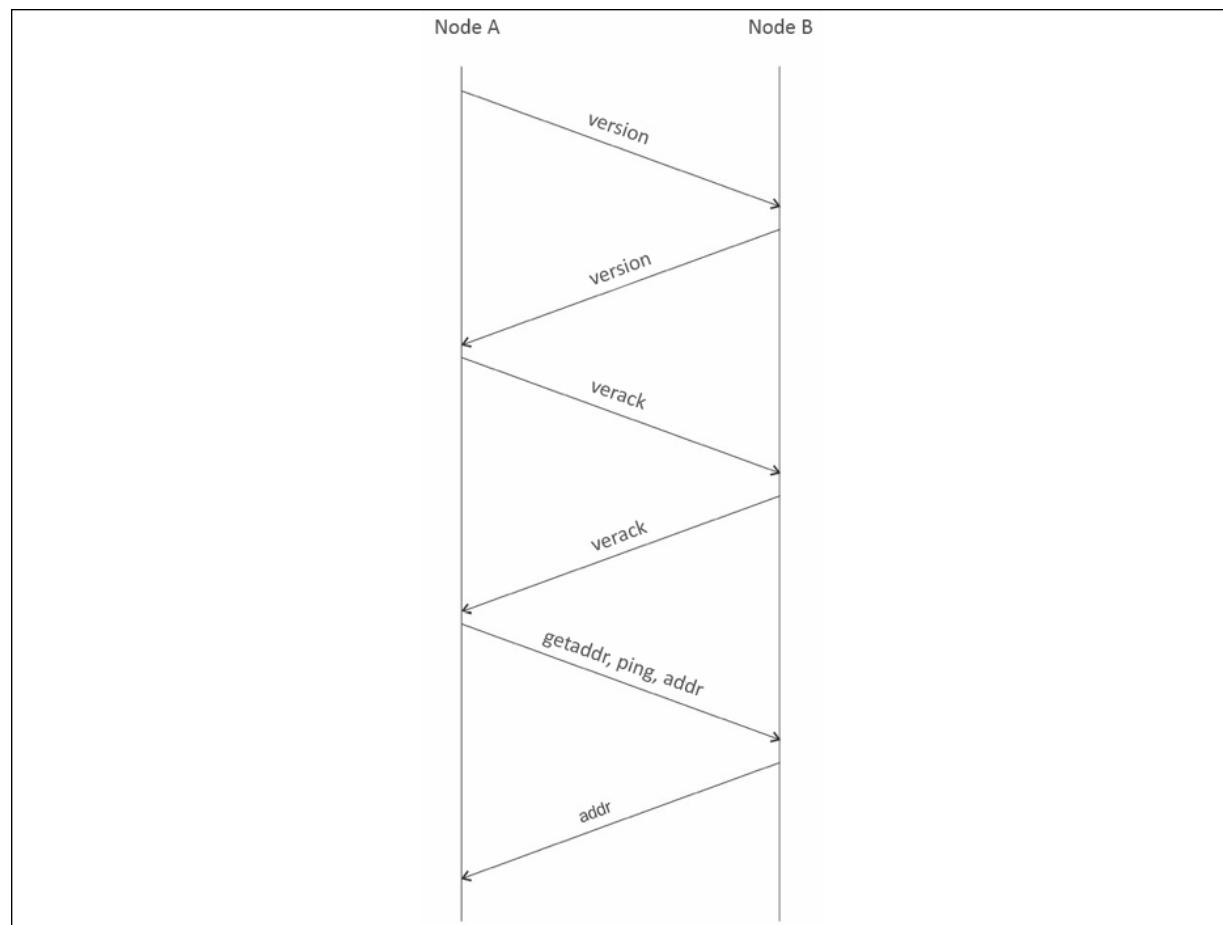


Figure 7.1: Visualization of node discovery protocol

This network protocol sequence diagram shows communication between two Bitcoin nodes during initial connectivity. **Node A** is shown on the left-hand side and **Node B** on the right. First, **Node A** starts the connection by sending a version message that contains the version number and current

time to the remote peer, **Node B**. **Node B** then responds with its own version message containing the version number and current time. **Node A** and **Node B** then exchange a `verack` message, indicating that the connection has been successfully established. After this connection is successful, the peers can exchange `getaddr` and `addr` messages to discover other peers on the network.

Now, the block download can begin. If the node already has all the blocks fully synchronized, then it listens for new blocks using the `inv` protocol message; otherwise, it first checks whether it has a response to `inv` messages and has inventories already. If it does, then it requests the blocks using the `getdata` protocol message; if not, then it requests inventories using the `getblocks` message. This method was used until version 0.9.3. This was a slower process known as the blocks-first approach and was replaced with the headers-first approach in 0.10.0.

The initial block download can use the blocks-first or headers-first method to synchronize blocks, depending on the version of the Bitcoin Core client. The blocks-first method is very slow and was discontinued on 16th February 2015 with the release of version 0.10.0.

Since version 0.10.0, the initial block download method named headers-first was introduced. This resulted in major performance improvement, and blockchain synchronization that used to take days to complete started taking only a few hours. The core idea is that the new node first asks peers for block headers and then validates them. Once this is completed, blocks are requested in parallel from all available peers. This happens because the blueprint of the complete chain is already downloaded in the form of the block header chain.

In this method, when the client starts up, it checks whether the blockchain is fully synchronized if the header chain is already synchronized; if not, which is the case the first time the client starts up, it requests headers from other peers using the `getheaders` message. If the blockchain is fully synchronized, it listens for new blocks via `inv` messages, and if it already has a fully synchronized header chain, then it requests blocks using `getdata` protocol messages. The node also checks whether the header

chain has more headers than blocks, and then it requests blocks by issuing the `getdata` protocol message:

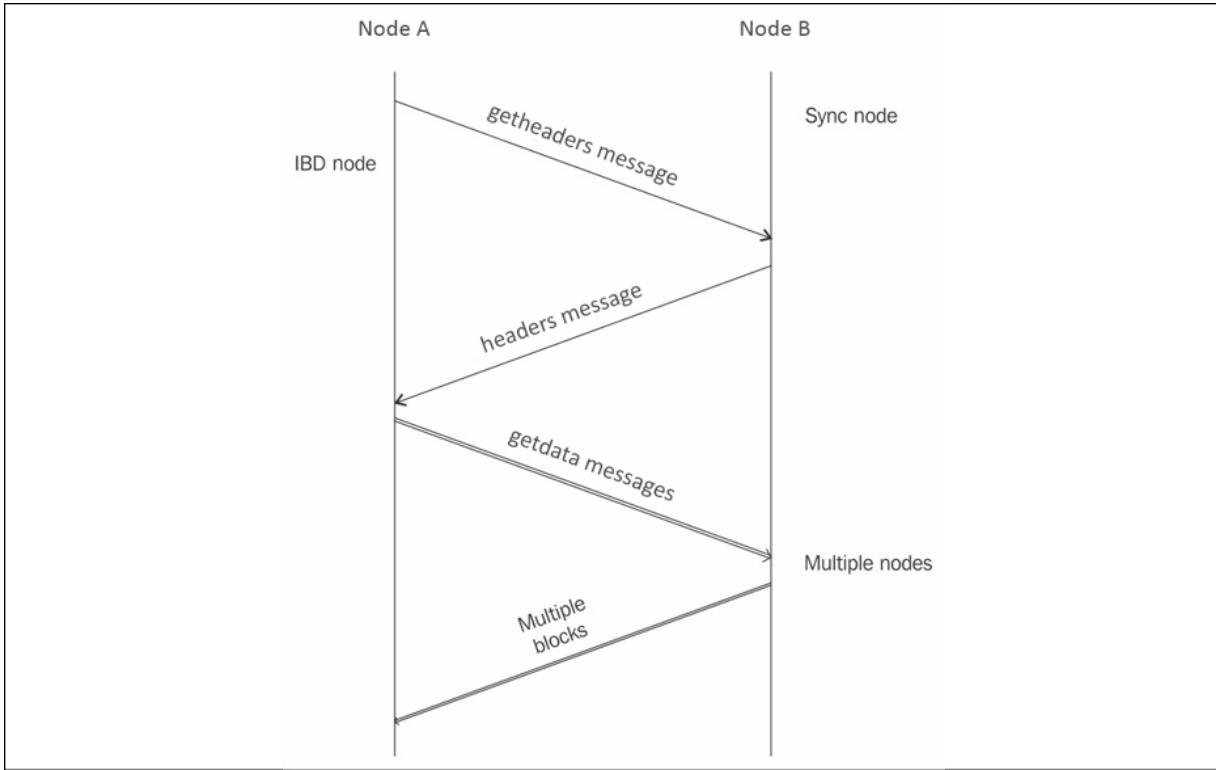


Figure 7.2: Bitcoin Core client >= 0.10.0 header and block synchronization

The preceding diagram shows the Bitcoin block synchronization process between two nodes on the Bitcoin network. **Node A**, shown on the left-hand side, is called an **Initial Block Download (IBD)** node, and **Node B**, shown on the right, is called a **sync node**.

IBD node means that this is the node that is requesting the blocks, while **sync node** means the node where the blocks are being requested from. The process starts by Node A first sending the `getheaders` message, which is met with a `getheaders` message response from the sync node. The payload of the `getheaders` message is one or more header hashes. If it's a new node, then there is only the first genesis block's header hash. Sync Node B replies by sending up to 2,000 block headers to IBD Node A. After this, the IBD node, Node A, starts to download more headers from Node B and blocks from multiple nodes in parallel; that is, it acts as the IBD and receives multiple blocks from multiple nodes, including Node B. If the sync

node does not have more headers than 2,000 when the IBD node makes a `getheaders` request, the IBD node sends a `getheaders` message to other nodes. This process continues in parallel until the blockchain synchronization is complete.

The `Getblockchaininfo` and `getpeerinfo` **Remote Procedure Calls (RPCs)** were updated with a new functionality to cater for this change. An RPC known as `getchaintips` is used to list all known branches of the blockchain. This also includes headers-only blocks. `Getblockchaininfo` is used to provide information about the current state of the blockchain. `getpeerinfo` is used to list both the number of blocks and the headers that are common between peers.

Wireshark can also be used to visualize message exchange between peers and can serve as an invaluable tool to learn about the Bitcoin protocol. A sample of this is shown here. This is a basic example showing the `version`, `verack`, `getaddr`, `ping`, `addr`, and `inv` messages.

In the details, valuable information such as the packet type, command name, and results of the protocol messages can be seen:

No.	Time	Source	Destination	Protocol	Length	Info
131	98.598526000	192.168.0.13	52.1.165.219	Bitcoin	192	version
150	99.180294000	192.168.0.13	52.1.165.219	Bitcoin	90	verack
151	99.180421000	192.168.0.13	52.1.165.219	Bitcoin	122	getaddr, ping
152	99.180715000	192.168.0.13	52.1.165.219	Bitcoin	1288	addr, getheaders[Malformed Packet]
486	112.053746000	192.168.0.13	52.1.165.219	Bitcoin	127	inv
818	143.630367000	192.168.0.13	52.1.165.219	Bitcoin	127	inv
1004	178.729768000	192.168.0.13	52.1.165.219	Bitcoin	127	inv

Transmission Control Protocol, Src Port: 52864 (52864), Dst Port: 18333 (18333), Seq: 207, Ack: 1291, Len: 1222

Bitcoin protocol

Packet magic: 0xb110907

Command name: addr

Payload Length: 31

Payload checksum: 0xa03fc07d

Address message

Count: 1

Address: afbd025800ffff...

Node services: 0x0000000000000000

..... = Network node: Not set

Node address: ::ffff:86.15.44.209 (::ffff:86.15.44.209)

Node port: 18333

Address timestamp: Oct 16, 2016 00:37:19.000000000 BST

Bitcoin protocol

Packet magic: 0xb110907

Command name: getheaders

Payload Length: 1029

Payload checksum: 0x4e54961d

Getheaders message

Count: 126

Starting hash: 1101001f152142abccc039503abc56b149bd56c2b3925b65...

Starting hash: 000000001980703bd53b0c7bf0ac995bccfeffd5cdc780...

Starting hash: 000000007ad1fed813d20301b1762895a2e5b08c8a58b3ea...

Starting hash: 000000003624c451f726a3e983d02279d9c7cf672d36f1d...

Figure 7.3: A sample block message in Wireshark

A protocol graph showing the flow of data between the two peers can be seen in the preceding screenshot. This can help you understand when a node starts up and what type of messages are used.

In the following example, the Bitcoin dissector is used to analyze the traffic and identify the Bitcoin protocol commands.

The exchange of messages such as `version`, `getaddr`, and `getdata` can be seen in the following example, along with the appropriate comment describing the message name.

This exercise can be very useful in order to learn about the Bitcoin protocol and it is recommended that the experiments be carried out on the Bitcoin testnet (<https://en.bitcoin.it/wiki/Testnet>), where various messages and transactions can be sent over the network and then be analyzed by Wireshark.



Wireshark is a network analysis tool and is available at
<https://www.wireshark.org>.

The analysis being performed here by Wireshark shows messages being exchanged between two nodes. If you look closely, you'll notice that the top three messages show the node discovery protocol that we introduced earlier:

Time	192.168.0.13 136.243.139.96	Comment
97.734135000	(57868) [version] → (18333)	Bitcoin: version
98.025045000	(57868) [verack] → (18333)	Bitcoin: verack
98.025177000	(57868) [getaddr.ping] → (18333)	Bitcoin: getaddr, ping, addr
98.025468000	(57868) [getheaders...] → (18333)	Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers
98.160419000	(57868) [TCP Retran...] → (18333)	Bitcoin: [TCP Retransmission], getheaders, [unknown command], [unknown command], [unknown command]
98.598399000	(57868) [getdata] → (18333)	Bitcoin: getdata
144.343544000	(57868) [inv] → (18333)	Bitcoin: inv
176.152240000	(57868) [getdata] → (18333)	Bitcoin: getdata
179.493755000	(57868) [getdata] → (18333)	Bitcoin: getdata
218.101646000	(57868) [ping] → (18333)	Bitcoin: ping
218.192004000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
218.444431000	(57868) [TCP Retran...] → (18333)	Bitcoin: [TCP Retransmission], [unknown command]
336.234936000	(57868) [getdata] → (18333)	Bitcoin: getdata
337.843423000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
338.143885000	(57868) [ping] → (18333)	Bitcoin: ping
448.764093000	(57868) [getdata] → (18333)	Bitcoin: getdata
457.894823000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
458.195265000	(57868) [ping] → (18333)	Bitcoin: ping
578.011774000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
578.212044000	(57868) [ping] → (18333)	Bitcoin: ping
585.587671000	(57868) [inv] → (18333)	Bitcoin: inv
647.169633000	(57868) [inv] → (18333)	Bitcoin: inv
671.962545000	(57868) [getdata] → (18333)	Bitcoin: getdata
698.037067000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
698.237350000	(57868) [ping] → (18333)	Bitcoin: ping
701.563581000	(57868) [inv] → (18333)	Bitcoin: inv
701.986269000	(57868) [inv] → (18333)	Bitcoin: inv
705.022173000	(57868) [inv] → (18333)	Bitcoin: inv
812.115878000	(57868) [inv] → (18333)	Bitcoin: inv
818.198570000	(57868) [Unknown co...] → (18333)	Bitcoin: [unknown command]
818.298733000	(57868) [ping] → (18333)	Bitcoin: ping

Figure 7.4: Bitcoin node discovery protocol in Wireshark

Nodes run different Bitcoin client software. The most common are full and SPV clients. We introduced these briefly at the start of this chapter, but we'll have a deeper look at these clients and related concepts, such as bloom filters, in the next sections.

Full client and SPV client

Bitcoin network nodes can fundamentally operate in two modes: full client or lightweight SPV client. Full clients are thick clients or full nodes that download the entire blockchain; this is the most secure method of validating the blockchain as a client. SPV clients are used to verify payments without requiring the download of a full blockchain. SPV nodes only keep a copy of block headers of the current longest valid blockchain. Verification is performed by looking at the Merkle branch, which links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more pragmatic approach, which was implemented with

BIP37 (you can see this at <https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki>), where bloom filters were used to filter for relevant transactions only.

Bloom filters

A bloom filter is a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It provides probabilistic lookup with false positives but no false negatives. This means that this filter can produce an output where an element that is not a member of the set being tested is wrongly considered to be in the set. Still, it can never produce an output where an element does exist in the set, but it asserts that it does not. In other words, false positives are possible, but false negatives are not.

Elements are added to the bloom filter after hashing them several times and then setting the corresponding bits in the bit vector to 1 via the corresponding index. To check the presence of an element in the bloom filter, the same hash functions are applied and then compared with the bits in the bit vector to see whether the same bits are set to 1.

Note that not every hash function (such as SHA1) is suitable for bloom filters as they need to be fast, independent, and uniformly distributed. The most commonly used hash functions for bloom filters are `fnv`, `murmur`, and `Jenkins`.

These filters are mainly used by simple payment verification SPV clients to request transactions and the Merkle blocks that they are interested in. A Merkle block is a lightweight version of the block, which includes a block header, some hashes, a list of 1-bit flags, and a transaction count. This information can then be used to build a Merkle tree. This is achieved by creating a filter that matches only those transactions and blocks that have been requested by the SPV client. Once version messages have been exchanged and the connection is established between the peers, the nodes can set filters according to their requirements.

These probabilistic filters offer a varying degree of privacy or precision, depending on how accurately or loosely they have been set. A strict bloom filter will only filter transactions that have been requested by the node, but at the expense of the possibility of revealing the user addresses to adversaries who can correlate transactions with their IP addresses, thus compromising privacy.

On the other hand, a loosely set filter can result in retrieving more unrelated transactions but will offer more privacy. Also, for SPV clients, bloom filters allow them to use low bandwidth as opposed to downloading all transactions for verification.

BIP37 proposed the Bitcoin implementation of bloom filters and introduced three new messages to the Bitcoin protocol:

- `filterload` : This is used to set the bloom filter on the connection.
- `filteradd` : This adds a new data element to the current filter.
- `filterclear` : This deletes the currently loaded filter.



More details can be found in the BIP37 specification. This is available at <https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki>.

Now, we'll move to a different but relevant topic. So far, we've discussed that, on a Bitcoin network, there are full clients (nodes), which perform the function of storing a complete blockchain. If you cannot run a full node, then SPV clients can be used to verify that particular transactions are present in a block by only downloading the block headers instead of the entire blockchain. At times, even running an SPV node is not feasible (especially on low-resource devices such as mobile phones) and the requirement is only to be able to send and receive Bitcoin somehow. For this purpose, wallets (wallet software) are used that do not require downloading even the block headers. We'll introduce wallets and some different types next.

Wallets

The wallet software is used to generate and store cryptographic keys. It performs various useful functions, such as receiving and sending Bitcoin, backing up keys, and keeping track of the balance available. Bitcoin client software usually offers both functionalities: Bitcoin client and wallet. On disk, the Bitcoin Core client wallets are stored as a `Berkeley DB` file:

```
$ file wallet.dat
wallet.dat: Berkeley DB (Btree, version 9, native byte-order)
```

Private keys are generated by randomly choosing a 256-bit number provided by the wallet software. The rules of generation are predefined and were discussed in *Chapter 4, Public Key Cryptography*. Private keys are used by wallets to sign the outgoing transactions. Wallets do not store any coins, and there is no concept of wallets storing balance or coins for a user. In fact, in the Bitcoin network, coins do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the number of bitcoins.

In Bitcoin, there are different types of wallets that can be used to store private keys. As software, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network. Let's take a look at the common types of wallets.

Non-deterministic wallets

These wallets contain randomly generated private keys and are also called **Just a Bunch of Key** wallets. The Bitcoin Core client generates some keys when first started and also generates keys as and when required. Managing a large number of keys is very difficult and an error-prone process that can lead to the theft and loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them in order to prevent theft or loss.

Deterministic wallets

In this type of wallet, keys are derived from a seed value via hash functions. This seed number is generated randomly and is commonly represented by human-readable **mnemonic code** words. Mnemonic code words are defined in BIP39, a Bitcoin improvement proposal for Mnemonic code for generating deterministic keys. This BIP is available at <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>. These phrases can be used to recover all keys and make private key management comparatively easier.

Hierarchical deterministic wallets

Defined in BIP32 and BIP44, HD wallets store keys in a tree structure derived from a seed. The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys. The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known. It is because of this property that HD wallets are very easy to maintain and are highly portable. There are many free and commercially available HD wallets available, for example, **Trezor** (<https://trezor.io>), **Jaxx** (<https://jaxx.io/>), and **Electrum** (<https://electrum.org/>).

Brain wallets

The master private key can also be derived from the hashes of passwords that are memorized. The key idea is that this passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as a brain wallet. This method is prone to password guessing and brute-force attacks,

but techniques such as key stretching can be used to slow down the progress made by the attacker.

Paper wallets

As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored.



Paper wallets can be generated online from various service providers, such as <https://bitcoinpaperwallet.com/> or <https://www.bitaddress.org/>.

Hardware wallets

Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built. With the advent of NFC-enabled phones, this can also be a **secure element (SE)** in NFC phones. Trezor and Ledger wallets (various types) are the most commonly used Bitcoin hardware wallets:



Figure 7.5: A Trezor wallet

Online wallets

Online wallets, as the name implies, are stored entirely online and are provided as a service usually via the cloud. They provide a web interface to the users to manage their wallets and perform various functions, such as making and receiving payments. They are easy to use but imply that the user trusts the online wallet service provider. An example of an online wallet is **GreenAddress**, which is available at <https://greenaddress.it/en/>.

Mobile wallets

Mobile wallets, as the name suggests, are installed on mobile devices. They can provide us with various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments.

Mobile wallets are available for Android and iOS and include Blockchain Wallet, Breadwallet, Copay, and Jaxx:



Figure 7.6: Jaxx mobile wallet

The choice of Bitcoin wallet depends on several factors such as security, ease of use, and available features. Out of all these attributes, security, of course, comes first, and when deciding about which wallet to use, security

should be of paramount importance. Hardware wallets tend to be more secure compared to web wallets because of their tamper-resistant design. Web wallets, by their very nature, are hosted on websites, which may not be as secure as a tamper-resistant hardware device.

Generally, mobile wallets for smart phone devices are quite popular due to a balanced combination of features and security. There are many companies offering these wallets on the iOS App Store and Google Play. It is, however, quite difficult to suggest which type of wallet should be used as it also depends on personal preferences and the features available in the wallet. Security should be kept in mind while making a decision on which wallet to choose.



SPV client, which we introduced earlier, is also a type of Bitcoin wallet. Other types of Bitcoin wallets, especially mobile wallets, are mostly API wallets that rely on a mechanism where private and public keys are stored locally on the device where the wallet software is installed. Here, trusted backend servers are used for providing blockchain data via APIs.

Now that we've discussed the topic of Bitcoin wallets, let's see how Bitcoin payments can be adopted for business.

Bitcoin payments

Bitcoin can be accepted as payment using various techniques. Bitcoin is not recognized as a legal currency in many jurisdictions, but it is increasingly being accepted as a payment method by many online merchants and e-commerce websites. There are a number of ways in which buyers can pay a business that accepts Bitcoin. For example, on an online shop, Bitcoin merchant solutions can be used, whereas in traditional, physical shops, **Point of Sale (POS)** terminals and other specialized hardware can be used. Customers can simply scan the QR barcode with the seller's payment URI in it and pay using their mobile devices. Bitcoin URIs allow users to make payments by simply clicking on links or scanning QR codes. A **Uniform**

Resource Identifier (URI) is basically a string that represents the transaction information. It is defined in BIP21. The QR code can be displayed near the point of the sale terminal. Nearly all Bitcoin wallets support this feature.

Businesses can use the following image to advertise that they can accept Bitcoin as payment from customers:



Figure 7.7: "Bitcoin accepted here" logo

Various payment solutions, such as **XBT terminal** and the 34 Bytes Bitcoin POS terminal, are available commercially.

Generally, these solutions work by following these steps:

1. The salesperson enters the amount of money to be charged in fiat currency; for example, US dollars.
2. Once the value is entered in the system, the terminal prints a receipt with a QR code on it and other relevant information, such as the amount to be paid.
3. The customer can then scan this QR code using their mobile Bitcoin wallet to send the payment to the Bitcoin address of the seller embedded within the QR code.
4. Once the payment is received at the designated Bitcoin address, a receipt is printed out as physical evidence of the sale.

A Bitcoin POS device from **34 Bytes** is shown in the following image:



Figure 7.8: 34 Bytes POS solution

Bitcoin payment processors are offered by many online service providers. It allows integration with e-commerce websites to facilitate Bitcoin payments. These payment processors can be used to accept Bitcoin as payment. Some service providers also allow secure storage of Bitcoin, for example, **BitPay** (<https://bitpay.com>). Another example is the Bitcoin merchant solutions available at <https://www.bitcoin.com/merchant-solutions>.

Various BIPs have been proposed and finalized in order to introduce and standardize Bitcoin payments. Most notably, BIP70 (secure payment protocol) describes the protocol for secure communication between a merchant and customers. This protocol uses X.509 certificates for authentication and runs over HTTP and HTTPS. There are three messages in this protocol: **PaymentRequest**, **Payment**, and **PaymentACK**. The key features of this proposal are defense against man-in-the-middle attacks and secure proof of payment. Man-in-the-middle attacks can result in a scenario where the attacker is sitting between the merchant and the buyer, and it would seem to the buyer that they are talking to the merchant, but in fact, the man in the middle is interacting with the buyer instead of the merchant. This can result in manipulation of the merchant's Bitcoin address to defraud the buyer.

Several other BIPs, such as BIP71 (Payment Protocol MIME types) and BIP72 (URI extensions for Payment Protocol), have also been implemented to standardize payment scheme to support BIP70 (Payment Protocol).

Another innovative development is the Lightning Network. It is a solution for scalable off-chain instant payments. It was introduced in early 2016 and allows off-blockchain payments. This network makes use of payments channels that run off the blockchain, which allows greater speed and scalability of Bitcoin.



This paper is available at <https://lightning.network/> and those of you who are interested are encouraged to read the paper in order to understand the theory and rationale behind this invention.

So far, we've covered a lot of concepts related to Bitcoin payments and pertinent concepts and techniques. We have completed our discussion regarding Bitcoin payments here and will now move on to the different but important topic of innovation. Bitcoin, and blockchain technology in general, are under continuous evolution, and we'll discuss some of the most relevant ideas next.

Innovation in Bitcoin

Bitcoin has undergone many changes and is still evolving into a more and more robust and better system by addressing various weaknesses in the system. Performance has been a topic of hot debate among Bitcoin experts and enthusiasts for many years. As such, various proposals have been made in the last few years to improve Bitcoin performance, resulting in greater transaction speed, increased security, payment standardization, and overall performance improvement at the protocol level.

These improvement proposals are usually made in the form of **Bitcoin Improvement Proposals (BIPs)** or fundamentally new versions of Bitcoin protocols, resulting in new networks altogether. Some of the changes

proposed can be implemented via a soft fork, but a few need a hard fork and, as a result, give birth to a new currency.

In the following sections, we will look at the various BIPs that can be proposed for improvement in Bitcoin. Then, we will discuss some advanced protocols that have been proposed and implemented to address various weaknesses in Bitcoin.

Bitcoin Improvement Proposals

These documents, also referred to as **BIPs**, are used to propose improvements or inform the Bitcoin community about the improvements suggested, the design issues, or some aspects of the Bitcoin ecosystem. There are three types of BIPs:

- **Standard BIP:** Used to describe the major changes that have a major impact on the Bitcoin system; for example, block size changes, network protocol changes, or transaction verification changes.
- **Process BIP:** A major difference between standard and process BIPs is that standard BIPs cover protocol changes, whereas process BIPs usually deal with proposing a change in a process that is outside the core Bitcoin protocol. These are implemented only after a consensus among Bitcoin users.
- **Informational BIP:** These are usually used to just advise or record some information about the Bitcoin ecosystem, such as design issues.

Now that we've provided this brief discussion on Bitcoin improvement and evolution, let's look at some of the excellent ideas that have emerged from research and innovation efforts related to Bitcoin.

Advanced protocols

In this section, we'll introduce various advanced protocols that have been suggested or implemented for improving the Bitcoin protocol.

For example, transaction throughput is one of the critical issues that need a solution. The Bitcoin network can only process approximately three to seven transactions per second, which is a tiny number compared to other financial networks. For example, the Visa network can process approximately, on average, 24,000 transactions per second. PayPal can process approximately 200 transactions per second, whereas Ethereum can process up to, on average, 20. As the Bitcoin network has grown exponentially over the last few years, these issues have started to grow even further. The difference in processing speed is also shown in the following graph, which shows the scale of difference between Bitcoin and other networks' transaction speeds. The graph uses a logarithmic scale, which demonstrates the vast difference between the networks' transaction speeds:

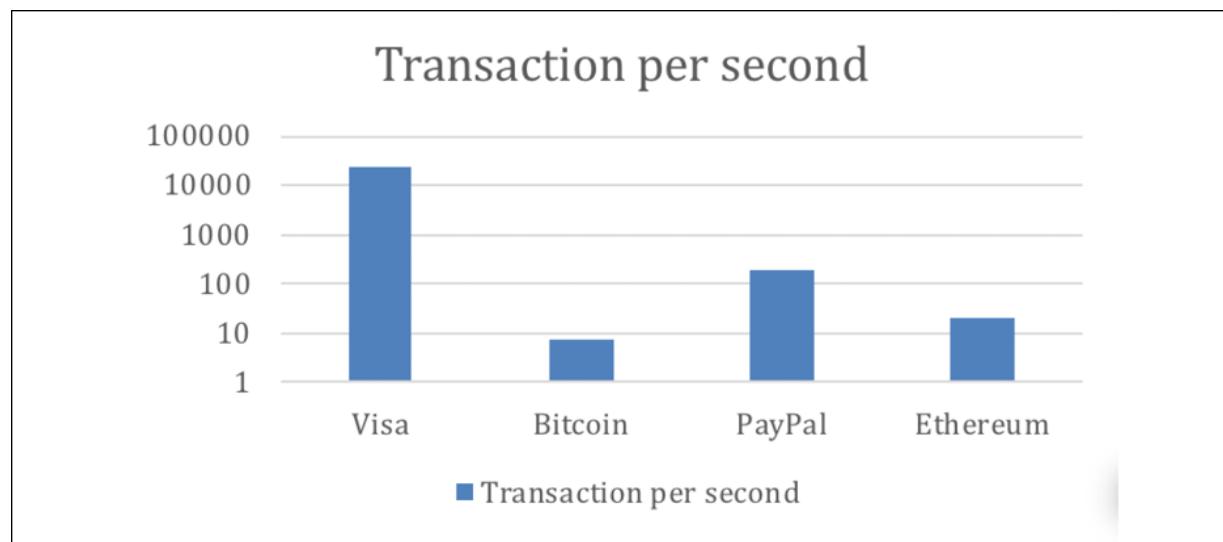


Figure 7.9: Bitcoin transaction speed compared to other networks (on a logarithmic scale)

Also, security issues such as transaction malleability are of real concern and can result in denial of service. Various proposals have been made to improve the Bitcoin proposal to address various weaknesses. A selection of these proposals will be presented here.

Segregated Witness

The **SegWit** or **Segregated Witness** is a soft fork-based upgrade of the Bitcoin protocol that addresses weaknesses such as throughput and security in the Bitcoin protocol. SegWit offers a number of improvements, as listed here:

- Fix for transaction malleability due to the separation of signature data from transactional data. In this case, it is no longer possible to modify the transaction ID because it is no longer calculated based on the signature data present within the transaction.
- By segregating the signature data and transaction data, lightweight clients do not need to download the transactions with all signatures unnecessarily. The transactions can be verified without the useless signatures, which allows for increased bandwidth efficiency.
- Reduction in transaction signing and verification times, which results in faster transactions. A new transaction hashing algorithm for signature verification has been introduced and is detailed in BIP0143 (https://en.bitcoin.it/wiki/BIP_0143). Due to this change, the verification time grows linearly with the number of inputs instead of in a quadratic manner, resulting in quicker verification time.
- Script versioning capability, which allows for easier script language upgrades. The version number is prefixed to the locking scripts to depict the version. This change allows upgrades and improvements to be made to the scripting language, without requiring a hard fork, by just increasing the version number of the script.
- Increased block size by introducing a weight limit instead of a size limit on the block and the removal of signature data. This concept will be explained in more detail shortly.
- An improved address format, also called a "bc1 address," which is encoded using the **Bech32** mechanism instead of base58. This improvement allows for better error detection and correction. Also, all characters are lowercase, which helps with readability. Moreover, this helps with distinguishing between legacy transactions and SegWit transactions. More information is available at this link:
<https://en.bitcoin.it/wiki/Bech32>.

SegWit was proposed in BIP 141, BIP 143, BIP 144, and BIP 145. It was activated on Bitcoin's main network on August 24, 2017 at block number 481824. The key idea behind SegWit is the separation of signature data from transaction data (that is, a transaction Merkle tree), which results in the size of the transaction being reduced. This change allows the block size to increase up to 4 MB in size. However, the practical limit is between 1.6 MB and 2 MB. Instead of a hard size limit of 1 MB blocks, SegWit introduced a new concept of a block weight limit.

Block weight is a new restriction mechanism where each transaction has a weight associated with it. This **weight** is calculated on a per-transaction basis. The formula used to calculate it is:

$$\text{Weight} = (\text{Transaction size without witness data}) \times 3 + (\text{Transaction size})$$

Blocks can have a maximum of four million weight units. As a comparison, a byte in a legacy 1 MB block is equivalent to 4 weight units, but a byte in a SegWit block weighs only 1 weight unit. This modification immediately results in increased block capacity.

To spend an **unspent transaction output (UTXO)** in Bitcoin, a valid signature needs to be provided. In the pre-SegWit scenario, this signature is provided within the locking script, whereas in SegWit this signature is not part of the transaction and is provided separately.

There are four types of transactions introduced by SegWit. These types are:

1. **Pay to Witness Public Key Hash (P2WPKH)**: This type of script is similar to the usual P2PKH, but the crucial difference is that the transaction signature used as a proof of ownership in ScriptSig is moved to a separate structure known as the "witness" of the input. The signature is the same as P2PKH but is no longer part of ScriptSig; it is simply empty. The PubKey is also moved to the witness field. This script is identified by a 20-byte hash. The ScriptPubKey is modified to a simpler format, as shown here:
 - P2PKH ScriptPubKey:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

- P2WPKH ScriptPubKey:

```
OP_0 <pubKeyHash>
```

2. Pay to Script Hash - Pay to Witness PubKey Hash (P2SH-P2WPKH):

This is a mechanism introduced to make SegWit transactions backward-compatible. This is made possible by nesting the P2WPKH inside the usual P2SH.

3. Pay to Witness Script Hash (P2WSH):

This script is similar to legacy P2SH but the signature and redeem script are moved to the separate witness field. This means that ScriptSig is simply empty. This script is identified by a 32-byte SHA-256 hash. P2WSH is a simpler script compared to P2SH and has just two fields. The ScriptPubKey is modified as follows:

- P2SH ScriptPubKey:

```
OP_HASH160 <pubKeyHash> OP_EQUAL
```

- P2WSH ScriptPubKey:

```
OP_0 <pubKeyHash>
```

4. Pay to Script Hash - Pay to Witness Script Hash (P2SH-P2WSH):

Similar to P2SH-P2WPKH, this is a mechanism that allows backward-compatibility with legacy Bitcoin nodes.

SegWit adoption is still in progress as not all users of the network agree or have started to use SegWit.

Next, we'll introduce some other innovative ideas in the Bitcoin space. Not only has the original Bitcoin evolved quite significantly since its introduction, but there are also new blockchains that are either forks of Bitcoin or novel implementations of the Bitcoin protocol with advanced features.

Bitcoin Cash

Bitcoin Cash (BCH) increases the block limit to 8 MB. This change immediately increases the number of transactions that can be processed in one block to a much larger number compared to the 1 MB limit in the original Bitcoin protocol. It uses **Proof of Work (PoW)** as a consensus algorithm, and mining hardware is still ASIC-based. The block interval is changed from 10 minutes to 10 seconds and up to 2 hours. It also provides replay protection and wipe-out protection, which means that because BCH uses a different hashing algorithm, it prevents it being replayed on the Bitcoin blockchain. It also has a different type of signature compared to Bitcoin to differentiate between two blockchains.

The BCH wallet and relevant information is available on their website:
<https://www.bitcoincash.org>.

Bitcoin Unlimited

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



Its client is available for download at
<https://www.bitcoinunlimited.info>.

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

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

Bitcoin Gold

This proposal has been implemented as a hard fork since block 491407 of the original Bitcoin blockchain. Being a hard fork, it resulted in a new blockchain, named Bitcoin Gold. The core idea behind this concept is to address the issue of mining centralization, which has hurt the original Bitcoin idea of decentralized digital cash, whereby more hash power has resulted in a power shift toward miners with more hashing power. Bitcoin Gold uses the Equihash algorithm as its mining algorithm instead of PoW; hence, it is inherently ASIC resistant and uses GPUs for mining.

Bitcoin Gold and relevant information is available at
<https://bitcoingold.org>.

There are other proposals like Bitcoin Next Generation, Solidus, Spectre, and Segwit2x, which will be discussed later in this book in *Chapter 21, Scalability and Other Challenges*, in the context of performance improvement in blockchain networks.

Now that we have covered some new blockchains and implementations of the Bitcoin protocol, let's introduce some basic concepts around Bitcoin investment and how to sell and buy Bitcoin.

Bitcoin investment and buying and selling Bitcoin

There are many online exchanges where users can buy and sell Bitcoin. This is a big business on the internet now and it offers Bitcoin trading, CFDs, spread betting, margin trading, and various other choices. Traders can buy Bitcoin or trade by opening long or short positions to make a profit when the price of Bitcoin goes up or down. Several other features, such as exchanging Bitcoin for other virtual currencies, are also possible, and many online Bitcoin exchanges provide this function. Advanced market data, trading strategies, charts, and relevant data to support traders is also available. An example is shown from CEX (<https://cex.io>) here. Other exchanges offer similar types of services:

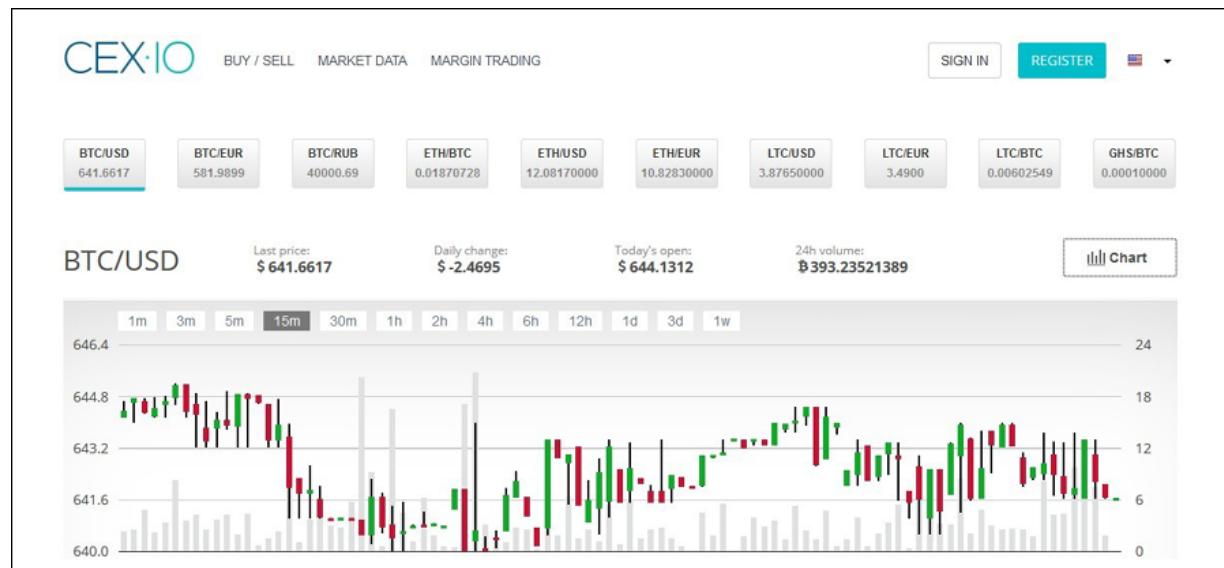


Figure 7.10: Example of the Bitcoin exchange on cex.io

The following screenshot shows the order book at the exchange, where all buy and sell orders are listed:

Sell Orders			Buy Orders		
Price per BTC	BTC Amount	Total: (USD)	Price per BTC	BTC Amount	Total: (USD)
642.4085	฿0.20450000	\$ 131.38	641.6210	฿0.01390000	\$ 8.92
642.4915	฿0.20910000	\$ 134.35	641.6201	฿0.23162780	\$ 148.62
643.4470	฿0.05000000	\$ 32.18	641.6200	฿0.12050000	\$ 77.32
643.4900	฿0.11944972	\$ 76.87	641.6117	฿1.83477084	\$ 1177.22
643.5000	฿1.85748652	\$ 1195.30	641.5584	฿0.30000000	\$ 192.47
643.6500	฿3.00000000	\$ 1930.95	641.5217	฿0.18180000	\$ 116.63
643.6999	฿0.13844181	\$ 89.12	641.0217	฿0.10000000	\$ 64.11
643.7000	฿45.80000000	\$ 29481.46	640.5300	฿0.67323160	\$ 431.23
643.7487	฿1.22995538	\$ 791.79	640.5000	฿0.40815400	\$ 261.43

Figure 7.11: Example of a Bitcoin order book at the exchange of cex.io

The order book shown here displays sell and buy orders. Sell orders are also called ask orders, while buy orders are also called bid orders. This means that the ask price is what the seller is willing to sell the bitcoin at, whereas the bid price is what the buyer is willing to pay. If the bid and ask prices match, then a trade can occur. The most common order types are market orders and limit orders. Market orders mean that as soon as the prices match, the order will be fulfilled immediately. Limit orders allow for buying and selling a set number of bitcoins at a specified price or better. Also, a period of time can be set, during which the order can be left open. If it's not executed, then it will be canceled. We will introduce trading concepts in more detail in *Chapter 18, Tokenization*.

Summary

We started this chapter with an introduction to the Bitcoin network, followed by a discussion on Bitcoin node discovery and block synchronization protocols. Moreover, we presented different types of network messages. Then, we examined different types of Bitcoin wallets and discussed the various attributes and features of each type. Following this, we looked at Bitcoin payments and payment processors. In the last section, we discussed Bitcoin innovations, which included topics such as

BIPs and advanced Bitcoin protocols. Finally, we presented a basic introduction to Bitcoin buying and selling.

In the next chapter, we will discuss Bitcoin clients, such as the Bitcoin Core client, which can be used to interact with the Bitcoin blockchain and also acts as a wallet. In addition, we will explore some of the APIs that are available for programming Bitcoin applications.

Bitcoin Clients and APIs

This chapter introduces Bitcoin client installation, and a basic overview of various APIs and tools that are available for developing Bitcoin applications and interacting with the Bitcoin blockchain. We will examine how to set up a Bitcoin node and test networks. Also, we will introduce various commands and tools that are used to perform important functions in Bitcoin systems. The chapter will be made up of the following main topics:

- Bitcoin client installation
- Experimenting further with bitcoin-cli
- Bitcoin programming

Let's begin by running through the Bitcoin client installation process.

Bitcoin client installation

The Bitcoin core client can be installed from <https://bitcoin.org/en/download>. This is available for different architectures and platforms, ranging from x86 Windows to ARM Linux, as shown in the following screenshot:

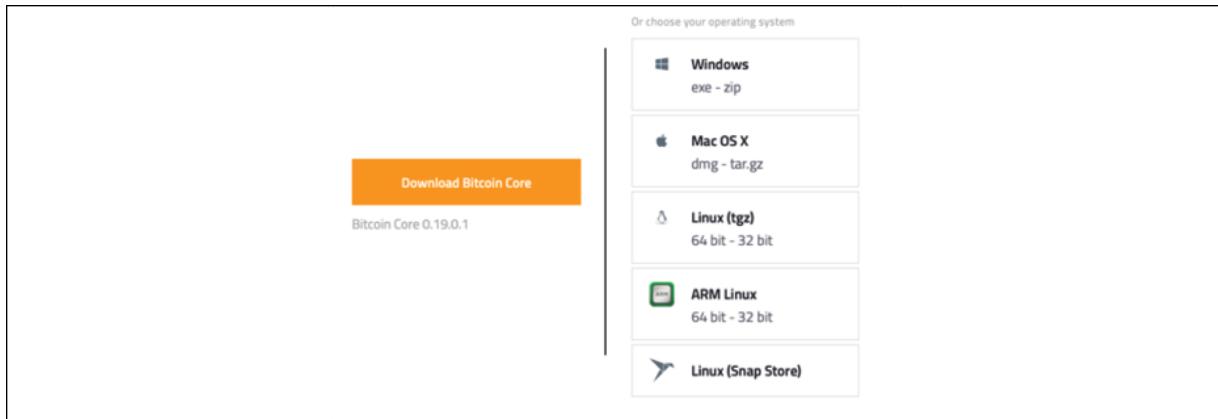


Figure 8.1: Download Bitcoin Core

We will discuss a number of topics relating to Bitcoin's installation and setup. We'll begin by discussing different Bitcoin clients available and their associated tools, which enable you to run and manage the Bitcoin client and interact with the Bitcoin blockchain.

Types of clients and tools

There are different types of Bitcoin core clients and relevant tools. A Bitcoin client is a piece of software that is responsible for generating private/public key pairs and facilitates Bitcoin payments using the Bitcoin blockchain. In addition, a client can implement full synchronization function with a blockchain or choose to only implement basic wallet functionality or simple payment verification. A client can also provide other useful functions such as network monitoring, secure storage of keys, and user-friendly interfaces for interaction with the Bitcoin blockchain. Next, we'll discuss some of the core elements of the Bitcoin Core client and associated tools.

bitcoind

This is the core client software that runs as a daemon (as a service), and it provides the JSON-RPC interface.

bitcoin-cli

This is the command-line feature-rich tool for interacting with the Bitcoin daemon; the daemon then interacts with the blockchain and performs various functions. `bitcoin-cli` only calls JSON-RPC functions and does not perform any actions on its own on the blockchain.

bitcoin-qt

This is the Bitcoin Core client GUI. When the wallet software starts up, first, it verifies the blocks on the disk, and then starts up and shows the following GUI:

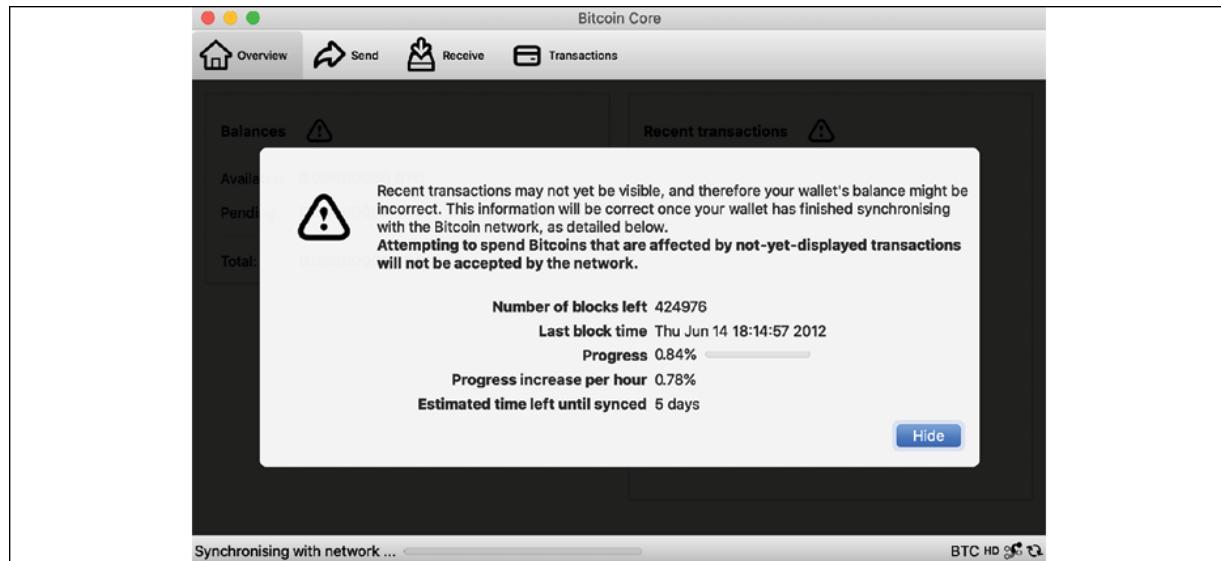


Figure 8.2: Bitcoin Core QT client, just after installation on macOS, showing that blockchain is not in sync

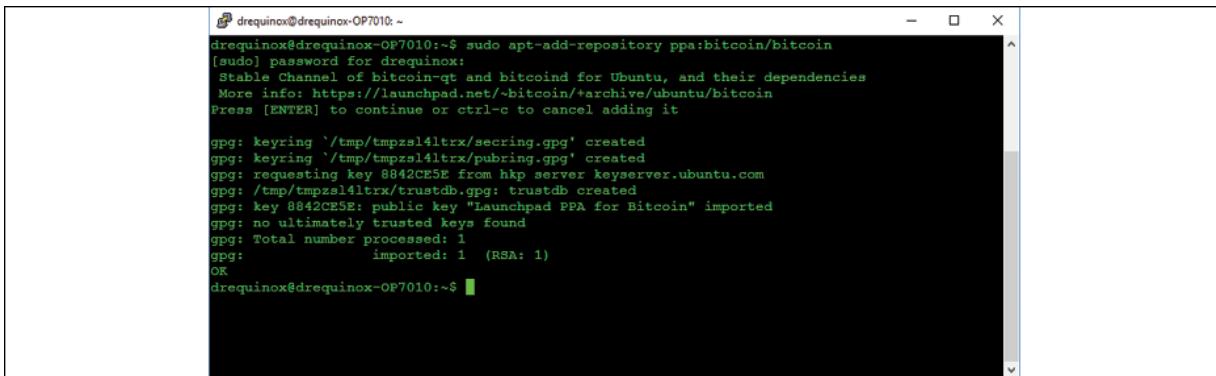
The verification process is not specific to the `bitcoin-qt` client; it is performed by the `bitcoind` client as well.

There are also other clients available such as `btcd`, which is a full node Bitcoin client written in Golang. It is available at <https://github.com/btcsuite/btcd>.

Setting up a Bitcoin node

In this section, we will explore how we can set up a Bitcoin node in order to interact with the Bitcoin network, and to interact with a Bitcoin node using the command-line interface.

A sample run of the Bitcoin Core installation on the Ubuntu Xenial operating system is shown here; for other platforms, you can get details from www.bitcoin.org:



The screenshot shows a terminal window with the following text:

```
drequinox@drequinox-OP7010:~$ sudo apt-add-repository ppa:bitcoin/bitcoin
[sudo] password for drequinox:
Stable Channel of bitcoin-qt and bitcoind for Ubuntu, and their dependencies
More info: https://launchpad.net/~bitcoin/+archive/ubuntu/bitcoin
Press [ENTER] to continue or ctrl-c to cancel adding it

gpg: keyring '/tmp/tmpzs14ltrx/secreting.gpg' created
gpg: keyring '/tmp/tmpzs14ltrx/pubring.gpg' created
gpg: requesting key 8842CE5E from hkp server keyserver.ubuntu.com
gpg: /tmp/tmpzs14ltrx/trustdb.gpg: trustdb created
gpg: key 8842CE5E: public key "Launchpad PPA for Bitcoin" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:           imported: 1  (RSA: 1)
OK
drequinox@drequinox-OP7010:~$
```

Figure 8.3: Bitcoin setup

We'll start by running the `update` command, which is as follows:

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

For installation, users can use either of the following first two commands depending on the client required, or they can issue both commands at once, using the third line only:

```
$ sudo apt-get install bitcoind
$ sudo apt-get install bitcoin-qt
$ sudo apt-get install bitcoin-qt bitcoind
Reading package lists... Done
Building dependency tree
Reading state information... Done
.....
```

Having set up a Bitcoin node, the next step is to set up the Bitcoin source code, which we'll cover next.

Setting up the source code

The Bitcoin source code can be downloaded and compiled if users wish to participate in the Bitcoin code or for learning purposes. The `git` command can be used to download the Bitcoin source code:

```
$ sudo apt-get install git  
$ mkdir bcsource  
$ cd bcsource  
$ git clone https://github.com/bitcoin/bitcoin.git  
Cloning into 'bitcoin'...  
remote: Counting objects: 78960, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 78960 (delta 0), reused 0 (delta 0), pack-reused 7  
Checking connectivity... done.
```

Change the directory to `bitcoin`:

```
$ cd bitcoin
```

After the preceding steps are completed, the code can be compiled:

```
$ ./autogen.sh  
$ ./configure.sh  
$ make  
$ sudo make install
```



Note that the `make` command shown here may take around 30 minutes to complete, depending on the speed of your computer.

Setting up bitcoin.conf

The `bitcoin.conf` file is a configuration file that is used by the Bitcoin Core client to save configuration settings. All command-line options for the `bitcoind` client, with the exception of the `-conf` switch, can be set up in the configuration file, and when `bitcoin-qt` or `bitcoind` starts up, it will take the configuration information from that file.

In Linux systems, this is usually found in `$HOME/.bitcoin/`, but it can also be specified in the command line using the `-conf=<file>` switch to the `bitcoind` core client software.

A detailed example configuration is available at:

<https://github.com/bitcoin/bitcoin/blob/master/share/examples/bitcoin.conf>

Now that we have set up the Bitcoin client, let's see how to start up the Bitcoin client for use.

Starting up a node in the testnet

The Bitcoin node can be started in the testnet if you want to test the Bitcoin network and run some experiments. This is a faster network compared to the live network and has relaxed rules for mining and transactions.

The key differences between the mainnet and the testnet are shown here:

Component	Mainnet	Testnet
Listen port.	TCP 8333	TCP 18333
RPC connection port.	TCP 8332	TCP 18332

DNS Seeds are different for bootstrapping.	Mainnet-specific	Testnet-specific
Different ADDRESSVERSION field in addresses to ensure testnet addresses do not work on Bitcoin's mainnet.	0x00	0x6F
Genesis block.	Mainnet-specific	Testnet-specific
<code>IsStandard()</code> check to ensure a transaction is standard.	Enabled	Disabled

Various faucet services are also available for the Bitcoin test network. These services are used to get some test Bitcoin for testnet accounts. A list of faucets is available here at the Bitcoin wiki: <https://en.bitcoin.it/wiki/Testnet#Faucets>. The availability of test coins is very useful for experimentation on the testnet.

The command line to start up the Bitcoin testnet is as follows.

To run the Bitcoin daemon for the testnet:

```
$ bitcoind --testnet -daemon
```

To run the Bitcoin command line interface:

```
$ bitcoin-cli --testnet <command>
```

To allow the Bitcoin GUI to run in the testnet:

```
$ bitcoin-qt -testnet
```

A sample run is shown here:

1. Start up the Bitcoin node in daemon (as a background process) mode on the testnet:

```
$ bitcoind --testnet -daemon  
Bitcoin server starting
```

2. Check the number of blocks and difficulty. Note that there is a long list of various commands that the Bitcoin client supports. This is just an example to show how the Bitcoin command-line interface works:

```
$ bitcoin-cli --testnet getmininginfo  
{  
    "blocks": 566251,  
    "difficulty": 400.6820950060902,  
    "networkhashps": 572058533067.9225,  
    "pooledtx": 0,  
    "chain": "test",  
    "warnings": ""  
}
```

3. A complete list of commands can be obtained by running the following command:

```
$ bitcoin-cli --testnet help  
== Blockchain ==  
getbestblockhash  
getblock "blockhash" ( verbosity )  
getblockchaininfo  
getblockcount  
getblockhash height  
getblockheader "blockhash" ( verbose )  
. . .  
== Zmq ==  
getzmqnotifications
```



Note that only the initial few lines of the output are shown here; the rest of the output is not shown. The preceding output shows various command-line options available in bitcoin-cli, the Bitcoin command-line interface. These commands can be used to query the blockchain, send transactions, and control the local node.

4. We now can stop the Bitcoin daemon by using the command shown here:

```
$ bitcoin-cli --testnet stop  
Bitcoin server stopping
```

With this, we have completed a basic introduction to the Bitcoin testnet. We will do some more experimentation with this shortly, but first, we will look at another mode in which the Bitcoin node can run and which is especially useful for testing purposes.

Starting up a node in regtest

Regtest mode (regression testing mode) can be used to create a local private blockchain for testing purposes. In this mode, the user can control block generation for experimentation and testing, and a number of blocks with no value can be generated.

The following commands can be used to start up a node in regtest mode:

1. Start up the Bitcoin daemon in regtest mode:

```
$ bitcoind -regtest -daemon  
Bitcoin server starting
```

2. Check the balance:

```
$ bitcoin-cli -regtest getbalance  
0.00000000
```

3. Generate blocks and addresses:

```
$ bitcoin-cli -regtest generatetoaddress 200 $(bitcoin-cli -regtest getnewaddress)  
[  
  "366fce3c35031eaa3b085ae7d2631cb5b212bac7e3447bd8ffddb17e:  
  "7e5d8fe6969bb1c498389054fbddc783974562d73c247153909aa5b:  
  "5e17136ca3429aee54636e46e6b5a1b5f8c108b84465c73bb990fdcfe:  
  .  
  .  
  .  
  "33361a74d2586259d69a724921ff7b931cc6c95bd52f09fc05a4b89056  
]
```



The reason why we generated 200 blocks in the preceding command is because on a regtest, a block must have 100 confirmations before the associated reward can be utilized. Therefore, we must generate more than 100 blocks to get access to this reward. In this command, we have generated 200 blocks, which will generate 5,000 Bitcoin due to the miner reward of 50 Bitcoin.

4. Now, we can get the balance by running the following command:

```
$ bitcoin-cli -regtest getbalance  
5000.00000000
```

5. Run a command, for example, `getmininginfo`:

```
$ bitcoin-cli -regtest getmininginfo  
{  
    "blocks": 200,  
    "currentblockweight": 4000,  
    "currentblocktx": 0,  
    "difficulty": 4.656542373906925e-10,  
    "networkhashps": 12,  
    "pooleddtx": 0,  
    "chain": "regtest",  
    "warnings": ""  
}
```

6. We can also get information about the blockchain by using the following command:

```
$ bitcoin-cli -regtest getblockchaininfo  
{  
    "chain": "regtest",  
    "blocks": 200,  
    "headers": 200,  
    "bestblockhash": "1cafd1e540b6772f4fe4ab561def0de69945f84c",  
    "difficulty": 4.656542373906925e-10,  
    "mediantime": 1577225980,  
    .  
    .  
    .  
    "warnings": ""  
}
```



Note that the complete output is not shown here due to its long length, but it is enough to explain the concept.

7. Stop the Bitcoin daemon:

```
$ bitcoin-cli -regtest stop  
Bitcoin server stopping
```

If you want to delete the previous regtest node and start a new one, simply delete the directory named `regtest` under your computer's `$HOME` directory. On a Mac (macOS), it is located at `/$HOME/Library/Application Support/Bitcoin`.

After deleting the `regtest` directory, run the command shown in the first step again in this section to create a new `regtest` environment.

In this section, we covered how to start up a Bitcoin node in test and development (regtest) modes and how to interact with the Bitcoin blockchain using the `bitcoin-cli`, the command-line tool for the Bitcoin client. Next, we will experiment further with some Bitcoin commands and interfaces.

Experimenting further with `bitcoin-cli`

As we've seen so far, `bitcoin-cli` is a powerful and feature-rich command-line interface available with the Bitcoin Core client and can be used to perform various functions using the RPC interface provided by the Bitcoin Core client.

We will now see how to send Bitcoin to an address using the command line. For this, we will use the Bitcoin command-line interface on the Bitcoin regtest:

1. Generate a new address using the following command:

```
$ bitcoin-cli -regtest getnewaddress  
2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd
```

2. Send 20 BTC to the newly generated address:

```
$ bitcoin-cli -regtest sendtoaddress \ 2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd 20
```

The output of this command will show the transaction ID, which is:

```
a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5
```

3. We can now generate a few more blocks to get some confirmation for this:

```
$ bitcoin-cli -regtest generatetoaddress 7 $(bitcoin-cli -regtest getnewaddress)
```

4. We can also query the transaction information by using the following command:

```
$ bitcoin-cli -regtest gettransaction \ a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5
```

This will show an output similar to the one shown here. Note that we use the same transaction ID hash output that was generated in *step 2* previously:

```
{
  "amount": 0.00000000,
  "fee": -0.00003320,
  "confirmations": 7,
  "blockhash": "7c50e79b54dcda17e32cc7b7b53fc095584befef4e952422bdf096de3b93fe539",
  "blockindex": 1,
  "blocktime": 1577228072,
  "txid": "a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5f2980",
  "walletconflicts": [
  ],
  "time": 1577227733,
  "timereceived": 1577227733,
  "bip125-replaceable": "no",
  "details": [
    {
      "address": "2NC31WFFRwRkwd3S4TpjN5GGDY7E63GSVd",
      "category": "send",
      "amount": -20.00000000,
      "label": "",
      "vout": 0,
      "fee": -0.00003320,
      "abandoned": false
    },
    {
      "address": "2NC31WFFRwRkwd3S4TpjN5GGDY7E63GSVd",
      "category": "receive",
      "amount": 20.00000000,
      "label": "",
      "vout": 0
    }
  ],
  "hex": "020000000000101871203019a3456fc50b2044e79a4f078bc0ceb278734d44faf82ce4f2435770000000017a914cefefaf4c71513d952194695adefad119faf8f87870851d0b20000000017a914cc730f514cc654f222cecc09faf6a8e87d07c9a44a0220273478b3f452bec625d3d87002cb00831476gef37cf310609bce20e575c800000"
}
}
```

Figure 8.4: gettransaction output

We will now see an example of how to use the Bitcoin JSON-RPC interface. Note that we are using the Bitcoin mainnet for this example.

At a minimum, in order to use the JSON-RPC interface, we need to configure the RPC username and password in the `bitcoin.conf` file. We can easily do that. A sample configuration that will be used in this example is shown here:

```
$ cat bitcoin.conf
rpcuser=test1
rpcpassword=testpassword
```

Now, we will query the 100th block of the Bitcoin blockchain with hash 000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a .

So far, we've used `bitcoin-cli` on `regtest`. However, we can use `bitcoin-cli` on any Bitcoin network, for example, the testnet or the mainnet. We simply use the `bitcoin-cli` command without specifying any network to query the mainnet blockchain. Next, we will show a quick example of querying the

Bitcoin mainnet blockchain. The Bitcoin client provides three methods for interacting with the blockchain, as listed here:

- **Bitcoin command-line interface (CLI) – bitcoin-cli**
- **JSON-RPC Interface**
- **HTTP REST Interface**

First, we'll see an example of bitcoin-cli querying the blockchain using the `getblock` method. We will then see how the same `getblock` method can be invoked using the JSON-RPC interface and the HTTP REST interface.

Using the Bitcoin command-line tool – bitcoin-cli

We can use the bitcoin-cli for this purpose of using the Bitcoin command-line tool, as shown here:

```
$ bitcoin-cli getblock \ "000000007bc154e0fa7ea32218a72fe2c1bb9f
```

The output of the preceding command is shown here:

```
{
  "hash": "000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715",
  "confirmations": 232839,
  "strippedsize": 215,
  "size": 215,
  "weight": 860,
  "height": 100,
  "version": 1,
  "versionHex": "00000001",
  "merkleroot": "2d05f0c9c3e1c226e63b5fac240137687544cf631cd616f",
  "tx": [
    "2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc90"
  ],
  "time": 1231660825,
  "mediantime": 1231656204,
  "nonce": 1573057331,
```

As an alternative to bitcoin-cli, we can query the blockchain using the JSON-RPC interface provided by the Bitcoin client as well. We'll show an example of that next.

Using the JSON RPC interface

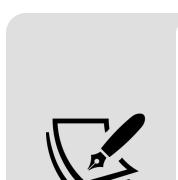
Now, we will run the same command but using the JSON-RPC. We can use the `curl` command-line tool to interact with the JSON-RPC API, as shown here:

This will ask for the required password. Enter the password that has been set in the `bitcoin.conf` file:

Enter host password for user 'test1':

If the password is correct, after executing the command, the result shown here will be displayed in JSON format:

{"result": {"hash": "000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8cc5"}},



`curl` is an excellent command-line tool that is used to transfer data using URLs. It is commonly used for interacting with REST APIs using HTTP.

More information about `curl` is available at
<https://curl.haxx.se>.

Starting from Bitcoin Core client 0.10.0, the HTTP REST interface is also available. By default, this runs on the same TCP port (`8332`) as the JSON-RPC interface and requires no authentication. It is enabled either in the `bitcoin.conf` file by adding the `rest=1` option or on the `bitcoind` command-line via the `-rest` flag.

An example run will be shown next, which queries the same block that we queried in the previous command, but now using the HTTP REST interface.

Using the HTTP REST interface

We can use `curl` again for this purpose, as shown here:

```
$ curl http://localhost:8332/rest/block/000000007bc154e0fa7ea322
```

The output of the preceding command is shown here:

```
{"hash": "000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ec"}
```

With this, we have completed our basic introduction to the Bitcoin command-line tools and related interfaces. A complete introduction to all Bitcoin client RPC calls is not possible here. You are encouraged to check the comprehensive documentation of Bitcoin Core 0.19.0 RPC, which is available at <https://bitcoincore.org/en/doc/0.19.0/rpc/>.

Bitcoin programming

Bitcoin programming is a very rich field. The Bitcoin Core client exposes various JSON-RPC commands that can be used to construct raw transactions and perform other functions via custom scripts or programs. Also, the command-line tool bitcoin-cli is available, which makes use of the JSON-RPC interface and provides a rich toolset to work with Bitcoin.

These APIs are also available via many online service providers in the form of Bitcoin APIs, and they provide a simple HTTP REST interface. Bitcoin APIs, such as blockchain.info (<https://blockchain.info/api>), BitPay (<https://bitpay.com/api>), block.io (<https://www.block.io>), and many others, offer a myriad of options to develop Bitcoin-based solutions.

Various libraries are available for Bitcoin programming. A list is shown as follows. Those of you who are interested can explore the libraries further:

- **Libbitcoin**: Available at <https://libbitcoin.dyne.org/> and provides powerful command-line utilities and clients.
- **Pycoin**: Available at <https://github.com/richardkiss/pycoin>, this is a library for Python.
- **Bitcoinj**: This library is available at <https://bitcoinj.github.io/> and is implemented in Java.

There are many online Bitcoin APIs available; the most commonly used APIs are listed as follows:

- <https://bitcore.io/>
- <https://bitcoinjs.org/>
- <https://blockchain.info/api>

As all APIs offer an almost similar type of functionality, it can get confusing to decide which one to use. It is also difficult to recommend which API is the best because all APIs are similarly feature-rich. One thing to keep in mind, however, is security. Therefore, whenever you evaluate an

API for usage, in addition to assessing the offered features, also evaluate how secure the design of the API is.

Summary

This chapter started with an introduction to Bitcoin installation, followed by some discussion on source code setup and how to set up Bitcoin clients for various networks. After this, we examined various command-line options available in Bitcoin clients. Lastly, we saw which APIs are available for Bitcoin programming and the main points to keep in mind while evaluating APIs for usage. In the next chapter, we will introduce alternative electronic cash blockchain projects, that is, the digital currency projects other than Bitcoin, such as Litecoin and Namecoin. We will also discuss various ideas that led to the development of alternative coin projects.

Alternative Coins

Since the initial success of Bitcoin, many alternative currency projects have been launched. Bitcoin was released in 2009, and the first alternative coin project (named Namecoin) was introduced in 2011. In 2013 and 2014, the **alternative coins (altcoin)** market grew exponentially, and many different types of alternative coin project were started.

A few of those became a success, whereas many were unpopular due to less interest and as a result, they did not succeed. A few were *pump and dump* scams that surfaced for some time but soon disappeared. Alternative approaches to Bitcoin can be divided broadly into two categories, based on the primary purpose of their development. If the primary goal is to build a decentralized blockchain platform, they are called alternative chains; if the sole purpose of the alternative project is to introduce a new virtual currency, it is called an altcoin.

Alternative blockchains are discussed in detail in this book's bonus online content pages, here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

This chapter is mainly dedicated to the foundations and features of altcoins, whose primary purpose is to introduce a new virtual currency (coin), although some material will also be presented on the topic of alternative protocols built on top of Bitcoin to provide various services. These include concepts such as Namecoin, covered in this book's bonus content repository, the primary purpose of which is to provide decentralized naming and identity services instead of currency.

Introducing altcoins

Currently, as of mid-2020, there are thousands of altcoins on the market, and they hold some monetary value, such as Namecoin, Zcash, Litecoin, and many others. Zcash is a more successful altcoin, introduced in 2016. On the other hand, Primecoin did not gain much popularity, but it is still in use. Many of these alternative projects are direct forks of Bitcoin source code, although some have been written from scratch. Some altcoins set out to address Bitcoin limitations such as privacy. Some others offer different types of mining, changes in block times, and distribution schemes.

By definition, an altcoin is generated in the case of a hard fork. If Bitcoin has a harder fork, then the other, older chain is effectively considered another coin. However, there is no established rule as to which chain becomes the altcoin. This has happened with Ethereum, where a hard fork caused a new currency, **Ethereum Classic (ETC)**, to come into existence in addition to the **Ethereum (ETH)** currency.

Ethereum classic is the old chain and Ethereum is the new chain after the fork. Such a contentious hard fork is not desirable for a few reasons. First, it is against the true spirit of decentralization as the Ethereum foundation, a central entity, decided to go ahead with the hard fork, even though not everyone agreed to the proposition; second, it also splits the user community due to disagreement over the hard fork. Although a hard fork, in theory, generates an altcoin, it is limited in what it can offer because, even if the change results in a hard fork, usually, there are no drastic changes around the fundamental parameters of the coin. They typically remain the same. For this reason, it is desirable to either write a new coin from scratch or fork the Bitcoin (or another coin's source code) to create a new currency with the desired parameters and features.

Altcoins must be able to attract new users, trades, and miners; otherwise, the currency will have no value. Currency gains its value, especially in the virtual currency space, due to the network effect and its acceptability by the community. If a coin fails to attract enough users, then soon, it will be forgotten. Users can be attracted by providing an initial amount of coins and can be achieved by using various methods. There is, however, a risk that if

the new coin does not perform well, then their initial investment may be lost. Methods of providing an initial number of altcoins are as follows:

1. **Create a new blockchain:** Altcoins can create a new blockchain and allocate coins to initial miners, but this approach is now unpopular due to many scam schemes, or *pump and dump* schemes, where initial miners made a profit with the launch of a new currency and then disappeared.
2. **Proof of Burn (PoB):** Another approach to allocating initial funds to a new altcoin is PoB, also called a one-way peg or price ceiling. In this method, users permanently destroy a certain quantity of bitcoins in proportion to the number of altcoins to be claimed. For example, if 10 bitcoin were destroyed, then altcoins can have a value no greater than the Bitcoin that was destroyed. This means that bitcoins are being converted into altcoin by burning them.
3. **Proof of ownership:** Instead of permanently destroying bitcoins, an alternative method is to prove that users own a certain number of bitcoins. This proof of ownership can be used to claim altcoins by tethering altcoin blocks to Bitcoin blocks. For example, this can be achieved by merged mining in which, effectively, Bitcoin miners can mine altcoin blocks while mining for bitcoins without any extra work. Merged mining will be explained later in this chapter.
4. **Pegged sidechains:** Sidechains, as the name suggests, are blockchains separate from the Bitcoin network, but Bitcoin can be transferred to them. Altcoins can also be transferred back to the Bitcoin network. This concept is called a **two-way peg**.

Investing and trading these alternative coins is also big business, albeit not as big as Bitcoin but enough to attract new investors and traders and provide liquidity to the market. Alternative coin market capitalization is shown as follows:

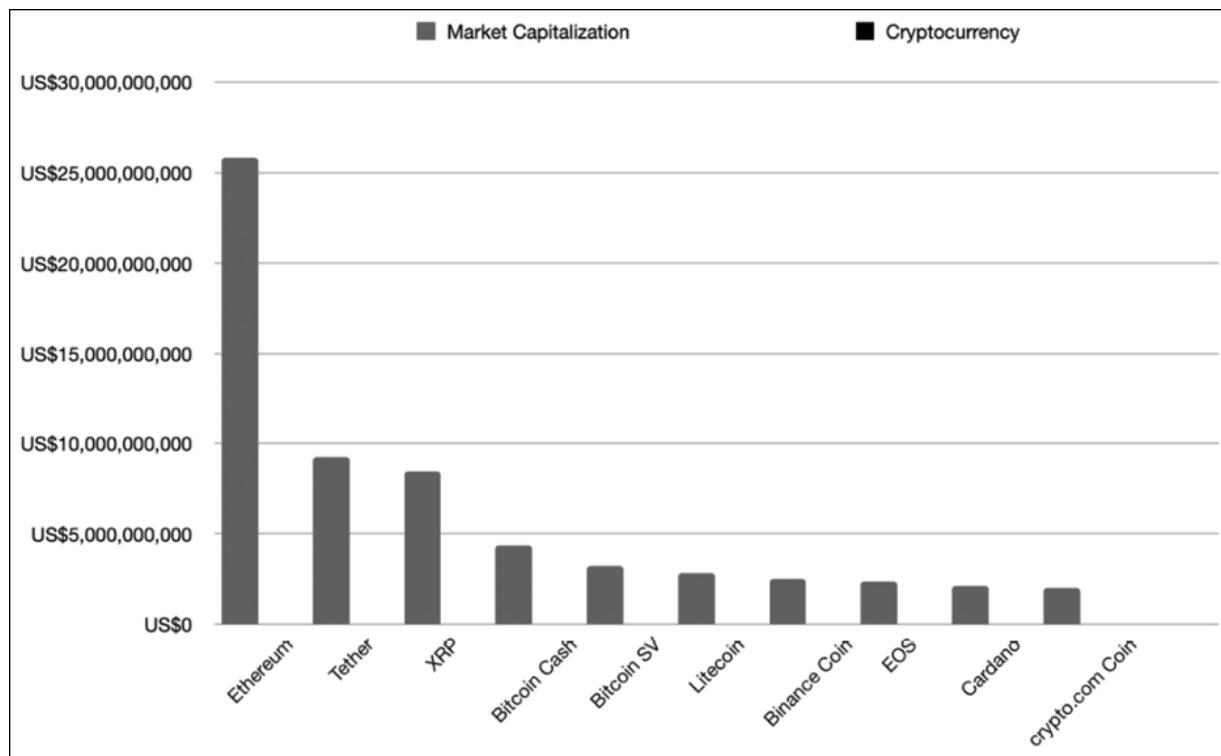


Figure 9.1: Top 10 cryptocurrencies, excluding Bitcoin, which is the topmost

The current market cap (as of June, 2020), from <https://coinmarketcap.com/>) of the top 10 coins is shown as follows:

Cryptocurrency	Market cap in US\$	Price in US\$
Bitcoin	\$173,083,267,448	\$9,402.42
Ethereum	\$25,844,303,523	\$231.98
Tether	\$9,217,432,271	\$1.00
XRP	\$8,460,691,410	\$0.191168
Bitcoin Cash	\$4,372,112,481	\$237.11

Bitcoin SV	\$3,232,271,412	\$175.31
Litecoin	\$2,831,861,371	\$43.58
Binance Coin	\$2,521,729,503	\$16.21
EOS	\$2,368,331,983	\$2.54
Cardano	\$2,127,729,709	\$0.08

This table shows that the Combined Altcoin Market Capitalization is roughly around USD 234 billion.

The following pie chart shows the market cap of the top three cryptocurrencies as of late 2019. This shows that Bitcoin has the largest market capitalization.

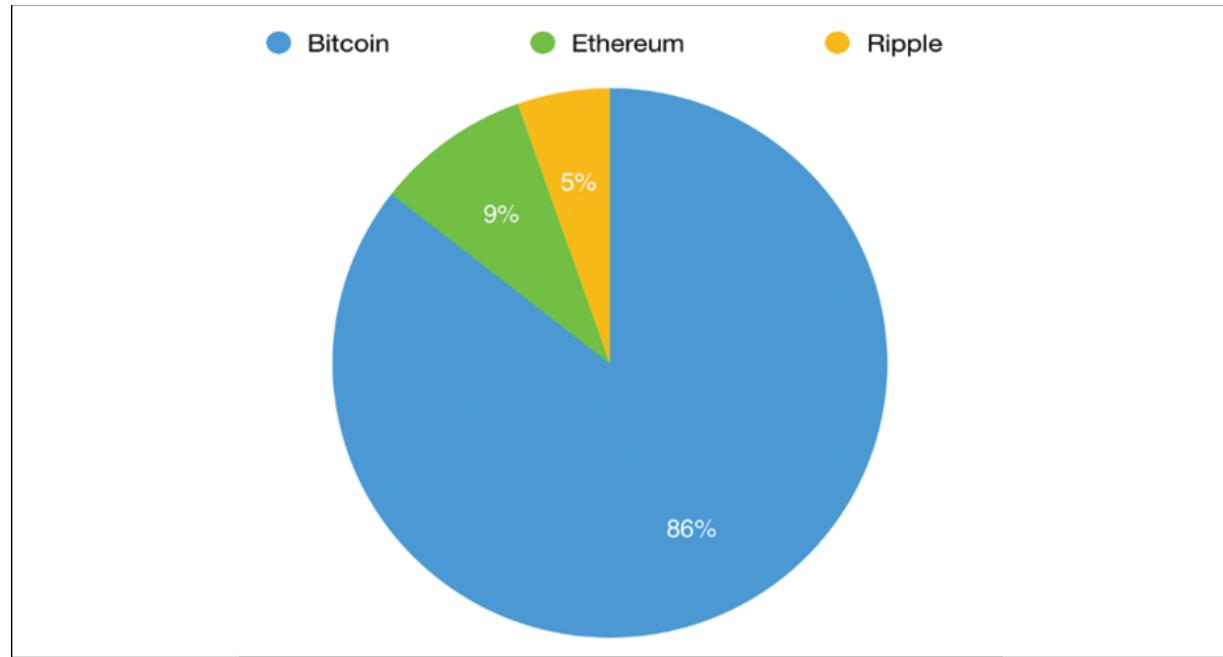


Figure 9.2: Top three cryptocurrencies

Note that market capitalization varies with time and up-to-date data can be obtained from

<https://coinmarketcap.com/charts/#dominance-percentage> or similar websites on the internet.

There are various factors and new concepts that have been introduced with alternative coins. Many concepts were invented even before Bitcoin, but with Bitcoin, not only new concepts, such as a solution to the double spending problem, were introduced, but also previous ideas such as **hashcash** and **Proof of Work (PoW)** were used ingeniously and came into the limelight.

Since then, with the introduction of alternative coin projects, various new techniques and concepts have been developed and introduced. To appreciate the current landscape of alternative cryptocurrencies, it is essential to understand some theoretical concepts first.

Theoretical foundations

In this section, various theoretical concepts will be introduced that have been developed with the introduction of different altcoins in the past few years.

Alternatives to Proof of Work

The PoW scheme in the context of cryptocurrency was first used in Bitcoin and served as a mechanism to provide assurance that a miner had completed the required amount of work to find a block. This process, in turn, provided decentralization, security, and stability for the blockchain. This is the primary vehicle in Bitcoin for providing decentralized distributed consensus. PoW schemes are required to have a much-desired property called **progress freeness**, which means that the reward for consuming computational resources should be random and proportional to the contribution made by the miners. In this case, some chance of winning the block reward is given to even those miners who have comparatively less computational power.

The term **progress freeness** was introduced by Arvind Narayanan *et al.* in the book *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction* (Princeton University Press, 2016). Other requirements for mining computational puzzles include **adjustable difficulty** and **quick verification**. Adjustable difficulty ensures that the difficulty target for mining on the blockchain is regulated accordingly in response to increased hashing power and the number of users.

Quick verification is a property that means that mining computational puzzles should be easy and quick to verify. Another aspect of the PoW scheme, especially the one used in Bitcoin (Double SHA-256), is that since the introduction of ASICs, the power is shifting toward miners or mining pools who can afford to operate large-scale ASIC farms. This power shift challenges the core philosophy of the decentralization of Bitcoin.

There are a few alternatives that have been proposed such as ASIC-resistant puzzles, which are designed in such a way that building ASICs for solving this puzzle is infeasible and does not result in a major performance gain over commodity hardware. A common technique used for this purpose is to apply a class of computationally hard problems called **memory hard computational puzzles**. The core idea behind this method is that as puzzle solving requires a large amount of memory, it is not feasible to be implemented on ASIC-based systems.

This technique was initially used in Litecoin and Tenebrix, where the Scrypt hash function was used as an ASIC-resistant PoW scheme. Even though this scheme was initially advertised as ASIC resistant, Scrypt ASICs became available a few years ago, disproving the original claim by Litecoin. This happened because even if Scrypt is a memory-intensive mechanism, initially, it was thought that building ASICs with large memories is difficult due to technical and cost limitations. This is no longer the case because memory hardware is increasingly becoming cheaper. Also, with the ability to produce nanometer-scale circuits, it is possible to build ASICs that can run the Scrypt algorithm.

Another approach to ASIC resistance is where multiple hash functions are required to be calculated to provide PoW. This is also called a **chained hashing scheme**. The rationale behind this idea is that designing multiple

hash functions on an ASIC is not very feasible. The most common example is the X11 memory hard function implemented in Dash. X11 comprises 11 SHA-3 contestants where one algorithm outputs the calculated hash to the next algorithm until all 11 algorithms are used in a sequence. These algorithms include BLAKE, BMW, Groestl, JH, Keccak, Skein, Luffa, CubeHash, SHAvite, SIMD, and ECHO.

This approach did provide some resistance to ASIC development initially, but now, ASIC miners are available commercially and support mining of X11 and similar schemes. A recent example is ASIC Baikal Miner, which supports X11, X13, X14, and X15 mining. Other examples include miners such as the iBeLink DM384M X11 miner and the PinIdea X11 ASIC miner.

Perhaps another approach could be to design self-mutating puzzles that intelligently or randomly change the PoW scheme or its requirements as a function of time. This strategy will make it almost impossible to be implemented in ASICs as it will require multiple ASICs to be designed for each function. Also, randomly changing schemes would be practically impossible to handle in ASICs. At the moment, it is unclear how this can be achieved practically.

PoW does have various drawbacks, and the biggest of all is energy consumption. It is estimated that the total electricity consumed by Bitcoin miners currently is more than that of Greece at **59.61 Terawatt hash (TWh)**. This is huge, and all that power is, in a way, wasted; in fact, no useful purpose is served except mining. Environmentalists have raised real concerns about this situation. In addition to electricity consumption, the carbon footprint is also very high, with it currently being estimated at around 253 kg of CO₂ per transaction.

The following graph shows the scale of Bitcoin energy consumption compared to other countries. This consumption is only expected to grow, and it is estimated that by the end of 2020, the energy consumption will reach approximately 125 TWh per year:

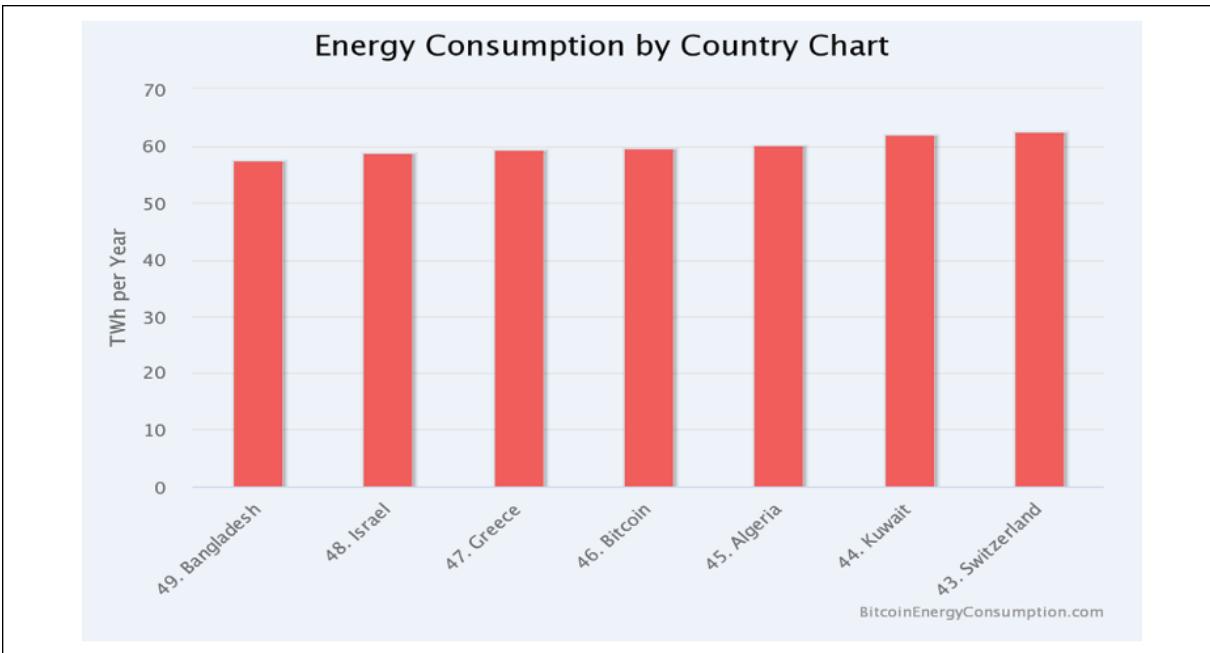


Figure 9.3: Energy consumption by country

The preceding graph has taken from the website that tracks this subject. It is available at <https://digiconomist.net/bitcoin-energy-consumption>.

It has been proposed that PoW puzzles can be designed in such a way that they serve two purposes. First, their primary purpose is in consensus mechanisms, and second, they serve to perform some useful scientific computation. This way, not only can the schemes be used in mining, but they can also help to solve other scientific problems. This proof of useful work has been recently put into practice by Primecoin, where the requirement is to find special prime number chains known as Cunningham chains and bi-twin chains. As the study of prime number distribution has special significance in scientific disciplines such as physics, mining Primecoin miners not only achieves the block reward but also helps in finding the special prime numbers.

Proof of Storage

Also known as **proof of retrievability**, this is another type of proof of useful work that requires storage of a large amount of data. Introduced by Microsoft Research, this scheme provides a useful benefit of distributing the storage of archival data. Miners are required to store a pseudo, randomly-selected subset of large data to perform mining.

Proof of Stake (PoS)

This proof is also called **virtual mining**. This is another type of mining puzzle that has been proposed as an alternative to traditional PoW schemes. It was first proposed in Peercoin in August 2012. In this scheme, the idea is that users are required to demonstrate the possession of a certain amount of currency (coins), thus proving that they have a stake in the coin.

The simplest form of this stake is where mining is made comparatively easier for those users who demonstrably own larger amounts of digital currency. The benefits of this scheme are twofold; first, acquiring large amounts of digital currency is relatively difficult as compared to buying high-end ASIC devices, and second, it results in saving computational resources. Various forms of stake have been proposed and are briefly discussed in the following subsection.

Various types of stake

Different type of stakes will now be introduced in the following subsections.

Proof of coinage

The age of a coin is the time since the coins were last used or held. This is a different approach from the usual form of PoS, where mining is made easier for users who have the highest stake in the altcoin. In the coin-age-based approach, the age of the coin (coinage) is reset every time a block is mined. The miner is rewarded for holding and not spending coins for a period of time. This mechanism has been implemented in Peercoin combined with PoW in a creative way.

The difficulty of mining puzzles (PoW) is inversely proportional to the coinage, meaning that if miners consume some coinage using coin-stake transactions, then the PoW requirements are relieved.

Proof of Deposit (PoD)

This is a type of PoS. The core idea behind this scheme is that newly minted blocks by miners are made unspendable for a certain period. More precisely, the coins get locked for a set number of blocks during the mining operation. The scheme works by allowing miners to perform mining at the cost of freezing a certain number of coins for some time.

Proof of Burn (PoB)

As an alternate expenditure to computing power, **PoB**, in fact, destroys a certain number of Bitcoins to get equivalent altcoins. This is commonly used when starting up a new coin projects as a means to provide a fair initial distribution. This can be considered an alternative mining scheme where the value of the new coins comes from the fact that, previously, a certain number of coins have been destroyed.

Proof of Activity (PoA)

This scheme is a hybrid of PoW and PoS. In this scheme, blocks are initially produced using PoW, but then each block randomly assigns three stakeholders that are required to digitally sign it. The validity of subsequent blocks is dependent on the successful signing of previously randomly chosen blocks.

There is, however, a possible issue known of the **nothing at stake** problem, where it would be trivial to create a fork of the blockchain. This is possible because in PoW, appropriate computational resources are required to mine, whereas in PoS, there is no such requirement; as a result, an attacker can try to mine on multiple chains using the same coin.

Non-outsourceable puzzles

The key motivation behind this puzzle is to develop resistance against the development of mining pools. Mining pools, as previously discussed, offer rewards to all participants in proportion to the computing power they consume. However, in this model, the mining pool operator is a central authority to whom all the rewards go and who can enforce specific rules. Also, in this model, all miners only trust each other because they are working toward a common goal, in the hope of the pool manager getting the reward. Non-outsourceable puzzles are a scheme that allows miners to claim rewards for themselves; consequently, pool formation becomes unlikely due to inherent mistrust between anonymous miners.

There are also various other alternatives to PoW, some of which have been described in *Chapter 1, Blockchain 101*. Some will be explained later in this book in *Chapter 17, Hyperledger*, and *Chapter 21, Scalability and Other Challenges*. As this is an ongoing area of research, new alternatives will keep emerging as blockchain technology grows.

In this section, we covered alternatives to **PoW** schemes. These schemes largely fall under the umbrella of **PoS** mechanisms. We know that PoW schemes are based on the difficulty of performing some work to prove that the required amount of resources has been spent to win the right to announce a new block. A question arises here regarding how this difficulty is calculated in a blockchain, as well as how the balance between difficulty and coin production is maintained. Also, in order to maintain a constant supply (production) of coins, the difficulty of the network also needs to be adjusted overtime. For this purpose, PoW algorithms use difficulty adjustment and retargeting algorithms, which are simple arithmetic formulas that help adjust the difficulty of solving the PoW problem. We'll introduce the core ideas of these calculations in the next section.

Difficulty adjustment and retargeting algorithms

The concept of difficulty retargeting algorithms has been introduced with the advent of Bitcoin and associated consensus mechanisms. In Bitcoin, a difficulty target is calculated simply by the following equation; other coins have either developed their own algorithms or implemented a modified version of the Bitcoin difficulty algorithm:

$$T = \text{Time previous} * \text{time actual} / 2016 * 10 \text{ min}$$

The core idea behind difficulty regulation in Bitcoin is that a generation of 2016 blocks should take roughly around 2 weeks (inter-block time should be around 10 minutes). If it takes longer than 2 weeks to mine 2016 blocks, then the difficulty is decreased, and if it takes less than 2 weeks to mine 2016 blocks, then the difficulty is increased. When ASICs were introduced due to a high block generation rate, the difficulty increased exponentially, and that is one drawback of PoW algorithms that are not ASIC resistant. This leads to mining power centralization.

This also poses another problem; if a new coin starts now with the same PoW based on SHA-256 as Bitcoin uses, then it would be easy for a malicious adversary to use an ASIC miner and control the entire network. This attack would be more practical if there is less interest in the new altcoin, and someone decides to take over the network by consuming adequately high computing resources. This attack may not be possible if other miners with comparable computing power also join the altcoin network because then, miners will be competing with each other.

Also, multipools pose a more substantial threat, which is where a group of miners can automatically switch to the currency that is becoming profitable. This phenomenon is known as pool hopping and can adversely affect a blockchain and, consequently, the growth of the altcoin. Pool hopping impacts the network negatively because pool hoppers join the network only when the difficulty is low so that they can gain quick rewards. If the

moment difficulty goes up (or is readjusted), they hop off and then come back again when the difficulty is adjusted back.

For example, if a multipool consumes its resources when mining a new coin too quick, the difficulty will increase very quickly; when the multipool leaves the currency network, it becomes almost unusable because now, the difficulty has risen to such a level that it is no longer profitable for solo miners and can no longer be maintained. The only fix for this problem is to initiate a hard fork, which is usually undesirable for the community.

There are a few algorithms that have come into existence to address this issue and are discussed next in this chapter. All these algorithms are based on the idea of readjusting various parameters in response to hash rate changes; these parameters include the number of previous blocks, the difficulty of previous blocks, the ratio of adjustment, and the number by which the difficulty can be readjusted back or up.

In the following section, you will be introduced to the few difficulty algorithms being used in and proposed for various altcoins.

Kimoto Gravity Well

This algorithm is used in various altcoins to regulate difficulty. This method was first introduced in Megacoin and used to adjust the difficulty of the network every block adaptively. The logic of the algorithm is shown as follows:

$$KGW = 1 + (0.7084 * \text{pow}((\text{double}(\text{PastBlocksMass})/\text{double}(144)), -1.228))$$

The algorithm runs in a loop that goes through a set of predetermined blocks (*PastBlocksMass*) and calculates a new readjustment value. The core idea behind this algorithm is to develop an adaptive difficulty regulation mechanism that can readjust the difficulty in response to rapid spikes in hash rates. **Kimoto Gravity Well (KGW)** ensures that the time between blocks remains approximately the same. In Bitcoin, the difficulty is adjusted every 2016 blocks, but in KGW, the difficulty is adjusted at every block.

This algorithm is vulnerable to **time warp attacks**, which allow an attacker to enjoy less difficulty in creating new blocks temporarily. This attack allows a time window where the difficulty becomes low and the attacker can quickly generate many coins at a fast rate.



More information can be found at
<https://cryptofrenzy.wordpress.com/2014/02/09/multi-pools-vs-gravity-well/>.

Dark Gravity Wave

Dark Gravity Wave (DGW) is a new algorithm designed to address certain flaws such as the time warp attack in the KGW algorithm. This concept was first introduced in Dash, previously known as Darkcoin. It makes use of multiple exponential moving averages and simple moving averages to achieve a smoother readjustment mechanism. The formula is as follows:

$$2222222 / (((Difficulty + 2600) / 9)^2)$$

This formula is implemented in Dashcoin, Bitcoin SegWit2X, and various other altcoins as a mechanism to readjust difficulty.

DGW version 3.0 is the latest implementation of the DGW algorithm and allows improved difficulty retargeting compared to KGW.



More information can be found at
<https://dashpay.atlassian.net/wiki/spaces/DOC/pages/1146926/Dark+Gravity+Wave>.

DigiShield

This is another difficulty retargeting algorithm that has recently been used in Zcash with slight variations and after adequate experimentation. This algorithm works by going through a fixed number of previous blocks to calculate the time they took to be generated, and then readjusts the difficulty to the difficulty of the previous block by dividing the actual time span by averaging the target time. In this scheme, the retargeting is calculated much more rapidly, and the recovery from a sudden increase or decrease in hash rate is quick. This algorithm protects against multi-pools, which can result in rapid hash rate increases.

The network difficulty is readjusted every block or every minute, depending on the implementation. The key innovation is that it has faster readjusting times compared to KGW.

Zcash uses DigiShield v3.0, which uses the following formula for difficulty adjustment:

$$(New\ difficulty) = (previous\ difficulty) \times \text{SQRT} [(150\ seconds) / (last\ solve\ time)]$$



There is a detailed discussion regarding this topic that is available at the following link:
<https://github.com/zcash/zcash/issues/147#issuecomment-245140908>

MIDAS

Multi-Interval Difficulty Adjustment System (MIDAS) is an algorithm that is comparatively more complex than the algorithms discussed previously due to the number of parameters it uses. This method responds much more rapidly to abrupt changes in hash rates. This algorithm also protects against time warp attacks.



The original post about this is now available via web archive at
<https://web.archive.org/web/20161005171345/http://d>



illingers.com/blog/2015/04/21/altcoin-difficulty-adjustment-with-midas/.

This concludes our introduction to various difficulty adjustment algorithms.

Many alternative cryptocurrencies and protocols have emerged as an attempt to address various limitations in Bitcoin. We will introduce some of these now.

Bitcoin limitations

Various limitations in Bitcoin have also sparked some interest in altcoins. Some alternative coins were explicitly developed to address shortcomings in Bitcoin. The most prominent and widely discussed limitation is the lack of anonymity in Bitcoin. We will now discuss some of the limitations of Bitcoin.

Privacy and anonymity

As the blockchain is a public ledger of all transactions and is openly available, it becomes easy to analyze it. When combined with traffic analyses, transactions can be linked back to their source IP addresses, thus possibly revealing a transaction's originator. This is a big concern from a privacy point of view.

In the Bitcoin domain, it is a recommended and common practice to generate a new address for every transaction, which allows some level of unlinkability. However, this is not enough, and various techniques were developed and successfully used to trace the flow of transactions throughout the network and link them back to their originator. These techniques analyze blockchains by using transaction graphs, address graphs, and entity graphs, which facilitate linking users back to the transactions, thus raising privacy concerns.

The techniques mentioned earlier can be further enriched by using publicly available information (for example, public internet forum users' Bitcoin addresses) about transactions and linking them to the actual users. There are open source block parsers available that can be used to extract transaction information, balances, and scripts from the blockchain database.

Various proposals have been made to address the privacy issue in Bitcoin. These proposals fall into three categories: mixing protocols, third-party mixing networks, and inherent anonymity.

A brief discussion of each category is presented as follows.

Mixing protocols

These schemes are used to provide anonymity to Bitcoin transactions. In this model, a mixing service provider (an intermediary or a shared wallet) is used. Users send coins to this shared wallet as a deposit, and then the shared wallet can send some other coins (of the same value deposited by some other users) to the destination. Users can also receive coins that were sent by others via this intermediary. This way, the link between outputs and inputs is no longer there, and transaction graph analysis will not be able to reveal the actual relationship between senders and receivers:

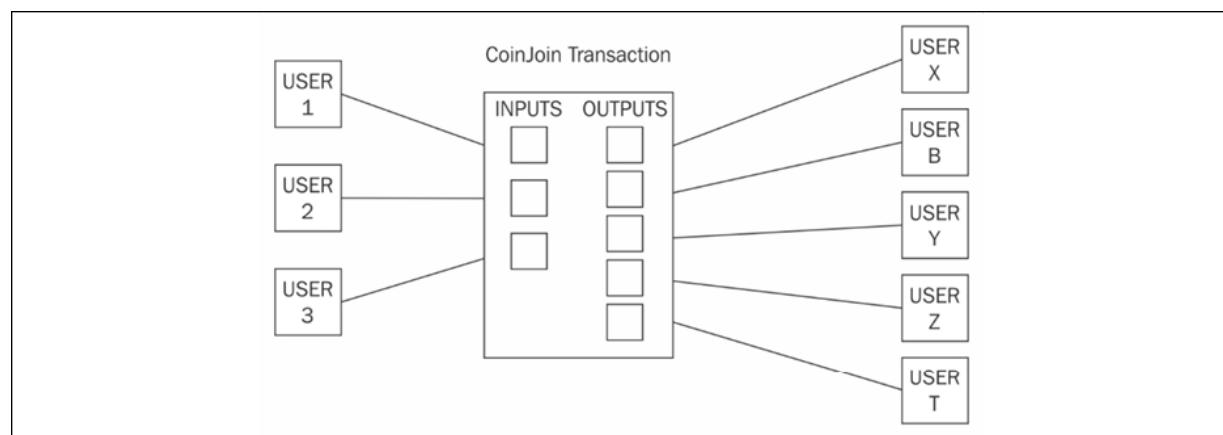


Figure 9.4: CoinJoin transaction with three users joining their transaction into a single larger CoinJoin transaction

CoinJoin is one example of mixing protocols, where two transactions are joined together to form a single transaction while keeping the inputs and outputs unchanged. The core idea behind CoinJoin is to build a shared transaction that is signed by all participants. This technique improves privacy for all participants involved in the transactions.

Third-party mixing protocols

Various third-party mixing services are available, but if the service is centralized, then it poses the threat of tracing the mapping between the senders and receivers. This is because the mixing service knows about all inputs and outputs. In addition to this, fully centralized miners pose the risk of the administrators of the service stealing the coins.

Various services, with varying degrees of complexity, such as CoinShuffle, Coinmux, and Darksend in Dash (coin), are available that are based on the idea of CoinJoin (mixing) transactions. CoinShuffle is a decentralized alternative to traditional mixing services as it does not require a trusted third party.

CoinJoin-based schemes, however, have some weaknesses, most prominently the possibility of launching a denial of service attack by users who committed to signing the transactions initially but now are not providing their signature, thus delaying or stopping joint transactions altogether.

Inherent anonymity

This category includes coins that support privacy inherently and is built into the design of the currency. The most popular is Zcash, which uses **zero-knowledge proofs (ZKPs)** to achieve anonymity. We discussed ZKPs in *Chapter 4, Public Key Cryptography*, and will discuss them in more detail in the section on Zcash in this chapter. Other examples include Monero (<https://web.getmonero.org>), which makes use of ring signatures to provide anonymity services.

Other schemes that have been proposed for Bitcoin privacy include:

- CoinSwap
- TumbleBit
- Dandelion

We will discuss each of these next.

CoinSwap

CoinSwap is a privacy mechanism that is based on the idea of **atomic swaps**. Atomic swaps allow two parties to exchange coins without requiring a trusted third party. CoinSwap can also be used for cross-chain swaps. CoinSwap works by utilizing a third party in the transaction flow and also requires private communication channels between all parties. This way, the addresses of the sender and the receiver cannot be linked. This third party receives funds from the sender and sends them to the receiver. The technique here is that the third party pays funds to the receiver by using a totally different source of funds, thus disconnecting the link between the sender and the receiver. This disconnection between the sender and the receiver results in providing an unlinkable transaction and thus privacy. The sender uses multi-signature transactions (usually 2 of 2—sender and third party) to allow transactions to be spent, while the receiver also requires multi-signature transactions (usually 2 of 2—receiver and third party) for transactions to be spent. In other words, the sender and the third party will sign the transaction output to send the Bitcoin to the third party, while the receiver and the third party will sign the transaction output to send the Bitcoin to the receiver. CoinSwap uses hash locked transactions where a pre-image of the hash is required to unlock the transaction. Using hash locked transactions prevents the third party from stealing the Bitcoin.



More information on CoinSwap can be found at the following link, where it was originally proposed by Gregory Maxwell:

<https://bitcointalk.org/index.php?topic=321228>.

TumbleBit

The TumbleBit protocol was introduced in 2016. TumbleBit is fully compatible with the Bitcoin protocol. It is an anonymous, fast, and off-chain payments (unlinkability) protocol that allows parties to transfer funds via an untrusted third party or intermediary called a **tumbler**. In this protocol, even the tumbler is unable to deanonymize the payers and payees involved in a payment. It involves using two fair exchange protocols that prevent any malicious activity, such as cheating participants or the tumbler. TumbleBit relies on a protocol called **RSA puzzle solver**, which allows a payer to make payments to the tumbler. Unless tumbler solves this RSA puzzle, it cannot claim any bitcoin paid by the payer. Another fair exchange protocol, called the **puzzle-promise** protocol, is used between the tumbler and the payee to claim the payment.

TumbleBit consists of three phases, as listed here:

1. Escrow phase, where all payment channels are set up
2. Payments phase, where payers transfer funds
3. Cash-out phase, where payers and payees close the payment channels



More information on TumbleBit can be found here:
<https://eprint.iacr.org/2016/575.pdf>.

A proof of concept implementation of TumbleBit is available here:
<https://github.com/BUSEC/TumbleBit>.

A recent innovation is the introduction of TumbleBit++, which is an improved mixing protocol based on TumbleBit. It provides anonymity and confidentiality of transaction amounts by combining confidential transactions and a centralized untrusted anonymous payment hub.



More information regarding TumbleBit++ is available here:
https://link.springer.com/chapter/10.1007/978-3-030-31919-9_21.

We will now discuss **Dandelion**, which provides inherent anonymity and is implemented by redesigning the network layer of the Bitcoin protocol.

Dandelion

In addition to the approaches mentioned previously, a recent proposal called **Dandelion** has also been made. Dandelion (the improved version is called Dandelion++) is a proposal that aims to make transactions on a Bitcoin network untraceable. This protocol will allow anonymous transactions to occur on the Bitcoin network as opposed to pseudonymous transactions, where an adversary, by using network analysis methods, can trace the transaction back to its source node. Consequently, the adversary can then discover the original IP address of the transaction sender.



Deanonymization of Bitcoin users is a known problem, and a number of research papers are available on this topic. For example, the paper *Deanonymization of clients in Bitcoin P2P network* is available at <https://arxiv.org/pdf/1405.7418.pdf>.

We can say that Dandelion is a mechanism to provide inherent anonymity to the Bitcoin transactions because, once implemented, the P2P layer of the network Bitcoin protocol is modified in such a way that tracing transactions back to their source node and IP would become extremely difficult. A Bitcoin improvement proposal is available at <https://github.com/bitcoin/bips/blob/master/bip-0156.mediawiki>.

The original Dandelion proposal paper is available at <https://arxiv.org/pdf/1701.04439.pdf>. The Dandelion++ research paper is available at <https://arxiv.org/pdf/1805.11060.pdf>.

In the Bitcoin network, the **epidemic flooding** mechanism (Gossip protocol) is used for transaction propagation. On top of this mechanism, there is a somewhat effective method called **diffusion**, which is used to provide some level of anonymity. In this protocol, each node introduces independent and exponential delays for spreading transactions to its

neighbors. This scheme results in reducing the symmetry of the epidemic protocol, which makes network analysis difficult and unreliable. However, this scheme is predictable and hence does not provide sufficient anonymity guarantees.

Dandelion proposes a new but backward compatible routing mechanism. In this protocol:

- First, a privacy graph (anonymity set) is constructed. This phase is called the **private graph construction** phase. It is a sub-graph of the existing Bitcoin P2P network and each node selects a subset of its outbound peers in this phase. This is where the **random line of nodes** is selected.
- The messages (transactions) are then routed through this privacy graph during the **Stem** phase. This is where the message is propagated on a random line of nodes.
- Finally, there is the **Fluff** phase, where the messages are routed (broadcast) to the entire network by diffusion.

In summary, the dandelion protocol is composed of an **anonymity** phase and a **spreading** phase. In the anonymity phase, this protocol spreads a message over a random line for a number of random hops. After this step, the message is broadcast over the whole network using diffusion. With this combination of random path selection and diffusion, the **Dandelion** protocol provides near-optimal anonymity guarantees.

This protocol can be visualized using the following diagram:

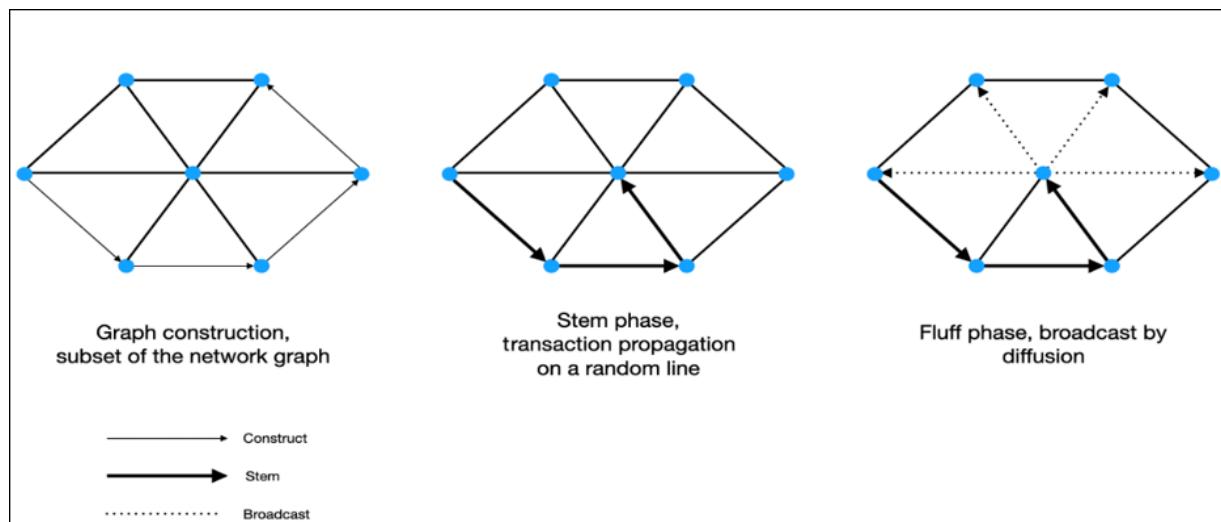


Figure 9.5: Dandelion protocol routing

In the preceding diagram, first, graph construction is performed, which is shown on the left-hand side. Then, in the middle, we have the transaction propagation along a random line of nodes. Finally, the broadcast, on the right-hand side, is shown.



More information on Dandelion is available in the following paper:
<https://arxiv.org/pdf/1701.04439.pdf>.

Now that we have discussed various privacy techniques, let's move on to the next section, which introduces various other enhancements that have been made or have been proposed to extend the Bitcoin protocol.

Extended protocols on top of Bitcoin

Several protocols, as discussed in the following sections, have been proposed and implemented on top of Bitcoin to enhance and extend the

Bitcoin protocol, as well as to be used for various other purposes instead of just as a virtual currency.

Colored coins

Colored coins are a set of methods that have been developed to represent digital assets on the Bitcoin blockchain. Coloring a bitcoin refers colloquially to updating it with some metadata representing a digital asset (smart property). The coin still works and operates as a Bitcoin, but additionally carries some metadata that represents some assets. This can be some information related to the asset, some calculations related to transactions, or any arbitrary data. This mechanism allows issuing and tracking specific bitcoins. Metadata can be recorded using the Bitcoin's `OP_RETURN` opcode or optionally in multi-signature addresses. The metadata can also be encrypted if required to address any privacy concerns. Some implementations also support the storage of metadata on publicly available torrent networks, which means that virtually unlimited amounts of metadata can be stored. Usually, these are JSON objects representing various attributes of the colored coin. Moreover, smart contracts are also supported.

Colored coins can be used to represent a multitude of assets, including, but not limited to, commodities, certificates, shares, bonds, and voting. It should also be noted that to work with colored coins, a wallet that interprets colored coins is necessary and that normal Bitcoin wallets will not work. Normal Bitcoin wallets will not work because they cannot differentiate between **colored coins** and **not colored coins**.

The idea of colored coins is very appealing as it does not require any modifications to be made to the existing Bitcoin protocol, and they can also make use of the already existing secure Bitcoin network. In addition to the traditional representation of digital assets, there is also the possibility of creating smart assets that behave according to the parameters and conditions defined for them. These parameters include time validation, restrictions on transferability, and fees. This opens up the possibility of creating smart contracts, which we will discuss in *Chapter 10, Smart Contracts*.

A significant use case can be the issuance of financial instruments on the blockchain. This will ensure low transaction fees, valid and mathematically secure proof of ownership, fast transferability without requiring some intermediary, and instant dividend payouts to investors.



There were a few services available online that used to provide colored coins services, such as *Colu* by *Coinprism* (<https://en.bitcoin.it/wiki/Coinprism>), but they are no longer active.

Counterparty

This is another service that can be used to create custom tokens that act as a cryptocurrency and can be used for various purposes, such as issuing digital assets on top of the Bitcoin blockchain. This is quite a robust platform and runs on Bitcoin blockchains at their core, but has developed its client and other components so that they support issuing digital assets. The architecture consists of the following components:

1. **Counterparty server:** This is the reference client and implements the core counterparty protocol. It is a combination of `counterparty-lib` and `counterparty-cli`.
2. **Counter block:** This component provides services in addition to the Counterparty server.
3. **Counter wallet:** This is a web wallet for Bitcoin and **Counterparty coin (XCP)**.
4. **armory_utxsvr:** This is a service used for offline armory transactions.

Counterparty works based on the same idea as colored coins by embedding data into regular Bitcoin transactions, but provides a much more productive library and a set of powerful tools to support the handling of digital assets. This embedding is also called **embedded consensus** because the counterparty transactions are embedded within Bitcoin transactions. The method of embedding the data is by using `OP_RETURN` opcode in Bitcoin.

The currency produced and used by Counterparty is known as XCP and is used by smart contracts as the fee for running the contract. At the time of writing, its price is 1.03 USD. XCPs were created by using the PoB method discussed previously.

Counterparty allows the development of smart contracts on Ethereum using the Solidity language and allows interaction with the Bitcoin blockchain. To achieve this, BTC Relay is used as a means to provide interoperability between Ethereum and Bitcoin. This is a clever concept where Ethereum contracts can talk to the Bitcoin blockchain and transactions through BTC Relay. The relayers (the nodes that are running BTC Relay) fetch the Bitcoin block headers and relay them to a smart contract on the Ethereum network that verifies the PoW. This process verifies that a transaction has occurred on the Bitcoin network.



Technically, this is an Ethereum contract that is capable of storing and verifying Bitcoin block headers, just like Bitcoin simple payment verification lightweight clients do by using bloom filters. SPV clients were discussed in detail in the previous chapter. This idea can be visualized with the following diagram:

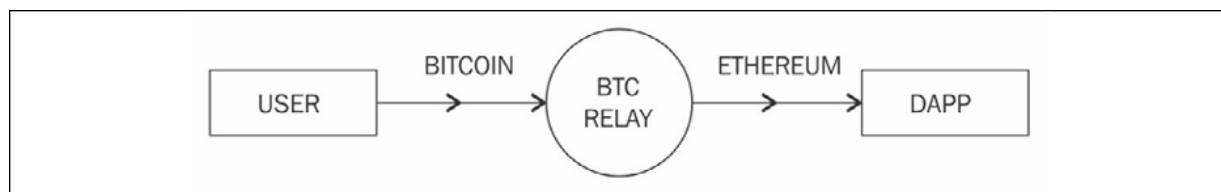


Figure 9.6: BTC relay concept



Now, we will move on to a different topic, which explains how altcoins are developed, how they work, and how difficult it is to create a new coin.

Development of altcoins

Altcoin projects can be started very quickly from a coding point of view by simply forking the Bitcoin or another coin's source code, but this probably is not enough. When a new coin project is started, several things need to be considered to ensure a successful launch and the coin's longevity. Usually, the code base is written in C++, as was the case with Bitcoin, but almost any language can be used to develop coin projects; for example, Golang or Rust.

Writing code or forking the code for an existing coin is the easy part. The challenging issue is how to start a new currency so that investors and users can be attracted to it.

From a technical point of view, in the case of forking the code of another coin, for example, Bitcoin, there are various parameters that can be changed to effectively create a new coin. These parameters are required to be tweaked or introduced in order to create a new coin. These parameters can include, but are not limited to, the following.

Consensus algorithms

There is a choice of consensus algorithms available, for example, PoW, which is used in Bitcoin, or PoS, which is used in Peercoin. There are also other algorithms available, such as **Proof of Capacity (PoC)** and a few others, but PoW and PoS are the most common choices.

Hashing algorithms

This is either SHA-256, Scrypt, X11, X13, X15, or any other hashing algorithm that is adequate for use as a consensus algorithm.

Difficulty adjustment algorithms

Various options are available in this category to provide difficulty retargeting mechanisms. The most prominent examples are KGW, DGW, Nite's Gravity Wave, and DigiShield. Also, all these algorithms can be tweaked based on requirements to produce different results; therefore, many variants are possible.

Inter-block time

This is the time that has elapsed between the generation of each block. For Bitcoin, the blocks are generated every 10 minutes, while for Litecoin, it's 2.5 minutes. Any value can be used, but an appropriate value is usually around a few minutes. If the generation time is too fast, it might destabilize the blockchain, while if it's too slow, it may not attract many users.

Block rewards

A block reward is for the miner who solves the mining puzzle and is allowed to have a coinbase transaction that contains the reward. This used to be 50 coins in Bitcoin initially and now many altcoins set this parameter to a very high number; for example, in Dogecoin, currently, it is 10,000. However, changing the block reward is optional and depends on the choice of the coin creator.

Reward halving rate

This is another important factor; in Bitcoin, it is halved every 4 years and now is set to 12.5 Bitcoin. It's a variable number that can be set to any time period or none at all, depending on the requirements.

Block size and transaction size

This is another important factor that determines how high or low the transaction rate can be on the network. Block sizes in Bitcoin are limited to 1 MB, but in altcoins, it can vary depending on the requirements.

Interest rate

This property applies only to PoS systems where the owner of the coins can earn interest at a rate defined by the network, in return for some coins that are held on the network as a stake to protect the network. This interest rate keeps inflation under control. If the interest rate is too low, then it can cause hyperinflation.

Coinage

This parameter defines how long the coin has to remain unspent in order for it to become eligible to be considered stake worthy.

Total supply of coins

This number sets the total limit of the coins that can ever be generated. For example, in Bitcoin, the limit is 21 million, whereas in Dogecoin, it's unlimited. This limit is fixed by the block reward and halving schedule discussed earlier.

There are two options to create your own virtual currency: forking existing established cryptocurrency source code or writing a new one from scratch. The latter option is less popular, but the first option is easier and has allowed the creation of many virtual currencies. Fundamentally, the idea is that first a cryptocurrency source code is forked, and then appropriate changes are made at different strategic locations in the source code to effectively create a new currency.



In the earlier days of the evolution of the crypto landscape, any coin project other than Bitcoin was called **alternative coin**, or **altcoin** for short. Over the years, the term **cryptocurrency** or just **crypto** is now generally used to refer to either Bitcoin or any other digital currency. Cryptocurrency is defined as a currency that runs on a digital platform and makes use of cryptography to secure its operations.

It is not possible to cover all alternative coins (cryptocurrencies) in this chapter, but a few selected coins will be discussed in this book's bonus content pages, here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf. The coins discussed in the resource pack have been selected based on longevity, market cap, and innovation. Each coin will be discussed from different perspectives, such as theoretical foundations, mining, and trading.



There are thousands of cryptocurrencies (as of June 2020) in the market. A list can be found at <https://coinmarketcap.com/all/views/all/>.

So far, we've discussed a number of the features of alternative coins, and you should now be familiar with the foundations of some different Altcoin protocols and algorithms. Before we move on to a different but relevant topic of **tokens** on blockchain, let's clarify a common ambiguity regarding the use of the terms **token** and **cryptocurrency**.

Token versus cryptocurrency

Sometimes, the terms **token** and **cryptocurrency** are used interchangeably, which is not correct. Tokens are a different concept from cryptocurrency. A cryptocurrency is a native coin for a standalone blockchain. In contrast, a token is a representation of the value of some asset. It is usually built on top of an existing blockchain. For example, ether is the native cryptocurrency or coin of the Ethereum blockchain.

Similarly, Bitcoin is the original (inherent) coin or cryptocurrency of the Bitcoin blockchain. On the other hand, a token is created on top of an existing blockchain and does not have its own native and dedicated blockchain. For example, Binance token, 0x, Tether, and **Basic Attention Token (BAT)** exist on top of the Ethereum blockchain and do not have their native blockchain. There are thousands of tokens that exist on Ethereum and other blockchains.



A list of all tokens on the Ethereum blockchain can be found here:

<https://etherscan.io/tokens>.

In this chapter, we touched briefly on the vast subject of tokenization. We will discuss tokenization and different tokens, such as Tether USD and Maker and related concepts, in greater detail in *Chapter 18, Tokenization*.

Certainly, launching new alternative coins or tokens requires some funding. Traditionally, new businesses are funded by methods like **Initial public offering (IPOs)**, **venture capital funds (VCs)**, and so on, but in the cryptocurrency world, an equivalent but different mechanism called **Initial Coin Offering (ICO)** is used. We will explain what an ICO is now.

Initial Coin Offerings (ICOs)

ICO is a method for crowdfunding. Crowdfunding is a method of raising money where the funds are acquired usually in smaller amounts from a large number of investors to fund a new business. This is done usually via the internet. ICOs has been the method of choice for crowdfunding startups offering new cryptocurrencies and tokens.

ICOs are comparable to the IPO. Just as an IPO is launched to raise capital by a firm, similarly, ICOs are launched to generate money for a startup project. The critical difference is that IPOs are regulated and fall under the umbrella of the securities market (shares in the company), whereas ICOs

are unregulated and do not fall under any strict category of already established market structures.

However, there are a few suggestions stating that ICOs should be treated as securities in light of some scam ICO schemes that were launched in the past, as well as growing concerns around investor protection. The **Securities and Exchange Commission (SEC)** has suggested that all coins, ICOs, and digital assets should fall under the definition of **security**. This means that the same laws would be applicable to ICOs, Bitcoin, and other digital coins as are applicable to securities. Also, an introduction of formal **Know Your Customer (KYC)** and **Anti-Money Laundering (AML)** is also being recommended to address issues related to money laundering. Experts are recommending the **Howey Test** as part of the criteria for any ICO to be considered a **security**.



More information on the Howey Test can be found at
<https://www.investopedia.com/terms/h/howey-test.asp>.

Another difference is that ICOs, by design, usually require investors to invest using cryptocurrencies and payouts are paid using cryptocurrencies. Most commonly, this is the new token (a new cryptocurrency) introduced by the ICO. This can also be Fiat currency, but most commonly, cryptocurrency is used. For example, in the Ethereum crowdfunding campaign, a new token, ether was introduced. The use of the name token sale instead of crowdfunding is also quite popular and both terms are sometimes used interchangeably. ICOs are also called crowd sales.

When a new blockchain-based application or organization is launched, a new token can be launched with it as a token to access and use the application, and also to gain incentives that are paid in the very same token that was introduced by the ICO. This token is released to the public in exchange for some already established cryptocurrency (for example, Bitcoin or Ethereum) or Fiat currency. The advantage is that when the usage of the application or product launched increases, the value of the new token

also increases with it. This way, the investors who invested initially gain a good incentive.

Since 2017, ICOs have become a leading tool for raising capital for new startups. The first successful ICO was that of Ethereum, which raised 18 million USD in 2014. A recent success is Tezos, which made 232 million USD in a few weeks. Another example is Filecoin, which raised more than 250 million USD. A more recent example is EOS, which raised a record of over 4 billion USD.

The process of creating a new token has been standardized on Ethereum blockchain, thus making it relatively easy to launch an ICO and issue new tokens in exchange for ether, Bitcoin, or some other cryptocurrency. This standard is called ERC20 and is described in the next section. It's worth noting that using ERC20 is not a requirement, and a completely new cryptocurrency can be invented on a new blockchain to start an ICO, but ERC20 has been used in many ICOs and provides an easier and quicker way to build a token for an ICO.

ERC20 standard

The ERC20 standard is an interface that defines various functions dictating the requirements of the token. It does not, however, provide implementation details and has been left to the implementer to decide. ERC is an abbreviation for **Ethereum Request for Comments**, which is equivalent to Bitcoin's BIPs for suggesting improvements in the Ethereum blockchain.



This is defined under EIP 20, which you can read more about here:

<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md>

Ethereum has become a platform of choice for ICOs due to its ability to create new tokens and with the ERC20 standard, it has become even more accessible.

The ERC20 token standard defines various functions that describe various properties, rules, and attributes of the new token. These include the total supply of the coins, the total balance of holders, transfer functions, and approval and allowance functions.



There are other standards making improvements on ERC20, such as ERC223, ERC777, and ERC827 that are also under development. You can refer to the followings links to learn more:

<https://github.com/ethereum/EIPs/issues/827>

<https://github.com/ethereum/EIPs/issues/223>

<https://github.com/ethereum/EIPs/issues/777>

We will cover tokenization and relevant concepts such as **Security Token Offerings (STOs)** and **Initial Exchange Offerings (IEOs)**, which are alternatives to ICOs, in more detail in *Chapter 18, Tokenization*.

Summary

In this chapter, we introduced you to the overall cryptocurrency landscape. We discussed several altcoins in detail, especially Zcash and Namecoin. Cryptocurrencies are a very active area for research. Topics such as scalability, privacy, and security are the main focus of many researchers. Further research needs to be carried out in the areas of the privacy and scalability of blockchain. Some research has also been conducted to invent new difficulty retargeting algorithms to thwart the threat of centralization in cryptocurrencies. Altcoins or cryptocurrencies in general are a fascinating field of research, and they open many possibilities for a decentralized future.

We also discussed a few practical aspects, such as mining and starting a new currency project, which will give you a strong foundation, enabling

you to explore these areas further. Now, you should be able to appreciate the subject of altcoins and understand the various motivations behind them.

In the next chapter, we will see what smart contracts are and discuss relevant ideas and concepts that are essential to understanding the blockchain technology fully.

10

Smart Contracts

This chapter introduces smart contracts. This concept is not new; however, with the advent of blockchain technology, interest in this idea has revived. Smart contracts are now an ongoing and intense area of research in the blockchain space. Many blockchains have emerged that support smart contracts.

Due to benefits such as the increased security, cost-saving, and transparency that smart contracts can bring to many industries (especially the finance industry), rigorous research is in progress at various commercial and academic institutions to make the implementation of smart contracts easier, more practical, business-friendly, and more secure as soon as possible.

In this chapter, we will explore the ideas that led to the development of smart contracts, including its relevant history and key definitions. We will also cover concepts like the deployment of smart contracts, notable security aspects of smart contracts, and, finally, oracles. First, we will look at the history of smart contracts.

History

Nick Szabo first theorized smart contracts in the 1990s, in an article called *Formalizing and Securing Relationships on Public Networks*. This theory was presented almost 20 years before the real potential and benefits of smart contracts were appreciated, that is, before the invention of Bitcoin and the subsequent development of other more advanced blockchain platforms, such as Ethereum.

Smart contracts are described by Szabo as follows:

"A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs."



The original article that was written by Szabo is available at <http://firstmonday.org/ojs/index.php/fm/article/view/548>.

Smart contract functionality was implemented in a limited fashion in Bitcoin in 2009. Bitcoin supports a restricted scripting language called **script**, which allows the transfer of bitcoins between users. However, this is not a Turing complete language and does not support arbitrary program development. It can be regarded as a limited function calculator with only simple arithmetic operations, whereas smart contracts can be considered general-purpose computers that support writing any program.

Definition

There are many definitions of smart contracts. An online search reveals many definitions. However, while the definitions provided online at various

sources are correct and useful, in my opinion, they are not complete. It is crucial to properly and completely define what a smart contract is. The following is my attempt to provide a comprehensive generalized definition of a smart contract:

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

Dissecting this definition reveals that a smart contract is, fundamentally, a computer program that is written in a language that a computer or target machine can understand. Also, it encompasses agreements between parties in the form of business logic. Another fundamental idea is that smart contracts are automatically executed according to the instruction that is coded in, for example, when certain conditions satisfy. They are enforceable, which means that all contractual terms perform as specified and expected, even in the presence of adversaries.

Enforcement is a broader term that encompasses traditional enforcement in the form of a law, along with the implementation of specific measures and controls that make it possible to execute contract terms without requiring any intervention.

Preferably, smart contracts should not rely on any traditional methods of enforcement. Instead, they should work on the principle that *code is the law*, which means that there is no need for an arbitrator or a third party to enforce, control, or influence the execution of a smart contract. Smart contracts are self-enforcing as opposed to legally enforceable. This idea may sound like a libertarian's dream, but it is entirely possible and is in line with the true spirit of smart contracts.

Moreover, they are secure and unstoppable, which means that these computer programs are fault-tolerant and executable in a reasonable (finite) amount of time. These programs should be able to execute and maintain a healthy internal state, even if external factors are unfavorable. For example, imagine a typical computer program that is encoded with some logic and executes according to the instruction coded within it. However, if the environment it is running in or the external factors it relies on deviate from

the usual or expected state, the program may react arbitrarily or abort. Smart contracts must be immune to this type of issue.



Blockchain platforms play a vital role in providing the necessary underlying network with security guarantees required to run the smart contracts.

In some scenarios, security and unstoppability may well be considered optional features. Still, it will provide more significant benefits in the long run if security and unstoppable properties are included in the smart contract definition. This inclusion will allow researchers to focus on these aspects from the start and will help to build strong foundations on which further research can then be based. There is also a suggestion by some researchers that smart contracts do not need to be automatically executable; instead, they can be what's called automatable, due to the manual human input required in some scenarios. For example, the manual verification of a medical record might be needed by a qualified medical professional. In such cases, fully automated approaches may not work best. While it is true that, in some instances, human input and control are helpful, they are not necessary. For a contract to be truly smart, in my opinion, it has to be fully automated. Certain inputs that need to be provided by people can and should also be automated. Oracles can be used for this purpose. We will discuss oracles in more detail later on in this chapter.

Smart contracts usually operate by managing their internal state using a state machine model provided by the underlying blockchain. This allows the development of a practical framework for programming smart contracts, where the state of a smart contract is advanced further based on some predefined criteria and conditions.

There is also an ongoing debate on whether computer code is acceptable as a conventional contract in a court of law. A smart contract is different in presentation from traditional legal prose, albeit they do represent and enforce all required contractual clauses. Still, a court of law does not understand computer code. This dilemma raises several questions about how a smart contract can be legally binding: can it be developed in such a way that it is readily acceptable and understandable in a court of law? Is it

possible for dispute resolution be implemented within the code? Moreover, regulatory and compliance requirements are other topics that require addressing before smart contracts can become as efficient as traditional legal documents.

The legality of smart contracts is uncertain in many jurisdictions. Still, a recent exciting development in this space made crypto assets and smart contracts valid in English law by recognizing crypto assets as tradeable property and smart contracts as enforceable agreements. This announcement was made by the **UK Jurisdiction Taskforce (UKJT)** of the Lawtech Delivery Panel. More information about this, including the full legal statement, is available here:

<https://technation.io/news/uk-takes-significant-step-in-legal-certainty-for-smart-contracts-and-cryptocurrencies/>.

Even though smart contracts are named smart, they only do what they have been programmed to do. This very property of smart contracts ensures that smart contracts produce the same output every time they are executed. The deterministic nature of smart contracts is highly desirable in blockchain platforms due to consistency requirements.

Now, this gives rise to a problem whereby a large gap between the real world and the blockchain world emerges. In this situation, natural language is not understood by the smart contract, and, similarly, computer code is not acceptable for the natural world. So, a few questions arise: how can real-life contracts be deployed on a blockchain? How can this bridge between the real world and the smart contract world be built?

These questions open up various possibilities, such as making smart contract code readable not only by machines but also by people. If humans and machines can both understand the code written in a smart contract, it might become acceptable in legal situations, as opposed to just a piece of code that no one understands except for programmers. This desirable property is an area that is ripe for research, and a significant research effort has been expended in this area to answer questions around the semantics, meaning, and interpretation of smart contracts.

Some work has already been done to describe natural language contracts formally, by combining both smart contract code and natural language contracts through linking contract terms with machine-understandable elements. This is achieved using a markup language called the **Legal Knowledge Interchange Format (LKIF)**, which is an XML schema for representing theories and proofs. It was developed under the Estrella project in 2008.



More information is available in the research paper at
https://doi.org/10.1007/978-3-642-15402-7_30.

Further details can be found here:
http://www.estrellaproject.org/?page_id=5.

Another requirement regarding the properties of smart contracts is that they must be deterministic. This property will allow a smart contract to be run by any node on a network and achieve the same result. If the result differs even slightly between nodes, then a consensus cannot be reached, and a whole paradigm of distributed consensuses on the blockchain can fail. Moreover, it is also desirable that the contract language itself is deterministic, thus ensuring the integrity and stability of the smart contracts. A deterministic property means that for the same input, there is always the same output. In other words, the system must not manifest different behavior for the same input in different runs. In a blockchain network, this would mean that smart contracts developed using the smart contract programming language do not produce different results on different nodes.

Let's take, for example, various floating-point operations calculated by various functions in a variety of programming languages that can produce different results in different runtime environments. Another example is some math functions in JavaScript, which can produce different results for the same input on different browsers and can, in turn, lead to various bugs. This scenario is unacceptable in smart contracts because if the results are inconsistent between the nodes, then a consensus will never be achieved.

The deterministic feature ensures that smart contracts always produce the same output for a specific input. In other words, programs, when executed,

produce a reliable and accurate business logic that is entirely in line with the requirements programmed in the high-level code.

In summary, a smart contract has the following properties:

- **Automatically executable**: It is self-executable on a blockchain without requiring any intervention.
- **Enforceable**: This means that all contract conditions are enforced automatically.
- **Secure**: This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programming language and the smart contract code themselves must be correct, valid, and verified.
- **Deterministic**: The deterministic feature ensures that smart contracts always produce the same output for a specific input. Even though it can be considered to be part of the secure property, defining it here separately ensures that the deterministic property is considered one of the important properties.
- **Semantically sound**: This means that they are complete and meaningful to both people and computers.
- **Unstoppable**: This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

It could be argued that the first four properties are required as a minimum, whereas the latter two may not be necessary or applicable in some scenarios and can be relaxed. For example, a financial derivatives contract does not, perhaps, need to be semantically sound and unstoppable but should at least be automatically executable, enforceable, deterministic, and secure. On the other hand, a title deed needs to be semantically sound and complete; therefore, for it to be implemented as a smart contract, the language that it is written in must be understood by both computers and people.

Ian Grigg addressed this issue of interpretation with his invention of Ricardian contracts, which we will introduce in the next section.

Ricardian contracts

Ricardian contracts were initially proposed in the paper, *Financial Cryptography in 7 Layers*, by Ian Grigg, in the late 1990s.

This paper is available at <https://iang.org/papers/fc7.html>.

Ricardian contracts were initially used in a bond trading and payment system called **Ricardo**. The fundamental idea behind this contract is to write a document that is understood and accepted by both a court of law and computer software. Ricardian contracts address the challenge of the issuance of value over the internet. A Ricardian contract identifies the issuer and captures all the terms and clauses of the contract in a document to make it acceptable as a legally binding contract.

A Ricardian contract is a document that has several of the following properties:

- It is a contract offered by an issuer to holders
- It is a valuable right held by holders and managed by the issuer
- It can be easily read by people (like a contract on paper)
- It can be read by programs (parsable, like a database)
- It is digitally signed
- It carries the keys and server information
- It is allied with a unique and secure identifier



The preceding information is based on the original definition by Ian Grigg at http://iang.org/papers/ricardian_contract.html.

In practice, the contracts are implemented by producing a single document that contains the terms of the contract in legal prose and the required machine-readable tags. This document is digitally signed by the issuer using their private key. This document is then hashed using a message digest function to produce a hash by which the document can be identified. This hash is then further used and signed by parties during the performance of the contract to link each transaction with the identifier hash, therefore serving as evidence of intent. This is depicted in the next diagram and is usually called a bowtie model.

The diagram shows a number of elements:

- The **World of Law** is on the left-hand side from where the document originates. This document is a written contract in legal prose with some machine-readable tags.
- This document is then hashed.
- The resultant message digest is used as an identifier throughout the **World of Accountancy**, as shown on the right-hand side of the diagram.

The **World of Accountancy** element represents any accounting, trading, and information systems that are being used in the business to perform various business operations. The idea behind this flow is that the message digest generated by hashing the document is first used in a so-called **genesis transaction**, or first transaction, and then it is used in every transaction as an identifier throughout the operational execution of the contract. This way, a secure link is created between the original written contract and every transaction in the **World of Accounting**:

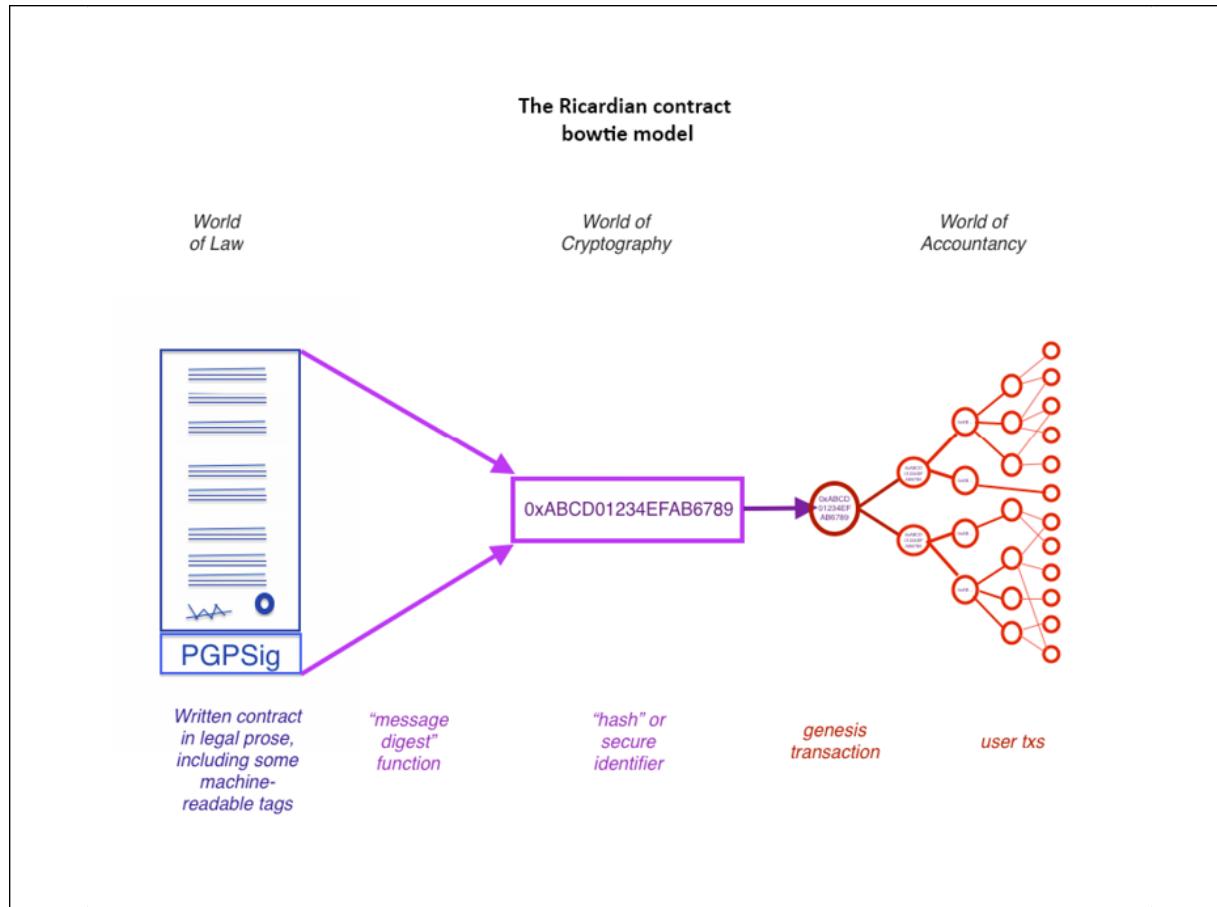


Figure 10.1: The Ricardian Contract bowtie diagram

A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a contract can be divided into two types: operational semantics and denotational semantics.

The first type defines the actual execution, correctness, and safety of the contract, and the latter is concerned with the real-world meaning of the full contract. Some researchers have differentiated between smart contract code and smart legal contracts, where a smart contract is only concerned with the execution of the contract. The second type encompasses both the denotational and operational semantics of a legal agreement. It perhaps makes sense to categorize smart contracts based on the difference between

the semantics, but it is better to consider a smart contract as a standalone entity that is capable of encoding legal prose and code (business logic).

In Bitcoin, a straightforward implementation of basic smart contracts (conditional logic) can be observed, which is entirely oriented toward the execution and performance of the contract, whereas a Ricardian contract is more geared toward producing a document that is understood by humans with some parts that a computer program can understand. This can be viewed as legal semantics versus operational performance (semantics versus performance), as shown in the following diagram. The diagram shows that Ricardian contracts are more semantically-rich, whereas smart contracts are more performance-rich. This concept was initially proposed by *Ian Grigg* in his paper, *On the intersection of Ricardian and Smart Contracts*.



The paper is available at
https://iang.org/papers/intersection_ricardian_smart.html.

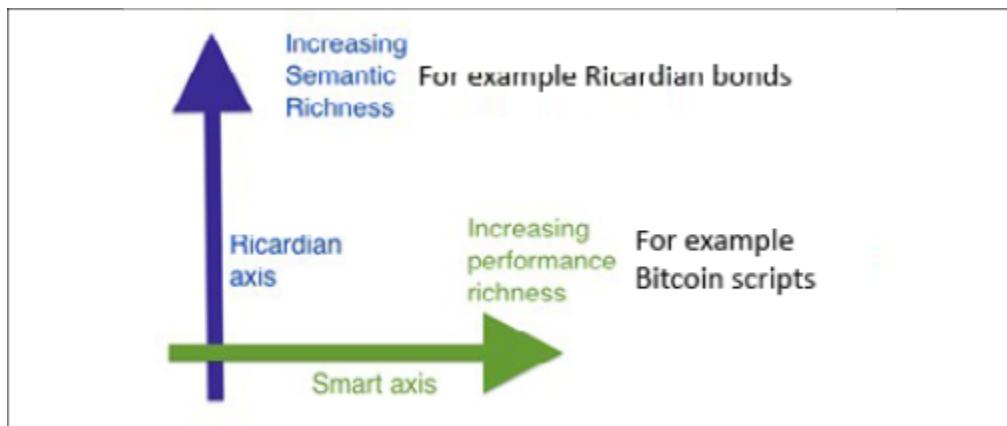


Figure 10.2: Diagram explaining that performance versus semantics is an orthogonal issue, as described by Ian Grigg; it is slightly modified to show examples of different types of contracts on both axes

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

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

Ricardian contracts have been implemented in many systems, such as CommonAccord (<http://www.commonaccord.org>) and OpenBazaar (<https://openbazaar.org>).

Now that we understand what Ricardian contracts are, let's take a look at the concept of **smart contract templates**. These have been built on the idea of Ricardian contracts, and they aim to support the management of the entire life cycle of legal smart contracts.

Smart contract templates

Smart contracts can be implemented in any industry where they are required, but the most popular use cases relate to the financial sector. This is because blockchain first found many use cases in the finance industry and, therefore, sparked enormous research interest in the financial industry long before other areas. Recent work in the smart contract space specific to the financial sector has proposed the idea of smart contract templates. The idea is to build standard templates that provide a framework to support legal agreements for financial instruments.

Christopher D. Clack et al. proposed this idea in their paper published in 2016, named *Smart Contract Templates: Foundations, design landscape and research directions*.



The paper is available at
<https://arxiv.org/pdf/1608.00771.pdf>.

The paper also suggested that **domain-specific languages (DSLs)** should be built to support the design and implementation of smart contract templates. A language named **common language for augmented contract knowledge (CLACK)** has been proposed, and research has started to develop this language. This language is intended to be very rich and is expected to provide a large variety of functions ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

Clack et al. also carried out work to develop smart contract templates that support legally-enforceable smart contracts. This proposal has been discussed in their research paper, *Smart Contract Templates: essential requirements and design options*.

This paper is available at

<https://arxiv.org/pdf/1612.04496.pdf>.

The main aim of this paper is to investigate how legal prose could be linked with code using a markup language. It also covers how smart legal agreements can be created, formatted, executed, and serialized for storage and transmission. This work is ongoing and remains an open area for further research and development.

Contracts in the finance industry are not a new concept, and various DSLs are already in use in the financial services industry to provide a specific language for a particular domain. For example, there are DSLs available that support the development of insurance products, represent energy derivatives, or are being used to build trading strategies.



A comprehensive list of financial DSLs can be found at
<http://www.dslfin.org/resources.html>.

It is also essential to understand the concept of DSLs, as this type of programming language can be developed to program smart contracts. DSLs are different from **general-purpose programming languages (GPLs)**. DSLs have limited expressiveness for a particular application or area of

interest. These languages possess a small set of features that are sufficient and optimized for a specific domain only. Unlike GPLs, they are not suitable for building large general-purpose application programs.

Based on the design philosophy of DSLs, it can be envisaged that such languages will be developed specifically to write smart contracts. Some work has already been done, and **Solidity** is one such language that has been introduced with the Ethereum blockchain to write smart contracts. Vyper is another language that has been recently introduced for Ethereum smart contract development.

This idea of DSLs for smart contract programming can be further extended to a **GPL**. A smart contract modeling platform can be developed where a domain expert (not a programmer but a front desk dealer, for example) can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract. Once the flow is drawn and completed, it can be emulated first to test it and then be deployed from the same system to the target platform, which can be a smart contract on a blockchain or even a complete **decentralized application (DApp)**. This concept is also not new, and a similar approach is already used in a non-blockchain domain, in the Tibco StreamBase product, which is a Java-based system used for building event-driven, high-frequency trading systems.

It has been proposed that research should also be conducted in the area of developing high-level DSLs that can be used to program a smart contract in a user-friendly graphical user interface, thus allowing a non-programmer domain expert (for example, a lawyer) to design smart contracts.

Apart from DSLs, there is also a growing interest in using general-purpose, already established programming languages like Java, Go, and C++ to be used for smart contract programming. This idea is appealing, especially from a usability point of view, where a programmer who is already familiar with, for example, Java, can use their skills to write Java code instead of learning a new language. The high-level language code can then be compiled into a low-level bytecode for execution on the target platform. There are already some examples of such systems, such as in EOSIO

blockchains, where C++ can be used to write smart contracts, which are compiled down to the web assembly for execution.

An inherent limitation with smart contracts is that they are unable to access any external data. The concept of oracles was introduced to address this issue. An oracle is an off-chain source of information that provides the required information to the smart contracts on the blockchain.

Some earlier discussions and thoughts about oracles can be found here:



[https://bitcointalk.org/index.php?
topic=8821.msg133523#msg133523](https://bitcointalk.org/index.php?topic=8821.msg133523#msg133523)

<https://github.com/orisi/wiki/wiki/Orisi-White-Paper>

<http://gavintech.blogspot.com/2014/06/bit-thereum.html>

Now we will discuss oracles in more detail.

Oracles

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation with smart contracts is that they cannot access external data because blockchains are closed systems without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract; for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts. An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract.

The following diagram shows a generic model of an oracle and smart contract ecosystem:

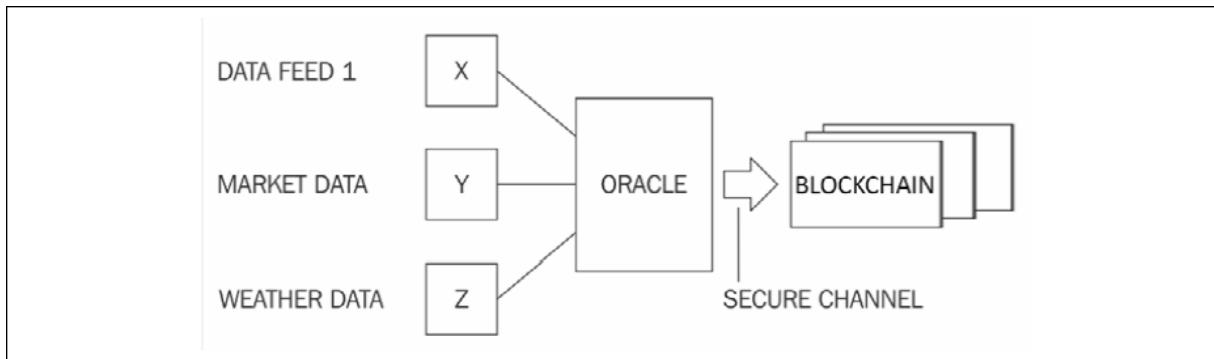


Figure 10.3: A generic model of an oracle and smart contract ecosystem

Depending on the industry and use case requirements, oracles can deliver different types of data ranging from weather reports, real-world news, and corporate actions to data coming from an **Internet of Things (IoT)** device.

A list of some of the common use cases of oracles is shown here:

Type of data	Examples	Use case
Market data	Live price feeds of financial instruments. Exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies.	DApps related to financial services, for example, decentralized exchanges and decentralized finance (DeFi)
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps
Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets

Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps
-----------	---	--

here are different methods used by oracles to write data into a blockchain, depending on the type of blockchain used. For example, in a Bitcoin blockchain, an oracle can write data to a specific transaction, and a smart contract can monitor that transaction in the blockchain and read the data. Other methods include storing the fetched data in a smart contract's storage, which can then be accessed by other smart contracts on the blockchain via requests between smart contracts depending on the platform. For example, in Ethereum, this can be achieved by using message calls.

The standard mechanics of how oracles work is presented here:

1. A smart contract sends a request for data to an oracle.
2. The request is executed and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity). Usually, TLSNotary is used for this purpose (<https://tlsnotary.org>). Other techniques include **Android proofs**, **Ledger proofs**, and **trusted hardware-assisted proofs**, which we will explain shortly.
4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system such as Swarm or IPFS and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are of a large size and

sending them to the requesting smart contracts (storing them on the chain) is not feasible.

6. Finally, the data, with the proof of validity, is sent to the smart contract.

This process can be visualized in the following diagram:

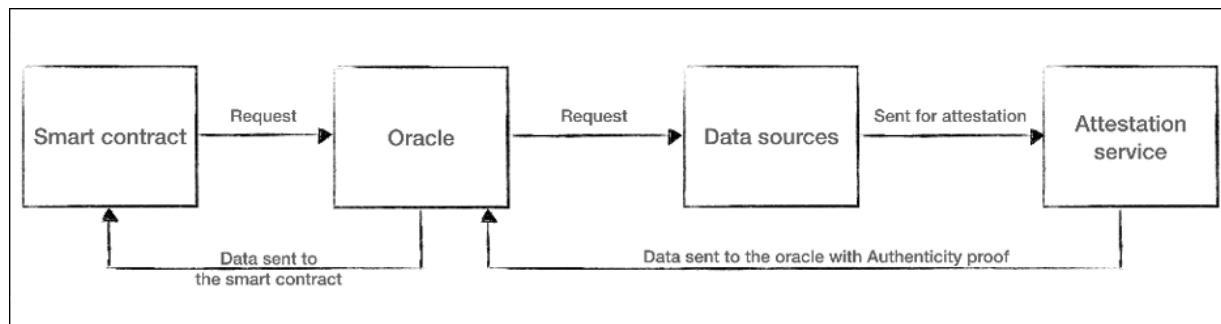


Figure 10.4: A generic oracle data flow

The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle. The oracle then requests the data from the data source, which is then sent to the attestation service for notarization. The data is sent to the oracle with proof of authenticity. Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid.

Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity** or **proof of authenticity**.

Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts. It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data. Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system. Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes. We will now introduce different mechanisms to produce cryptographic proof of data authenticity.

Software and network-assisted proofs

As the name suggests, these types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary, which is a technology developed to be primarily used in the PageSigner project (<https://tlsnotary.org/pagesigner.html>) to provide web page notarization. This mechanism can also be used to provide the required security services to oracles.

TLSNotary

This protocol provides a piece of irrefutable evidence to an auditor that specific web traffic has occurred between a client and a server. It is based on **Transport Layer Security (TLS)**, which is a standard security mechanism that enables secure, bidirectional communication between hosts. It is extensively used on the internet to secure websites and allow HTTPS traffic.

A discussion of the internals of this protocol is beyond the scope of this book. Interested readers can read the standards document at <http://www.ietf.org/rfc/rfc2246.txt>. The link provided is only for TLS version 1.0 as TLSNotary only supports TLS version 1.0 or 1.1.

The key idea behind using TLSNotary is to utilize the TLS handshake protocol's feature, which allows the splitting of the TLS master key into three parts. Each part is allocated to the server, the auditee, and the auditor. The oracle service provider (<https://provable.xyz>) becomes the auditee, whereas an **Amazon Web Services (AWS)** instance, which is secure and locked down, serves as the auditor.

In summary, to prove the authenticity of the data retrieved by oracles from external sources, attestation mechanisms such as TLSNotary are used, which produce verifiable and auditable proofs of communication between the data source and the oracle. This proof of authenticity ensures that the data fed back to the smart contract is indeed retrieved from the source.

TLS-N based mechanism

This mechanism is one of the latest developments in this space. TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy-preserving and non-interactive proofs of the content of a TLS session. TLS-N based oracles do not need to trust any third-party hardware such as Intel SGX or TLSNotary type service to provide authenticity proofs of data web content (data) to the blockchain. In contrast to TLSNotary, this scheme works on the latest TLS 1.3 standard, which allows for improved security. More information regarding this protocol is available at

<https://eprint.iacr.org/2017/578.pdf>.

Hardware device-assisted proofs

As the name suggests, these proofs rely on some hardware elements to provide proof of authenticity. In other words, they require specific hardware to work. Different mechanisms come under this category, and we will briefly introduce those next.

Android proof

This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device. SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure device provides the guarantee and confidence (that is, a proof of authenticity) that the data is authentic.

More information regarding SafetyNet and hardware attestation is available here:



<https://developer.android.com/training/safetynet>

<https://developer.android.com/training/articles/security-key-attestation.html>

Ledger proof

Ledger proof relies on the hardware cryptocurrency wallets built by the ledger company (<https://www.ledger.com>). Two hardware wallets, **Ledger Nano S** and **Ledger Blue**, can be used for these proofs. The primary purpose of these devices is as secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called **Blockchain Open Ledger Operating System (BOLOS)**, which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the **Trusted Execution Environment (TEE)** of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.



Ledger proofs, Android proofs, and TLSNotary proofs are used in provable oracles. The official documentation for these methods can be found here: <http://docs.provable.xyz>.

Currently, as these devices do not connect to the internet directly, the ledger devices cannot be used to fetch data from the internet.

Trusted hardware-assisted proofs

This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is Intel SGX. A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device.

A prime example of a trusted hardware-assisted proof is Town Crier (<https://www.town-crier.org>), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.



Intel SGX technology is developed by Intel, which provides a hardware TEE. More information about this is available here: <https://software.intel.com/en-us/sgx>. It is used by many different oracle service providers such as Town Crier and iExec (<https://iex.ec/decentralized-oracles/>).

Town Crier also provides a confidentiality service, which allows you to run confidential queries. The query for the data request is processed inside SGX Enclave, which provides a trusted execution guarantee, and the requested data is transmitted using a TLS-secured network connection, which provides additional data integrity guarantees.



It should be noted that in all of the proof techniques mentioned earlier, there are many types of resources used to provide security guarantees, including hardware, network, and software. The categorization provided here is based on the principal element, either hardware or software, which plays a critical role in the overall security mechanism to provide security.

An issue can already be seen here, and that is the issue of trust. With oracles, we are effectively trusting a third party to provide us with the correct data. What if these data sources turn malicious, or simply due to a fault start provide incorrect data to the oracles? What if the oracle itself fails or the data source stops sending data? This issue can then damage the whole blockchain trust model. This phenomenon is called the **Blockchain oracle**

problem. How do you trust a third party about the quality and authenticity of the data they provide? This question is especially real in the financial world, where market data must be accurate and reliable.

There are several proposed ways to overcome this issue. These solutions range from merely trusting a reputable third party to decentralized oracles. We have discussed some of the attestation techniques earlier; however, a third party due to a genuine fault or a malicious intent may still provide data that is incorrect. Even if it is attested later on, the actual data itself is not guaranteed to be accurate. It might be acceptable for a smart contract designer in a use case to accept data for an oracle that is provided by a large, reputable, and trusted third party. For example, the source of the data can be from a reputable weather reporting agency or airport information system directly relaying the flight delays, which can give some level of confidence. However, the issue of centralization remains.



The blockchain oracle problem can be defined formally as the conflict of trust between presumably trusted Oracles (trusted third-party data sources) and completely trustless blockchain.

Based on the evolution of blockchain over the last few years, several types of blockchain oracles have emerged. We informally provided some background on these earlier, but now we will define these formally.

Types of blockchain oracles

There are various types of blockchain oracles, ranging from simple software oracles to complex hardware assisted and decentralized oracles. Broadly speaking, we can categorize oracles into two categories: inbound oracles and outbound oracles. The following section will examine some of these in more detail.

Inbound oracles

This class represents oracles that receive incoming data from external services, and feed it into the smart contract. We will shortly discuss software, hardware, and several other types of inbound oracle.

Software oracles

These oracles are responsible for acquiring information from online services on the Internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise-specific data. These types of oracle can also be called standard or simple oracles.

Hardware oracles

This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim.

These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of IoT device's data and an anti-tampering mechanism on the IoT device, which renders the device useless in case of tampering attempts.

Computation oracles

These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive

operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain high-performance computing infrastructure and get the verified results back via an oracle. The use of oracle, in this case, provides data integrity and authenticity guarantees.

An example of such an oracle is Truebit (<https://truebit.io>). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive.

Aggregation based oracles

In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources reports the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an average of multiple values, sometimes a median is taken of all the values, and sometimes it is the maximum value. Regardless of the aggregation mechanism, the essential requirement here is to get the value that is valid and authentic, which eventually feeds into the system.

An excellent example of price feed oracles is MakerDAO (<https://makerdao.com/en/>) price feed oracle (<https://developer.makerdao.com/feeds/>), which collates price data from multiple external price feed sources and provides a median ETHUSD price to MakerDAO.

Crowd wisdom driven oracles

This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to deduce the most appropriate data eventually. In other words, it solves the problem where a single source of data may not be trustworthy or

accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, the data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites. This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

Decentralized oracles

Another type of oracles, which primarily emerged due to the decentralization requirements, is called **decentralized** oracles. Remember that in all types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the *Blockchain Oracle Problem*.

This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by distributed consensus, thus ensuring the authenticity of data. For example, one institution running their private blockchain can publish their data feed via an oracle that can then be consumed by other blockchains.

A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.

Augur (visit <https://www.augur.net/whitepaper.pdf> for Jack Peterson et al.'s essay, *A Decentralized Oracle and Prediction Market Platform*) is a prime example of such type of oracles. The Augur white

paper is also available here:
<https://arxiv.org/abs/1501.01042>.

The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which the information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.



Decentralized, crowd wisdom based and aggregation supported oracles can be categorized into a broader category of oracles called "consensus driven oracles". Augur is based on Crowd Wisdom based oracle.

Smart oracles

An idea of smart oracle has also been proposed by **Ripple labs (codius)**. Its original whitepaper is available at <https://github.com/codius/codius-wiki/wiki/White-Paper#from-oracles-to-smart-oracles>. Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

Outbound oracles

This type, also called **reverse oracles**, are used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain. The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle would be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to the legacy settlement and backend reporting systems.

Now that we have discussed different types of oracles, we now introduce different service providers that provide these services. Several service providers provide oracle services for blockchain, some of these we introduce following.

Blockchain oracle services

Various online services are now available that provide oracle services. These can also be called **oracle-as-a-service platforms**. All of these services aim to enable a smart contract to securely acquire the off-chain data it needs to execute and make decisions:

- Town Crier: <https://www.town-crier.org>
- Provable: <https://provable.xyz>
- Witnet: <https://witnet.io>
- Chainlink: <https://chain.link>
- The Realitio project: <https://realit.io>
- TrueBit: <https://truebit.io>
- iExec: <https://iex.ec>

Another service at <https://smartcontract.com/> is also available, where Ethereum, Bitcoin, and Town Crier oracles can be created. It allows smart contracts to connect to applications and allows you to feed data into the smart contracts from off-chain sources.

There are many oracle services available now, and it is challenging to cover all of them here. A random selection is presented in the preceding list.

Now that we've covered oracles in detail, let's take a look at smart contracts again and see how smart contracts can be deployed at a fundamental level.

Deploying smart contracts

Smart contracts may or may not be deployed on a blockchain, but it makes sense to do so on a blockchain due to the security and decentralized consensus mechanism provided by the blockchain. Ethereum is an example of a blockchain platform that natively supports the development and deployment of smart contracts. We will cover Ethereum in more detail later in this book, in *Chapter 11, Ethereum 101*. Smart contracts on an Ethereum blockchain are typically part of a broader DApp.

In comparison, in a Bitcoin blockchain, the transaction timelocks, such as the `nLocktime` field, the `CHECKLOCKTIMEVERIFY` (CLTV), and the `CHECKSEQUENCEVERIFY` script operator in the Bitcoin transaction, can be seen as an enabler of a simple version of a smart contract. These timelocks enable a transaction to be locked until a specified time or until a number of blocks, thus enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (elapsed time or number of blocks) are met. For example, you can implement conditions such as *Pay party X, N number of bitcoins after 3 months*. However, this is very limited and should only be viewed as an example of a basic smart contract. In addition to the example mentioned earlier, Bitcoin scripting language, though limited, can be used to construct basic smart contracts. One example of a basic smart contract is to fund a Bitcoin address that can be spent by anyone who demonstrates a **hash collision attack**.

This was a contest that was announced on the Bitcointalk forum where bitcoins were set as a reward for whoever manages to find hash collisions (we discussed this concept in *Chapter 3, Symmetric Cryptography*) for hash

functions. This conditional unlocking of Bitcoin solely on the demonstration of a successful attack is a basic type of smart contract.



This idea was presented on the Bitcointalk forum, and more information can be found at <https://bitcointalk.org/index.php?topic=293382.0>. This can be considered a basic form of a smart contract.

Various other blockchain platforms support smart contracts such as Monax, Lisk, Counterparty, Stellar, Hyperledger Fabric, Axoni core, Neo, EOSIO, and Tezos. Smart contracts can be developed in various languages, either DSLs or general-purpose languages. The critical requirement, however, is determinism, which is very important because it is vital that regardless of where the smart contract code executes, it produces the same result every time and everywhere. This requirement of the deterministic nature of smart contracts also implies that smart contract code is absolutely bug-free. Validation and verification of smart contracts is an active area of research and a detailed discussion of this topic will be presented in *Chapter 21, Scalability and Other Challenges*.

Various languages have been developed to build smart contracts such as Solidity, which runs on **Ethereum Virtual Machine (EVM)**. It's worth noting that there are platforms that already support mainstream languages for smart contract development, such as Lisk, which supports JavaScript. Another prominent example is Hyperledger Fabric, which supports Golang, Java, and JavaScript for smart contract development. A more recent example is EOSIO, which supports writing smart contracts in C++.

Security is of paramount importance for smart contracts. However, there are many vulnerabilities discovered in prevalent blockchain platforms and relevant smart contract development languages. These vulnerabilities result in some high-profile incidents, such as the DAO attack.

The DAO

The **Decentralized Autonomous Organization (DAO)**, started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug, called the **reentrancy bug**, in the code, it was hacked in June 2016. An equivalent of approximately 3.6 million ether (roughly 50 million US dollars) was siphoned out of the DAO into another account.

Even though the term hacked is used here, it was not really hacked. The smart contract did what it was asked to do but due to the vulnerabilities in the smart contracts, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack.

The DAO attack exploited a vulnerability (reentrancy bug) in the DAO code where it was possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state to indicate that how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost record of any repeated withdrawals.

The notion of *code is the law or unstoppable smart contracts* should be viewed with some skepticism as the implementation of these concepts is still not mature enough to deserve complete and unquestionable trust. This is evident from the events after the DAO incident, where the Ethereum foundation was able to stop and change the execution of the DAO by introducing a hard fork on the Ethereum blockchain. Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization, immutability, and the notion that *code is the law*. Subsequently, resistance against this hard fork resulted in the creation of Ethereum Classic, where a large number of users decided to keep mining on the old chain. This chain is the original, non-forked Ethereum blockchain that still contains the DAO. It can be said that on this chain, *the code is still the law*.

There are some interesting message threads and announcements related to this event, which readers may find informative and entertaining:

- An open letter from *The Attacker* of the DAO:
<https://pastebin.com/CcGUBgDG>
- Announcement from Ethereum core dev:
<https://twitter.com/avsa/status/745313647514226688>
- Reddit discussion on the hard fork, where points made in favor and against can be seen:
https://www.reddit.com/r/ethereum/comments/4sgihm/ethcore_blog_post_in_support_of_a_hard_fork/
- Hard fork specification: <https://blog.slock.it/hard-fork-specification-24b889e70703>

The DAO attack highlights the dangers of not formally and thoroughly testing smart contracts. It also highlights the absolute need to develop a formal language for the development and verification of smart contracts. The attack also highlighted the importance of thorough testing to avoid the issues that the DAO experienced. There have been various vulnerabilities discovered in Ethereum over the last few years regarding the smart contract development language. Therefore, it is of utmost importance that a standard framework is developed to address all these issues.

Some work has already begun, for example, an online service at <https://securify.ch>, which provides tools to formally verify smart contract code.

Another example is Michelson (<https://www.michelson.org>) for writing smart contracts in the Tezos blockchain. It is a functional programming language suitable for formal verification. We will introduce the Tezos blockchain in more detail in this book's bonus content pages, here https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

Vyper is another language that aims to provide a secure language for developing smart contracts for EVM. It aims for goals such as security, simplicity, and audibility. It is a strongly typed language with support for overflow checking and signed integers. All these features make Vyper a reasonable choice for writing secure smart contracts.

Even though many different initiatives are aiming to explore and address the security of smart contracts, this field still requires further research to address limitations in smart contract programming languages.

The security of smart contracts is an area of deep interest for researchers. A major area of interest is the formal verification of smart contracts. We will discuss these topics further in *Chapter 21, Scalability and Other Challenges*.

Summary

This chapter began by introducing a history of smart contracts followed by a detailed discussion of the definition of a smart contract. As there is no agreement on the standard definition of a smart contract, we attempted to introduce a definition that encompasses the crux of smart contracts.

An introduction to Ricardian contracts was also provided, and the difference between Ricardian contracts and smart contracts was explained, highlighting the fact that Ricardian contracts are concerned with the definition of the contract, whereas smart contracts are geared toward the actual execution of the contract.

The concept of smart contract templates was also discussed, in which high-quality active research is currently being conducted in academia and industry. Some ideas about the possibility of creating high-level DSLs were also discussed to create smart contracts or smart contract templates. In the later sections of the chapter, oracles were introduced followed by a brief discussion on the DAO along with security issues in DAO and smart contracts.

A discussion regarding formal verification and the security of smart contracts will be presented later in this book, in *Chapter 21, Scalability and Other Challenges*.

In the next chapter, we will introduce Ethereum, which is one of the most popular blockchain platforms that inherently supports smart contracts.

Ethereum 101

This chapter is an introduction to the Ethereum blockchain. We will introduce the fundamentals and various theoretical concepts behind Ethereum. A discussion on the different components, protocols, and algorithms relevant to the Ethereum blockchain is also presented.

We cover different elements of Ethereum such as transactions, accounts, the world state, the **Ethereum Virtual Machine (EVM)**, and the relevant protocols so that readers can develop strong technical foundations before exploring more advanced concepts in later chapters. The main topics we will explore in this chapter are as follows:

- The Ethereum network
- Components of the Ethereum ecosystem
- The Ethereum Virtual Machine (EVM)
- Smart contracts
- Trading and investment

Let's begin with a brief overview of the foundation, architecture, and use of the Ethereum blockchain.

Ethereum – an overview

Vitalik Buterin (<https://vitalik.ca>) conceptualized Ethereum (<https://ethereum.org>) in November, 2013. The core idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and **Decentralized Applications (DApps)**. This concept is in contrast to Bitcoin, where the scripting language is limited and only allows necessary operations.

The first version of Ethereum, called Olympic, was released in May, 2015. Two months later, a version of Ethereum called Frontier was released in July. Another version named Homestead with various improvements was released in March, 2016. The latest Ethereum release is called Muir Glacier, which delays the difficulty bomb

(<https://eips.ethereum.org/EIPS/eip-2384>). A major release before that was Istanbul, which included changes around privacy and scaling capabilities.



The difficulty bomb is a difficulty adjustment mechanism that eventually forces all miners to stop mining on Ethereum 1 and move to Ethereum 2. Once activated, over time, it makes mining on Ethereum 1.x so prohibitively slow that mining becomes infeasible and results in a so-called "ice age". In other words, this mechanism exponentially increases the **Proof-of-Work (PoW)** mining difficulty to a level where block generation becomes impossible, thus forcing the miners to migrate to Ethereum 2.0's **Proof-of-Stake (PoS)** system.

The following table shows all the major upgrades of Ethereum, starting from the first release to the planned final release:

Release	Description	Release date	Details and original announcement
Olympic	Pre-release	May 9, 2015	https://blog.ethereum.org/2015/05/09/olympic-frontier-pre-release/
Frontier	First live network	July 30, 2015	https://blog.ethereum.org/2015/03/12/getting-to-the-frontier/
Frontier Thawing	Initial protocol adjustments	September 08, 2015	https://blog.ethereum.org/2015/08/04/the-thawing-frontier/
Homestead	Second major version	March 15, 2016	https://blog.ethereum.org/2016/02/29/homestead-release/
Spurious Dragon	EIP55	November 23, 2016	https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/
Byzantium	Metropolis part 1	October 16, 2017	https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/
Constantinople St. Petersburg	Metropolis part 2	February 28, 2019	https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/
Istanbul	Metropolis part 3	December 08, 2019	https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement/

Muir Glacier	Unplanned release	January 01, 2020	https://blog.ethereum.org/2019/12/23/ethereum-muir-glacier-upgrade-announcement/
Berlin	Second phase of Istanbul	Q3 2020	https://eips.ethereum.org/EIPS/eip-2070
Serenity	ETH 2.0	To be implemented in different phases	

The final planned release of Ethereum is called Serenity and is envisaged to introduce the final version of the PoS based-blockchain, replacing PoW. This chapter covers Istanbul, which at the time of writing is the latest major upgrade of Ethereum. We will cover Serenity and relevant concepts in *Chapter 16, Serenity*.

A list of all releases as announced is maintained at
<https://github.com/ethereum/go-ethereum/releases>.

The formal specification of Ethereum has been described in the *yellow paper*, which can be used to develop Ethereum implementations.

The yellow paper

The Ethereum yellow paper (<https://ethereum.github.io/yellowpaper/paper.pdf>) was written by Dr. Gavin Wood, the founder of Ethereum and Parity (<http://gavwood.com>), and serves as a formal specification of the Ethereum protocol. Anyone can implement an Ethereum client by following the protocol specifications defined in the paper.

While this paper can be somewhat challenging to read, especially for those who do not have a background in algebra or mathematics and are not familiar with mathematical notation, it contains a complete formal specification of Ethereum. This specification can be used to implement a fully compliant Ethereum client. Therefore, it appears necessary to understand this paper at least at a high level.

The list of all symbols with their meanings used in the paper is provided here with the anticipation that it will make reading the Ethereum yellow paper more accessible.

Useful mathematical symbols

The following table shows the mathematical symbols used in the yellow paper, along with their meaning:

Symbol	Meaning	Symbol	Meaning
\equiv	Is defined as	\leq	Less than or equal to
$=$	Is equal to	σ	Sigma, the world state
$\not\equiv$	Is not equal to	μ	Mu, the machine state
$\ \dots\ $	Length of	μ	Upsilon, Ethereum state transition function
\in	Is an element of	\prod	Block-level state transition function
\notin	Is not an element of	.	Sequence concatenation
\forall	For all	\exists	There exists
\cup	Union	Δ	Contract creation function
\wedge	Logical AND	Δ	Increment
:	Such that	$\lfloor \dots \rfloor$	Floor, lowest element
{}	Set	$\lceil \dots \rceil$	Ceiling, highest element
\circ	Function of tuple	$ \dots $	Number of bytes
[]	Array indexing	\oplus	Exclusive OR
\vee	Logical OR	(a,b)	Real numbers $\geq a$ and $< b$
$>$	Is greater than	\emptyset	Empty set, null
$+$	Addition		
$-$	Subtraction		

Σ	Summation		
{	Describing various cases of if, otherwise		

Now, in the next and upcoming sections, we will introduce the Ethereum blockchain and its core elements.

The Ethereum blockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This definition is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood.

The core idea is that in the Ethereum blockchain, a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state. In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition:

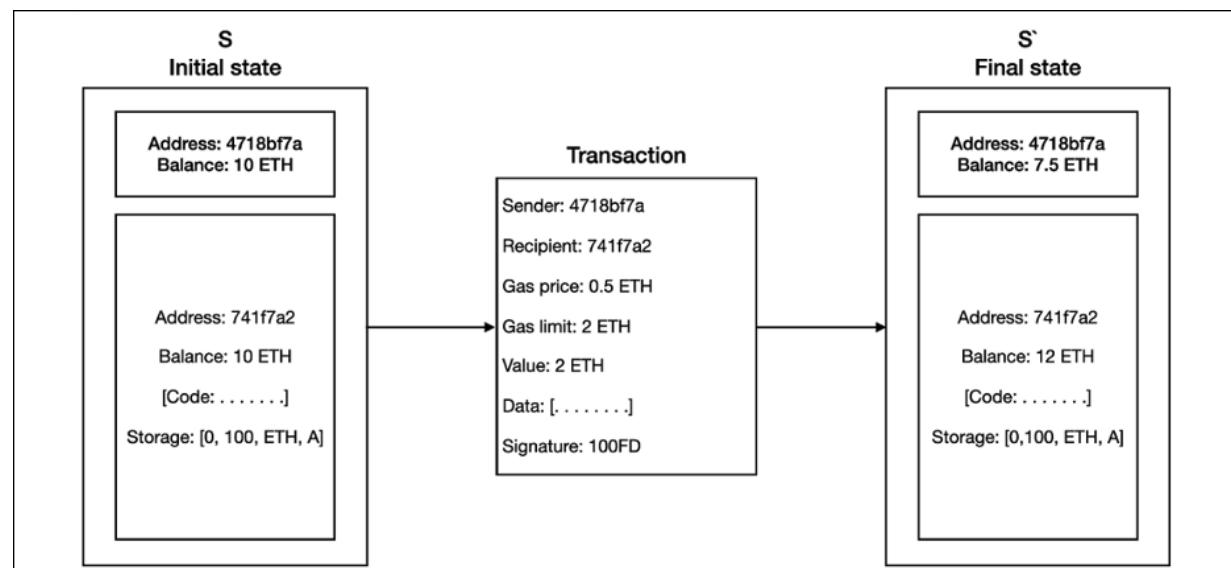


Figure 11.1: Ethereum state transition function

In the preceding example, a transfer of two ether from address **4718bf7a** to address **741f7a2** is initiated. The initial state represents the state before the transaction execution, and the final state is what the morphed state looks like. Mining plays a central role in state transition, and we will elaborate the mining process in detail in the later sections. The state is stored on the Ethereum network as the *world state*. This is the global state of the Ethereum blockchain.

More on this will be presented later in the *Nodes and miners* section in *Chapter 12, Further Ethereum*, in the context of state storage.

Ethereum – a user's perspective

In this section, we will see how Ethereum works from a user's point of view. For this purpose, we will present the most common use case of transferring funds – in our use case, from one user (Bashir) to another (Irshad). We will use two Ethereum clients, one for sending funds and the other for receiving. There are several steps involved in this process, as follows:

1. First, either a user requests money by sending the request to the sender, or the sender decides to send money to the receiver. We'll be using the Jaxx Ethereum wallet software on iOS – you can download the Jaxx wallet from <https://jaxx.io>. The request can be sent by sending the receiver's Ethereum address to the sender. For example, there are two users, Bashir and Irshad. If Irshad requests money from Bashir, then she can send a request to Bashir by using a QR code. Once Bashir receives this request he will either scan the QR code or manually type in Irshad's Ethereum address and send the ether to Irshad's address. This request is encoded as a QR code, shown in the following screenshot, that can be shared via email, text, or any other communication method:



Figure 11.2: QR code as shown in the blockchain wallet application

2. Once Bashir receives this request he will either scan this QR code or copy the Ethereum address in the Ethereum wallet software and initiate a transaction. This process is shown in the following screenshot, where Jaxx is used to send money to Irshad. The following screenshot shows that the sender has entered both the amount and the destination address to send the ether to receiver. Just before sending the ether, the final step is to confirm the transaction, which is also shown here:





We have used a version of the Jaxx wallet and blockchain wallet in this example, but any wallet software can be used to achieve the same functionality. The aim of these examples is not to introduce wallet software, but to show how the transaction flow works in fundamentally the same way for all wallet software. There are many different instances of wallet software available online for the iOS, Android, and desktop operating systems.

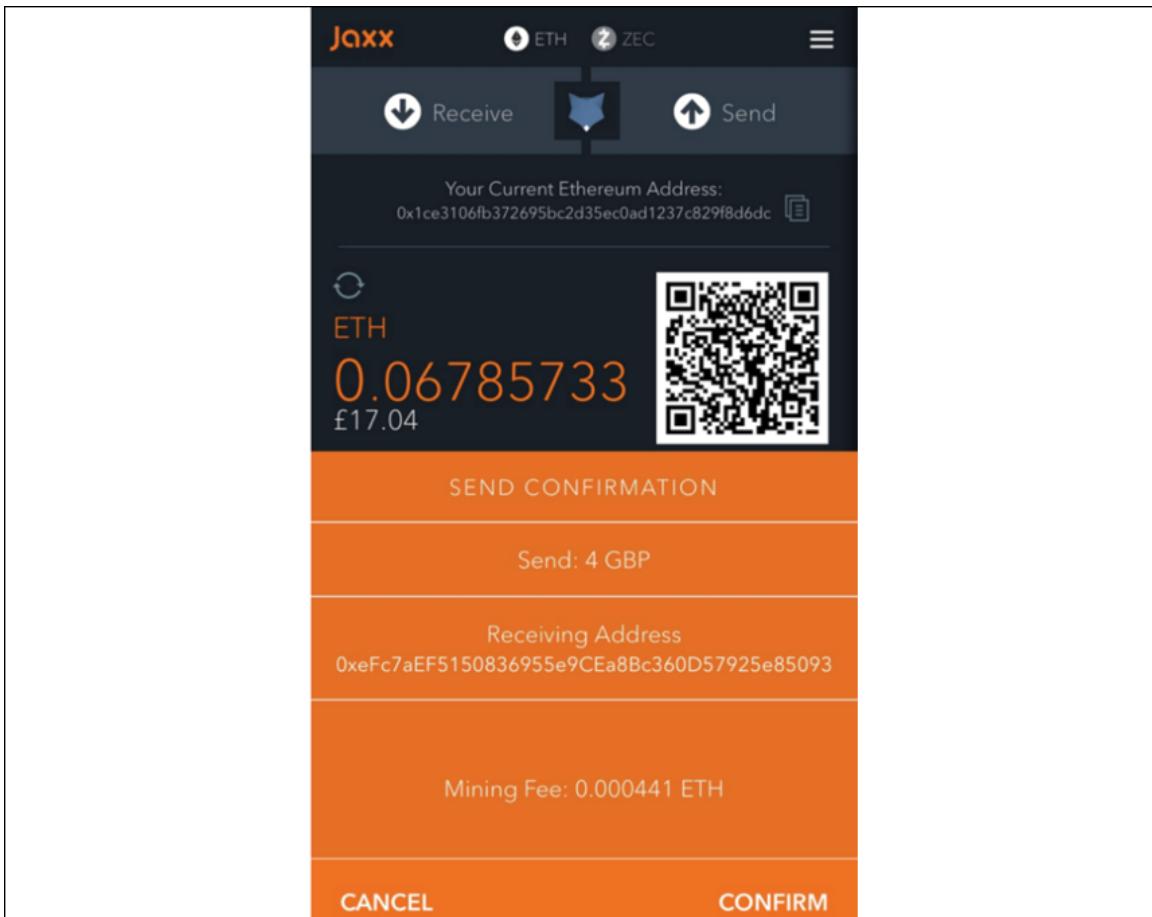


Figure 11.3: Confirmation of funds sent in the Jaxx wallet from Bashir

3. Once the request (transaction) for money to be sent is constructed in the wallet software, it is then broadcasted to the Ethereum network. The transaction is digitally signed by the sender as proof that he is the owner of the ether.
4. This transaction is then picked up by nodes called miners on the Ethereum network for verification and inclusion in the block. At this stage, the transaction is still unconfirmed.
5. Once it is verified and included in the block, the PoW process starts. We will explain this process in more detail later in *Chapter 12, Further Ethereum*.
6. Once a miner finds the answer to the PoW problem by repeatedly hashing the block with a new nonce, this block is immediately broadcasted to the rest of the nodes, which then verifies the block and PoW.

7. If all the checks pass then this block is added to the blockchain, and miners are paid rewards accordingly.
8. Finally, Irshad gets the ether, and it is shown in her wallet software. This is shown in the following screenshot:

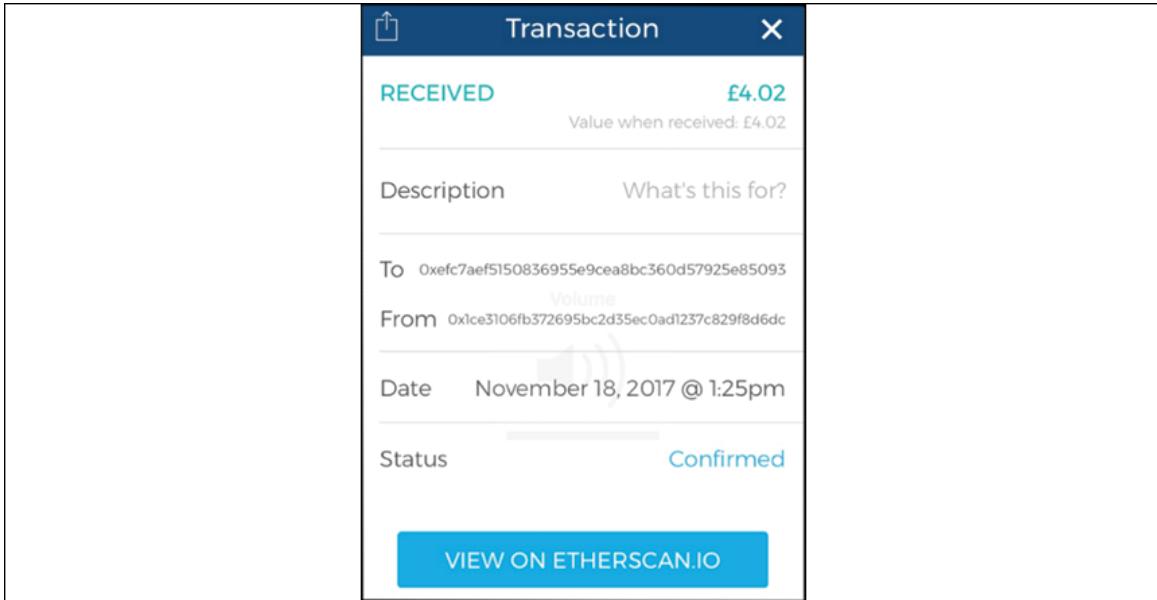


Figure 11.4: The transaction received in Irshad's blockchain wallet

On the blockchain, this transaction is identified by the following transaction hash:

0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e

The details of this transaction can be visualized on the block explorer at <https://etherscan.io/>, as shown in the following screenshot:

Overview	State Changes	Comments
⑦ Transaction Hash:	0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e	
⑦ Status:	Success	
⑦ Block:	4576084	4659657 Block Confirmations
⑦ Timestamp:	⑧ 780 days 7 hrs ago (Nov-18-2017 01:25:54 PM +UTC)	
⑦ From:	0x1ce3106fb372695bc2d35ec0ad1237c829f8d6dc	
⑦ To:	0xfc7aef5150836955e9cea8bc360d57925e85093	
⑦ Value:	0.015927244142974896 Ether	(\$2.29)
⑦ Transaction Fee:	0.000441 Ether	(\$0.06)
⑦ Gas Limit:	21,000	
⑦ Gas Used by Transaction:	21,000 (100%)	
⑦ Gas Price:	0.000000021 Ether (21 Gwei)	
⑦ Nonce	Position	1 4

Figure 11.5: Etherscan Ethereum blockchain block explorer



Note the transaction hash (TxHash) at the top. Later in the next chapter, and we will use this hash to see how this transaction is constructed, processed, and stored in the blockchain. This hash is the unique ID of the transaction that can be used to trace this transaction throughout the blockchain network.

With this example, we complete our discussion on the most common usage of the Ethereum network: transferring ether from a user to another. This case was just a quick overview of the transaction process in order to introduce the concept. More in-depth technical details will be explained in the upcoming sections of this chapter where we discuss various components of the Ethereum ecosystem.

In the next section, we will see what components make up the Ethereum ecosystem. Once we have understood all the theoretical concepts, we will see in detail how the aforementioned transaction travels through the Ethereum blockchain to reach the beneficiary's address. At this point, we will be able to correlate the technical concepts with the preceding transfer transaction example.

The Ethereum network

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

The mainnet

The **mainnet** is the current live network of Ethereum. Its network ID is 1 and its chain ID is also 1. The network and chain IDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at <https://etherscan.io>. This can be used to explore the Ethereum blockchain.

Testnets

There is a number of testnets available for Ethereum testing. The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain. Moreover, being test networks, they also allow experimentation and research. The main testnet is called *Ropsten*, which contains all the features of other smaller and special-purpose testnets that were created for specific releases. For example, other testnets include *Kovan* and *Rinkeby*, which were developed for testing Byzantium releases. The changes that were implemented on these smaller testnets have also been implemented in Ropsten. Now the Ropsten test network contains all properties of Kovan and Rinkeby.

Private nets

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



A table detailing some major Ethereum networks, including their network IDs and other relevant details, can be found in this book's online resource page here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf

Network IDs and chain IDs are used by Ethereum clients to identify the network. Chain ID was introduced in EIP155 as part of replay protection mechanism. EIP155 is detailed at <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md>.



A comprehensive list of Ethereum networks is maintained and available at
<https://chainid.network>.

More discussion on how to connect to a testnet and how to set up private nets will be had in *Chapter 13, Ethereum Development Environment*.

Components of the Ethereum ecosystem

The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer Ethereum network. Secondly, there's an Ethereum client (usually Geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally. It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network. Another component is the `web3.js` library that allows interaction with the `geth` client via the **Remote Procedure Call (RPC)** interface.

The overall Ethereum ecosystem architecture is visualized in the following diagram:

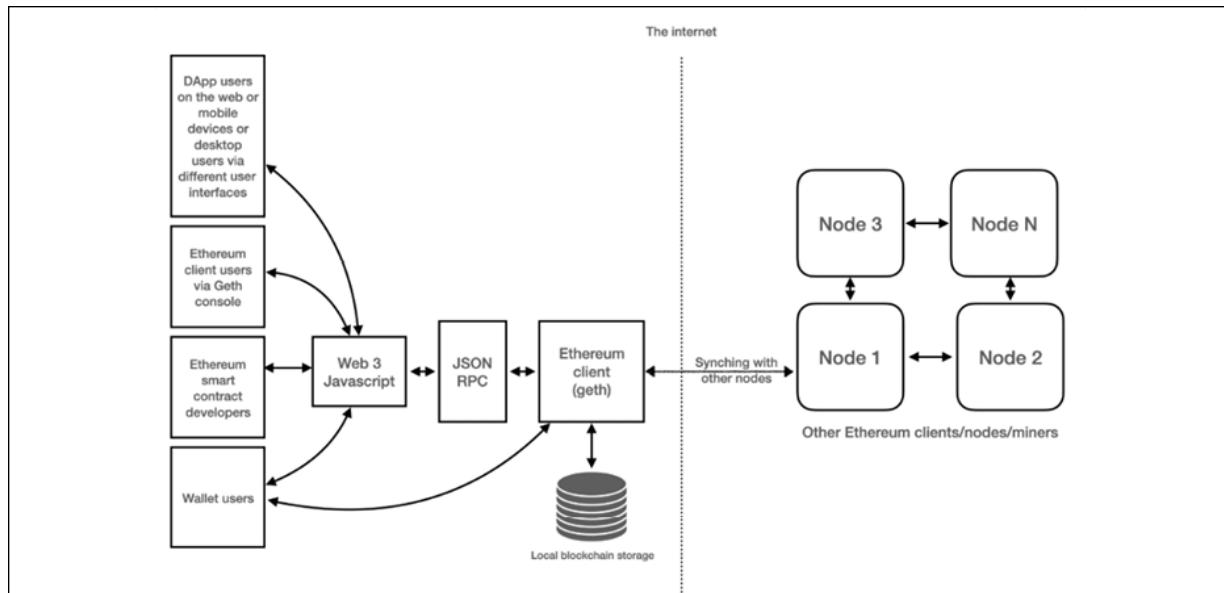


Figure 11.6: Ethereum high-level ecosystem

A list of elements present in the Ethereum blockchain is presented here:

- Keys and addresses

- Accounts
- Transactions and messages
- Ether cryptocurrency/tokens
- The EVM
- Smart contracts and native contracts

In the following sections, we will discuss each of these one by one. Other components such as wallets, clients, development tools, and environments will be discussed in the upcoming chapters.

We will also discuss relevant technical concepts related to a high-level element within that section. First, let's have a look at keys and addresses.

Keys and addresses

Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether. The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas a public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer) under the rules defined by the elliptic curve `secp256k1` specification (in the range $[1, \text{secp256k1n} - 1]$).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm (ECDSA)** recovery function. We will discuss this in the following *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

An example of how keys and addresses look in Ethereum is shown as follows:

- A private key:

```
b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4
```

- A public key:

```
3aa5b8eef12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847  
f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260
```

- An address:

```
0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32
```



The preceding private key is shown here only as an example, and should not be reused.

Another key element in Ethereum is an account, which is required in order to interact with the blockchain. It either represents a user or a smart contract.

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. They are defined by pairs of private and public keys. Accounts are used by users to interact with the blockchain via transactions. A transaction is digitally signed by an account before submitting it to the network via a node. Ethereum, being a *transaction-driven state machine*, the state is created or updated as a result of the interaction between accounts and transaction executions. All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. Operations performed between and on the accounts represent state transitions. The state transition is achieved using what's called the Ethereum state transition function, which works as follows:

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

Now as we understand accounts in Ethereum generally, let's examine the different types of accounts in Ethereum.

Types of accounts

Two kinds of accounts exist in Ethereum:

- **Externally Owned Accounts (EOAs)**
- **Contract Accounts (CAs)**

The first type is EOAs, and the other is CAs. EOAs are similar to accounts that are controlled by a private key in Bitcoin. CAs are the accounts that have code associated with them along with the private key.

The various properties of each type of account are as follows:

EOAs

- They have a state.
- They are associated with a human user, hence are also called user accounts.
- EOAs have an ether balance.
- They are capable of sending transactions.
- They have no associated code.
- They are controlled by private keys.
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

CAs

- They have a state.
- They are not intrinsically associated with any user or actor on the blockchain.
- CAs have an ether balance.
- They have associated code that is kept in memory/storage on the blockchain. They have access to storage.
- They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within CAs can be of any level of complexity. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter in the *The Ethereum Virtual Machine (EVM)* section.
- Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in the Serenity (Ethereum 2.0) release, the distinction between EOAs and CAs may be eliminated.

- CAs cannot start transaction messages.
- CAs can initiate a call message.
- CAs contain a key-value store.
- CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

Accounts allow interaction with the blockchain via transactions. We will now explain what an Ethereum transaction is and consider its different types.

Transactions and messages

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

- **Message call transactions:** This transaction simply produces a message call that is used to pass messages from one CA to another.
- **Contract creation transactions:** As the name suggests, these transactions result in the creation of a new CA. This means that when this transaction is executed successfully, it creates an account with the associated code.

Both of these transactions are composed of some standard fields, which are described as follows:

- **Nonce:** The nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce value can only be used once. This is used for replay protection on the network.
- **Gas price:** The **gas price** field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction.



Wei is the smallest denomination of ether; therefore, it is used to count ether.

- **Gas limit:** The **gas limit** field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction. The concept of gas and gas limits will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the fee amount, in ether, that a user (for example, the sender of the transaction) is willing to pay for computation.

- **To:** As the name suggests, the **To** field is a value that represents the address of the recipient of the transaction. This is a 20 byte value.
- **Value:** Value represents the total number of Wei to be transferred to the recipient; in the case of a CA, this represents the balance that the contract will hold.
- **Signature:** The signature is composed of three fields, namely **V**, **R**, and **S**. These values represent the digital signature (R , S) and some information that can be used to recover the public key (V). Also, the sender of the transaction can also be determined from these values. The signature is based on the ECDSA scheme and makes use of the `secp256k1` curve. The theory of **Elliptic Curve Cryptography (ECC)** was discussed in *Chapter 4, Public Key Cryptography*. In this section, ECDSA will be presented in the context of its usage in Ethereum.

V is a single byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. V is used in the ECDSA recovery contract as a recovery ID. This value is used to recover (derive) the public key from the private key. In `secp256k1`, the recovery ID is expected to be either 0 or 1. In Ethereum, this is offset by 27. More details on the `ECDSARECOVER` function will be provided later in this chapter.

R is derived from a calculated point on the curve. First, a random number is picked, which is multiplied by the generator of the curve to calculate a point on the curve. The x -coordinate part of this point is R . R is encoded as a 32-byte sequence. R must be greater than 0 and less than the `secp256k1n` limit

(`115792089237316195423570985008687907852837564279074904382605163141518161494337`).

S is calculated by multiplying R with the private key and adding it into the hash of the message to be signed, and then finally dividing it by the random number chosen to calculate R . S is also a 32-byte sequence. R and S together represent the signature.

To sign a transaction, the `ECDSASIGN` function is used, which takes the message to be signed and the private key as an input and produces V , a single-byte value; R , a 32-byte value; and S , another 32-byte value. The equation is as follows:

$$ECDSASIGN (Message, Private Key) = (V, R, S)$$

- **Init:** The **Init** field is used only in transactions that are intended to create contracts, that is, contract creation transactions. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process. The code contained in this field is executed only once when the account is created for the first time, it (init) gets destroyed immediately after that. Init also returns another code section called the *body*, which persists and runs in response to message calls that the CA may receive. These message calls may be sent via a transaction or an internal code execution.
- **Data:** If the transaction is a message call, then the **Data** field is used instead of **init**, and represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This structure is visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a **transaction trie** (a modified **Merkle-Patricia tree (MPT)**) composed of the transactions to be included. Finally, the root node of the transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block:

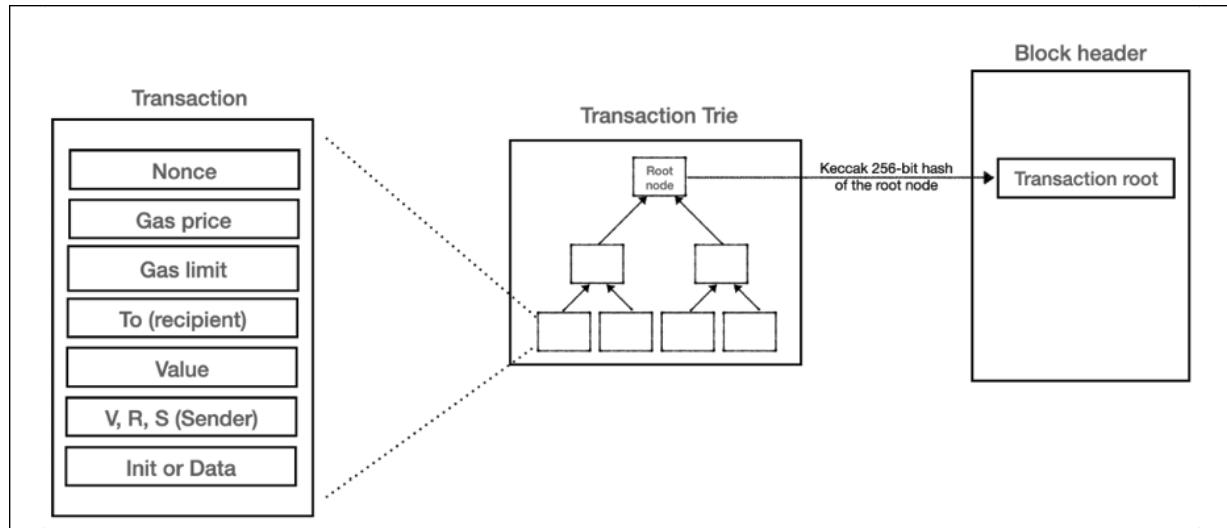


Figure 11.7: The relationship between the transaction, transaction trie, and block header

A block is a data structure that contains batches of transactions. Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. When a mining node starts its operation of verifying blocks, it starts with the highest-paying transactions in the transaction pool and executes them one by one. When the gas limit is reached, or no more transactions are left to be processed in the transaction pool, the mining starts.

In this process, the block is repeatedly hashed until a valid nonce is found, such that once hashed with the block, it results in a value less than the difficulty target. Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network. This process is similar to Bitcoin's mining process discussed in the previous chapters, *Chapter 6, Introducing Bitcoin* and *Chapter 7, Bitcoin Network and Payments*. Ethereum's PoW algorithm is called Ethash, and it was originally intended to be ASIC-resistant where finding a nonce requires large amounts of memory. However, now some ASICs are available for Ethash too.

Consequently, there is also an agreement among Ethereum core developers to implement *ProgPow*, a new, more ASIC-resistant PoW consensus mechanism. We will explain ASICs and relevant concepts in *Chapter 12, Further Ethereum*, where we discuss mining in detail. We will also discuss block data structure in greater detail in this chapter.

The question that arises here is how all these accounts, transactions, and related messages flow through the Ethereum network and how are they stored. We saw earlier, in the

Ethereum – a user's perspective section, a transaction flow example of how funds can be sent over the network from one user to another.

We need to know how this data looks on the network. So, before we move on to the different types of transactions and messages in Ethereum, let's explore how Ethereum data is encoded for storage and transmission. For this purpose, a new encoding scheme called **Recursive Length Prefix (RLP)** was developed, which we will cover here in detail.

RLP

To define RLP, we first need to understand the concept of serialization. Serialization is simply a mechanism commonly used in computing to encode data structures into a format (a sequence of bytes) that is suitable for storage and/or transmission over the communication links in a network. Once a receiver receives the serialized data, it de-serializes it to obtain the original data structure. Serialization and deserialization are also referred as marshaling and un-marshaling, respectively. Some commonly used serialization formats include XML, JSON, YAML, protocol buffers, and XDR. There are two types of serialization formats, namely *text* and *binary*. In a blockchain, there is a need to serialize and deserialize different types of data such as blocks, accounts, and transactions to support transmission over the network and storage on clients.



Why do we need a new encoding scheme when there are so many different serialization formats already available? The answer to this question is that RLP is a deterministic scheme, whereas other schemes may produce different results for the same input, which is absolutely unacceptable on a blockchain. Even a small change will lead to a totally different hash and will result in data integrity problems that will render the entire blockchain useless.

RLP is an encoding scheme developed by Ethereum developers. It is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in a Patricia tree on storage media. It is a deterministic and consistent binary encoding scheme used to serialize objects on the Ethereum blockchain, such as account state, transactions, messages, and blocks. It operates on strings and lists to produce raw bytes that are suitable for storage and transmission. RLP is a minimalistic and simple-to-implement serialization format that does not define any data types and simply stores structures as nested arrays. In other words, RLP does not encode specific data types; instead, its primary purpose is to encode structures.



More information on RLP is available on the Ethereum wiki, at
<https://eth.wiki/en/fundamentals/rlp>.

Now, having defined RLP, we can delve deeper into transactions and other relevant elements of the Ethereum blockchain. We will start by exploring different types of transactions on the Ethereum blockchain.

Contract creation transactions

A contract creation transaction is used to create smart contracts on the blockchain. There are a few essential parameters required for a contract creation transaction. These parameters are listed as follows:

- The sender
- The transaction originator
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- The current depth of the message call/contract-creation stack (current depth means the number of items that are already present in the stack)

Addresses generated as a result of a contract creation transaction are 160 bits in length. Precisely as defined in the yellow paper, they are the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. The storage is also set to empty. The code hash is a Keccak 256-bit hash of the empty string.

The new account is initialized when the EVM code (the initialization EVM code, mentioned earlier) is executed. In the case of any exception during code execution, such as not having enough gas (running **Out of Gas**, or **OOG**), the state does not change. If the execution is successful, then the account is created after the payment of appropriate gas costs.

Since **Ethereum Homestead**, the result of a contract creation transaction is either a new contract with its balance, or no new contract is created with no transfer of value. This is in contrast to versions prior to Homestead, where the contract would be created regardless of the contract code deployment being successful or not due to an OOG exception.

Message call transactions

A message call requires several parameters for execution, which are listed as follows:

- The sender
- The transaction originator
- The recipient
- The account whose code is to be executed (usually the same as the recipient)

- Available gas
- The value
- The gas price
- An arbitrary-length byte array
- The input data of the call
- The current depth of the message call/contract creation stack

Message calls result in a state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by EVM code, the output produced by the transaction execution is used. As defined in the yellow paper, a message call is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the EVM will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object (external actor), then the call passes back any data returned from the EVM operation.

The state is altered by transactions. These transactions are created by external factors (users) and are signed and then broadcasted to the Ethereum network.

Messages are passed between accounts using message calls. A description of messages and message calls is presented next.

Messages

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

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

A message consists of the following components:

- The sender of the message
- The recipient of the message
- Amount of Wei to transfer and the message to be sent to the contract address
- An optional data field (input data for the contract)
- The maximum amount of gas (`startgas`) that can be consumed

Messages are generated when the `CALL` or `DELEGATECALL` opcodes are executed by the contract running in the EVM.

In the following diagram, the segregation between the two types of transactions (**contract creation** and **message calls**) is shown:

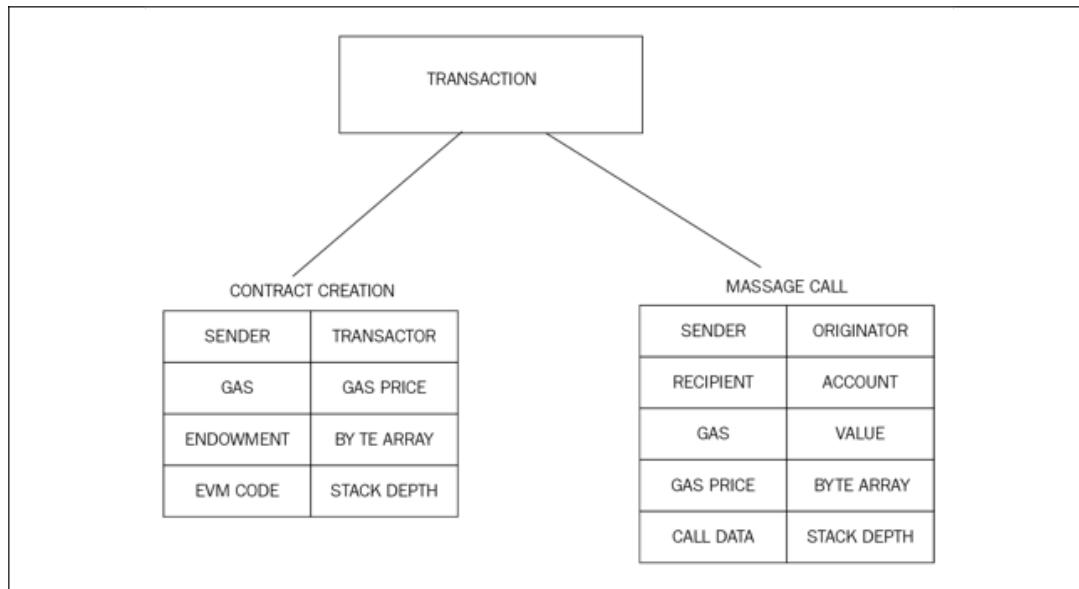


Figure 11.8: Types of transactions and the required parameters for execution

Each of these transactions has fields, which are shown with each type.

Calls

A call does not broadcast anything to the blockchain; instead, it is a local call and executes locally on the Ethereum node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. It is akin to a dry run or a simulated run. Calls also do not allow ether transfer to CAs. Calls are executed locally on a node EVM and do not result in any state change because they are never mined. Calls are processed synchronously and they usually return the result immediately.



Do not confuse a *call* with a *message call transaction*, which in fact results in a state change. A call basically runs message call transactions locally on the client and never costs gas nor results in a state change. It is available in the `web3.js` JavaScript API and can be seen as almost a simulated mode of the message call transaction. On the other hand, a *message call transaction* is a `write` operation and is used for invoking functions in a CA (Contract Account, or smart contract), which does cost gas and results in a state change.

Transaction validation and execution

Transactions are executed after their validity has been verified. The initial checks are listed as follows:

- A transaction must be well formed and RLP-encoded without any additional trailing bytes
- The digital signature used to sign the transaction must be valid
- The transaction nonce must be equal to the sender's account's current nonce
- The gas limit must not be less than the gas used by the transaction
- The sender's account must contain sufficient balance to cover the execution cost

The transaction substate

A transaction substate is created during the execution of the transaction and is processed immediately after the execution completes. This transaction substate is a tuple that is composed of four items. These items are as follows:

- **Suicide set or self-destruct set:** This element contains the list of accounts (if any) that are disposed of after the transaction executes.
- **Log series:** This is an indexed series of checkpoints that allows the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked, or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage. Events will be covered with practical examples in *Chapter 15, Introducing Web3*.
- **Refund balance:** This is the total price of gas for the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to offset the total execution cost partially.
- **Touched accounts:** Touched accounts can be defined as those accounts which are involved any potential state changing operation. Empty accounts from this set are deleted at the end of the transaction. The final state is reached after deleting accounts in the self-destruct set and emptying accounts from the touched accounts set.

State storage in the Ethereum blockchain

At a fundamental level, the Ethereum blockchain is a transaction- and consensus-driven state machine. The state needs to be stored permanently in the blockchain. For this purpose, the world state, transactions, and transaction receipts are stored on the blockchain in blocks. We discuss these components next.

The world state

This is a *mapping* between Ethereum addresses and account states. The addresses are 20 bytes (160 bits) long. This mapping is a data structure that is serialized using RLP.

The account state

The account state consists of four fields: nonce, balance, storage root, and code hash, and is described in detail here:

- **Nonce:** This is a value that is incremented every time a transaction is sent from the address. In the case of CAs, it represents the number of contracts created by the account.
- **Balance:** This value represents the number of weis, which is the smallest unit of the currency (ether) in Ethereum, held by the given address.
- **Storage root:** This field represents the root node of an MPT that encodes the storage contents of the account.
- **Code hash:** This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.

The world state and its relationship with the accounts trie, accounts, and block header are visualized in the following diagram. It shows the **account state**, or data structure, which contains a **storage root** hash derived from the **root node** of the **account storage trie** shown on the left. The account data structure is then used in the **world state trie**, which is a mapping between addresses and account states.

The accounts trie is an MPT used to encode the storage contents of an account. The contents are stored as a mapping between Keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Finally, the **root node** of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the **block header** data structure, which is shown on the right-hand side of the diagram as the **state root** hash:

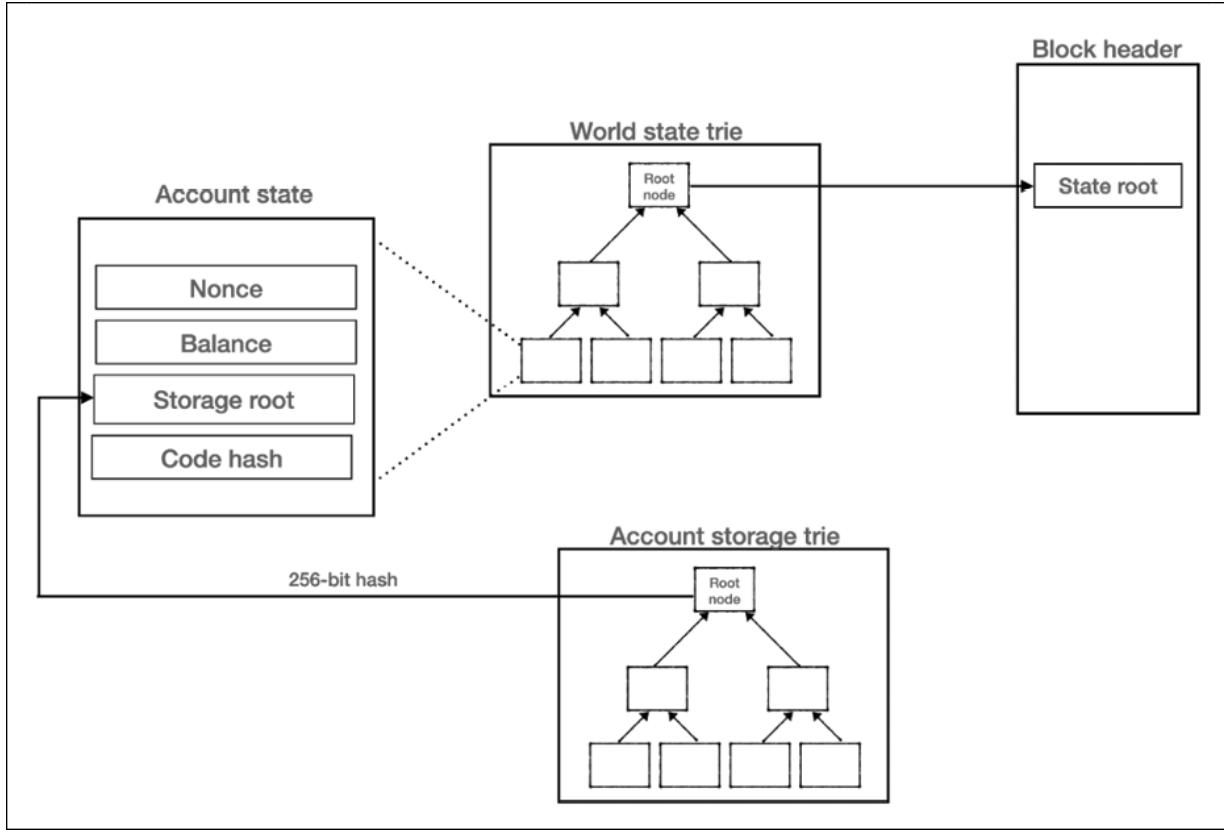


Figure 11.9: The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationship

Transaction receipts

Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. It is produced after the execution of each transaction. All receipts are stored in an index-keyed trie. The hash (a 256-bit Keccak hash) of the root of this trie is placed in the block header as the receipt's root. It is composed of four elements as follows:

- **The post-transaction state:** This item is a trie structure that holds the state after the transaction has been executed. It is encoded as a byte array.
- **Gas used:** This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is expected to be a non-negative integer.
- **Set of logs:** This field shows the set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.
- **The bloom filter:** A bloom filter is created from the information contained in the set of logs discussed earlier. A log entry is reduced to a hash of 256 bytes, which is then

embedded in the header of the block as the logs bloom. A log entry is composed of the logger's address, log topics, and log data. Log topics are encoded as a series of 32-byte data structures. The log data is made up of a few bytes of data.

Since the Byzantium release, an additional field returning the success (1) or failure (0) of the transaction is also available.



More information about this change is available at
<https://github.com/ethereum/EIPs/pull/658>.

This process of transaction receipt generation is visualized in the following diagram:

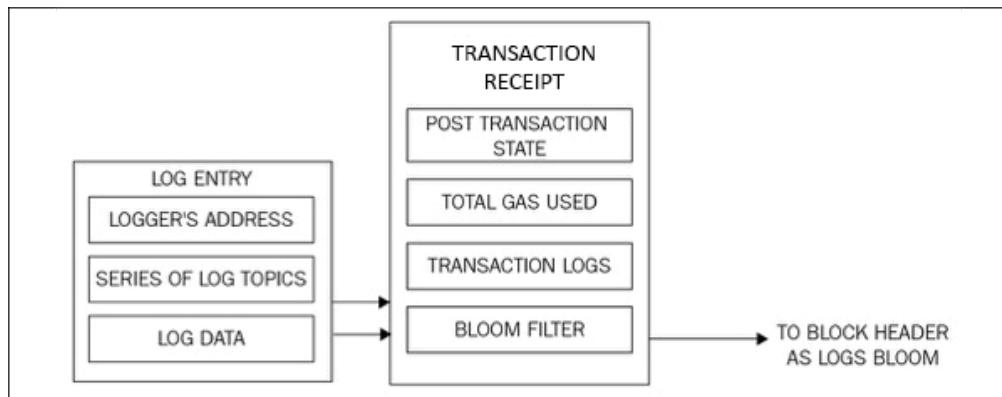


Figure 11.10: Transaction receipts and logs bloom

As a result of the transaction execution process, the state morphs from an initial state to a target state. This concept was discussed briefly at the beginning of the chapter. This state needs to be stored and made available globally in the blockchain. We will see how this process works in the next section.

Ether cryptocurrency/tokens (ETC and ETH)

As an incentive to the miners, Ethereum rewards its own native currency called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization (DAO)** hack described in *Chapter 10, Smart Contracts*, a hard fork was proposed in order to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its own following with a dedicated community that is further developing ETC, which is the unforked original version of Ethereum.

This chapter is focused on ETH, which is currently the most active and official Ethereum blockchain.

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

The denomination table is as follows:

Unit	Alternative name	Wei value	Number of weis
Wei	Wei	1 Wei	1
KWei	Babbage	1e3 Wei	1,000
MWei	Lovelace	1e6 Wei	1,000,000
GWei	Shannon	1e9 Wei	1,000,000,000
microether	Szabo	1e12 Wei	1,000,000,000,000
milliether	Finney	1e15 Wei	1,000,000,000,000,000
ether	ether	1e18 Wei	1,000,000,000,000,000,000

Fees are charged for each computation performed by the EVM on the blockchain. A detailed fee schedule is described in *Chapter 12, Further Ethereum*.

The Ethereum Virtual Machine (EVM)

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256-bit. The stack size is limited to 1,024 elements and is based on the **Last In, First Out (LIFO)** queue. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements. The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing

invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain.

As discussed earlier, the EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

There are three main types of storage available for contracts and the EVM:

- **Memory:** The first type is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. `write` operations to the memory can be of 8 or 256 bits, whereas `read` operations are limited to 256-bit words. Memory is unlimited but constrained by gas fee requirements.
- **Storage:** The other type is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack:** EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory (virtual ROM)** that is accessible using the `CODECOPY` instruction. The `CODECOPY` instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM.

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

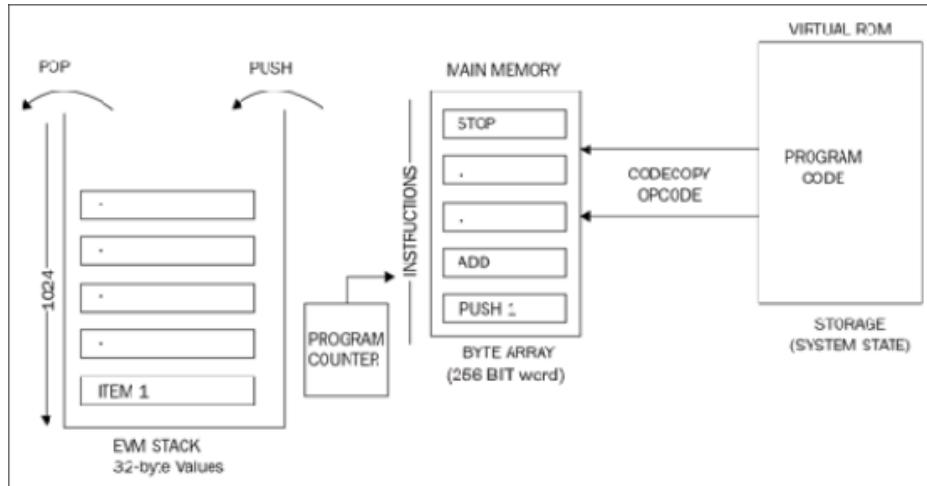


Figure 11.11: EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained, and is incremented with instructions being read from the main memory. The main memory gets the program code from the virtual ROM/storage via the `CODECOPY` instruction.

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly** (`ewasm`)—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly (Wasm)** was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed. More information and GitHub repository of Ethereum-flavored Wasm is available at <https://github.com/ewasm>.

Another intermediate language called YUL, which can compile to various backends such as the EVM and `ewasm`, is under development. More information on this language can be found at <https://solidity.readthedocs.io/en/latest/yul.html>.

Execution environment

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

- The system state.
- The remaining gas for execution.
- The address of the account that owns the executing code.

- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
- The gas price of the transaction that initiated the execution.
- Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
- The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it is the address of the account.
- The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- The number of message calls or contract creation transactions (CALLs, CREATEs or CREATE2s) currently in execution.
- Permission to make modifications to the state.

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

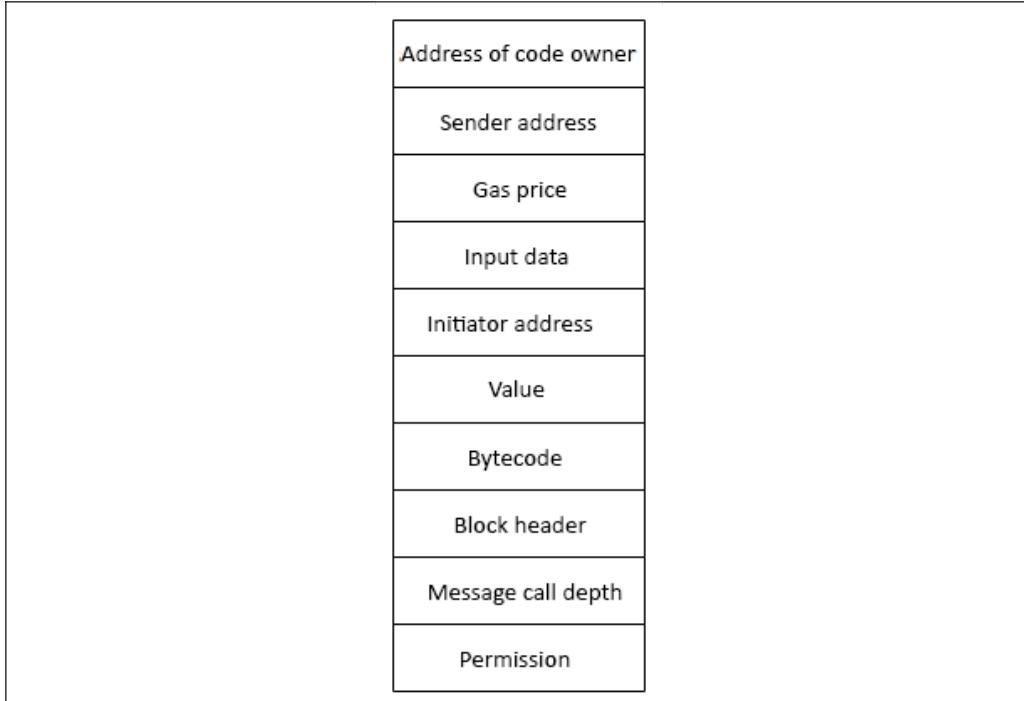


Figure 11.12: Execution environment tuple

The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is also maintained internally, and updated after each execution cycle of the EVM. An iterator function (detailed in the next section) runs in the EVM, which outputs the results of a single cycle of the state machine.

The machine state is a tuple that consists of the following elements:

- Available gas
- The program counter, which is a positive integer of up to 256
- The contents of the memory (a series of zeroes of size 2^{256})
- The active number of words in memory (counting continuously from position 0)
- The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions should occur:

- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

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

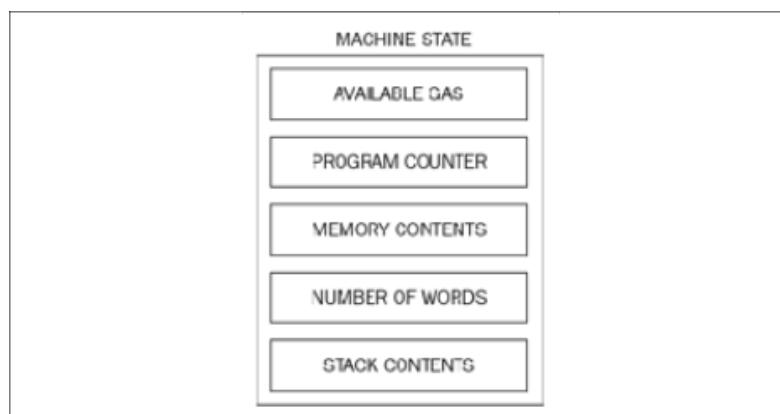


Figure 11.13: Machine state tuple

The iterator function

The iterator function mentioned earlier performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

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

The EVM is also able to halt in normal conditions if `STOP`, `SUICIDE`, or `RETURN` opcodes are encountered during the execution cycle.

Smart contracts

We discussed smart contracts at length in the previous *Chapter 10, Smart Contracts*. It is sufficient to say here that Ethereum supports the development of smart contracts that run on the EVM. Different languages can be used to build smart contracts, and we will discuss this in the programming section and at a deeper level in *Chapter 14, Development Tools and Frameworks* and *Chapter 15, Introducing Web3*.

There are also various contracts that are available in precompiled format in the Ethereum blockchain to support different functions. These contracts, known as **precompiled contracts** or **native contracts**, are described in the following subsection.

These are not strictly smart contracts in the sense of user-programmed Solidity smart contracts, but are in fact functions that are available natively to support various computationally intensive tasks. They run on the local node and are coded within the Ethereum client; for example, `parity` or `geth`.

Native contracts

There are nine **precompiled contracts** or **native contracts** in the Ethereum Istanbul release. The following subsections outline these contracts and their details.

The elliptic curve public key recovery function

`ECDSARECOVER` (the ECDSA recovery function) is available at address `0x1`. It is denoted as `ECREC` and requires 3,000 gas in fees for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in ECC.

The ECDSA recovery function is shown as follows:

$$\text{ECDSARECOVER}(H, V, R, S) = \text{Public Key}$$

It takes four inputs: H , which is a 32-byte hash of the message to be signed, and V, R , and S , which represent the ECDSA signature with the recovery ID and produce a 64-byte public key. V, R , and S have been discussed in detail previously in this chapter.

The SHA-256-bit hash function

The SHA-256-bit hash function is a precompiled contract that is available at address `0x2` and produces a `SHA256` hash of the input. The gas requirement for SHA-256 (`SHA256`) depends on the input data size. The output is a 32-byte value.

The RIPEMD-160-bit hash function

The RIPEMD-160-bit hash function is used to provide a RIPEMD 160-bit hash and is available at address `0x3`. The output of this function is a 20-byte value. The gas requirement, similar to SHA-256, is dependent on the amount of input data.

The identity/datacopy function

The identity function is available at address `0x4` and is denoted by the `ID`. It simply defines output as input; in other words, whatever input is given to the `ID` function, it will output the same value. The gas requirement is calculated by a simple formula: $15 + 3/[Id/32]$, where Id is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data, albeit with some calculation performed, as shown in the preceding equation.

Big mod exponentiation function

This function implements a native big integer exponential modular operation. This functionality allows for RSA signature verification and other cryptographic operations. This is available at address `0x05`.

Elliptic curve point addition function

We discussed elliptic curve addition in detail at a theoretical level in *Chapter 4, Public Key Cryptography*. This is the implementation of the same elliptic curve point addition function. This contract is available at address `0x06`.

Elliptic curve scalar multiplication

We discussed elliptic curve multiplication (point doubling) in detail at a theoretical level in *Chapter 4, Public Key Cryptography*. This is the implementation of the same elliptic curve point multiplication function. Both elliptic curve addition and doubling functions allow for ZK-SNARKS and the implementation of other cryptographic constructs. This contract is available at `0x07`.

Elliptic curve pairing

The elliptic curve pairing functionality allows for performing elliptic curve pairing (bilinear maps) operations, which enables zk-SNARKS verification. This contract is available at address `0x08`.

Blake2 compression function 'F'

This precompiled contract allows the `BLAKE2b` hash function and other related variants to run on the EVM. This improves interoperability with Zcash and other Equihash-based PoW chains. This precompiled contract is implemented at address `0x09`. The EIP is available at <https://eips.ethereum.org/EIPS/eip-152>.



More information on the Blake hash function is available at <https://blake2.net>.

All the aforementioned precompiled contracts can potentially become native extensions and might be included in the EVM opcodes in the future.



All pre-compiled contracts are present in the `contracts.go` file in the Ethereum source code. It is available at the following link:

<https://github.com/ethereum/go-ethereum/blob/8bd37a1d919194d9a488d1cee823eaecfeb5ed9a/core/vm/contracts.go#L66>



Next hard fork release of Ethereum called Berlin will introduce new pre-compiled contracts for BLS12-381 curve operations along with other updates. You can follow the EIP-2070 for further updates on this release. EIP is available here at

<https://eips.ethereum.org/EIPS/eip-2070>

We will continue our discussion on Ethereum in the next chapter. Readers are also encouraged to explore some extra content on Ethereum networks, and trading and investing, on this book's online resource page here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

Summary

This chapter started with a discussion on the history of Ethereum, the motivation behind Ethereum development, and Ethereum clients. Then, we introduced the core concepts of the Ethereum blockchain, such as the state machine model, the world and machine states, accounts, and types of accounts. Moreover, a detailed introduction to the core components of the EVM was also presented, along with some of the fundamentals of Ethereum trading and investing.

In the next chapter, we will continue to explore more Ethereum concepts. We will look at more ideas such as programming languages, blockchain data structures, blocks, mining, and various Ethereum clients.

12

Further Ethereum

This chapter is a continuation of the previous chapter, in which we will examine more Ethereum-based concepts in more detail.

We will cover both a practical and theoretical in-depth introduction to wallet software, mining, and setting up Ethereum nodes. Material relating to various challenges, such as security and scalability faced by Ethereum, will also be introduced. Moreover, prominent advanced supporting protocols, such as Swarm and Whisper, will also be introduced later in the chapter. Finally, Ethereum has several programming languages built in to support smart contract development. We will conclude with an overview of these programming languages.

This chapter continues the discussion of the Ethereum blockchain network elements that we started in the previous chapter, *Chapter 11, Ethereum 101*. These elements include:

- Blocks and blockchain
- Wallets and client software
- Nodes and miners
- APIs and tools
- Supporting protocols
- Programming languages

First, we will introduce blocks and blockchain, in continuation of the discussion from the previous chapter.

Blocks and blockchain

Blocks are the main building structure of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

- The block header
- The transactions list
- The list of headers of ommers or uncles



An uncle block is a block that is the child of a parent but does not have any child block. Ommers or uncles are valid, but stale, blocks that are not part of the main chain but contribute to security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is also included in the block.

Block header: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are described in detail here:

- **Parent hash:** This is the Keccak 256-bit hash of the parent (previous) block's header.
- **Ommers hash:** This is the Keccak 256-bit hash of the list of ommers (or uncles) blocks included in the block.
- **The beneficiary:** The beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.
- **State root:** The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated once all transactions have been processed and finalized.
- **Transactions root:** The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. The transaction trie represents the list of transactions included in the block.

- **Receipts root:** The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.
- **Logs bloom:** The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.
- **Difficulty:** The difficulty level of the current block.
- **Number:** The total number of all previous blocks; the genesis block is block zero.
- **Gas limit:** This field contains the value that represents the limit set on the gas consumption per block.
- **Gas used:** This field contains the total gas consumed by the transactions included in the block.
- **Timestamp:** The timestamp is the epoch Unix time of the time of block initialization.
- **Extra data:** The extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.
- **Mixhash:** The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **PoW**) has been spent in order to create this block.
- **Nonce:** Nonce is a 64-bit hash (a number) that is used to prove, in combination with the *mixhash* field, that adequate computational effort (PoW) has been spent in order to create this block.

The following diagram shows the detailed structure of the block and block header:

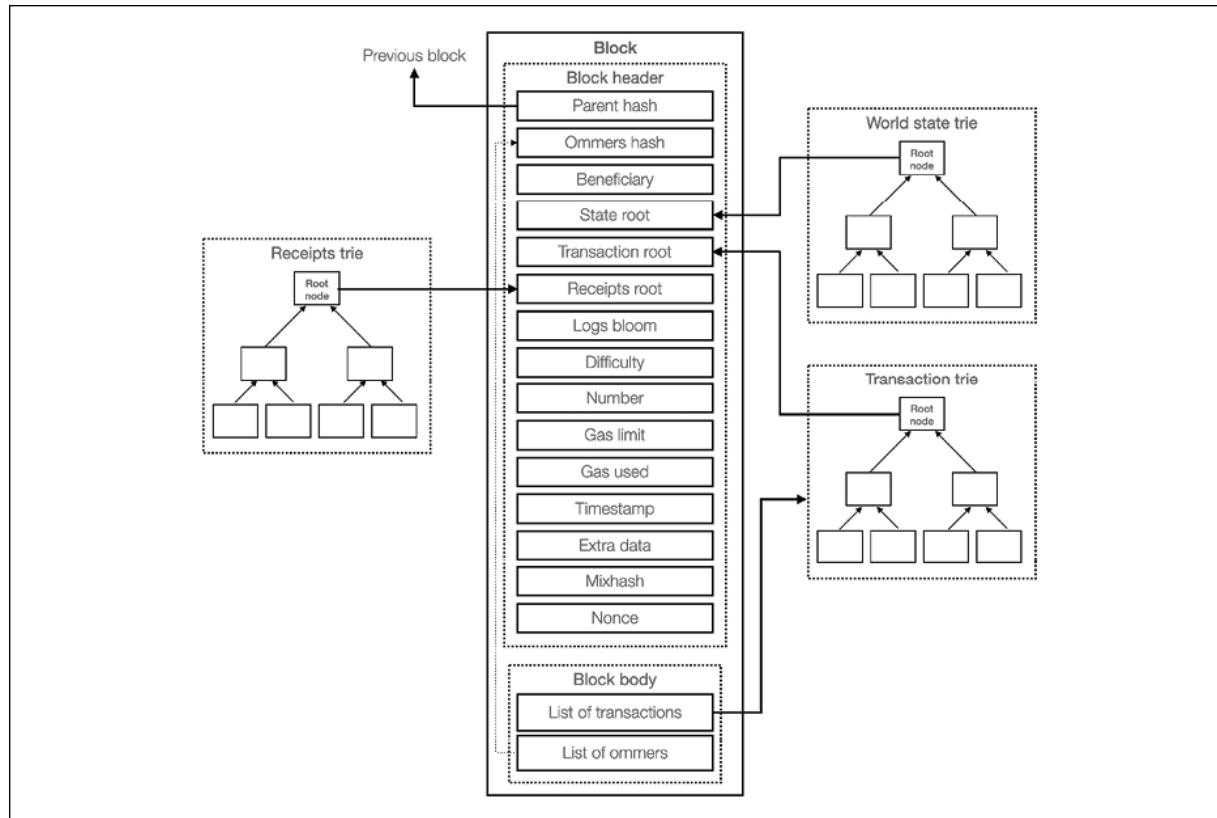


Figure 12.1: A detailed diagram of the block structure with a block header and relationship with tries

The genesis block

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created. It contains 15 items that are described here.

From <https://etherscan.io/>, the actual version is shown as follows:

Element	Description
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block

The block validation mechanism

An Ethereum block is considered valid if it passes the following checks:

- If it is consistent with uncles and transactions. This means that all ommers satisfy the property that they are indeed uncles and also if the PoW for uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).
- If any of these checks fails, the block will be rejected. A list of errors for which the block can be rejected is presented here:
 - The timestamp is older than the parent
 - There are too many uncles
 - There is a duplicate uncle
 - The uncle is an ancestor
 - The uncle's parent is not an ancestor
 - There is non-positive difficulty
 - There is an invalid mix digest
 - There is an invalid PoW

Block finalization

Block finalization is a process that is run by miners to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail:

1. **Ommers validation.** In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.

2. **Transaction validation.** In the case of mining, determine transactions. This process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction, in other words, the cumulative gas used by the transactions included in the block.
3. **Reward application.** Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 2 ether. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum. Later, in the Constantinople release (<https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/>), it was reduced further to 2 ether. A block can have a maximum of two uncles.
4. **State and nonce validation.** Verify the state and block nonce. In the case of mining, compute a valid state and block nonce.

After the high-level view of the block validation mechanism, we now look into how a block is received and processed by a node. We will also see how it is updated in the local blockchain:

1. When an Ethereum full node receives a newly mined block, the header and the body of the block are detached from each other. Now remember, in the last chapter, when we introduced the fact that there are three **Merkle Patricia tries (MPTs)** in an Ethereum blockchain. The roots of those MPTs or tries are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. We will now learn how these tries are used to validate the blocks.
2. A new MPT is constructed that comprises all transactions from the block.
3. All transactions from this new MPT are executed one by one in a sequence. This execution occurs locally on the node within the **Ethereum Virtual Machine (EVM)**. As a result of this execution,

new transaction receipts are generated that are organized in a new receipts MPT. Also, the global state is modified accordingly, which updates the state MPT (trie).

4. The root nodes of each respective trie, in other words, the state root, transaction root, and receipts root are compared with the header of the block that was split in the first step. If both the roots of the newly constructed tries and the trie roots that already exist in the header are equal, then the block is verified and valid.
5. Once the block is validated, new transaction, receipt, and state tries are written into the local blockchain database.

Block difficulty mechanism

Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's **Homestead** release is as follows:

```
block_diff = parent_diff + parent_diff // 2048 *  
max(1 - (block_timestamp - parent_timestamp) // 10, -99) +  
int(2**((block.number // 100000) - 2))
```



Note that `//` is the integer division operator.

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up by $parent_diff // 2048 * 1$. If the time difference is between 10 and 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficulty level decreases. This decrease is proportional to the time difference from $parent_diff // 2048 * -1$ to a maximum decrease of $parent_diff // 2048 * -99$.

In addition to timestamp difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to **Proof of Stake (PoS)**, since mining on the PoW chain will eventually become prohibitively difficult.



The difficulty time bomb was delayed via EIP-649 (<https://github.com/ethereum/EIPs/pull/669>) for around 18 months and no clear time frame has yet been suggested.

According to the original estimates based on the algorithm, the block generation time would have become significantly higher during the second half of 2017, and in 2021, it would become so high that it would be virtually impossible to mine on the PoW chain, even for dedicated mining centers. This way, miners will have no choice but to switch to the PoS scheme proposed by Ethereum, called **Casper**.



More information about Casper is available here:
https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf.

This ice age proposal has been postponed with the release of **Byzantium**. Instead, the mining reward has been reduced from 5 ETH to 3 ETH, in preparation for PoS implementation in Serenity.

In the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for difficulty calculation. This new formula is shown here:

$$\text{adj_factor} = \max((2 \text{ if } \text{len}(\text{parent.uncles}) \text{ else } 1) - ((\text{timestamp} - \text{parent.timestamp}) // 9), -99)$$



Soon after the **Istanbul** upgrade, the difficulty bomb was delayed once again, under the **Muir Glacier** network upgrade with a hard fork, <https://eips.ethereum.org/EIPS/eip-2384>, for roughly another 611 days. The change was activated at block number 9,200,000 on January 2, 2020.

We know that blocks are composed of transactions and that there are various operations associated with transaction creation, validation, and finalization. Each of these operations on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas. We will now introduce this mechanism in detail.

Gas

Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator.

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund.

Transaction costs can be estimated using the following formula:

$$\text{Total cost} = \text{gasUsed} * \text{gasPrice}$$

Here, *gasUsed* is the total gas that is supposed to be used by the transaction during the execution, and *gasPrice* is specified by the transaction originator as an incentive to the miners to include the transaction in the next block.

This is specified in ETH. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or fewer operations than intended initially and can result in consuming more or less gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and some gas remains, then it is returned to the transaction originator.



A website that keeps track of the latest gas price and provides other valuable statistics and calculators is available at
<https://ethgasstation.info/index.php>.

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

Operation name	Gas cost
Stop	0
SHA3	30
SLOAD	800
Transaction	21000
Contract creation	32000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA-3 operation can be calculated as follows:

- SHA-3 costs 30 gas.

- Assume that the current gas price is 25 GWei, and convert it into ETH, which is 0.000000025 ETH. After multiplying both, $0.000000025 * 30$, we get 0.00000075 ETH.
- In total, 0.00000075 ETH is the total gas that will be charged.

Fee schedule

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

- The computation of an operation
- For contract creation or message calls
- An increase in the use of memory

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

Now let's move onto another topic, clients and wallets, which are responsible for mining, payments, and management functions, such as account creation on an Ethereum network.

Wallets and client software

As Ethereum is under heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years.

Wallets

A wallet is a generic program that can store private keys and, based on the addresses stored within it, it can compute the existing balance of ether associated with the addresses by querying the blockchain. It can also be

used to deploy smart contracts. In this chapter, we will introduce the **MetaMask** wallet, which has become the tool of choice for developers.

Having discussed the role of wallets within Ethereum, let's now discuss a number of common clients. The following is a non-exhaustive list of the client software and wallets that are available with Ethereum.

Geth

This is the official **Go** implementation of the Ethereum client.



The latest version is available at the following link:
<https://geth.ethereum.org/downloads/>.

Eth

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



Eth is available at the GitHub repository:
<https://github.com/ethereum/aleth>.

Parity

This implementation is built using **Rust** and developed by **Parity** technologies.



Parity can be downloaded from the following link:
<https://wwwparity.io/>

Note that Parity is now OpenEthereum. Henceforth, we will use the name OpenEthereum.

Trinity

Trinity is the implementation of the Ethereum protocol. It is written in **Python**.



Trinity can be downloaded from <https://github.com/ethereum/trinity>. The official website of Trinity client is <https://trinity.ethereum.org>.

Light clients

Simple Payment Verification (SPV) clients download only a small subset of the blockchain. This allows low resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify the transactions.

A complete Ethereum blockchain and node are not required in this case, and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to Bitcoin SPV clients.



There is a wallet available from Jaxx (<https://jaxx.io/>), which can be installed on iOS and Android, which provides the SPV functionality.

The critical difference between clients and wallets is that clients are full implementations of the Ethereum protocol, which support mining, account management, and wallet functions. In contrast, wallets only store the public and private keys, provide essential account management, and interact with the blockchain for usually only payment (transfer of funds) purposes.

Next, we discuss the installation procedure and usage of some of these clients.

Installation and usage

First, we describe Geth and explore various operations that can be performed using this client.

Geth

The following installation procedure describes the installation of Ethereum clients on macOS and Linux.



Instructions for other operating systems are available on the official Ethereum documentation site at
<https://geth.ethereum.org/docs/install-and-build/installing-geth>.

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

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

On a macOS using homebrew, Geth can be installed by executing the following commands:

```
$ brew tap ethereum/ethereum
$ brew install ethereum
```

If an older version of Ethereum `geth` client is already installed, it can be upgraded by issuing the following command:

```
$ brew upgrade ethereum
```

Once installation is complete, Geth can be launched simply by issuing the `geth` command at the terminal. It comes preconfigured with all the required parameters to connect to the live Ethereum network (mainnet):

```
$ geth
```

This will produce output similar to the following:

```
INFO [01-18|21:00:06.590] Bumping default cache on mainnet provided=1024 updated=4096
WARN [01-18|21:00:06.590] Sanitizing cache to Go's GC limits provided=4096 updated=1314
INFO [01-18|21:00:06.592] Maximum peer count ETH=50 LES=0 total=50
INFO [01-18|21:00:06.595] Starting peer-to-peer node instance=Geth/v1.9.9-stable-01744997/Linux-amd64/go1.13.4
INFO [01-18|21:00:06.597] Allocated trie memory caches clean=328.00MiB dirty=328.00MiB
INFO [01-18|21:00:06.598] Allocated cache and file handles database=/home/vagrant/.ethereum/geth/chaindata cache=657.00MiB handles=524288
INFO [01-18|21:00:06.749] Opened ancient database database=/home/vagrant/.ethereum/geth/chaindata/ancient
INFO [01-18|21:00:06.793] Initialised chain configuration config="(ChainID: 1 Homestead: 1150000 DAO: 1920000 DAOSupport: true EIP150: 2463000 EIP155
: 2675000 EIP158: 2675000 Byzantium: 4370000 Constantinople: 7280000 Petersburg: 7280000 Istanbul: 9069000 Muir Glacier: 9200000, Engine: ethash)"
INFO [01-18|21:00:06.794] Disk storage enabled for ethash caches dir=/home/vagrant/.ethereum/geth/ethash count=3
INFO [01-18|21:00:06.794] Disk storage enabled for ethash DAGs dir=/home/vagrant/.ethereum/ethash count=2
INFO [01-18|21:00:06.795] Initialising Ethereum protocol versions="[64 63]" network=1 dbversion=7
INFO [01-18|21:00:06.801] Loaded most recent local header number=2496 hash=f3bf5_246460 td=83203260865481 age=4y6mo1w
INFO [01-18|21:00:06.803] Loaded most recent local full block number=0 hash=d4e567_c8bf03 td=17179869184 age=50y9mo1w
INFO [01-18|21:00:06.804] Loaded most recent local fast block number=1541 hash=90af6_ca6647 td=39237279173210 age=4y6mo1w
INFO [01-18|21:00:06.806] Loaded local transaction journal transactions=0 dropped=0
INFO [01-18|21:00:06.808] Regenerated local transaction journal transactions=0 accounts=0
INFO [01-18|21:00:06.839] Allocated fast sync bloom size=656.00MiB
INFO [01-18|21:00:07.154] New local node record seq=7 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:07.171] Started P2P networking self=enode://8d8a401311d88e7016edda12fa73a0b8ea8e563c0272e66ba1a3de055bfb4ec5a61bae6c054802
c9c3a3314ec7f81155820ea99d8985989b51ee1426260dd17@127.0.0.1:30303
INFO [01-18|21:00:07.173] IPC endpoint opened url=/home/vagrant/.ethereum/geth.ipc
INFO [01-18|21:00:07.962] Initialized fast sync bloom items=13046 errorrate=0.000 elapsed=1.120s
WARN [01-18|21:00:11.009] Dropping unsynced node during fast sync id=04471e22084d2e5d conn=dyndial addr=211.228.238.136:30303 type=Geth/v1.9.3-stable-cfb969
d/linux-amd64/go1.11.5
INFO [01-18|21:00:11.633] New local node record seq=8 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:13.137] New local node record seq=9 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:15.371] New local node record seq=10 id=bd0df49fbe92d6a9 ip=82.2.27.41 udp=54057 tcp=30303
INFO [01-18|21:00:27.175] Block synchronisation started count=0 elapsed=8.796ms number=1733 hash=3ddbe0_378d04 age=4y6mo1w ignored=192
INFO [01-18|21:00:40.927] Imported new block headers count=0 elapsed=11.186ms number=1925 hash=052de6_33e43d age=4y6mo1w ignored=192
INFO [01-18|21:00:41.391] Imported new block headers count=7 elapsed=29.573ms number=1548 hash=3e6adc_4a9bb5 age=4y6mo1w size=5.54KiB
INFO [01-18|21:00:41.403] Imported new block receipts
```

Figure 12.2: Geth

When the Ethereum client starts up, it starts to synchronize with the rest of the network. There are three types of synchronization mechanisms available, namely, full, fast, and light.

- **Full:** In this synchronization mode, the Geth client downloads the complete blockchain to its local node. This means that it gets all the block headers and block bodies and validates all transaction and blocks since the genesis block. Currently (as at early 2020), the Ethereum blockchain size is roughly 210 GB, and downloading and maintaining that could be a problem. Usually, SSDs are recommended for a full node, so that disk latency cannot cause any processing delays.
- **Fast:** The client downloads the full blockchain, but it retrieves and verifies only the previous 64 blocks from the current block. After this, it verifies the new blocks in full. It does not replay and verify all historic transactions since the genesis block; instead it only does the state downloads. This also reduces the on-disc size of the blockchain

database quite significantly. This is the default mode of Geth client synchronization.

- **Light**: This is the quickest mode and only downloads and stores the current state trie. In this mode, the client does not download any historic blocks and only processes newer blocks.

Synchronization mode is configurable on the `geth` client via the flag:

```
--syncmode value
```

Here, the value can be either `fast`, `full`, or `light`.

Ethereum account management using Geth

New accounts can be created via the command line using Geth or OpenEthereum (formerly Parity Ethereum) command-line interface. This process is shown in the next section for `geth`, which we installed in the previous *Installation and usage* section.

Creating a Geth new account

Execute the following command to add a new account:

```
$ geth account new
```

This command will produce output similar to the following:

```
INFO [01-18|17:04:01.460] Maximum peer count          ETH=50 LES=0 total=50
Your new account is locked with a password. Please give a password. Do not forget this password.
Password:
Repeat password:

Your new key was generated

Public address of the key: 0x8A9425E0b5747726402b0d5D8E4d5F0d5D70AdF5
Path of the secret key file: /Users/drequinox/Library/Ethereum/keystore/UTC--2020-01-18T17-04-06.852541000Z--8a9425e0b5747726402bdd5d8e4d5f0d5d70adf5

- You can share your public address with anyone. Others need it to interact with you.
- You must NEVER share the secret key with anyone! The key controls access to your funds!
- You must BACKUP your key file! Without the key, it's impossible to access account funds!
- You must REMEMBER your password! Without the password, it's impossible to decrypt the key!

+ ~ ||
```

Figure 12.3: Geth account generation

The list of accounts can be displayed using the `geth` client by issuing the following command:

```
$ geth account list
```

This command will produce output similar to the following:

```
INFO [01-18|17:59:25.595] Maximum peer count      ETH=50 LES=0  
Account #0: {07668e548be1e17f3dcfa2c4263a0f5f88aca402} keystore:  
Account #1: {ba94fb1f306e4d53587fcfdcd7eab8109a2e183c4} keystore:  
Account #2: {1df5ae40636d6a1a4f8db8a0c65addce5a992a14} keystore:
```



Note that you will see different addresses and directory paths when you run this on your computer.

In this example, we saw how to create a new account and obtain a list of accounts. Next, we'll see different methods to interact with the blockchain.

How to query the blockchain using Geth

There are different methods available to query with the blockchain. First, in order to connect to the running instance of the client, either a local IPC or RPC API can be used.

There are three methods of interacting with the blockchain using Geth:

1. Geth console
2. Geth attach
3. Geth JSON RPC

Geth console and Geth attach are used to interact with the blockchain using an REPL JavaScript environment. Geth JSON RPC will be discussed in the *APIs, tools, and DApps* section.

Geth console

Geth can be started in console mode by running the following command:

```
$ geth console
```

This will also start the interactive JavaScript environment in which JavaScript commands can be run to interact with the Ethereum blockchain.

Geth attach

When a `geth` client is already running, the interactive JavaScript console can be invoked by attaching to that instance. This is possible by running the `geth attach` command.

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

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

```
$ geth attach
```

This command will produce output similar to the following:

```
Welcome to the Geth JavaScript console!  
instance: Geth/v1.9.9-stable-01744997/linux-amd64/go1.13.4  
coinbase: 0x07668e548be1e17f3dcfa2c4263a0f5f88aca402  
at block: 0 (Thu, 01 Jan 1970 00:00:00 UTC)  
datadir: /home/vagrant/.ethereum  
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0  
> |
```

Figure 12.4: Geth client

Once Geth is successfully attached with the running instance of the Ethereum client, it will display the command prompt, `>`, which provides an

interactive command-line interface to interact with the Ethereum client using JavaScript notations.

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

```
> personal.newAccount()  
Password:  
Repeat password:  
"0x76bcb051e3cedfcc9c7ff0b25a57383dacbf833b"
```



Note that readers will see a different address.

The list of accounts can also be displayed similarly:

```
> eth.accounts  
["0x07668e548be1e17f3dcfa2c4263a0f5f88aca402", "0xba94fb1f306e4c
```

Now that we understand how accounts are created using Geth, a question arises: when a new account is created, where does Geth store the related public and private key information?

The answer to this question is that when an account in Ethereum is created, it stores the generated public and private key pair on disk in a **keystore**. It is important to understand what this keystore contains and where it is created. We explain this in the next section.

Ethereum keystore

As we saw in the previous chapter, accounts are represented by private keys. The public and private key pair generated by the new account creation process is stored in key files located locally on the disk. These key files are stored in the `keystore` directory present in the relevant path according to the OS in use.

On the Linux OS, its location is as follows: `~/.ethereum/keystore`

The content of a `keystore` file is shown here as an example. Can you correlate some of the names with what we have already learned in *Chapter 3, Symmetric Cryptography*, and *Chapter 4, Public Key Cryptography*? It is a JSON file. In the following example, it has been formatted for better visibility:

```
{  
  "address": "ba94fb1f306e4d53587fcfdcd7eab8109a2e183c4",  
  "crypto":  
  {  
    "cipher": "aes-128-ctr",  
    "ciphertext":  
      "b2ab4f94f5f44ce98e61d99641cd28eb00fd794129be25beb8a5fae  
    "cipherparams":  
      {"iv": "a0fdf0a6d314a62ba6a370f438faa57"},  
    "kdf": "scrypt",  
    "kdfparams":  
      {"dklen": 32, "n": 262144, "p": 1, "r": 8},  
    "salt": "be3e99203c24ffcb71a6be2823fc7a211c8cc10d66bc6b448fef420f  
    "mac": "1b0a42d4bf7a8e96d308179e9714718e902727ead7041b97a646ef1c9  
    "id": "7e0772e0-965e-4a05-ad93-fc5d11245ba3",  
    "version": 3  
  }
```

The private key is stored in an encrypted format in the `keystore` file. It is generated when a new account is created using the password and private key. The `keystore` file is also referred to as a UTC file as the naming format of it starts with UTC with the date timestamp therein. A sample name of the `keystore` file is shown here:

```
UTC--2020-01-18T17-46-46.604174215Z--ba94fb1f306e4d53587fcfdcd7eab8109a2e183c4
```

On macOS, the `keystore` file is stored at the following location:

```
~/Library/Ethereum/keystore/
```





It is imperative to keep safe the associated passwords created at the time of creating the accounts and when the key files are produced.

As the `keystore` file is present on the disk, it is very important to keep it safe. It is recommended that it is backed up as well. If the `keystore` files are lost or overwritten or somehow corrupted, there is no way to recover them. This means that any ether associated with the private key will be irrecoverable, too.

Now we have described the elements of the password `keystore` file and what they represent. The key purpose of this file is to store the configuration, which, when provided with the account password, generated a decrypted account private key. This private key is then used to sign transactions.

- `Address` : This is the public address of the account that is used to identify the sender or receiver.
- `Crypto` : This field contains the cryptography parameters.
- `Cipher` : This is the cipher used to encrypt the private key. In the following diagram, **AES-128-CTR** indicates the Advanced Encryption Standard – 128 bit – in counter mode. Remember that we covered this in *Chapter 3, Symmetric Key Cryptography*.
- `Ciphertext` : This is the encrypted private key.
- `Cipherparams` : This represents the parameters required for the encryption algorithm, **AES-128-CTR**.
- `IV` : This is the 128-bit initialization vector for the encryption algorithm.
- `KDF` : This is the key derivation function. It is `scrypt` in this case.
- `KDFParams` : These are the parameters for the **key derivation function (KDF)**.
- `Dklen` : This is the derived key length. It is 32 in our example.
- `N` : This is the iteration count.
- `P` : This is the parallelization factor; the default value is 1.

- `R` : This is the block size of the underlying hash function. It is set to 8, which is the default setting.
- `Salt` : This is the random value of salt for the key derivation function, KDF.
- `Mac` : This is the Keccak-256 hash output obtained following concatenation of the second leftmost 16 bytes of the derived key together with the ciphertext.
- `ID` : This is a random identification number.
- `Version` : The version number of the file format, currently version 3.

Now we will explore how all these elements work together. This whole process can be visualized in the following diagram:

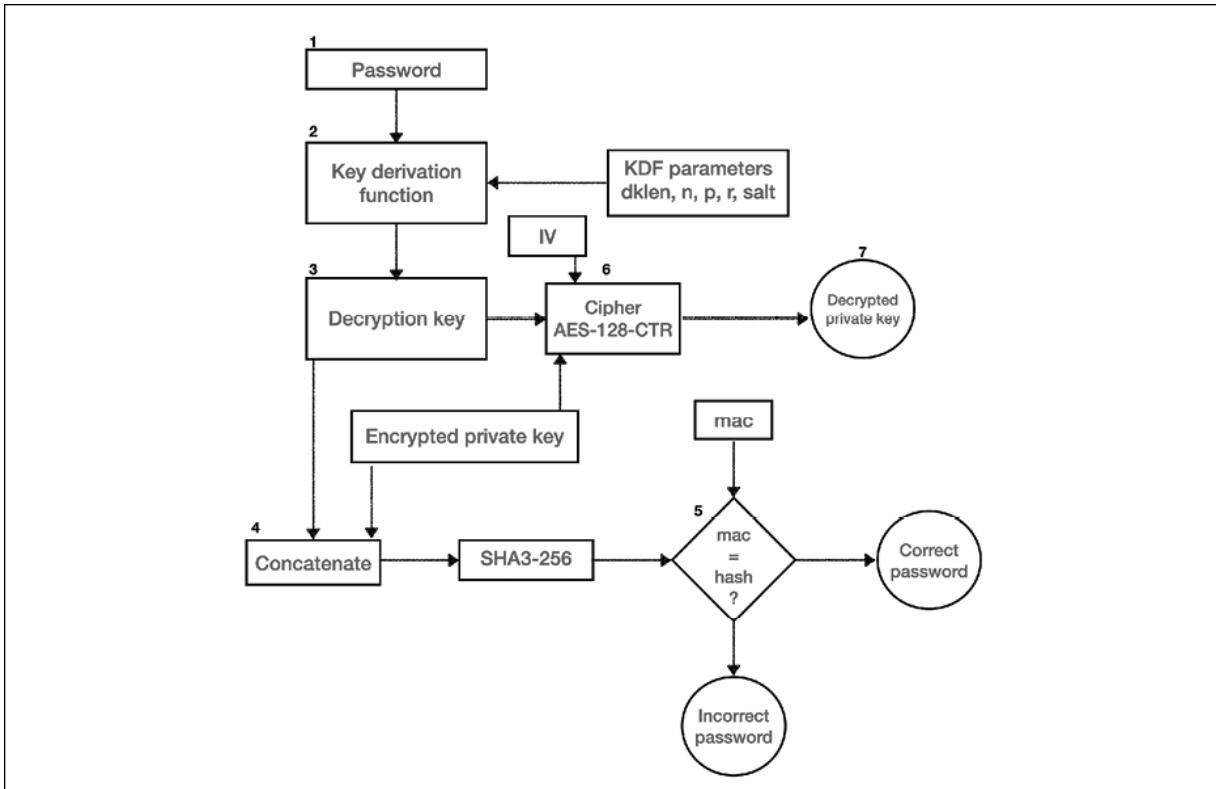


Figure 12.5: Private key decryption process

As shown in the preceding diagram, we can divide the process into different steps:

1. First, the password is fed into the **KDF**.
2. KDF takes several parameters, namely, **dklen**, **n**, **p**, **r**, and **salt**, and produces a decryption key.
3. The decryption key is concatenated with an **encrypted private key** (ciphertext). The decryption key is also fed into the cipher algorithm, **AES-128-CTR**.
4. The concatenated **decryption key** and ciphertext are hashed using the **SHA3-256** hash function.
5. **mac** is fed into the checking function where the hash produced from step 4 is compared with the **mac**. If both match, then the password is valid, otherwise the password is incorrect.
6. The cipher function, which is fed with the initialization vector (**IV**), encrypted private key (ciphertext), and decryption key, decrypts the encrypted private key.
7. The **decrypted private key** is produced.

This decrypted private key is then used to sign transactions on the Ethereum network.

With this, we have completed the introduction to Geth. Next, we will briefly touch on Eth and OpenEthereum before looking at MetaMask in detail. We'll start with installation.

Eth installation

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

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

OpenEthereum installation

OpenEthereum is another implementation of the Ethereum protocol. It has been written using the **Rust** programming language. The key aims behind

the development of OpenEthereum are high performance, a small footprint, modularization, and reliability.



It can be downloaded from Open Ethereum GitHub at
<https://github.com/openethereum/openethereum>.

OpenEthereum is the rebranded version of Parity Ethereum, which was a famous Ethereum client developed by Parity technologies. Parity Ethereum has been transitioned into a DAO-based ownership and maintainer model that is expected to provide a decentralized platform for collaboration, governance, and further development of the Parity Ethereum code base.



More information on this transition is available here:
<https://www.parity.io/parity-ethereum-openethereum-dao/>.

For all major operating systems, the latest release of the software is available here:

<https://github.com/openethereum/openethereum/releases/latest>.

OpenEthereum can be installed using the `brew` package manager on a macOS system:

```
$ brew tap paritytech/paritytech
$ brew install parity
```

If Parity is already installed, it can be upgraded using the following command.

```
$ brew upgrade parity
```

If you are on an OS other than macOS, the software for that OS can be downloaded from the preceding GitHub link.

Once installation has completed successfully, Parity can be run by using the following command:

```
$ parity
```

This command will produce output similar to the following:

```
2020-04-16 23:34:56 Starting Parity-Ethereum/v2.7.2-stable-d961010f61-20200705/x86_64-apple-darwin/rustc1.41.0
2020-04-16 23:34:56 Keys path /Users/drequinox/Library/Application Support/io.parity.ethereum/keys/ethereum
2020-04-16 23:34:56 DB path /Users/drequinox/Library/Application Support/io.parity.ethereum/chains/ethereum/db/9b6a3e69aec8c8d
2020-04-16 23:34:56 State DB configuration: fast
2020-04-16 23:34:56 Operating mode: active
2020-04-16 23:34:57 Configured for Ethereum using Ethash engine
2020-04-16 23:34:58 Updated conversion rate to Eth = US$172.19 (27654944 wei/gas)
2020-04-16 23:35:03 Public node URL: enode://227ca51c75cb263ef28973aa5b02c3a68cf4f0c5e3b5e0145c4148a19571f39cfecfde1ab691d25fd587bb6aa0248037219f625f6054c28ea60db5b4135a8b7@192.168.0.18:30303
2020-04-16 23:35:17 Syncing snapshot 0/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 2 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:22 Syncing snapshot 12/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 3 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:27 Syncing snapshot 38/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 5 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:32 Syncing snapshot 70/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 7 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:37 Syncing snapshot 99/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 7 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:42 Syncing snapshot 107/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 7 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:48 Syncing snapshot 115/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 11 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:54 Syncing snapshot 122/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 11 KIB sync RPC: 0 conn, 0 req/s, 0 µs
2020-04-16 23:35:58 Syncing snapshot 127/4515 #0 3/25 peers 920 bytes chain 3 MiB db 0 bytes queue 11 KIB sync RPC: 0 conn, 0 req/s, 0 µs
```

Figure 12.6: Parity startup

Creating accounts using the Parity command line

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

```
$ parity account new
Please note that password is NOT RECOVERABLE.
Type password:
Repeat password:
0x398aa52c557a6deed04d0e5dd486bc40dd374d00
```

Now we move to a different type of software used for interaction with the Ethereum blockchain: wallets.

Various wallets are available for Ethereum for desktop, mobile, and web platforms. A popular wallet that we will use for examples in the next chapter is named **MetaMask**, which is a tool of choice when it comes to development for Ethereum.

MetaMask

MetaMask runs as a plugin or add-on in the web browser. It is available for the Chrome, Firefox, Opera, and Brave browsers. The key idea behind the development of MetaMask is to provide an interface with the Ethereum blockchain. It allows efficient account management and connectivity to the Ethereum blockchain without running the Ethereum node software locally. MetaMask allows connectivity to the Ethereum blockchain through the infrastructure available at Infura (<https://infura.io>). This allows users to interact with the blockchain without having to host any node locally.

Installation

MetaMask is available for download at <https://metamask.io>. Here, we'll go over the installation process.

Browse to <https://metamask.io/>, where links to the relevant source are available to download the extension for your browser:

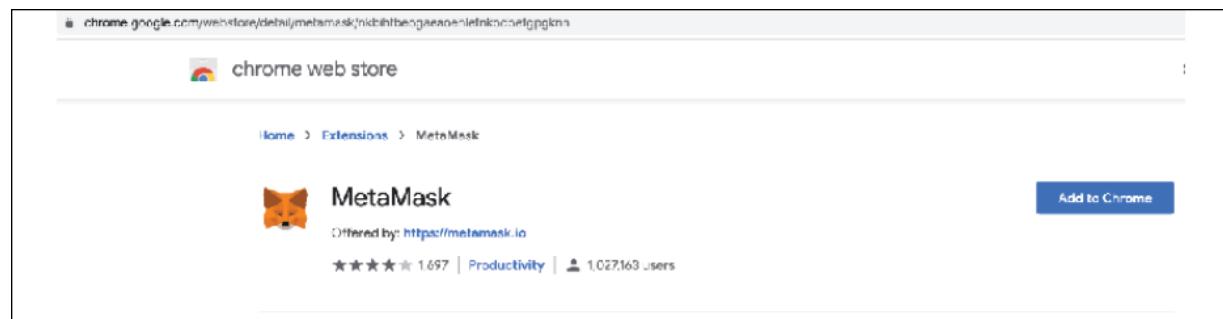


Figure 12.7: MetaMask download

The preceding image shows the MetaMask plugin on Chrome Web Store.

1. Click the **Add to Chrome** button and it will install the extension for the browser. It will install quickly, and if all goes well, you will see the following window:

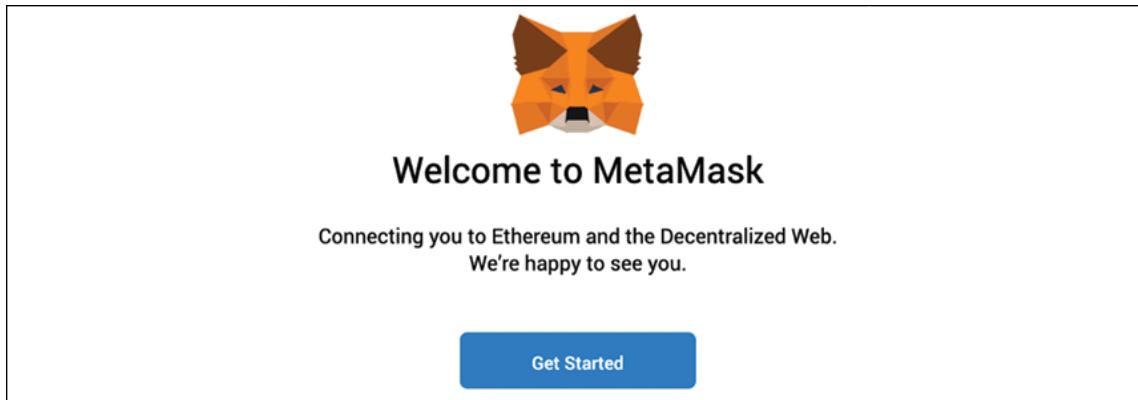


Figure 12.8: MetaMask welcome

2. Click on **Get Started**, and it will show two options, either to import an already existing wallet using the seed, or to create a new wallet. We will select **Create a Wallet** here:

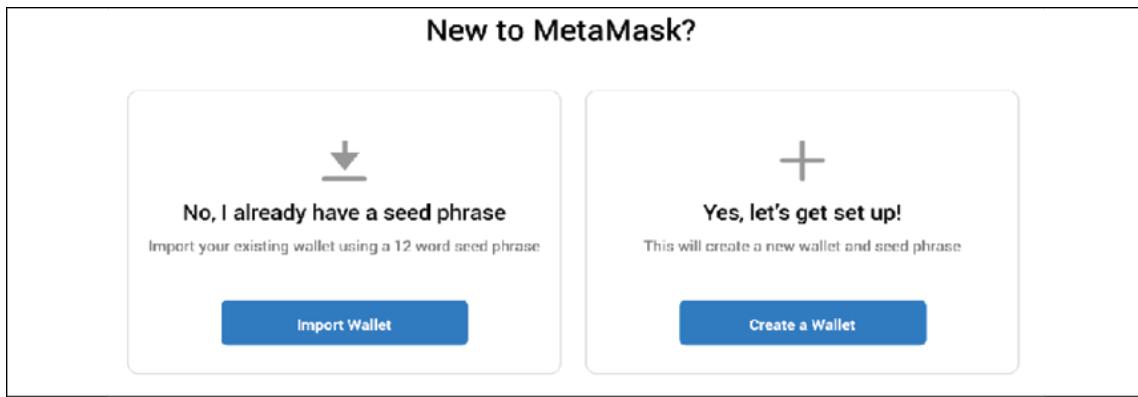


Figure 12.9: MetaMask wallet management

3. Next, create a **Password**:

Create Password

New Password (min 8 chars)

Confirm Password

I have read and agree to the [Terms of Use](#)

Create

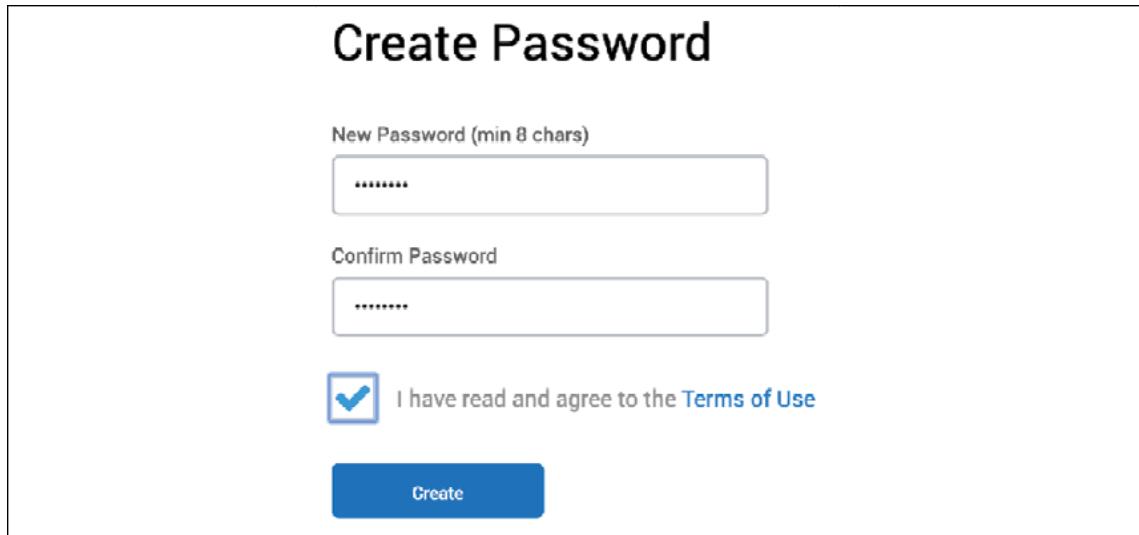


Figure 12.10: Create password

4. Optionally, set up a **Secret Backup Phrase**:

Secret Backup Phrase

Your secret backup phrase makes it easy to back up and restore your account.

WARNING: Never disclose your backup phrase. Anyone with this phrase can take your Ether forever.

buddy away super duty nephew
gym still tube peace year pupil
improve

Tips:

Store this phrase in a password manager like 1Password.

Write this phrase on a piece of paper and store in a secure location. If you want even more security, write it down on multiple pieces of paper and store each in 2 - 3 different locations.

Memorize this phrase.

[Download this Secret Backup Phrase and keep it stored safely on an external encrypted hard drive or storage medium.](#)

Remind me later **Next**

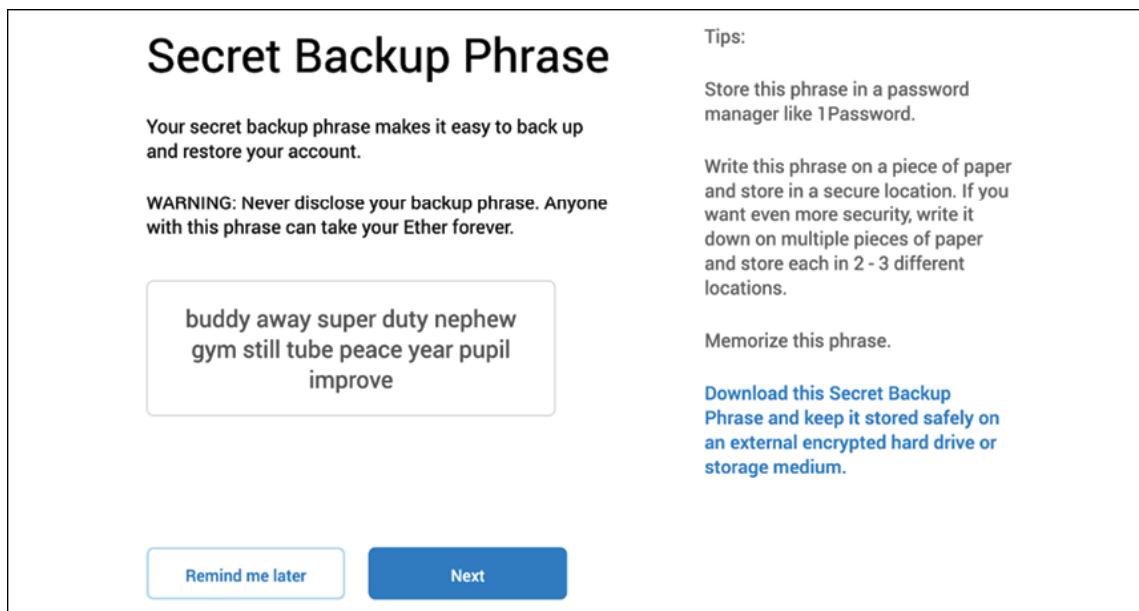


Figure 12.11: Secret backup phrase

5. Once everything has been completed, you will see the following message:

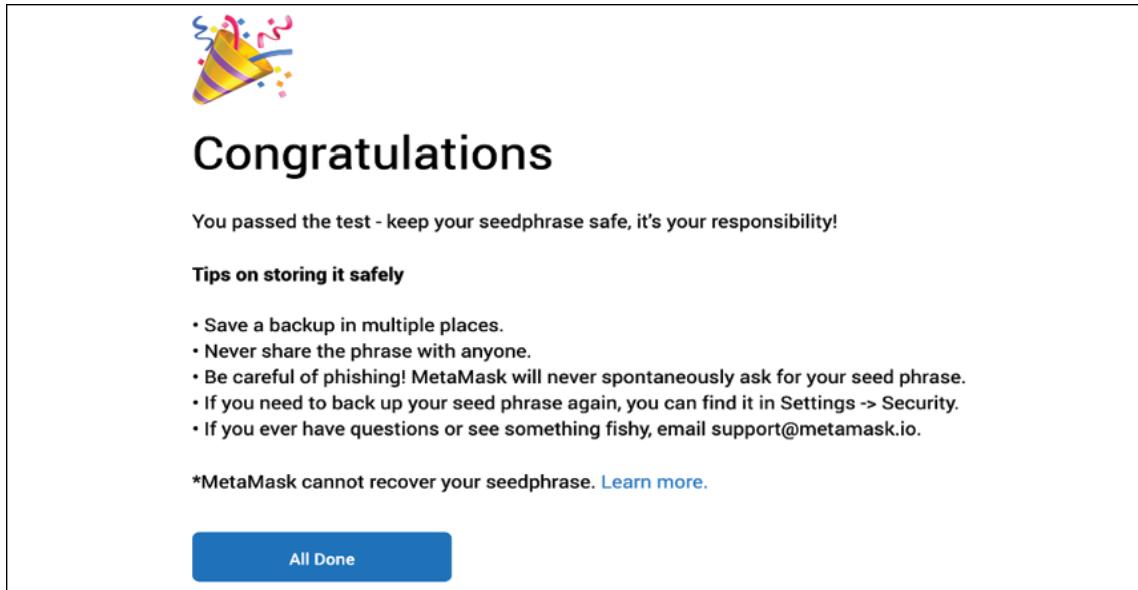


Figure 12.12: Congratulations in MetaMask

6. After clicking on the **All Done** button, the main MetaMask main view will open:

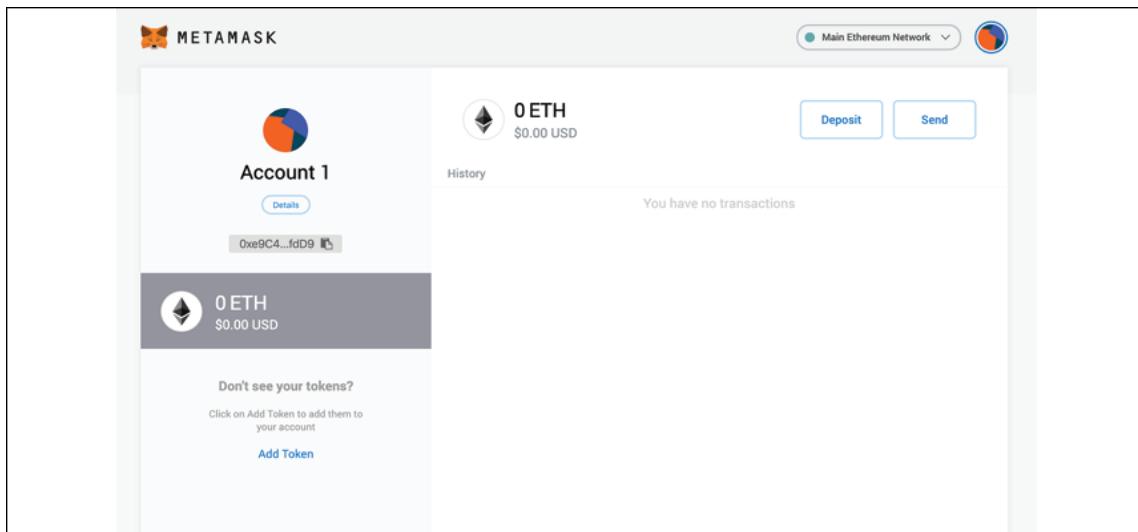


Figure 12.13: MetaMask main view

7. Also, in the top right-hand corner, you will see a small fox icon. Click that and you will see the following view:

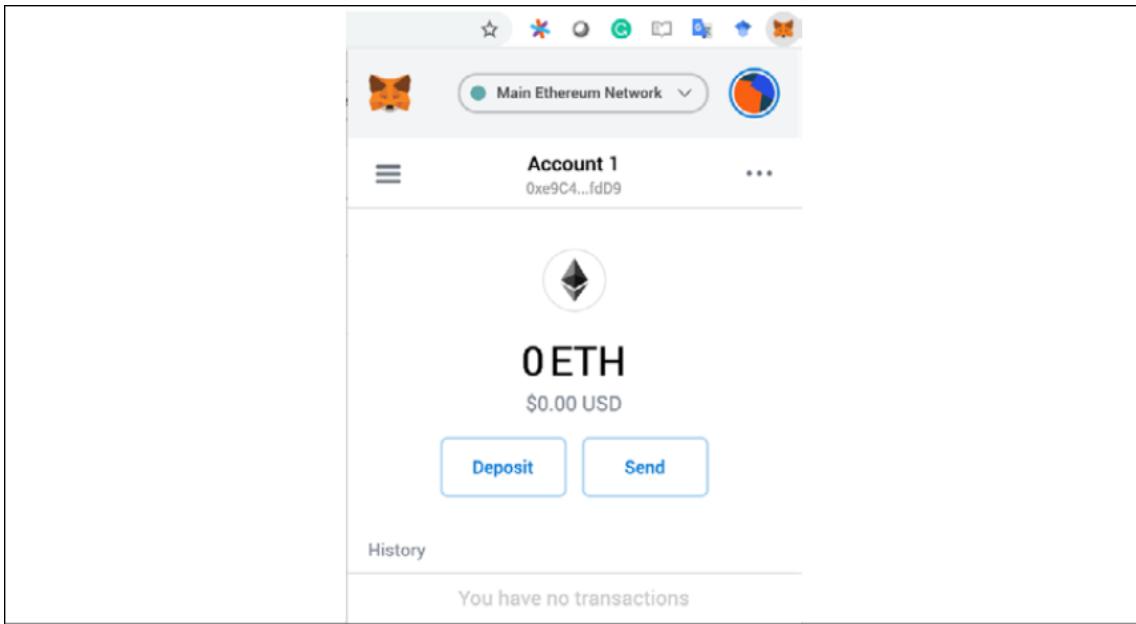


Figure 12.14: MetaMask view

Once installed, the Ethereum provider API will be available in the browser context, which can be used to interact with the blockchain. MetaMask injects a global API into websites at `window.ethereum`.

An example of the Ethereum API is shown here using the JavaScript console in Chrome:

```
> window.ethereum.chainId
< "0x2a"
> window.ethereum.isConnected()
< true
> window.ethereum.networkVersion
< "42"
> window.ethereum.autoRefreshOnNetworkChange
< true
    autoRefreshOnNetworkChange
    rpcEngine
    selectedAddress
    send
    sendAsync
    _events
    _eventsCount
    _maxListeners
    _metamask
```

Figure 12.15: Ethereum object

MetaMask can connect to various networks. By default, it connects to the Ethereum mainnet.

Other included networks include, but are not limited to, the **Ropsten** test network, the **Kovan** test network, the **Rinkeby** test network and the **Goerli** test network.

In addition, it can connect to various testnets and local nodes via `localhost 8545`, and to *Custom RPC*, which means that it can connect to any network as long as RPC connection information is available.

Now, let's see how account management works in MetaMask.

Creating and funding an account using MetaMask

Let's now create an account and fund it with some ETH, all using MetaMask. Note that we are connected to the Kovan test network. Open MetaMask and ensure that it is connected to the Kovan test network, as shown here:

1. Click on **Create Account**:

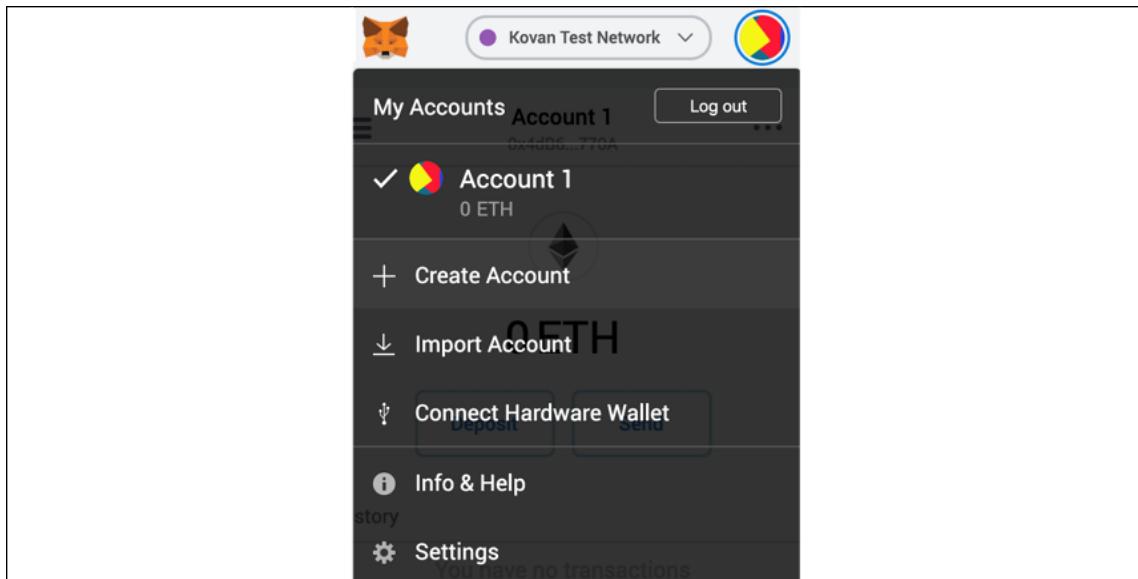


Figure 12.16: Creating an account

2. Enter the new account name and click on **Create**, which will immediately create a new account:

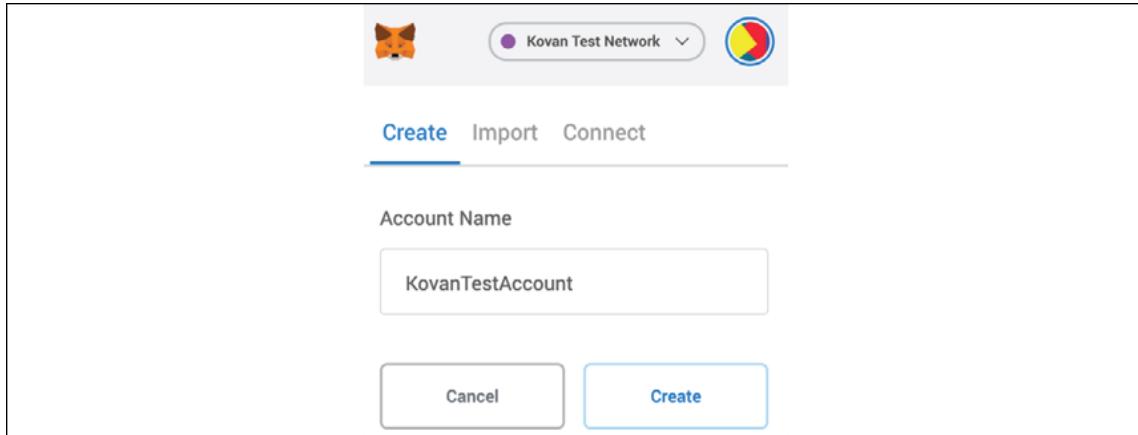


Figure 12.17: Creating a Kovan test account

Once created, we can fund the account with the following steps:

3. Copy the account address to the clipboard:

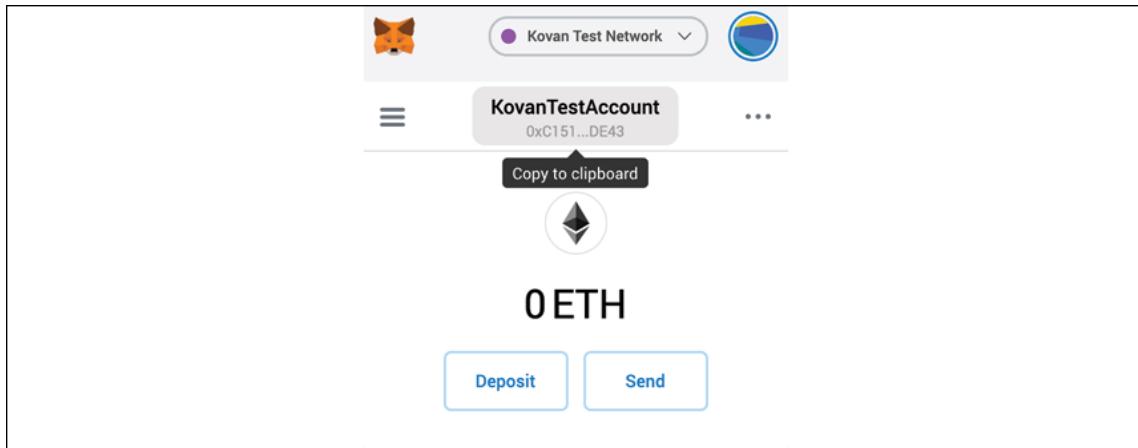


Figure 12.18: Kovan test account

4. Go to <https://gitter.im/kovan-testnet/faucet> and enter the **KovanTestAccount** address. You will receive some ETH in your account:



Figure 12.19: Funding using Kovan faucet

Now, notice in MetaMask that 3 ETH are available:

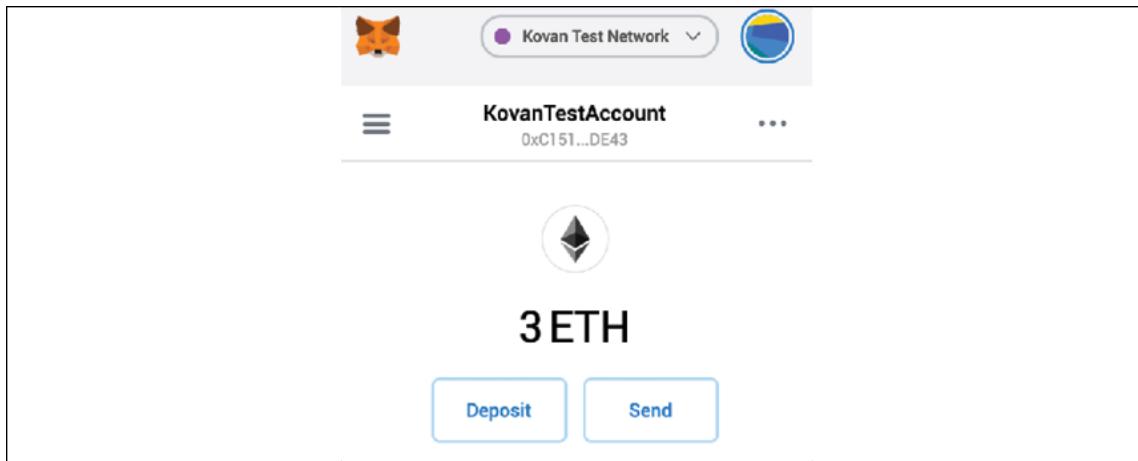


Figure 12.20: Kovan 3 ether

5. Now create another account and transfer some ETH from **KovanTestAccount** to the new account:

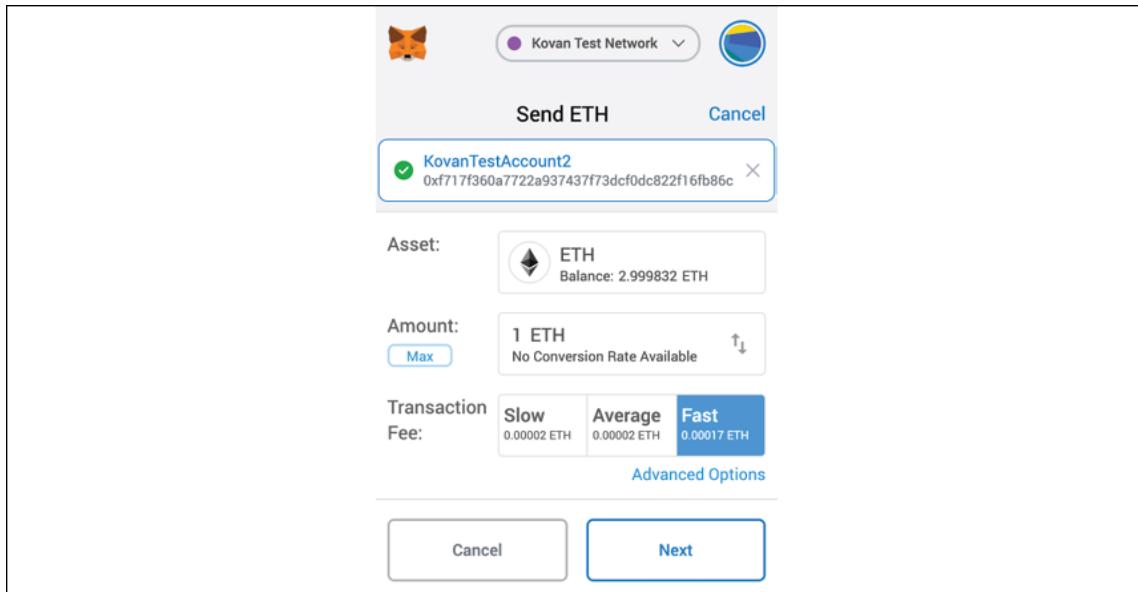


Figure 12.21: Sending ether

6. You will observe that there is an **Advanced Options** button. Click it to customize gas.

We can now see several options related to gas and transaction fees. We usually don't need to do that, but in order to control the transaction fee and transaction speed, these options are adjusted accordingly. You can select your transaction to be fast or slow. This depends on the amount of gas you are willing to pay. The higher the gas, the greater the chances are that miners will pick it up because of higher transaction fees. Remember that we discussed gas concepts earlier. Here, these concepts will become clearer.

Notice the gas price and gas limit in the following screenshot:

- The **Gas Price** is **8**, which is the amount of ether one individual is willing to pay for each unit of gas.
- The **Gas Limit** is **21,000**, which is the maximum amount of gas the transaction sender is willing to spend.

Now, can we calculate the gas using the preceding information?

Note that the gas price is in GWei, which is also called nano-ether and represents the ninth power of the ETH in fractions. This means that 0.000000001 ETH is, in fact, 1 GWei. Notice that there are 8 zeros after the decimal point.

Now, if the gas price is 8 and the gas limit is 21,000 GWei, we use the following simple formula of $(21000 * 8) * 0.000000001$, to get 0.000168, which is exactly what is shown in the following screenshot:

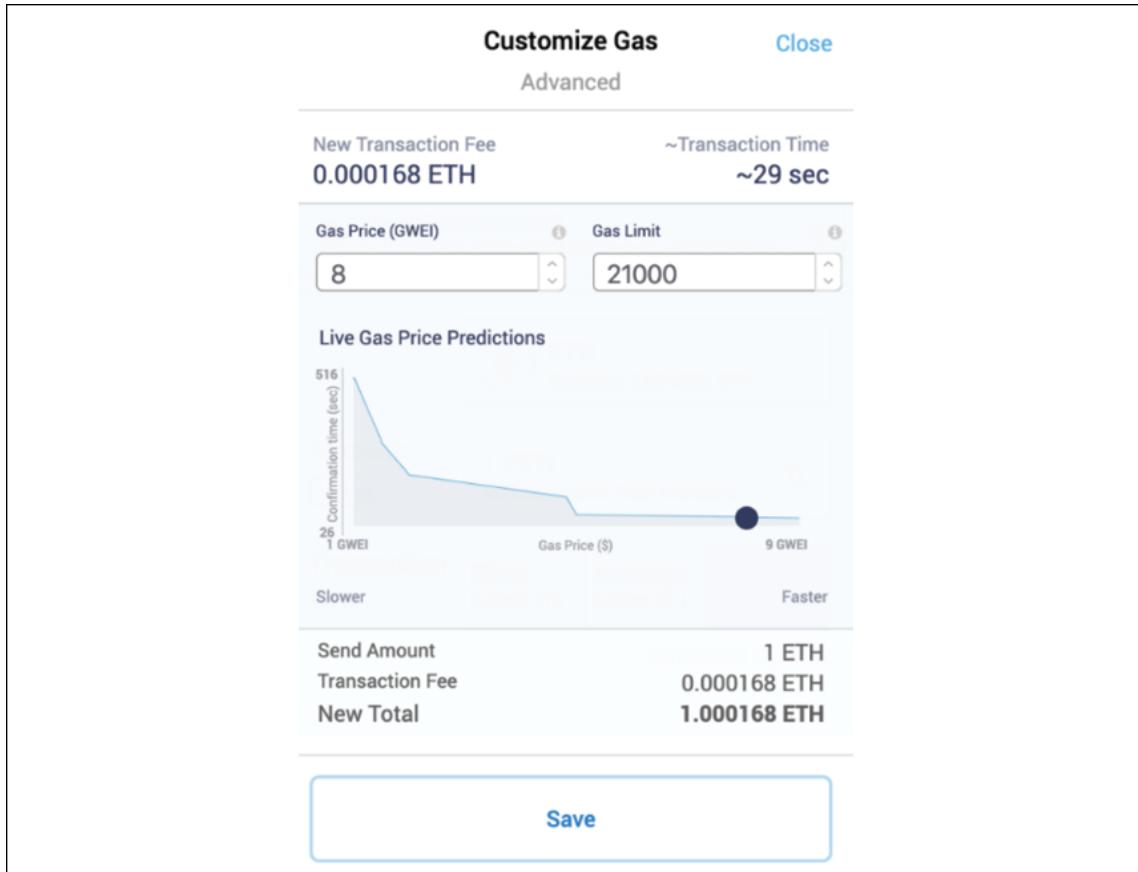


Figure 12.22: Customize Gas

7. Confirm the transaction:

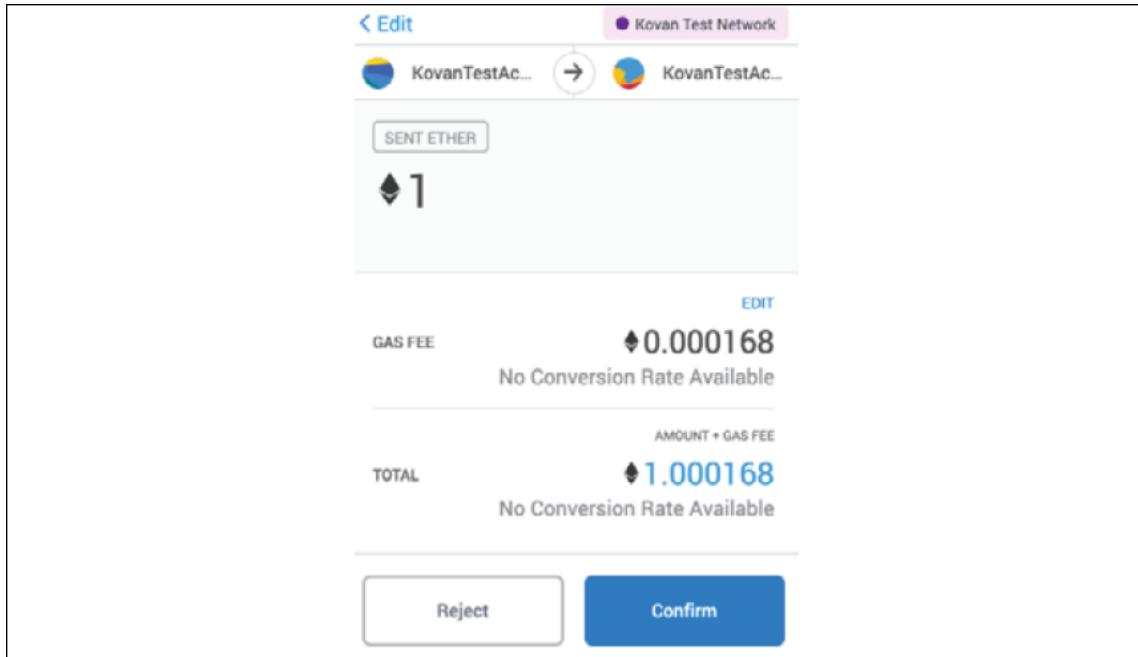


Figure 12.23: Confirming the transaction

Now, notice that the ether has been sent to the other account:

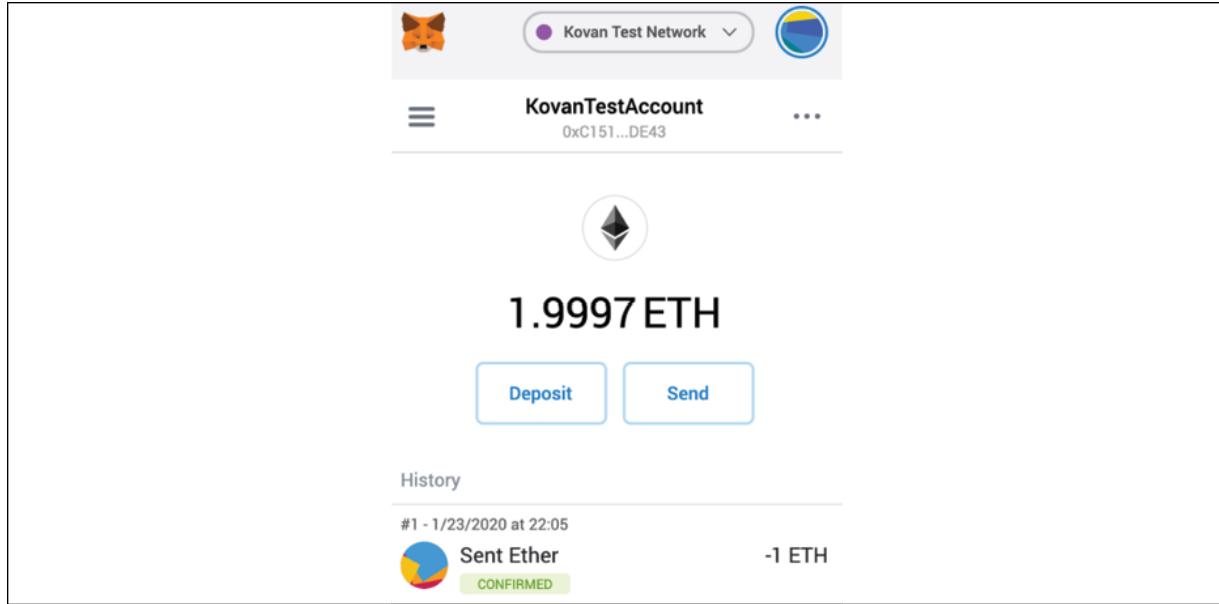


Figure 12.24: Ether sent in Kovan

Also note the detailed logs and information pertaining to the transaction:

Details

From: 0xC1516b10c74b96... > To: 0xf717f360A7722a937...

Transaction

Amount	1 ETH
Gas Limit (Units)	21000
Gas Used (Units)	21000
Gas Price (GWEI)	8
Total	1.000168 ETH

Activity Log

- Transaction created with a value of 1 ETH at 22:05 on 1/23/2020.
- Transaction submitted with gas fee of 168000 GWEI at 22:06 on 1/23/2020.
- Transaction confirmed at 22:06 on 1/23/2020.

Figure 12.25: Log of the transfer

This information can also be viewed in the block explorer:

1.9997 ETH

History

#1 - 1/23/2020 at 22:05

Sent Ether -1 ETH
CONFIRMED

Details

From: 0xC1516b10c74b96... > To: 0xf717f360A7722a937...

Transaction

Amount	1 ETH
Gas Limit (Units)	21000
Gas Used (Units)	21000
Gas Price (GWEI)	8

Figure 12.26: Block explorer in Kovan

The explorer can be opened by clicking on the blue-colored right arrow pointing at 45 degrees, as shown in the preceding diagram, or by following this link to the transaction summary:

<https://kovan.etherscan.io/tx/0x884cc20881017aa4d49afa7c31a8db6f2ade8911c2959100aab1d10e5fcfdf72>.

The screenshot shows a transaction details page from the Kovan Testnet. At the top, it says "[This is a Kovan Testnet transaction only]". Below that, the transaction hash is listed as 0x884cc20881017aa4d49afa7c31a8db6f2ade8911c2959100aab1d10e5fcfdf72. The status is marked as "Success". The block number is 16313779 with 36 block confirmations. The timestamp is 2 mins ago (Jan-23-2020 10:06:24 PM +UTC). The transaction originated from address 0xc1516b10c74b96e3a4f2bb66a8a67511af9de43 and was sent to address 0xf717f360a7722a937437f73dcf0dc822f16fb86c. The value transferred was 1 Ether (\$0.00). The transaction fee was 0.000168 Ether (\$0.000000). The gas limit was 21,000, and the gas used by the transaction was 21,000 (100%). The gas price was 0.000000008 Ether (8 Gwei). The nonce was 1, and the input data field contains 0x.

Figure 12.27: Kovan block explorer

With this, we have completed our introduction to MetaMask. We will now focus on account management using the Geth client. MetaMask and its use in development will be discussed at length in the next chapter, *Chapter 13, Ethereum Development Environment*.

We have now covered some popular Ethereum clients, wallets, and related usage concepts.

Next, we will introduce two of the most important elements of the Ethereum blockchain, nodes and miners. Miners perform the most important operation of the Ethereum blockchain, called mining.

Nodes and miners

The Ethereum network contains different nodes. Some nodes act only as wallets, some are light clients, and a few are full clients running the full blockchain. One of the most important types of nodes are mining nodes. We will see what constitutes mining in this section.



Mining is the process by which new blocks are selected via a consensus mechanism and added to the blockchain.

As a result of the mining operation, currency (ether) is awarded to the nodes that perform mining operations. These mining nodes are known as **miners**. Miners are paid in ether as an incentive for them to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

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

- It listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
- It determines stale or older blocks and includes them in the blockchain.
- It updates the account balance with the reward earned from successfully mining the block.
- Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.

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

The PoW algorithm is due to be replaced by the PoS algorithm with the release of **Serenity**. There is no set date for the release of Serenity, as this will be the final version of Ethereum. Considerable research work has been

carried out to build the PoS algorithm, which is suitable for the Ethereum network.



More information on PoS research work is available at
<https://ethresear.ch/t/initial-explorations-on-full-pos-proposal-mechanisms/925>.

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



More information about Casper can be found here:
<https://github.com/ethereum/research/tree/master/casper4>.

Miners play a vital role in reaching a consensus regarding the canonical state of the blockchain. The consensus mechanism that they contribute to is explained in the next section.

The consensus mechanism

The consensus mechanism in Ethereum is based on the **Greedy Heaviest Observed Subtree (GHOST)** protocol proposed initially by Zohar and Sompolinsky in December 2013.



Readers interested in this topic can find more information in the original paper at <http://eprint.iacr.org/2013/881.pdf>.

Ethereum uses a simpler version of this protocol, where the chain that has the most computational effort spent on it to build it is identified as the definite version. Another way of looking at it is to find the longest chain, as

the longest chain must have been built by consuming adequate mining efforts. The GHOST protocol was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to stale or orphaned blocks.

In GHOST, stale blocks, or ommers, are added in calculations to figure out the longest and heaviest chain of blocks.

The following diagram shows a quick comparison between the longest and heaviest chains:

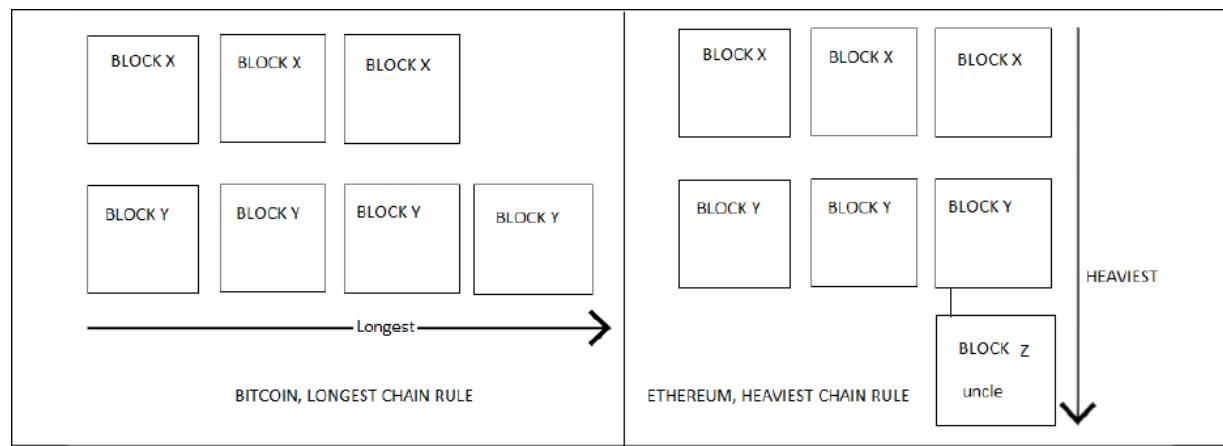


Figure 12.28: Longest versus the heaviest chain

The preceding diagram shows two rules of figuring out which blockchain is the canonical version of truth. In the case of Bitcoin, shown on the left-hand side in the diagram, the longest chain rule is applied, which means that the active chain (true chain) is the one that has the most amount of PoW done. In the case of Ethereum, the concept is similar from the point of view of the longest chain, but it also includes ommers, the *orphaned* blocks, which means that it also rewards those blocks that were competing with other blocks during mining to be selected and performed significant PoW, or were mined exactly at the same time as others but did not make it to the main chain. This makes the chain the *heaviest* instead of the *longest* because it also contains the *orphaned* blocks. This is shown on the right-hand side of the diagram.

As the blockchain progresses (more blocks are added to the blockchain) governed by the consensus mechanism, on occasion, the blockchain can split into two. This phenomenon is called **forking**.

Forks in the blockchain

A fork occurs when a blockchain splits into two. This can be intentional or non-intentional. Usually, as a result of a major protocol upgrade, a hard fork is created, while an unintentional fork can be created due to bugs in the software.

It can also be temporary as discussed previously; in other words, the longest and heaviest chain. This temporary fork occurs when a block is created almost at the same time and the chain splits into two, until it finds the longest or heaviest chain to achieve eventual consistency.

The release of **Homestead** involved major protocol upgrades, which resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum, known as **Frontier**, to the second version. The latest version is called **Byzantium**, which is the first part of the **Metropolis** release. This was released as a hard fork at block number 4,370,000.

An unintentional fork, which occurred on November 24, 2016, at 14:12:07 UTC, was due to a bug in Ethereum's Geth client journaling mechanism.

As a result, a network fork occurred at block number 2,686,351. This bug resulted in Geth failing to prevent empty account deletions in the case of the empty out-of-gas exception. This was not an issue in **Parity** (another popular Ethereum client). This means that from block number 2,686,351, the Ethereum blockchain is split into two, one running with the Parity clients and the other with Geth. This issue was resolved with the release of Geth version 1.5.3.

As a result of the DAO hack, which was discussed in *Chapter 10, Smart Contracts*, the Ethereum blockchain was also forked to recover from the attack.

To facilitate consensus, a PoW algorithm is used. In Ethereum, the algorithm used for this purpose is called **Ethash**, which will be covered in the next section.

Ethash

Ethash is the name of the PoW algorithm used in Ethereum. Originally, this was proposed as the **Dagger-Hashimoto algorithm**, but much has changed since the first implementation, and the PoW algorithm has now evolved into what's known as Ethash.

Similar to Bitcoin, the core idea behind mining is to find a nonce (a random arbitrary number), which, once concatenated with the block header and hashed, results in a number that is lower than the current network difficulty level. Initially, the difficulty was low when Ethereum was new, and even CPU and single GPU mining was profitable to a certain extent, but that is no longer the case. Now, only either pooled mining or large GPU mining farms are used for profitable mining purposes.

Ethash is a memory-hard algorithm, which makes it difficult to be implemented on specialized hardware. As in Bitcoin, ASICs have been developed, which have resulted in mining centralization over the years, but memory-hard PoW algorithms are one way of thwarting this threat, and Ethereum implements Ethash to discourage ASIC development for mining. Ethash is a **memory-hard** algorithm and developing ASICs with large and fast memories is not feasible. This algorithm requires subsets of a fixed resource called **Directed Acyclic Graph (DAG)** to be chosen, depending on the nonce and block headers.

DAG is a large, pseudo-randomly generated dataset. This graph is represented as a matrix in the DAG file created during the Ethereum mining process. The Ethash algorithm expects the DAG as a two-dimensional array of 32-bit unsigned integers.

Mining can only start when DAG is completely generated the first time a mining node starts. This DAG is used as a seed by the algorithm called

Ethash. According to current specifications, the epoch time is defined as 30,000 blocks, or roughly 6 days.

The Ethash algorithm requires a DAG file to work. A DAG file is generated every epoch, which is 30,000 blocks. DAG grows linearly as the chain size grows. Currently, the DAG size is around 3.5 GB (as of block 9325164) and epoch number 310.

The protocol works as follows:

1. First, the header from the previous block and a 32-bit random nonce is combined using Keccak-256.
2. This produces a 128-bit structure called `mix`.
3. `mix` determines which data is to be picked up from the DAG.
4. Once the data is fetched from the DAG, it is "mixed" with the `mix` to produce the next `mix`, which is then again used to fetch data from the DAG and subsequently mixed. This process is repeated 64 times.
5. Eventually, the 64th `mix` is run through a digest function to produce a 32-byte sequence.
6. This sequence is compared with the difficulty target. If it is less than the difficulty target, the nonce is valid, and the PoW is solved. As a result, the block is mined. If not, then the algorithm repeats with a new nonce.



More technical details at code level can be found here:
<https://github.com/ethereum/go-ethereum/blob/master/consensus/ethash/algorithm.go>

The current reward scheme is 2 ETH for successfully finding a valid nonce. In addition to receiving 2 ether, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (uncles) in the block. A maximum of two uncles are allowed per block and are rewarded with 7/8 of the normal block reward. In order to achieve a 12-second block time, block difficulty is adjusted at every block. The rewards are proportional to the miner's hash rate, which

means how fast a miner can hash. You can use an ether mining calculator to calculate what hash rate is required to generate profit.



One example of such a calculator is
<https://etherscan.io/ether-mining-calculator>.

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

In the next section, various methods of mining will be explored.

CPU mining

Even though not profitable on mainnet, CPU mining is still valuable on the test network or even a private network to experiment with mining and contract deployment. Private and test networks will be discussed with practical examples in the next chapter, *Chapter 13, Ethereum Development Environment*. A Geth example is provided here on how to start CPU mining. Geth can be started with a mine switch in order to initiate mining:

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

CPU mining can also be started using the Web3 Geth console. The Geth console can be started by issuing the following command:

```
$ geth attach
```

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

```
miner.start(4)  
true
```

Number `4` here represents the number of threads that will run for mining. It can be any number depending on the number of CPUs you have.

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

```
miner.stop()  
true
```

The preceding command will stop the miner. The command will return `True` if successful.

GPU mining

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

```
geth --rpc
```

Once `geth` is up and running, and the blockchain is fully downloaded, **Ethminer** can be run in order to start mining. `ethminer` is a standalone miner that can also be used in farm mode to contribute to mining pools:



You can download Ethminer from GitHub here:
<https://github.com/Genoil/cpp-ethereum/tree/117/releases>.

```
$ ethminer -G
```

Running with the `-G` switch assumes that the appropriate graphics card is installed and configured correctly.

If no appropriate graphics cards are found, `ethminer` will return an error, as shown in the following screenshot:

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

Figure 12.29: Error in the case that no appropriate GPUs can be found

GPU mining requires an AMD or NVIDIA graphics card and an applicable OpenCL SDK.



For an NVIDIA chipset, it can be downloaded from <https://developer.nvidia.com/cuda-downloads>. For AMD chipsets, it is available at <http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk>.

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

Benchmarking

`ethminer` can also be used to run benchmarking, as shown in the following screenshot. Two modes can be invoked for benchmarking. It can either be CPU or GPU. The commands are shown here:

CPU benchmarking:

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

GPU benchmarking:

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

The following screenshot is shown as an example of CPU mining benchmarking:

```
drequinox@drequinox-OP7010:~$ ethminer -M -C
  ◇ 22:43:30.560 ethminer #00004000...
Benchmarking on platform: 8-thread CPU
Preparing DAG...
  □ 22:43:30.561 miner0 Loading full DAG of seedhash: #00000000...
Warming up...
Trial 1... 0
Trial 2... DAG 22:43:38.310 miner0 Generating DAG file. Progress: 0 %
0
Trial 3... 0
Trial 4... DAG 22:43:45.336 miner0 Generating DAG file. Progress: 1 %
0
```

Figure 12.30: CPU benchmarking

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

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

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

Mining rigs

As difficulty increased over time in relation to mining ether, mining rigs with multiple GPUs started to be built by the miners. A mining rig usually contains around five GPU cards with all of them working in parallel for mining, thereby improving the chances of finding a valid nonce for mining.

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

- **Motherboard:** A specialized motherboard with multiple PCI-E x1 or x16 slots, for example, BIOSTAR Hi-Fi or ASRock H81, is required.
- **SSD hard drive:** An SSD hard drive is required. The SSD drive is recommended because of its much faster performance vis-à-vis the

analog equivalent. This will mainly be used to store the blockchain. It is recommended that you have roughly over 350 GB of free space on your hard disk.



You can always check chain data size here and adjust the disk space accordingly:

<https://etherscan.io/chartsync/chaindefault>.

- **GPU:** The GPU is the most critical component of the rig as it is the primary workhorse that will be used for mining. For example, it can be a Sapphire AMD Radeon R9 380 with 4 GB RAM. A page that maintains these benchmark metrics is available at <https://www.miningbenchmark.net>.
- **OS:** Linux Ubuntu's latest version is usually chosen as the OS for the rig because it is more reliable and gives better performance as compared to Windows. Also, it allows you to run a bare minimum OS required for mining, and essential operations as compared to heavy graphical interfaces that another OS may have. There is also another variant of Linux available, called **EthOS** (available at <http://ethosdistro.com/>) that is specially built for Ethereum mining and supports mining operations natively.
- **Mining software:** Finally, mining software such as Ethminer and Geth are installed. Additionally, some remote monitoring and administration software is also installed so that rigs can be monitored and managed remotely if required. It is also important to put proper air conditioning or cooling mechanisms in place, since running multiple GPUs can generate a large amount of heat. This also necessitates the need to use an appropriate monitoring software that can alert users if there are any problems with the hardware, for example, if the GPUs are overheating.
- **Power supply units (PSUs):** In a mining rig, there are multiple GPUs running in parallel. Therefore, there is a need for a constant powerful supply of electricity and it is necessary to use PSUs that can take the load and provide enough power to the GPUs in order for them to operate. Usually, 1,000 watts of power is required to be produced by PSUs. An excellent comparison of PSUs is available at

<https://www.thegeekpub.com/11488/best-power-supply-mining-cryptocurrency/>:

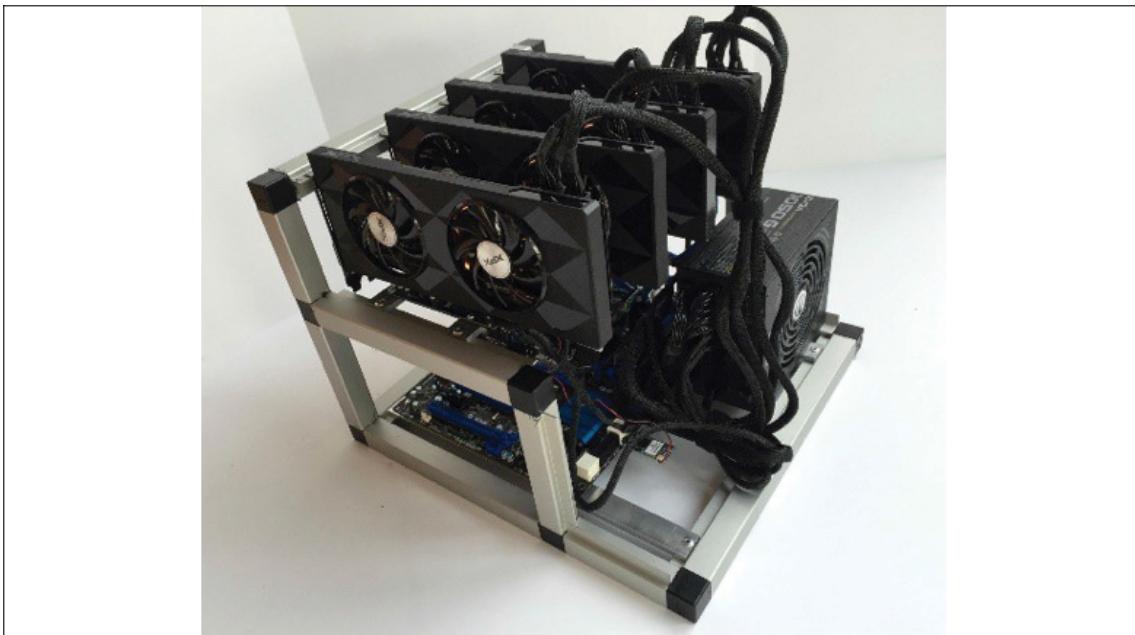


Figure 12.31: A mining rig for Ethereum

Mining pools

Many online mining pools offer Ethereum mining. Ethminer can be used to connect to a mining pool using the following command. Each pool publishes its instructions, but generally, the process of connecting to a pool is similar. An example from <http://ethereumpool.co> is shown here:

```
$ ethminer -C -F http://ethereumpool.co/?miner=0.1@0x024a20cc5fe
```

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

```
miner 23:50:53.046 ethminer Getting work package . . .
```

This output means that the `ethminer` is running and receiving `work` from the pool to process.

In the next section, we introduce specialized mining hardware called **ASIC**. ASIC is an abbreviation of **Application Specific Integrated Circuit**, which is an integrated circuit (chip) specially developed for a particular use. In this case, the particular use is mining, for which ASICs are currently the fastest method.

ASICs

Even though Ethereum was originally considered ASIC-resistant, there are now ASICs being built for ether mining that perform significantly better than GPUs. ASIC resistance in Ethereum comes from the fact that the Ethash consensus algorithm (introduced earlier in this chapter) requires a large amount of memory to store the DAG. Since maintaining such a large memory of usually around 4 GB is quite difficult on ASICs, this feature offers some resistance against ASICs.

However, there are some companies that are building Ethereum ASICs, but these are quite difficult to acquire as they are in very high demand. A transition from the existing Ethash consensus algorithm to ProgPoW (<https://github.com/ifdefelse/ProgPOW>) has also been agreed among developers, which is also slowing down ASIC hardware adoption. However, the difficulty is increasing on the Ethereum network due to the existence of these ASIC devices. Eventually, a move to a PoS system in Ethereum 2.0 will permanently address this issue.

So far, we have covered various clients, installation, relevant commands, and mining. Next, we will introduce some basic concepts related to Ethereum APIs, development, and protocols.

APIs, tools, and DApps

The Web3 JavaScript API provides an interface to the Ethereum blockchain via JavaScript. It provides an object called `web3`, which contains other objects that expose different methods to support interaction with the blockchain. This API covers methods related to administration of the blockchain, debugging, account-related operations, supporting protocol methods for Whisper, as well as storage and other network-related operations.

This API will be discussed in detail in the next chapter, *Chapter 13, Ethereum Development Environment*, where we will see how to interact with the Ethereum blockchain with it.

Applications (DApps and DAOs) developed on Ethereum

There are various implementations of DAOs and smart contracts in Ethereum, most notably, the DAO, which was recently misused due to a weakness in the code and required a hard fork for funds to be recovered that had been syphoned out by the attackers. The DAO was created to serve as a decentralized platform to collect and distribute investments.

Augur is another DApp that has been implemented on Ethereum, which is a decentralized prediction market.



Many other decentralized applications are listed on
<https://www.stateofthedapps.com/>

Tools

Various frameworks and tools have been developed to support decentralized application development, such as **Truffle**, **MetaMask**, **Ganache**, and many more. We will talk about these in *Chapter 14, Development Tools and Frameworks*.

Earlier in this chapter, we looked at different methods to connect to the blockchain, in other words, the Geth console and Geth attach. Now, we will demonstrate the use of another method that is commonly used—the Geth JSON RPC API.

Geth JSON RPC API

JSON RPC is a remote procedure call mechanism that makes use of JSON data format to encode its calls. In simpler terms, it is an RPC encoded in JSON. **JSON** stands for **JavaScript Object Notation**. It is a lightweight and easy-to-understand text format used for data transmission and storage. A remote procedure call is a distributed systems concept. It is a mechanism used to invoke a procedure in a different computer. It appears as if a local procedure call is being made because there is a requirement to write code to handle remote interactions.



Further details on RPC and JSON can be found here:
<https://www.jsonrpc.org>.

For this chapter, it is sufficient to know that the JSON-RPC API is used in Ethereum extensively to allow users and DApps to interact with the blockchain.

There are a number of methods available to interact with the blockchain. One is to use the Geth console, which makes use of the Web3 API to query the blockchain. The Web3 API makes use of the JSON-RPC API. Another method is to make JSON-RPC calls directly, without using the Web3 API. In this method, direct RPC calls can be made to the Geth client over HTTP. By default, Geth RPC listens at TCP port 8545.

Now, we will look at some examples involving utilization of the JSON RPC API. We will use a common utility curl (<https://curl.haxx.se>) for this purpose.

Examples

For these examples to work, first, the Geth client needs to be started up with appropriate flags. If there is an existing session of Geth running with other parameters, stop that instance and run the `geth` command as shown here, which will start Geth with the RPC available. The user can also control which APIs are exposed:

```
$ geth --rpc --rpccapi "eth,net,web3,personal"
```

In this command, Geth is started up with `--rpc` and `--rpccapi` flags, along with a list of APIs that are exposed:

- The `rpc` flag enables the HTTP-RPC server.
- The `rpccapi` flag is used to define which API's are made available over the HTTP-RPC interface. This includes a number of APIs such as `eth`, `net`, `web3`, and `personal`.

For each of the following examples, run the `curl` command in the terminal, as shown in each of the following examples.

List accounts

The list of accounts can be obtained by issuing the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --
```

This will display the following JSON output, which lists all the Ethereum accounts owned by the client:

```
{"jsonrpc":"2.0","id":64,"result":["0x07668e548be1e17f3dcfa2c426
```

Check if the network is up

We can query if the network is up by using the command shown here:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --
```

This will display the output as shown here with the result `true`, indicating that the network is up:

```
{"jsonrpc":"2.0","id":64,"result":true}
```

Providing the Geth client version

We can find the Geth client version using this command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --
```

This will display the `geth` client version:

```
{"jsonrpc":"2.0","id":64,"result":"Geth/v1.9.9-stable-01744997/1
```

Synching information

To check the latest synchronization status, we can use the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --
```

This will display the data pertaining to the synchronization status or return `false`:

```
{"jsonrpc":"2.0","id":64,"result":{"currentBlock":"0x1d202","hiç
```

Finding the Coinbase address

The Coinbase address can be queried using:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --
```

This will display the output shown here, indicating the client `coinbase` address:

```
{"jsonrpc":"2.0","id":64,"result":"0x07668e548be1e17f3dcfa2c4263}
```

These are just a few examples of extremely rich APIs that are available in Ethereum's `geth` client.



More information and official documents regarding Geth RPC APIs are available at the following link: <https://eth.wiki/json-rpc/API>.

In this section, we have covered Geth JSON RPC APIs and seen some examples on how to query the blockchain via the RPC interface. In the next section, we will introduce a number of supporting protocols that are part of the complete decentralized ecosystem based on the Ethereum blockchain.

Supporting protocols

Various supporting protocols are available to assist the complete decentralized ecosystem. This includes the Whisper and Swarm protocols. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized in order to achieve a complete decentralized ecosystem. This includes decentralized storage and decentralized messaging.

Whisper, which is being developed for Ethereum, is a decentralized messaging protocol, whereas **Swarm** is a decentralized storage protocol.

Both of these technologies provide the basis for a fully decentralized web, and are described in the following sections.

Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network. In essence, Whisper is a communication protocol that DApps use to communicate with each other. The data and routing of messages are encrypted within Whisper communications. Whisper makes use of the **DEVp2p** wire protocol for exchanging messages between nodes on the network. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required. Whisper is also designed to provide a communication layer that cannot be traced and provides *dark communication* between parties. Blockchain can be used for communication, but that is expensive, and a consensus is not really required for messages exchanged between nodes. Therefore, Whisper can be used as a protocol that allows censor-resistant communication. Whisper messages are ephemeral and have an associated **time to live (TTL)**. Whisper is already available with Geth and can be enabled using the `-shh` option while running the Geth Ethereum client.



Official Whisper documentation is available at
<https://eth.wiki/concepts/whisper/whisper>.

Swarm

Swarm has been developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to-peer storage network. Files in this network are addressed by the hash of their content. This is in contrast to the traditional centralized services, where storage is available at a central location only. It has been developed as a native base layer service for the Ethereum Web3 stack. Swarm is integrated with DEVp2p, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a **Distributed Denial of Service (DDOS)**-resistant and fault-tolerant

distributed storage layer for Ethereum Web 3.0. Similar to `shh` in Whisper, Swarm has a protocol called `bzz`, which is used by each Swarm node to perform various Swarm protocol operations.



Official Swarm documentation is available at: <https://swarm-guide.readthedocs.io/en/latest/>.

The following diagram gives a high-level overview of how Swarm and Whisper fit together and work with the Ethereum blockchain:

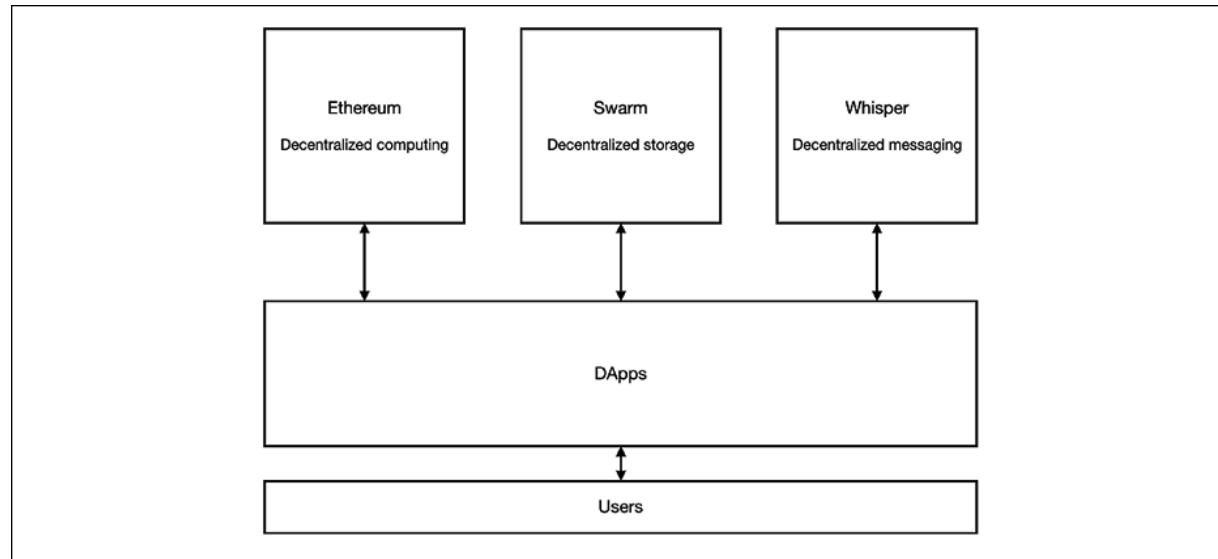


Figure 12.32: Blockchain, Whisper, and Swarm

With the development of Whisper and Swarm, a complete decentralized ecosystem emerges, where Ethereum serves as a decentralized computer, Whisper as a decentralized means of communication, and Swarm as a decentralized means of storage. In simpler words, Ethereum or more precisely, EVM, provides compute services, Whisper handles messaging and Swarm provides storage.

If you recall, in *Chapter 2, Decentralization*, we mentioned that decentralization of the whole ecosystem is highly desirable as opposed to decentralization of just the core computation element. The development of

Whisper for decentralized communication and Swarm for decentralized storage is a step toward decentralization of the entire blockchain ecosystem.

Now, we will briefly introduce the programming languages that are being used in the development of smart contracts on the Ethereum blockchain. Also, we will cover the underlying instruction set that underpins the operation of EVM.

Programming languages

Code for smart contracts in Ethereum is written in high-level languages such as Serpent, **Low-level Lisp-like Language (LLL)**, Solidity, or Vyper, and is converted into the bytecode that the EVM understands for it to be executed.

Solidity is one of the high-level languages that has been developed for Ethereum. It uses JavaScript-like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called **solc**.



Official Solidity documentation is available at
<http://solidity.readthedocs.io/en/latest/>.

LLL is another language that is used to write smart contract code.

Serpent is a Python-like, high-level language that can be used to write smart contracts for Ethereum.

Vyper is a newer language that has been developed from scratch to achieve a secure, simple, and auditable language.



More information regarding Vyper is available at
<https://github.com/ethereum/vyper>.

LLL and Serpent are no longer supported by the community and their usage has almost vanished. The most commonly used language is Solidity, which we will discuss at length in this chapter.

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

```
pragma solidity ^0.6.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

This program is converted into bytecode, as shown in the following subsection. Details on how to compile Solidity code with examples will be provided in *Chapter 14, Development Tools and Frameworks*.

Runtime bytecode

The runtime bytecode is what executes on the EVM. The smart contract code (`contract Test1`) from the previous section, is translated into binary runtime and opcodes, as follows:

Binary of the runtime (Raw hex codes):

```
6080604052348015600f57600080fd5b506004361060285760003560e01c8063
```

Opcodes:

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALU
```

Opcodes

There are many different **opcodes** that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. A list of opcodes, their meanings, and usages is available in the extra online content repository for this book here:

https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

Summary

This chapter started with some general concepts, including blocks, block structure, gas, and messages, which were introduced and discussed in detail. The later sections of the chapter introduced the practical installation and management of Ethereum wallets and clients. The two most commonly-used clients, Geth and OpenEthereum, were discussed. We also explored an introduction of programming languages for programming smart contracts in Ethereum.

Supporting protocols and topics related to challenges faced by Ethereum were also presented. Ethereum is under continuous development, and new improvements are being made by a dedicated community of developers on a regular basis. Ethereum improvement proposals, available at <https://github.com/ethereum/EIPs>, are also an indication of the magnitude of research and the keen interest exhibited by the community in this technology.

In the next chapter, we will explore Ethereum smart contract development, along with the relevant tools and frameworks.

13

Ethereum Development Environment

This chapter introduces the Ethereum development environment. Several examples will be presented in this chapter to complement the theoretical concepts provided in the earlier chapters. This chapter will mainly cover the setup of the development environment and how to use relevant tools to create and deploy smart contracts using the Ethereum blockchain.

The first task is to set up an Ethereum development environment. This involves setting up a private net and installing tools that aid the development of a decentralization application on Ethereum. We will also look at **testnets**, which can play a vital role in testing the smart contracts before deploying on the mainnet. Ethereum has several testnets. A common testnet is called Ropsten and is used by developers or users as a test platform to test smart contracts and other blockchain-related proposals. Then, we will look at the private net option in Ethereum, which allows the creation of an independent private network. This private network can be used as a shared distributed ledger between participating entities and for the development and testing of smart contracts. While there are other clients available for Ethereum, Geth is the leading client for Ethereum and the standard tool of choice, and as such, this chapter uses Geth for the examples.

There are various ways to perform development for Ethereum blockchain. We will see some of the most popular options in this chapter. The topics that we will cover in this chapter are listed here:

- Test networks
- Private network components

- Starting up the private network
- Mining on the private network
- Remix IDE
- MetaMask
- Using MetaMask and Remix IDE to deploy a smart contract

Overview

There are multiple ways to develop smart contracts on Ethereum. A usual and sensible approach is to develop and test Ethereum smart contracts either in a local private net or a simulated environment like Ganache. Then it can be deployed on a public testnet. After all the relevant tests are successful on a public testnet, the contracts can then be deployed to the public mainnet. There are, however, variations in this process. Many developers opt to only develop and test contracts on a local simulated environment and then deploy on to the public mainnet or their private/enterprise production blockchain networks. Developing first on a simulated environment and then deploying directly to a public network can lead to faster time to production, as setting up private networks may take longer compared to setting a local development environment with a blockchain simulator. We will explore all these approaches in *Chapter 14, Development Tools and Frameworks* and *Chapter 15, Introducing Web3*.

There are new tools and frameworks available, like **Truffle** and **Ganache**, which make development and testing for Ethereum easier. We will look into these tools in more depth in *Chapter 14, Development Tools and Frameworks*, but first, we will use a manual approach whereby we develop a smart contract and deploy it manually via the command line to the private network. This will allow us to see what actually happens in the background. Also, we will cover the use of MetaMask, as it has become the tool of choice for developers.

Frameworks and tools make development easier, but hide most of the finer "under the hood" details that are useful for beginners to understand to build

a strong foundation of knowledge. Therefore, first we will demonstrate development using the native tools available in Ethereum, and once we have understood all the foundational knowledge, we can start using development frameworks like Truffle, which indeed makes development and testing very easy.

Let us start by connecting to a test network.

Test networks

The Ethereum Go client (<https://geth.ethereum.org>), Geth, can be connected to the test network using the following command:

```
$ geth --testnet
```

This command will connect to the Ropsten network, which is a pre-configured **proof of work (PoW)** test network.



Geth installation was described in *Chapter 12, Further Ethereum*. Please refer back to this chapter for more detail.

A sample output is shown in the following screenshot. The screenshot shows the type of the network chosen and various other pieces of information regarding the blockchain download:

```

+ - geth --testnet
INFO [06-30|19:43:31.395] Maximum peer count          ETH=50 LES=0 total=50
INFO [06-30|19:43:31.423] Starting peer-to-peer node   instance=Geth/v1.9.10-stable/darwin-amd64/go1.13.6
INFO [06-30|19:43:31.424] Allocated trie memory caches   clean=256.00MiB dirty=256.00MiB
INFO [06-30|19:43:31.424] Allocated cache and file handles   database=/Users/drequinonx/Library/Ethereum/testnet/geth/chaindata cache=612.00MiB handles=5120
INFO [06-30|19:43:34.503] Opened ancient database        nodes=355 size=50.67KiB time=1.288303ms gcnodes=0 gcsizo=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [06-30|19:43:34.524] Persisted trie from memory database   config="(ChainID: 3 Homestead: 0 DAO: <nil> DAOSupport: true EIP150: 0 EIP155: 10 EIP158: 10 Byzantium: 1780000 Constantinople: 4230000 Petersburg: 4939394 Istanbul: 6485846, Muij Glacier: 7117117, Engine: ethash)"
INFO [06-30|19:43:34.524] Initialised chain configuration   dir=/Users/drequinonx/Library/Ethereum/testnet/geth/ethash count=3
INFO [06-30|19:43:34.525] Disk storage enabled for ethash DAGs   versions="[64 63]" network=3 dbversion=7
INFO [06-30|19:43:34.525] Initialising Ethereum protocol   number=89816 hash=1921f5..1bd680 td=8567527775346 age=3y7mo2w
INFO [06-30|19:43:34.529] Loaded most recent local header   number=89816 hash=1921f5..1bd680 td=8567527775346 age=3y7mo2w
INFO [06-30|19:43:34.529] Loaded most recent local full block   number=89816 hash=1921f5..1bd680 td=8567527775346 age=3y7mo2w
INFO [06-30|19:43:34.529] Loaded most recent local fast block   transactions=0 dropped=0
INFO [06-30|19:43:34.532] Loaded local transaction journal   transactions=0 accounts=0
INFO [06-30|19:43:34.533] Regenerated local transaction journal
WARN [06-30|19:43:34.533] Switch sync mode from fast sync to full sync
INFO [06-30|19:43:34.610] New local node record   seqm9 id=36466bb1d703662 ip=127.0.0.1 udp=30303 tcp=30303
INFO [06-30|19:43:34.612] Started P2P networking   self=enode://2aaeed8e005cc0db9cbad34fb3d931de87d8fe3345dfb279196273812cc384617dd21ff79e1e9bd2671d32a
F2293b123812cc4984d425b1f6@ec23fe#ea@050127.8.0.1:30303
INFO [06-30|19:43:34.619] IPC endpoint opened   url=/Users/drequinonx/Library/Ethereum/testnet/geth.ipc
INFO [06-30|19:43:37.081] New local node record   seqm10 id=36466bb1d703662 ip=82.27.41 udp=30303 tcp=30303
INFO [06-30|19:43:37.773] Mapped network port   nrtnmrcv=0 extprt=30303 intprt=30303 interface="UPNP Tfnv2-IP1"
INFO [06-30|19:43:38.573] Mapped network port   proto=udp extprt=30303 intprt=30303 interface="UPNP IGv2-IP1"
INFO [06-30|19:45:14.657] Block synchronisation started
INFO [06-30|19:45:20.080] Imported new chain segment   blocks=2 txs=38 mgas=3.957 elapsed=460.697ms mgasps=8.590 number=80818 hash=d0317a..f163f9 age=3y7mo2w dirty=142.16KiB

```

Figure 13.1: The output of the geth command connecting to Ethereum testnet Ropsten

A blockchain explorer for testnet is located at <https://ropsten.etherscan.io> and can be used to trace transactions and blocks on the Ethereum test network.

There are other test networks available too, such as **Rinkeby** and **Görl** (also referred to as **Goerli**). Geth can be issued with a command-line flag to connect to the desired network:

```
--testnet Ropsten network: pre-configured proof-of-work test net
--rinkeby Rinkeby network: pre-configured proof-of-authority test net
--goerli Görli network: pre-configured proof-of-authority test net
```

Now let us experiment with building a private network and then we will see how a contract can be deployed on this network using Mist and command-line tools.

Components of a private network

A private net allows the creation of an entirely new blockchain usually on a local network. This is different from testnet or mainnet in the sense that it

uses its own genesis block and network ID. In order to create a private net, three components are needed:

- Network ID
- The genesis file
- Data directory to store blockchain data

Even though the data directory does not strictly need to be mentioned, if there is more than one blockchain already active on the system, then the data directory should be specified so that a separate directory is used for the new blockchain.

On the mainnet, the Geth Ethereum client is capable of discovering **boot nodes** by default as they are hardcoded in the Geth client, and connects automatically, but on a private network, Geth needs to be configured by specifying appropriate flags and configurations in order for it to discover, or be discoverable by, other peers. We will see how this is achieved shortly.

In addition to the previously mentioned three components, it is desirable that you disable node discovery so that other nodes on the internet cannot discover your private network and so that it is secure. This can be achieved by running Geth with the flag `--nodiscover`, which disables the peer discovery mechanism. If other networks happen to have the same genesis file and network ID, they may connect to your private net, which can result in security issues. The chance of having the same network ID and genesis block is very low, but, nevertheless, disabling node discovery is a good practice, and is recommended. Another network or node connecting to your private network can result in security breaches, information leakage, and other undesirable security incidents. However, note that private networks are usually run within the enterprise environments and are protected by usual enterprise security practices such as firewalls.

Disabling peer discovery also allows us to define a list of static peers that we have on our network. This gives us the additional ability to control who can join our private network.

Just before we move onto creating our private network, let's get some theoretical aspects covered that are related to discovery and understand what actually happens when we disable node discovery. For this, first we'll see how a node normally discovers other nodes on an Ethereum network and what protocols are involved to do so. The suite of protocols that is responsible for node discovery and communication between Ethereum nodes is called RLPx.



Note that RLPx is named after RLP, the serialization protocol introduced in *Chapter 11, Ethereum 101*. However, it is not related to the serialization protocol RLP, and the name is not an acronym.

Ethereum node discovery is based on the **Kademlia protocol**. Kademlia is a UDP-based **distributed hash table (DHT)** protocol for distributed peer-to-peer networks.

The Ethereum network consists of four elements or layers: **Discovery**, **RLPx**, **DevP2P** and **application level sub-protocols**:

- The **Discovery** protocol is responsible for discovering other nodes on the network by using node discovery mechanism based on Kademlia protocol's routing algorithm. This protocol works by using UDP. In Ethereum there are two discovery protocols, named DiscV4 and DiscV5. DiscV4 is currently production ready and implemented in Ethereum whereas DiscV5 is in development.



Specification for DiscV4 can be found here:

https://github.com/ethereum/devp2p/blob/master/_discv4.md.

Specification for DiscV5 can be found here:

https://github.com/ethereum/devp2p/blob/master/_discv5/_discv5.md.

For discovery, a new Ethereum node joining the network makes use of hardcoded bootstrap nodes, which provide an initial entry point into the network, from where which further discovery processes then start.

This list of bootstrap nodes can be found here in the `bootnodes.go` file:

<https://github.com/ethereum/go-ethereum/blob/490b380a04437d7eb780635e941fc8fa017413e7/params/bootnodes.go#L23>

- **RLPx** is a TCP-based transport protocol responsible for enabling secure communication between Ethereum nodes. It achieves this by using an asymmetric encryption mechanism called **Elliptic Curve Integrated Encryption Scheme (ECIES)** for handshaking and key exchange.



Handshaking is a term used in computing to refer to a mechanism of exchanging initial information (signals, data, or messages) between different devices on a network to establish a connection for communication as per the protocol in use.

More information on ECIES and the RLPx specification can be found here:

<https://github.com/ethereum/devp2p/blob/master/rlpx.md>.

- **DEVP2P** (also called the wire protocol) is responsible for negotiating an application session between two nodes that have been discovered and have established a secure channel using RLPx. This is where the `HELLO` message is sent between nodes to provide each other with details of the version of the DevP2P protocol: the client name, supported application sub-protocols, and the port numbers the nodes are listening on. `PING` and `PONG` messages are used to check the availability of the nodes and `DISCONNECT` is sent if a response from a node is not received.
- Finally, the fourth element of the Ethereum network stack is where different **Ethereum sub-protocols** exist. After discovering and establishing a secure transport channel and negotiating an application session, the nodes exchange messages using so-called "capability protocols" or application sub-protocols. This includes **Eth** (versions 62, 63, and 64), **Light Ethereum Sub-protocol (LES)**, **Whisper**, and **Swarm**. These capability protocols are responsible for different

application-level communications: for example, Eth is responsible for block synchronization. It makes use of several protocol messages such as `Status`, `Transactions`, `GetBlockHeaders`, and `GetBlockBodies` messages to exchange blockchain information between nodes. Note that Eth is also referred to as "Ethereum wire protocol."



More detail on the Eth capability protocol can be found here:
<https://github.com/ethereum/devp2p/blob/master/caps/eth.md>

All four of these elements can be visualized in the following diagram:

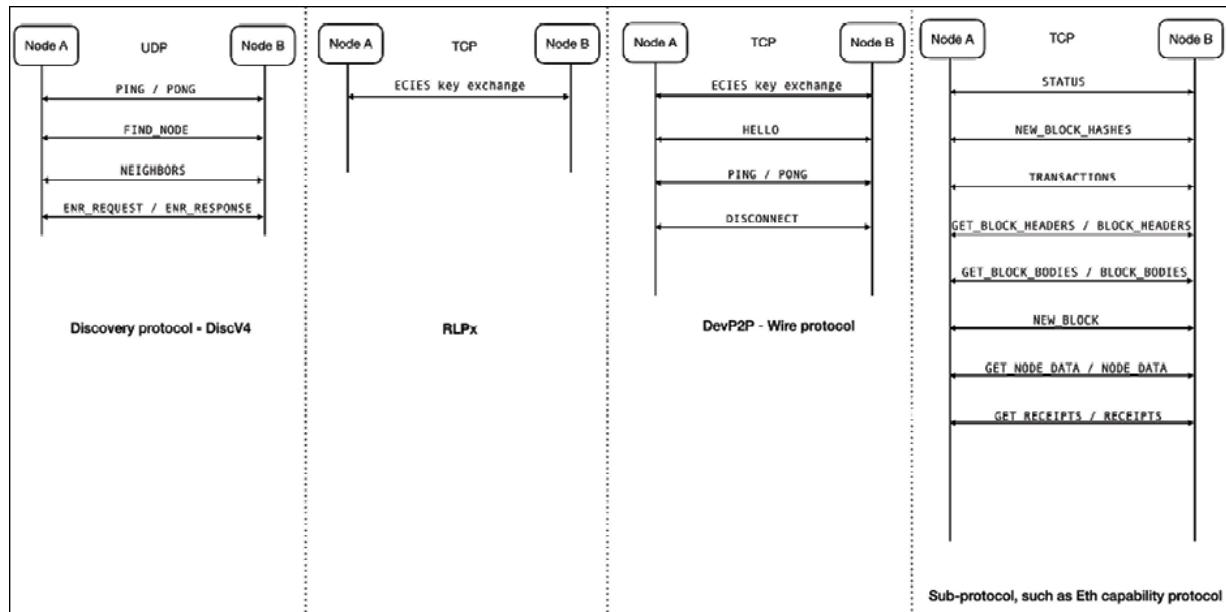


Figure 13.2: Ethereum node discovery, RLPx, DevP2P, and capability protocols

So here in our example, when we use the `--nodiscover` flag, it disables the peer discovery mechanism described previously. We can then add nodes manually to the network by maintaining a static list of peers.

Now, after this introduction to understand some underlying theory, all the parameters required for creating a private network will be discussed in detail.

Network ID

The network ID can be any positive number except any number that is already in use by another Ethereum network. For example, `1` and `3` are in use by Ethereum mainnet and testnet (Ropsten), respectively.



A list of Ethereum networks is maintained here <https://chainid.network> and network IDs chosen already should not be used to avoid conflict.

Network ID 786 has been chosen for the example private network discussed later in this section.

The genesis file

The genesis file contains the necessary fields required for a custom genesis block. This is the first block in the network and does not point to any previous block. The Ethereum protocol performs checking in order to ensure that no other node on the internet can participate in the consensus mechanism unless they have the same genesis block. Chain ID is usually used as an identification of the network.

A custom genesis file that will be used later in the example is shown here:

```
        "homesteadBlock": 0,
        "eip150Block": 0,
        "eip155Block": 0,
        "eip158Block": 0
    }
}
```

This file is saved as a text file with the JSON extension; for example, `privategenesis.json`. Optionally, Ether can be pre-allocated by specifying the beneficiary's addresses and the amount of Wei, but this is usually not necessary, as being on the private network, Ether can be mined very quickly.

In order to pre-allocate an account with Ether, a section can be added to the genesis file, as shown here:

```
"alloc": {
    "0xcf61d213faa9acadb0d110e1397caf20445c58f": {
        "balance": "100000"
    }
}
```

Now let's see what each of these parameters means.

- `nonce` : This is a 64-bit hash used to prove that PoW has been sufficiently completed. This works in combination with the `mixhash` parameter.
- `timestamp` : This is the Unix timestamp of the block. This is used to verify the sequence of the blocks and for difficulty adjustment. For example, if blocks are being generated too quickly, that difficulty goes higher.
- `parentHash` : Being the genesis (first) block, this is always zero as there is no parent block.
- `extraData` : This parameter allows a 32-bit arbitrary value to be saved with the block.
- `gasLimit` : This is the limit on the expenditure of gas per block.

- `difficulty` : This parameter is used to determine the mining target. It represents the difficulty level of the hash required to prove the PoW.
- `mixhash` : This is a 256-bit hash that works in combination with `nonce` to prove that a sufficient amount of computational resources has been spent in order to complete the PoW requirements.
- `coinbase` : This is the 160-bit address where the mining reward is sent as a result of successful mining.
- `alloc` : This parameter contains the list of pre-allocated wallets. The long hex digit is the account to which the balance is allocated.
- `config` : This section contains various configuration information defining the chain ID and blockchain hard fork block numbers. This parameter is not required to be used in private networks.

Data directory

This is the directory where the blockchain data for the private Ethereum network will be saved. For example, in the following example, it is

`~/etherprivate/`.

In the Geth client, a number of parameters are specified in order to run the Ethereum node, fine-tune the configuration, and launch the private network. These flags are listed in the following section.

If connectivity to only specific nodes is required, which is usually the case in private networks, then a list of static nodes is provided as a JSON file. This configuration file is read by the Geth client at the time of starting up and the Geth client only connects to the peers that are present in the configuration file.

Flags and their meaning

The following are the flags used with the Geth client:

- `--nodiscover` : This flag disables the peer discovery mechanism, which allows you to add specific peers manually.

- `--maxpeers` : This flag is used to specify the number of peers allowed to be connected to the private net. If it is set to `0`, then no one will be able to connect, which might be desirable in a few scenarios, such as private testing.
- `--rpc` : This is used to enable the RPC interface in Geth.
- `--rpccapi` : This flag takes a list of APIs to be allowed as a parameter. For example, `eth, web3` will enable the Eth and Web3 interface over RPC.
- `--rpcport` : This sets up the TCP RPC port; for example, `9999`.
- `--rpccorsdomain` : This flag specifies the URL that is allowed to connect to the private Geth node and perform RPC operations.



`CORS` in `--rpccorsdomain` means cross-origin resource sharing.

- `--port` : This specifies the TCP port that will be used to listen to the incoming connections from other peers.
- `--identity` : This flag is a string that specifies the name of a private node.

Static nodes

In the case of private networks, usually the connectivity is limited to a specific set of peers. In order to configure this list, the node IDs of these nodes are added to a configuration file called `static-nodes.json`. This file is usually placed in the data directory of the Geth (Ethereum client) executable. This directory is also where the `chaindata` (database) and `keystore` files are saved. By default, the data directory is located at `<user's home directory>/Library/Ethereum` but can be configured by using the flag `--datadir`.

The filename should be `static-nodes.json` under the data directory. This is valuable in a private network because this way the network is limited to only known nodes. An example of the `static-nodes.json` file is shown as follows:

```
[  
"enode:// 44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff12  
]
```

Here, `xxx` is the IP address and `TCP_PORT` can be any valid and available TCP port on the system. The long hex string is the node ID.

As we now understand the various aspects and components required to set up a private network, including the genesis file and other relevant configuration files, let's now move on to actually setting up our own private network using Ethereum.

Starting up the private network

First, let's start up our private network and prepare it for use. The first step is to create a directory named `etherprivate` under the home directory of the user.

```
$ mkdir ~/etherprivate
```

This command will create the directory. Once the directory is created, place the `privategenesis.json` file shown earlier in *The genesis file* section. At this point, stored under the home directory of the user, we have a directory named `~/etherprivate`, which contains the genesis file called `privategenesis.json`. We are ready to start our network. The initial command to start the private network is shown as follows:

```
$ geth init ~/etherprivate/privategenesis.json --datadir ~/ether
```

This will produce an output similar to what is shown in the following screenshot:

```

Maximum peer count          ETH=50 LES=0 total=50
Allocated cache and file handles
persisted trie from memory database
Successfully wrote genesis state
Allocated cache and file handles
persisted trie from memory database
Successfully wrote genesis state
database=/Users/drequinox/etherprivate/geth/chaindata cache=16.00MiB handles=16
nodes=0 size=0.00B time=478.681µs gcnodes=0 gcsiz=0.00B gctime=0s livenodes=1 livesize=0.00B
database=chaindata hash=6650a0..b5c158
database=/Users/drequinox/etherprivate/geth/lightchaindata cache=16.00MiB handles=16
nodes=0 size=0.00B time=14.268µs gcnodes=0 gcsiz=0.00B gctime=0s livenodes=1 livesize=0.00B
database=lightchaindata hash=6650a0..b5c158

```

Figure 13.3: Private network initialization

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

```
$ ./geth --datadir ~/etherprivate/ --networkid 786 --rpc --rpccap
```

This will produce the following output:

```

INFO [07-09|19:28:16.641] Maximum peer count          ETH=50 LES=0 total=50
INFO [07-09|19:28:16.670] Starting peer-to-peer node
INFO [07-09|19:28:16.670] Allocated trie memory caches
INFO [07-09|19:28:16.670] Allocated cache and file handles
INFO [07-09|19:28:16.728] Opened ancient database
INFO [07-09|19:28:16.733] Initialised chain configuration
0 EIP158: 0 Byzantium: <nil> Constantinople: <nil> Petersburg: <nil> Istanbul: <nil>, Muir Glacier: <nil>, Engine: unknown"
INFO [07-09|19:28:16.734] Disk storage enabled for ethash caches
INFO [07-09|19:28:16.734] Disk storage enabled for ethash DAGs
INFO [07-09|19:28:16.735] Initialising Ethereum protocol
INFO [07-09|19:28:16.741] Loaded most recent local header
INFO [07-09|19:28:16.741] Loaded most recent local full block
INFO [07-09|19:28:16.741] Loaded most recent local fast block
INFO [07-09|19:28:16.742] Loaded local transaction journal
INFO [07-09|19:28:16.743] Regenerated local transaction journal
WARN [07-09|19:28:16.743] Switch sync mode from fast sync to full sync
INFO [07-09|19:28:16.865] New local node record
INFO [07-09|19:28:16.866] Started P2P networking
5874edd6aca9aabf8371ed2e99ff9f0ee98da3b17d6adf4b5c6eed7fbaa8@127.0.0.1:30303
INFO [07-09|19:28:16.869] IPC endpoint opened
INFO [07-09|19:28:16.869] HTTP endpoint opened
INFO [07-09|19:28:18.470] New local node record
INFO [07-09|19:28:19.391] Mapped network port
INFO [07-09|19:28:19.593] Mapped network port
ETH=50 LES=0 total=50
instance=Geth/v1.9.10-stable/darwin-amd64/go1.13.6
clean=256.00MiB dirty=256.00MiB
database=/Users/drequinox/etherprivate/geth/chaindata cache=512.00MiB
database=/Users/drequinox/etherprivate/geth/lightchaindata/ancient config={"ChainID": 786, "Homestead": 0, "DAO": <nil>, "DAOSupport": false, "EIP158": 0, "Byzantium": <nil>, "Constantinople": <nil>, "Petersburg": <nil>, "Istanbul": <nil>, "MuirGlacier": <nil>, "Engine": "unknown"}, "Ethash": {"count": 3, "dir": "/Users/drequinox/etherprivate/geth/ethash"}, "dbversion": 7, "versions": "[64 63]", "network": 786, "dbversion": 7, "number": 5906, "hash": "b43696..004e81", "td": 2388807985, "age": "1mo2w3d", "number": 5906, "hash": "b43696..004e81", "td": 2388807985, "age": "1mo2w3d", "number": 5906, "hash": "b43696..004e81", "td": 2388807985, "age": "1mo2w3d", "transactions": 0, "dropped": 0, "transactions": 0, "accounts": 0
seq=53 id=4d5806a5e05ac77c ip=127.0.0.1 udp=30303 tcp=30303
self=node://96578daac05df98e5895d83a86558ae7503bcc4b0d85075ae218
url=/Users/drequinox/etherprivate/geth.ipc
url=http://127.0.0.1:8545 cors=*, vhosts=localhost
seq=54 id=4d5806a5e05ac77c ip=82.2.27.41 udp=30303 tcp=30303
proto=tcp extport=30303 intport=30303 interface="UPNP IGDrv1-IP1"
proto=udp extport=30303 intport=30303 interface="UPNP IGDrv1-IP1"

```

Figure 13.4: Starting Geth for a private network

An important part of the output to note is the following log lines:

```

INFO [07-09|19:49:01.504] IPC endpoint opened
url=/Users/drequinox/etherprivate/geth.ipc
INFO [07-09|19:49:01.504] HTTP endpoint opened
url=http://127.0.0.1:8545
INFO [07-09|19:49:05.734] Etherbase automatically configured
address=0xc9Bf76271b9E42E4bF7E1888e0F52351bDb65811

```

These lines show the information about the **Inter-Process Communications (IPC)** endpoint, HTTP endpoint, and Etherbase

(coinbase) account information. This information is useful for the examples provided later in this section.



IPC is a mechanism to allow communication between processes running on the computer locally. More information on this can be found here:

https://en.wikipedia.org/wiki/Inter-process_communication

Now Geth can be attached via IPC to the running Geth client on a private network using the following command:

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

As shown in *Figure 13.5*, this will open the interactive JavaScript console for the running private net session:

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.10-stable/darwin-amd64/go1.13.6
coinbase: 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
at block: 5906 (Sat, 23 May 2020 13:49:16 BST)
datadir: /Users/drequinox/etherprivate
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0
```

Figure 13.5: Starting Geth to attach with private net 786

Note that the time of the last block creation is also shown, seen in the preceding output:

```
at block: 5906 (Sat, 23 May 2020 13:49:16 BST).
```

You may notice that a warning message appears when Geth starts up:

```
WARNING: No etherbase set and no accounts found as default.
```

This message appears because there are no accounts currently available in the new test network and no account is set as etherbase to receive mining rewards. This issue can be addressed by creating a new account and setting

that account as etherbase. This will also be required when mining is carried out on the test network.

This is shown in the following commands. Note that these commands are entered in the Geth JavaScript console, which is shown in the preceding screenshot. The following command creates a new account. In this context, the account will be created on the private network ID 786 :

```
> personal.newAccount ("Password123")
"0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811"
```

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

```
> miner.setEtherbase(personal.listAccounts[0])
true
```

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

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

In this section, we created a private network with a custom genesis file. Also, we created a new account in our private network. Now we are ready to start mining on our private network.

Mining on the private network

Now that we have started up our private network, mining can start by simply issuing the following command. This command takes one parameter: the number of threads. In the following example, two threads

will be allocated to the mining process by specifying `2` as an argument to the `start` function:

```
> miner.start()
true
```



Here we can also provide an integer parameter. For example, if we provide `1`, it will only use one CPU core for mining, which helps with performance issues, if using all CPU resources is reducing system performance. An example command of using only one CPU is `miner.start(1)`. On systems where there is only one CPU, issuing the preceding command will inevitably use only one CPU. However, on a multicore system, providing the number of cores that can be used for mining helps to address any performance concerns.

After the preceding command is issued as preparation for mining, the DAG generation process starts, which produces an output similar to the one shown here:

```
INFO [01-25|13:39:53.143] Commit new mining work
INFO [01-25|13:39:55.208] Generating DAG in progress
INFO [01-25|13:39:57.712] Generating DAG in progress
INFO [01-25|13:40:01.249] Generating DAG in progress
INFO [01-25|13:40:02.867] Generating DAG in progress
INFO [01-25|13:40:05.213] Generating DAG in progress
INFO [01-25|13:40:07.222] Generating DAG in progress
INFO [01-25|13:40:09.772] Generating DAG in progress
INFO [01-25|13:40:12.061] Generating DAG in progress
INFO [01-25|13:40:13.519] Generating DAG in progress
INFO [01-25|13:40:15.099] Generating DAG in progress
INFO [01-25|13:40:16.652] Generating DAG in progress
INFO [01-25|13:40:17.958] Generating DAG in progress
INFO [01-25|13:40:19.491] Generating DAG in progress
INFO [01-25|13:40:20.962] Generating DAG in progress
INFO [01-25|13:40:22.668] Generating DAG in progress
INFO [01-25|13:40:24.630] Generating DAG in progress
INFO [01-25|13:40:26.263] Generating DAG in progress
INFO [01-25|13:40:27.737] Generating DAG in progress
INFO [01-25|13:40:29.253] Generating DAG in progress
INFO [01-25|13:40:30.765] Generating DAG in progress
INFO [01-25|13:40:32.439] Generating DAG in progress
INFO [01-25|13:40:33.949] Generating DAG in progress
INFO [01-25|13:40:35.444] Generating DAG in progress
INFO [01-25|13:40:36.001] Generating DAG in progress
number=1 sealhash=4be1cd...a8db43 uncles=0 txs=0 gas=0 fees=0 elapsed=11.009ms
epoch=0 percentage=0 elapsed=3.388s
epoch=0 percentage=1 elapsed=3.898s
epoch=0 percentage=2 elapsed=7.429s
epoch=0 percentage=3 elapsed=9.847s
epoch=0 percentage=4 elapsed=11.393s
epoch=0 percentage=5 elapsed=13.483s
epoch=0 percentage=6 elapsed=15.953s
epoch=0 percentage=7 elapsed=18.242s
epoch=0 percentage=8 elapsed=19.699s
epoch=0 percentage=9 elapsed=21.279s
epoch=0 percentage=10 elapsed=22.732s
epoch=0 percentage=11 elapsed=24.130s
epoch=0 percentage=12 elapsed=25.671s
epoch=0 percentage=13 elapsed=27.142s
epoch=0 percentage=14 elapsed=28.848s
epoch=0 percentage=15 elapsed=30.810s
epoch=0 percentage=16 elapsed=32.443s
epoch=0 percentage=17 elapsed=33.917s
epoch=0 percentage=18 elapsed=35.433s
epoch=0 percentage=19 elapsed=36.945s
epoch=0 percentage=20 elapsed=38.619s
epoch=0 percentage=21 elapsed=40.129s
epoch=0 percentage=22 elapsed=41.624s
epoch=0 percentage=23 elapsed=42.095s
```

Figure 13.6: DAG generation

DAG stands for **Directed Acyclic Graph**. We briefly introduced this when discussing **Ethash** in *Chapter 12, Further Ethereum*. In the context of Ethereum's Ethash PoW algorithm, DAG refers to **Dagger**, which is a memory-hard PoW algorithm based on moderately connected directed acyclic graphs. The aim of the Dagger algorithm is to provide an ASIC-resistant memory-hard PoW algorithm. The Dagger algorithm works by generating a DAG of a few gigabytes every 30,000 blocks. This DAG

serves as a resource for PoW where the PoW mechanism needs to choose subsets from the DAG, which is dependent on the nonce and the block header. This is in contrast to Bitcoin PoW, which is based on calculating SHA-256, which is not resistant to specialized hardware such as ASICs.

Once DAG generation is finished, the mining process starts and blocks are produced. Geth will produce an output similar to that shown in the following. It can be seen that blocks are being mined successfully with the `mined potential block` message:

```
INFO [01-25|13:58:07.405] Successfully sealed new block
number=96 sealhash=02334c...f691fe hash=75302b...a016d5 elapsed=1.33
INFO [01-25|13:58:07.405] ↗ mined potential block
number=96 hash=75302b...a016d5
INFO [01-25|13:58:07.405] Commit new mining work
number=97 sealhash=4d4e6d...2906a8 uncles=0 txs=0 gas=0 fees=0 elapsed=0.00
INFO [01-25|13:58:18.527] Successfully sealed new block
number=97 sealhash=4d4e6d...2906a8 hash=817c8f...8012f6 elapsed=11.1
INFO [01-25|13:58:18.529] Commit new mining work
number=98 sealhash=9e8370...008b58 uncles=0 txs=0 gas=0 fees=0 elapsed=0.00
INFO [01-25|13:58:18.529] ↗ mined potential block
number=97 hash=817c8f...8012f6
INFO [01-25|13:58:18.634] Successfully sealed new block
number=98 sealhash=9e8370...008b58 hash=caba0f...cc206e elapsed=105.
```

Mining can be stopped using the following command:

```
> miner.stop()
null
```

Now, before the further exploration of different methods, let's understand some basics of using the JavaScript console.

In the JavaScript console, we can perform several operations. A general tip is that if two spaces and two tabs on the keyboard are pressed in a sequence, a complete list of the available objects will be displayed. This is shown in the following screenshot:

Array	String	encodeURI	parseFloat
BigNumber	SyntaxError	encodeURIComponent	parseInt
Boolean	TypeError	escape	personal
Date	URIError	eth	propertyIsEnumerable
Error	Web3	ethash	require
EvalError	XMLHttpRequest	eval	rpc
Function	_setInterval	hasOwnProperty	setInterval
Infinity	_setTimeout	inspect	setTimeout
JSON	admin	isFinite	toLocaleString
Math	clearInterval	isNaN	toString
NaN	clearTimeout	isPrototypeOf	txpool
Number	console	jeth	undefined
Object	constructor	loadScript	unescape
RangeError	debug	message	valueOf
ReferenceError	decodeURI	miner	web3
RegExp	decodeURIComponent	net	

Figure 13.7: Available objects

Furthermore, when a command is typed, it can be autocompleted by pressing Tab twice. If two tabs are pressed, then the list of available methods is also displayed. This is shown in the following screenshot:

```
> personal.
personal._requestManager  personal.importRawKey      personal.openWallet
personal.constructor        personal.initializeWallet personal.sendTransaction
personal.deriveAccount     personal.listAccounts    personal.sign
personal.ecRecover         personal.listWallets   personal.signTransaction
personal.getListAccounts  personal.lockAccount  personal.unlockAccount
personal.getListWallets   personal.newAccount   personal.unpair
> net.
net._requestManager       net.getListening      net.getVersion      net.peerCount
net.constructor            net.getPeerCount     net.listening      net.version
```

Figure 13.8: Available methods

In addition to the previously mentioned command, in order to get a list of available methods of an object, after typing a command, a semicolon, ;, is entered. An example is shown in the next screenshot, which shows a list of all the methods available for `net`:

```
[> net;
{
  listening: true,
  peerCount: 0,
  version: "786",
  getListening: function(callback),
  getPeerCount: function(callback),
  getVersion: function(callback)
}
```

Figure 13.9: List of methods

There is also the `eth` object available, which has several methods. While there are several methods under this object, the most common is `getBalance`, which we can use to query the current balance of Ether. This is shown in the following example.

```
> eth.getBalance(eth.coinbase)
55000000000000000000000000000000
```

After mining, a significant amount can be seen here. Mining is extremely fast as it is a private network with no competition for solving the PoW, and also in the genesis file, the network difficulty has been set to quite low.

The preceding balance is shown in Wei. If we want to see the output in ether, we can use the `web3` object, as shown here.

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")
550
```

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

- Get the current gas price:

```
> eth.gasPrice
1000000000
```

- Get the latest block number:

```
> eth.blockNumber  
110
```

`debug` can come in handy when debugging issues. A sample command is shown here; however, there are many other methods available. A list of these methods can be viewed by typing `debug`.

- The following method will return the RLP of block 0:

The preceding output shows the block 0 in RLP-encoded format.

- Create a new account. Note that `Password123` is the password chosen as an example, but you can choose any:

```
> personal.newAccount("Password123")
"0xd6e364a137e8f528ddb2d2bb2356d124c9a08206"
```

- Unlock the account before sending transactions:

```
> personal.unlockAccount("0xd6e364a137e8f528ddbad2bb2356d124c9a08206")
Unlock account 0xd6e364a137e8f528ddbad2bb2356d124c9a08206
Password:
true
```

- We use `allow unsecure unlocking`, otherwise the accounts cannot be unlocked with HTTP access. If that is the case you will see an error message as follows:

```
> personal.unlockAccount("0xd6e364a137e8f528ddbad2bb2356d12")
Unlock account 0xd6e364a137e8f528ddbad2bb2356d124c9a08206
Password:
Error: account unlock with HTTP access is forbidden
```

To circumvent the error, restart Geth using the `--allow-insecure-unlock` flag.

```
s geth --datadir ~/etherprivate/ --allow-insecure-unlock -network testnet
```

Now we unlock both accounts that we created earlier with the `personal.newAccount()` command at the start of the private network we're creating, and just before the `unlockAccount()` command on this page. The first account we created at the start of our private network is "0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811" and another one "0xd6e364a137e8f528ddbad2bb2356d124c9a08206" just at the start of this page.

```
> personal.unlockAccount("0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811  
Unlock account 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811  
Password:  
true  
> personal.unlockAccount("0xd6e364a137e8f528ddbad2bb2356d124c9a08206  
Unlock account 0xd6e364a137e8f528ddbad2bb2356d124c9a08206  
Password:  
true
```

Now once we have these accounts unlocked we can issue some further command to query the balances they hold. First, let's check the balance of our account that we created earlier when we started our private network the first time.

```
> web3.fromWei(eth.getBalance("0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811")  
550
```

As this account is also our coinbase account, which is the default account that receives the mining reward, we can also query the balance slightly differently by specifying `eth.coinbase` in the command , as shown here.

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")  
550
```

Finally, we check the balance of the other account that we created. As this has not received any mining reward, the balance is 0 as expected.

```
> web3.fromWei(eth.getBalance("0xd6e364a137e8f528ddbad2bb2356d120
```

Now let's try to send transactions from one account to another. In this example we'll send ether from account

`0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811` to
`0xd6e364a137e8f528ddbad2bb2356d124c9a08206` using the
`sendTransaction` commands shown as follows.

First, we send a value of `100`:

```
> eth.sendTransaction({from: "0xc9bf76271b9e42e4bf7e1888e0f52351
```

This command outputs the transaction hash, which is a unique identifier used to identify a specific transaction. The output is shown as follows:

```
"0xe88e0cc21d59832c4aeabad7a2028fee036e6be17ba04491aa2bf30d1eeef7
```

This command will transfer 100 Wei (the smallest ether unit) to the target account. To transfer the value in ether, in our example 100 ETH, we can use the command slightly differently and use `web3.toWei`, which will convert the value from ether into Wei. To achieve this we issue the command as shown here:

```
> eth.sendTransaction({from: "0xc9bf76271b9e42e4bf7e1888e0f52351
```

This command outputs the transaction hash as shown here:

```
"0xa39eba9403b623477c90cdeaa4733c056cb9f3d9066b9ec3d8c6caa50e43c
```

The preceding command makes use of `web3.toWei`, which takes two parameters, `value` which is 100 in our example, and a string, `ether`, which is the unit of value. This means that 100 ETH will be converted into its equivalent Wei and will be used in the `sendTransaction` command. The result of this command will be transferring 100 ETH from our source account to the target account. Also notice, in the logs in the other console/terminal window from where we ran the Geth command to run our private network earlier at the start of the private network, the log message:

```
INFO [01-25|14:36:27.458] Submitted transaction  
fullhash=0xa39eba9403b623477c90cdeaa4733c056cb9f3d9066b9ec3d8c6c
```

This message shows that the transaction has been submitted, but it needs to be mined first in order for the transaction to take effect. As we stopped mining earlier, for this transaction to go through we need to start mining again, otherwise the transaction won't be processed:

```
> miner.start()  
true
```

Now check in the logs that the mining has started to work:

```
INFO [01-25|14:36:34.322] Updated mining threads  
threads=4  
INFO [01-25|14:36:34.324] Transaction pool price threshold updat  
INFO [01-25|14:36:34.325] Commit new mining work  
number=121 sealhash=0c1d94...a39646 uncles=0 txs=0 gas=0 fees=0 el  
INFO [01-25|14:36:34.325] Commit new mining work  
number=121 sealhash=6826a1...4bdc64 uncles=0 txs=1 gas=21000 fees=  
INFO [01-25|14:36:50.554] Successfully sealed new block  
number=121 sealhash=6826a1...4bdc64 hash=8bc2d3...ff29d6 elapsed=16.  
INFO [01-25|14:36:50.556] Commit new mining work  
number=122 sealhash=df3cd3...d28edc uncles=0 txs=0 gas=0 fees=  
INFO [01-25|14:36:50.557] ↗ mined potential block  
number=121 hash=8bc2d3...ff29d6
```

Note that the amount of funds has been transferred.

```
> web3.fromWei(eth.getBalance("0xd6e364a137e8f528ddbad2bb2356d12  
100.0000000000000001  
> web3.fromWei(eth.getBalance("0xc9bf76271b9e42e4bf7e1888e0f5235  
634.9999999999999999
```



Remember that as the mining has progressed, you will see slightly different numbers and more ether in the source account (coinbase) and the target account will now have 100 ETH, which we transferred.

It could be a bit cumbersome to type all these account IDs. Instead of typing these long account IDs, we can also use the `listAccounts[]` method, which takes an integer parameter to address the account. For example, `0`, which represents the first account that we created.

As `listAccounts[]` returns all the account addresses of all the keys in the keystore, by providing `[0]` as a parameter we can refer to the first account that we created.

Now let's see an example of using the `listAccounts[]` method, as shown here:

```
> eth.sendTransaction({from: personal.listAccounts[0], to: persc  
"0xd8e1911a3783d1976a21018be9981ade7adb4f38c09d4ec5587c4abd710ac
```

We can also now query information about the transaction that we executed earlier. Remember how it returned a transaction hash? We can use that to find out details about the transaction.

This will produce the output shown as follows:

Figure 13.10: Get Transaction Receipt (getTransactionReceipt)

Notice the `root` in the preceding output, the transaction root, which will be available in the block header as the Merkle root of the transaction trie.

Similarly, we can query more information about the transaction using the `getTransaction` method.

```
> eth.getTransaction("0xa39eba9403b623477c90cd0aa4733c056cb9f3dc")
```

This will produce the output shown as follows:

```
{  
    blockHash: "0x8bc2d3d36419da0a52cd1cc283de7e98e2fc3ded90f6b61d97f1090249ff29d6",  
    blockNumber: 121,  
    from: "0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811",  
    gas: 21000,  
    gasPrice: 1000000000,  
    hash: "0xa39eba9403b623477c90cdeaa4733c056cb9f3d9066b9ec3d8c6caa50e43df14",  
    input: "0x",  
    nonce: 1,  
    r: "0xfeffaad48f5e5978ba8eac19e88699ad8e63add8b4bce409407bb198b9a718",  
    s: "0x6fb7cef0e97f67c91318bad16c05da6c3c917feb42427e17b2e0e72520488187",  
    to: "0xd6e364a137e8f528ddbad2bb2356d124c9a08206",  
    transactionIndex: 0,  
    v: "0x647",  
    value: 10000000000000000000  
}
```

Figure 13.11: Get Transaction (getTransaction)

In this section, we covered how to start mining on a private network and ran some transactions. We also saw how transactions can be created and balance can be queried and performed an ether transfer transaction from one account to another. We also covered how the transaction results can be queried using a transaction receipt and other relevant methods available via RPC in the Geth client.

We've seen in the examples previously that there is a rich interface available with hundreds of methods to interact with the blockchain. While this console- / terminal-based mechanism is quite useful, it can become a bit difficult to manage and deploy smart contracts using the command-line console. For this, we need better alternatives, which we will discuss next. Just like any other development ecosystem in IT, the Ethereum ecosystem has come a long way toward providing high-quality and user-friendly development tools.

We will introduce some of these in the next section.

Remix IDE

There are various **Integrated Development Environments (IDEs)** available for Solidity development. Most of the IDEs are available online and are presented via web interfaces. Remix (formerly browser Solidity) is the most commonly used IDE for building and debugging smart contracts. It is discussed here.

Remix is the web-based environment for the development and testing of contracts using Solidity. It is a feature-rich IDE that does not run on a live blockchain; in fact, it is a simulated environment in which contracts can be deployed, tested, and debugged. It is available at <https://remix.ethereum.org>.

An example interface is shown as follows:

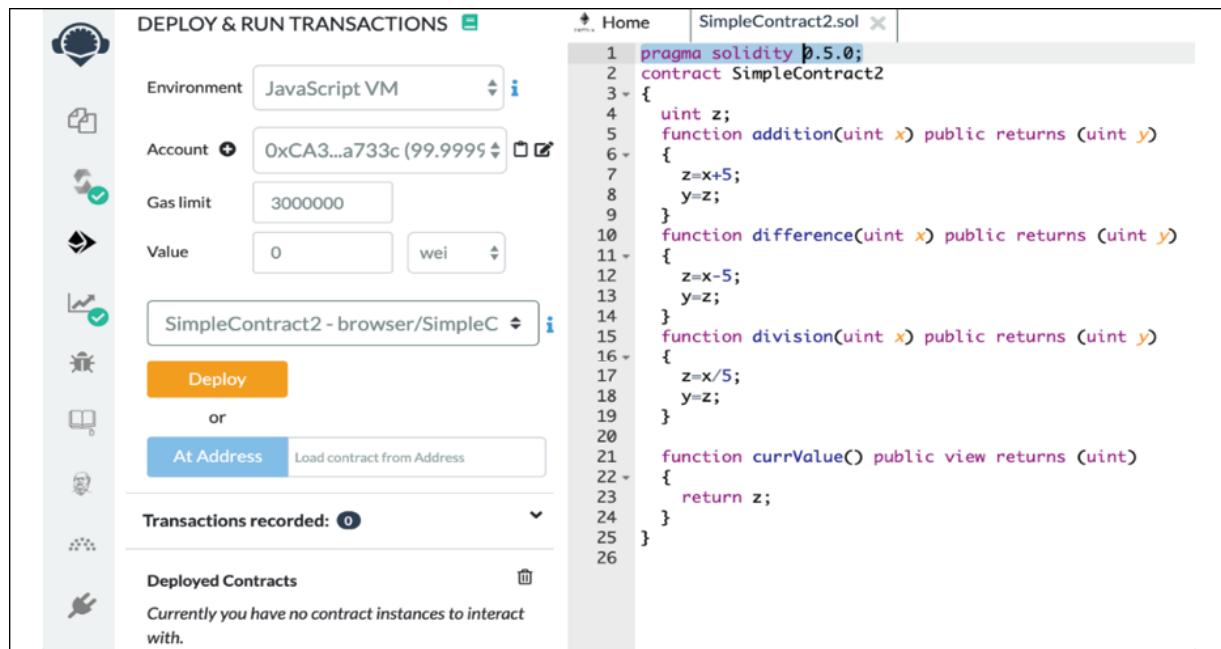


Figure 13.12: Remix IDE

On the left-hand side, there is a column with different icons. These icons represent various plugins of the Remix IDE. When you run Remix for the first time, it won't show any of the plugins. In order to add plugins to Remix IDE, you need to access the plugin manager to activate the plugins you need. This is shown in the following figure.

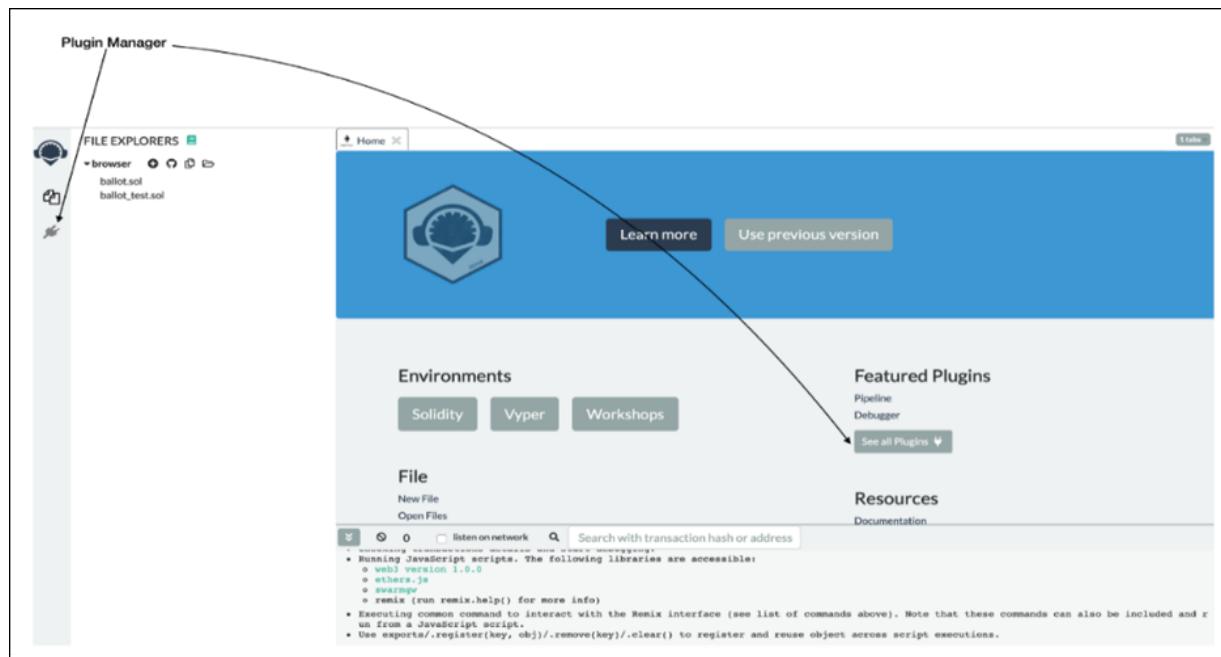


Figure 13.13: Remix IDE with default plugins on the first start

There are a number of plugins available. A sample screenshot is shown here.

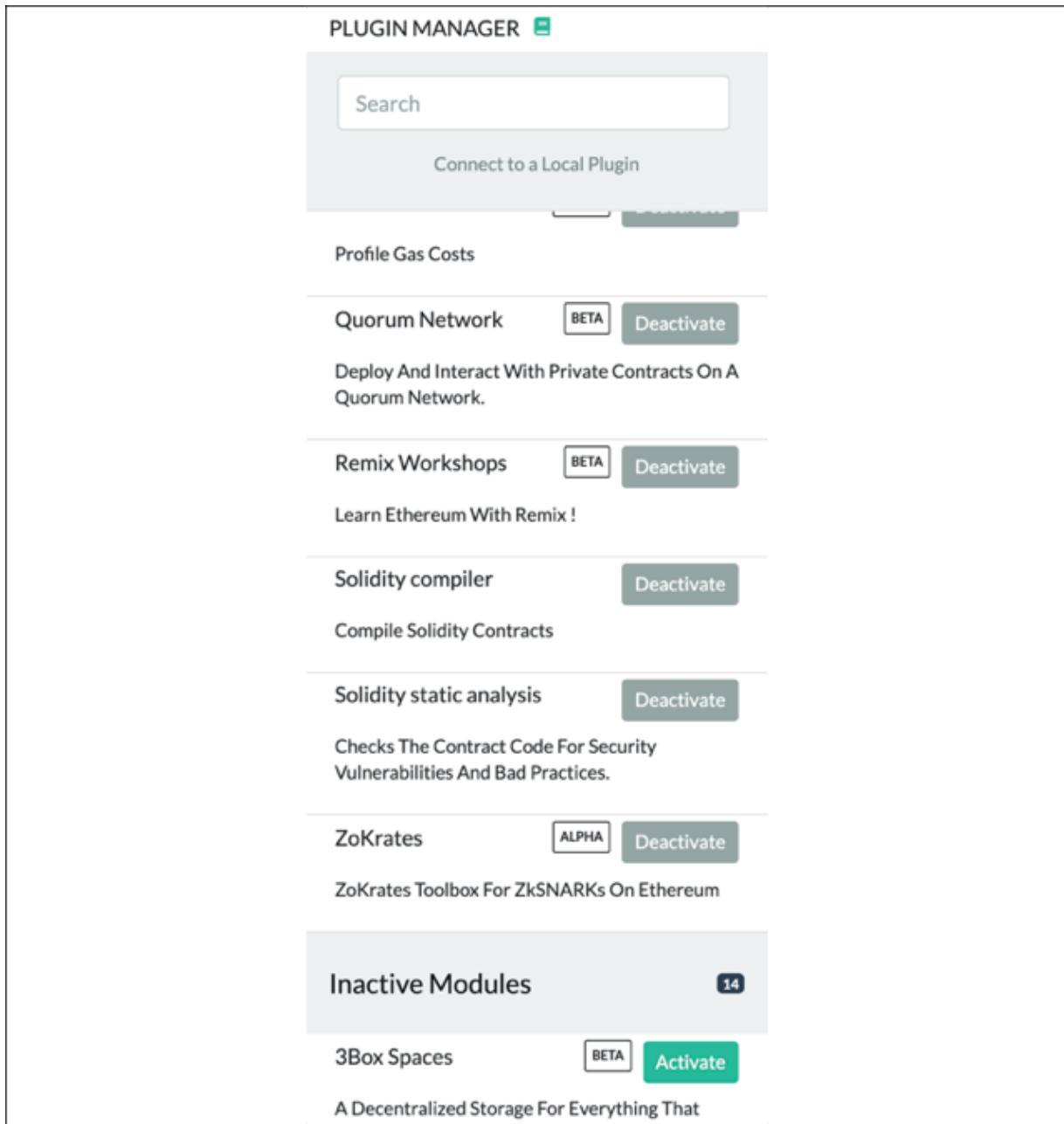


Figure 13.14: Remix Plugin Manager

Once activated, the plugins will appear in the left-most column of the IDE, as shown in the following image. Again note that this will only show the

plugins that are activated, and other plugins or local plugins can be activated as required. The following screenshot provides more details of different elements of the Remix IDE.

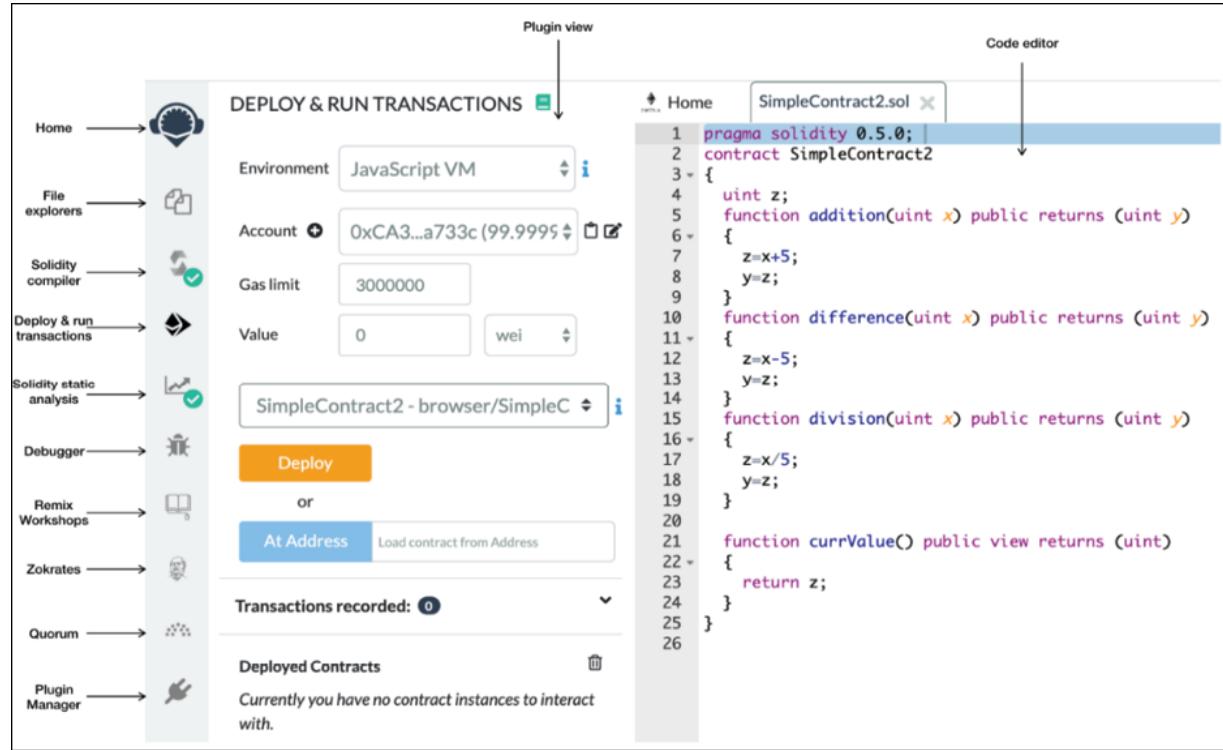


Figure 13.15: Remix IDE with plugins

On the right-hand side, there is a code editor with syntax highlighting and code formatting, and on the left-hand side, there are a number of plugins available that can be used to deploy, debug, test, and interact with the contract.

Various features, such as transaction interaction, options to connect to the JavaScript VM, the configuration of an execution environment, debugger, formal verification, and static analysis are available. They can be configured to connect to execution environments such as the JavaScript VM, injected Web3—where Mist, MetaMask, or a similar environment has provided the execution environment—or the Web3 provider, which allows connection to the locally running Ethereum client (for example, `geth`) via IPC or RPC over HTTP (Web3 provider endpoint).

Remix also has a debugger plugin for EVM that is very powerful and can be used to perform detailed level tracing and analysis of the EVM bytecode. An example is shown here:

The screenshot displays the Remix IDE's DEBUGGER interface. On the right, the source code for a Solidity contract named 'Addition' is shown:

```

1 pragma solidity ^0.5.0;
2 contract Addition
3 {
4     uint8 x;
5     function addx(uint8 y, uint8 z) public
6     {
7         x = y + z;
8     }
9     function retrievex() view public returns (uint8)
10    {
11        return x;
12    }
13 }
14

```

On the left, the EVM trace log shows the following steps:

- 136 ADD
- 137 SWAP1
- 138 SWAP3
- 139 SWAP2
- 140 SWAP1
- 141 POP
- 142 POP
- 143 POP

Information below the log includes:

- vm trace step: 99
- execution step: 99
- add memory:
- gas: 0
- remaining gas: 2957213
- loaded address: 0x08970fed061e7747cd9a38d680a601510cb659fb

At the bottom, there are navigation buttons for the trace log and sections for Solidity Locals, Solidity State (showing x: 10 uint8), and Stack.

On the far right, transaction details are listed:

status	0x1 Transaction mined and execution succeed
transaction hash	0xefea566ec440475cc7580d5d9e4a6d9ba90eb2da97cb253fa82c0d23dff79e09
contract address	0x08970fed061e7747cd9a38d680a601510cb659fb
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Addition.(constructor)
gas	3000000 gas
transaction cost	130931 gas
execution cost	58711 gas
hash	0xefea566ec440475cc7580d5d9e4a6d9ba90eb2da97cb253fa82c0d23dff79e09
input	0x608...10029
decoded input	{}
decoded output	-

Figure 13.16: Remix IDE, debugging

The preceding screenshot shows different elements of the Remix IDE when **DEBUGGER** is running. On the right-hand side, the source code is shown. Below that is the output log, which shows informational messages and data related to compilation and the execution status, and transaction information of the transaction/contract.

The following screenshot shows the Remix debugger in more detail. It has the source code decoded into EVM instructions. The user can step through the instructions one by one and examine what the source code does when executed:

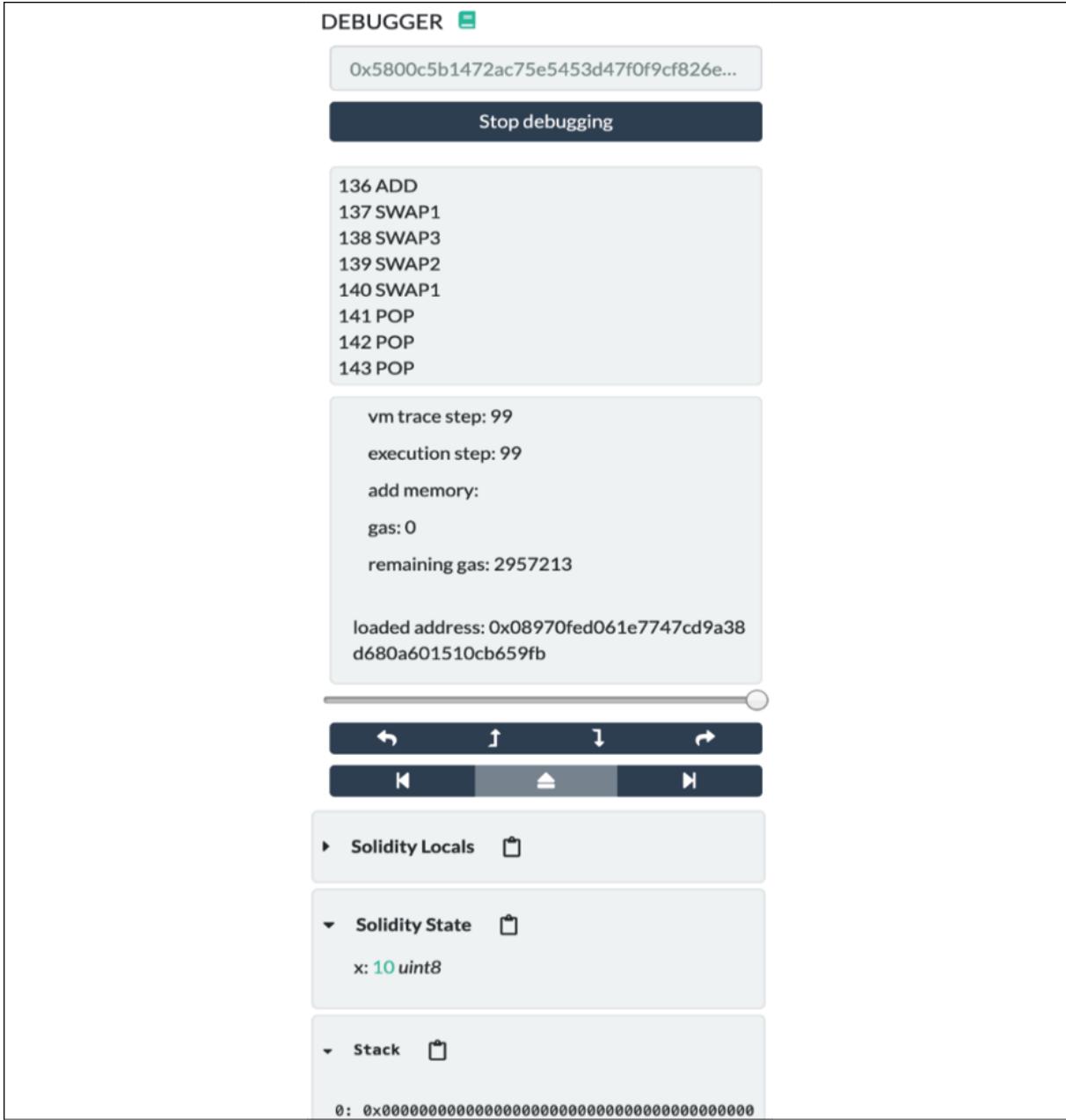


Figure 13.17 Remix Debugger

Note that in the preceding screenshot the opcodes are displayed, along with memory, state, and stack information. Notice that Solidity state has a value of `10`, assigned to the variable `x` after the execution of the transaction. There are also a number of other related information points shown in the debugger. This is a very useful feature and comes in handy, especially in complex code debugging.

In the next section, we'll be using MetaMask, a browser extension that serves as a cryptocurrency wallet and an interface to blockchains and DApps.

MetaMask

MetaMask allows interaction with Ethereum blockchain via the Firefox and Chrome browsers. It injects a `web3` object within the running websites' JavaScript context, which allows immediate interface capability for DApps. This injection allows DApps to interact directly with the blockchain.



It is available at <https://metamask.io/>.

Further information is available at
<https://github.com/MetaMask/metamask-plugin>.

MetaMask also allows account management. This acts as a verification method before any transaction is executed on the blockchain. The user is shown a secure interface to review the transaction for approval or rejection before it can reach the target blockchain.

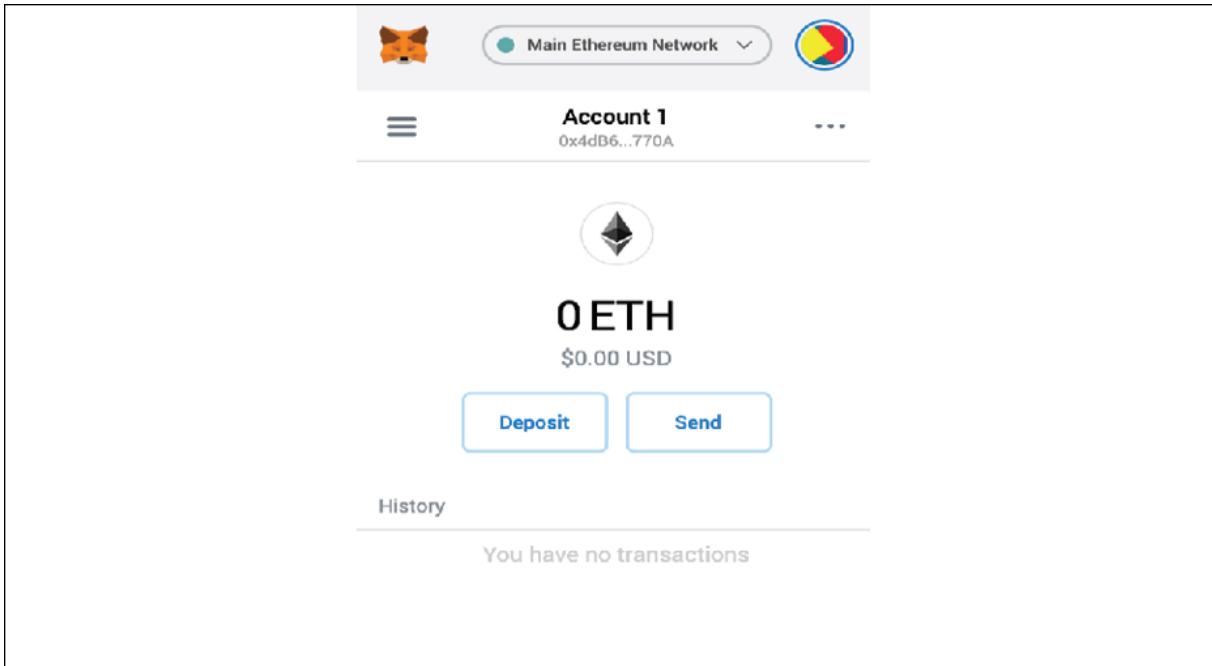


Figure 13.18: MetaMask

It allows connectivity with various Ethereum networks as shown in the following screenshot. This is a screenshot of the MetaMask **User Interface (UI)**, where it allows users to select the network of their choice:

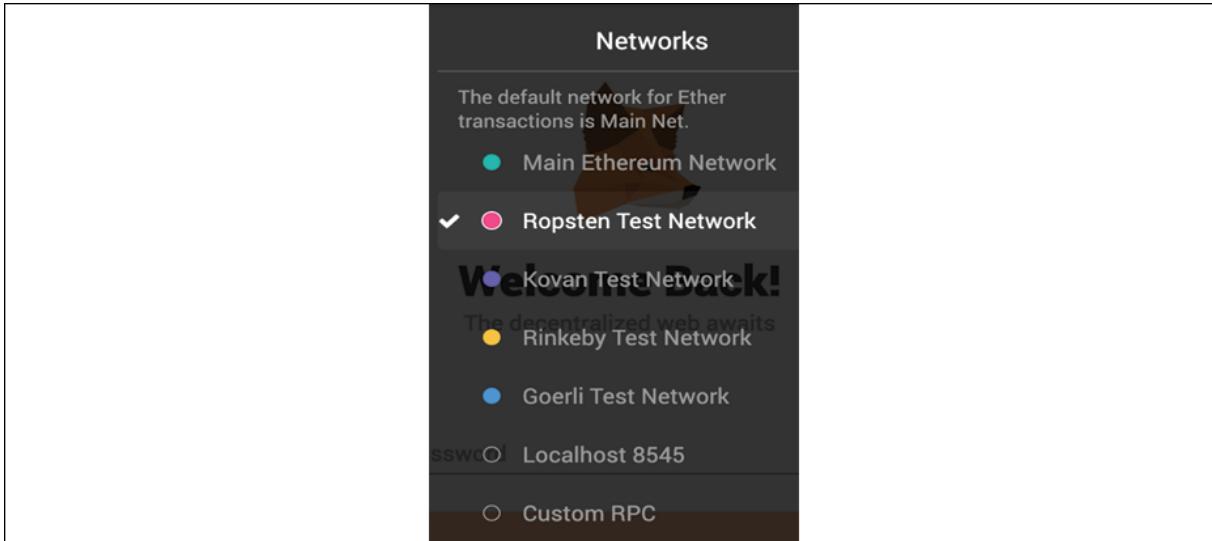


Figure 13.19: MetaMask networks as shown in the MetaMask UI

An interesting feature to note is that MetaMask can connect to any Ethereum blockchain via the custom RPC connection. It can connect to not

only remote blockchains but also to locally running blockchains. All it needs is an RPC connection exposed by a node running on the blockchain. If it's available, MetaMask can connect and will allow a web browser to connect to it via the `web3` object. MetaMask can also be used to connect to a locally running test (or simulated) blockchain like Ganache and TestRPC.

MetaMask allows account management and also records all transactions for these accounts. This is shown in the following screenshot:

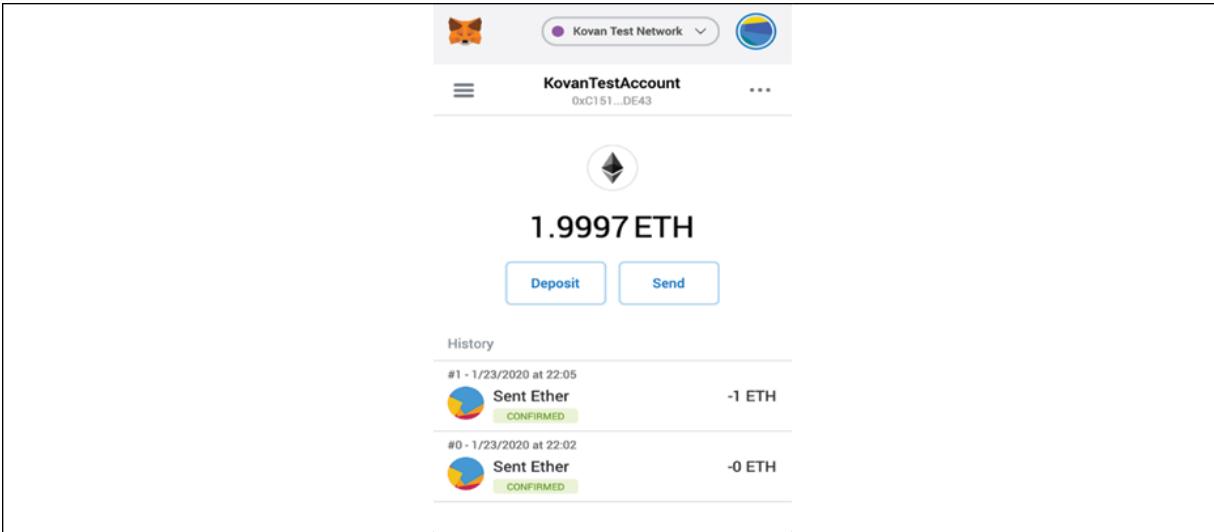


Figure 13.20: MetaMask accounts and transactions view

As we now understand what MetaMask is, we can now delve deeper and see how the Remix IDE can deploy smart contracts through MetaMask.

Using MetaMask and Remix IDE to deploy a smart contract

MetaMask was introduced in *Chapter 12, Further Ethereum*. It is an interface between the Ethereum blockchain and the web browser. It enables easy access to the blockchain and is quite useful for development activities.

As MetaMask injects a `web3` object into the browser, we can use it with Remix IDE to deploy contracts to the blockchain.

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

We will use Remix IDE and MetaMask to deploy a sample smart contract to the local running private blockchain that we just created in the last section.

In the exercise, a simple contract that can perform various simple arithmetic calculations on the input parameter will be used. As we have not yet introduced **Solidity**, the aim here is to demonstrate the contract deployment and interaction process only.

More information on coding and Solidity will be provided later in *Chapter 14, Development Tools and Frameworks* and *Chapter 15, Introducing Web3*, after which the following code will become easy to understand. Those of you who are already familiar with JavaScript or any other similar language such as the C language will find the code almost self-explanatory.

Now, let's look at some examples of how MetaMask can be used in practice with the private net that we have created earlier in this chapter.

Adding a custom network to MetaMask and connecting Remix IDE with MetaMask

First ensure that you have MetaMask available as we set it up in *Chapter 12, Further Ethereum*. If not, you can refer back to the previous chapter for an installation guide. In this section, we will add our local private network in MetaMask and then interact with it using Remix IDE.

Open the Google Chrome web browser, where MetaMask is installed. Select **localhost 8545**, where the Geth instance of our private net is listening on:

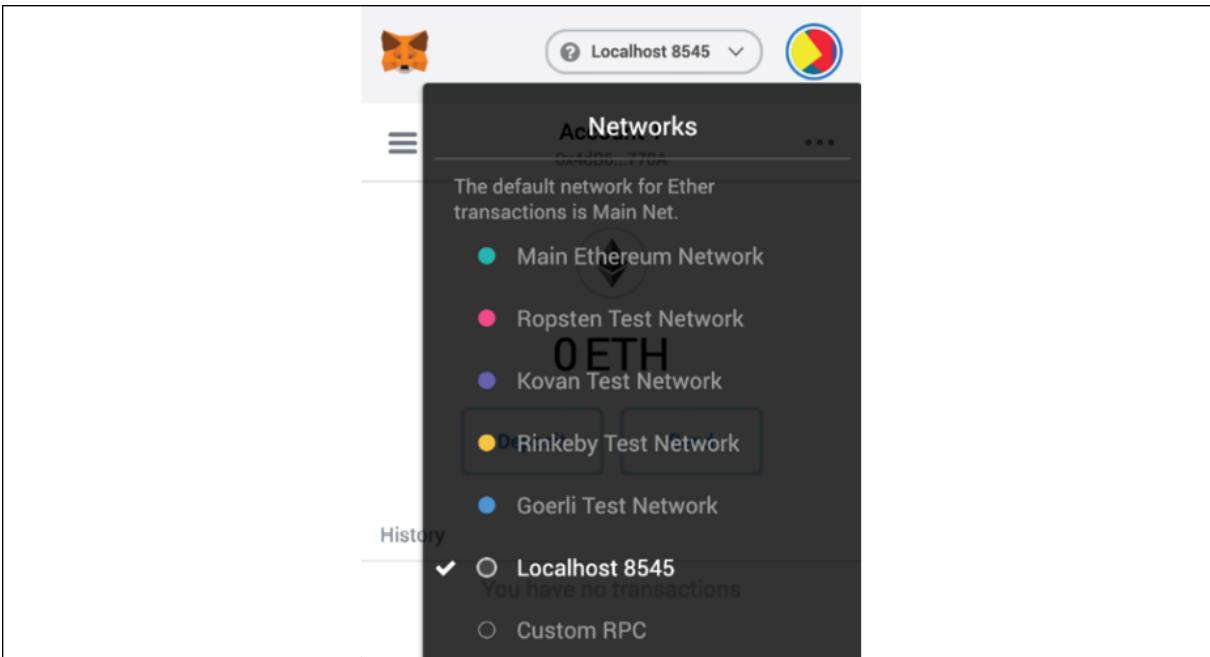


Figure 13.21: MetaMask network selection

Navigate to Remix IDE for Ethereum smart contract development on your browser at <https://remix.ethereum.org>.

Once on the website, notice the **DEPLOY & RUN TRANSACTIONS** option in the left-hand column. Choose **Injected Web3** as the **Environment**. This is shown in the following screenshot on the left-hand side:

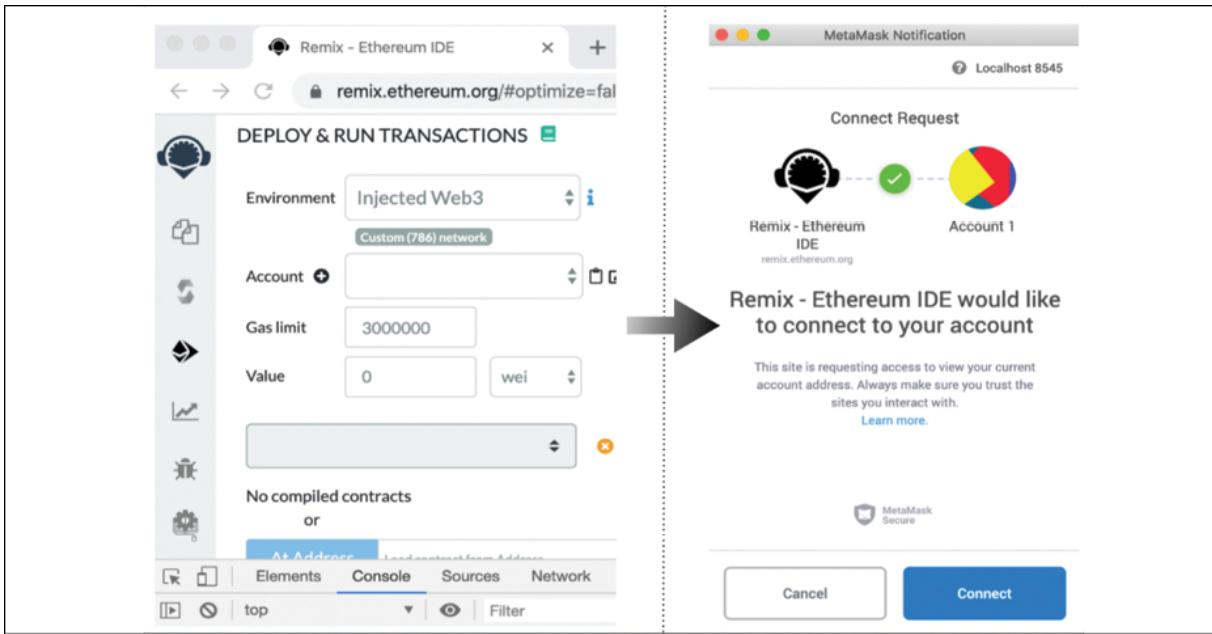


Figure 13.22: Remix IDE and interaction with MetaMask

A MetaMask window will open as shown in the preceding screenshot, on the right-hand side.

Note that when Remix IDE is connected to MetaMask, it will show network information, such as **Custom (786) network**, and account information, as shown in the following screenshot.

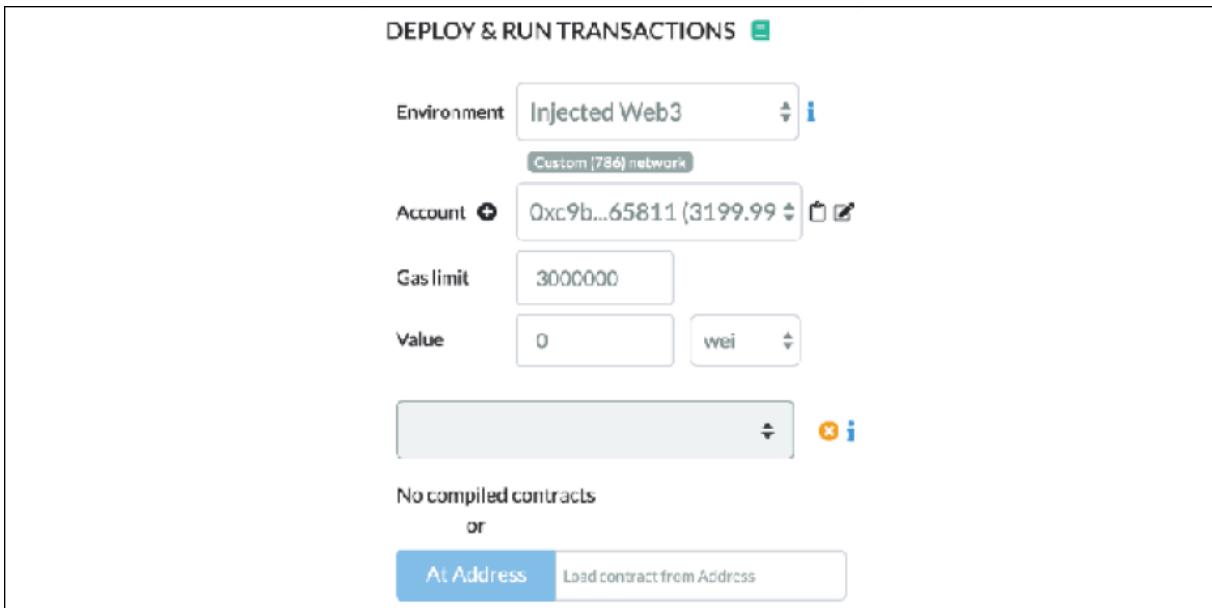


Figure 13.23: Network information and account information in Remix

In this section, we saw how Remix can connect to MetaMask using the injected Web3 environment. MetaMask can be connected to the mainnet or any other network (local or test network) but from Remix's point of view, as long as the injected Web3 object is available, it will connect to it.

In our example, as we are using a private network, we have connected to our private network through MetaMask, and Remix is connecting to MetaMask using the injected Web3 environment. Now even if we have this connectivity available, MetaMask does not know about the accounts that we have created in our private network. In order for MetaMask to operate on existing accounts, we need to import them from the existing keystore, in our case, the private network `786` keystore. We will see how this is done in the next section.

Importing accounts into MetaMask using keystore files

We can import existing Ethereum accounts into MetaMask by importing the keystore files generated by Geth as a result of creating accounts. To demonstrate how this works we will now import the accounts from our private network blockchain, named `786`, into MetaMask.

In MetaMask, the **Import Account** option is available under the **My Accounts** menu, as shown in the following screenshot.

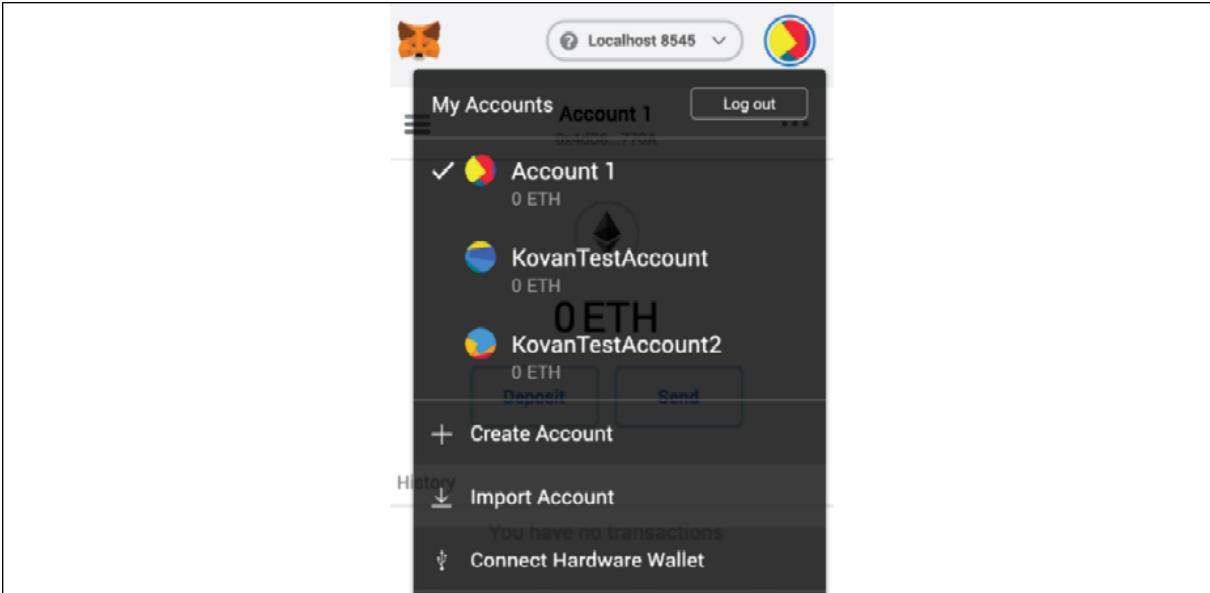


Figure 13.24: MetaMask Import Account option

Now let's import the accounts. Remember we created two accounts earlier in our private network [786](#):

```
> eth.accounts
["0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811", "0xd6e364a137e8f5
```

We can import these accounts into MetaMask using their associated keystore JSON files.

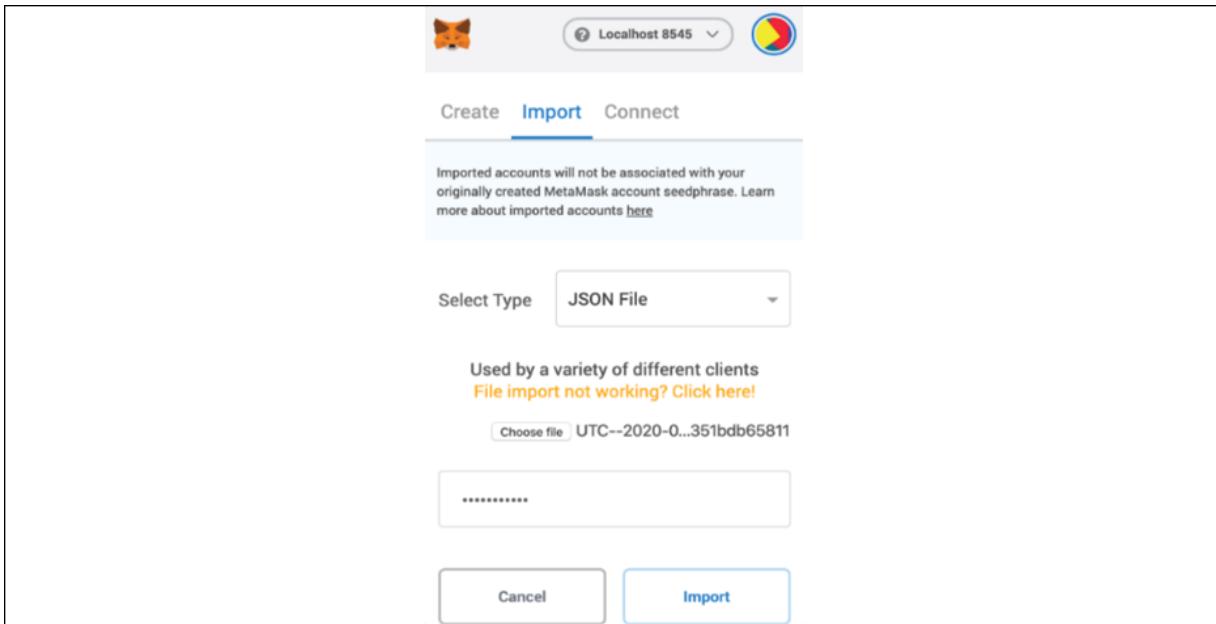
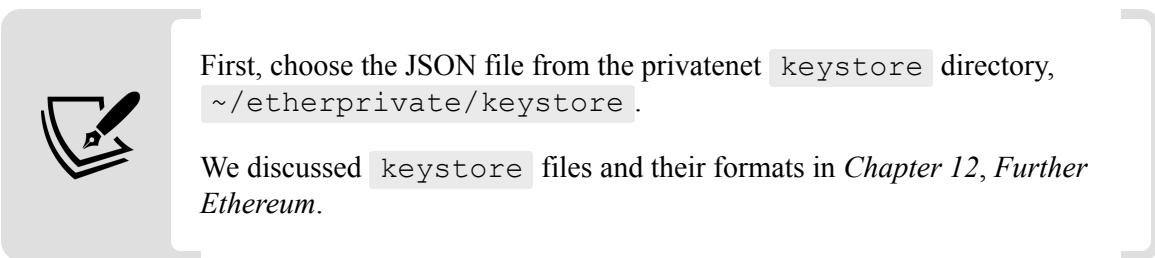


Figure 13.25: MetaMask JSON import file

The steps are as follows.



The `keystore` files for our privatenet are listed here:

```
UTC--2020-01-25T13-38-31.465900000Z--c9bf76271b9e42e4bf7e1888e0f  
UTC--2020-01-25T14-17-28.553531000Z--d6e364a137e8f528ddbad2bb235
```

Simply browse to the key store and select the `keystore` file, then enter the password (earlier, when we created the accounts for the first time, we set the password as `Password123`) and click **Import**.

It may take a few seconds to import. When imported, the account will be visible in the MetaMask window as shown here:

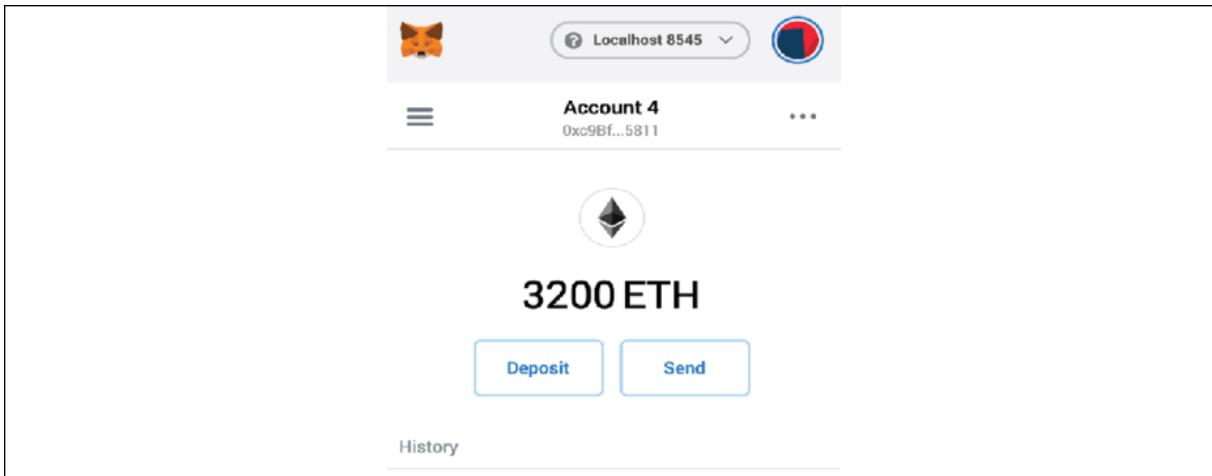


Figure 13.26: Account imported in MetaMask

Similarly, we can import the other account, by selecting the `keystore` file and importing it. Finally, we will have two accounts listed in MetaMask, as shown in the following image:

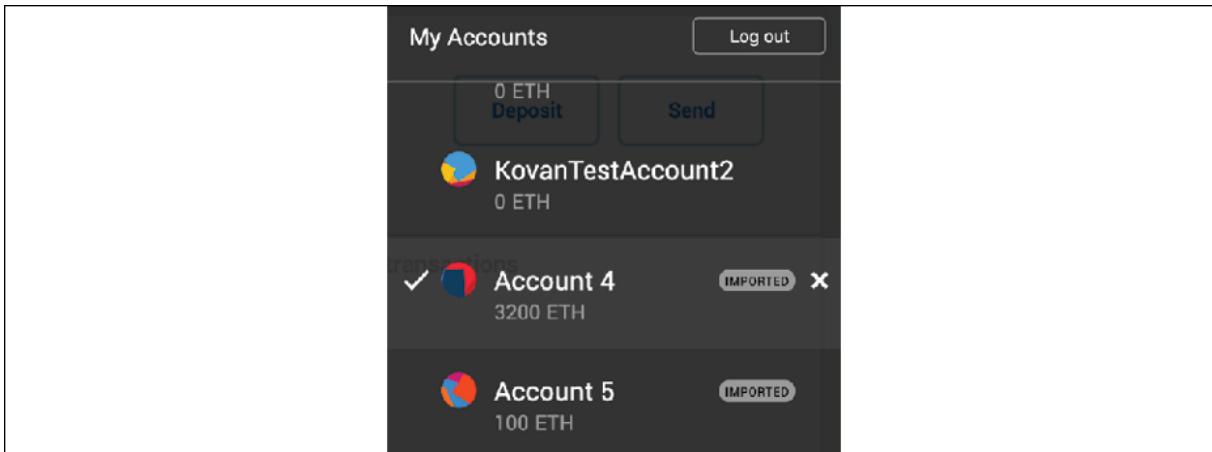


Figure 13.27: MetaMask – two accounts imported

Now that our accounts have been imported successfully, we can move onto using MetaMask to deploy a contract in our private network.

Deploying a contract with MetaMask

In this section, we will write a simple smart contract and deploy it on our private network using MetaMask. First, in Remix, we create a new file:

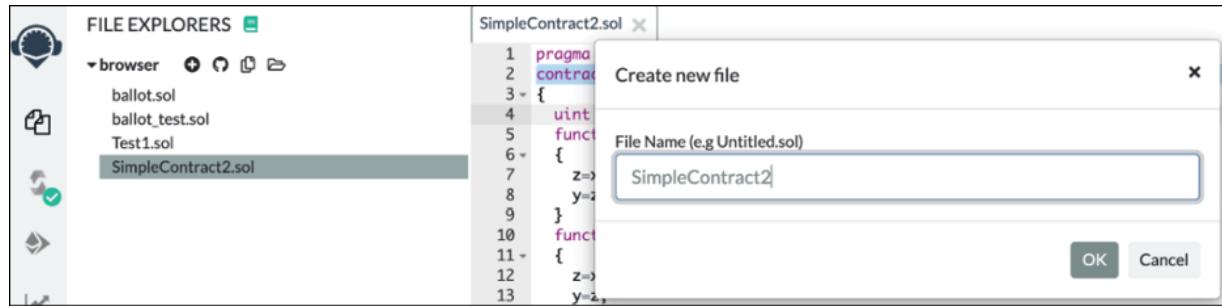


Figure 13.28: Creating a new file in Remix

Enter the code shown for **SimpleContract2** as follows:

```
pragma solidity 0.5.0;
contract SimpleContract2
{
    uint z;
    function addition(uint x) public returns (uint y)
    {
        z=x+5;
        y=z;
    }
    function difference(uint x) public returns (uint y)
    {
        z=x-5;
        y=z;
    }
    function division(uint x) public returns (uint y)
    {
        z=x/5;
        y=z;
    }

    function currValue() public view returns (uint)
    {
        return z;
    }
}
```

The code will look like this in Remix.

The screenshot shows the Remix IDE interface. On the left, the 'FILE EXPLORERS' sidebar lists several Solidity files: 'browser', 'ballot.sol', 'ballot_test.sol', 'Test1.sol', and 'SimpleContract2.sol'. The 'SimpleContract2.sol' file is currently selected and highlighted with a dark grey background. The main workspace on the right displays the source code for 'SimpleContract2.sol'. The code defines a contract named 'SimpleContract2' with four public functions: 'addition', 'difference', 'division', and 'currValue'. The 'addition' function takes an input 'x' and returns 'y'. The 'difference' function takes an input 'x' and returns 'y'. The 'division' function takes an input 'x' and returns 'y'. The 'currValue' function returns the current value of 'z'. The code uses the pragma directive `pragma solidity ^0.5.0;` at the top.

```
1 pragma solidity ^0.5.0;
2 contract SimpleContract2
3 {
4     uint z;
5     function addition(uint x) public returns (uint y)
6     {
7         z=x+5;
8         y=z;
9     }
10    function difference(uint x) public returns (uint y)
11    {
12        z=x-5;
13        y=z;
14    }
15    function division(uint x) public returns (uint y)
16    {
17        z=x/5;
18        y=z;
19    }
20    function currValue() public view returns (uint)
21    {
22        return z;
23    }
24 }
25
26 }
```

Figure 13.29: SimpleContract2 in Remix

Now we compile the smart contract `SmartContract2` by clicking on the **Compile SimpleContract2.sol** button, as shown here:

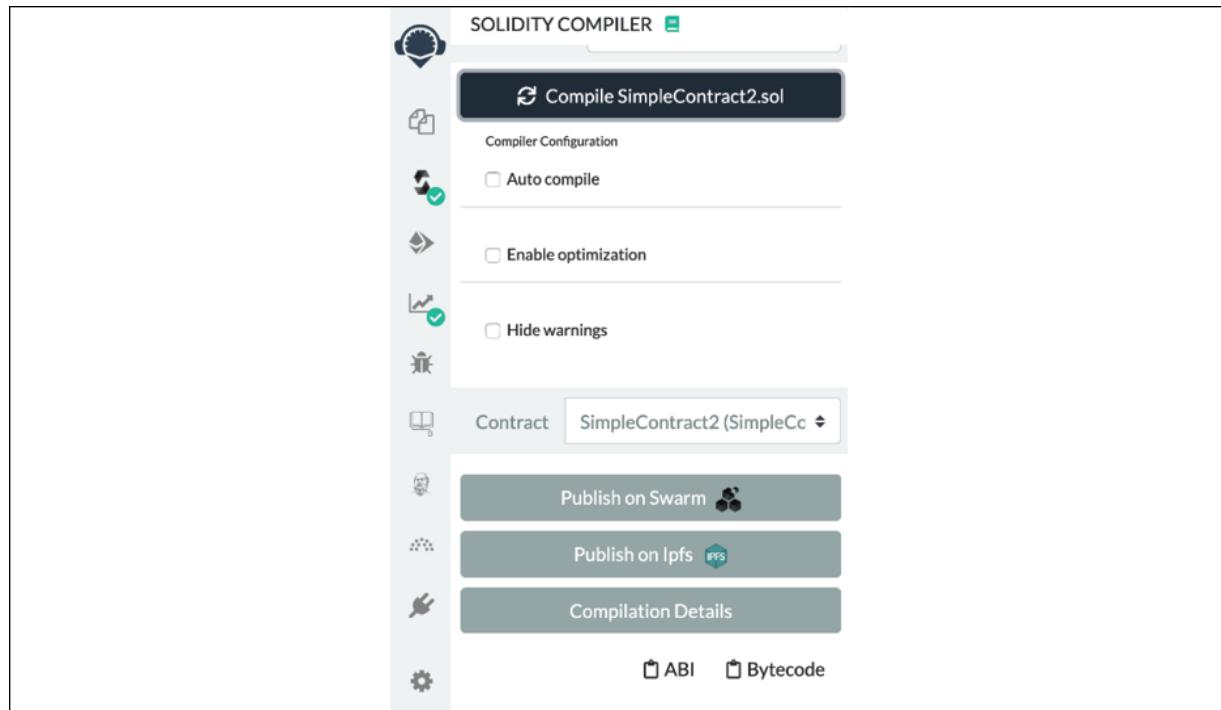


Figure 13.30: The Compile option in Remix

Once compiled successfully, we deploy the smart contract in the private net:

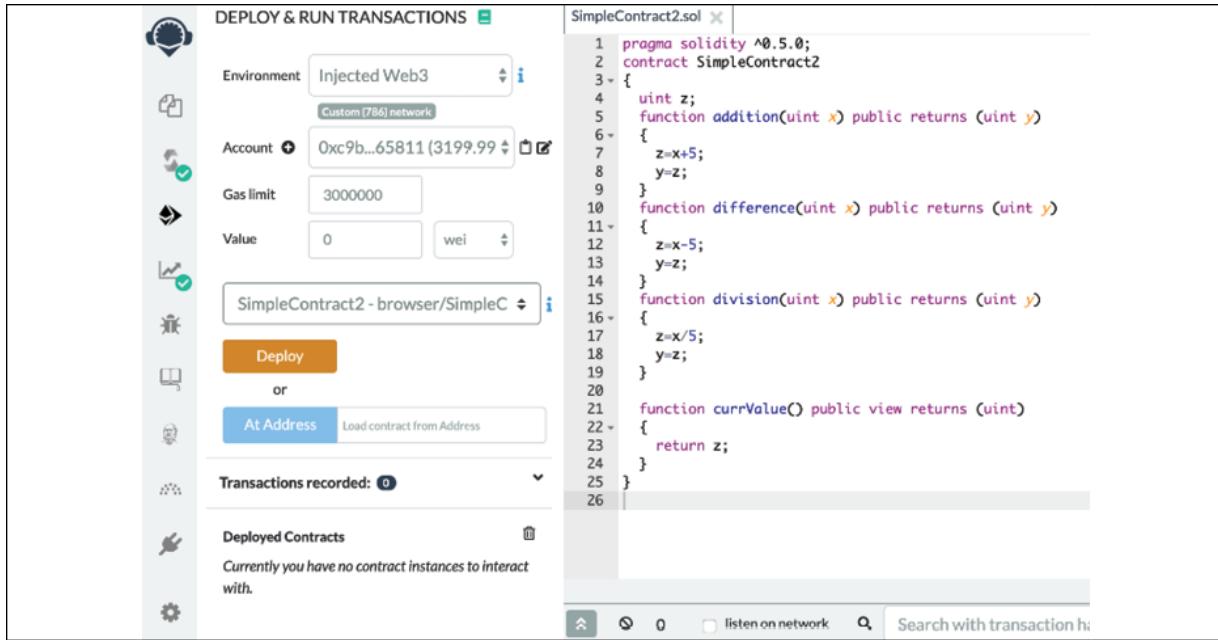


Figure 13.31: Smart contract deployment

When we click **Deploy**, the following window is displayed (shown on the left-hand side). Also, in the **DATA** tab, you can see the contract code (shown on the right-hand side).

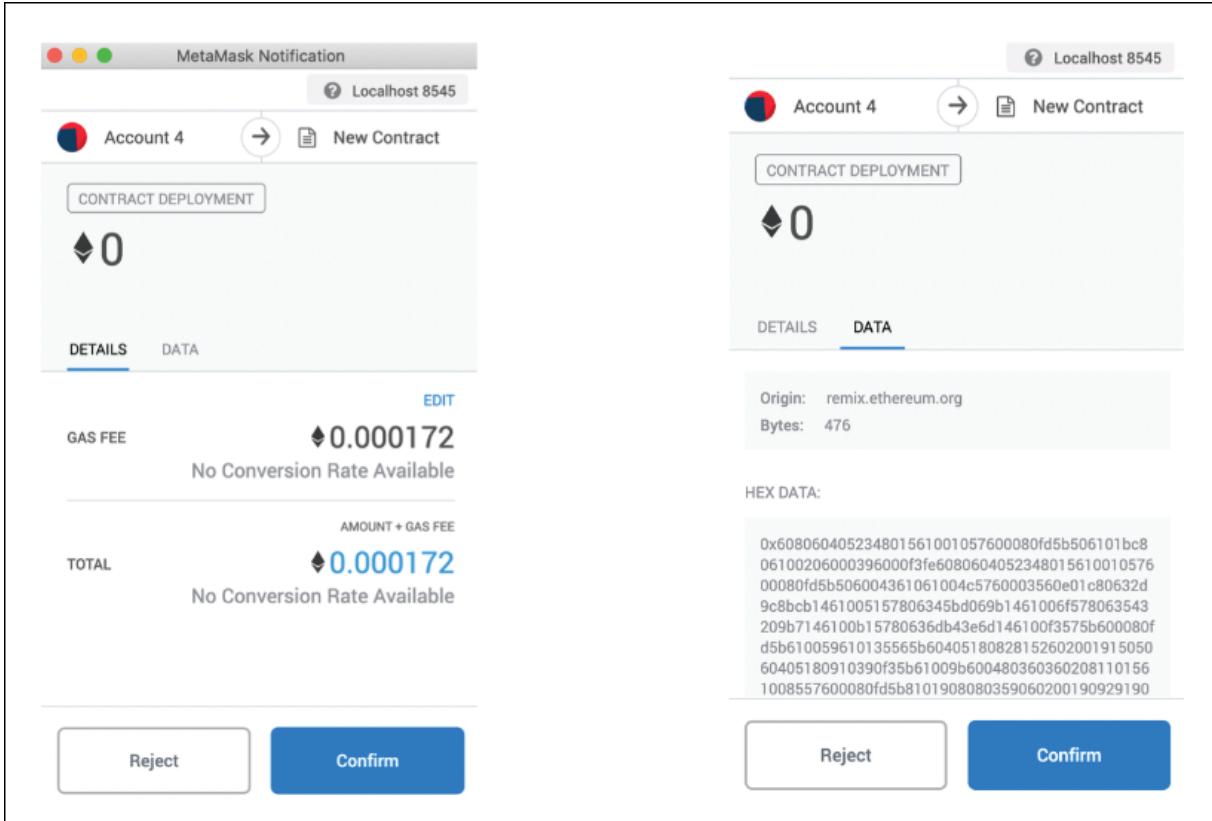


Figure 13.32: MetaMask contract deployment

Notice how in the Geth logs, we see `Submitted contract creation`, which means that a contract creation transaction has been submitted and acknowledged as a result of the deploy action from the Remix IDE and the **Confirm** action from MetaMask. Note that it also shows the full hash and contract address.

```
INFO [01-25|20:03:54.560] Setting new local account
address=0xc9Bf76271b9E42E4bF7E1888e0F52351bDb65811
INFO [01-25|20:03:54.562] Submitted contract creation
fullhash=0x626b57a4f2661587ffe0ea0342029ad3cdc59b2f1e21a573f06f9
```

Now we start mining again, by issuing the following command in the JavaScript console of Geth. If mining is already running, this step is not required.

```
> miner.start()
null
```

Once the contract is mined and deployed, you can see it in Remix, under the **Deployed Contracts** view:

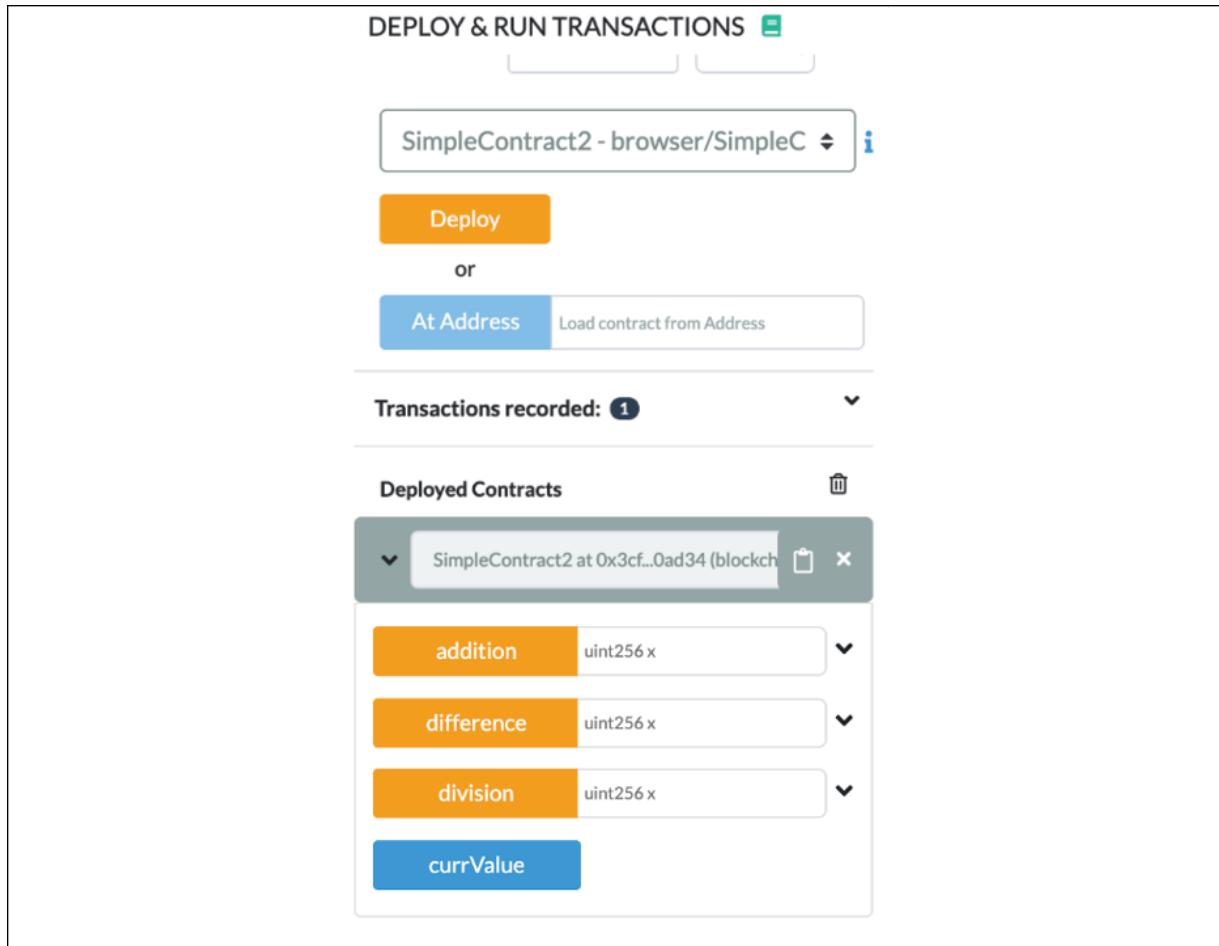


Figure 13.33: MetaMask deployed contract

With this, we have successfully deployed our example smart contract on the private network. Next, we'll see how we can interact with this contract using Remix IDE and MetaMask.

Interacting with a contract through MetaMask using Remix IDE

Now we can interact with the contract using MetaMask. We run the **addition** option, enter a value of `100` in the box, and click on the **addition** button, as shown in the following screenshot, on the left-hand side. This will invoke the MetaMask window, as shown in the following screenshot, on the right-hand side.

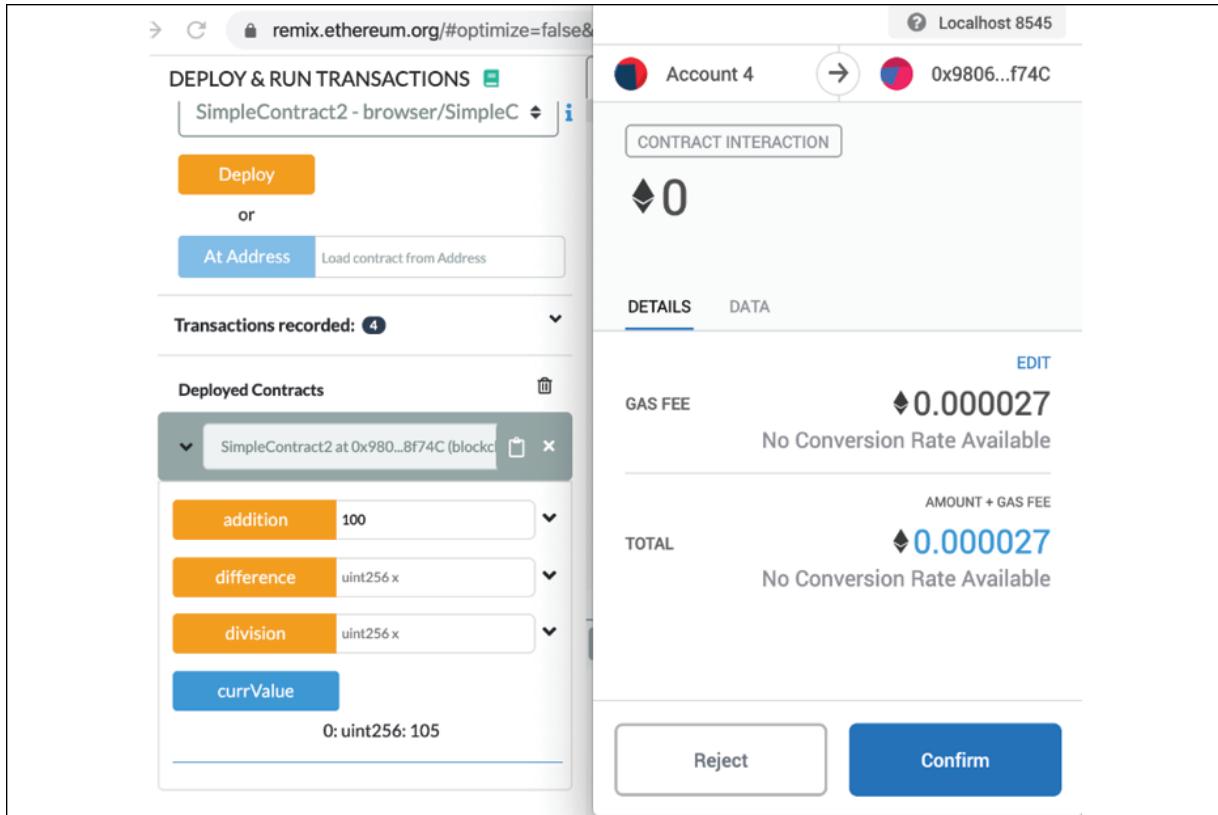


Figure 13.34: MetaMask contract interaction

Click **Confirm**, and the transaction will be mined as usual because we have a miner running the background in Geth.

Now click on the **currValue** button, which will read the contract to find the current value, which is, as expected, `105`. This value is the result of the addition operation being performed.

Recall that we provided a value of `100` and in the smart contract we have already hardcoded `5` in the `addition` function, as shown here.

```

function addition(uint x) public returns (uint y)
{
    z=x+5;
    y=z;
}

```

This means that `100` provided by us as input is added to `5`, which is hardcoded in the smart contract `addition` function, and the result of this calculation is `105`, as expected.



Figure 13.35: Retrieve the current value from the smart contract

This is a very simple program, but the aim of this exercise is to demonstrate the use of MetaMask and how it can connect with the local private chain using the `localhost` RPC provider running in Geth on port `8545`.

Pay special attention to the Solidity compiler settings. Choose the appropriate compiler version and the EVM version according to your source code and your Geth client's EVM version. If this is not selected appropriately, you can run into issues. For example, if an incorrect version of EVM (as shown in the following screenshot) is chosen, then when interacting with the contract, you may see the error message shown in the second screenshot with the caption **Gas estimation failed** error.

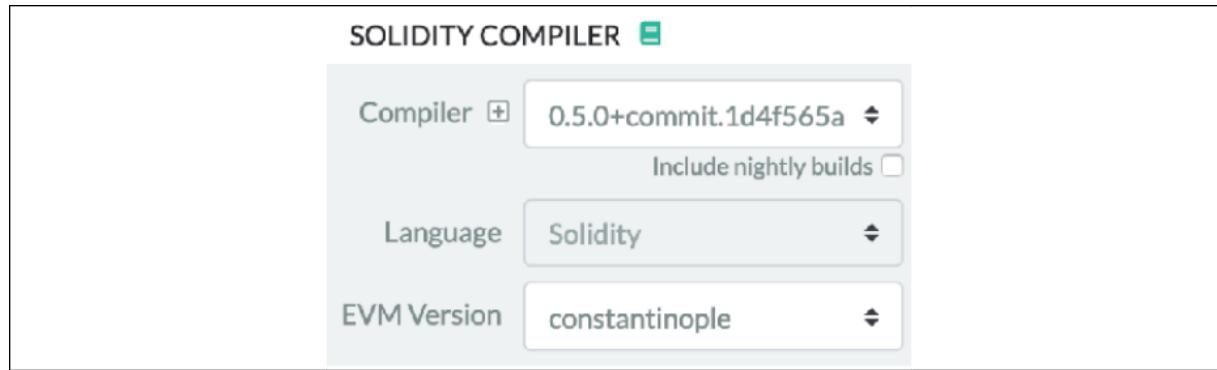


Figure 13.36: Remix compiler and EVM options



Figure 13.37: Gas estimation failed error

Similarly, if an incorrect EVM version is chosen, then you will also see an error message in the Remix window, as shown in the following screenshot.

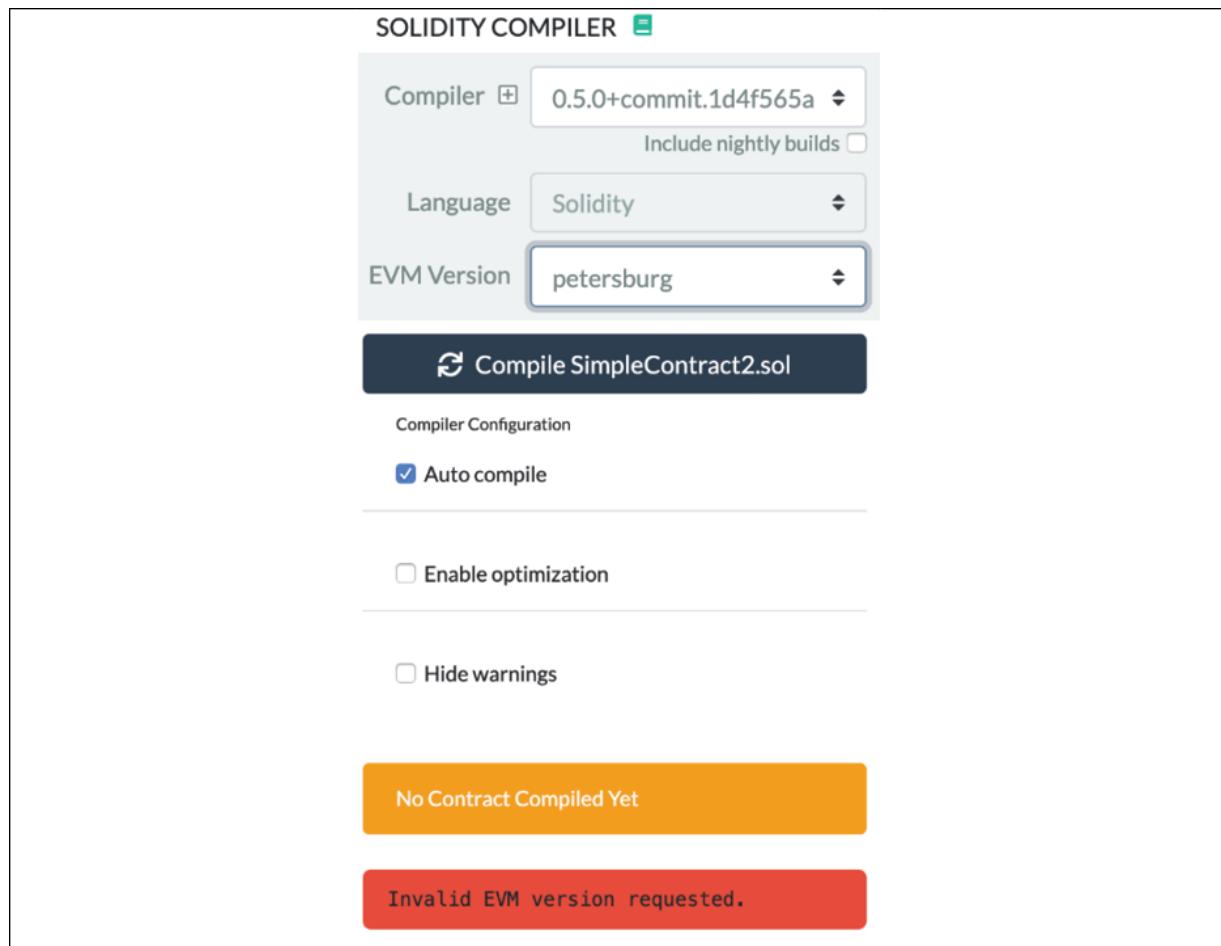
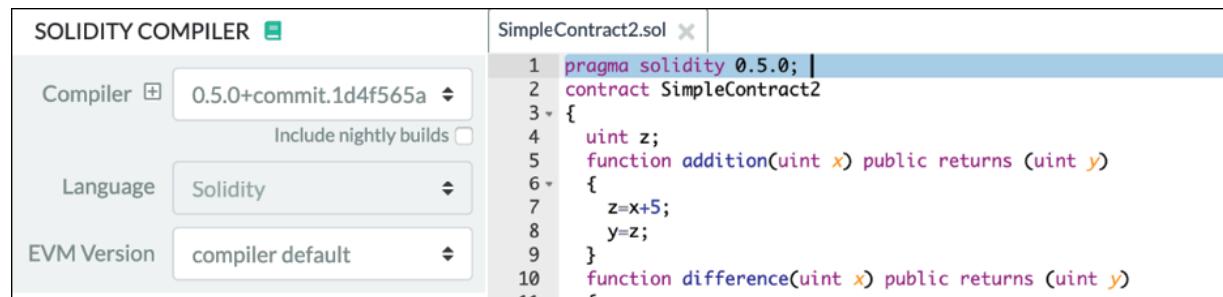


Figure 13.38: Invalid EVM version requested

It's best to configure the settings correctly, which can be left as defaults in most cases, unless there are specific features of an EVM version that you are testing.

Compiling for the wrong version of the EVM can result in incorrect and failing behavior. For example, if you have chosen the EVM version as **petersburg** (as shown in *Figure 13.38*) and you've used some opcodes that were introduced in Istanbul (a later release), then even if the compilation is successful, the bytecode at runtime will fail to run correctly and will report an error saying that the `opcode is invalid`. In some other cases, the runtime may even fail silently or with little indication in the Geth logs about what happened, leading to hours of wasted time in debugging. This is especially true in private networks, therefore make sure that the correct EVM version that matches with the Geth client release is used.

Also notice in the following screenshot that the compiler version is set to 0.5.0 automatically. This is due to the compiler version specified in Solidity source code as 0.5.0, in the line `pragma solidity 0.5.0`. Also notice the EVM version, which is set to default, which is the latest EVM release.



The screenshot shows the Solidity Compiler interface in Remix. On the left, there are settings for the Compiler (version 0.5.0+commit.1d4f565a), Language (Solidity), and EVM Version (compiler default). On the right, the Solidity code for `SimpleContract2.sol` is displayed:

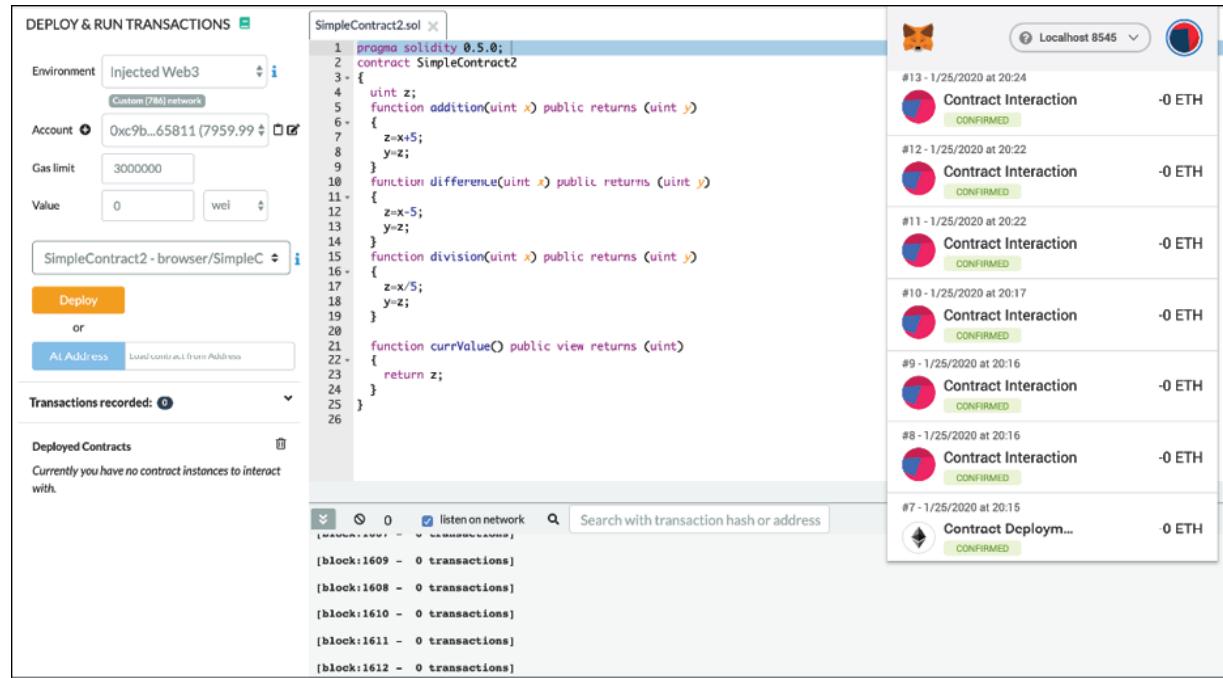
```

1 pragma solidity 0.5.0;
2 contract SimpleContract2
3 {
4     uint z;
5     function addition(uint x) public returns (uint y)
6     {
7         z=x+5;
8         y=z;
9     }
10    function difference(uint x) public returns (uint y)
11    {
12        z=x-5;
13        y=z;
14    }
15    function division(uint x) public returns (uint y)
16    {
17        z=x/5;
18        y=z;
19    }
20    function currValue() public view returns (uint)
21    {
22        return z;
23    }
24 }
25

```

Figure 13.39: Remix EVM version and compiler settings

In the MetaMask view, you can also see the transaction history of the contract.



The screenshot shows the MetaMask interface. On the left, the "DEPLOY & RUN TRANSACTIONS" window is open, showing deployment details for `SimpleContract2.sol`. The code is identical to the one in Figure 13.39. On the right, the transaction history is listed under "localhost 8545". It shows several confirmed interactions with the contract, starting from transaction #7 on January 25, 2020, at 20:15, up to transaction #13 at 20:24. The history includes Deploy, Contract Interaction, and Contract Deploy... entries.

Transaction ID	Date	Type	Value
#13	1/25/2020 at 20:24	Contract Interaction	-0 ETH
#12	1/25/2020 at 20:22	Contract Interaction	-0 ETH
#11	1/25/2020 at 20:22	Contract Interaction	-0 ETH
#10	1/25/2020 at 20:17	Contract Interaction	-0 ETH
#9	1/25/2020 at 20:16	Contract Interaction	-0 ETH
#8	1/25/2020 at 20:16	Contract Interaction	-0 ETH
#7	1/25/2020 at 20:15	Contract Deploy...	0 ETH

Figure 13.40: Transaction history in MetaMask

The preceding diagram shows a number of different elements. On the left-hand side, we have the **DEPLOY & RUN TRANSACTIONS** window,

which is part of the Remix IDE. The middle window is where we see the contract's source code. The window just below that is a log window where the result of the transactions and any interaction and results are shown. It can also optionally listen to the network, if **listen on network** is selected.

On the right-hand side, we have the MetaMask windows, which is not part of the Remix IDE, but is just superimposed on top of the Remix IDE to show the MetaMask transaction history.

Another way to connect to the local privatenet (Geth node) is to directly connect using the Web3 provider. This will allow direct communication with the blockchain through Remix via RPC without requiring an injected Web3 environment.

One advantage of this method is that it is a quick and easy method to connect Remix with the underlying blockchain (Geth node). There is no need for any additional tools, all you need is Remix IDE (which is also browser-based) and your local running Geth node. The disadvantage, however, is that this method is not very secure.

This can be achieved using the following option, available in Remix:

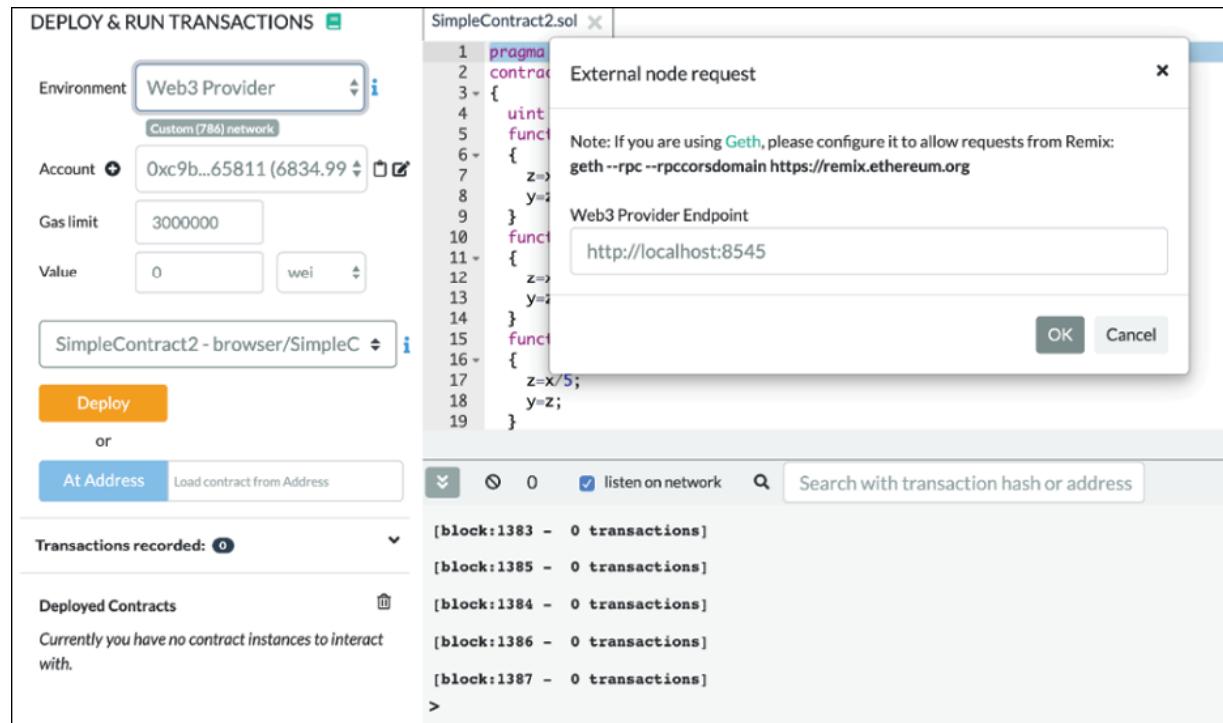


Figure 13.41: Remix Web3 provider

Notice in the preceding screenshot that **Environment** is set to **Web3 Provider**, which simply connects to `http://localhost:8545`.

Notice that our `geth` process is running with the required parameters already, which allows Web3 provider connection, as shown here.

```
$ geth --datadir ~/etherprivate/ --networkid 786 --rpc --rpccapi
```

Notice that in the preceding command, we have the `--rpccorsdomain` flag set. If we didn't have this flag set, then the only option would have been to connect via injected Web3, which in fact is a more secure way to connect to the chain using MetaMask. As such, we use that as a standard, but if you have a simple local test network, you can always use the **Web3 Provider** option to quickly interact with the blockchain using Remix IDE.



Do not use the `--rpccorsdomain` flag with `*` on public networks especially if you have ETH and/or private keys stored locally on your node. This flag used with `*` will allow all remote connections to your Ethereum node via the exposed RPC endpoint, which can result in security breaches.

Using the flag `-rpccorsdomain` with the wildcard (`--rpccorsdomain '*'`) is not advisable. Using the wildcard `*` will allow everyone to connect to the node. Therefore, it is recommended that the access is restricted by specifying only trusted URLs. For example, `--rpccorsdomain 'https://remix.ethereum.org'`. However, `--rpccorsdomain` with the wildcard `*` can be used with a local test chain with test accounts. Always specify the URL in the case of real accounts or on the mainnet.

Now, we can stop the miner:

```
> miner.stop()
null
```

With this example, we have now covered how MetaMask can be used to connect to the local network and how Remix and MetaMask, or only Remix, can be used to connect to the local network and deploy smart contracts. We also saw how to interact with a deployed contract.

Remix is a very feature-rich IDE and we have not covered everything; however, this chapter has included a formal introduction to Remix, describing most of the main features in detail, and we will keep using it in the chapters to come.

It is also useful to have a mechanism to view a consolidated list and details of all the transactions in the blockchain. For this purpose, block explorers are used. There are many services available for public blockchains on the internet. For more details on how to use a local block explorer go to the companion website of this book at https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

Summary

In this chapter, we have explored Ethereum test networks and how to set up private Ethereum networks. After the initial introduction to private network setup, we also saw how the Geth command-line tool can be used to perform various functions and how we can interact with the Ethereum blockchain. We also saw how MetaMask and Remix can be used to deploy smart contracts. We also introduced how block explorers play a vital role in monitoring the blockchain networks and saw a basic example of installing and using an open source block explorer.

In the next chapter, we will see in greater detail what tools, programming languages, and frameworks are available for the development of smart contracts on Ethereum.

14

Development Tools and Frameworks

This chapter is an introduction to the development tools, languages, and frameworks used for Ethereum smart contract development. We will examine different methods of developing smart contracts for the Ethereum blockchain. We will discuss various constructs of the **Solidity** language in detail, which is currently the most popular development language for smart contract development on Ethereum.

In this chapter, we will cover the following topics:

- Languages
- Compilers
- Tools and libraries
- Frameworks
- Contract development and deployment
- The layout of a Solidity source code file
- The Solidity language

There are a number of tools available for Ethereum development. The following diagram shows the taxonomy of various development tools, clients, IDEs, and development frameworks for Ethereum:

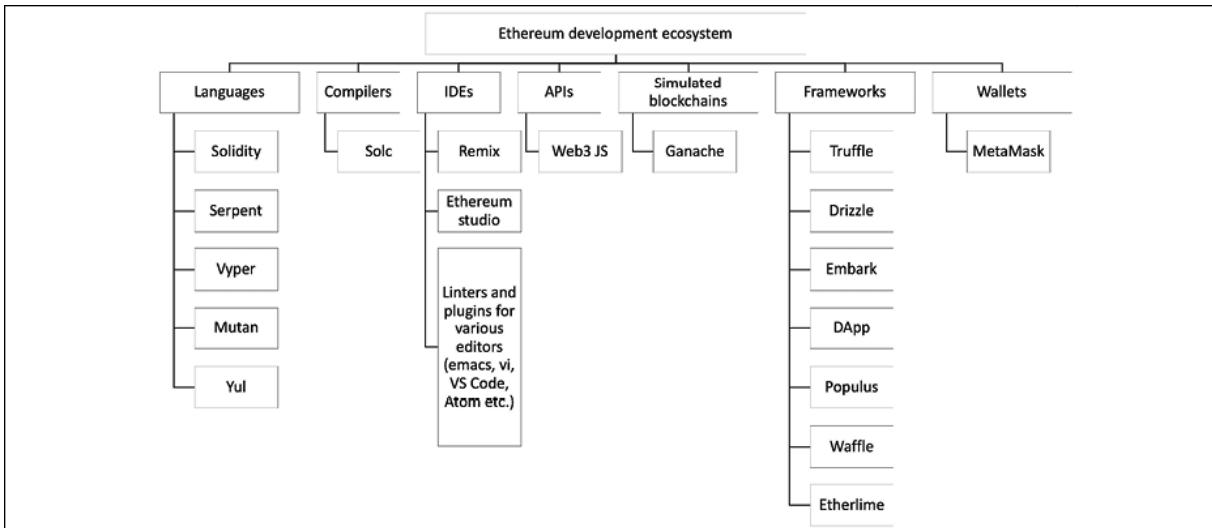


Figure 14.1: Taxonomy of Ethereum development ecosystem components

The preceding taxonomy does not include all frameworks and tools that are out there for development on Ethereum. It shows the most commonly used tools and frameworks, including some that we will use in our examples throughout this chapter.



There are a number of resources available related to development tools for Ethereum at the following address:
<http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#developer-tools>

Since we have discussed some of the main tools available in Ethereum in previous chapters, such as the Remix IDE and MetaMask, this chapter will focus mainly on Geth, Solidity, Ganache, `solc`, and Truffle. Some other elements, such as prerequisites (Node.js), will also be discussed briefly.

We'll start by exploring some of the programming languages that can be used in the Ethereum blockchain.

Languages

Smart contracts can be programmed in a variety of languages for the Ethereum blockchain. There are six main languages that can be used to write contracts:

- **Solidity**: This language has now become almost a standard for contract writing for Ethereum. This language is the focus of this chapter and is discussed in detail in later sections.
- **Vyper**: This language is a Python-like experimental language that is being developed to bring security, simplicity, and auditability to smart contract development.
- **Yul**: This is an intermediate language that has the ability to compile to different backends such as EVM and eWasm. The design goals of Yul mainly include readability, easy control flow, optimization, formal verification, and simplicity.
- **Mutan**: This is a Go-style language, which was deprecated in early 2015 and is no longer used.
- **LLL**: This is a **Low-Level Lisp-Like Language**, hence the name LLL. This is also no longer used.
- **Serpent**: This is a simple and clean Python-like language. It is not used for contract development anymore and is not supported by the community.

As Solidity code needs to be compiled into bytecode, we need a compiler to do so. In the next section, we will introduce the Solidity compiler.

Compilers

Compilers are used to convert high-level contract source code into the format that the Ethereum execution environment understands. The Solidity compiler, `solc`, is the most common one in use, which is discussed next.

The Solidity compiler

`solc` converts from a high-level Solidity language into **Ethereum Virtual Machine (EVM)** bytecode so that it can be executed on the blockchain by the EVM.

Installation

`solc` can be installed on a Linux Ubuntu operating system using the following commands:

```
$ sudo apt-get install solc
```

If **Personal Package Archives (PPAs)** are not already installed, those can be installed by running the following commands:

```
$ sudo add-apt-repository ppa:ethereum/ethereum  
$ sudo apt-get update
```

To install `solc` on macOS, execute the following commands:

```
$ brew tap ethereum/ethereum
```

This command will add the Ethereum repository to the list of `brew` formulas:

```
$ brew install solidity
```

This command will produce a long output and may take a few minutes to complete. If there are no errors produced, then eventually it will install Solidity.

In order to verify that the Solidity compiler is installed and to validate the version of the compiler, the following command can be used:

```
$ solc --version
```

This command will produce the output shown as follows, displaying the version of the Solidity compiler.

```
solc, the solidity compiler commandline interface
Version: 0.6.1+commit.e6f7d5a4.Darwin.appleclang
```

This output confirms that the Solidity compiler is installed successfully.

Functions

`solc` supports a variety of functions. A few examples are shown as follows:

As an example, we'll use a simple contract, `Addition.sol`:

```
pragma solidity ^0.5.0;
contract Addition
{
    uint8 x;
    function addx(uint8 y, uint8 z) public
    {
        x = y + z;
    }
    function retrievex() view public returns (uint8)
    {
        return x;
    }
}
```

Displaying the contract in a binary format

In order to see the smart contract in compiled binary format, we can use the following command:

```
$ solc --bin Addition.sol
```

This command will produce an output similar to the following:

```
===== Addition.sol:Addition =====
Binary:
608060405234801561001057600080fd5b50610100806100206000396000f3fe
```

This output shows the binary translation of the `Addition.sol` contract code represented in hex.

Estimating gas

As a gas fee is charged for every operation that the EVM executes, it's a good practice to estimate gas before deploying a contract on a live network. Estimating gas gives a good approximation of how much gas will be consumed by the operations specified in the contract code, which gives an indication of how much ether is required to be spent in order to run a contract. We can use the `--gas` flag for this purpose, as shown in the following example:

```
$ solc --gas Addition.sol
```

This will give the following output:

```
===== Addition.sol:Addition =====
Gas estimation:
construction:
    99 + 51200 = 51299
external:
    addx(uint8,uint8): 21120
    retrievex(): 1061
```

This output shows how much gas usage is expected for these operations in the `Addition.sol` smart contract. The gas estimation is shown next to each function. For example, the `addx()` function is expected to use `21120` gas.

Generating the ABI

We can generate the **Application Binary Interface (ABI)** using `solc`, which is a standard way to interact with the contracts:

```
$ solc --abi Addition.sol
```

This command will produce a file named `Addition.abi` as output. The following are the contents of the output file `Addition.abi`:

```
===== Addition.sol:Addition =====
Contract JSON ABI
[{"inputs": [{"internalType": "uint8", "name": "y", "type": "uint8"}], {
```

The preceding output displays the contents of the `Addition.abi` file, which are formatted in JSON style. It consists of inputs and outputs along with their types. We will generate and use ABIs later in this chapter to interact with the deployed smart contracts.

Compilation

Another useful command to compile and produce a binary compiled file along with an ABI is shown here:

```
$ solc --bin --abi -o bin Addition.sol
```

The message displays if the compiler run is successful, otherwise errors are reported.

```
Compiler run successful. Artifact(s) can be found in directory bin
```

This command will produce a message and two files in the output directory `bin`:

- `Addition.abi`: This contains the ABI of the smart contract in JSON format.
- `Addition.bin`: This contains the hex representation of binary of the smart contract code.

The output of both files is shown in the following screenshot:

```
[+] ↗ bin cat Addition.bin
[608060405234801561801057600080fd5b50610100806100206000396000f3fe6080604052348015600f57600080fd5b506004361060325760]
[003560e01c806336718d80146037578063ac04e0a0146072575b600080fd5b60760048036036040811015604b57600080fd5b810190808035
[60ff169060200190929190803560ff1690602001909291905050506094565b005b607860b4565b604051808260ff1660ff1681526020019150
[5060405180910390f35b8082016000806101000a81548160ff021916908360ff16021790555050565b60008060009054906101000a900460
[ff1690509056fea2646970667358221220bae3e7dea36338ad4ced23dee3370621e38b08377e773854fcc1b22260924d64736f6c63430006
[010033] ↗
[+] ↗ bin cat Addition.abi
[{"inputs":[{"internalType":"uint8","name":"y","type":"uint8"}, {"internalType":"uint8","name":"z","type":"uint8"}], "name":"addx", "outputs":[], "stateMutability":"nonpayable", "type":"function"}, {"inputs":[],"name":"retrievevex", "outputs":[{"internalType":"uint8","name":"","type":"uint8"}], "stateMutability":"view", "type":"function"}] ↗
[+] ↗ bin
```

Figure 14.2: ABI and binary output of the solidity compiler



The ABI encodes information about smart contracts' functions and events. It acts as an interface between EVM-level bytecode and high-level smart contract program code. To interact with a smart contract deployed on the Ethereum blockchain, external programs require an ABI and the address of the smart contract.

`solc` is a very powerful command and further options can be explored using the `--help` flag, which will display detailed options. However, the preceding commands used for compilation, ABI generation, and gas estimation should be sufficient for most development and deployment requirements.

Tools and libraries

There are various tools and libraries available for Ethereum. The most common ones are discussed here. In this section, we will first install the prerequisites that are required for developing applications for Ethereum.

The first requirement is Node.js, which we'll install next.

Node.js

As Node.js is required for most of the tools and libraries, it can be installed using the following commands.

Node.js can either be installed directly from the website or by using `bash`:

```
$ curl "https://nodejs.org/dist/latest/node-$VERSION:$ (wget -c
```

Alternatively, Node.js can be installed using `brew` on macOS:

```
$ brew install node
```

Or, go to <https://nodejs.org/en/download/> and install Node.js for your operating system, as shown in the following screenshot:

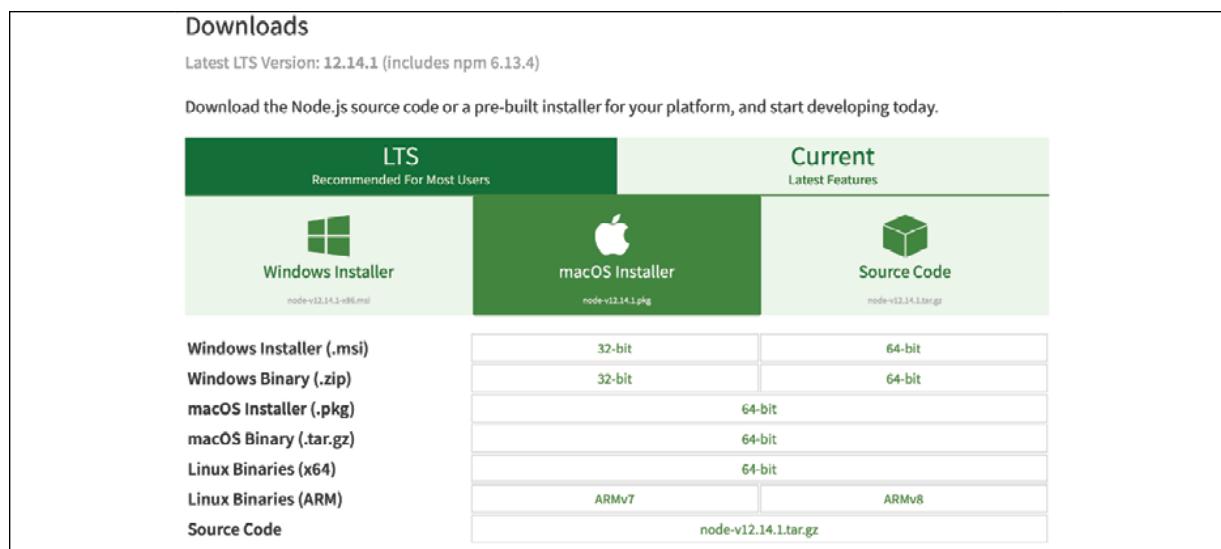


Figure 14.3: Downloads available for Node.js

Ganache CLI

At times, it is not possible to test on the testnet, and the mainnet is obviously not a place to test contracts. A private network can be time-consuming to set up at times. Ganache CLI (formerly EthereumJS) comes in handy when quick testing is required and no testnet is available. It simulates the Ethereum `geth` client behavior and allows faster

development testing. Ganache CLI is available via `npm` as a Node.js package.

Before installing TestRPC, Node.js should already have been installed and the `npm` package manager should also be available.

`ganache-cli` can be installed using this command:

```
$ npm install -g ganache-cli
```

In order to start `ganache-cli`, simply issue this command, keep it running in the background, and open another terminal to work on contracts:

```
$ ganache-cli
```

When TestRPC runs, it will display an output similar to the one shown in the following screenshot. It will automatically generate 10 accounts and private keys, along with an HD wallet. It will start to listen for incoming connections on TCP port `8545`:

```

Ganache CLI v6.9.0 (ganache-core: 2.10.1)
eth_blockNumber
eth_blockNumber
eth_blockNumber
eth_blockNumber
eth_blockNumber
eth_blockNumber
eth_blockNumber

Available Accounts
=====
(0) 0xFE65a55F165AF41EaD742fE3e1C0c7A7dF904638 (100 ETH)
(1) 0x3f50Ba4b7192a501879b425d799c8ED9a9dC2D6 (100 ETH)
(2) 0x2A18346D9A3E74577c79868f46042fD2cDDdC471 (100 ETH)
(3) 0xBaf7bea3c3cBf2E229760C287Cdb502d88b6 (100 ETH)
(4) 0xa0739d43E565012276a9866e30eC2B8A2cfa6f77 (100 ETH)
(5) 0x2b1e707ea8E0431b0805edc1732913a258Ebba0ab (100 ETH)
(6) 0xd5324EC46B1a1f07d7dE6e7181D6a8F9E991046A (100 ETH)
(7) 0x18e034b635fa8819dACACd1c2d745BB86378c0 (100 ETH)
(8) 0x15C9fD2548ee0dEaF69D3De244aE1A4Bc3d48Ea (100 ETH)
(9) 0x0eF9150417c473288396A0597e467fA76C2b3e4a (100 ETH)

Private Keys
=====
(0) 0x6609618a07c98b30b75179fc71044b4696c173f834550d51c8dbe07711629b5
(1) 0x4fb761fe0ccf4b9788c280a377d1511dd8dc6e1f656c3e2a252513c5ea95945
(2) 0xe90ec2cd75244a395d14abe0e67692ea3b79bef3eaafa5ef25b2a54d6b27989
(3) 0xd40afda5129194a994119b914757ce0c493b3319d18e9442506ec7d5ac367748
(4) 0xd72ef0f123f90c32c67462e30886cd7aba0ddc06d24c84bc7ea04be33c69984b
(5) 0x74bd9ed1b77ef42b150ce2c2b5fc323ac2f42a5119c7173d65caf0497d62b84
(6) 0xb8b5c51227c628be0653a150933bf78c04b04c65e5ebcbcc64763b14e2d86ff29
(7) 0xaf2128fa0d4a4931f60fc4b7a4ca76a3e8b7acc623eb50d494988d8ce3ac9b63
(8) 0x51851a23ec13d100dbd3bbc857ab6765b5ab72716881dc4488024096e04827
(9) 0x4d5fec001e0d6d8559e71ca4382ed839e4ebcc26f3a6ecd2d00ff58e35d3ef8b

HD Wallet
=====
Mnemonic: real away arrow grow flag verb excite choose entry involve pudding engine
Base HD Path: m/44'/60'/0'/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

```

Figure 14.4: Ganache CLI

`ganache-cli` has a number of flags to customize the simulated chain according to your requirements. For example, flag `-a` allows you to specify the number of accounts to generate at startup. Similarly `-b` allows the user to configure block time for mining.

Detailed help is available using the following command:

```
$ ganache-cli --help
```

Ganache CLI is a command-line tool. However, at times, it is desirable to have a fully featured tool with a rich **graphical user interface (GUI)**. A GUI equivalent of Ganache CLI is Ganache, which we will introduce next.

Ganache

Ganache is a notable addition to the ever-growing set of development tools and libraries developed for Ethereum. This is a simulated personal blockchain with a user-friendly GUI to view transactions, blocks, and relevant details. This is a fully working personal blockchain that supports the simulation of a number of different hard forks, such as **Homestead**, **Byzantium**, **Istanbul**, or **Petersburg**.



Ganache is based on a JavaScript implementation of the Ethereum blockchain, with built-in block explorer and mining, making testing locally on the system very easy. As shown in the following screenshot, you can view transactions, blocks, and addresses in detail on the frontend:

A screenshot of the Ganache web interface. At the top, there's a navigation bar with links for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS, along with a search bar and some status indicators like current block (0), gas price (20000000000), gas limit (6721975), hardfork (PETERSBURG), network ID (5777), RPC server (HTTP://127.0.0.1:7545), mining status (AUTOMINING), and workspace (JAZZY-GIRL). Below the bar, a mnemonic phrase "kick abstract strong shrug forward enlist puppy reunion elephant hip suffer base" is displayed above a tree diagram labeled "HD PATH m/44'/60'/0'/0/account_index". The main area shows a table of accounts:

ADDRESS	BALANCE	TX COUNT	INDEX	Copy
0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	100.00 ETH	0	0	🔗
0x6805940005154aEdfe6a00A39C588ED668E64D9D	100.00 ETH	0	1	🔗
0x56C21294F4e17dF32486b3d4D12E72D023861edF	100.00 ETH	0	2	🔗
0xAfACDB553412071bB538E36d57ED92d6745C5f94	100.00 ETH	0	3	🔗
0x694AE93a42C43B8E3d10c3F6769ADF2566C1863B	100.00 ETH	0	4	🔗

Figure 14.5: Ganache, a personal Ethereum blockchain

When you start Ganache for the first time, it will ask whether the user wants to create a quick blockchain or create a new workspace that can be saved, and it also has advanced setup options:

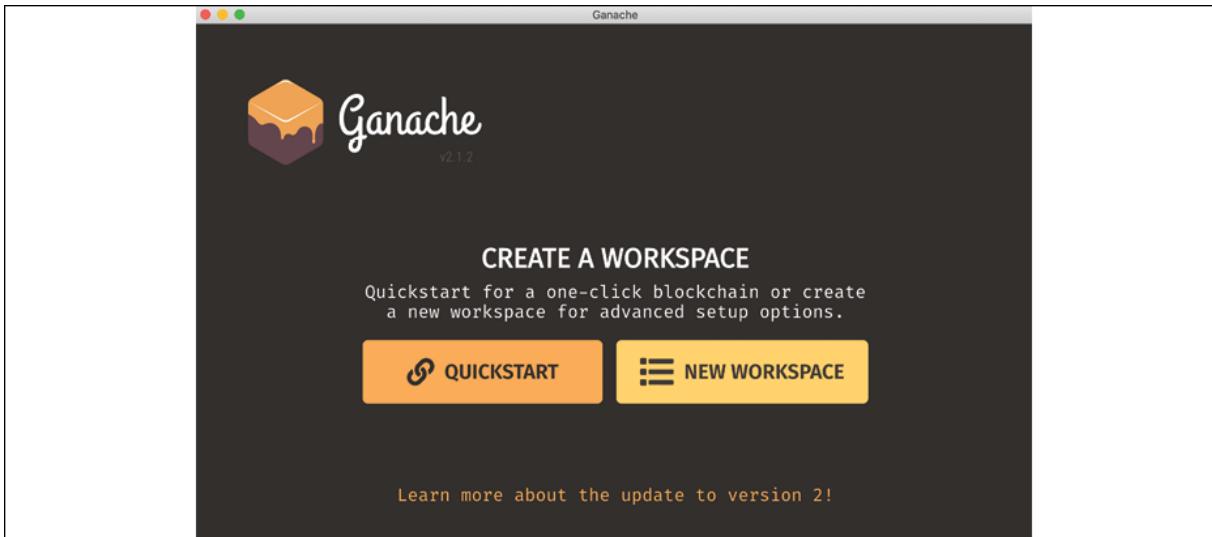


Figure 14.6: Creating a workspace

Select the **NEW WORKSPACE** or **QUICKSTART** option as required. We will choose **NEW WORKSPACE** as we want to explore more advanced features. For a quicker temporary setup with default options, which could be useful for simple testing, you can choose **QUICKSTART**.

If **NEW WORKSPACE** is selected, there are a number of options available to configure the blockchain. The configuration options are **WORKSPACE NAME**, where you can specify a name for your project. Additionally, Truffle projects can also be added here—we will cover Truffle in more detail later in the chapter:

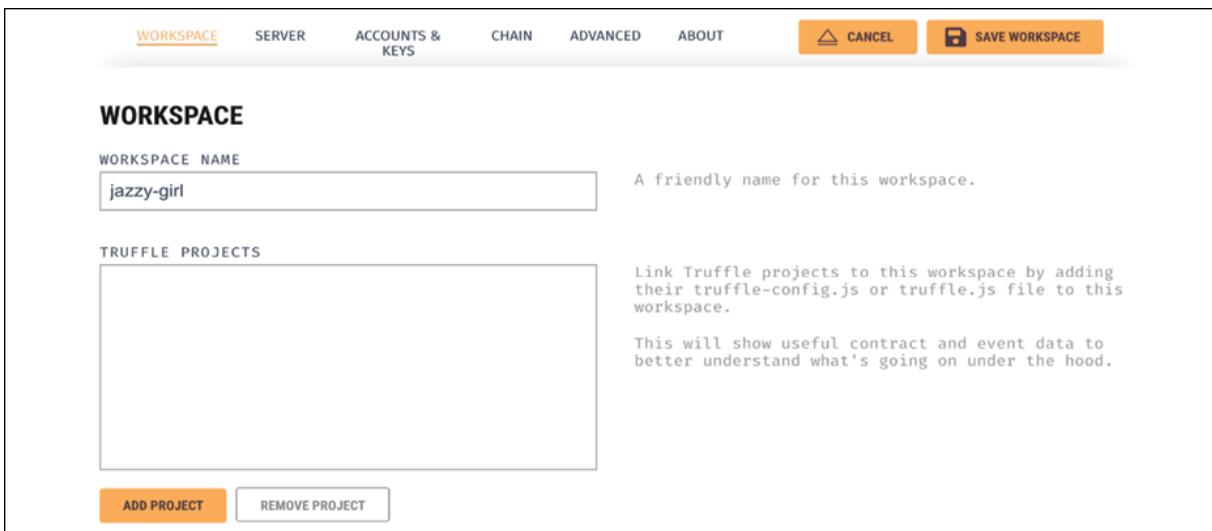


Figure 14.7: Workspace in Ganache

Other options include **SERVER**, **ACCOUNTS & KEYS**, **CHAIN**, and **ADVANCED**.

The **SERVER** tab is used to configure RPC connectivity by specifying the hostname, port number, and network ID:

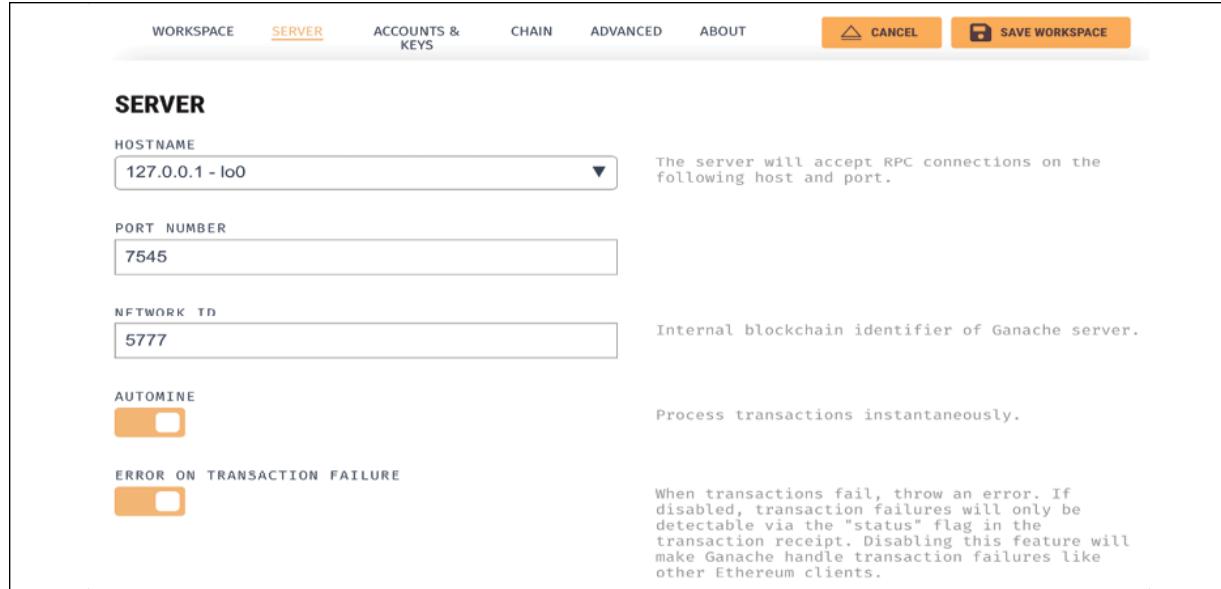


Figure 14.8: Server configuration

ACCOUNTS & KEYS provides options to configure balance and the number of accounts to generate. The **CHAIN** option provides a configuration interface for specifying the gas limit, gas price, and hard fork, which is required to be simulated, such as Byzantine or Petersburg:

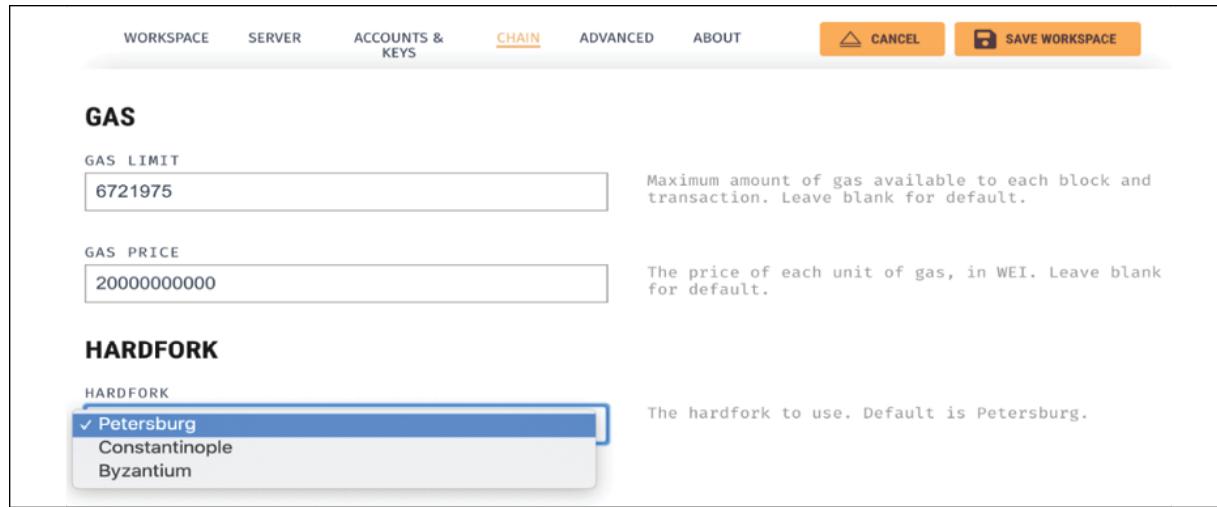


Figure 14.9: Chain configuration

The **ADVANCED** option is available to configure logging and analytics-related settings. Once you have all the configuration options set, save the workspace by selecting **SAVE WORKSPACE**, and the main transaction view of the Ganache personal blockchain will show:

ADDRESS	BALANCE	TX COUNT	INDEX	
0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	100.00 ETH	0	0	
0x6805940005154aEdfe6a00A39C588ED668E64D9D	100.00 ETH	0	1	
0x56C21294F4e17dF32486b3d4D12E72D023861edF	100.00 ETH	0	2	
0xAfACDB553412071bB538E36d57ED92d6745C5f94	100.00 ETH	0	3	
0x694AE93a42C43B8E3d10c3F6769Adf2566C1863B	100.00 ETH	0	4	
0x52eA83ae62213f076E3A93054F8168907AaB3313	100.00 ETH	0	5	

Figure 14.10: Ganache main view

With this, we conclude our introduction to Ganache, a mainstream tool used in blockchain development. Now we will move on to different development frameworks that are available for Ethereum.

Frameworks

There are a number of other notable frameworks available for Ethereum development. It is almost impossible to cover all of them, but an introduction to some of the mainstream frameworks and tools is given as follows.

Truffle

Truffle (available at <https://www.trufflesuite.com>) is a development environment that makes it easier and simpler to test and deploy Ethereum contracts. Truffle provides contract compilation and linking along with an automated testing framework using Mocha and Chai. It also makes it easier to deploy the contracts to any private net, public, or testnet Ethereum blockchain. Also, an asset pipeline is provided, which makes it easier for all JavaScript files to be processed, making them ready for use by a browser.

Before installation, it is assumed that `node` is available, which can be queried as shown here. If `node` is not available, then the installation of `node` is required first in order to install `truffle`:

```
$ node --version  
v12.14.1
```

The installation of `truffle` is very simple and can be done using the following command via **Node Package Manager (npm)**:

```
$ npm install truffle -g
```

This will take a few minutes; once it is installed, the `truffle` command can be used to display help information and verify that it is installed correctly:



```
/usr/local/bin/truffle -> /usr/local/lib/node_modules/truffle/bui  
/usr/local/lib  
└─ truffle@4.0.1
```

Type `truffle` in the Terminal to display usage help:

```
$ truffle
```

This will display the following output:

```
Truffle v5.1.11 - a development framework for Ethereum

Usage: truffle <command> [options]

Commands:
  build      Execute build pipeline (if configuration present)
  compile    Compile contract source files
  config     Set user-level configuration options
  console    Run a console with contract abstractions and commands available
  create     Helper to create new contracts, migrations and tests
  debug      Interactively debug any transaction on the blockchain (experimental)
  deploy     (alias for migrate)
  develop    Open a console with a local development blockchain
  exec       Execute a JS module within this Truffle environment
  help       List all commands or provide information about a specific command
  init       Initialize new and empty Ethereum project
  install   Install a package from the Ethereum Package Registry
  migrate   Run migrations to deploy contracts
  networks  Show addresses for deployed contracts on each network
  obtain    Fetch and cache a specified compiler
  opcode    Print the compiled opcodes for a given contract
  publish   Publish a package to the Ethereum Package Registry
  run       Run a third-party command
  test      Run JavaScript and Solidity tests
  unbox    Download a Truffle Box, a pre-built Truffle project
  version   Show version number and exit
  watch     Watch filesystem for changes and rebuild the project automatically

See more at http://truffleframework.com/docs
```

Figure 14.11: Truffle help

Alternatively, the repository is available at <https://github.com/trufflesuite/truffle>, which can be cloned locally to install `truffle`. **Git** can be used to clone the repository using the following command:

```
$ git clone https://github.com/trufflesuite/truffle.git
```

We will use Truffle later in *Chapter 15, Introducing Web3*, to test and deploy smart contracts on the Ethereum blockchain. For now, we'll continue

to explore some of the frameworks used for development on the Ethereum blockchain.

Drizzle

Web **User Interface (UI)** development is an important part of DApp development. As such, many web techniques and tools, ranging from simple HTML and JavaScript to advanced frameworks such as Redux and React, are used to develop web UIs for DApps.

We will cover basic JavaScript- and HTML-based UIs, along with an advanced framework called **Drizzle**, in the upcoming chapters.

Drizzle is a collection of frontend libraries that allows the easy development of web UIs for decentralized applications. It is based on the Redux store and allows seamless synchronization of contract and transaction data.

Drizzle is installed using the following command:

```
$ npm install drizzle --save
```

In *Chapter 15, Introducing Web3*, we will explore Drizzle in detail and use it in some examples.

Embark

Embark is a complete and powerful developer platform for building and deploying decentralization applications. It is used for smart contract development, configuration, testing, and deployment. It also integrates with **Swarm**, **IPFS**, and **Whisper**. There is also a web interface called **Cockpit** available with Embark, which provides an integrated development environment for easy development and debugging of decentralized applications.

Embark can be installed by using the following command:

```
$ npm install -g embark
```

More details on Embark are available on the official website at <https://framework.embarklabs.io>.

Brownie

Brownie is a Python-based framework for Ethereum smart contract development and testing. It has the full support of Solidity and Vyper with relevant testing and debugging tools. More information is available at <https://eth-brownie.readthedocs.io/en/stable/>.

Waffle

Waffle is a framework for smart contract development and testing. It is claimed to be faster than Truffle. More details are available on the official website at <https://getwaffle.io>.

Etherlime

This framework allows DApp development, debugging, testing, and testing in Solidity and Vyper. It is based on `Ether.js` (<https://github.com/ethers-io/ethers.js/>).

OpenZeppelin

This toolkit has a rich set of tools that allow easy smart contract development. It supports compiling, deploying, upgrading, and interacting with smart contracts. Further information is available here: <https://openzeppelin.com/sdk/>.

In this section, we have covered some of the mainstream frameworks that are used in the Ethereum ecosystem for development. In the next section, we will explore which tools are available for writing and deploying smart contracts.

Contract development and deployment

There are various steps that need to be taken in order to develop and deploy the contracts. Broadly, these can be divided into three steps: writing, testing, and deployment. After deployment, the next optional step is to create the UI and present it to the end users via a web server. A web interface is sometimes not needed in the contracts where no human input or monitoring is required, but usually there is a requirement to create a web interface so that end users can interact with the contract using familiar web-based interfaces.

Writing smart contracts

The writing step is concerned with writing the contract source code in Solidity. This can be done in any text editor. There are various plugins and add-ons available for Vim in Linux, Atom, and other editors that provide syntax highlighting and formatting for Solidity source code.

Visual Studio Code has become quite popular and is used commonly for Solidity development. There is a Solidity plugin available that allows syntax highlighting, formatting, and **IntelliSense**.

It can be installed via the **Extensions** option in Visual Studio Code:

The screenshot shows a Visual Studio Code window with the following details:

- File Explorer:** On the left, it shows "OPEN EDITORS 1 UNSAVED" with "Patent.sol" listed. Below that are sections for "ETHEREUM" and "node_modules", also containing "Patent.sol".
- Code Editor:** The main area displays Solidity code for a contract named "PatentIdea". The code includes functions for saving and checking hashed ideas.
- Status Bar:** At the bottom, it shows "Ln 14, Col 13" and "Spaces: 2". It also indicates "UTF-8 LF" and "Solidity". There are icons for file operations (Save, Undo, Redo) and a gear icon.

```
Patent.sol — ethereum
1 pragma solidity ^0.4.0;
2 contract PatentIdea {
3     mapping (bytes32 => bool) private hashes;
4     bool alreadyStored;
5     event IdeaHashed(bool);
6
7     function saveHash(bytes32 hash) private {
8         hashes[hash] = true;
9     }
10    function SaveIdeaHash(string idea) public returns (bool){
11        var hashedIdea = HashtheIdea(idea);
12        if (alreadyHashed(HashtheIdea(idea))) {
13            alreadyStored=true;
14            IdeaHashed(false);
15        } else {
16            SaveIdeaHash
17            saveHash(hashedIdea);
18            ideaHashed(true);
19        }
20    }
21
22    function alreadyHashed(bytes32 hash) constant private returns(bool) {
23        return hashes[hash];
24    }
25
26    function isAlreadyHashed(string idea) constant public returns (bool) {
27        var hashedIdea = HashtheIdea(idea);
28        return alreadyHashed(hashedIdea);
29    }
30
31    function HashtheIdea(string idea) pure private returns (bytes32) {
32        return keccak256(idea);
33    }
34
35}
36
```

Figure 14.12: Visual Studio Code

The preceding screenshot displays a Visual Studio Code window that comprises a file explorer on the left-hand side and a code editor window. Due to syntax highlighting and IntelliSense being enabled by the Solidity plugin, it becomes easy to write smart contract code. The Solidity plugin for Visual Studio is available in the Visual Studio marketplace at <https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity>.

Testing smart contracts

Testing is usually performed by automated means. Earlier in the chapter, you were introduced to Truffle, which uses the Mocha framework to test contracts. However, manual functional testing can be performed as well by using the Remix IDE, which was discussed in *Chapter 13, Ethereum*.

Development Environment, and running functions manually and validating results. We will cover this in *Chapter 15, Introducing Web3*.

Deploying smart contracts

Once the contract is verified, working, and tested on a simulated environment (for example, Ganache) or on a private net, it can be deployed to a public testnet such as **Ropsten** and eventually to the live blockchain (mainnet).

We will cover all these steps, including verification, development, and creating a web interface, in the next chapter, *Chapter 15, Introducing Web3*.

Now that we have covered which tools can be used to write Solidity smart contracts, we will introduce the Solidity language. This will be a brief introduction to Solidity, which should provide the base knowledge required to write smart contracts. The syntax of the language is very similar to C and JavaScript, and it is quite easy to program. We'll start by exploring what a smart contract written in the Solidity language looks like.

The layout of a Solidity source code file

In the following subsections, we will look at the components of a Solidity source code file, which is important to cover before we move on to writing smart contracts in the next section.

Version pragma

In order to address compatibility issues that may arise from future versions of the `solc` version, `pragma` can be used to specify the version of the

compatible compiler as in the following example:

```
pragma solidity ^0.5.0
```

This will ensure that the source file does not compile with versions lower than `0.5.0` and versions starting from `0.6.0`.

Import

Import in Solidity allows the importing of symbols from the existing Solidity files into the current global scope. This is similar to `import` statements available in JavaScript, as in the following:, for example:

```
import "module-name";
```

Comments

Comments can be added to the Solidity source code file in a manner similar to the C language. Multiple-line comments are enclosed in `/*` and `*/`, whereas single-line comments start with `//`.

An example `solidity` program is as follows, showing the use of `pragma`, `import`, and comments:

```
1 pragma solidity ^0.5.0; //specify the solidity compiler version
2 /*
3 this is a simple value checker contract that checks the value provided
4 and returns boolean value (true or false) based on the condition expression
5 evaluation
6 */
7 import "./mapping.sol"; //import a file
8 contract valuechecker {
9     uint price = 10;
10    //price variable declared and initialized with a value of 10
11    event valueEvent(bool returnValue);
12    function Matcher (uint8 x) public returns (bool) {
13        if (x >= price )
14        {
15            emit valueEvent(true);
16            return true;
17        }
18    }
19 }
```

Figure 14.13: Sample Solidity program as shown in the Remix IDE

In this section, we examined what Solidity code of a smart contract looks like. Now it's time to learn about the Solidity language.

The Solidity language

Solidity is a domain-specific language of choice for programming contracts in Ethereum. There are other languages that can be used, such as Serpent, Mutan, and LLL, but Solidity is the most popular. Its syntax is closer to both JavaScript and C.

Solidity has evolved into a mature language over the last few years and is quite easy to use, but it still has a long way to go before it can become advanced, standardized, and feature-rich, like other well-established languages such as Java, C, or C#. Nevertheless, this is the most widely used language currently available for programming contracts.

It is a statically typed language, which means that variable type checking in Solidity is carried out at compile time. Each variable, either state or local, must be specified with a type at compile time. This is beneficial in the sense that any validation and checking is completed at compile time and certain types of bugs, such as the interpretation of data types, can be caught earlier in the development cycle instead of at runtime, which could be costly,

especially in the case of the blockchain/smart contract paradigm. Other features of the language include inheritance, libraries, and the ability to define composite data types.

Solidity is also called a contract-oriented language. In Solidity, contracts are equivalent to the concept of classes in other object-oriented programming languages.

Variables

Just like any programming language, variables in Solidity are the named memory locations that hold values in a program. There are three types of variables in Solidity: **local variables**, **global variables**, and **state variables**.

Local variables

These variables have a scope limited to only within the function they are declared in. In other words, their values are present only during the execution of the function in which they are declared.

Global variables

These variables are available globally as they exist in the global namespace. They are used for performing various functions such as ABI encoding, cryptographic functions, and querying blockchain and transaction information.

Solidity provides a number of global variables that are always available in the global namespace. These variables provide information about blocks and transactions. Additionally, cryptographic functions, ABI encoding/decoding, and address-related variables are available. A subset of available variables is shown as follows.

This function is used to compute the Keccak-256 hash of the argument provided to the function:

```
keccak256(...) returns (bytes32)
```

This function returns the associated address of the public key from the elliptic curve signature:

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (
```

This returns the current block number:

```
block.number
```

This returns the gas price of the transaction:

```
tx.gasprice (uint)
```



There are a number of other global variables available. A comprehensive list and details can be found in Solidity's official documentation:
<https://solidity.readthedocs.io/en/latest/units-and-global-variables.html>.

State variables

State variables have their values permanently stored in smart contract storage. State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists:

```
pragma solidity >=0.5.0;
contract Addition {
    uint x; // State variable
}
```

Here, `x` is a state variable whose value will be stored in contract storage.

There are three types of state variables, based on their visibility scope

- **Private:** These variables are only accessible internally from within the contract that they are originally defined in. They are also not accessible from any derived contract from the original contract.
- **Public:** These variables are part of the contract interface. In simple words, anyone is able to get the value of these variables. They are accessible within the contract internally by using the `this` keyword. They can also be called from other contracts and transactions. A `getter` function is automatically created for all public variables.
- **Internal:** These variables are only accessible internally within the contract that they are defined in. In contrast to private state variables, they are also accessible from any derived contract from the original (parent) contract.

In the next section, we will introduce the data types supported in Solidity.

Data types

Solidity has two categories of data types: **value types** and **reference types**.

Value types are variables that are always passed by a value. This means that value types hold their value or data directly, allowing a variable's value held in memory to be directly accessible by accessing the variable.

Reference types store the address of the memory location where the value is stored. This is in contrast with value types, which store the actual value of a variable directly with the variable itself. When using reference types, it is essential to explicitly specify the storage area where the type is stored, for example, *memory*, *storage*, or *calldata*.

Remember that EVM can read and write data in different locations.

- Stack: This is a temporary storage area where EVM opcodes pop and push data during execution of the bytecode.



- Calldata: This is a read-only location that holds the data field of a transaction. It holds function execution data and parameters.
- Memory: This is a temporary memory location that is available to functions of the smart contract during their execution. As soon as the execution of the transaction finishes, this memory is cleared.
- Storage: This is the persistent global memory available to all functions in a smart contract. State variables use storage.
- Code: This is where the code that is executing is stored. This can also be used as a static data storage location.
- Logs: This is the area where the output of the events emitting from the smart contracts is stored.

The specific location used for storing values of a variable depends on the data type of the variable and where the variable has been declared. For example, function parameter variables are stored in *memory*, whereas state variables are stored in *storage*.

Now we'll describe value types in detail.

Value types

Value types mainly include **Booleans**, **integers**, **addresses**, and **literals**, which are explained in detail here.

Boolean

This data type has two possible values, `true` or `false`, for example:

```
bool v = true;
```

or

```
bool v = false;
```

This statement assigns the value `true` or `false` to `v` depending on the assignment.

Integers

This data type represents integers. The following table shows various keywords used to declare integer data types:

Keyword	Types	Details
<code>int</code>	Signed integer	<code>int8</code> to <code>int256</code> , which means that keywords are available from <code>int8</code> up to <code>int256</code> in increments of 8, for example, <code>int8</code> , <code>int16</code> , and <code>int24</code> .
<code>uint</code>	Unsigned integer	<code>uint8</code> , <code>uint16</code> , ... to <code>uint256</code> , unsigned integer from 8 bits to 256 bits. Usage is dependent on how many bits are required to be stored in the variable.

For example, in this code, note that `uint` is an alias for `uint256`:

```
uint256 x;  
uint y;  
uint256 z;
```

These types can also be declared with the `constant` keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:

```
uint constant z=10+10;
```

Address

This data type holds a 160-bit long (20 byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:

- **Balance:** The `balance` member returns the balance of the address in Wei.
- **Send:** This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and returns `true` or `false` depending on the result of the transaction, for example, the following:

```
address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;
address from = this;
if (to.balance < 10 && from.balance > 50) to.send(20);
```

- **Call functions:** The `call`, `callcode`, and `delegatecall` functions are provided in order to interact with functions that do not have an ABI. These functions should be used with caution as they are not safe to use due to the impact on type safety and security of the contracts.
- **Array value types (fixed-size and dynamically sized byte arrays):** Solidity has fixed-size and dynamically sized byte arrays. Fixed-size keywords range from `bytes1` to `bytes32`, whereas dynamically sized keywords include `bytes` and `string`. The `bytes` keyword is used for raw byte data, and `string` is used for strings encoded in UTF-8. As these arrays are returned by the value, calling them will incur a gas cost. `length` is a member of array value types and returns the length of the byte array.

- An example of a static (fixed size) array is as follows:

```
bytes32[10] bankAccounts;
```

- An example of a dynamically sized array is as follows:

```
bytes32[] trades;
```

- Get the length of trades by using the following code:

```
trades.length;
```

Literals

These are used to represent a fixed value. There are different types of literals that are described as follows:

- **Integer literals:** These are a sequence of decimal numbers in the range of 0–9. An example is shown as follows:

```
uint8 x = 2;
```

- **String literals:** This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt' "packt"
```

- **Hexadecimal literals:** These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

- **Enums:** This allows the creation of user-defined types. An example is shown as follows:

```
enum Order {Filled, Placed, Expired};  
Order private ord;  
ord=Order.Filled;
```

Explicit conversion to and from all integer types is allowed with enums.

Reference types

As the name suggests, these types are passed by reference and are discussed in the following section. These are also known as **complex types**. Reference types include **arrays**, **structs**, and **mappings**.

Arrays

Arrays represent a contiguous set of elements of the same size and type laid out at a memory location. The concept is the same as any other programming language. Arrays have two members, named `length` and `push`.

Structs

These constructs can be used to group a set of dissimilar data types under a logical group. These can be used to define new custom types, as shown in the following example:

```
pragma solidity ^0.4.0;
contract TestStruct {
    struct Trade
    {
        uint tradeid;
        uint quantity;
        uint price;
        string trader;
    }
    //This struct can be initialized and used as below
    Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trader:"equinox"});
}
```

In the preceding code, we declared a `struct` named `Trade` that has four fields. `tradeid`, `quantity`, and `price` are of the `uint` type, whereas `trader` is of the `string` type. Once the `struct` is declared, we can initialize and use it. We initialize it by using `Trade tStruct` and assigning `123` to `tradeid`, `1` to `quantity`, and `"equinox"` to `trader`.

Data location

Sometimes it is desirable to choose the location of the variable data storage. This choice allows for better gas expense management. We can use the data location name to specify where a particular complex data type will be stored. Depending on the default or annotation specified, the location can be `storage`, `memory`, or `calldata`. This is applicable to arrays and structs and can be specified using the `storage` or `memory` keywords. `calldata` behaves almost like `memory`. It is an unmodifiable and temporary area that can be used to store function arguments.

For example, in the preceding structs example, if we want to use only `memory` (temporarily) we can do that by using the `memory` keyword when using the structure and assigning values to fields in the `struct`, as shown here:

```
Trade memory tStruct;
tStruct.tradeid = 123;
```

As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times.

Parameters of external functions use **calldata** memory. By default, parameters of functions are stored in **memory**, whereas all other local variables make use of **storage**. State variables, on the other hand, are required to use storage.

Mappings

Mappings are used for a key-to-value mapping. This is a way to associate a value with a key. All values in this map are already initialized with all zeroes, as in the following for example:

```
mapping (address => uint) offers;
```

This example shows that `offers` is declared as a mapping. Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table, where string values are mapped to integer values. The mapping named `bids` has the string `packt` mapped to value `10`.

Control structures

The control structures available in the Solidity language are `if...else`, `do`, `while`, `for`, `break`, `continue`, and `return`. They work exactly the same as other languages, such as the C language or JavaScript.

Some examples are shown here:

- `if` : If `x` is equal to `0`, then assign value `0` to `y`, else assign `1` to `z` :

```
if (x == 0)
    y = 0;
else
    z = 1;
```

- `do` : Increment `x` while `z` is greater than `1` :

```
do{
    x++;
} (while z>1);
```

- `while` : Increment `z` while `x` is greater than `0` :

```
while(x > 0) {
    z++;
}
```

- `for` , `break` , and `continue` : Perform some work until `x` is less than or equal to `10` . This `for` loop will run `10` times; if `z` is `5` , then break the `for` loop:

```
for(uint8 x=0; x<=10; x++)
{
    //perform some work
    z++
    if(z == 5) break;
}
```

It will continue the work in a similar vein, but when the condition is met, the loop will start again.

- `return` : `return` is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

It will stop the execution and return a value of `0` .

Events

Events in Solidity can be used to log certain events in EVM logs. These are quite useful when external interfaces are required to be notified of any change or event in the contract. These logs are stored on the blockchain in transaction logs. Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.

In a simple example here, the `valueEvent` event will return `true` if the `x` parameter passed to the function `Matcher` is equal to or greater than `10`:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Inheritance

Inheritance is supported in Solidity. The `is` keyword is used to derive a contract from another contract. In the following example, `valueChecker2` is derived from the `valueChecker` contract. The derived contract has access to all non-private members of the parent contract:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);
```

```

function Matcher(uint8 x) public returns (bool)
{
    if (x>=price)
    {
        emit valueEvent(true);
        return true;
    }
}
contract valueChecker2 is valueChecker
{
    function Matcher2() public view returns (uint)
    {
        return price+10;
    }
}

```

In the preceding example, if the `uint8 price = 20` is changed to `uint8 private price = 20`, then it will not be accessible by the `valueChecker2` contract. This is because now the member is declared as `private`, and thus it is not allowed to be accessed by any other contract. The error message that you will see when attempting to compile this contract is as follows:

```

browser/valuechecker.sol:20:8: DeclarationError: Undeclared identifier.
return price+10;
^---^

```

Libraries

Libraries are deployed only once at a specific address and their code is called via the `CALLCODE` or `DELEGATECALL` opcode of the EVM. The key idea behind libraries is code reusability. They are similar to contracts and act as base contracts to the calling contracts. A library can be declared as shown in the following example:

```

library Addition
{
    function Add(uint x,uint y) returns (uint z)

```

```
    {
        return x + y;
    }
}
```

This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether.

Functions

Functions are pieces of code within a smart contract. For example, look at the following code block:

```
pragma solidity >5.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

In the preceding code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

In this case, `2` is supplied via variable `x`, and the function will return `4` by adding `2` to the value of `x`. It is a simple function, but demonstrates how functions work and what their different elements are.

There are two function types: *internal* and *external* functions.

- **Internal functions** can be used only within the context of the current contract.
- **External functions** can be called via external function calls.

A **function** in Solidity can be marked as a constant. Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas. This is the practical implementation of the concept of *call* as discussed in the previous chapter.



Note that the state mutability modifier `constant` was removed in Solidity version 0.5.0. Instead, use the `view` or `pure` modifier if you are using Solidity version 0.5.0 or greater.

The syntax to declare a function is shown as follows:

```
function <nameofthefunction> (<parameter types> <name of the var  
{internal|external} [state mutability modifier] [returns (<retur
```

For example:

```
function addition1() public view returns (uint y)
```

Here, `function` is the keyword used to declare the function, `addition1` is the name of the function, `public` is the visibility modifier, `view` is the state mutability modifier, `returns` is the keyword to specify what is returned from the function, and `uint` is the return type with the variable name `y`.

Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifiers, state mutability modifiers, and an optional return type. This is shown in the following example:

```
function orderMatcher (uint x)
private view returns (bool return value)
```

In the preceding code, `function` is the keyword used to declare the function. `orderMatcher` is the function name, `uint x` is an optional parameter, `private` is the **access modifier** or **specifier** that controls access to the function from external contracts, `view` is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract, and `returns (bool return value)` is the optional return type of the function. After this introduction to functions in Solidity, let's consider some of their key elements, parameters, and modifiers:

- **How to define a function:** The syntax of defining a function is shown as follows:

```
function <name of the function>(<parameters>) <visibility s]
(<return data type> <name of the variable>
{
    <function body>
}
```

- **Function signature:** Functions in Solidity are identified by their **signature**, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in the Remix IDE, as shown in the following screenshot. `f9d55e21` is the first four bytes of the 32-byte Keccak-256 hash of the function named `Matcher`:

```

FUNCTIONHASHES 🗂️ 🎐

{
    "f9d55e21": "Matcher(uint8)"
}

```

Figure 14.14: Function hash as shown in the Remix IDE

In this example function, `Matcher` has the signature hash of `d99c89cb`. This information is useful in order to build interfaces.

- **Input parameters of a function:** Input parameters of a function are declared in the form of `<data type> <parameter name>`. This example clarifies the concept where `uint x` and `uint y` are input parameters of the `checkValues` function:

```

contract myContract
{
    function checkValues(uint x, uint y)
    {
    }
}

```

- **Output parameters of a function:** Output parameters of a function are declared in the form of `<data type> <parameter name>`. This example shows a simple function returning a `uint` value:

```

contract myContract
{
    function getValue() returns (uint z)
    {
        z=x+y;
    }
}

```

A function can return multiple values as well as take multiple inputs. In the preceding example function, `getValue` only returns one value, but a function can return up to 14 values of different data types. The names of the unused return parameters can optionally be omitted. An example of such a function could be:

```

pragma solidity ^0.5.0;
contract Test1
{
    function addition1(uint x, uint y) public pure returns
    {
        z= x+y ;
        a=x+y;
        return (z,a);
    }
}

```

Here when the code runs, it will take two parameters as input `x` and `y`, add both, and then assign them to `z` and `a`, and finally return `z` and `a`. For example, if we provide 1 and 1 for `x` and `y`, respectively, then when the variables `z` and `a` are returned by the function, both will contain 2 as the result.

- **Internal function calls:** Functions within the context of the current contract can be called internally in a direct manner. These calls are made to call the functions that exist within the same contract. These calls result in simple `JUMP` calls at the EVM bytecode level.
- **External function calls:** External function calls are made via message calls from a contract to another contract. In this case, all function parameters are copied to the memory. If a call to an internal function is made using the `this` keyword, it is also considered an external call. The `this` variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members of a contract are inherited from the address.
- **Fallback functions:** This is an unnamed function in a contract with no arguments and return data. This function executes every time ether is received. It is required to be implemented within a contract if the contract is intended to receive ether; otherwise, an exception will be thrown and ether will be returned. This function also executes if no other function signatures match in the contract. If the contract is expected to receive ether, then the fallback function should be declared with the payable **modifier**.

The payable is required; otherwise, this function will not be able to receive any ether. This function can be called using the `address.call()` method as, for example, in the following:

```
function ()  
{  
    throw;  
}
```

In this case, if the fallback function is called according to the conditions described earlier; it will call `throw`, which will roll back the state to what it was before making the call. It can also be some other construct than `throw`; for example, it can log an event that can be used as an alert to feed back the outcome of the call to the calling application.

- **Modifier functions:** These functions are used to change the behavior of a function and can be called before other functions. Usually, they are used to check some conditions or verification before executing the function. `_` (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be *guarded*. This concept is similar to guard functions in other languages.
- **Constructor function:** This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.
- **Function visibility specifiers (access modifiers/access levels):** Functions can be defined with four access specifiers as follows:
 - **External:** These functions are accessible from other contracts and transactions. They cannot be called internally unless the `this` keyword is used.
 - **Public:** By default, functions are public. They can be called either internally or by using messages.
 - **Internal:** Internal functions are visible to other derived contracts from the parent contract.

- **Private**: Private functions are only visible to the same contract they are declared in.
- **Function modifiers**:
 - **pure**: This modifier prohibits access or modification to state.
 - **view**: This modifier disables any modification to state.
 - **payable**: This modifier allows payment of ether with a call.
 - **constant**: This modifier disallows access or modification to state. This is available before version 0.5.0 of Solidity.

Now we've considered some of the defining features of functions, let's consider how Solidity approaches handling errors.

Error handling

Solidity provides various functions for error handling. By default, in Solidity, whenever an error occurs, the state does not change and reverts back to the original state.

Some constructs and convenience functions that are available for error handling in Solidity are introduced as follows:

- **Assert**: This is used to check for conditions and throw an exception if the condition is not met. Assert is intended to be used for internal errors and invariant checking. When called, this method results in an invalid opcode and any changes in the state are reverted back.
- **Require**: Similar to *assert*, this is used for checking conditions and throws an exception if the condition is not met. The difference is that *require* is used for validating inputs, return values, or calls to external contracts. The method also results in reverting back to the original state. It can also take an optional parameter to provide a custom error message.
- **Revert**: This method aborts the execution and reverts the current call. It can also optionally take a parameter to return a custom error message to the caller.

- **Try/Catch:** This construct is used to handle a failure in an external call.
- **Throw:** Throw is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

This completes a brief introduction to the Solidity language. The language is very rich and under constant improvement. Detailed documentation and coding guidelines are available online at
<http://solidity.readthedocs.io/en/latest/>.

Summary

This chapter started with the introduction of development tools for Ethereum, such as Ganache CLI. The installation of Node.js was also introduced, as most of the tools are JavaScript- and Node.js-based. Then we discussed some frameworks such as Truffle, along with local blockchain solutions for development and testing, such as Ganache, and Drizzle. We also introduced Solidity in this chapter and explored different concepts, such as value types, reference types, functions, and error handling concepts. We also learned how to write contracts using Solidity.

In the next chapter, we will explore the topic of Web3, a JavaScript API that is used to communicate with the Ethereum blockchain.

15

Introducing Web3

Web3 is a JavaScript library that can be used to communicate with an Ethereum node via RPC communication. Web3 works by exposing methods that have been enabled over RPC. This allows the development of **user interfaces (UIs)** that make use of the Web3 library in order to interact with contracts deployed over the blockchain.

In this chapter, we'll explore the Web3 API, and introduce some detailed examples of how smart contracts are written, tested, and deployed to the Ethereum blockchain. We will use various tools such as the Remix IDE and Ganache to develop and test smart contracts, and look at the methods used to deploy smart contracts to Ethereum test networks and private networks. The chapter will explore how HTML and JavaScript frontends can be developed to interact with smart contracts deployed on the blockchain, and introduce advanced libraries such as Drizzle. The topics we will cover are as follows:

- Exploring Web3 with Geth
- Contract deployment
- Interacting with contracts via frontends
- Development frameworks

We will start with Web3, and gradually build our knowledge with various tools and techniques for smart contract development. You will be able to test your knowledge in the bonus content pages for this book, where we will develop a project using all the techniques that this chapter will cover.

Exploring Web3 with Geth

In order to expose the required APIs via `geth`, the following command can be used:

```
$ geth --datadir ~/etherprivate --networkid 786 --rpc --rpccapi "
```



The `--rpccapi` flag in the preceding command allows the `web3`, `eth`, `net`, and `debug` methods. There are other APIs such as `personal`, `miner`, and `admin` that can be exposed by adding their names to this list.

Web3 is a powerful API and can be explored by attaching a `geth` instance. Later in this section, you will be introduced to the concepts and techniques of making use of Web3 via JavaScript/HTML frontends. The `geth` instance can be attached using the following command:

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

Once the `geth` JavaScript console is running, Web3 can be queried:

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.10-stable/darwin-amd64/go1.13.6
coinbase: 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
at block: 3521 (Sun, 26 Jan 2020 15:39:39 GMT)
datadir: /Users/drequinox/etherprivate
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

|> web3.version
{
  api: "0.20.1",
  ethereum: "0x40",
  network: "786",
  node: "Geth/v1.9.10-stable/darwin-amd64/go1.13.6",
  whisper: undefined,
  getEthereum: function(callback),
  getNetwork: function(callback),
  getNode: function(callback),
  getWhisper: function(callback)
}
```

Figure 15.1: Web3 via geth.ipc

Now that we've introduced Web3, let's consider how the Remix IDE can be used to deploy a contract, and how the `geth` console can interact with the deployed contract.

Contract deployment

A simple contract can be deployed using Geth and interacted with using Web3 via the **command-line interface (CLI)** that `geth` provides (`console` or `attach`). The following are the steps to achieve that. As an example, the following source code will be used:

```
pragma solidity ^0.4.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Run the `geth` client using the following command:

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure
```

or

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure
```

Alternatively, the following command can be run:

```
$ geth --datadir ~/etherprivate/ --networkid 786 --rpc --rpccapi
```

We've used all these commands in different contexts before, and you can use any of them to run Geth. The difference is that the first command allows connectivity with the Remix IDE as we have specified `--rpccorsdomain https://remix.ethereum.org`. If you need to use the Remix IDE then use the first command.

The second command allows `localhost:7777` to access the RPC server exposed by `geth`. This option is useful if you have an application running on this interface and you want to give it access to RPC. It also exposes RPC on port `8001` via the flag `--rpcport 8001`, which is useful in case you have some other service or application listening already on port `8545`, which would mean that `geth` won't be able to use that already-in-use port. This is because Geth listens on port `8545` for the HTTP-RPC server by default.

Alternatively, you can simply run the last command, which allows all incoming connections as `--rpccorsdomain` is set to `*`.

If the Geth console is not already running, open another terminal and run the following command:

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

In order to deploy a smart contract, we use the Web3 deployment script. The main elements of the script, the **Application Binary Interface (ABI)** and the bytecode, can be generated from the Remix IDE. To learn how to download and use the Remix IDE, refer to *Chapter 13, Ethereum Development Environment*. First, paste the following source code, as mentioned at the beginning of the section, in the Remix IDE.

We will discuss the Remix IDE in more detail later in this chapter; for now, we are using this IDE only to get the required elements (ABI and bytecode)

for the Web3 deployment script used for deployment of the contract:

```
pragma solidity ^0.4.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Once the code is pasted in the Remix IDE, it will appear as follows in the Solidity compiler:

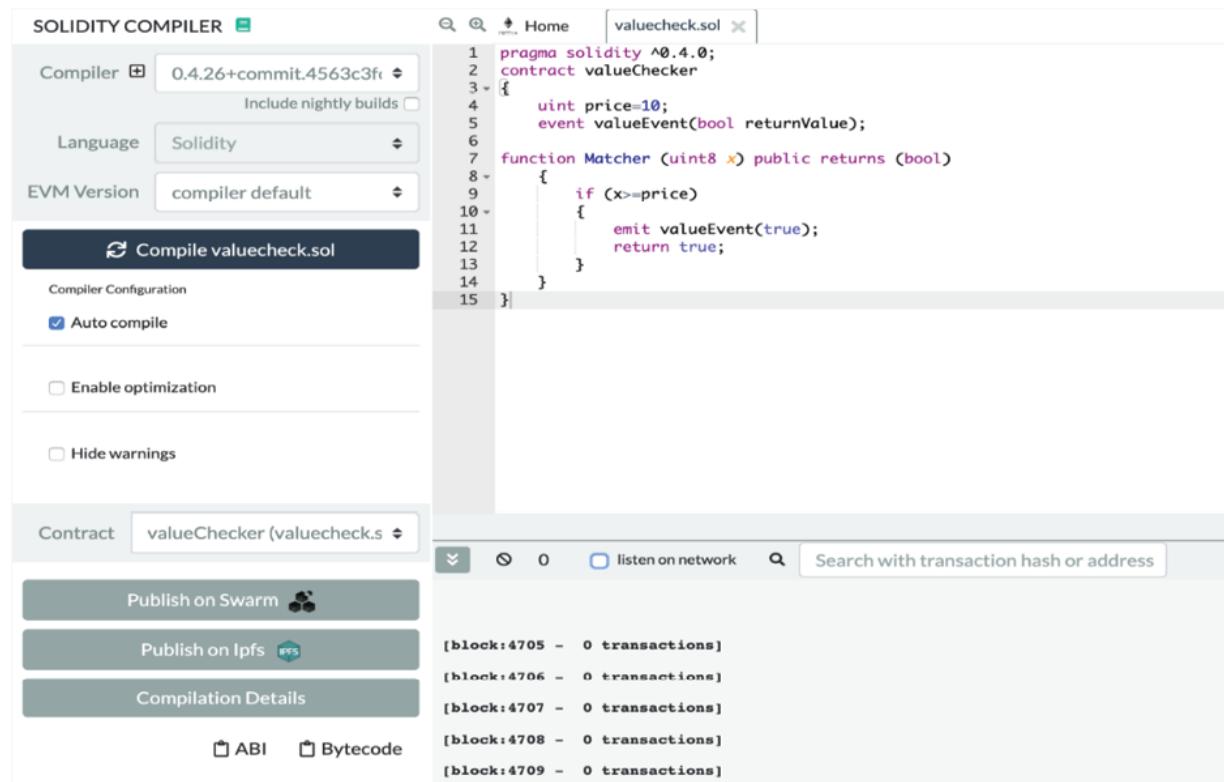


Figure 15.2: Code shown in Remix

Now create the Web3 deployment script as shown as follows. First generate the ABI and bytecode by using the **ABI** and **Bytecode** buttons in the Remix IDE (shown at the lower-left side in *Figure 15.2*) and paste them in the following script after the `web3.eth.contract()` and `data:` elements, respectively:

```
var valuecheckerContract = web3.eth.contract([{ "anonymous":false
var valuechecker = valuecheckerContract.new(
{
    from: web3.eth.accounts[0],
    data: '0x6080604052600a60005534801561001557600080fd5b506101
    gas: '4700000'
}, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
        console.log('Contract mined! address: ' + contract.addr
    }
})
})
```

Ensure that the accounts are unlocked. First list the accounts by using the following command, which outputs account `0` (first account), as shown here:

```
> personal.listAccounts[0]
"0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811"
```

Now unlock the account using the following command. It will need the passphrase (password) that you used originally when creating this account. Enter the password to unlock the account:

```
> personal.unlockAccount(personal.listAccounts[0])
Unlock account 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
Password:
true
```

For more convenience, the account can be unlocked permanently for the length of the `geth` console/attach session by using the command shown here.

Now, we can open the `geth` console that has been opened previously, and deploy the contract. However, before deploying the contract, make sure that mining is running on the `geth` node. The following command can be used to start mining under the `geth` console.

```
> miner.start()
```

Now paste the previously mentioned Web3 deployment script in the `geth` console as shown in the following screenshot:

Figure 15.3: The Web3 deployment script deployment using Geth

The previous screenshot shows the output as it looks when the Web3 deployment script is pasted in the Geth console for deployment. Note the data (code) section, which contains the bytecode of the smart contract.

ABI and code can also be obtained by using the Solidity compiler, as shown in the following code snippets.

In order to generate the ABI we can use the command shown as follows:

```
$ solc --abi valuechecker.sol
```

This command will produce the following output, with the contract ABI in **JSON** format:

```
===== valuechecker.sol:valueChecker =====
Contract JSON ABI
[{"anonymous":false,"inputs":[{"indexed":false,"internalType":"k
```

The next step is to generate code, which we can use the following command to do:

```
$ solc --bin valuechecker.sol
```

This command will produce the binary (represented as hex) of the smart contract code:

```
===== valuechecker.sol:valueChecker =====
Binary:
6080604052600a60005534801561001557600080fd5b5061010d80610025600c
```

We can also see the relevant message in the `geth` logs to verify that the contract creation transaction has been submitted; you will see messages similar to the one shown as follows:

```
INFO . . . . . Submitted contract creation fullhas
```

Also notice that in the Geth console, the following message appears as soon as the transaction is mined, indicating that the contract has been successfully mined:

```
Contract mined! address: 0x82012b7601fd23669b50bb7dd79460970ce38
```

Notice that in the preceding output, the transaction hash

`0x73fceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a198efab` is also shown.

After the contract is deployed successfully, you can query various attributes related to this contract, which we will also use later in this example, such as the contract address and ABI definition, as shown in the following screenshot. Remember, all of these commands are issued via the `geth` console, which we have already opened and used for contract deployment.

```
> valuechecker.
valuechecker.Matcher      valuechecker.abi           valuechecker.allEvents   valuechecker.transactionHash
valuechecker._eth          valuechecker.address       valuechecker.constructor  valuechecker.valueEvent
> valuechecker.abi
[{
  anonymous: false,
  inputs: [{
    indexed: false,
    internalType: "bool",
    name: "returnValue",
    type: "bool"
  }],
  name: "valueEvent",
  type: "event"
}, {
  inputs: [{
    internalType: "uint8",
    name: "x",
    type: "uint8"
  }],
  name: "Matcher",
  outputs: [{
    internalType: "bool",
    name: "",
    type: "bool"
  }],
  stateMutability: "nonpayable",
  type: "function"
}]
>
```

Figure 15.4: Value checker attributes

In order to make interaction with the contract easier, the address of the account can be assigned to a variable. There are a number of methods that are now exposed, and the contract can be further queried now, for example:

```
> eth.getBalance(valuechecker.address)
0
```

We can now call the actual methods in the contract. A list of the various methods that have been exposed now can be seen as follows:

<code>> valuechecker.</code>	<code>valuechecker.Matcher</code>	<code>valuechecker.abi</code>	<code>valuechecker.allEvents</code>	<code>valuechecker.transactionHash</code>
	<code>valuechecker._eth</code>	<code>valuechecker.address</code>	<code>valuechecker.constructor</code>	<code>valuechecker.valueEvent</code>

Figure 15.5: Various methods for valuechecker

The contract can be further queried as follows.

First, we find the transaction hash, which identifies the transaction:

```
> valuechecker.transactionHash
```

The output of this command is the transaction hash of the contract creation transaction.

```
"0x73fcceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a198e
```

We can also query the ABI, using the following command:

```
> valuechecker.abi
```

The output will be as shown as follows. Note that it shows all the inputs and outputs of our example contract.

```
[ {
    anonymous: false,
    inputs: [ {
        indexed: false,
        internalType: "bool",
        name: "returnValue",
        type: "bool"
    }],
    name: "valueEvent",
    type: "event"
}, {
    inputs: [ {
        internalType: "uint8",
        name: "x",
        type: "uint8"
    }],
    name: "Matcher",
    outputs: [ {
        internalType: "bool",
        name: "",
        type: "bool"
    }],
    stateMutability: "nonpayable",
}
```

```
        type: "function"
    } ]
```

In the following example, the `Matcher` function is called with the arguments. Arguments, also called parameters, are the values passed to the functions. Remember that in the smart contract code shown in *Figure 15.2*, there is a condition that checks if the value is equal to or greater than 10, and if so, the function returns `true`; otherwise, it returns `false`. To test this, type the following commands into the `geth` console that you have open.

Pass `12` as an argument, which will return `true` as it is greater than 10:

```
> valuechecker.Matcher.call(12)
true
```

Pass `10` as an argument, which will return `true` as it is equal to 10:

```
> valuechecker.Matcher.call(10)
true
```

Pass `9` as an argument, which will return `false` as it is less than 10:

```
> valuechecker.Matcher.call(9)
false
```

In this section, we learned how to use the Remix IDE to create and deploy contracts. We also learned how the `geth` console can be used to interact with a smart contract and explored which methods are available to interact with smart contracts on the blockchain. Now we'll see how we can interact with `geth` using JSON RPC over HTTP.

POST requests

It is possible to interact with `geth` via JSON RPC over HTTP. For this purpose, the `curl` tool can be used.



`curl` is available at <https://curl.haxx.se/>.

An example is shown here to familiarize you with the POST request and show how to make POST requests using `curl`.



POST is a request method supported by HTTP. You can read more about POST here: [https://en.wikipedia.org/wiki/POST_\(HTTP\)](https://en.wikipedia.org/wiki/POST_(HTTP)).

Before using the JSON RPC interface over HTTP, the `geth` client should be started up with appropriate switches, as shown here:

```
--rpcapi web3
```

This switch will enable the `web3` interface over HTTP. The Linux command, `curl`, can be used for the purpose of communicating over HTTP, as shown in the following example.

Retrieving the list of accounts

For example, in order to retrieve the list of accounts using the `personal_listAccounts` method, the following command can be used:

```
$ curl --request POST --data '{"jsonrpc":"2.0","method":"personal
```

This will return the output, a JSON object with the list of accounts:

```
{"jsonrpc":"2.0","id":4,"result":["0xc9bf76271b9e42e4bf7e1888e0f
```

In the preceding `curl` command, `--request` is used to specify the request command, `POST` is the request, and `--data` is used to specify the parameters and values. Finally, `localhost:8545` is used where the HTTP endpoint from Geth is opened.

In this section, we covered how we can interact with the smart contract using the JSON RPC over HTTP. While this is a common way of interacting with the contracts, the examples we have seen so far are command line-based. In the next section, we'll see how we can interact with the contracts by creating a user-friendly web interface.

Interacting with contracts via frontends

It is desirable to interact with the contracts in a user-friendly manner via a webpage. It is possible to interact with the contracts using the `web3.js` library from HTML-/JS-/CSS-based webpages.

The HTML and JavaScript frontend

The HTML content can be served using any HTTP web server, whereas `web3.js` can connect via local RPC to the running Ethereum client (`geth`) and provide an interface to the contracts on the blockchain. This architecture can be visualized in the following diagram:

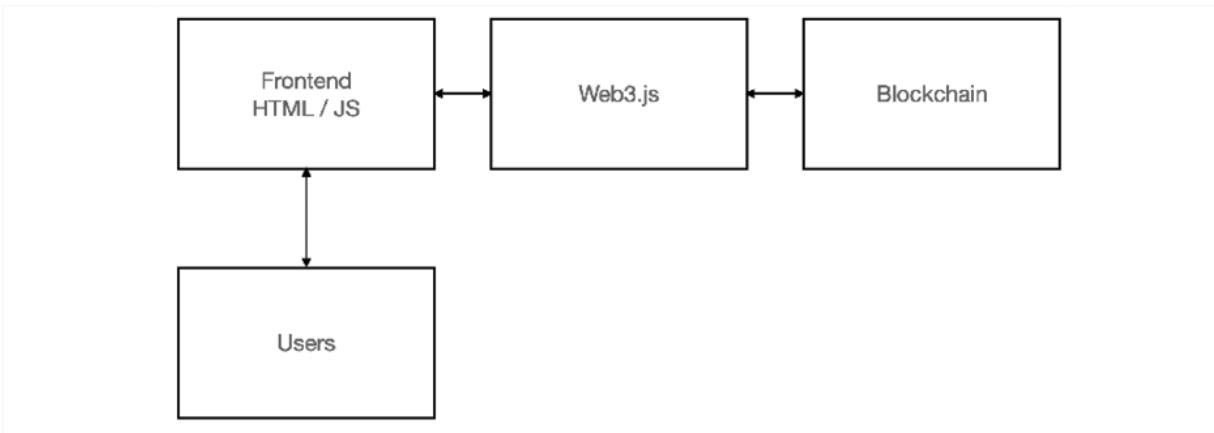


Figure 15.6: web3.js, frontend, and blockchain interaction architecture

If `web3.js` is not JavaScript frontend already installed, use these steps; otherwise, move on to the next section, *Interacting with contracts via a web frontend*.

Installing Web3.js JavaScript library

Web3, which we discussed earlier in this chapter, was looked at in the context of the Web3 API exposed by `geth`. In this section, we will introduce the Web3 JavaScript library (`web3.js`), which is used to introduce different functionalities related to the Ethereum ecosystem in DApps. The `web3.js` library is a collection of several modules, which are listed as follows with the functionality that they provide.

- `web3-eth` : Ethereum blockchain and smart contracts
- `web3-shh` : Whisper protocol (P2P communication and broadcast)
- `web3-bzz` : Swarm protocol, which provides decentralized storage
- `web3-utils` : Provides helper functions for DApp development

The `web3.js` library can be installed via `npm` by simply issuing the following command:

```
$ npm install web3
```



`web3.js` can also be directly downloaded from
<https://github.com/ethereum/web3.js>.

`web3.min.js`, downloaded via `npm`, can be referenced in the HTML files. This can be found under `node_modules`, for example,
`/home/drequinox/netstats/node_modules/web3/dist/web3.js`.



Note that `drequinox` is specific to the user under which these examples were developed; you will see the name of the user that you are running these commands under.

The file can optionally be copied into the directory where the main application is and can be used from there. Once the file is successfully referenced in HTML or JavaScript, Web3 needs to be initialized by providing an HTTP provider. This is usually the link to the `localhost` HTTP endpoint exposed by running the `geth` client. This can be achieved using the following code:

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'))
```

Once the provider is set, further interaction with the contracts and blockchain can be done using the `web3` object and its available methods. The `web3` object can be created using the following code:

```
if (typeof web3 !== 'undefined')  
{  
    web3 = new Web3(web3.currentProvider);  
}  
else  
{  
    web3 = new Web3(new  
        Web3.providers.HttpProvider("http://localhost:8545"));  
}
```

In this section, we have explored how to install `web3.js`, the Ethereum JavaScript API library, and how to create a `web3` object that can be used to

interact with the smart contracts using the HTTP provider running on the `localhost` as part of the Geth instance.

Interacting with contracts via a web frontend

In the following section, an example will be presented that will make use of `web3.js` to allow interaction with the contracts, via a webpage served over a simple HTTP web server. So far, we have seen how we can interact with a contract using the `geth` console via the command line, but in order for an application to be usable by end users, who will mostly be familiar with web interfaces, it becomes necessary to build web frontends so that users can communicate with the backend smart contracts using familiar webpage interfaces.

Creating an `app.js` JavaScript file

This can be achieved by following these steps. First, create a directory named `/simplecontract/app`, the home directory. This is the main directory under your user ID on Linux or macOS. This can be any directory, but in this example, the home directory is used.

Then, create a file named `app.js`, and write or copy the following code into it:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined') {
    web3 = new Web3(web3.currentProvider);
} else {
    // set the provider you want from Web3.providers
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
}
web3.eth.defaultAccount = web3.eth.accounts[0];
var SimpleContract = web3.eth.contract([
{
    "constant": false,
    "inputs": [
        {
            "name": "x",
            "type": "uint256"
        }
    ],
    "name": "SimpleContract",
    "outputs": [
        {
            "name": "y",
            "type": "uint256"
        }
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "contract"
}]);
```

```

        "type": "uint8"
    }
],
"name": "Matcher",
"outputs": [
{
    "name": "",
    "type": "bool"
}
],
"payable": false,
"stateMutability": "nonpayable",
"type": "function"
},
{
    "anonymous": false,
    "inputs": [
{
    "indexed": false,
    "name": "returnValue",
    "type": "bool"
}
],
"name": "valueEvent",
"type": "event"
}
]);
var simplecontract = SimpleContract.at("0x82012b7601fd23669b50bk
    console.log(simplecontract);
function callMatchertrue()
{
var txn = simplecontract.Matcher.call(12);
{
};
console.log("return value: " + txn);
}
function callMatcherfalse()
{
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}
function myFunction()
{
    var x = document.getElementById("txtValue").value;
    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
}

```

```
        document.getElementById("decision").innerHTML = txn;
    }
```

This file contains various elements; first, we created the `web3` object and provided a `localhost geth` instance listening on port `8545` as the Web3 provider. After this, the `web3.eth.accounts[0]` is selected as the account with which all the interactions will be performed with the smart contract. Next, the ABI is provided, which serves as the interface between the user and the contract. It can be queried using `geth`, generated using the Solidity compiler, or copied directly from the Remix IDE contract details. After this, the `simplecontract` is created, which refers to the smart contract with address `0x82012b7601fd23669b50bb7dd79460970ce386e3`. Finally, we declared three functions: `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`, which we will explain further shortly.

First, we create the frontend webpage. For this, we create a file named `index.html` with the source code shown as follows:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src=".//web3.js"></script>
<script src=".//app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the
<button onclick="myFunction()">get decision</button>
<p id="decision"></p>
<p>Calling the contract manually with hardcoded values, result 1
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

This file will serve as the frontend of our decentralized application. In other words, it provides the UI for interacting with the smart contracts. First, we referred to the JavaScript `web3.js` library and `app.js`, which we created earlier in this section. This will allow the HTML file to call required functions from these files.

After that, standard HTML is used to create an input text field so that users can enter the values. Then we used the `onclick` event to call the `myFunction()` function that we declared in our JavaScript `app.js` file. Finally, two `onclick` events with buttons `callTrue` and `callFalse` are used to call the `callMatchertrue()` and `callMatcherfalse()` functions, respectively.

We are keeping this very simple on purpose; there is no direct need to use jQuery, React, or Angular here, which would be a separate topic. Nevertheless, these frontend frameworks make development easier and a lot faster, and are commonly used for blockchain-related JavaScript frontend development.

In order to keep things simple, we are not going to use any frontend JavaScript frameworks in this section, as the main aim is to focus on blockchain technology and not the HTML, CSS, or JavaScript UI frameworks. However, in the bonus resource pack for this chapter, we will see an example of creating UIs using `react` with Drizzle.

In this part of the example, we have created a web frontend and a JavaScript file backend, where we have defined all the functions required for our application.

The `app.js` file we created in this section is the main JavaScript file that contains the code to create a `web3` object. It also provides methods that are used to interact with the contract on the blockchain. An explanation of the code previously used is given in the next sections.

Creating a Web3 object

The first step when creating a `web3.js`-based application is to create the `web3` object. It is created by selecting the appropriate available Web3 provider, which serves as an "entry point" to the blockchain through the HTTP RPC server exposed on a locally running Geth node.

```
if (typeof web3 !== 'undefined')  
{  
    web3 = new Web3(web3.currentProvider);  
}  
else  
{  
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:  
})
```

This code checks whether there is already an available Web3 provider; if yes, then it will set the provider to the current provider. Otherwise, it sets the provider to `localhost: 8545`; this is where the local instance of `geth` is exposing the HTTP-RPC server. In other words, the `geth` instance is running an HTTP-RPC server, which is listening on port `8545`.

Checking availability by calling the Web3 method

This line of code simply uses `console.log` to print the simple contract attributes, in order to verify the successful creation of the contract:

```
var simplecontract = SimpleContract.at("0x82012b7601fd23669b50bk  
console.log(simplecontract);
```

Once this call is executed, it will display various contract attributes indicating that the `web3` object has been created successfully and `HttpProvider` is available. Any other call can be used to verify the availability, but here, printing simple contract attributes has been used.

In this part of the example, we have created a `simplecontract` instance and then used `console.log` to display some attributes of the contract, which indicates the successful creation of the `simplecontract` instance. In the

next stage of this example, we will explore how the contract functions can be called.

Calling contract functions

Once the `web3` object is correctly created and a `simplecontract` instance is created (in the previous section), calls to the contract functions can be made easily as shown in the following code:

```
function callMatchertrue()
{
  var txn = simplecontract.Matcher.call(12);
}
;
console.log("return value: " + txn);
}

function callMatcherfalse()
{
  var txn = simplecontract.Matcher.call(1);
}
;
console.log("return value: " + txn);
}

function myFunction()
{
  var x = document.getElementById("txtValue").value;
  var txn = simplecontract.Matcher.call(x);
}
;
console.log("return value: " + txn);
  document.getElementById("decision").innerHTML = txn;
}
```

The preceding code shows three simple functions, `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`. `callMatchertrue()` simply calls the smart contract function `matcher` using the `simplecontract` object we created in the `app.js` file we created earlier. Similarly, `callMatcherfalse()` calls the smart contract's `Matcher` function by providing a value of `1`. Finally, the `myFunction()` function is defined, which contains simple logic to read the value provided by the user on the webpage in a `txtValue` textbox and uses that value to call the smart contract's `matcher` function.

After that, the return value is also logged in the debug console, available in browsers by using `console.log`.

Calls can be made using `simplecontractinstance.Matcher.call` and then by passing the value for the argument. Recall the `Matcher` function in the Solidity code:

```
function Matcher (uint8 x) returns (bool)
```

It takes one argument `x` of type `uint8` and returns a Boolean value, either `true` or `false`. Accordingly, the call is made to the contract, as shown here:

```
var txn = simplecontractinstance.Matcher.call(12);
```

In the preceding example, `console.log` is used to print the value returned by the function call. Once the result of the call is available in the `txn` variable, it can be used anywhere throughout the program, for example, as a parameter for another JavaScript function.

Finally, the HTML file named `index.html` is created with the following code. This HTML file will serve as the frontend UI for the users, who can browse to this page served via an HTTP server to interact with the smart contract:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the
<button onclick="myFunction()">get decision</button>
```

```
<p id="decision"></p>
<p>Calling the contract manually with hardcoded values, result 1<br/>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

It is recommended that a web server is running in order to serve the HTML content (`index.html` as an example). Alternatively, the file can be browsed from the filesystem but that can cause some issues related to serving the content correctly with larger projects; as a good practice, always use a web server.

A web server in Python can be started using the following command. This server will serve the HTML content from the same directory that it has been run from:

```
$ python -m SimpleHTTPServer 7777
Serving HTTP on 0.0.0.0 port 7777 ...
```



The web server does not have to be in Python; it can be an Apache server or any other web container.

Now any browser can be used to view the webpage served over TCP port `7777`. This is shown in the following screenshot:

```

> Contract {_eth: Eth, transactionHash: null, address: "0x82012b7601fd23669b50bb7dd79460970ce386e3", abi: Array(2), Matcher: f, ...}
  return value: true
  return value: false
  return value: false
  return value: true
  return value: false

```

Figure 15.7: Interaction with the contract



It should be noted that the output shown here is in the browser's console window. The browser's console must be enabled in order to see the output. For example, in Chrome you can use keyboard shortcuts to open the console. On Windows and Linux, Ctrl + Shift + J, and on Mac, Cmd + Option + J are used.

As the values are hardcoded in the code for simplicity, two buttons shown in the screenshot, **callTrue** and **callFalse**, have been created in `index.html`. Both of these buttons call functions with hardcoded values. This is just to demonstrate that parameters are being passed to the contract via Web3 and values are being returned accordingly.

There are three functions being called behind these buttons. We will describe them as follows:

1. The `get decision` button returns the decision from the contract:

```

<button onclick="myFunction () ">get decision</button>
function myFunction()
{
  var x = document.getElementById("txtValue").value;

```

```
    var txn = simplecontract.Matcher.call(x);  
};  
console.log("return value: " + txn);  
document.getElementById("decision").innerHTML = txn;  
}
```

The `get decision` button invokes the smart contract's `Matcher` function with the value entered on the webpage. The variable `x` contains the value passed to this function via the webpage, which is `12`. As the value is `12`, which is greater than `10`, the `get decision` button will return `true`.

2. The `callTrue` button will call the `Matcher` function with a value that is always greater than 10, such as 12, returning always `true`. The `callMatchertrue()` method has a hardcoded value of `12`, which is sent to the contract using the following code:

```
simplecontractinstance.Matcher.call(12)
```

The return value is printed in the console using the following code, which first invokes the `Matcher` function and then assigns the value to the `txn` variable to be printed later in the console:

```
simplecontractinstance.Matcher.call(1) function callMatcher()  
{  
var txn = simplecontractinstance.Matcher.call(12);{  
};  
console.log("return value: " + txn);  
}
```

3. The `callFalse` button: invokes the `callMatcherfalse()` function. The `callMatcherfalse()` function works by passing a hardcoded value of `1` to the contract using this code:

```
simplecontractinstance.Matcher.call(1)
```

The return value is printed accordingly:

```
console.log("return value: " + txn);  
function callMatcherfalse()  
{  
var txn = simplecontractinstance.Matcher.call(1);{  
};
```

```
    console.log("return value: " + txn);
}
```



Note that there is no real need for the `callTrue` and `callFalse` methods here; they are just presented for pedagogical reasons so that readers can correlate the functions with the hardcoded values and then to the called function within the smart contract, with `value` as a parameter.

This example demonstrates how the Web3 library can be used to interact with the contracts on the Ethereum blockchain. First, we created a web frontend using the JavaScript `app.js` file and the HTML file. We also included the Web3 library in our HTML so that we could create the `web3` object and use that to interact with the deployed smart contract.

In the next section, we will explore some development frameworks that aid Ethereum development, including a commonly used framework called Truffle.

Development frameworks

There are various development frameworks now available for Ethereum. As seen in the examples discussed earlier, it can be quite time-consuming to deploy the contracts via manual means. This is where **Truffle** and similar frameworks such as Embark can be used to make the process simpler and quicker. We have chosen Truffle because it has a more active developer community and is currently the most widely used framework for Ethereum development. However, note that there is no best framework as all frameworks aim to provide methods to make development, testing, and deployment easier.



You can read more about Embark here:
<https://github.com/embark-framework/embark>.

In the next section, you will be introduced to an example project to demonstrate the usage of the Truffle framework.

Using Truffle to develop a decentralized application

We discussed Truffle briefly in *Chapter 14, Development Tools and Frameworks*. In this section, we will see an example project that will demonstrate how Truffle can be used to develop a decentralized application. We will see all the steps involved in this process such as initialization, testing, migration, and deployment. First, we will see the installation process.

Installing and initializing Truffle

If Truffle is not already installed, it can be installed by running the following command:

```
$ npm install -g truffle
```

Next, Truffle can be initialized by running the following commands. First, create a directory for the project, for example:

```
$ mkdir testdapp
```

Then, change the directory to the newly created `testdapp` and run the following command:

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

Once the command is successful, it will create the directory structure, as shown here. This can be viewed using the `tree` command in Linux:

```
$ tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
└── test
    └── truffle-config.js
3 directories, 3 files
```

This command creates three main directories named `contracts`, `migrations`, and `test`. As seen in the preceding example, a total of 3 directories and 4 files have been created. The directories are defined here:

- `contracts`: This directory contains Solidity contract source code files. This is where Truffle will look for Solidity contract files during migration.
- `migration`: This directory has all the deployment scripts.
- `test`: As the name suggests, this directory contains relevant test files for applications and contracts.

Finally, Truffle configuration is stored in the `truffle.js` file, which is created in the root folder of the project from where `truffle init` was run.

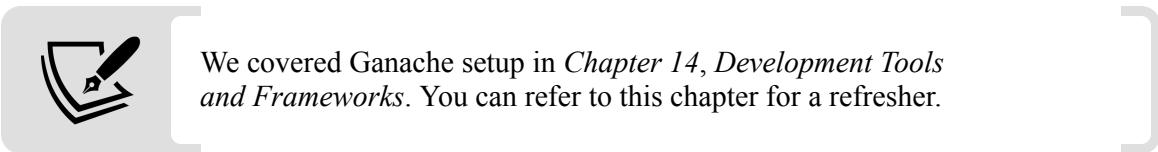
When `truffle init` is run, it will create a skeleton tree with files and directories. In previous versions of Truffle, this used to produce a project named **MetaCoin** but now, this project is now available as a **Truffle box**. Now that we have initialized Truffle, let's see how it is used to compile, test, and migrate smart contracts.

Compiling, testing, and migrating using Truffle

In this section, we will demonstrate how to use various operations available in Truffle. We will introduce how to use compilation, testing, and migration commands in Truffle to deploy and test **Truffle boxes**, which are essentially

sample projects available with Truffle. We will use the MetaCoin Truffle box. Later, further examples will be shown on how to use Truffle for custom projects.

We will use Ganache as a local blockchain to provide the RPC interface. Make sure that Ganache is running in the background and mining.



In the following example, Ganache is running on port 7545 with 10 accounts. These options can be changed in the **Settings** option in Ganache as shown in the following screenshot:

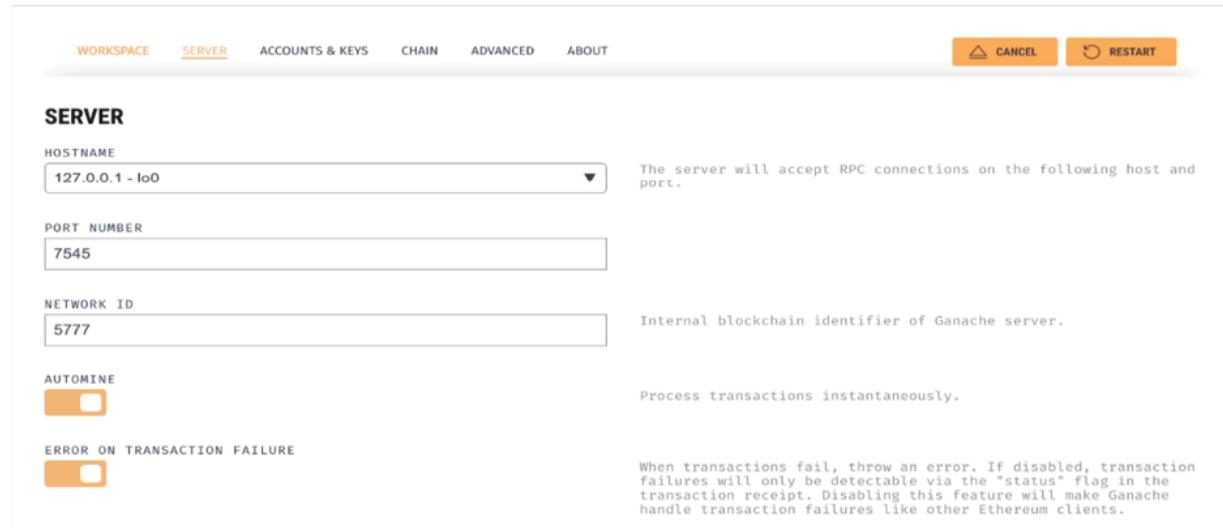


Figure 15.8: Ganache settings

We will use the Ganache workspace that we saved in *Chapter 14, Development Tools and Frameworks*. Alternatively, a new environment can also be set up.

ADDRESS	BALANCE	TX COUNT	INDEX	
<code>0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA</code>	<code>99.99 ETH</code>	13	0	
<code>0x6805940005154aEdfe6a00A39C588ED668E64D9D</code>	<code>100.00 ETH</code>	0	1	
<code>0x56C21294F4e17dF32486b3d4D12E72D023861edF</code>	<code>100.00 ETH</code>	0	2	
<code>0xAfACDB553412071bB538E36d57ED92d6745C5f94</code>	<code>100.00 ETH</code>	0	3	
<code>0x694AE93a42C43B8E3d10c3F6769Adf2566C1863B</code>	<code>100.00 ETH</code>	0	4	
<code>0x52eA83ae62213f076E3A93054F8168907AaB3313</code>	<code>100.00 ETH</code>	0	5	

Figure 15.9: Ganache view

After the successful setup of Ganache, the following steps need to be performed in order to unpack the webpack Truffle box and run the MetaCoin project. This example provides a solid foundation for the upcoming sections. With this exercise, we will learn how a sample project available with Truffle can be downloaded, and how we perform compilation, testing, and migration of the contracts available with this sample onto Ganache:

1. First, create a new directory:

```
$ mkdir tproject
$ cd tproject
```

2. Now, unbox the webpack sample from Truffle:

```
$ truffle unbox metacoin
[✓] Preparing to download box
[✓] Downloading
[✓] cleaning up temporary files
[✓] Setting up box
```

3. Edit the `truffle.js` file: if Ganache is running on a different port then change the port from the default to where Ganache is listening. Note the settings provided in the screenshot in *Figure 15.8*:

```
$ cat truffle-config.js
module.exports = {
  // Uncommenting the defaults below
  // provides for an easier quick-start with Ganache.
  // You can also follow this format for other networks;
  // see <http://truffleframework.com/docs/advanced/configu
  // for more details on how to specify configuration option
  //
  //networks: {
  //  development: {
  //    host: "127.0.0.1",
  //    port: 7545,
  //    network_id: "*"
  //  },
  //  test: {
  //    host: "127.0.0.1",
  //    port: 7545,
  //    network_id: "*"
  //  }
  //}
  //
};

};
```

Edit the file and uncomment the defaults. The file should look like the one shown here:

```
module.exports = {
  // Uncommenting the defaults below
  // provides for an easier quick-start with Ganache.
  // You can also follow this format for other networks;
  // see <http://truffleframework.com/docs/advanced/configu
  // for more details on how to specify configuration option
  //
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    },
    test: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    }
  }
};
```

Now, after unboxing the webpack sample and making the necessary configuration changes, we are ready to compile all the contracts.

4. Now run the following command to compile all the contracts:

```
$ truffle compile
```

This will show the following output:

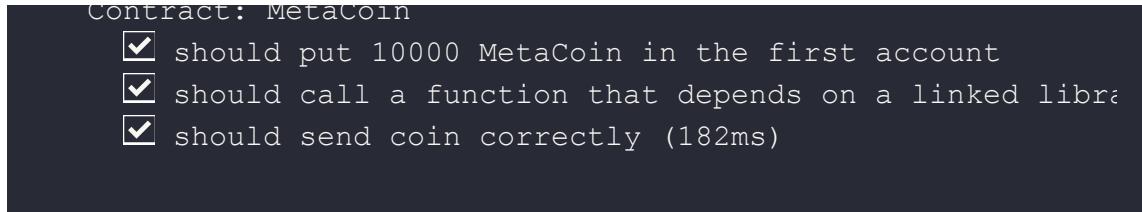
```
Compiling your contracts...
=====
> Compiling ./contracts/ConvertLib.sol
> Compiling ./contracts/MetaCoin.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/drequinox/tproject/build/cont:
> Compiled successfully using:
  - solc: 0.5.16+commit.9c3226ce.Emscripten clang
```

5. Now we can test using the `truffle test` command as shown here:

```
$ truffle test
```

This command will produce the output shown as follows, indicating the progress of the testing process:

```
Using network 'development'.
Compiling your contracts...
=====
> Compiling ./test/TestMetaCoin.sol
> Artifacts written to /var/folders/82/5r_y_y_13wq4nqb0fw6s
> Compiled successfully using:
  - solc: 0.5.16+commit.9c3226ce.Emscripten clang
TruffleConfig {
  _deepCopy: [ 'compilers' ],
  _values:
  .
  .
  . . . . . text not shown for brevity . . . .
  .
  test_files:
    [ '/Users/drequinox/tproject/test/metacoin.js',
      '/Users/drequinox/tproject/test/TestMetaCoin.sol' ] }
TestMetaCoin
  ✓ testInitialBalanceUsingDeployedContract (122ms)
  ✓ testInitialBalanceWithNewMetaCoin (186ms)
```



We can also see that in Ganache, the transactions are being processed as a result of running the test:

TX HASH	FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE
0x1caa31ace52b8e09711e431e78bffb21b1aa3ee436ddad7d7b3c69708658d3	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0x83B0ECf99994469535dB2C41178Cf912ED91D58A	50880	0
0x31473d8d65e1be80742cc59a511b9a969fc27f35b158bc573a392127751d0ca5	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0xC8293bfb4Db0580dD64139D52667f7d5ab6267da	27001	0
0xab113d9f6829378bb6e545b91c8e02cf3e74bd2992912006635c6f9afa387cee	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0x83B0ECf99994469535dB2C41178Cf912ED91D58A	340697	0
0xfaa362eb0d4daf693051732ab27320f77d7b1733376f9ad68d5a83fe1d646fff	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0x582C799161dcD373Fd8dAAF1347fa2c29DA6348B	105974	0

Figure 15.10: Truffle screen as a result of testing

- Once testing is completed, we can migrate to the blockchain using the following command. Migration will use the settings available in `truffle.js` that we edited in the second step of this process to point to the port where Ganache is running. This is achieved by issuing the following command:

```
$ truffle migrate
```

The output is shown as follows. Notice that when migration runs it will reflect on Ganache; for example, the balance of accounts will go down and you can also view transactions that have been executed. Also notice that the account shown in the screenshot corresponds with what is shown in Ganache:

```

Starting migrations...
=====

```

```
> Network name:      'development'
> Network id:       5777
Block gas limit: 0x6691b7
1_initial_migration.js
=====
Deploying 'Migrations'
-----
transaction hash: 0x47ce80861f3965036a4c6b78cb4cc03b3a77
> Blocks: 0          Seconds: 0
> contract address: 0xe91Ff793A3e328672c0d4B6A837679bbB0
> block number:     20
> block timestamp:  1581113420
> account:          0x2366e9848803cB00CB82E6E6De3F6D17C4
> balance:          99.97009608
> gas used:         188483
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00376966 ETH
Saving migration to chain.
Saving artifacts
-----
> Total cost:       0.00376966 ETH
2_deploy_contracts.js
=====
Deploying 'ConvertLib'
-----
transaction hash: 0xfe01171736d2a01602e19d23995163766cfde
> Blocks: 0          Seconds: 0
> contract address: 0x71E41231ee8546970897A46F4F34B7AF00
> block number:     22
> block timestamp:  1581113421
> account:          0x2366e9848803cB00CB82E6E6De3F6D17C4
> balance:          99.96713658
> gas used:         105974
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00211948 ETH
Linking
-----
Contract: MetaCoin <--> Library: ConvertLib (at address: 0xE515F3ce9Eb980215e68D34826E9cD0978)
Deploying 'MetaCoin'
-----
transaction hash: 0x9ad9e47f9c9caa6c0ea7e4a1e3348808fec4
> Blocks: 0          Seconds: 0
> contract address: 0xE515F3ce9Eb980215e68D34826E9cD0978
> block number:     23
> block timestamp:  1581113421
> account:          0x2366e9848803cB00CB82E6E6De3F6D17C4
... 00000000
```

```

> balance:          99.96032392
> gas used:         340633
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00681266 ETH
Saving migration to chain.
Saving artifacts
-----
> Total cost:      0.00893214 ETH
Summary
=====
> Total deployments: 3
> Final cost:       0.0127018 ETH

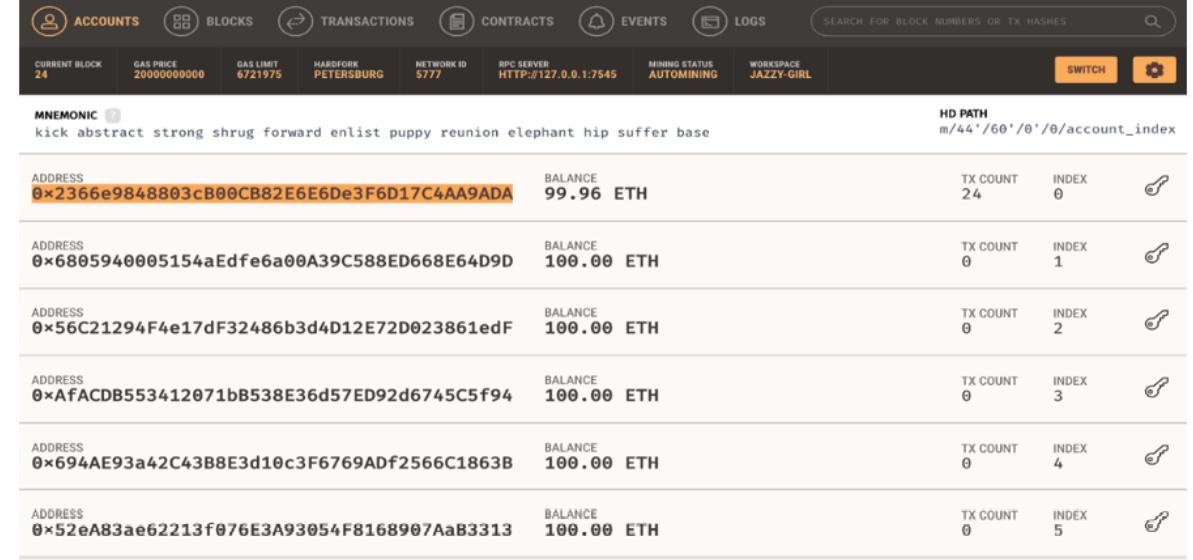
```

Notice that the migration we have just performed is reflected in Ganache with the accounts shown previously:

TX HASH	FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE
0x887cad1f30eec87b995fdे784099ff4d5c429616e1616c55f97c3bf1051c0c9c	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0xe91FF793A3e328672c0d4B6A837679bbB028E238	27801	0
0x9ad9e47f9c9caa6c0ea7e4a1e3348808fec4771752999ed2a92c83a9a0fde16c	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0xE515F3ce9Eb980215e68D34826E9cD097829E75F	340633	0
0xe01171736d2a01602e19d23995163766cfdd9aae86356c4c930580ccd2764c8	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0x71E41231ee854697897A46F4F34B7AF007cc583	105974	0
0x9e9940b87b60a0d97f88a7ddc759b53b938188bd1d556263fd9d4e7108f826d	0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	0xe91FF793A3e328672c0d4B6A837679bbB028E238	42001	0

Figure 15.11: Ganache displaying transactions

You can see **BALANCE** updating in Ganache, as the transactions run and ETH is consumed:



The screenshot shows the Ganache interface with the following details:

- MNEMONIC:** kick abstract strong shrug forward enlist puppy reunion elephant hip suffer base
- HD PATH:** m/44'/60'/0'/0/account_index
- CURRENT BLOCK:** 24
- GAS PRICE:** 20000000000
- GAS LIMIT:** 6721975
- HARDFORK:** PETERSBURG
- NETWORK ID:** 5777
- RPC SERVER:** HTTP://127.0.0.1:7545
- MINING STATUS:** AUTOMINING
- WORKSPACE:** JAZZY-GIRL
- SWITCH** and **Gear** icons

ADDRESS	BALANCE	TX COUNT	INDEX	Copy
0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	99.96 ETH	24	0	🔗
0x6805940005154aEdfe6a00A39C588ED668E64D9D	100.00 ETH	0	1	🔗
0x56C21294F4e17dF32486b3d4D12E72D023861edF	100.00 ETH	0	2	🔗
0xAfACDB553412071bB538E36d57ED92d6745C5f94	100.00 ETH	0	3	🔗
0x694AE93a42C43B8E3d10c3F6769AdF2566C1863B	100.00 ETH	0	4	🔗
0x52eA83ae62213f076E3A93054F8168907AaB3313	100.00 ETH	0	5	🔗

Figure 15.12: Ganache displaying accounts

In the Ganache workspace, we can also add the Truffle project to enable extra features. For this, click on the upper right-hand corner gear icon, and open the settings screen. Add the Truffle project as shown here:

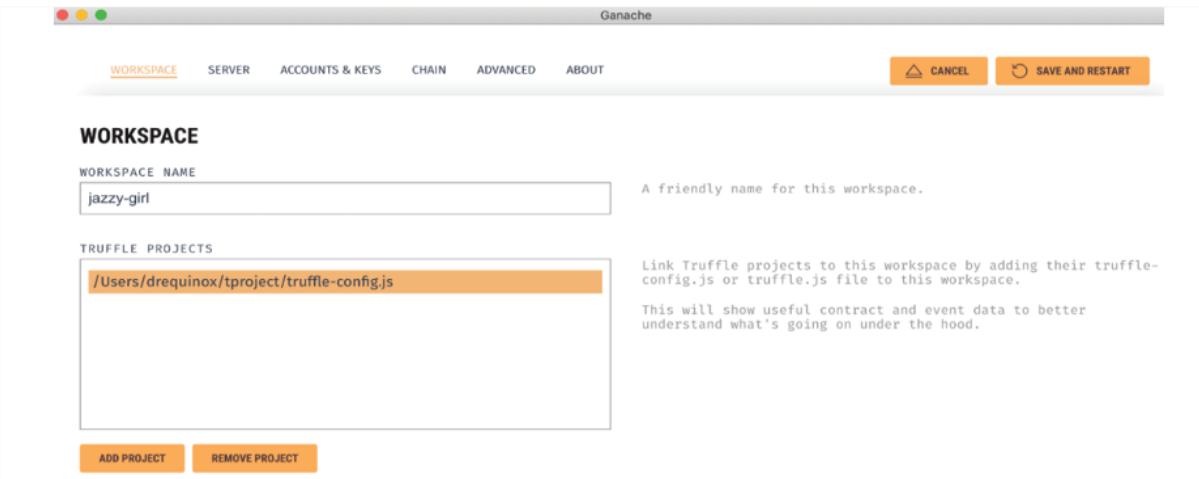


Figure 15.13: Adding the Truffle project to Ganache

Save and restart to commit the changes. When Ganache is back up again, you will be able to see additional information about the contract as shown in the following screenshot:

The screenshot shows the Ethereum Wallet interface. At the top, there are navigation links: ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, LOGS, and a search bar. Below the header, there are several status indicators: CURRENT BLOCK (29), GAS PRICE (2000000000), GAS LIMIT (6721975), HARDFORK (PETERSBURG), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:7545), MINING STATUS (AUTOMINING), and WORKSPACE (JAZZY-GIRL). On the right side, there are 'SWITCH' and '⚙️' buttons.

The main content area displays a table of deployed contracts for the project 'tproject' located at '/Users/drequinox/tproject'. The table has columns for NAME, ADDRESS, TX COUNT, and STATUS (DEPLOYED).

NAME	ADDRESS	TX COUNT	STATUS
ConvertLib	0x58e62Cb54a8b0bd8bdc61061f24A98D88683197C	0	DEPLOYED
MetaCoin	0xeA0DB2Cee9093D7eA4230C63B55AA1cD79739c7C	0	DEPLOYED
Migrations	0x087DF90f98b81602B7b91095D3a97688317632E0	1	DEPLOYED

Figure 15.14: More contract details are visible after adding the Truffle project

In this section so far, we've explored how Truffle can be used to compile, test, and deploy smart contracts on the blockchain. We used Ganache, the Ethereum personal blockchain (a simulated version of the Ethereum blockchain), to perform all the exercises.



Note that you will see slightly different outputs and screen depending on your local environment and Ganache and Truffle versions.

Now, we can interact with the contract using the Truffle console. We will explore this in the following section.

Interacting with the contract

Truffle also provides a console (a CLI) that allows interaction with the contracts. All deployed contracts are already instantiated and ready to use in the console. This is an **REPL**-based interface, meaning **Read**, **Evaluate**, and **Print Loop**. Similarly, in the `geth` client (via `attach` or `console`), REPL is used via exposing the **JavaScript runtime environment (JSRE)**.

1. The console can be accessed by issuing the following command:

```
$ truffle console
```

2. This will open a CLI:

```
truffle (development) >
```

Once the console is available, various methods can be run in order to query the contract. A list of methods can be displayed by typing the preceding command and tab-completing:

```
|truffle(development)> MetaCoin.  
MetaCoin.__defineGetter__ MetaCoin.__defineSetter__ MetaCoin.__lookupGetter__ MetaCoin.__lookupSetter__  
MetaCoin.__proto__ MetaCoin.hasOwnProperty MetaCoin.isPrototypeOf MetaCoin.propertyIsEnumerable  
MetaCoin.toLocaleString MetaCoin.valueOf  
  
MetaCoin.apply MetaCoin.bind MetaCoin.call MetaCoin.constructor  
MetaCoin.toString  
  
MetaCoin._constructorMethods MetaCoin._json MetaCoin._properties MetaCoin._property_values  
MetaCoin.abi MetaCoin.addProp MetaCoin.address MetaCoin.arguments  
MetaCoin.ast MetaCoin.at MetaCoin.autoGas MetaCoin.binary  
MetaCoin.bytecode MetaCoin.caller MetaCoin.class_defaults MetaCoin.clone  
MetaCoin.compiler MetaCoin.configureNetwork MetaCoin.contractName MetaCoin.contract_name  
MetaCoin.currentProvider MetaCoin.decodeLogs MetaCoin.defaults MetaCoin.deployed  
MetaCoin.deployedBinary MetaCoin.deployedBytecode MetaCoin.deployedSourceMap MetaCoin.detectNetwork  
MetaCoin.devdoc MetaCoin.ens MetaCoin.events MetaCoin.gasMultiplier  
MetaCoin.hasNetwork MetaCoin.interfaceAdapter MetaCoin.isDeployed MetaCoin.legacyAST  
MetaCoin.length MetaCoin.link MetaCoin.links MetaCoin.metadata  
MetaCoin.name MetaCoin.network MetaCoin.networkType MetaCoin.network_id  
MetaCoin.networks MetaCoin.new MetaCoin.numberFormat MetaCoin.prototype  
MetaCoin.resetAddress MetaCoin.schemaVersion MetaCoin.schema_version MetaCoin.setNetwork  
MetaCoin.setNetworkType MetaCoin.setProvider MetaCoin.setWallet MetaCoin.source  
MetaCoin.sourceMap MetaCoin.sourcePath MetaCoin.timeoutBlocks MetaCoin.toJSON  
MetaCoin.transactionHash MetaCoin.unlinked_binary MetaCoin.updatedAt MetaCoin.updated_at  
MetaCoin.userdoc MetaCoin.web3
```

Figure 15.15: Available methods

3. Other methods can also be called in order to interact with the contract; for example, in order to retrieve the address of the contract, the following method can be called using the truffle console :

```
truffle(development)> MetaCoin.address  
'0xeA0DB2Cee9093D7eA4230C63B55AA1cD79739c7C'
```

This address is also shown in contract creation transaction in Ganache:

Figure 15.16: Contract creation transaction shown in Ganache

A few examples of other methods that we can call in the `truffle console` are shown here.

4. To query the accounts available, enter the following command:

```
truffle(development)> MetaCoin.web3.eth.getAccounts()
```

This will return the output shown here:

```
[ '0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA',
  '0x6805940005154aEdfe6a00A39C588ED668E64D9D',
  '0x56C21294F4e17df32486b3d4D12E72D023861edF',
  '0xAFACDB553412071bB538E36d57ED92d6745C5f94',
  '0x694AE93a42C43B8E3d10c3F6769Adf2566C1863B',
  '0x52eA83ae62213f076E3A93054F8168907AaB3313',
  '0x4d4AAD022169bE203052173C09Da3027B66258b9',
  '0xC87B00b4105bc55FD38E6EC07d242Fa4906733b8',
  '0x605203ff0e161d9730F535Fa91FA6E095FaB8462',
  '0xCf5A1df3cc67eBAa085EcA037940671e81D4DB82' ]
```

5. To query the balance of the contract, enter the following command:

```
truffle(development)> MetaCoin.web3.eth.getBalance("0x2366e
```

This will show the output shown here:

```
'99945700780000000000'
```

This is the first account shown in Ganache in the preceding screenshot, `0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA`, and the output returned a string with the value `'99945700780000000000'`.

6. To end a session with the `truffle console`, the `.exit` command is used.

This completes our introduction to the sample webpack Truffle box and the MetaCoin application using Truffle. In this section, we discovered how Truffle can be used to interact with the deployed smart contracts. MetaCoin in this section is an example of a decentralized application, however we used this merely as an example to learn how Truffle works. In this chapter's bonus content pages, we will use the techniques that we learned in this section to develop our own decentralized application.

In the next section, we will see how a contract can be developed from scratch, tested and deployed using Truffle, Ganache, and our private net.

Using Truffle to test and deploy smart contracts

This section will demonstrate how we can use Truffle for testing and deploying smart contracts. Let's look at an example of a simple contract in Solidity, which performs addition. We will see how migrations and tests can be created for this contract with the following steps.

1. Create a directory named `simple`:

```
$ mkdir simple
```

2. Change directory to `simple`:

```
$ cd simple
```

3. Initialize Truffle to create a skeleton structure for smart contract development:

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

The tree structure produced by the `init` command is as follows:

```
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
└── test
    └── truffle-config.js
3 directories, 3 files
```

4. Place the two files `Addition.sol` and `Migrations.sol` in the `contracts` directory. The code for both of these files is listed as

follows:

Addition.sol :

```
pragma solidity ^0.5.0;

contract Addition
{
    uint8 x; //declare variable x

    // define function addx with two parameters y and z, and return x
    function addx(uint8 y, uint8 z) public
    {
        x = y + z; //performs addition
    }
    // define function retrievex to retrieve the value stored in x
    function retrievex() view public returns (uint8)
    {
        return x;
    }
}
```

Migrations.sol :

```
pragma solidity >=0.4.21 <0.7.0;
contract Migrations {
    address public owner;
    uint public last_completed_migration;
    constructor() public {
        owner = msg.sender;
    }
    modifier restricted() {
        if (msg.sender == owner) _;
    }
    function setCompleted(uint completed) public restricted {
        last_completed_migration = completed;
    }
}
```

5. Now compile the contracts:

```
$truffle compile
Compiling your contracts...
=====
> Compiling ./contracts/Additions.sol
> Compiling ./contracts/Migrations.sol
```

```
> Artifacts written to /Users/drequinox/simple/build/contrac
> Compiled successfully using:
- solc: 0.5.16+commit.9c3226ce.Emscripten.clang
```

6. Under the `migration` folder, place two files `1_initial_migration.js` and `2_deploy_contracts.js` as shown here:

`1_initial_migration.js`:

```
var Migrations = artifacts.require("./Migrations.sol");
module.exports = function(deployer) {
  deployer.deploy(Migrations);
};
```

`2_deploy_contracts.js`:

```
var SimpleStorage = artifacts.require("Addition");
module.exports = function(deployer) {
  deployer.deploy(SimpleStorage);
};
```

7. Under the `test` folder, place the file `TestAddition.sol`. This will be used for unit testing:

`TestAddition.sol`:

```
pragma solidity ^0.4.2;
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/Addition.sol";
contract TestAddition {
  function testAddition() public {
    Addition adder = Addition(DeployedAddresses.Addition());
    adder.addx(100,100);
    uint returnedResult = adder.retrieve();
    uint expected = 200;
    Assert.equal(returnedResult, expected, "should result 200");
  }
}
```

8. The test is run using the command shown here:

```
$truffle test
Using network 'development'.
Compiling your contracts...
=====
> Compiling ./contracts/Addition.sol
=====
```

```

> Compiling ./test/TestAddition.sol
> Artifacts written to /var/folders/82/5r_y_y_13wq4nqb0fw6s
> Compiled successfully using:
  - solc: 0.5.16+commit.9c3226ce.Emscripten clang
TruffleConfig {
  _deepCopy: [ 'compilers' ],
.
.
.
.
.
.
TestAddition
  ✓ testAddition (313ms)
1 passing (9s)

```

9. This means that our tests are successful. Once the tests are successful, we can deploy it to the network, in our case, Ganache:

```

$ truffle migrate
Compiling your contracts...
=====
> Compiling ./contracts/Addition.sol
> Artifacts written to /Users/drequinox/simple/build/contrac
> Compiled successfully using:
  - solc: 0.5.16+commit.9c3226ce.Emscripten clang
Warning: Both truffle-config.js and truffle.js were found. !
Warning: Both truffle-config.js and truffle.js were found. !
Starting migrations...
=====
> Network name:    'development'
> Network id:      5777
> Block gas limit: 0x6691b7
2_deploy_contracts.js
=====
Deploying 'Addition'
-----
> transaction hash: 0xc9bfe69f2788cf49d8e44012b7882ab
> Blocks: 0          Seconds: 0
> contract address: 0x4B53F7227901f73b4B14fbA4Bd55601
> block number:     106
> block timestamp:  1581187718
> account:          0x2366e9848803cB00CB82E6E6De3F6D1
> balance:           98.82759464
> gas used:         122459
> gas price:        20 gwei
> value sent:       0 ETH
> total cost:       0.00244918 ETH
> Saving migration to chain.

```

```
> Saving artifacts
-----
> Total cost:          0.00244918 ETH
summary
=====
Total deployments:    1
Final cost:           0.00244918 ETH
```

In Ganache we see this activity, which corresponds to our migration activity:

Figure 15.17: Ganache deployed contract—contract creation transaction

As the `Addition` contract is already instantiated and available in the `truffle console`, it becomes quite easy to interact with the contract. In order to interact with the contract, the following methods can be used:

1. Run the following command:

\$ truffle console

This will open the `truffle console`, which will allow interaction with the contract. For example, in order to retrieve the address of the deployed contract, the following method can be called:

```
truffle(development)> Addition.address  
'0x4B53F7227901f73b4B14fbA4Bd55601B57293333'
```

2. To call the functions from within the contract, the deployed method is used with the contract functions. An example is shown here. First instantiate the contract:

```
truffle(development)> let additioncontract = await Addition
undefined
```

3. Now call the `addr` function:

```
truffle(development)> additioncontract.addx(2,2)
```

This will produce the following output indicating the execution status of the transaction that was created as a result of calling the `addrx` function:

We see the transaction in Ganache too:

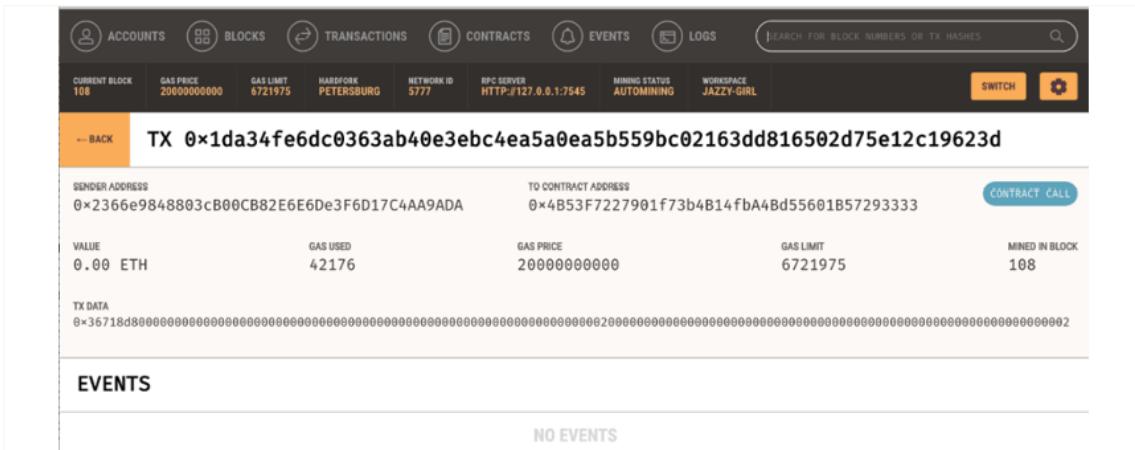


Figure 15.18: Transaction in Ganache

4. Finally, we can call the `retrievex` function to see the current value after `addition`:

```
truffle(development)> additioncontract.retrievev()  
<BN: 4>
```

Finally, we see the value `4` returned by the contract.

In this section, we created a simple smart contract that performs an addition function and learned how the Truffle framework can be used to test and deploy the smart contract. We also learned how to interact with the smart contract using the Truffle console.

In the next section, we will look at IPFS, which can serve as the decentralized storage layer for a decentralized ecosystem. We will now see how we can host one of our DApps on IPFS.

Deployment on decentralized storage using IPFS

As discussed in *Chapter 1, Blockchain 101*, in order to fully benefit from decentralized platforms, it is desirable that you decentralize the storage and communication layer too in addition to decentralized state/computation (blockchain). Traditionally, web content is served via centralized servers,

but that part can also be decentralized using distributed filesystems. The HTML content shown in the earlier examples can be stored on a distributed and decentralized IPFS network in order to achieve enhanced decentralization.



IPFS is available at <https://ipfs.io/>.

Note that IPFS is under heavy development and is an alpha release. Therefore, there is the possibility of security bugs. Security notes are available here:

<https://ipfs.io/ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/security-notes>

IPFS can be installed by following this process:

1. Download the IPFS package from <https://dist.ipfs.io/#go-ipfs> and decompress the `.gz` file:

```
$ tar xvfz go-ipfs_v0.4.23_darwin-amd64.tar
```

2. Change directory to where the files have been decompressed:

```
$ cd go-ipfs
```

3. Start the installation:

```
$ ./install.sh
```

This will produce the following output:

```
Moved ./ipfs to /usr/local/bin
```

4. Check the version to verify the installation:

```
$ ipfs version
```

This will produce the following output:

```
ipfs version 0.4.23
```

5. Initialize the IPFS node:

```
$ ipfs init
```

This will produce the following output:

```
initializing IPFS node at /Users/drequinox/.ipfs
generating 2048-bit RSA keypair...done
peer identity: QmSxkXkCwqM2qbFoxMEfjbk9w17zofXFin4ZeuxDcPRe
to get started, enter:
```

6. Enter the following command to ensure that IPFS has been successfully installed:

```
$ ipfs cat /ipfs/QmS4ustL54uo8FzR9455qaxZwuMiUhqvMcX9Ba8nUH
```

This will produce the following output:



Figure 15.19: IPFS installation

7. Now, start the IPFS daemon:

```
$ ipfs daemon
```

This will produce the following output:

```
Initializing daemon...
go-ipfs version: 0.4.23-
Repo version: 7
System version: amd64/darwin
```

```
Golang version: go1.13.7
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.0.18/tcp/4001
Swarm listening on /ip6/:1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/192.168.0.18/tcp/4001
Swarm announcing /ip4/82.2.27.41/tcp/4001
Swarm announcing /ip6/:1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/80
Daemon is ready
```

8. Copy files onto IPFS using the following command:

```
$ ipfs add . --recursive -progress
```

This will produce the following output, indicating the progress of adding the pages to IPFS:

```
added QmczxMDDHXvxr97XZMsaf3zZYUZL9fUdiTVQUjYJnpAy7C simple
added QmSjKPg7L52x33JiKsjDaVDXbK2ntxY75w7w6TBMD9pN6i simple
added QmS9SkKyt6ZdjBxnK36rbgWh46Q17A2MK3cuNYzKeKhcZc simple
added QmfSKCZ7NR5zBdfbjdUffFFuAEvZLiqPGUZwYog8u1Zg1XJ simple
added QmdVnjZvZXiJS7H7jbsKUmu4JVyz6ftSPpggy7q13w2Q8q simple
added QmWrnrBaV5ksG5E2yyvM6izV73iwfG7tvv6MrN716286Bh simple
added QmZdUWWgL2DD6JMBszyju5EcKbDN66UTQMPLCQ3kGATZj simple
added QmcxaWKQ2yPkQBmnpXmcuxPZQUMuUudmPmX3We8Z9vtAf7 simple
added QmUfCSF2HxrscKRAd7Z6NrRTo9QnkFFfDqLavYQDtGD7Wqs simple
added QmSCf4G68ZC6EWJpyE4E2kLn1LuSs3EbdZLJ27eDkLZPH3 simple
added QmfRLQWpu9tjrgX4p9nJHEWLxzdmQbQZDnxVnwYPPPEjNK simple
added QmXatyNNTYubxjRpMiZHEUz6yiqqgDmBDaVXjnM37h3HUAQ simple
added QmXT5f7TPv1BU4XbVgXYutdMeuEBvo9C7fzshmyTpxn1D5 simple
added QmPbEVcrjsd8xkE4ePJ6hNN1C9yPshLR8moAft3w3MtMRU simple
added QmTDHJX5UwMCYwoHj3hQ32E7xtKmpXg9ZhDXHntGzshdwB simple
added QmQGyeiRY9MwX3E8JjubAoziacqQf9QNH4RXeZZzN5xLH12 simple
added QmQmSgckTHGN6oCZ7YFrd5mHQGSucNt1gLjDoAZca54n5p simple
added QmUgAfZs6DfdgqfcCLYGAfFdPZ5RRoLw2ma2Y8fvvP8y4do simple
added QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv simple
```

Now it can be accessed in the browser as follows:

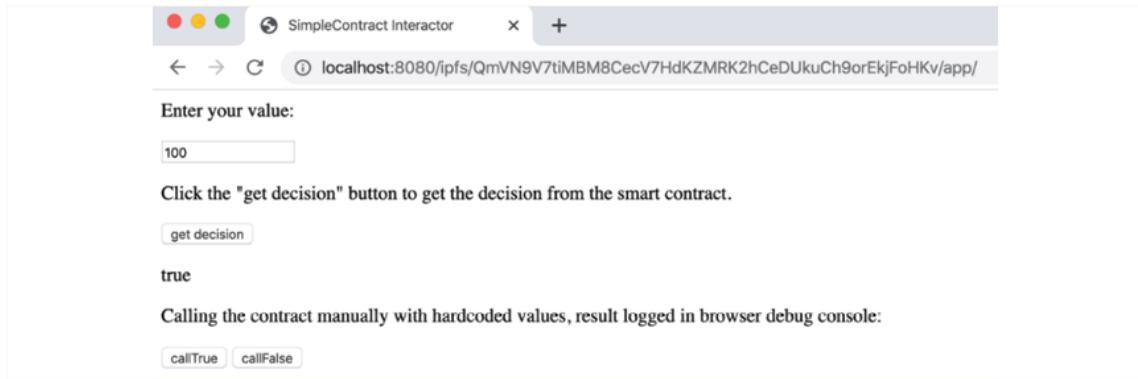


Figure 15.20: Example Truffle DApp running on IPFS and served via web host



Note that the URL is pointing to the IPFS filesystem,
`http://localhost:8080/ipfs/QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv/app/`.

- Finally, in order to make the changes permanent, the following command can be used:

```
$ ipfs pin add QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoH
```

This will show the following output:

```
pinned QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv recursi
```

The preceding example demonstrated how IPFS can be used to provide decentralized storage for the web part (UI) of smart contracts.

Remember that in *Chapter 1, Blockchain 101*, we described that a decentralized application consists of a frontend interface (usually a web interface), backend smart contracts on a blockchain, and the underlying blockchain. We have covered all these elements in this example and created a decentralized application.

To try your hand at a start-to-finish application deployment project, please go to this book's bonus online content pages here:

[https://static.packt-](https://static.packt-.)

cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf. You will build and deploy a proof of idea contract using Truffle, before creating a UI frontend for it with a tool called Drizzle!

Summary

This chapter started with the introduction of Web3. We explored various methods to develop smart contracts. Also, we saw how the contract can be tested and verified using local test blockchain before implementation on a public blockchain or private production blockchain.

We worked with various tools such as Ganache, the Geth client console, and the Remix IDE to develop, test, and deploy smart contracts. Moreover, the Truffle framework was also used to test and migrate smart contracts. We also explored how IPFS can be used to host the webpages that we created for our DApp, serving as the decentralized storage layer of the blockchain ecosystem.

In the bonus content for this chapter, which we strongly encourage you to use, we practiced the techniques we learned in this chapter, plus other advanced topics such as Drizzle, to create the frontends for DApps easily.

In the next chapter, we will introduce Serenity, Ethereum 2.0, which is the final version of Ethereum.

16

Serenity

Ethereum 1.0 is the current version of Ethereum, with Berlin being the latest release, or update. The public Ethereum mainnet available today runs the Ethereum 1.0 protocol. Serenity is the name given to Ethereum version 2.0. The vision behind Ethereum 2.0 is to ultimately transition into a more scalable, performant, and secure version of Ethereum that will serve as the **World Computer**, which is Ethereum's original vision. The concept of a world computer was introduced with Ethereum 1.0, in which a global, decentralized network of interconnected nodes runs peer-to-peer contracts without the limitation of shutting down, or being censored or hacked. This vision started with Ethereum 1.0 and gained a lot of traction; however, challenges such as scalability, privacy, and performance somewhat hindered mass adoption. Ethereum 2.0 is expected to address these issues in pursuit of becoming a world computer.

In *Chapter 11, Ethereum 101*, we saw that Ethereum is being released in a number of phases, with the first version being called Frontier and with Serenity being the projected final version. Currently, we are at Metropolitan, with the latest hard fork upgrade of Ethereum being called Istanbul. All these upgrades are taking Ethereum closer to Ethereum 2.0, which is currently being built, with Phase 0 expected to be launched by late 2020.

In this chapter, we will cover Serenity, the update that will introduce Ethereum version 2.0. We will explore the theoretical foundations and understand many concepts behind the need for and development of Ethereum 2.0. In the book's bonus content pages, which we strongly encourage readers to utilize, we will also explore how Ethereum 2.0 clients can be set up. The topics that we cover are listed as follows:

- Ethereum 2.0—an overview
- Development phases

- Architecture

Before we delve into the Ethereum 2.0 architecture and other technical details, it is important to understand the vision and motivation behind its development. We'll describe these in the next section.

Ethereum 2.0—an overview

Based on the problem presented by **Vitalik Buterin**, an Ethereum co-founder, in 2015, it is understood that only two of the three main core properties of a blockchain can be utilized at a time. These three core properties are the following:

- Decentralization
- Scalability
- Security (or consistency)

This is known as the **Scalability Trilemma** and is seen as a fundamental problem that needs to be addressed before global adoption of the Ethereum blockchain.

This concept is visualized in the following diagram:

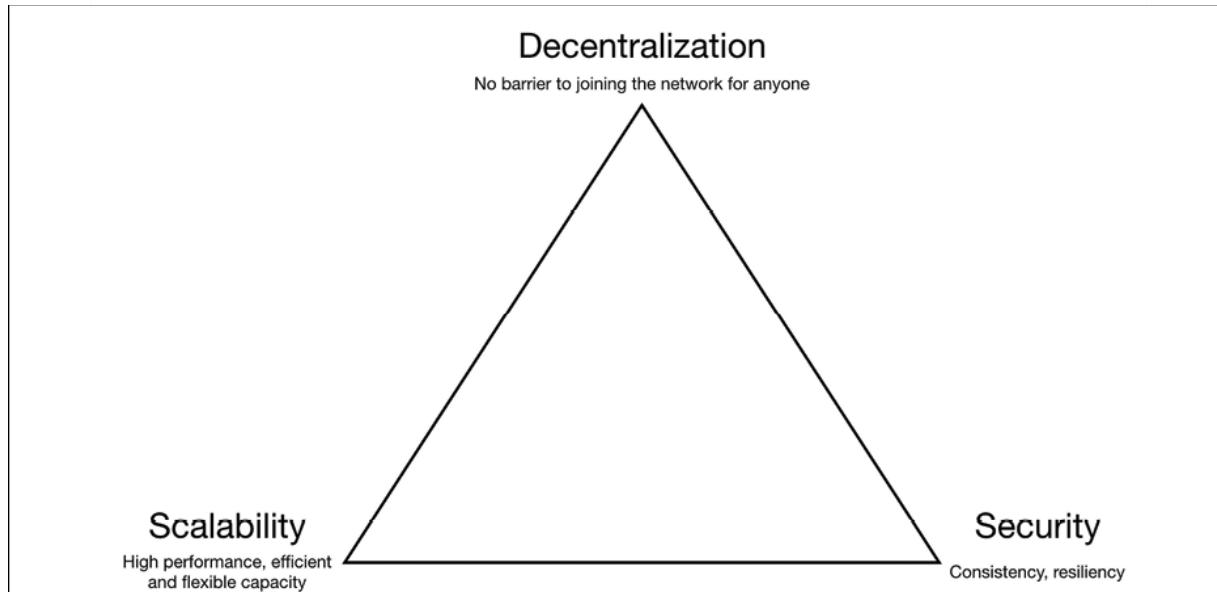


Figure 16.1: Blockchain trilemma

Intensive research is underway with Phase 0 already near release. The aim is to address this trilemma and find the right balance between all three properties, instead of compromising and only choosing two of the three.

With the release of **Proof of Stake (PoS)**, **Sharding**, and **ewasm**, a balanced combination of all these three properties, that is, decentralization, scalability, and security (consistency) can be achieved.

Having now discussed the high-level problem facing Ethereum, we can go on to look at some fundamental elements of Ethereum 2.0, starting with some specific goals.

Goals

The major goals behind the development of Serenity can be divided into the following different dimensions:

- **Reduced complexity:** One of the main aims of Ethereum 2.0 is simplicity, even if there are some tradeoffs with efficiency.
- **Resiliency:** The goal of resiliency ensures that the liveness of the network can be maintained even in the event of major network partitions.

- **Increased security:** With the advent of quantum cryptography and in order to mitigate the threats faced by cryptography in the post-quantum world, it is envisaged that in Ethereum 2.0, quantum-resistant cryptography (or at least easy pluggability of quantum-resistant elements) will be introduced, so that the blockchain remains secure even in a post-quantum world.
- **Increased participation:** It is also expected that greater participation of validators will help with improved security and overall participation.
- **Increased performance:** Instead of requiring specialist hardware or high-end CPUs and GPUs for performing validation activities, in Ethereum 2.0 it is expected that validation activities will be performed by using typical consumer computers with the usual resources. With the introduction of PoS, faster block time is expected. This also results in increased performance.

Now that we have covered the main goals of Ethereum 2.0, we can dive into its more specific features.

Main features

The main features of Serenity include a switch from a **Proof of Work (PoW)**-based chain to a PoS-based chain, which will make the network more decentralized, efficient, and faster. Secondly, **ewasm**, an improved version of the **Ethereum Virtual Machine (EVM)** is envisaged, which will make code execution faster, hence resulting in an overall performance gain.

Another feature is **sharding**, which introduces the concept of shard chains and allows increased transaction throughput and immediate finality. Finally, sidechains will be used to enhance privacy, permissions, and governance.

We touched briefly earlier on the roadmap of Ethereum, and next, we describe the roadmap in detail so that readers can understand the past, present, and future of Ethereum.

Roadmap of Ethereum

In this section, we look at the roadmap of Ethereum. The following list of Ethereum's main releases shows how Ethereum has evolved over time, along with its vision of eventually achieving a world computer—a scalable, decentralized, and secure blockchain network:

- July 2015: Frontier
- March 2016: Homestead
- October 2017: Byzantium
- February 2019: Constantinople
- December 2019: Istanbul
- Late 2020: Berlin
- Late 2020: Beacon chain—Serenity Phase 0
- 2020: Shard chains—Serenity Phase 1
- 2021: Execution environments, ewasm—Serenity Phase 2
- 2022: Further scalability and improvements—Serenity Phase 3

As indicated here, the development of Ethereum 2.0 is divided into phases. In the next section, we will discuss the four major phases of Serenity development, and what is involved in each phase.

Development phases

The four phases in the development of Ethereum 2.0 are focused on various features of the blockchain.

The main work included in **Phase 0** is around the management of validators and the staking mechanism. In addition, this phase also includes block proposer selection, consensus mechanism rules, and a reward/penalty mechanism.

Phase 1 includes the development of shard chains and blocks along with the cross-linking (anchoring) of shard blocks to the beacon chain (the core system chain).

Phase 1.5 also known as "the merge" is where ethereum mainnet will transit into Ethereum 2 most likely by becoming a shard chain of Ethereum 2.0.

The next stage, **Phase 2**, is focused on execution environment upgrades, and includes a VM upgrade to ewasm along with cross-shard communication and access to execution environments. Moreover, Phase 2 introduces the ability to run and interact with smart contracts. Currently, the specification is not final and might be subject to change.

Phase 3 is envisaged to be a continuous improvement and maintenance stage, where a complete Ethereum 2.0 chain will be running and evolving into a more mature and feature-rich blockchain.

Now, we will explain each phase and its features in detail.

Phase 0

Phase 0 is the first phase of Ethereum 2.0 development. This stage involves the creation of a core system chain called the beacon chain, which can be considered the backbone of the entire Ethereum 2.0 chain system. After Phase 0 is complete, there will be two chains in existence: the original Ethereum 1.0 chain and a new chain called the beacon chain. These chains will operate in parallel.

The Phase 0 specification and development can be further divided into multiple categories, the first of which is the beacon chain.

The beacon chain

The **beacon chain** is the core system chain, and manages the **Casper PoS** consensus mechanism for itself and all of the shard chains. It is also used to store and maintain the registry of validators. Validators are nodes that participate in the PoS consensus mechanism of Ethereum 2.0. The beacon chain has a number of features, listed as follows:

- Production of good-quality randomness, which will be used for selecting block proposers and attestation committees without bias. Currently, a

RANDAO-based scheme is prominent, however other options such as BLS-based and STARK-based schemes have been considered in the past. RANDAO is a pseudorandom process that selects proposers for each slot in every epoch and rearranges the validators in the committees.

- Provision of attestation, which essentially means availability votes for a block in a shard chain. An adequate number of attestations for a shard block will result in the creation of a **cross link**, which provides confirmation for shard blocks in the beacon chain.
- Validator and stake management.
- Provision of validator sets (committees) for voting on proposed blocks.
- Enforcement of the consensus mechanism and the reward and penalty mechanisms.
- Serving as a central system chain where shards can write their states so that cross-shard transactions can be enabled.

In order to participate in the beacon chain, some beacon chain client (or node) software is required. We will discuss this next.

Beacon nodes

The **beacon node** is the primary link in the beacon chain that forms the central core of the Ethereum 2.0 blockchain. The beacon node's responsibilities include synchronization of the beacon chain with other peers. It performs attestation of blocks, and runs an RPC server to provide critical information to the validators regarding assignments and block attestation. In addition, it also handles state transition and acts as a source of randomness for the validator assignment process.



Note that we use node and client interchangeably. However, there is a subtle difference between the terms client and node: when we say client, it actually means the software client that performs the functions, whereas a node can be seen as a combination of the client software and the hardware (computer) that it is running on. Generally, however client and node mean the same thing.

The beacon node also serves as a listener for deposit events in the Ethereum 2.0 chain, as well as creating and managing validator sets. A beacon node

synchronizes with the shards and it keeps a synchronized clock with other nodes on the beacon chain to ensure the application of penalty rules (slashing). In a beacon node, various services based on the responsibilities mentioned previously are implemented. These include the blockchain service, synchronization service, operations service, Ethereum 2.0 service, public RPC server, and the P2P/networking service.

To participate in the beacon chain, a beacon chain client, or node, is required. Based on the Ethereum 2.0 specification, this client is being developed by a number of different teams. In the supplementary resources associated with this chapter, we will look at Prysm. There are, however many other teams that are developing Ethereum 2.0 clients in a variety of languages. A non-exhaustive list is presented as follows:

Client	Language	Team	GitHub
Cortex	.NET	Nethermind	https://github.com/NethermindEth/cortex
Harmony	Java	Harmony	https://github.com/harmony-dev/beacon-chain-java
Lighthouse	Rust	Sigma Prime	https://github.com/sigp/lighthouse
Lodestar	JavaScript	ChainSafe	https://github.com/ChainSafe/lodestar
Nimbus	Nim	Status	https://github.com/status-im/nimbus
Prysm	Go	Prysmatic Labs	https://github.com/prysmaticlabs/prysm
Substrate Shasper	Rust	Parity Technologies	https://github.com/paritytech/shasper
Teku	Java	PegaSys	https://github.com/PegaSysEng/teku

By accessing this book's bonus content pages, here:

https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf, we will learn how the Ethereum 2.0 client from Prysmatic Labs can be installed and run. In addition, we will also perform some experiments to explore various features of Ethereum 2.0.

In the next section, we introduce validator nodes, which serve as block proposers and attesters on the network as part of the PoS mechanism in Ethereum 2.0. Validator nodes can be thought of in the same way as the miners in Ethereum 1.0, and are developed as per the specification finalized for Phase 0 of Ethereum 2.0.

Validator nodes

A **validator node**, also referred to as a validator client, actively participates in the consensus mechanism to propose blocks and provide attestations.

A user can stake a minimum of 32 ETH to get the ability to verify and attest to the validity of blocks. As validators participate in the consensus mechanism to secure the protocol, they are incentivized financially for their effort. The validators earn **ether** which is valid in **Ethereum 2.0 (ETH 2)** as a reward if their attestation is accepted.

A validator has to perform a number of configurations before they can deposit ether as a stake to become a validator. Once they are accepted, they become part of the validator registry that is stored and maintained on the beacon chain.

The first step for a node to become a validator is to generate a public/private key pair. For this purpose, the BLS12-381 curve is used, which is a new **zk-SNARK** curve construction.



More details on this curve are available at
<https://electriccoin.co/blog/new-snark-curve/>.

A second private/public key pair, called the BLS withdrawal key, is also required. These withdrawal credentials are used to withdraw funds.

After this, the users have to deposit a minimum of 32 ETH (normal Ethereum 1.0 PoW chain ether) to a deposit contract. Once the deposit is made and processed, the validator is activated. At this point, it becomes part of the beacon chain's set of validators.

The key functions that a validator client performs are listed as follows:

- New block proposal
- Attestation provision for blocks proposed by other validators
- Attestation aggregation
- Connection with a trusted beacon node to listen for new validator assignments and shuffling
- Shard chain synchronization with the beacon chain

A validator is assigned a status based on its current state. It can have six statuses: **deposited**, **pending**, **activated**, **slashed**, **exited**, and **withdrawable**. Deposited status means that a validator has made a valid deposit and is registered in the beacon chain state. Pending means that the validator is eligible for activation. Activated means that the activator is active. Slashed means the validator has lost a portion of their stake, exited means the validator has exited from the validator set, and withdrawable indicates that the validator is able to withdraw funds. A withdrawable state occurs after a validator has exited and roughly 27 hours have passed. A validator can be in multiple states simultaneously—for example, a validator can be in both the active and slashed states at the same time.

Honest validator behavior is incentivized as part of the PoS mechanism, and dishonest behavior results in what is referred to as **slashing**. Slashing results in the removal of a validator from the validator committee (the active validator set) and a burning of a portion of its deposited funds.

Slashing serves two purposes. First, it makes it prohibitively expensive to attack Ethereum 2.0, and secondly, it encourages (enforces) the validators to

actively perform their duties (functions). For example, a validator going offline when it is supposed to be actively participating in the consensus is penalized for its inactivity.

There are three scenarios when slashing can occur. First is proposer slashing, which is when a validator signs two different blocks on the beacon chain in the same epoch. The second case where slashing can occur is when a validator signs two conflicting attestations. Thirdly, slashing can also occur when a validator, when attesting, signs an attestation that surrounds another attestation. In other words, it means that this scenario occurs when a validator first attests to one version of the chain and then later attests another version of the chain, resulting in confusion as to which chain the validator actually supports.

When any of the three preceding scenarios occur, the offending node is reported by a **whistleblowing validator**. The whistleblowing node creates a message containing evidence of the offense, sends it to a proposer to include it in a block, and propagates it on the network. In Phase 0, the total slashing reward is taken by the proposer and the whistleblowing validator does not get any reward. This has the potential to change in Phase 1, where both actors might be rewarded.

Also, remember that penalization and slashing are two related but different concepts. Penalization results in a decrease in the balance of a validator as result of, for example, being inactive or going offline. On the other hand, slashing results in a forceful exit from the beacon chain along with the responsible validator's deposit being penalized in every epoch for as long as it has to wait for its turn in the exit queue to leave the chain.

Slashing and penalty calculations are based on several factors with various variables such as the length of validator inactivity and the type of the offense that triggered the slashing. Moreover, a penalty is applied at various points in the slashing process. For example, a minimum penalty of *effective balance of slashed validator / 32* is applied when a validator proposer includes the message reporting the offence from the whistleblower in a block. After that, at the beginning of each epoch a penalty calculated as $3 * \text{base reward}$ is applied. Another penalty is applied between the time of inclusion of the whistleblowing message in a block, and the time when the slashed validator is able to withdraw.

Earning rewards by staking ether in the deposit contract also depends on several factors. A simple example is that if someone stakes 32 ETH with a current price of, for example, USD 240 per ETH, with validator uptime of 80%, the annual interest earned will be around 8%. The base reward is calculated as per the following formula from the Phase 0 specification:

$$\text{base_reward} = \text{effective_balance} * \text{BASE_REWARD_FACTOR} / \text{integer_squareroot}(\text{total_balance}) / \text{BASE_REWARDS_PER_EPOCH}$$

Here, *BASE_REWARD_FACTOR* is set at the default value of 64, and the *BASE_REWARDS_PER_EPOCH* value is 4. The effective balance is used to calculate the proportion of rewards and penalties applied to a validator. It is based on the validator's balance and the previous effective balance. The maximum effective balance can always be only up to 32 ETH. Even if the actual balance is 1000 ETH, the effective balance will still be 32 ETH.



More details on this can be found in the Ethereum 2.0 Phase 0 specifications. Rewards and penalties are specified at <https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/beacon-chain.md#rewards-and-penalties-1> in the Ethereum 2.0 specification.

There are various calculators available online that can calculate the expected return of staking ether (from Ethereum 1.0) in the Ethereum 2.0 network.



A calculator is available here:
<https://docs.google.com/spreadsheets/d/15tmPOvOgi3wKxJw7KQJKoUe-uonbYR6HF7u83LR5Mj4/htmlview#>

As these sections have demonstrated, in Ethereum 2.0 Phase 0 there are three main components: the beacon chain, beacon nodes, and validator nodes. In the next section, we highlight some key differences between a beacon node and a validator node.

Beacon and validator node comparison

The key differences between the beacon and validator nodes are listed in the following table. The architecture of Ethereum 2.0 is no longer a single node-based architecture like Ethereum, where there is only a single type of node that can perform all functions, such as mining and securing the chain.

In Ethereum 2.0, there are two distinct types of nodes: beacon chain nodes and validator nodes. The key differences between these node types are presented in the following table.

Feature	Beacon nodes	Validator nodes
Networking	Connected via P2P to other beacon nodes	Dedicated connection with a single beacon node
Staking	No staking requirements to participate in the network	Ether staking required to participate in the network
Block creation	Attest validations and propagate blocks across the beacon chain	Propose and sign blocks
Operation	Read	Write

Being a PoS-based blockchain, Ethereum 2.0 must have the ability to take deposits from users as a stake. This requirement in Phase 0 is addressed by a deposit contract. We will explain the features of a deposit contract next.

Deposit contracts

Deposit contracts were created on the Ethereum 1.0 chain. This kind of smart contract is used for depositing ETH on the beacon chain. Currently, there is a test deposit contract written in Vyper available on the Goerli testnet for curious readers, which can be accessed at the following address:

<https://goerli.etherscan.io/address/0x0F0Fc0530007361933EaB5DB97d09aCDD6C1c8#code>



Also note that once a deposit is made to this contract, the deposited ether is effectively burned and no longer useable on the Ethereum 1.0 chain. Therefore this process must be done with care, especially on the production beacon chain (which is due to be released in late 2020).

An event is emitted every time a deposit is made. This event is composed of several fields as shown here:

```
DepositEvent (bytes pubkey, bytes withdrawal_credentials, bytes an
```

This log is consumed by the beacon chain.



The code of the deposit contract is available here:

https://github.com/ethereum/eth2.0-specs/blob/master/deposit_contract/contracts/validator_registration.vy

Another important feature of the Ethereum 2.0 beacon chain is a new rule for deciding which chain to choose in the case of a fork. We will discuss that next.

Fork choice

The fork choice rule in Ethereum 2.0 is called **Latest Message Driven Greediest Heaviest Observed SubTree**, (or **LMD GHOST**).

Remember we discussed the **Greediest Heaviest Observed SubTree** rule, **GHOST**, in the context of Ethereum mining in *Chapter 11, Ethereum 101*. LMD GHOST is a variant of GHOST that was implemented in Ethereum with some modifications. More information can also be found at <https://ethereum.org/en/whitepaper/#modified-ghost-implementation>.

LMD GHOST governs a fork-handling mechanism to ensure that in the case of a fork, the correct and honest chain is automatically chosen. As a general rule, the honest chain is the one that has the most attestations and stake

(weight). In the event of a fork, the clients will use this fork choice rule to select the correct, honest chain. Forks may occur due to the actions of colluding participants, however, beacon chains' random selection of validators mitigates that to some extent, because validators do not know in advance when they will be selected.



The original research paper on the topic is available at the following URL:

Buterin, V., Hernandez, D., Kamphefner, T., Pham, K., Qiao, Z., Ryan, D., Sin, J., Wang, Y. and Zhang, Y.X., 2020. *Combining GHOST and Casper*. arXiv preprint arXiv:2003.03052.

<https://arxiv.org/pdf/2003.03052>

Another mechanism relevant to chain integrity is finality, which is the assurance that a block, once finalized, will not revert back. To achieve this, a mechanism called **Casper the friendly finality gadget (Casper FFG)** is implemented in Ethereum 2.0.



The original paper on the topic is available at the following URL:

Buterin, V. and Griffith, V., 2017. *Casper the friendly finality gadget*. arXiv preprint arXiv:1710.09437.

<https://ethresear.ch/uploads/default/original/1X/1493a5e9434627fcf6d8ae62783b1f687c88c45c.pdf>

In the next section, we introduce another important element that deals with all the networking requirements of Ethereum 2.0: the P2P interface.

P2P interface (networking)

This element deals with the networking interfaces and protocols for the Ethereum 2.0 blockchain. There are three main dimensions addressed in the development of Ethereum P2P/networking elements:

- The gossip domain
- The discovery domain

- The request/response domain



Currently, libP2P is used in various clients for this purpose. More details on this topic are available at <https://libp2p.io>.

The networking specification also covers the essentials of a test network where multiple clients can run simultaneously, that is, the interoperability of the test network and mainnet launch.

In order to achieve interoperability all Ethereum 2.0 client implementations must support the TCP libp2p transport. This must also be enabled for both incoming and outgoing connections.



Incoming connections are also called inbound connections or listening.
Outgoing connections are also called outbound connections or dialing.

Implementors may choose not to implement an IPv4 addressing scheme for the mainnet. Ethereum 2.0 may implement an inbound connectivity feature only for IPv6, but the clients must support both inbound and outbound connections for both IPv4 and IPv6 addresses.

Some further products of Phase 0 development are introduced next.

ETH 2

ETH 2 is the new digital currency for Ethereum 2.0, which is a new asset of validator nodes (stakers) in the consensus mechanism. It will be created as an incentive (reward) for validating the beacon chain and shards, and by purchasing it via the deposit contract using ETH 1 currency (also called a registration contract).

ETH 1 is expected to transition into ETH 2 completely, but ETH 1 may still exist alongside ETH 2. It is speculated that ETH 1 may become a shard chain for the beacon chain, or ETH 1 could be left alone and only new shard chains will be created. Also, due to the current vast ecosystem built around Ethereum 1.0 and the technical complexities of the transition to Ethereum 2.0 as a shard

chain, it may result in some insuperable challenges, which could lead to Ethereum 1.0 being left as it is.

Simple Serialize

Simple Serialize, or **SSZ** for short, is the algorithm standard for providing serialization abilities for all data structures in Ethereum 2.0 client software. SSZ has support for many data types such as Boolean (`bool`), unsigned integers (`uint8`, `uint16`, `uint32`, and `uint64`), slices, arrays, structs, and pointers.

BLS cryptography

BLS cryptography (BLS12-381) is used extensively in Ethereum 2.0 to provide security and efficient verification of digital signatures. **BLS**, short for **Boneh-Lynn-Shacham** signatures, allows the aggregation of cryptographic signatures to contribute to the scalability of the network. BLS is used by validator clients to sign messages, which are then aggregated and eventually verified efficiently in the distributed network, thus increasing the overall efficiency of the network.



A Go implementation of the BLS12-381 pairing is available at
<https://github.com/phoreproject/bls>.

This completes our introduction to phase 0 of Ethereum 2.0. In the next section, we explore features of Phase 1.

Phase 1

The main feature of Phase 1 is shard chains. The specification is not fully complete yet, however, most of the details are complete.

Shard chains

Shard chains are the main scalability feature that will initially consist of 64 chains and will grow over time as required. The main design goals in this phase revolve around creation, validation, and consensus of the data of shard chains. The state of each shard chain will be written periodically to the beacon chain. This is called a **crosslink**, which is a set of signatures from a set of validators (the committee) that has attested to a block in a shard chain. This crosslink is included in the beacon chain representing the attestation of blocks of shard chains. The state is represented by the combined data Merkle root of the shard chain. The idea behind this mechanism is that when a block in the beacon chain is finalized, the corresponding block in the shard chain is also considered final. This way other shards know that what the finalized block is and they can rely on that for further transactions across different shards (cross-shard transactions).

The agreement on each block's content is achieved by utilizing randomly selected shard validators by the beacon chain for each shard at each slot. A slot is defined as a period of time in which a block proposer proposes a block for attestation. A slot may be empty or "filled" with attested blocks. Slots are 12 seconds apart. A relevant concept is that of an epoch, which represents a number of slots (currently 32) after which validators are reshuffled in their validator sets (committees).

The following diagram presents a process overview showing shards, the beacon chain, validators, the link with ETH 1, and the phases of Ethereum 2.0:

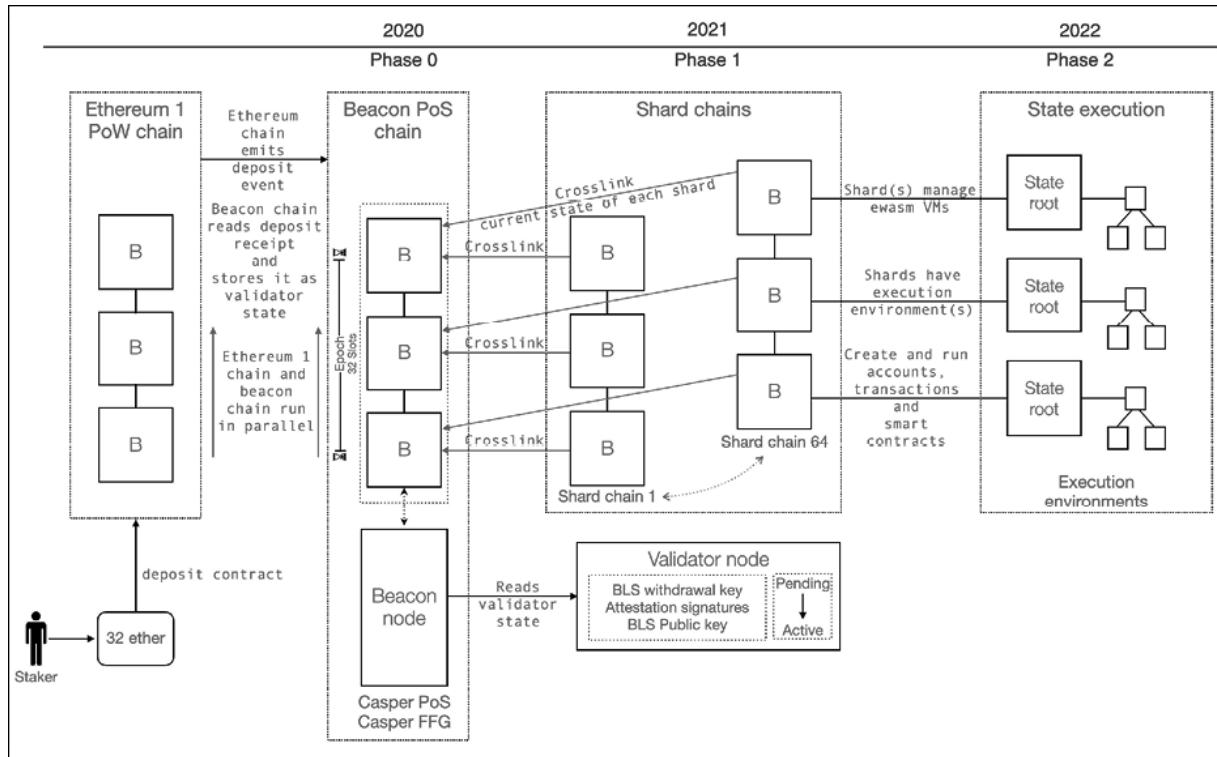


Figure 16.2: Ethereum 2.0 overall view

In Phase 1, there will be enhancements made to the Phase 0 design of the beacon chain to allow shard chains.

Transitioning from Ethereum 1 to Ethereum 2

Eventually, the aim is to transition completely into Ethereum 2.0. There are two options under discussion currently to achieve that: **stateless clients** and **merging**.

Stateless clients

This is a proposal where Ethereum 1.0 clients will be transitioned into stateless clients. These rely on proofs from peers rather than stored copies of the chain state to verify blocks. The core idea behind this proposal is to trim the existing chain into digital receipts that prove the authenticity of the PoW transactions. This option appears to be gaining more traction.



More details and discussion on this proposal are available at
<https://ethresear.ch/t/alternative-proposal-for-early-eth1-eth2-merge/6666>.

Merging

This is a proposal where the Ethereum 1.0 chain will be merged into the Ethereum 2.0 beacon chain "as is". The idea behind this proposal is to build interoperability bridges between ETH 1 and ETH 2 until Ethereum 2.0 is fully developed.

A major feature of any blockchain is that it can run transactions and smart contracts. This feature will come with Phase 2 of Ethereum 2.0, which is responsible for state execution.

Phase 2

Phase 2 is mainly concerned with state execution. Phase 2 will have the ability to run smart contracts and transactions on shard chains. In addition, constructs such as accounts will also be available with this phase. Each shard chain in Phase 2 will have its own ewasm EVM, which will allow for faster running of smart contracts.

There is also a concept of the **execution environment**, which allows shards in Phase 2 to build custom environments for specific use cases. This gives tremendous flexibility to developers, as it allows them to build fully customized execution environments. It is quite early to say how exactly these execution environments will be designed because of the rapidly evolving research in this space.



Readers can keep an eye on the discussion at the following URL in order to keep up to date with the latest research on Ethereum 2.0 Phase 2:
<https://ethresear.ch/c/eth2-phase-2>.

Phase 2 of Ethereum 2.0 will also focus on further scalability and continuous improvement of the protocol. There are many ideas, including **zk-STARKS**,

ZK-Rollups, and **commit chains**.

We will discuss scalability solutions in detail in *Chapter 21, Scalability and Other Challenges*.

Phase 3

After Phase 2, further improvement and updates will continue. We can call this Phase 3.

Next, let's have a look at the specific high-level architecture of Ethereum 2.0.

Architecture

The following diagram shows the high-level architecture of Ethereum 2.0 clients and how they work together with the beacon chain. After completing the *Running Ethereum 2.0 clients* exercise in the book's online content repository, we'll be able to see the architecture shown in the following diagram in reality:

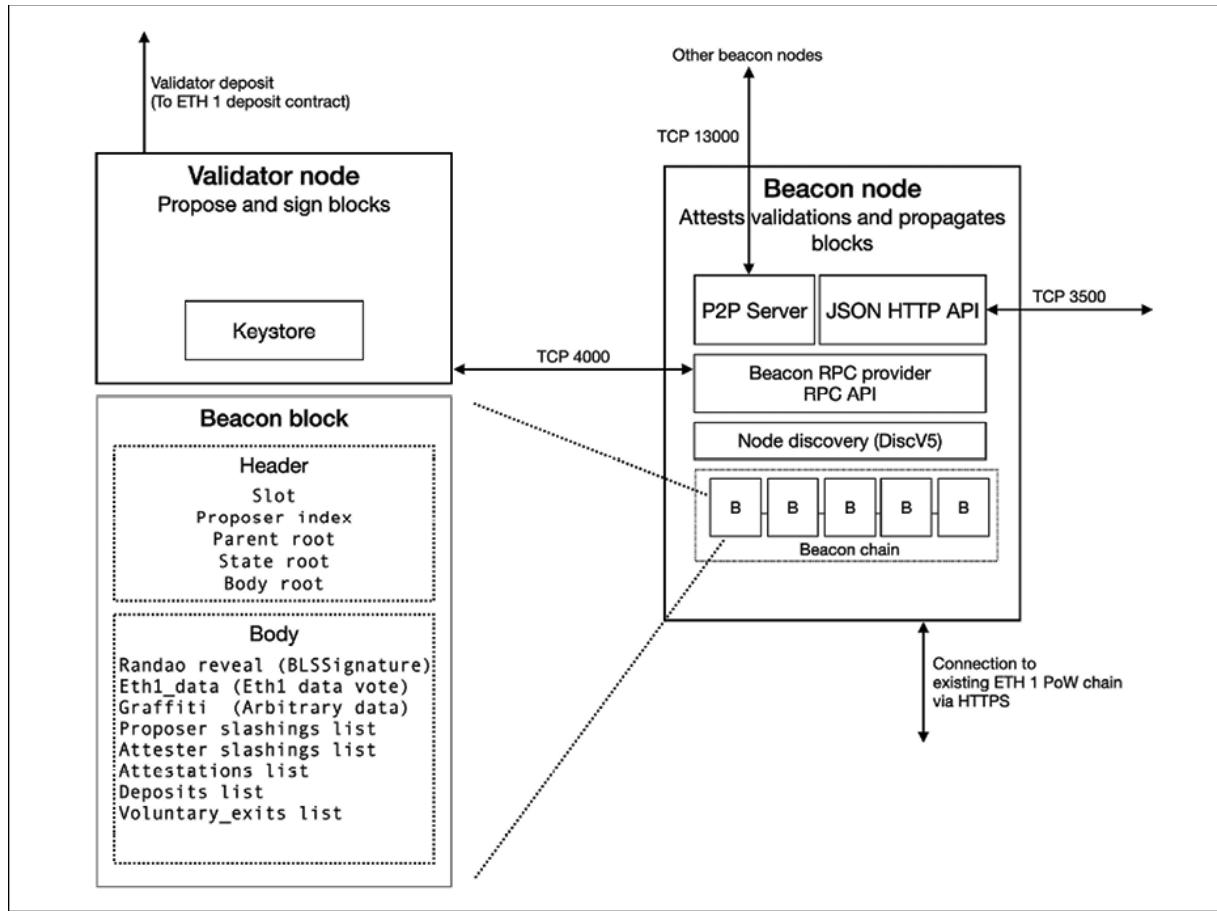


Figure 16.3: Ethereum 2.0 high-level architecture (based on Prysmatic clients and spec for Phase 0)

The preceding diagram shows a validator node and a beacon node with their different services running on different ports. The structure of a beacon block is also shown on the left-hand side, including the fields of the block header and body as per the Phase 0 specifications.

Now that we have a good understanding of Ethereum 2.0's design goals, its phases, and architecture, you can put your new knowledge into practice. Doing so is out of the scope of this chapter, but by accessing this book's bonus content pages here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf, you can follow a start-to-finish demonstration of how an Ethereum 2.0 client can be installed and run on the Onyx testnet!

The final and official testnet for ETH 2 has been released relevant details of which are available at the following link:



<https://github.com/goerli/medalla/blob/master/medalla/README.md>

In order to become a validator on Medalla testnet, follow the instructions at <https://medalla.launchpad.ethereum.org>

Summary

In this chapter, we looked at Ethereum 2.0. We explored various aspects of its architecture and the overall vision behind the development of Ethereum 2.0.

Currently, Ethereum 2.0 is under heavy development and only Phase 0, the beacon chain, is expected to launch in 2020. As the design of Phase 0 is finalized, nothing much is expected to change with the production release of the beacon chain. The techniques and examples used in this chapter can be used for the production chain too, once it's available.

We have also demonstrated in this book's bonus content pages, which we strongly encourage the reader to make use of, how to run Ethereum 2.0 beacon and validator clients. Remember that as the development of Ethereum 2.0 is rapidly evolving, some of the steps mentioned may change slightly, or the testnet that we used may be retracted. However, the steps are essentially the same; you may just have to slightly tweak the commands and run it on a different testnet. Readers are encouraged to keep an eye on <https://prylabs.net> for further developments.

In the next chapter, we'll introduce Hyperledger, which is a very active and popular blockchain project upon which multiple blockchain projects are being built.

Hyperledger

Hyperledger is not a blockchain, but a project that was initiated by the Linux Foundation in December 2015 to advance blockchain technology. This project is a collaborative effort by its members to build an open source distributed ledger framework that can be used to develop and implement cross-industry blockchain applications and systems. The principal focus is to create and run platforms that support global business transactions. The project also focuses on improving the reliability and performance of blockchain systems.

This chapter will, for the most part, discuss Hyperledger projects, and their various features and components. As Hyperledger is so vast it's not possible to cover all projects in detail in this short chapter. As such, we will introduce a variety of the projects available, and then provide detailed discussions on two mainstream projects, Fabric and Sawtooth. Projects under the Hyperledger umbrella undergo multiple stages of development, going from proposal, to incubation, and eventually graduating to an active state. Projects can also be deprecated or put in an end-of-life state where they are no longer actively developed. For a project to be able to move into the incubation stage, it must have a fully working code base along with an active community of developers. The Hyperledger project currently has more than 300 member organizations and is very active with many contributors, and regular meet-ups and talks organized around the globe.

We will first cover some of the various projects under Hyperledger. Then we will move on to examine the design, architecture, and implementation of two mainstream projects, Fabric and Sawtooth, in more detail, covering the following topics on the way:

- Projects under Hyperledger
- Hyperledger reference architecture
- Hyperledger Fabric
- Hyperledger Sawtooth
- Setting up a Sawtooth development environment

Projects under Hyperledger fall into different categories, and we'll start with the introduction of these categories in the next section.

Projects under Hyperledger

There are four categories of projects under Hyperledger. Under each category, there are multiple projects. The categories are:

- Distributed ledgers
- Libraries
- Tools
- Domain-specific

Currently, there are six distributed ledger projects under the Hyperledger umbrella: **Fabric**, **Sawtooth**, **Iroha**, **Indy**, **Besu**, and **Burrow**. Under libraries, there are the **Aries**, **Transact**, **Quilt**, and **Ursa** projects. The tools category of Hyperledger includes projects such as **Avalon**, **Cello**, **Caliper**, and **Explorer**. There are also domain-specific projects such as Hyperledger **Grid** and Hyperledger **Labs**.

The following diagram visualizes the categorization of Hyperledger projects:

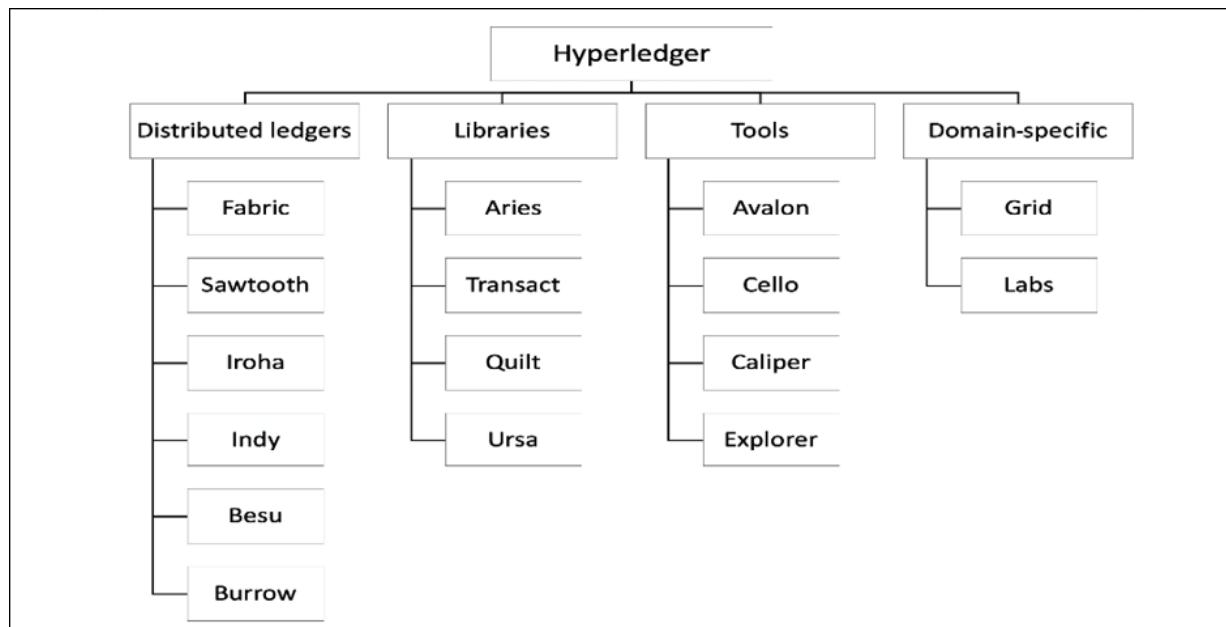


Figure 17.1: Projects under Hyperledger

A brief introduction to all of these projects follows.

Distributed ledgers

Distributed ledgers, as we covered in *Chapter 1, Blockchain 101*, are a broad category of distributed databases that are decentralized, shared among multiple participants, and updateable only via consensus.

Often, the terms blockchain and distributed ledger are used interchangeably. However, one key difference between distributed ledgers and blockchains is that blockchains are expected to have an architecture in which transactions are handled in batches of **blocks**, whereas distributed ledgers have no such requirements. All other attributes of distributed ledgers and blockchains are broadly the same. In Hyperledger, distributed ledger is a generic term used to denote both blockchains and distributed ledgers.

Fabric

Hyperledger Fabric is a blockchain project that was proposed by **IBM** and **Digital Asset Holdings (DAH)**. It is an enterprise-grade permissioned distributed ledger framework, which provides a framework for the development of blockchain solutions and applications. Fabric is based on a modular and pluggable architecture. This means that various components, such as the consensus engine and membership services, can be plugged into the system as required. Currently, its status is **active** and it is the first project to graduate from **incubation** to **active** state.



The source code is available at <https://github.com/hyperledger/fabric>.

Sawtooth

Hyperledger Sawtooth is a blockchain project proposed by Intel in April 2016. It was donated to the Linux foundation in 2016. Sawtooth introduced some novel innovations focusing on the decoupling of ledgers from transactions, flexible usage across multiple business areas using transaction families, and pluggable consensus. It is an enterprise-grade blockchain with a focus on privacy, security, and scalability.

Ledger decoupling can be explained more precisely by saying that the transactions are decoupled from the consensus layer by making use of a new concept called **transaction families**. Instead of transactions being individually coupled with the ledger, transaction families are used, which allows more flexibility, rich semantics, and the open design of business logic. Transactions follow the patterns and structures defined in the transaction families.

One of the innovative features that Intel has introduced with Hyperledger Sawtooth is a novel consensus algorithm called **Proof of Elapsed Time (PoET)**. It makes use of the **Trusted Execution Environment (TEE)** provided by **Intel Software Guard**

Extensions (Intel SGX) to provide a safe and random leader election process. It supports both permissioned and permissionless setups.



This project is available at <https://github.com/hyperledger/sawtooth-core>.

Iroha

Iroha was contributed by Soramitsu, Hitachi, NTT Data, and Colu in September 2016. Iroha aims to build a library of reusable components that users can choose to run on their own Hyperledger-based distributed ledgers.

Iroha's primary goal is to complement other Hyperledger projects by providing reusable components written in C++ with an emphasis on mobile development. This project has also proposed a novel consensus algorithm called **Sumeragi**, which is a chain-based Byzantine fault-tolerant consensus algorithm.



Iroha is available at <https://github.com/hyperledger/iroha>.

Various libraries have been proposed and are being worked on by Iroha, including, but not limited to, a digital signature library (ed25519), a SHA-3 hashing library, a transaction serialization library, a P2P library, an API server library, an iOS library, an Android library, and a JavaScript library.

Indy

This project is under incubation under Hyperledger. Indy is a distributed ledger developed for building decentralized identities. It provides tools, utility libraries, and modules, which can be used to build blockchain-based digital identities. These identities can be used across multiple blockchains, domains, and applications. Indy has its own distributed ledger and uses **Redundant Byzantine Fault Tolerance (RBFT)** for consensus.



The source code is available at <https://github.com/hyperledger/indy-node>.

Besu

Besu is a Java-based Ethereum client. It is the first project under Hyperledger that can operate on a public Ethereum network.



The source code of Besu is available at <https://github.com/hyperledger/besu>.

Burrow

This project is currently in the incubation state. Hyperledger Burrow was contributed by Monax, who develop blockchain development and deployment platforms for business. Hyperledger Burrow introduces a modular blockchain platform and an **Ethereum Virtual Machine (EVM)**-based smart contract execution environment. Burrow uses **Proof of Stake (PoS)** and a **Byzantine fault-tolerant (BFT)** Tendermint consensus mechanism. As a result, Burrow provides high throughput and transaction finality.



The source code is available at <https://github.com/hyperledger/burrow>.

With this, we have completed an introduction to distributed ledger projects under Hyperledger. Let's now look at the next category, libraries.

Libraries

A number of libraries are currently available under the Hyperledger project. This category includes projects that aim to support the blockchain ecosystem by introducing platforms for interoperability, identity, cryptography, and developer tools. We will now briefly describe each one of these.

Aries

Aries is not a blockchain; in fact, it is an infrastructure for blockchain-rooted, **peer-to-peer (P2P)** interactions. The aim of this project is to provide code for P2P interactions, secrets management, verifiable information exchange (such as verifiable credentials), interoperability, and secure messaging for decentralized systems. The eventual goal of this project is to provide a dynamic set of capabilities to store and exchange data related to blockchain-based identity.



The code for Aries is available at <https://github.com/hyperledger/aries>.

Transact

Transact provides a shared library that handles smart contract execution. This library makes the development of distributed ledger software easier by allowing the handling of scheduling, transaction dispatch, and state management via a shared software library. It provides an approach to implement new smart contract development languages named **smart contract engines**. Smart contract engines implement virtual machines or interpreters for smart contract code. Two main examples of such engines are **Seth** (for EVM smart contracts) and **Sabre** (for web assembly Wasm-based smart contracts).



Transact is available at <https://crates.io/crates/transact>.

Quilt

This utility implements the **Interledger** protocol, which facilitates interoperability across different distributed and non-distributed ledger networks.



Quilt is available at <https://github.com/hyperledger/quilt>.

Ursa

Ursa is a shared cryptography library that can be used by any project, Hyperledger or otherwise, to provide cryptography functionality. Ursa provides a C-callable library interface and a Rust crate (library). There are two sub-libraries available in Ursa: **LibUrsa** and **LibZmix**. LibUrsa is designed for cryptographic primitives such as digital signatures, standard encryptions schemes, and key exchange schemes. LibZmix provides a generic method to produce zero-knowledge proofs. It supports signature **Proofs of Knowledge (PoK)**, bullet proofs, range proofs, and set memberships.



Ursa is available at <https://github.com/hyperledger/ursa>.

We've now covered four library projects currently available under Hyperledger. In the next section, we'll explore different tools that can be used in Hyperledger projects.

Tools

There are a number of projects under the tools category in Hyperledger. Tools under Hyperledger focus on providing utilities and software tools that help to enhance the user experience. For example, visualization tools such as blockchain explorers, deployment tools, and benchmarking tools fall into this category. We'll describe some of these briefly.

Avalon

Avalon provides a trusted computer framework that enables privacy and allows off-chain processing of compute-intensive operations in a trusted and secure manner. It provides a guarantee that a computation was performed correctly and privately.

It also allows the improvement of scalability and performance and supports hardware-attested oracles. It is based on the **Trusted Compute Specifications** published by the Enterprise Ethereum Alliance and is implemented as a separate project.



Avalon is available at <https://github.com/hyperledger/avalon>.

Cello

The aim behind Cello is to allow the easy deployment of blockchains. This will provide an ability to allow **as a service** deployments of a blockchain service. Currently, this project is in the incubation stage.



The source code for Cello is available at <https://github.com/hyperledger/cello>.

Caliper

Caliper is a benchmarking framework for blockchains. It can be used to measure the performance of any blockchain. There are different performance indicators supported in the framework. These include **success rate**, **transaction read rate**, **throughput**, **latency**, and **hardware resource consumption**, such as CPU, memory, and I/O. Ethereum, Fabric, Sawtooth, Burrow, and Iroha are currently the supported blockchains in Caliper.



Caliper is available at <https://github.com/hyperledger/caliper>.

Explorer

This project aims to build a blockchain explorer for Hyperledger Fabric that can be used to view and query the transactions, blocks, and associated data from the blockchain. It also provides network information and the ability to interact with chain code.



The source code for Explorer is available at
<https://github.com/hyperledger/blockchain-explorer>.

With this, we complete the introduction to the Hyperledger project's category of tools. Next, we'll introduce domain-specific projects.

Domain-specific

Under Hyperledger, there are also some domain-specific projects that are created to address specific requirements. Currently, there are two projects under this category, which we will introduce now.

Grid

Grid is a Hyperledger project that provides a standard reference implementation of supply chain-related data types, data models, and relevant business logic encompassing smart contracts. It is currently in the incubation stage.



The code for Grid is available at <https://github.com/hyperledger/grid>.

Labs

Hyperledger labs provide a quick and easy way to start a project under Hyperledger without going through the official formal process of incubation and activation. This allows the development of research projects, hackathons, and demo projects.



Further details on this project are available at <https://github.com/hyperledger-labs>.

Each of the aforementioned projects is in various stages of development. This list is expected to grow as more and more members join the Hyperledger project and

contribute to the development of blockchain technology. Now, in the next section, we will examine the reference architecture of Hyperledger, which provides general principles and design philosophies that can be followed to build new Hyperledger projects.

Hyperledger aims to build new blockchain platforms that are driven by industry use cases. As there have been many contributions made to the Hyperledger project by the community, the Hyperledger blockchain platform is evolving into a protocol for business transactions. Hyperledger is also evolving into a specification that can be used as a reference to build blockchain platforms as compared to earlier blockchain solutions that address only a specific type of industry or requirement.

In the following section, a reference architecture model is presented that has been published by the Hyperledger project. This architecture can be used by a blockchain developer to build a blockchain that is in line with the specifications of the Hyperledger architecture.

Hyperledger reference architecture

Hyperledger has published a white paper that presents a reference architecture model that can serve as a guideline to build permissioned distributed ledgers. The reference architecture consists of various components that form a business blockchain.



Hyperledger's white paper is available here:

https://docs.google.com/document/d/1Z4M_qwILLRehPbVRUsJ3OF8Iir-gqs-ZYe7W-LE9gnE/edit#heading=h.m6iml6hqrnm2

These high-level components are shown in the reference architecture diagram here, which has been drawn from the aforementioned white paper:

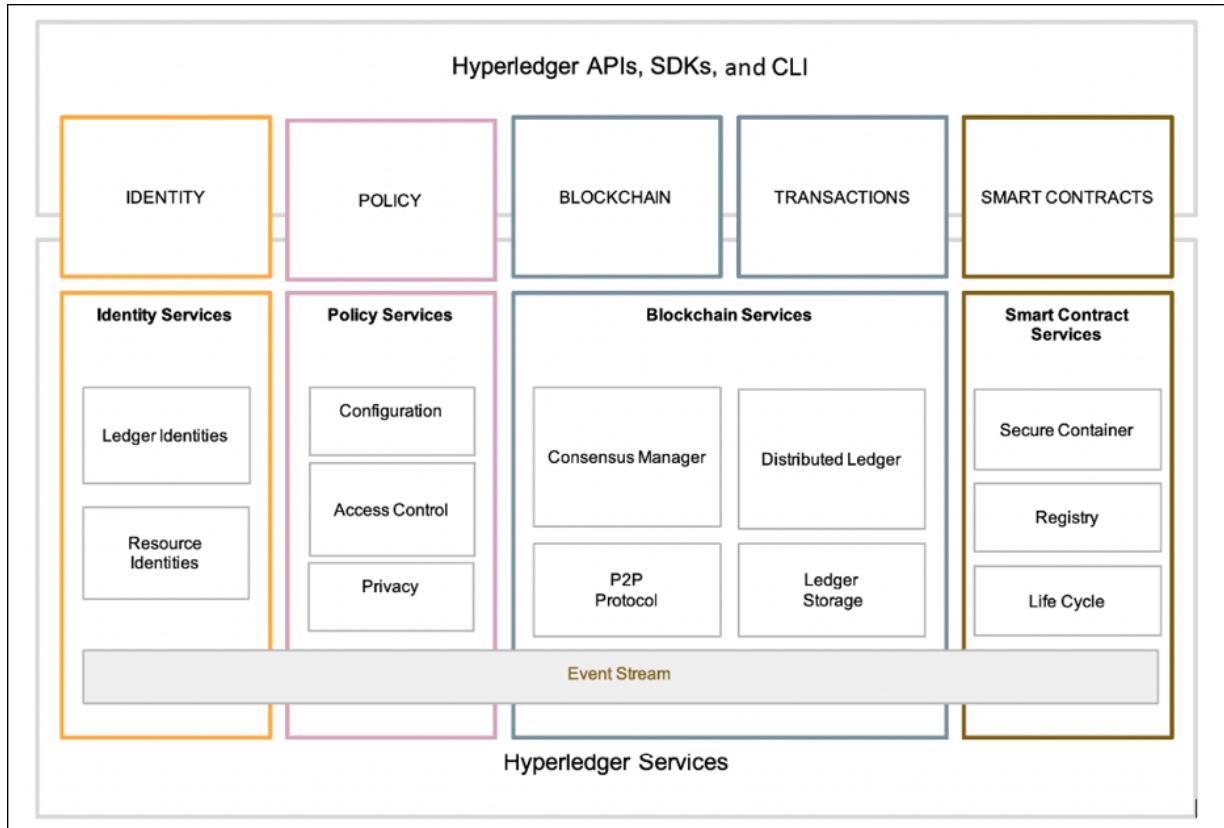


Figure 17.2: Reference architecture

In the preceding diagram, starting from the left, we see that we have five top-level components that provide various services. The first is **identity**, which provides authorization, identification, and authentication services under membership services. Then, we have the **policy** component, which provides policy services.

After this, we see **blockchain** and **transactions**, which consist of the **Consensus Manager**, **Distributed Ledger**, the network **P2P Protocol**, and **Ledger Storage** services. The consensus manager ensures that the ledger is updateable only through consensus among the participants of the blockchain network.

Finally, we have the **smart contracts** layer, which provides chaincode services in Hyperledger and makes use of **Secure Container** technology to host smart contracts. Chaincode can be considered as the Hyperledger equivalent of smart contracts.



Chaincode is a program that implements a specific interface. It is usually written in Java, Node.js, or Go. Even though the terms *smart contract* and *chaincode* are used interchangeably in Hyperledger Fabric, there is a subtle difference between chaincode and smart contract. A smart contract can be defined as a piece of code that defines the transaction logic, which controls the transaction life cycle and can result in updating the world state. Chaincode is a relevant but slightly different concept—chaincode is a deployable object that contains smart contracts packaged within it. A single chaincode can contain multiple smart

contracts and after deployment, all smart contracts contained within the chaincode are available for use. Generally, however, the terms are used interchangeably.

We will see all these in more detail in the *Hyperledger Fabric* section shortly.

From a components perspective, Hyperledger contains various elements, as described here:

- **Consensus:** These services are responsible for facilitating the agreement process between the participants on the blockchain network. Consensus is required to make sure that the order and state of transactions are validated and agreed upon in the blockchain network.
- **Smart contracts:** These services are responsible for implementing business logic as per the requirements of the users. Transactions are processed based on the logic defined in the smart contracts that reside on the blockchain.
- **Communication:** This component is responsible for message transmission and exchange between the nodes on the blockchain network.
- **Security and crypto:** These services are responsible for providing the capability to allow various cryptographic algorithms or modules to provide privacy, confidentiality, and non-repudiation services.
- **Data store:** This component provides an ability to use different data stores for storing the state of the ledger. This means that data stores are also pluggable, allowing the usage of any database backend, such as `couchdb` or `goleveldb`.
- **Policy services:** This set of services provides the ability to manage the different policies required for the blockchain network. This includes endorsement policy and consensus policy.
- **APIs and SDKs:** This component allows clients and applications to interact with the blockchain. An SDK is used to provide mechanisms to deploy and execute chaincode, query blocks, and monitor events on the blockchain.

There are certain requirements of a blockchain service. In the next section, as an example, we are going to discuss the design goals of Hyperledger Fabric.

Hyperledger design principles

There are certain requirements of a blockchain service. The reference architecture is driven by the needs and requirements raised by the participants of the Hyperledger project after studying the industry use cases. There are several categories of requirements that have been deduced from the study of industrial use cases and are seen

as principles of design philosophy. We'll describe these principles in the following sections.

Modular structure

The main requirement of Hyperledger is a modular structure. It is expected that a cross-industry blockchain will be used in many business scenarios, and as such, functions related to storage, policy, chaincode, access control, consensus, and many other blockchain services should be modular and pluggable. The specification provided in the Hyperledger reference architecture suggests that the modules should be "plug and play" and users should be able to easily remove and add a different module that meets the requirements of the business.

Privacy and confidentiality

This is one of the most critical factors. As traditional blockchains are permissionless, in a permissioned model it is of the utmost importance that transactions on the network are visible to only those who are allowed to view it. The privacy and confidentiality of transactions and contracts are of absolute importance in a business blockchain. As such, Hyperledger's vision is to provide support for a full range of cryptographic protocols and algorithms. We discussed cryptography in *Chapter 3, Symmetric Cryptography* and *Chapter 4, Public Key Cryptography*.

It is expected that users will be able to choose appropriate modules according to their business requirements. For example, if a business blockchain needs to be run only between already-trusted parties and performs very basic business operations, then perhaps there is no need to have advanced cryptographic support for confidentiality and privacy. Therefore, users should be able to remove that functionality (module) or replace it with a more appropriate module that suits their needs.

Similarly, if users need to run a cross-industry blockchain, then confidentiality and privacy can be of paramount importance. In this case, users should be able to plug an advanced cryptographic and access control mechanism (module) into the blockchain, which can even allow the usage of **hardware of security modules (HSMs)**.

The blockchain should also be able to handle sophisticated cryptographic algorithms without compromising performance. In addition to the previously mentioned scenarios, due to regulatory requirements in business, there should also be a provision to allow the implementation of privacy and confidentiality policies in conformance with regulatory and compliance requirements.

Identity

In order to provide privacy and confidentiality services, a flexible **Public Key Infrastructure (PKI)** model that can be used to handle the access control functionality is also required. The strength and type of cryptographic mechanisms are also expected to vary according to the needs and requirements of the users. In certain scenarios, it might be required for a user to hide their identity, and as such, Hyperledger is expected to provide this functionality.

Scalability

This is another major requirement that, once met, will allow reasonable transaction throughput, which will be sufficient for all business requirements and also a large number of users.

Deterministic transactions

This is a core requirement in any blockchain. If transactions do not produce the same result every time they are executed, then regardless of who and where the transaction is executed, achieving consensus is impossible. Therefore, deterministic transactions become a key requirement in any blockchain network. We discussed these concepts in *Chapter 10, Smart Contracts*.

Auditability

Auditability is another requirement of business hyperledgers. It is expected that an immutable audit trail of all identities, related operations, and any changes is kept.

Interoperability

Currently, there are many blockchain platforms available, but they cannot communicate with each other easily. This can be a limiting factor in the growth of a blockchain-based global business ecosystem. It is envisaged that many blockchain networks will operate in the business world for specific needs, but it is important that they are able to communicate with each other. There should be a common set of standards that all blockchains can follow in order to allow communication between different ledgers. There are different efforts already underway to achieve this, not only under the Hyperledger umbrella, such as Hyperledger Quilt, but also other projects such as Cosmos and Polkadot.

Portability

The portability requirement is concerned with the ability to run across multiple platforms and environments without the need to change anything at the code level.

Hyperledger Fabric, for example, is envisaged to be portable, not only at the infrastructure level, but also at the code, library, and API levels, so that it can support uniform development across various implementations of Hyperledger.

Rich data queries

The blockchain network should allow rich queries to be run on the network. This can be used to query the current state of the ledger using traditional query languages, which will allow wider adoption and ease of use.

All of the aforementioned points describe the general guiding principles that allow the development of blockchain solutions that are in line with the Hyperledger design philosophy.

In the next section, we will look at Hyperledger Fabric in detail, which is the first project to graduate to the active status under Hyperledger.

Hyperledger Fabric

Hyperledger Fabric, or **Fabric** for short, is the contribution made initially by IBM and Digital Assets to the Hyperledger project. This contribution aims to enable a modular, open, and flexible approach toward building blockchain networks.

Various functions in the fabric are pluggable, and it also allows the use of any language to develop smart contracts. This functionality is possible because it is based on container technology (Docker), which can host any language.

Chaincode is sandboxed in a secure container, which includes a secure operating system, the chaincode language, runtime environment, and SDKs for Go, Java, and Node.js. Other languages can be supported too in the future, if required, but this needs some development work. This ability is a compelling feature compared to domain-specific languages in Ethereum, or the limited scripted language in Bitcoin. It is a permissioned network that aims to address issues such as scalability, privacy, and confidentiality. The fundamental idea behind this is modularization, which would allow flexibility in the design and implementation of the business blockchain. This can then result in achieving scalability, privacy, and other desired attributes and fine-tuning them according to requirements.

Transactions in Fabric are private, confidential, and anonymous for general users, but they can still be traced back and linked to the users by authorized auditors. As a permissioned network, all participants are required to be registered with the membership

services to access the blockchain network. This ledger also provided auditability functionality to meet the regulatory and compliance needs required by the user.

Membership services

These services are used to provide access control capability for the users of the Fabric network. Membership services perform the following functions:

- User identity verification
- User registration
- Assign appropriate permissions to the users depending on their roles

Membership services make use of a **certificate authority (CA)** in order to support identity management and authorization operations. This CA can be internal (such as Fabric CA, which is a default interface in Hyperledger Fabric), or an organization can opt to use an external certificate authority. Fabric CA issues **enrolment certificates (E-Certs)**, which are produced by an **enrolment certificate authority (E-CA)**. Once peers are issued with an identity, they are allowed to join the blockchain network. There are also temporary certificates issued called **T-Certs**, which are used for one-time transactions.

All peers and applications are identified using a certificate authority. An authentication service is provided by the certificate authority. **Membership Service Providers (MSPs)** can also interface with existing identity services like LDAP.

This section covered the membership services implemented in Hyperledger Fabric. Next, we'll introduce some of Fabric's blockchain services.

Blockchain services

Blockchain services are at the core of Hyperledger Fabric. Components within this category are as follows.

Consensus services

A consensus service is responsible for providing the interface to the consensus mechanism. This serves as a module that is pluggable and receives the transaction from other Hyperledger entities and executes them under criteria according to the type of mechanism chosen.

Consensus in Hyperledger Fabric is implemented as a peer called **orderer**, which is responsible for ordering the transactions in sequence into a block. An orderer does not hold smart contracts or ledgers. Consensus is pluggable and currently, there are two basic types of ordering services available in Hyperledger Fabric:

- **SOLO**: This is a basic ordering service intended to be used for development and testing purposes.
- **Kafka**: This is an implementation of Apache Kafka, which provides an ordering service. It should be noted that currently, Kafka only provides crash fault-tolerance but does not provide Byzantine fault-tolerance. This is acceptable in a permissioned network where the chances of malicious actors are almost none.

In addition, there are also ordering services available in Hyperledger Fabric that are based on more familiar consensus mechanisms such as Raft and PBFT

- **Raft**: Raft is a crash fault-tolerant and leader-follower-based protocol for achieving distributed consensus.
- **PBFT**: PBFT is a state machine replication-based BFT protocol that can be used in Hyperledger Fabric to achieve consensus.



We discussed consensus mechanisms in appropriate detail in *Chapter 5, Consensus Algorithms*. Readers can refer to this chapter to review the Raft and PBFT mechanisms.

In addition to these mechanisms, other mechanisms may become available in the future that can be plugged into Hyperledger Fabric.

Distributed ledger

Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a cryptographically linked list of blocks (as introduced in *Chapter 1, Blockchain 101*) and world state is a key-value database. This database is used by smart contracts to store relevant states during execution by the transactions. A blockchain consists of blocks that contain transactions. These transactions contain chaincode, which runs transactions that can result in updating the world state. Each node saves the world state on disk in LevelDB or CouchDB depending on the implementation. As Fabric allows pluggable data stores, you can choose any data store for storage.

A block consists of three main components called **block header**, **block data (transactions)**, and **block metadata**.

The following diagram shows a blockchain depiction with the block and transaction structure in Hyperledger Fabric with the relevant fields:

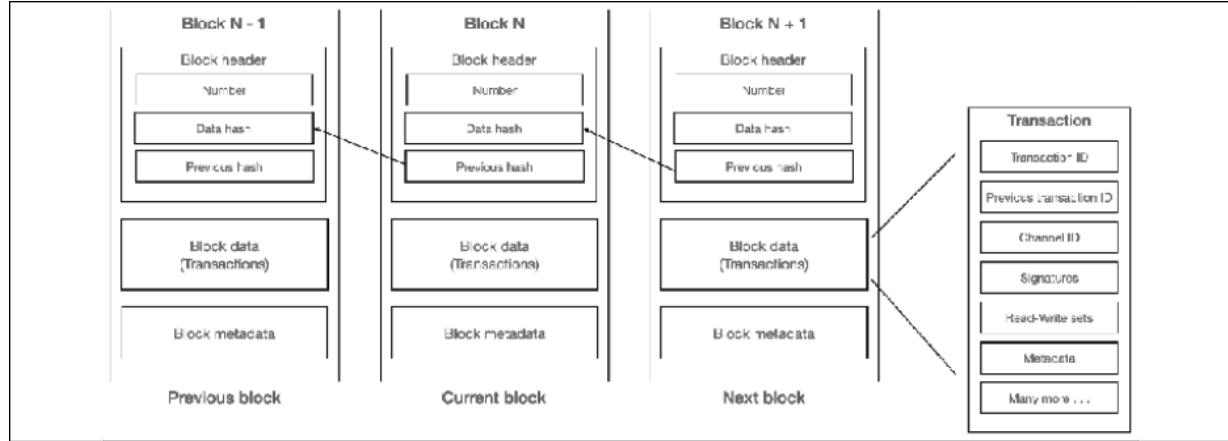


Figure 17.3: Blockchain and transaction structure

Block header consists of three fields, namely **Number**, **Data hash**, and **Previous hash**.

Block data contains an ordered list of transactions. A **Transaction** is made up of multiple fields such as **Transaction ID**, **Previous transaction ID**, **Channel ID**, **Signatures**, **Read-Write sets**, **Metadata**, and many more.

Block metadata mainly consists of the creator identity, time of creation, and relevant signatures.

The peer-to-peer protocol

The P2P protocol in Hyperledger Fabric is built using **Google RPC (gRPC)**. It uses protocol buffers to define the structure of the messages.

Messages are passed between nodes in order to perform various functions. There are four main types of messages in Hyperledger Fabric: **discovery**, **transaction**, **synchronization**, and **consensus**. Discovery messages are exchanged between nodes when starting up in order to discover other peers on the network. Transaction messages are used to deploy, invoke, and query transactions, and consensus messages are exchanged during consensus. Synchronization messages are passed between nodes to synchronize and keep the blockchain updated on all nodes.

Ledger storage

In order to save the state of the ledger, by default, LevelDB is used, which is available at each peer. An alternative is to use CouchDB, which provides the ability to run rich

queries.

Now that we've established some of the core blockchain services used in Hyperledger Fabric, let's look at some of the chaincode, or smart contract, services available, along with an overview of the APIs and CLIs available in Fabric.

Smart contract services

These services allow the creation of secure containers that are used to execute the chaincode. Components in this category are as follows:

- **Secure container:** Chaincode is deployed in Docker containers that provide a locked-down sandboxed environment for smart contract execution. Currently, Golang is supported as the main smart contract language, but any other mainstream languages can be added and enabled if required.
- **Secure registry:** This provides a record of all images containing smart contracts.
- **Events:** Events on the blockchain can be triggered by endorsers and smart contracts. External applications can listen to these events and react to them if required via event adapters. They are similar to the concept of events introduced in Solidity in *Chapter 15, Introducing Web3*.

APIs and CLIs

An application programming interface provides an interface into Fabric by exposing various REST APIs. Additionally, command-line interfaces that provide a subset of REST APIs and allow quick testing and limited interaction with the blockchain are also available.

In the preceding sections, we have covered the main elements of Hyperledger Fabric's reference architecture. Next, let's consider a variety of Hyperledger Fabric's components from a network perspective.

Components

There are various components that can be part of the Hyperledger Fabric blockchain. These components include, but are not limited to, the peers, clients, channels, world state database, transactions, membership service providers, smart contracts, and crypto service provider components.

Peers

Peers participate in maintaining the state of the distributed ledger. They also hold a local copy of the distributed ledger. Peers communicate via gossip protocol. There are three types of peers in the Hyperledger Fabric network:

- **Endorsing peers or endorsers**, which simulate the transaction execution and generate a read-write set. **Read** is a simulation of a transaction's reading of data from the ledger and **write** is the set of updates that would be made to the ledger if and when the transaction is executed and committed to the ledger. Endorsers execute and endorse transactions. It should be noted that an endorser is also a committer too. Endorsement policies are implemented with chaincode and specify the rules for transaction endorsement.
- **Committing peers or committers**, which receive transactions endorsed by endorsers, verify them, and then update the ledger with the read-write set. A committer verifies the read-write set generated by the endorsers along with transaction validation.
- **Orderer nodes** receive transactions from endorsers along with read-write sets, arrange them in a sequence, and send them to committing peers. Committing peers then perform validation and committing to the ledger.

All peers make use of certificates issued by membership services.

Clients

Clients are software that make use of APIs to interact with Hyperledger Fabric and propose transactions.

Channels

Channels allow the flow of confidential transactions between different parties on the network. They allow the use of the same blockchain network but with separate overlay blockchains. Channels allow only members of the channel to view the transactions related to them; all other members of the network will not be able to view the transactions.

World state database

World state reflects all the committed transactions on the blockchain. This is essentially a key-value store that is updated as a result of transactions and chaincode execution. For this purpose, either LevelDB or CouchDB is used. LevelDB is a key-value store whereas

CouchDB stores data as JSON objects, which allows rich queries to run against the database.

Transactions

Transaction messages can be divided into two types: **deployment transactions** and **invocation transactions**. The former is used to deploy new chaincode to the ledger, and the latter is used to call functions from the smart contract. Transactions can be either public or confidential. Public transactions are open and available to all participants, while confidential transactions are visible only in a channel open to its participants.

Membership Service Provider

The **MSP** is a modular component that is used to manage identities on the blockchain network. This provider is used to authenticate clients who want to join the blockchain network. A certificate authority is used in the MSP to provide identity verification and binding services.

Smart contracts

We discussed smart contracts in detail in *Chapter 10, Smart Contracts*. In Hyperledger Fabric, the same concept of smart contracts is implemented but they are called chaincode instead of smart contracts. They contain conditions and parameters to execute transactions and update the ledger. Chaincode is usually written in Golang or Java.

Crypto service provider

As the name suggests, this is a service that provides cryptographic algorithms and standards for use in the blockchain network. This service provides key management, signature and verification operations, and encryption-decryption mechanisms. This service is used with the membership service to provide support for cryptographic operations for elements of blockchain such as endorsers, clients, and other nodes and peers.

After this introduction to some of the components of Hyperledger Fabric, we will next see what an application looks like when on a Hyperledger network.

Applications on blockchain

A typical application on Fabric is simply composed of a user interface, usually written in JavaScript/HTML, that interacts with the backend chaincode (smart contract) stored on

the ledger via an API layer:

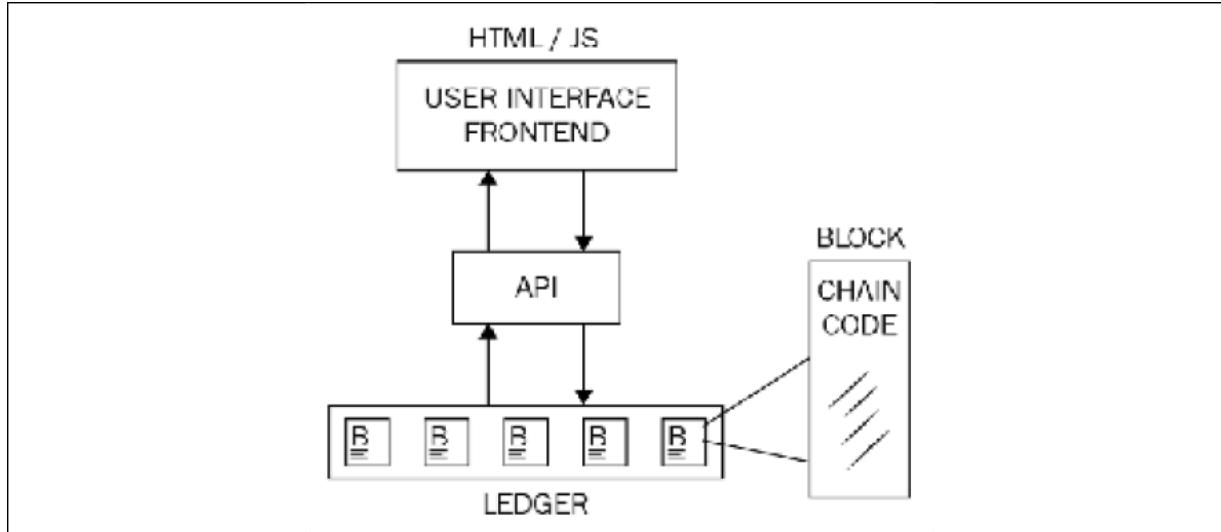


Figure 17.4: A typical Fabric application

Hyperledger provides various APIs and command-line interfaces to enable interaction with the ledger. These APIs include interfaces for identity, transactions, chaincode, ledger, network, storage, and events.

In the next section, we'll see how chaincode is implemented.

Chaincode implementation

Chaincode is usually written in Golang or Java. Chaincode can be public (visible to all on the network), confidential, or access-controlled. These code files serve as a smart contract that users can interact with via APIs. Users can call functions in the chaincode that result in a state change, and consequently update the ledger.

There are also functions that are only used to query the ledger and do not result in any state change. Chaincode implementation is performed by first creating the chaincode shim interface in the code. Shims provide APIs for accessing state variables and the transaction context of chaincode. It can either be in Java or Golang code.

The following four functions are required in order to implement the chaincode:

- `Init()` : This function is invoked when the chaincode is deployed onto the ledger. This initializes the chaincode and results in making a state change, which updates the ledger accordingly.
- `Invoke()` : This function is used when contracts are executed. It takes a function name as parameters along with an array of arguments. This function results in a

state change and writes to the ledger.

- `Query()` : This function is used to query the current state of a deployed chaincode. This function does not make any changes to the ledger.
- `Main()` : This function is executed when a peer deploys its own copy of the chaincode. The chaincode is registered with the peer using this function.

The following diagram illustrates the general overview of Hyperledger Fabric. Note that peer clusters at the top include all types of nodes: **Endorsers**, **Committers**, and **Orderers**:

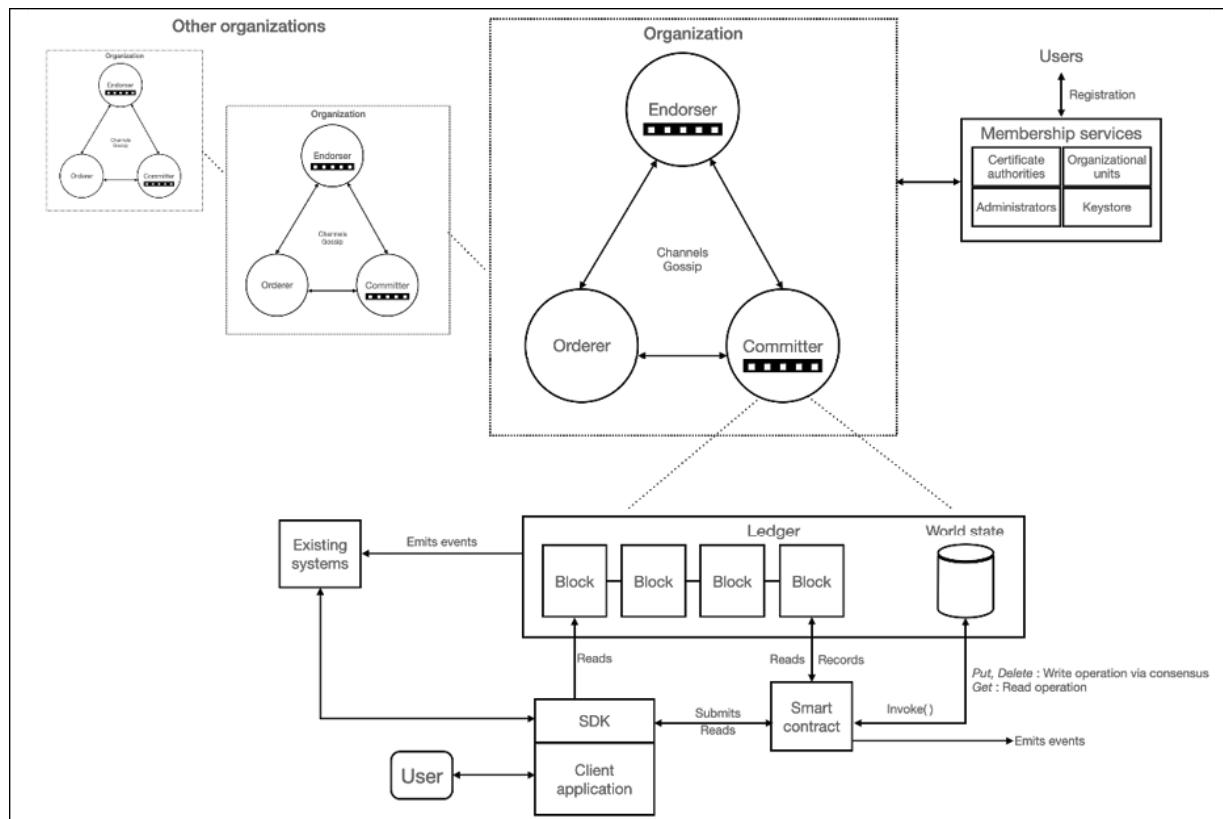


Figure 17.5: A high-level overview of a Hyperledger Fabric network

The preceding diagram shows that peers (shown center-top) communicate with each other and each node has a copy of blockchain. In the top-right corner, the membership services are shown, which validate and authenticate peers on the network by using a CA. At the bottom of the diagram, a magnified view of blockchain is shown whereby existing systems can produce events for the blockchain and can also listen for the blockchain events, which can then optionally trigger an action. At the bottom right-hand side, a user's interaction with the application is shown. The application in turn interacts with the smart contract via the `Invoke()` method, and smart contracts can query or update the state of the blockchain.

In this section, we saw how chaincode is implemented and examined the high-level architecture of Hyperledger Fabric. In the next section, we will discuss the application model.

The application model

Any blockchain application for Hyperledger Fabric follows the MVC-B architecture. This is based on the popular MVC design pattern. Components in this model are **View**, **Control**, **Model** (data model), and **Blockchain**:

- **View logic:** This is concerned with the user interface. It can be a desktop, web application, or mobile frontend.
- **Control logic:** This is the orchestrator between the user interface, data model, and APIs.
- **Data model:** This model is used to manage the off-chain data.
- **Blockchain logic:** This is used to manage the blockchain via the controller and the data model via transactions.



The IBM cloud service offers sample applications for blockchain under its blockchain as a service offering. It is available at <https://www.ibm.com/blockchain/platform/>. This service allows users to create their own blockchain networks in an easy-to-use environment.

After this brief introduction to the application model, let's move onto another important topic, consensus, which is important not only in Hyperledger Fabric, but is also central to blockchain design and architecture.

Consensus in Hyperledger Fabric

The consensus mechanism in Hyperledger Fabric consists of three steps:

- **Transaction endorsement:** This process endorses the transactions by simulating the transaction execution process.
- **Ordering:** This is a service provided by the cluster of **orderers**, which takes endorsed transactions and decides on a sequence in which the transactions will be written to the ledger.
- **Validation and commitment:** This process is executed by committing peers, which first validate the transactions received from the orderers and then commit that transaction to the ledger.

These steps are shown in the following diagram:

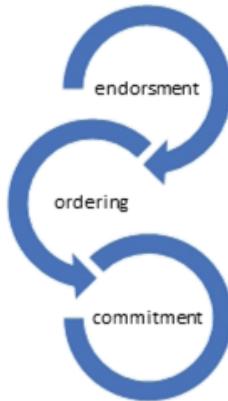


Figure 17.6: The consensus flow

The ordering step is also pluggable. There are a number of ordering services available for the Fabric consensus mechanism. Initially, in some versions of Fabric 1.0, only simple ordering services such as Apache Kafka were available. However, now, other advanced consensus algorithms such as the BFT-SMaRt, Honey Badger, and Simplified BFT implementations are also available.

In the next section, we'll describe how a transaction flows through various stages in Hyperledger Fabric before finally updating the ledger.

The transaction lifecycle in Hyperledger Fabric

There are several steps that are involved in a transaction flow in Hyperledger Fabric. These steps are visualized in the following diagram:

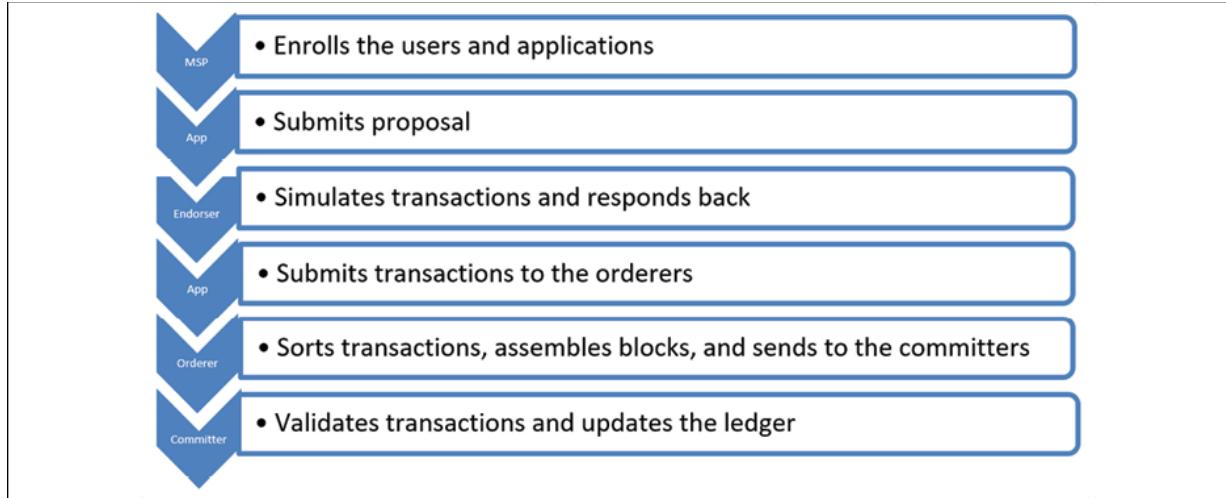


Figure 17.7: The high-level transaction lifecycle

The steps are described in detail as follows:

1. Transaction proposal by clients. This is the first step where a transaction is proposed by the clients and sent to endorsing peers on the distributed ledger network. All clients need to be enrolled via membership services before they can propose transactions.
2. The transaction is simulated by endorsers, which generates a **read-write (RW)** set. This is achieved by executing the chaincode, but instead of updating the ledger, only an RW set depicting any reads or updates to the ledger is created.
3. The endorsed transaction is sent back to the application.
4. The endorsed transactions and RW sets are submitted to the ordering service by the application.
5. The ordering service assembles all endorsed transactions and RW sets in order into a block and sorts them by channel ID.
6. The ordering service broadcasts the assembled block to all committing peers.
7. Committing peers validate the transactions.
8. Committing peers update the ledger.
9. Finally, notification of success or failure of the transaction by committing peers is sent back to the clients/applications.

The following diagram represents the aforementioned steps and the Fabric architecture from a transaction flow perspective:

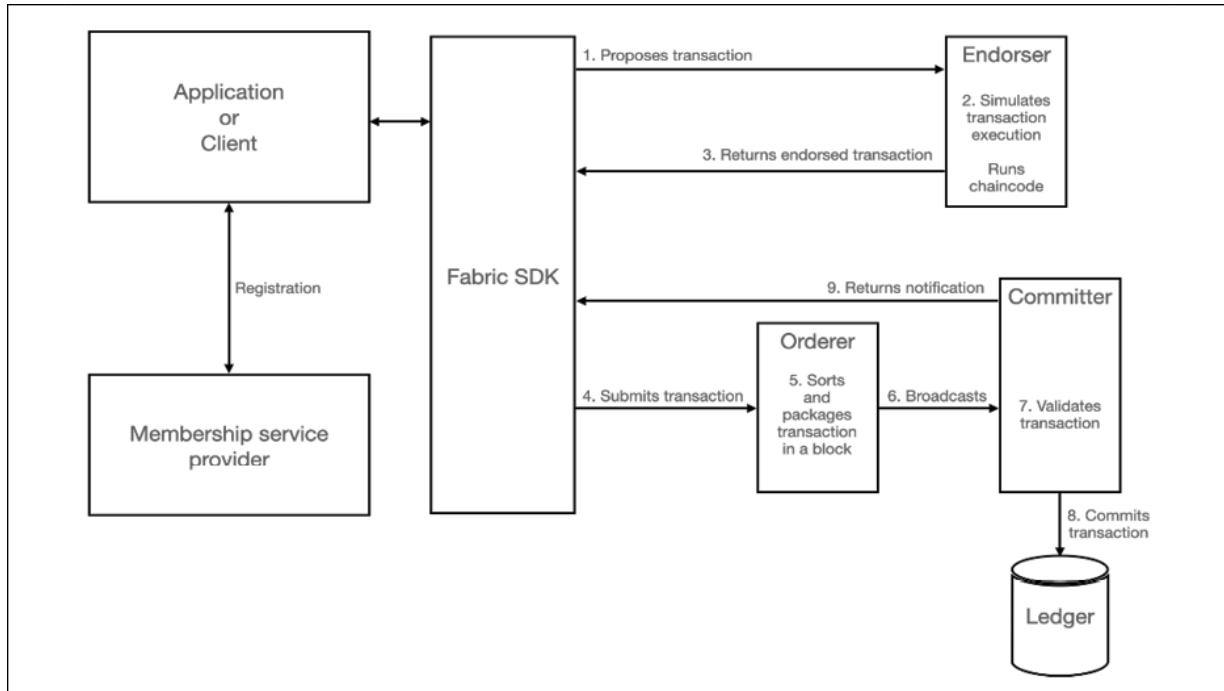


Figure 17.8: The transaction flow

As you can see in the preceding diagram, the first step is to propose transactions, which a client does via an SDK. Before this, it is assumed that all clients and peers are registered with the membership service provider.

So far, we have covered the core architecture of Hyperledger Fabric 1.0. This architecture works well but requires some improvements. In order to address these limitations, a new version of Hyperledger Fabric, Fabric 2.0, has been developed. In the next section, we will introduce the core features of Fabric 2.0.

Fabric 2.0

This is a major upgrade to the protocol. The fundamentals remain the same, but some very interesting new features and improvements have been made, which are introduced in the following sections.



Hyperledger Fabric 2.0 was released on January 30, 2010. The latest release is available at <https://github.com/hyperledger/fabric/releases/tag/v2.0.1>.

New chaincode lifecycle management

In Hyperledger Fabric 2.0, a new decentralized chaincode lifecycle management is implemented, which is the main notable difference between Fabric 1.0 and Fabric 2.0. In the former, an organization administrator can install the chaincode in its own organization and other organizations administrators have to unconditionally agree to instantiate (deploy or upgrade) the chaincode. In other words, there is no agreement process between organizations to agree on the installation of a chaincode. This problem is resolved by introducing checks, which enables peers to only participate in a chaincode if their organizational administrators have agreed to it.

Now let's look at the chaincode lifecycle in Fabric 1.0:

1. Each organization administrator installs chaincode on their peers.
2. An administrator instantiates the chaincode by deploying or upgrading.
3. After results are received, the transaction is submitted to the orderers.
4. After ordering, it is submitted to all peers and committed.

Now, we'll look at Fabric 2.0's lifecycle, which highlights the differences between it and Fabric 1.0:

1. Install chaincode by putting the package on the filesystem. Each organization administrator installs chaincode on their peers.
2. Provide approval by executing the `approveformyorg` function. This verifies that the organization agrees with the chaincode definition.
3. Send the ordering service to the orderer.
4. Commits to the organization collection. The collection is a private data collection that provides a mechanism to allow private data sharing only between those organizations that are party to the transaction, even if they are within the same channel. In other words, collections allow private data sharing between a subset of organizations on a channel. Even though other organizations are members of the same channel, they cannot see the data shared between a subset of organizations that are sharing data using collection, also called private data collection.
5. Other organizations' administrators perform the same *steps 1* to *4*.
6. After these steps are executed, there is a record in each organization indicating the agreement on the chaincode definition.
7. Define the chaincode as per the agreement in step 6.
8. Submit to ordering.
9. Commit to the definition of all peers.
10. The organization administrator who committed the chaincode definition now invokes the `init` function on its own peers and other organizations' peers.
11. Submit to ordering.

12. Commit all the peers.



Note that Fabric 2.0 supports both legacy and new lifecycle patterns, but eventually the legacy lifecycle will become deprecated.

In addition to the chaincode lifecycle management changes in Fabric 2.0, there are also new chaincode application patterns. We will introduce these next.

New chaincode application patterns

There are also new chaincode application patterns introduced in Fabric 2.0 for collaboration and consensus. These are automated checks that can be added by an organization to enable additional validation before a transaction is endorsed. In addition, decentralized agreement is also supported where multiple organizations can propose conditions for an agreement, and when those conditions are met, a collective agreement can be made on a transaction.

Enhanced data privacy

Fabric 2.0 enhances data privacy by introducing collection-level endorsement policies, per-organization collections, and the flexible sharing and verification of private data.

External chaincode launcher

In Hyperledger Fabric 2.0, by default, Docker APIs are used to build and run chaincode. In addition, external builders and launchers can be used if required. In previous versions, there was a requirement to have access to the Docker daemon to build and run chaincode, but in Fabric 2.0 there is no such dependency. There is also no requirement to run chaincode in Docker containers anymore: chaincode can be run in any environment deemed appropriate by the network operator.

Raft consensus

Raft is a **Crash Fault-tolerant (CFT)** ordering service for achieving consensus. It is based on the `etcd` library and is based on a leader-follower model. In this model, a leader is elected whose decision is then replicated by its followers.

Better performance

In Fabric 2.0, caching has been introduced at the CouchDB state database level to improve performance by reducing the read latency introduced due to expensive lookup

operations in previous versions.

With this section, our introduction to Hyperledger Fabric is complete. Hyperledger Fabric is the flagship project of Hyperledger and is used in a wide variety of applications. A list of cross-industry projects built with Hyperledger Fabric and other Hyperledger projects is being maintained here:

<https://www.hyperledger.org/learn/blockchain-showcase>. Readers can explore this further to understand how Hyperledger Fabric is used in different use cases.

Hyperledger Fabric is continually evolving and more features and changes are expected in future releases. However, no major design changes are expected since the introduction of version 2.0.

In the next section, we'll explore another popular Hyperledger project named Sawtooth.

Hyperledger Sawtooth

Sawtooth is an enterprise-grade distributed ledger that can run in both permissioned and non-permissioned modes. Sawtooth has several new features, which are introduced in the following sections.

Core features

These features include modular design, parallel transaction execution, global state agreement, dynamic consensus, and some other advanced features. We will now explore these features one by one.

Modular design

The modular design of Sawtooth enables separation between the application and the core system. This means that developers can focus on the business objectives instead of worrying about the underlying design of the system. The design of Sawtooth can be viewed as a layered architecture where transaction processors manage the application business logic and, on another layer, validators handle the verification and consensus on transactions. A separate layer called the transaction processing layer is responsible for managing transaction rules, permissions, and consensus-related settings. In summary, Sawtooth separates the core functionality of blockchain from the business logic.

Parallel transaction execution

Usually, blockchains process transactions sequentially. This serial processing is seen as a bottleneck and impacts performance, which makes some blockchains unsuitable for high-throughput use cases. Sawtooth has come up with a remarkable feature in which an advanced parallel scheduler splits the transactions into parallel flows, which results in increased performance due to the parallel execution of transactions.

Global state agreement

Sawtooth ensures that each node has an identical copy of the blockchain database and this database is also cryptographically verifiable. This means that Sawtooth is a cryptographically verifiable distributed ledger with global state agreement.

Dynamic and pluggable consensus algorithms

Sawtooth supports different types of consensus mechanism via a modular mechanism implemented as a separate process. A very interesting feature of Sawtooth is that consensus mechanisms can be changed while a blockchain is running. This means that a different consensus mechanism can be chosen for the same blockchain regardless of what was initially chosen at the network setup. This is a powerful feature that is made possible due to the isolation between the consensus mechanism and transaction semantics.

Multi-language support

Blockchain applications in Sawtooth can be written using various different languages. Sawtooth also includes SDKs for different languages, such as JavaScript, Python, and Go. This feature makes it a flexible choice for many developers.

Enhanced event mechanism

Sawtooth supports the creation and broadcasting of events where applications can subscribe to blockchain-related events or application-specific events. Also, it can relay information about the execution of a transaction to clients without the requirement to store that data in a state.

On-chain governance

Sawtooth allows the on-chain storage, management, and application of configuration settings and permissions, which makes it easier for all clients to refer to a single source of configuration.

Interoperability

Sawtooth supports integration with other blockchain technologies. For example, it introduced a project named `seth`, which allows EVM-based smart contracts to be deployed on Sawtooth.

Sawtooth has proposed some other novel features, such as the introduction of a new consensus algorithm called **PoET**, and the idea of **transaction families**. We'll discuss these concepts in more detail in the upcoming sections.

Consensus in Sawtooth

Sawtooth supports several types of algorithms. A novel mechanism introduced with Sawtooth is called **PoET**. PoET is a trusted, executed environment-based lottery function that elects a leader randomly based on the time a node has waited for a block proposal.

PoET

PoET is a novel consensus algorithm that allows a node to be selected randomly based on the time that the node has waited before proposing a block. This concept is in contrast to other leader election- and lottery-based proof of work algorithms, such as the PoW used in Bitcoin, where an enormous amount of electricity and computer resources are used in order to be elected as a block proposer. PoET is a type of PoW algorithm but, instead of spending computer resources, it uses a trusted computing model to provide a mechanism to fulfill the PoW requirements. PoET makes use of Intel's SGX architecture (**Software Guard Extensions**) to provide a TEE to ensure the randomness and cryptographic security of the process.

It should be noted that the current implementation of Hyperledger Sawtooth does not require real hardware SGX-based TEE, as it is simulated for experimental purposes only and, as such, should not be used in production environments. The fundamental idea in PoET is to provide a mechanism of leader election by waiting randomly to be elected as a leader for proposing new transactions.

PoET, however, has a limitation, which has been highlighted by Ittay Eyal. This limitation is called the **stale chip** problem.



The research paper is available at <https://eprint.iacr.org/2017/179.pdf>.

This limitation results in hardware wastage, which can result in the waste of resources. The stale chip problem revolves around the idea that it is financially beneficial for malicious actors to collect lots of old SGX chips, which will increase their chances of becoming the miner of the next block. Note that they can collect lots of old SGX chips to build mining rigs, serving only one purpose of mining, instead of buying modern CPUs with SGX, which will not only serve in PoET consensus but will also be useful for other general computation.

Instead, they can choose to just collect as many old SGX-enabled chips as they can and increase their chances of winning the mining lottery. Also, old SGX-enabled CPUs are cheap and can result in the increased use of old, inefficient CPUs. It is similar to Bitcoin miners racing to get as many fast ASICs as possible to increase their chances of becoming the miner. It results in hardware wastage. There is also the possibility of hacking the chip's hardware. If an SGX chip is compromised, the malicious node can win the mining round every time, which results in complete system compromise and undeserved incentivizing of miners. This problem is also called the **broken chip** problem.

As demonstrated next, there are two types of PoET consensus, one with Intel SGX and another without SGX, along with a development mode mechanism.

PoET CFT

PoET CFT makes use of a simulated SGX environment. It is not dependent on hardware and can run on any processor. It is only CFT and not BFT.

PoET SGX

This is an SGX-dependent consensus mechanism. It provides BFT tolerance and consumes very little CPU processing power.

Devmode

As the name suggests, this is a development mode consensus mechanism that is suitable for testing transaction processors with single validator deployments. This is not intended for production usage. It makes use of a simple, random leader election algorithm for development and testing.

Other consensus mechanisms include PBFT and Raft.

PBFT

PBFT is a leader-based Byzantine fault-tolerant consensus mechanism. This is a commonly used protocol for consortium networks and many newer BFT protocols are based on this protocol. We discussed PBFT in detail in *Chapter 5, Consensus Algorithms*.

Raft

Raft is a comparatively faster algorithm than PoET. It is a CFT algorithm that works in a leader-follower fashion where a leader is elected for a period (term) of arbitrary time. It has immediate finality due to a lack of forks.

There is another consensus type called **quorum voting**, which is an adaptation of consensus protocols built by Ripple and Stellar. This consensus algorithm allows instant transaction finality, which is usually desirable in permissioned networks.

In this section, we have looked at the core feature of Sawtooth and explored various consensus mechanisms that Sawtooth supports. In the next section, we'll look at how transactions are generated and processed in Sawtooth.

Just like any other blockchain, in Sawtooth, transactions follow a protocol and a lifecycle. We will explore that in the next section.

Transaction lifecycle

Before describing the transaction flow, let's define what a transaction is. It is a function that, when executed, results in changing the state of the blockchain. Transactions are always made part of a batch, even if there is only one transaction in a batch. Validators receive these batches for processing.

Now that we understand what a transaction is, let's explore how a transaction flows in a Sawtooth network:

1. A client submits a transaction to a validator. This is usually achieved by utilizing the REST API available with the validator. By default, port `TCP 8008` is exposed where REST API is available.
2. After the transaction is submitted, it is propagated across the validator network.
3. One of the validators is elected as a leader, which then creates a candidate block and publishes it on the network.
4. This candidate block is propagated across the entire validator network.
5. When validators receive this block, it is validated. In addition, transaction processors validate and execute all transactions present in the candidate block.

- Once the block is validated and verified, it is written in its respective local storage and the state is updated accordingly.

This process can be visualized using the following diagram (note that the smaller node displays on the right are identical to the main node display on the center-left):

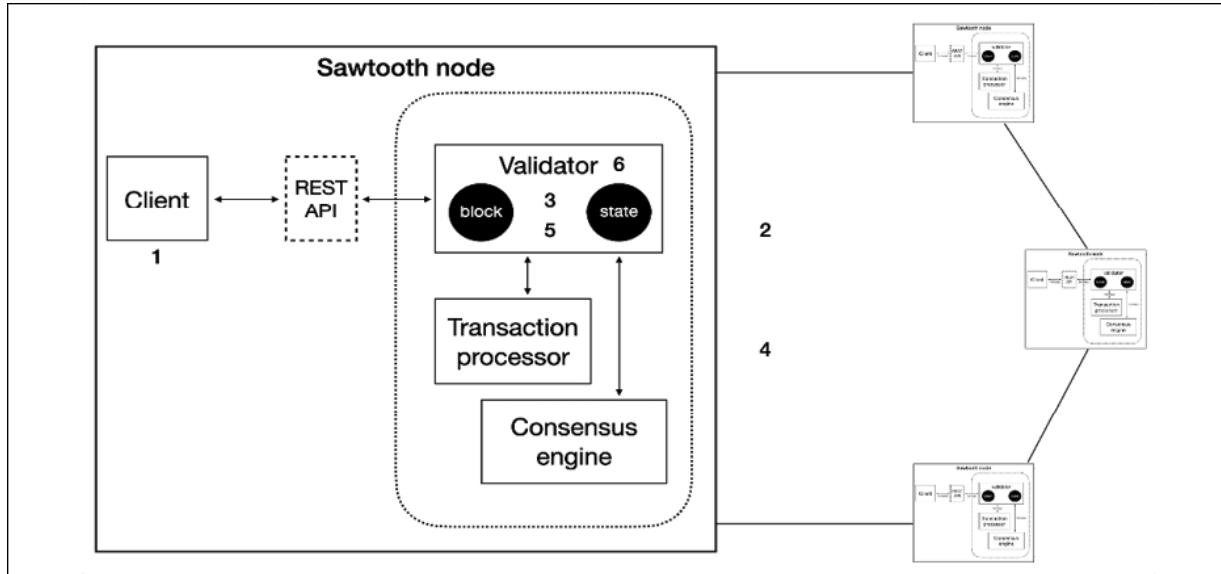


Figure 17.9: Sawtooth transaction lifecycle

Having understood the transaction flow in Sawtooth, let's explore in more detail what components Sawtooth is made up of and how they are interconnected.

Components

A Sawtooth node is composed of several components. A Sawtooth network is composed of several Sawtooth nodes. A generic Sawtooth node architecture and a network are shown in the following diagram note that the smaller node displays on the right are identical to the main node display on the center-left:

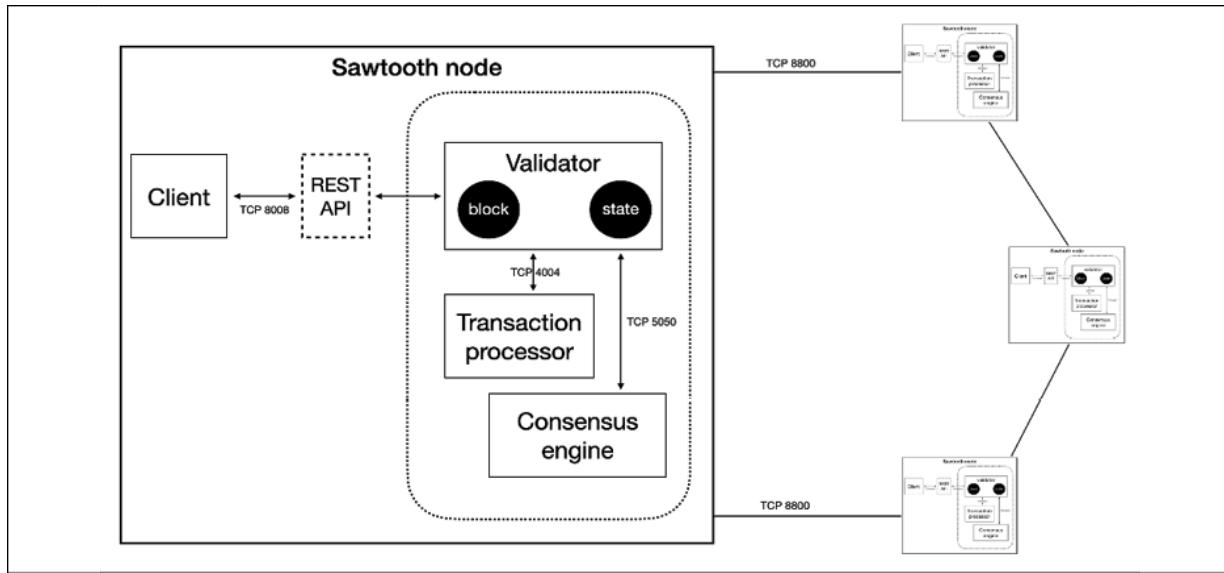


Figure 17.10: Sawtooth node and network

At a fundamental level, a Sawtooth node contains a validator, REST API, consensus engine, and transaction processors.

We will introduce each one of these components and their purpose in the following sections.

Validator

The validator is the component that is responsible for validating batches of transactions. It combines transactions into blocks and maintains consensus on the Sawtooth blockchain network. In addition, validators also perform coordination functions between clients, transaction processors, and other validators on the network. Validators communicate using the gossip protocol.

REST API

REST API is responsible for providing a communication interface between clients and validators. REST APIs are available by default over `TCP 8008`.

Client

The client is responsible for implementing the client logic. It is responsible for the following functions:

- Transaction creation and payload generation

- The handling of events in the blockchain and application, such as error handling, notifications, and blockchain state updates
- Displaying data to the users

State

The state in a blockchain is represented as a log of all changes made to the blockchain database since the start of the blockchain network or genesis block. All nodes in the Sawtooth network, just like any other blockchain, keep a local copy of the state database. The state is updated as a result of transaction execution and when new blocks are produced, each validator updates its local state database accordingly. The **global state**, on the other hand, represents the agreement on the state of the blockchain between all nodes on the network.

Transaction processors

Transaction processors are responsible for validating transactions and updating the state. State update and transaction validation is performed based on the rules defined in the associated transaction family. There is always a transaction family associated with a transaction processor. For example, there is a smallbank transaction family for the `smallbank_tp` transaction processor.

In summary, transaction processors are responsible for:

- Transaction validation
- Block creation
- Business logic or transaction rules

Transaction processors implement the business logic for a transaction family (or Sawtooth application). A transaction processor is responsible for transaction validation and state transition and updates governed by the rules defined in the Sawtooth application.

We will now explore the concept of a transaction family in detail.

Transaction families

A traditional smart contract paradigm provides a solution that is based on a general-purpose instruction set for all domains. For example, in the case of Ethereum, a set of opcodes has been developed for the EVM that can be used to build smart contracts to address any type of requirements for any industry.

While this model has its merits, it is becoming clear that this approach is not very secure as it provides a single interface into the ledger with a powerful and expressive language, which potentially offers a larger attack surface for malicious code. This complexity and generic virtual machine paradigm have resulted in several vulnerabilities that were found and exploited recently by hackers. A recent example is the **DAO hack** and further **Denial of Services (DoS)** attacks that exploited limitations in some EVM opcodes. The DAO hack was discussed in *Chapter 10, Smart Contracts*.

A model shown in the following figure describes the traditional smart contract model, where a **Generic virtual machine** has been used to provide the interface into the **Blockchain** for all domains:

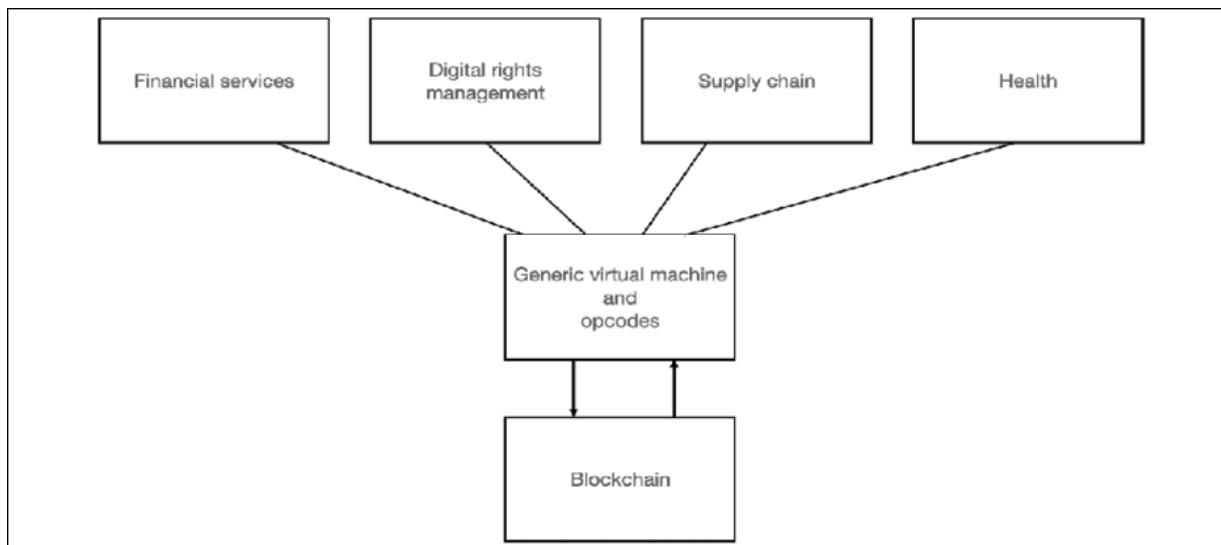


Figure 17.11: The traditional smart contract paradigm

In order to address this issue, Sawtooth has proposed the idea of **transaction families**. A transaction family is created by decomposing the logic layer into a set of rules and a composition layer for a specific domain. The key idea is that business logic is composed within transaction families, which provides a more secure and powerful way to build smart contracts. Transaction families contain the domain-specific rules and another layer that allows the creation of transactions for that domain. Another way of looking at it is that transaction families are a combination of a data model and a transaction language that implements a logic layer for a specific domain. The data model represents the current state of the blockchain (ledger), whereas the transaction language modifies the state of the ledger. It is expected that users will build their own transaction families according to their business requirements.



Note that a **transaction family** is also known as a Sawtooth application.

The following diagram represents this model, where each specific domain, such as **Financial services**, **Digital rights management (DRM)**, **Supply chain**, and the **Health** industry, has its own logic layer comprised of operations and services specific to that domain. This makes the logic layer both restrictive and powerful at the same time. Transaction families ensure that operations related to only the required domain are present in the control logic, thus removing the possibility of executing needless, arbitrary, and potentially harmful operations:

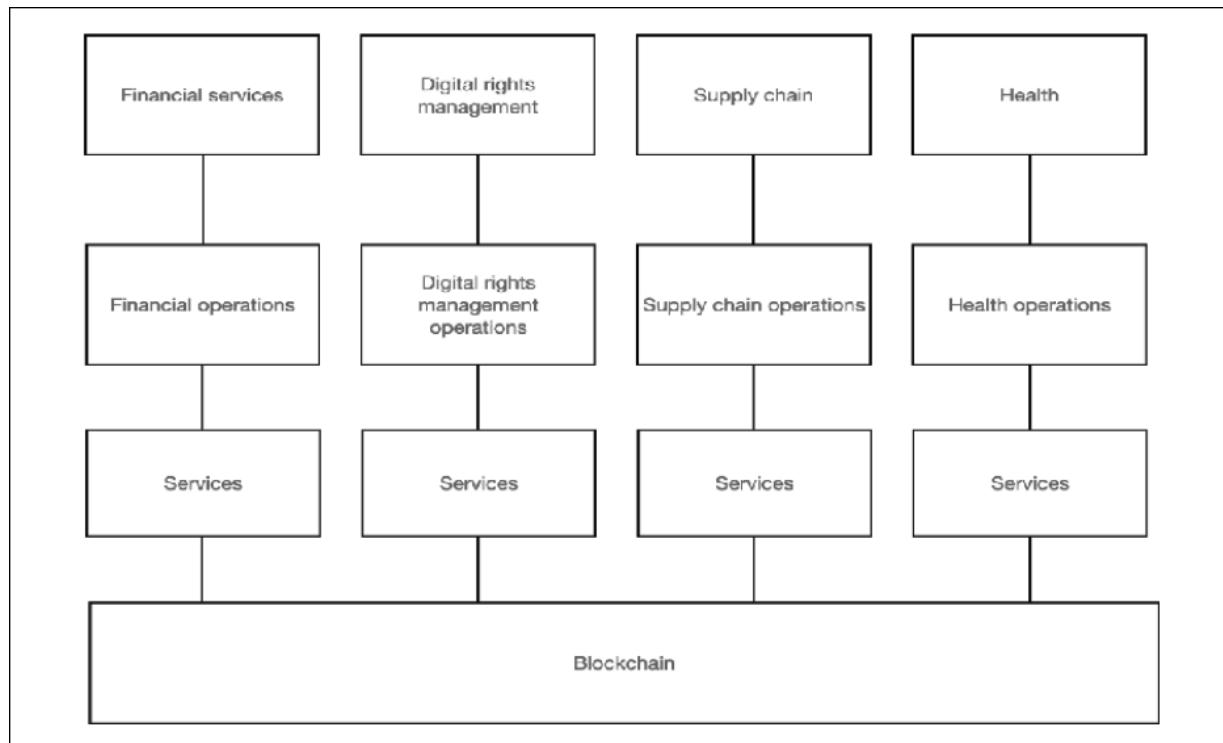


Figure 17.12: The Sawtooth (transaction families) smart contract paradigm

Sawtooth has provided several transaction families as examples. Each transaction family is identified by a name and has an associated transaction processor. The following table will explore these in detail.

Transaction family name	Function	Language	TF name	Transaction processor
BlockInfo	Provides a mechanism to store information about a configurable number of	Python	block_info	block-info-tp

	historic blocks.			
Identity	Mechanism for role- and policy-based permissioning.	Python	<code>sawtooth_identity</code>	<code>identity-tp</code>
Validator Registry	Provides a mechanism to add new validators to the network. Used in PoET to keep track of other validators.	Python	<code>sawtooth_validator_registry</code>	<code>poet-validator-registry-tp</code>
Settings	Used for storing on-chain configuration settings.	Python	<code>sawtooth_settings</code>	<code>settings-tp</code>
Smallbank	Provides a workload for comparing performance across different blockchain systems.	Go	<code>smallbank</code>	<code>smallbank-tp</code>
XO	An example that allows two users to play a simple game of tic-tac-toe.	Go, JavaScript/Node.js, and Python	<code>xo</code>	<code>xo-tp-python</code>

A transaction family encompasses application-specific business logic. This business logic defines various transaction types or operations that are permitted on the blockchain. Transaction families keep the transaction rules separate from the Sawtooth core blockchain functionality.

From an implementation point of view, transaction families implement data models and transaction languages for an application. A transaction family includes the following elements:

- A **transaction processor**, which defines the business logic for the application. A transaction processor performs the following functions:
 - Registration with the validator
 - Handling of transaction payloads
 - State update and retrieval as required
- A **data model**, which performs the following functions:
 - Records data
 - Stores data
 - Defines valid operations
 - Specifies data for the application (payload)



The data model must be the same between a client and a transaction processor. For example, encoding/serialization schemes and so on must be same.

- A **client**, which is responsible for the following functions:
 - Handling the client logic for the application
 - Creating and signing for the transactions
 - Creating batches of the transactions
 - Submitting transaction batches to the validator



Note that clients can use the REST API to submit transaction batches to the validator, or can choose to connect directly to the validator using ZeroMQ. However, usually the REST API is used in practice.

Transaction family is a new concept introduced with Hyperledger Sawtooth. It is a set of operations or types of transactions or a category of transactions that represents specific requirements of a use case. A data model and transaction language for specific applications are both implemented in a transaction family. Next, we will introduce REST APIs, which provide an API interface between a client and a transaction processor.

REST API

This is an optional component that runs as a service on a validator node, which facilitates communication between a client and a transaction processor. A REST API can also run as a separate system. Now that we've covered theoretical aspects and understand the Sawtooth architecture, let's introduce a practical example, which will help to better understand the theory covered so far.

Setting up a Sawtooth development environment

In this section, an introduction is given on how to set up a development environment for Sawtooth. There are a few prerequisites that are required in order to set up the development environment.

The easiest way to get Sawtooth up and running is by using Docker. In the following example, we will set up a 5-node network using Docker.

Prerequisites

For this process, you'll need to first install Docker. In this example we are using:

```
$ docker -v  
Docker version 19.03.8, build afacb8b
```

Sawtooth supports different consensus algorithms. In this example we will use PoET. However, other options are available too, such as PBFT. The YAML configuration files for both of these options are available at the links in the following sections.

Using PoET

The following link is available to download the YAML file for setting up Sawtooth with the PoET consensus algorithm:

https://sawtooth.hyperledger.org/docs/core/nightly/1-2/app_developers_guide/sawtooth-default-poet.yaml

Using PBFT

Here, you can access the YAML file for setting up Sawtooth with PBFT consensus:

https://sawtooth.hyperledger.org/docs/core/nightly/1-2/app_developers_guide/sawtooth-default-pbft.yaml

Setting up a Sawtooth network

In this section, we will see how a Sawtooth network can be created. First we start with creating a directory and then we will download specific configuration files, which will

help with the configuration of the network.

1. Create a directory named `sawtooth`:

```
$ mkdir sawtooth
```

2. Change the directory to `sawtooth`:

```
$ cd sawtooth
```

3. Download the PoET YAML file:

```
$ wget https://sawtooth.hyperledger.org/docs/core/nightly/1-2/app_devel
```

This command will show an output like the following, indicating that the `sawtooth-default-poet.yaml` file has been downloaded successfully. Note that only the final part of the output is shown for brevity:

```
...
2020-06-29 21:04:22 (51.9 KB/s) - 'sawtooth-default-poet.yaml' saved [1]
```

4. Start the network:

```
$ docker-compose -f sawtooth-default-poet.yaml up
```

This will show a long output and will take several minutes to complete. The output will be similar to the following, which shows the progress of the process:

```
Creating network "sawtooth_default" with the default driver
Creating volume "sawtooth_poet-shared" with default driver
Pulling intkey-tp-0 (hyperledger/sawtooth-intkey-tp-python:chime)...
chime: Pulling from hyperledger/sawtooth-intkey-tp-python
```

5. Once the network is started up successfully, we can verify that all the components are up by simply checking it in the Docker dashboard, as shown in the following screenshot:

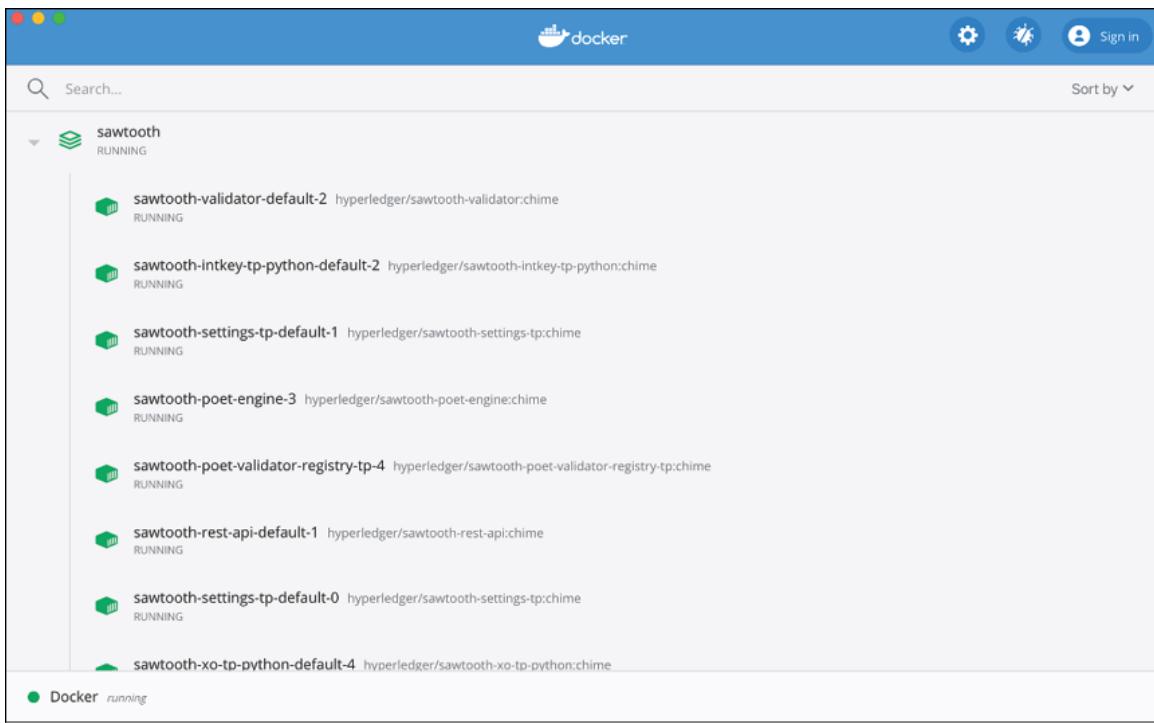


Figure 17.13: Docker dashboard showing the Sawtooth application



The Docker dashboard shown in *Figure 17.13* is available under the main menu in Docker Desktop. This step is optional, and only used to verify that all the required containers are running. As an alternative, an equivalent command line can also be used, for example, `docker ps`.

6. Confirm that the REST API is running by issuing the following command to connect to the REST API container on a node:

```
$ docker exec -it sawtooth-rest-api-default-0 bash
```

This will open a prompt like the following:

```
root@ea64259cf407:/#
```

7. Issue the command as shown here:

```
root@ea64259cf407:/# ps -eaf | grep 8008
```

This will show an output similar to the following, showing that a process is running for the REST API:

```
root      1      0  0 15:13 ?        00:00:01 /usr/bin/python3 /usr/l
```

Network functionality can be verified by using the following commands:

8. Connect to the shell container:

```
$ docker exec -it sawtooth-shell-default bash
```

This will open a prompt like the one shown here:

```
root@3893ec9be013:/#
```

9. Check whether the peers are up:

```
# curl http://sawtooth-rest-api-default-0:8008/peers
```

This will produce the following output, which displays the list of peers on the network:

```
{
  "data": [
    "tcp://validator-1:8800",
    "tcp://validator-3:8800",
    "tcp://validator-2:8800",
    "tcp://validator-4:8800"
  ],
  "link": http://sawtooth-rest-api-default-0:8008/peers
}
```

10. Another way to find peers is to use the `sawtooth` command:

```
# sawtooth peer list --url http://sawtooth-rest-api-default-0:8008
```

This command shows an output similar to the following. It shows the list of peers indicating the protocol used, the name of the validator, and the port:

```
tcp://validator-1:8800,tcp://validator-2:8800,tcp://validator-3:8800,tc
```

11. The peer list can also be retrieved using this command:

```
# sawnet peers list http://sawtooth-rest-api-default-0:8008
```

This command will show the output shown here, which shows that there are other validators available and running:

```
{
  "tcp://validator-0:8800": [
    "tcp://validator-1:8800",
    "tcp://validator-2:8800",
    "tcp://validator-3:8800",
    "tcp://validator-4:8800"
  ]
}
```

12. Similarly, the block list can also be retrieved:

```
# sawnet list-blocks http://sawtooth-rest-api-default-0:8008
```

This command will produce the following output, listing the height, ID, and previous hash of the blocks for `node 0`:

```
-- NODE 0 --
HEIGHT ID PREVIOUS
2 208f39df d22318b9
1 d22318b9 a1116c4f
0 a1116c4f 00000000
```

13. We can also view the Sawtooth settings, as shown in the following command:

```
$ sawtooth settings list --url http://sawtooth-rest-api-default-0:8008
```

This command shows all the settings of the blockchain:

```
sawtooth.consensus.algorithm.name: PoET
sawtooth.consensus.algorithm.version: 0.1
sawtooth.poet.initial_wait_time: 25
sawtooth.poet.report_public_key_pem:
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAOCAQ8AMIIIBCgKCAQEArMvzZi8GT+1I9KeZiInn
4CvFTiuyid+IN4dP1+m...
sawtooth.poet.target_wait_time: 5
sawtooth.poet.valid_enclave_basenames: b785c58b77152cbe7fd55ee3851c4990
sawtooth.poet.valid_enclave_measurements: c99f21955e38dbb03d2ca838d3af6
sawtooth.publisher.max_batches_per_block: 100
sawtooth.settings.vote.authorized_keys: 027ea24d40aa8d01ade0d9c6909e6bk
```

14. The block list can also be retrieved using the command shown here:

```
# sawtooth block list --url http://sawtooth-rest-api-default-0:8008
```

The output of this command displays the number, block IDs, batches, transactions and the signer:

NUM	BLOCK_ID	BATS	TXNS	SIGNER
2	208f39dfb18f264e416cd70ec262c9854cef9aff6dd0ff473eae7e8eccfa74	a0		
1	d22318b9f9741386e273f32c0ed76ad3360e4c619cb5a16d74b52ec9e729	b93b5		
0	a1116c4f18b44da5f9ea1c1517fee2029be900b054898c72a2cd99c907493	fed436d74		

15. Specific blocks can be queried using the command:

```
$ sawtooth block show
a1116c4f18b44da5f9ea1c1517fee2029be900b054898c72a2cd99c907493 fed436d74
```

This will show an output similar to the one shown here. The output is truncated for brevity:

```
batches:
- header:
```

```
signer_public_key: 02071c9bcd1d42440fdc3655f63561907a396fd0f39d56c
transaction_ids:
- bba77767e4ceaa405c2d6f33534d4aea840874ed8d577e1f51e1a9bb603451 32
header_signature: 5d9a20eff9cca693cb9e146c6285ddf3fd6a8e769e9628c79e5
trace: false
transactions:
- header:
```

This output shows the block structure with several fields including the header, transactions, state root hash, and many more.

In this section, we have seen how to set up a Sawtooth network and perform queries to find various pieces of information about the network, such as a list of peers and information about blocks and the blockchain.



An excellent online page where official Sawtooth examples are provided is available at <https://sawtooth.hyperledger.org/examples/>. Readers are encouraged to visit this page and explore these sample projects.

There are also sample projects available for Sawtooth. One example is `Sawtooth_bond`, which has been developed as a proof of concept to demonstrate a bond trading platform.

It is available at

<https://sawtooth.hyperledger.org/examples/bond.html>.

With this, a basic introduction to Sawtooth is complete. Sawtooth and Hyperledger in general are very large subjects and cannot be explained entirely in a single chapter. However, this chapter will serve as a solid foundation for further study.

Summary

In this chapter, we have gone through an introduction to the Hyperledger project. Firstly, the core ideas behind the Hyperledger project were discussed, and a brief introduction to all projects under Hyperledger was provided. Two main Hyperledger projects were discussed in detail, namely, Hyperledger Fabric and Sawtooth. All these projects are continuously improving and changes are expected in the next releases. However, the core concepts of all the projects mentioned previously are expected to remain unchanged or change only slightly. Readers are encouraged to visit the relevant links provided in the chapter to see the latest updates.

It is evident that a lot is going on in this space and projects like Hyperledger from the Linux Foundation are playing a pivotal role in the advancement of blockchain technology. Each of the projects discussed in this chapter has novel approaches toward

solving the issues faced in various industries, and any current limitations within the blockchain technology are also being addressed, such as scalability and privacy. It is expected that more projects will soon be proposed to the Hyperledger project, and it is envisaged that with this collaborative and open effort, blockchain technology will advance tremendously and collectively benefit the community as a whole.

In the next chapter, we will discuss tokenization which is one of the most prominent applications of blockchain technology and is disrupting many industries, especially the financial industry.

18

Tokenization

A token is a representation of an object. We use tokens in many different fields, including economics and computing. In daily life, tokens have been used to represent something of value, such as a gift voucher that is redeemable in exchange for items. In computing, different types of tokens are used, which can be defined as objects that represent eligibility to perform some operation. For example, an access token is used to identify a user and its privileges on a computer system. A hardware security token is used in computer systems to provide a means to authenticate a user (verify their identity) to a computer system. Tokens are also used in computer security mechanisms to substitute sensitive data with non-sensitive equivalents to prevent direct access to sensitive information. For example, in mobile payment systems, tokens are used to safeguard credit card information, which is represented by a token on the mobile device instead of the actual credit card data.

Just like the general use of tokens in day-to-day life, in the blockchain world, tokens also represent something of value. However, the critical difference is that the token exists digitally on a blockchain and is operated cryptographically, which means that they are generated, protected, and transferred using cryptographic protocols. Now that we understand what a token is, we can define tokenization as a process that converts an asset to a digital token on a blockchain. Specifically, it is a process of converting the ownership rights of a real-world asset into a cryptographic/digital token on a blockchain.

In this chapter, we will cover blockchain tokens (or crypto-tokens) and tokenization. We will use the term **token** from now on to represent tokens on a blockchain. We will also cover different types of tokens, relevant standards, tokenization platforms, applications of tokenization, and a

practical example of how to develop tokens on Ethereum. A guide to the specific topics we will cover in this chapter is provided here:

- Tokenization on the blockchain
- Types of token
- Token offerings
- Token standards
- Trading and finance
- DeFi
- Building an ERC-20 token
- Emerging concepts

Now, we'll begin our discussion on tokenization in the context of blockchain, which is the main aim of this chapter.

Tokenization on a blockchain

Tokenization, in the context of blockchain, is the process of representing an asset digitally on a blockchain. It can be used to represent commodities, real estate, ownership of art, currency, or anything else of value.



Remember, for the rest of the chapter when we refer to tokenization, it is in the context of blockchain.

After this brief definition, let's explore how tokenization can be beneficial.

Advantages of tokenization

Tokenization provides several benefits, including faster transaction processing, flexibility, low cost, decentralization, security, and transparency.

The following are some of the most important of these benefits:

- **Faster transaction processing:** As transactions and all relevant parties are present on the blockchain and readily available, there is no need to wait for a response from a counterparty or to wait for clearing and settlement operations. All these operations can be performed efficiently and quickly on a blockchain.
- **Flexibility:** Due to the worldwide adoption of systems that use tokens, such as payment systems, tokenization becomes simple due to easier cross-border use.
- **Low cost:** In comparison with traditional financial products, tokenization requires a lower cost to implement and incurs a lower cost to the end user due to digitization. Generally, in finance we have seen that digitization, starting from the introduction of digital accounts in the 1960s, has led to more efficient and cheaper financial systems. More recently, the introduction of digital payments has revolutionized the financial industry. In the same spirit, tokenization using blockchain can be considered another step toward achieving further efficiency and cost reduction. In fact, tokenization gives rise to a better, more efficient, and more democratic financial system. For example, just being able to share customer data recorded on one blockchain across all financial institutions reduces costs. Similarly, the possibility of converting illiquid assets into liquid assets is another way of increasing efficiency and profits.
- **Decentralization:** Tokens are presented on a public blockchain, leveraging the decentralization offered by blockchain technology. However, in some cases, a level of somewhat acceptable centralization is introduced due to the control and scrutiny required by investors and exchanges and other involved and interested parties.



We discussed the benefits of decentralization in detail in *Chapter 2, Decentralization*. Readers can refer to that chapter to review decentralization.

- **Security:** As tokens are cryptographically protected and produced using a blockchain, they are secure. However, note that proper

implementation must use good practice and meet established security industry standards, otherwise security flaws may result in exploitation by hackers, leading to financial loss.

- **Transparency:** Because they are on a blockchain, tokens are more transparent than traditional financial systems, meaning that any activity can be readily audited on a blockchain and is visible to everyone.
- **Trust:** As a result of the security and transparency guarantees, more trust is naturally placed in tokenization by investors.
- **Fractional ownership:** Imagine a scenario in which you own a painting by Vincent van Gogh. It is almost impossible in traditional scenarios to have multiple owners of the painting without immense legal, financial, and operational challenges. However, on a blockchain using tokens, any asset (such as our Van Gogh painting) can be fractionalized in such a way that it can be owned in part by many investors. The same situation is true for a property: if I wanted to have shared ownership with someone, it requires complicated legal procedures. However, with tokenization, fractional ownership of any asset, be it art, real estate, or any other off-chain real-world asset, is quick, easy, efficient, secure, and a lot less complicated than traditional methods.
- **Low entry barrier:** Traditional financial systems require traditional verification mechanisms, which can take a long time for a customer. While they are necessary and serve the purpose in traditional financial systems, these processes take a long time due to the necessary verification processes and the involvement of different entities. However, on a blockchain, this entry barrier is lowered because there is no need to go through the long verification checks. This is because not only is tokenization based on cryptographic guarantees provided by the blockchain, but for many **decentralized applications (DApps)** in this ecosystem, it is basically just a matter of downloading a DApp on your mobile device, depositing some funds if required, and becoming part of the ecosystem.
- **Innovative applications:** Tokenization has resulted in many innovative applications, including novel lending systems on blockchain, insurance, and other numerous financial applications,

including decentralized exchanges. Securities can now be tokenized and presented on blockchain, which results in client trust and satisfaction because of the better security and faster processing times. An example of this innovation is DeFi, which we will discuss later on in the chapter.

- **More liquidity:** This is due to the easy availability and accessibility of the tokens for the general public. By using tokens, even illiquid assets such as paintings can be tokenized and traded on an exchange with fractional ownership.

With all these advantages there are, however, some issues that must be addressed in the tokenization ecosystem. We'll discuss these next.

Disadvantages of tokenization

In this section, we present some of the disadvantages of tokenization. At the top of the list we have regulatory requirements:

- **Regulatory issues:** Regulation is a crucial subject of much debate. It is imperative that the tokens are regulated so that investors can have the same level of confidence that they have when they invest using traditional financial institutions. The big issue with tokenization and generally any blockchain technology is that they are mostly decentralized and in cases where no single organization is in control, it becomes quite difficult to hold someone responsible if something goes wrong. In a traditional system, we can go to the regulatory authorities or the relevant ombudsman services, but who is held responsible on a blockchain?

Some of this situation has changed with recent security tokenization standards and legislation, which consider tokens as securities. This means that security tokens will then be treated as securities, and the same legal, financial, and regulatory implications will be placed on these tokens that are applicable in the currently established financial industry. Refer to <https://www.sec.gov/answers/about-lawshtml.html>, where different laws that govern the securities industry are presented. This helps to increase customer confidence

levels and trust in the system; however, there are still many challenges that need to be addressed.

A new type of financial crime might be evolving with this tokenization ecosystem where, instead of using well-known and researched traditional financial fraud methods, criminals may choose to launch a technically sophisticated attack. For an average user, this type of attack is difficult to spot and understand as they are entirely on a blockchain and digitized. New forms of front running and market skewing on decentralized finance platforms is increasingly becoming a concern.

- **Legality of tokens:** This is a concern in some jurisdictions where tokens and cryptocurrency are illegal to own and trade.
- **Technology barrier:** Traditional financial systems have been the norm with brick and mortar banks. We are used to that system, but things have changed and are expected to change rapidly with tokenization. Tokenization-based financial ecosystems are easier to use for a lot of people, but for some, technological illiteracy can become an issue and could become a barrier. Sometimes the interfaces and software applications required to use tokenization platforms such as trading platforms are difficult to use for an average user.
- **Security issues:** The underlying blockchain technology is considered solid from a security point of view, and it is boasted sometimes that due to the use of cryptography, it is impossible to launch attacks and commit fraud on a blockchain. However, this is not entirely true, even with the firm security foundation that blockchains provide. The way tokenization platforms and DApps are implemented on the blockchain result in security issues that can be exploited by hackers, potentially leading to financial loss. In other words, even if the underlying blockchain is relatively secure, the tokenization DApp implemented on top may have vulnerabilities that could be exploited. These weaknesses could have been introduced by poor development practices or inherent limitations in the still-evolving smart contract languages.

In this section, we have looked at some of the pros and cons of tokenization. Next, let's look at some of the many types of token. Naturally, the way tokenization has been evolving in the last few years, there are now new and different types of tokens that are present in the ecosystem.

Types of tokens

With the rapid development of blockchain technology and the related applications, there has been a tremendous increase in the development of various types of token and relevant ecosystems. First, let's clarify the difference between a coin and token. Is a Bitcoin a token? Or are tokens and coins the same thing?

A coin is a native token of a blockchain. It is the default cryptocurrency of the blockchain on which it runs. Common examples of such a token are Bitcoin and ether. Both of these tokens or coins have their own native blockchain on which they run: the Bitcoin blockchain and the Ethereum blockchain.

On the other hand, a token is a representation of an asset that runs on top of a blockchain. For example, Ethereum not only has its own ether cryptocurrency as a native token (or coin) but also has thousands of other tokens that are built on top of Ethereum for different applications. Thanks to its support of smart contracts, Ethereum has become a platform for all sorts of different types of tokens ranging from simple utility tokens to game tokens and high-value, application-specific tokens.

Now that we understand the difference between a coin and a token, let's have a look at the usage and significance of different types of tokens. Tokens can be divided broadly into two categories based on their usage: **fungible** tokens and **non-fungible** tokens.

Fungible tokens

From an economics perspective, fungibility is the interchangeability of an asset with other assets of the same type. For example, a ten-pound note is interchangeable with a different ten-pound note or two five-pound notes. It does not matter which specific denominations they are: as long as they are of the same **type** (pound) and have same **value** (the sum of two five-pound notes), the notes are acceptable.

Fungible tokens work on the same principle. They are:

- **Indistinguishable:** Tokens of the same type are indistinguishable from each other. In other words, they are identical.
- **Interchangeable:** A token is fully interchangeable with another token of the same value.
- **Divisible:** Tokens are divisible into smaller fractions.

Non-fungible tokens

Let's consider **non-fungible tokens (NFTs)**, also called *nifty*, with the help of an example. We saw that fungibility allows the same types of token to be interchangeable, but non-fungible tokens are not interchangeable. For example, a collectible painting is a non-fungible asset, as it cannot be replaced with another painting of the same type and value. Each painting is unique and distinguishable from others.

Non-fungible tokens are:

- **Unique:** Non-fungible tokens are unique and different from other tokens of the same type or in the same category.
- **Non-interchangeable:** Since they are unique and represent specific attributes, these tokens are not interchangeable with tokens of the same type. For example, a rare painting is unique due to its attributes and is not interchangeable with another, even an exact-looking replica. We can also think about the certificate of authenticity that comes with a rare painting: that is also non-interchangeable with another due to its unique attributes representing the rare art.
- **Indivisible:** These tokens are available only as a complete unit. For example, a college diploma is a unique asset distinguishable from other diplomas of the same type. It is associated with a unique individual and is thus not interchangeable and it is not rational for it to be divided into fractions, making it indivisible.



One of the prime examples of NFTs is the game CryptoKitties—<https://www.cryptokitties.co>. It can be said that CryptoKitties played a big role in the popularity of non-fungible tokens. This is due to the fact that CryptoKitties was the first game (DApp) that was based on NFTs and, because of the interesting nature of the game, the community grew and many researchers and gamers developed interest not only in the game but also the underlying NFT mechanism. Also, this interest gave rise to more projects based on NFTs. A popular term, "crypto collectibles," also emerged with this development. Some other projects that use NFT and offer crypto collectibles include Gods Unchained, Decentraland, Skyweaver, and My Crypto Heroes.

In the next section, we introduce another interesting topic, stable tokens, which are a different type of token with interesting properties that make them more appealing to some investors.

Stable tokens

Stable tokens or stable coins are a type of token that has its value pegged to another asset's value, such as fiat currencies or precious metals. Stable tokens maintain a stable price against the price of another asset.

Bitcoin and other similar cryptocurrencies are inherently volatile and experience dramatic fluctuations in their price. This volatility makes them unsuitable for usual everyday use. Stable tokens emerged as a solution to this limitation in tokens.

The price stability is maintained by backing the token up by a stable asset. This is known as collateralization. Fiat currencies are stable because they are collateralized by some reserve, such as **foreign exchange (forex)** reserves or gold. Moreover, sound monetary policy and regulation by authorities play a vital role in the stability of a currency. Tokens or cryptocurrencies lack this type of support and thus suffer from volatility.

There are four types of stable coin.

Fiat collateralized

These stable tokens are backed by a traditional fiat currency such as US dollars. Fiat collateralized coins are the most common type of stable coins. Some of the common stable coins available today are **Gemini Dollar (GUSD)** (<https://gemini.com/dollar>), **Paxos (PAX)** (<https://www.paxos.com/pax/>), and **Libra** (<https://libra.org/>).

Commodity collateralized

As the name suggests, these stable coins are backed up by fungible commodities (assets) such as oil or gold. Common examples of such tokens are Digix gold tokens (<https://digix.global>) and Tether gold (<https://tether.to>).

Crypto collateralized

This type of stable coin is backed up by another cryptocurrency. For example, the Dai stable coin (<https://makerdao.com/en/>) accepts Ethereum-based assets as collateral in addition to soft pegging to the US dollar.

Algorithmically stable

This type of stable token is not collateralized by a reserve but stabilized by algorithms that keep track of market supply and demand. For example, Basecoin (<https://basecoin.cc>) can have its value pegged against the US dollar, a basket (a financial term for group, or collection of similar securities) of assets, or an index, and it also depends on a protocol to control the supply of tokens based on the exchange rate between itself and the peg used, such as the US dollar. This mechanism helps to maintain the stability of the token.

Security tokens

Security tokens derive their value from external tradeable assets. For example, security tokens can represent shares in a company. The difference is that traditional security is kept in a bank register and traded on traditional secondary markets, whereas security tokens are available on a blockchain. Being securities, they are governed by all traditional laws and regulations that apply to securities, but due to their somewhat decentralized nature, in which no middle man is required, security tokens are seen as a more efficient, scalable, and transparent option.

Now that we have covered different types of token, let's see how an asset can be tokenized by exploring the process of tokenization.

Process of tokenization

In this section, we'll present the process of tokenization, discuss what can be tokenized, and provide a basic example of the tokenization of assets.

Almost any asset can be tokenized and presented on a blockchain, such as commodities, bonds, stocks, real estate, precious metals, loans, and even intellectual property. Physical goods that are traditionally illiquid in nature, such as collectibles, intellectual property, and art, can also be tokenized and turned into liquid tradeable assets.

A generic process to tokenize an asset or, in other words, offer a security token, is described here. Note that there are many other technical details and intricacies involved in the process that we are skipping for brevity:

- The first step is to onboard an investor who is interested in tokenizing their asset.
- The asset that is presented for tokenization is scrutinized and audited, and ownership is confirmed. This audit is usually performed by the organization offering the tokenized security. It could be an exchange or a cryptocurrency start-up.

- The process of tokenized security starts, which leads to the **security token offering (STO)**.
- The physical asset is placed with a custodian (in the real world) for safekeeping.
- The **derivative token**, representing the asset, is created by the organization offering the token and issued to investors through a blockchain.
- Traders start to buy and sell these tokens using trading exchanges in a secondary market and these trades are settled (the buyer makes a payment and receives the goods) on a blockchain.

Common platforms for tokenization include Ethereum and EOS. Tezos is also emerging as another platform for tokenization due to its support for smart contracts. In fact, any blockchain that supports smart contracts can be used to build tokens.

At this point, a question arises about how all these different types of token reach the general public for investment. In the next section, we examine how this is achieved by first explaining what token offerings are and examining the different types.

Token offerings

Token offerings are mechanisms to raise funds and profit. There are a few different types of token offerings. We will introduce each of these separately now. One main common attribute of each of these mechanisms is that they are hosted on a blockchain and make use of tokens to facilitate different financial activities. These financial activities can include crowdfunding and trading securities.

Initial coin offerings

Initial coin offering or **initial currency offering (ICO)** is a mechanism to raise funds using cryptocurrencies. ICOs have been a very successful but somewhat controversial mechanism for raising capital for new cryptocurrency or token projects. ICOs are somewhat controversial sometimes due to bad design or poor governance, but at times some of the ICOs have turned out to be outright scams. A list of fraudulent schemes is available at <https://cryptochainuni.com/scam-list/>. These types of incidents have contributed toward the bad reputation and controversy of ICOs in general. While there are some scams, there are also many legitimate and successful ICOs. The **return on investment (ROI)** for quite a few of ICOs has been quite big and has contributed toward the unprecedented success of many of these projects. Some of these projects include Ethereum, NXT, NEO, IOTA, and QTUM.

A common question is asked here regarding the difference between the traditional **initial public offerings (IPOs)** and **ICOs**, as both of these mechanisms are fundamentally designed to raise capital. The difference is simple: ICOs are blockchain-based token offerings that are usually initiated by start-ups to raise capital by allowing investors to invest in their start-up. Usually in this case, contributions by investors are made using already existing and established cryptocurrencies such as Bitcoin or ether. As a return, when the project comes to fruition, the initial investors get their return on the investment.

On the other hand, IPOs are traditional mechanisms used by companies to distribute shares to the public. This is done using the services of underwriters, which are usually investment banks. IPOs are only usually allowed for those companies who are already well established, but ICOs on the other hand can be offered by start-ups. IPOs also offer dividends as returns, whereas ICOs offer tokens that are expected to rise in value once the project goes live.

Another key comparison is that IPOs are regulated, traditional mechanisms that are centralized in nature, while ICOs are decentralized and unregulated.

Because ICOs have been unregulated, which has resulted in a number of scams and people losing their money, another form of fundraising known as security token offerings was introduced. We'll discuss this next.

Security token offerings

Security token offerings (STOs) are a type of offering in which tokenized securities are traded at cryptocurrency exchanges. Tokenized securities or security tokens can represent any financial asset, including commodities and securities. The tokens offered under STOs are classified as securities. As such, they are more secure and can be regulated, in contrast with ICOs. If an STO is offered on a traditional stock exchange, then it is known as a tokenized IPO. Tokenized IPO is another name for an STO that is used when an STO is offered on a regulated stock exchange, which helps to differentiate between an STO offered on a traditional regulated exchange and one that is offered on cryptocurrency exchanges. STOs are regulated under the Markets in Financial Instruments Directive—MiFID II—in the European Union.

Initial exchange offerings

Initial exchange offering (IEO) is another innovation in the tokenization space. The key difference between an IEO and ICO is that in an ICO, the tokens are distributed via crowdfunding mechanisms to investors' wallets, but in an IEO, the tokens are made available through an exchange.

IEOs are more transparent and credible than ICOs due to the involvement of an exchange and due diligence performed by the exchange.

Equity token offerings

Equity token offerings (ETOs) are another variation of ICOs and STOs. ICOs offer utility tokens, whereas equity tokens are offered in ETOs. When compared with STOs, ETOs offer shares of a company, whereas STOs offer shares in any asset, such as currencies or commodities. From this point of view, ETOs can be regarded as a specific type of STO, where only shares in a company or venture are represented as tokens.

Decentralized autonomous initial coin offering

Decentralized Autonomous Initial Coin Offering (DAICO) is a combination of **decentralized autonomous organizations (DAOs)** and ICOs that enables investors to have more control over their investment and is seen as a more secure, decentralized, and automated version of ICOs.

Other token offerings

Different variations of ICOs and new concepts are being introduced quite regularly, and innovation is only expected to grow in this area.

A comparison of different types of token offering is presented in the following table:

Name	Concept	First introduced	Scale of decentralization	How to invest	Regulation
ICO	Crowdfunding through a utility token	In July 2013 with Mastercoin	Semi-decentralized	Investors send cryptocurrency to a smart contract released by the token offerer.	Low regulation
STO	Tokenized security such as bonds and stocks	2017	Somewhat centralized	Use the exchange provided platform	Regulated under established laws and directives in many jurisdictions
IEO	Tokens are made available	2018	Somewhat centralized	Use the exchange provided platform	Low regulation

	through an exchange				
ETO	Offers shares of any asset	December 2018 with the Neufund ETO	Somewhat centralized	Use the exchange provided platform	Mostly regulated
DAICO	Combination of DAO and ICO	May 2018 with ABYSS DAICO	Mostly decentralized	Investors send cryptocurrency to the DAICO smart contract	Low regulation

With all these different types of tokens and associated processes, a need to standardize naturally arises so that more adoption and interoperability can be achieved. To this end, several development standards have been proposed, which we discuss next.

Token standards

With the advent of smart contract platforms such as Ethereum, it has become quite easy to create a token using a smart contract. Technically, a token or digital currency can be created on Ethereum with a few lines of code, as shown in the following example:

```
pragma solidity ^0.5.0;
contract token {
    mapping (address => uint) public coinBalanceOf;
    event CoinTransfer(address sender, address receiver, uint amount);

    /* Initializes contract with initial supply tokens to the creator of the contract */
    function tokenx(uint supply) public {
        supply = 1000;
        coinBalanceOf[msg.sender] = supply;
    }
    /* Very simple trade function */
}
```

```
function sendCoin(address receiver, uint amount) public returns (bool)
{
    if (coinBalanceOf[msg.sender] < amount) return false;
    coinBalanceOf[msg.sender] -= amount;
    coinBalanceOf[receiver] += amount;
    emit CoinTransfer(msg.sender, receiver, amount);
    return true;
}
```



This code is based on one of the early codes published by Ethereum as an example on ethereum.org.

The preceding code works and can be used to create and issue tokens. However, the issue is that without any standard mechanism, everyone would implement tokenization smart contracts in their own way based on their requirements. This lack of standardization will result in interoperability and usability issues, which obstructs the widespread adoption of tokenization.

To remedy this, the first tokenization standard was proposed on Ethereum, known as ERC-20.

ERC-20

ERC-20 is the most famous token standard on the Ethereum platform. Many token offerings are based on ERC-20 and there are wallets available, such as MetaMask, that support ERC-20 tokens.

ERC-20 was introduced in November 2015 and since then has become a very popular standard for fungible tokens. There are almost 1,000 ERC-20 tokens listed on Etherscan (<https://etherscan.io/tokens>), which is a clear indication of this standard's popularity. It has been used in many ICOs and has resulted in valuable digital currencies (tokens) over the last few years, including EOS, Golem, and many others.

While ERC-20 defined a standard for fungible tokens and was widely adopted, it has some flaws, which results in some security and usability issues. For example, a security issue in ERC-20 results in a loss of funds if the tokens are sent to a smart contract that does not have the functionality to handle tokens. The effectively "burned" tokens result in a loss of funds for the user.

To address these shortcomings, ERC-223 was proposed.

ERC-223

ERC-223 is a fungible token standard. One major advantage of ERC-223 as compared to ERC-20 is that it consumes only 50% of ERC-20's gas consumption, which makes it less expensive to use on Ethereum mainnet. ERC-223 is backward compatible with ERC-20, and is used in a number of major token projects such as LINK and **CNexchange (CNEX)**.

ERC-777

The main aim of ERC-777 is to address some of the limitations of ERC-20 and ERC-223. It is backward compatible with ERC-20. It defines several advanced features to interact with ERC-20 tokens. It allows sending tokens on behalf of another address (contract or account). Moreover, it introduces a feature of "hooks," which allows token holders to have more control over their tokens.



The **Ethereum Improvement Proposal (EIP)** is available here:

<https://eips.ethereum.org/EIPS/eip-777>.

ERC-721

ERC-721 is a non-fungible token standard. ERC-721 mandates a number of rules that must be implemented in a smart contract for it to be ERC-721 compliant. These rules govern how these tokens can be managed and traded. ERC-721 was made famous by the **CryptoKitties** project. CryptoKitties is a blockchain game that allows players to create (breed) and trade different types of virtual cats on the blockchain. Each "kitty" is unique and tradeable for a value on the blockchain.

ERC-884

This is the standard for ERC-20 - compliant share tokens that are conformant with Delaware General Corporations Law.



The legislation is available here:

<https://legis.delaware.gov/json/BillDetail/GenerateHtmlDocument?legislationId=25730&legislationTypeId=1&docTypeId=2&legislationName=SB69>

The token standard EIP is available here:

<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-884.md>

ERC-1400

This is a security token standard that defines how to build, issue, trade, and manage security tokens. It is also referred to as ERC-1411 after it was renamed from ERC-1400.

Under ERC-1400, there are a few other standards, which are as follows.

- **ERC-1410:** A partially fungible token standard that defines a standard interface for grouping an owner's tokens into partitions.
- **ERC-1594:** This defines a core security token standard.
- **ERC-1643:** This is the document management standard.

- **ERC-1644:** This is the token controller operations standard.
- **ERC-1066:** This standard specifies a standard way to design Ethereum status codes.

The aim of ERC-1400 and the standards within it is to cover all activities related to the issuance, management, control, and processing of security tokens.

ERC-1404

This ERC standard allows the issuance of tokens with regulatory transfer restrictions. These restrictions enable users to control the transfer of tokens in different ways. For example, users can control when, to whom, and under what conditions the tokens can be transferred. For example, an issuer can choose to issue tokens only to a whitelisted recipient or check whether there are any timing restrictions on the senders' tokens.

ERC-1404 introduces two new functions in the ERC-20 standard to introduce restriction and control mechanisms. These two functions are listed as follows:

```
contract ERC1404 is ERC20 {  
    function detectTransferRestriction (address from, address to,  
    function messageForTransferRestriction (uint8 restrictionCode)  
}
```



More information on ERC-1404 is available at <https://erc1404.org>.

With this, we have completed our introduction to ERC standards.

Now let's have a quick introduction to finance and financial markets. This will provide a foundation for the material presented next, such as DeFi, as many of the terms and ideas are the same, albeit in a different context.

Trading and finance

Before we move onto exploring one of the largest ecosystems based on tokenization, **decentralized finance (DeFi)**, let's first look at some traditional finance and trading concepts. This will provide a solid foundation for the material presented in the next section and will covers token trading and related concepts such as decentralized exchanges and asset tokenization.

Financial markets

Financial markets enable the trading of financial securities such as bonds, equities, derivatives, and currencies. There are broadly three types of markets: money markets, credit markets, and capital markets:

- Money markets are short-term markets where money is lent to companies or banks for interbank lending. Foreign exchange, or forex, is another category of money markets where currencies are traded.
- Credit markets consist mostly of retail banks who borrow money from central banks and loan it to companies or households in the form of mortgages or loans.
- Capital markets facilitate the buying and selling of financial instruments, mainly stocks and bonds. There are many types of financial instruments, such as cash instruments, derivative instruments, loans, securities, and many more. Securitization is the process of creating a new security by transforming illiquid assets into tradeable financial instruments. Capital markets can be divided into two types: primary and secondary markets. Stocks are issued directly by the companies to investors in primary markets, whereas in secondary markets, investors resell their securities to other investors via stock exchanges. Various electronic trading systems are used by exchanges today to facilitate the trading of financial instruments.

Trading

A market is a place where parties engage in exchange. It can be either a physical location or an electronic or virtual location. Various financial instruments, including equities, stocks, foreign exchanges, commodities, and various types of derivatives are traded at these marketplaces. Recently, many financial institutions have introduced software platforms to trade various types of instruments from different asset classes.

Trading can be defined as an activity in which traders buy or sell various financial instruments to generate profit and hedge risk. Investors, borrowers, hedgers, asset exchangers, and gamblers are a few types of traders. Traders have a short position when they owe something; in other words, if they have sold a contract, they have a short position. When traders buy a contract, they have a long position. There are various ways to transact trades, such as through brokers or directly on an exchange or **over the counter (OTC)** where buyers and sellers trade directly with each other instead of using an exchange. Brokers are agents who arrange trades for their customers, and act on a client's behalf to deal at a given price or the best possible price.

Exchanges

Exchanges are usually considered to be a very safe, regulated, and reliable place for trading. During the last decade, electronic trading has gained popularity over traditional floor-based trading. Now, traders send orders to a central electronic order book from which the orders, prices, and related attributes are published to all associated systems using communications networks, thus in essence creating a virtual marketplace. Exchange trades can be performed only by members of the exchange. To trade without these limitations, the counterparties can participate in OTC trading directly.

Orders and order properties

Orders are instructions to trade, and they are the main building blocks of a trading system. They have the following general attributes:

- The instrument's name
- The quantity to be traded
- Direction (buy or sell)
- The type of the order that represents various conditions, for example, limit orders and stop orders



In finance, a limit order is a type of order that allows the selling or buying of an asset at a specific price or better. A stop order is similar, but the key difference is that a limit order is visible to the market, whereas a stop order only becomes active (as a market order) when the specified stop price is met.

Orders are traded by bid prices and offer prices. Traders show their intention to buy or sell by attaching bid and offer prices to their orders. The price at which a trader will buy is known as the *bid price*. The price at which a trader is willing to sell is known as the *offer price*.

Order management and routing systems

Order routing systems route and deliver orders to various destinations depending on the business logic. Customers use them to send orders to their brokers, who then send these orders to dealers, clearing houses, and exchanges.

There are different types of orders. The two most common ones are *market orders* and *limit orders*. A market order is an instruction to trade at the best price currently available in the market. These orders get filled immediately at spot prices.



In finance, *spot price* is a term used to refer to the current price of an asset in a marketplace at which it can be bought or sold for immediate delivery.

On the other hand, a limit order is an instruction to trade at the best price available, but only if it is not lower than the limit price set by the trader. This can also be higher depending on the direction of the order: either to sell or buy. All of these orders are managed in an *order book*, which is a list of orders maintained by the exchange, and it records the intention of buying or selling by the traders.

A position is a commitment to sell or buy a number of financial instruments, including securities, currencies, and commodities for a given price. The contracts, securities, commodities, and currencies that traders buy or sell are commonly known as trading instruments, and they come under the broad umbrella of **asset classes**. The most common classes are real assets, financial assets, derivative contracts, and insurance contracts.

Components of a trade

A trade ticket is the combination of all of the details related to a trade. However, there is some variation depending on the type of the instrument and the asset class. These elements are described here.

The underlying instrument

The underlying instrument is the basis of the trade. It can be a currency, a bond, an interest rate, a commodity, or an equity.

The attributes of financial instruments are discussed here.

General attributes

This includes the general identification information and essential features associated with every trade. Typical attributes include a unique ID, an instrument name, a type, a status, a trade date, and a time.

Economics

Economics are features related to the value of the trade; for example, the buy or sell value, ticker, exchange, price, and quantity.

Sales

Sales refers to the sales characteristic - related details, such as the name of the salesperson. It is just an informational field, usually without any impact on the trade lifecycle.

Counterparty

The counterparty is an essential component of a trade as it shows the other side (the other party involved in the trade) of the trade, and it is required to settle the trade successfully. The normal attributes include the counterparty name, address, payment type, any reference IDs, settlement date, and delivery type.

Trade lifecycle

A general trade lifecycle includes the various stages from order placement to execution and settlement. This lifecycle is described step-by-step as follows:

- **Pre-execution:** An order is placed at this stage.
- **Execution and booking:** When the order is matched and executed, it is converted into a trade. At this stage, the contract between counterparties is matured.
- **Confirmation:** This is where both counterparties agree to the particulars of the trade.
- **Post-booking:** This stage is concerned with various scrutiny and verification processes required to ascertain the correctness of the trade.
- **Settlement:** This is the most vital part of the trade life cycle. At this stage, the trade is final.
- **End-of-day processing:** End-of-day processes include report generation, profit and loss calculations, and various risk calculations.

This life cycle is also shown in the following image:

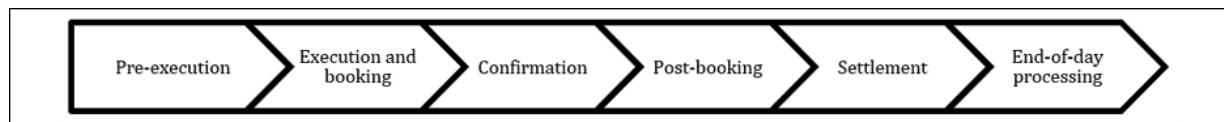


Figure 18.1: Trade life cycle

In all the aforementioned processes, many people and business functions are involved. Most commonly, these functions are divided into functions such as front office, middle office, and back office.

In the following section, we introduce some terminology that's relevant to crimes that occur in the financial industry.

Order anticipators

Order anticipators try to make a profit before other traders can carry out trading. This is based on the anticipation of a trader who knows how the activities of other trades will affect prices. Frontrunners, sentiment-oriented technical traders, and squeezers are some examples of order anticipators.

Market manipulation

Market manipulation is strictly illegal in many countries. Fraudulent traders can spread false information in the market, which can then result in price movements, enabling illegal profiteering. Usually, manipulative market conduct is trade-based, and it includes generalized and time-specific manipulations. Actions that can create an artificial shortage of stock, an impression of false activity, and price manipulation to gain criminal benefits are included in this category.

Both of these terms are relevant to financial crime. However, it is possible to develop blockchain-based systems that can thwart market abuse due to its transparency and security properties.

Now that we have gone through some basic financial and market concepts, let's dive into decentralized finance.

DeFi

DeFi stands for **decentralized finance**, which is the largest application of tokenization. It is an ecosystem that has emerged as a result of the development of many different types of financial applications built on top of blockchains. With blockchains being decentralized and the applications being related to finance, the term emerged as decentralized finance, or DeFi for short.

DeFi is an umbrella term used to describe a financial services ecosystem that is built on top of blockchains. **Centralized finance**, or **CeFi** for short, is a term now used to refer to the traditional financial services industry, which is centralized in nature, while DeFi is a movement to decentralize the traditional centralized financial services industry.



We covered the concept of decentralization in *Chapter 2, Decentralization*, which readers can refer to for review.

Tokenization plays a vital role in the DeFi ecosystem, which is the largest use of tokenization. DeFi is a vast subject with many different applications, protocols, assets, tokens, blockchains, and smart contracts.

The DeFi ecosystem is the fastest-growing infrastructure running on different blockchains. Most DeFi applications are running on Ethereum, but some are also running on EOS. The range of DeFi DApps includes, but is not limited to lending and borrowing, trading, asset management, insurance, tokenization, and prediction markets. All these decentralized applications, along with their smart contracts and infrastructure, make up the DeFi ecosystem.

In DeFi and in blockchain-based trading systems in general, the trading of tokens is the prime activity. We briefly describe this next.

Trading tokens

Tokens can be traded on exchanges. With the advent of DeFi, now **decentralized exchanges** are also emerging, abbreviated as **DEX**.

DEXs are decentralized and therefore require no central authority or intermediary to facilitate trading. Traders can trade with each other directly on a peer-to-peer basis. One of the advantages of DEXs due to decentralization is that traders are not required to transfer their funds to an intermediary; instead, funds can remain under their own control and they can directly trade with another counterparty without the involvement of an intermediary.

While the entry barrier to DEXs is low due to a lack of traditional **know your customer (KYC)** requirements, this also makes them risky for investors because if somehow the DEX is hacked, then there is no protection against loss of funds.



KYC is a standard due diligence practice in the financial services industry that ensures that the customer is legitimate and genuine.

Some examples of DEXs are Uniswap, Bancor, WavesDEX, and IDEX. This ecosystem is growing at a very fast pace and is only expected to grow further.

With all this activity in token trading, some malpractices are expected. In order to control these and ensure a fair and just system, regulation, which we'll discuss next, plays a vital role.

Regulation

Regulation of tokens is an important subject. In the last few years, especially after a wave of scam ICOs, there is genuine interest in protecting the investor. Due to this, many regulatory authorities have issued warnings and guidelines around ICOs, and some have also issued directives mandating the process of issuing and managing tokens. We will cover

regulation in a bit more detail in *Chapter 21, Scalability and Other Challenges*.

Now after all this theoretical background, let's see how a token can be built on Ethereum. We will build our own ERC-20 - compliant token, called **My ERC20 Token**, or **MET** for short.

Building an ERC-20 token

In this section, we will build an ERC-20 token. In previous chapters, we saw several ways of developing smart contracts, including writing smart contracts in Visual Studio Code and compiling them and then deploying them on a blockchain network. We also used the Remix IDE, Truffle, and MetaMask to experiment with various way of developing and deploying smart contracts. In this section, we will use a quick method to develop, test, and deploy our smart contract on a Goerli test network. We will not use Truffle or Visual Studio Code in this example as we have seen this method before; we are going to explore a quicker method to build and deploy our contract.

In this example, we will see how quickly and easily we can build and deploy our own token on the Ethereum blockchain network.

Pre requisites

We will use the following components in our example:

- **Remix IDE**, available at <http://remix.ethereum.org>. Note that in the following example, the user interface and steps required might be slightly different depending on the stage of development. This is because Remix IDE is under heavy development, and new features are being added to it at tremendous pace—as such, some changes are expected in the user interface too. However, the

fundamentals are not expected to change, and no major changes are expected in the user interface.

- **MetaMask**, which we installed in *Chapter 12, Further Ethereum*. We used it again in *Chapter 16, Serenity*, to deploy a contract on a Goerli test network, and we will use the same network in this example. You can create a new account if required.

Building the Solidity contract

Now let's start writing our code. First, we explore what the ERC-20 interface looks like, and then we will start writing our smart contract step by step in the Remix IDE.

The ERC-20 interface defines a number of functions and events that must be present in an ERC-20-compliant smart contract. Some more rules that must be present in an ERC-20-compliant token are listed here:

- `totalSupply` : This function returns the number of the total supply of tokens:

```
function totalSupply() public view returns (uint256)
```

- `balanceOf` : This function returns the balance of the token owner:

```
function balanceOf(address _owner) public view returns (uint256)
```

- `transfer` : This function takes the address of the recipient and a specified value as a number of tokens and transfers the amount to the address specified:

```
function transfer(address _to, uint256 _value) public returns (bool)
```

- `transferFrom` : This function takes `_from` (sender's address), `_to` (recipient's address), and `_value` (amount) as parameters and returns `true` or `false`. It is used to transfer funds from one account to another:

```
function transferFrom(address _from, address _to, uint256 _
```

- `approve`: This function takes `_spender` (recipient) and `_value` (number of tokens) as parameters and returns a Boolean, `true` or `false`, as a result. It is used to authorize the spender's address to make transfers on behalf of the token owner up to the approved amount (`_value`):

```
function approve(address _spender, uint256 _value) public r
```

- `allowance`: This function takes the address of the token owner and the spender's address and returns the remaining number of tokens that the spender has approval to withdraw from the token owner:

```
function allowance(address _owner, address _spender) public
```

There are three optional functions, which are listed as follows:

- `name`: This function returns the name of the token as a `string`. It is defined in code as follows:

```
function name() public view returns (string)
```

- `symbol`: This returns the symbol of the token as a `string`. It is defined as follows:

```
function symbol() public view returns (string)
```

- `decimals`: This function returns the number of decimals that the token uses as an integer. It is defined as follows:

```
function decimals() public view returns (uint8)
```

Finally, there are two events that must be present in an ERC-20-compliant token:

- `Transfer`: This event must trigger when tokens are transferred, including any zero-value transfers. The event is defined as follows:

```
event Transfer(address indexed _from, address indexed _to, i
```

- `Approval` : This event must trigger when a successful call is made to the `approve` function. The event is defined as follows:

```
event Approval(address indexed _owner, address indexed _spe
```

Now let's have a look at the source code of our ERC-20 token. This is written in Solidity, which we are familiar with and have explored in detail in *Chapter 14, Development Tools and Frameworks*.

Solidity contract source code

The source code for our ERC-20 token is as follows:

```
pragma solidity ^0.6.1;
contract MyERC20Token {
    mapping (address => uint256) _balances;
    mapping (address => mapping(address => uint256)) _allowed;

    string public name = "My ERC20 Token";
    string public symbol = "MET";
    uint8 public decimals = 0;
    uint256 private _totalSupply = 100;
    event Transfer(address indexed _from, address indexed _to, i
    event Approval(address indexed _owner, address indexed _spe
constructor() public {
    _balances[msg.sender] = _totalSupply;
    emit Transfer(address(0), msg.sender, _totalSupply);
}
function totalSupply() public view returns (uint) {
    return _totalSupply - _balances[address(0)];
}
function balanceOf(address _owner) public view returns (uint)
    return _balances[_owner];
}
function allowance(address _owner, address _spender) public
    return _allowed[_owner][_spender];
}
function transfer(address _to, uint256 _value) public returns (bool)
    require(_balances[msg.sender] >= _value, "value exceeds s
```

```

        _balances[msg.sender] -= _value;
        _balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
    function approve(address _spender, uint256 _value) public returns (bool)
    {
        _allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }
    function transferfrom(address _from, address _to, uint256 _value) public returns (bool)
    {
        require(_value <= _balances[_from], "Not enough balance");
        require(_value <= _allowed[_from][msg.sender], "Not enough allowance");
        _balances[_from] -= _value;
        _balances[_to] += _value;
        _allowed[_from][msg.sender] -= _value;
        emit Transfer(_from, _to, _value);
        return true;
    }
}

```

Let's explain this code step by step, before writing it into the Remix IDE.

First is the Solidity compiler and language version:

```
pragma solidity ^0.6.1;
```

The first line of the code specifies the compiler version using the `pragma` directive for which our program is written.

Then we create the `contract` object:

```
contract MyERC20Token {
```

After this, the `contract` object is defined with the name `MyERC20Token`.

Then we have the mappings:

```
mapping (address => uint256) _balances;
mapping (address => mapping(address => uint256)) _allowed;
```

These are the two mappings used in the ERC-20 smart contract. The first one is for keeping balances and the other one is used for allowances.

These are the state variables:

```
string public name = "My ERC20 Token";
string public symbol = "MET";
uint8 public decimals = 0;
uint256 private _totalSupply = 100;
```

They describe the name, symbol, and decimal precision points and the total supply of our token.

This is the `Transfer` event:

```
event Transfer(address indexed _from, address indexed _to, uint2
```

It has three parameters: `from`, `to`, and `value`. `from` represents the address from which the tokens are coming, `to` is the account to which tokens are being transferred, and `value` is the number of tokens.

This is the `Approval` event:

```
event Approval(address indexed _owner, address indexed _spender,
```

This has three parameters: owner address, recipient address, and `value`. The `indexed` keyword allows us to search for a specific log item instead of searching through all logs. It enables log filtration to search and extract only the required data instead of returning all logs.

This is the constructor that is executed when the contract is created:

```
constructor() public {
    _balances[msg.sender] = _totalSupply;
    emit Transfer(address(0), msg.sender, _totalSupply);
}
```

It is optional in Solidity and is used to run initialization code. In our example, the initialization code contains the statements to transfer the entire balance, `_totalSupply`, to the creator of the smart contract; in our case, it is the sender account. It also then emits the `Transfer` event, indicating that the transfer has taken place from `address(0)` to `msg.sender` (our contract creator) and `_totalSupply`, which is 100 in our case, just to keep things simple.

This is the `totalSupply` function:

```
function totalSupply() public view returns (uint) {
    return _totalSupply - _balances[address(0)];
}
```

This function returns the total amount of tokens after deducting it from the balance of the account.

This is the `balanceOf` function:

```
function balanceOf(address _owner) public view returns (uint bal
    return _balances[_owner];
}
```

The `balanceOf` function returns the balance of the token owner.

The `allowance` function is as follows:

```
function allowance(address _owner, address _spender) public view
    return _allowed[_owner][_spender];
}
```

The `allowance` function returns the total remainder of the tokens.

This is the `transfer` function, which returns `true` or `false` depending on the result of the execution:

```
function transfer(address _to, uint256 _value) public returns (bool)
{
    require(_balances[msg.sender] >= _value, "value exceeds sender's balance");
    _balances[msg.sender] -= _value;
    _balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

The `require` convenience function is used to check for certain conditions and throw an exception if the conditions are not met. In our example, this checks whether the value exceeds the sender's balance, and if it does, an error message will be generated stating that `value exceeds sender's balance`. If this check passes, the transfer occurs, and after emitting the `Transfer` event, the function returns `true`, indicating the successful transfer of tokens.

This is the `approve` function, which returns `true` or `false` depending on the result of the execution of the function:

```
function approve(address _spender, uint256 _value) public returns (bool)
{
    _allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

This function takes `_spender` (the user) and `_value` (number of tokens) as arguments and serves as a mechanism to provide approval to the user to acquire the allowed number of tokens from our ERC-20 contract.

This function is the `transferFrom` function, which can be used to automate the transfer of tokens from one address to another:

```
function transferfrom(address _from, address _to, uint256 _value
{
    require(_value <= _balances[_from], "Not enough balance")
    require(_value <= _allowed[_from][msg.sender], "Not enough allowance")
    _balances[_from] -= _value;
    _balances[_to] += _value;
    _allowed[_from][msg.sender] -= _value;
    emit Transfer(_from, _to, _value);
    return true;
}
```

It takes three parameters: `_from`, `_to`, and `_value`. It returns `true` or `false` depending upon the execution of the function. First, with the `require` functions, the balance and allowances are checked to ensure that enough balance and allowance is available. After that, the transfer occurs, and eventually the `Transfer` event is emitted, followed by a `true` Boolean value returned by the function indicating the successful transfer.

Now that we understand what our source code does, the next step is to write it in the Remix IDE and deploy it.

Deploying the contract on the Remix JavaScript virtual machine

In this step, we simply take the source code and write or simply paste it in the Remix IDE. Once pasted, this code will look like this:

The screenshot shows the Remix IDE interface. On the left, the 'SOLIDITY COMPILER' sidebar includes fields for 'COMPILER' (set to '0.6.6+commit.6c089d02'), 'LANGUAGE' (set to 'Solidity'), 'EVM VERSION' (set to 'compiler default'), and 'COMPILER CONFIGURATION' with options like 'Auto compile' (checked), 'Enable optimization' (unchecked), and 'Hide warnings' (unchecked). A prominent blue button labeled 'Compile erc20example.sol' is centered below these settings. To the right, the main workspace displays the Solidity source code for 'erc20example.sol'. The code defines a contract named 'MyERC20Token' with various functions for managing token balances, transfers, approvals, and total supply.

```
pragma solidity ^0.6.1;

contract MyERC20Token {
    mapping (address => uint256) _balances;
    mapping (address => mapping(address => uint256)) _allowed;

    string public name = "My ERC20 Token";
    string public symbol = "MET";
    uint8 public decimals = 0;
    uint256 private _totalSupply = 100;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    constructor() public {
        _balances[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
    }

    function totalSupply() public view returns (uint) {
        return _totalSupply - _balances[address(0)];
    }

    function balanceOf(address _owner) public view returns (uint balance) {
        return _balances[_owner];
    }

    function allowance(address _owner, address _spender) public view returns (uint remaining) {
        return _allowed[_owner][_spender];
    }

    function transfer(address _to, uint256 _value) public returns (bool success) {
        require(_balances[msg.sender] >= _value, "value exceeds senders balance");
        _balances[msg.sender] -= _value;
        _balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function approve(address _spender, uint256 _value) public returns (bool success) {
        _allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }
}
```

Figure 18.2: Remix IDE showing erc20example.sol

To achieve this, begin by opening the Remix IDE. When the Remix IDE opens up, it will show an interface similar to the following:

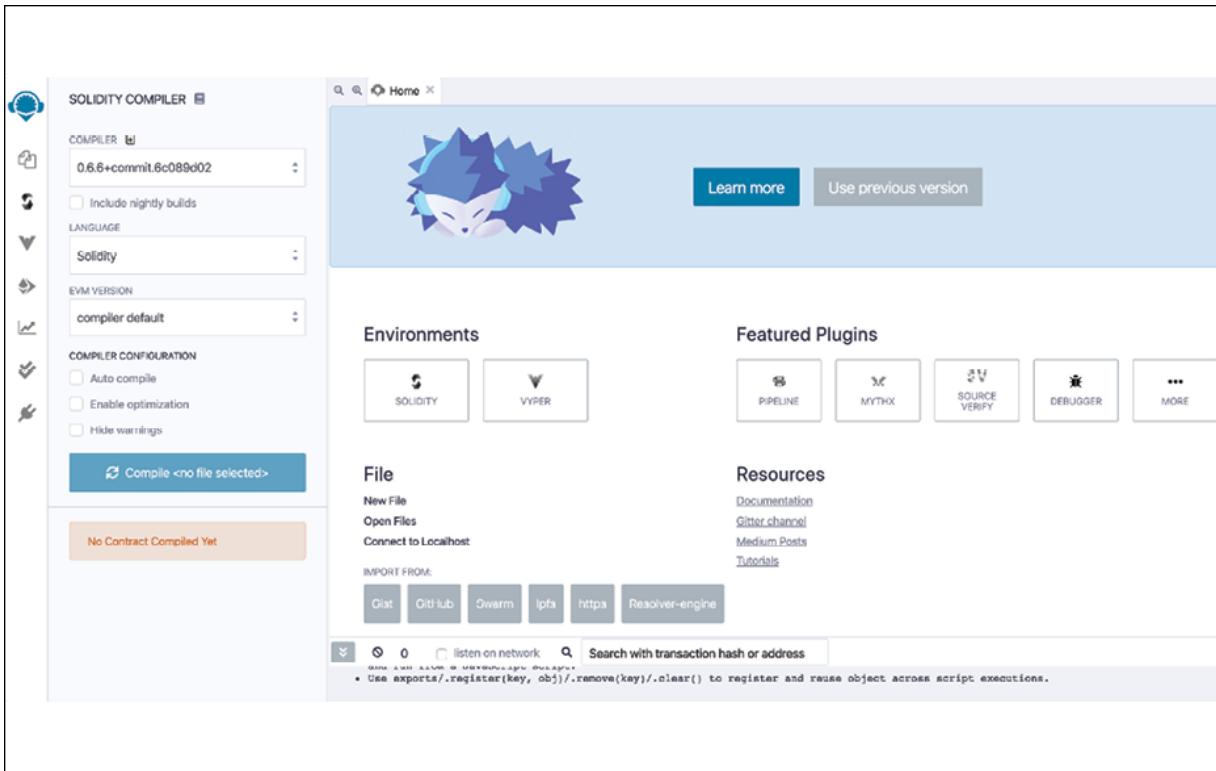


Figure 18.3: Remix IDE environments

Select **SOLIDITY** under **Environments**, as we are going to write smart contracts using the Solidity smart contract language.

After selecting the Solidity environment, create a new file by choosing the **FILE EXPLORERS** option from the list of icons on the left-hand side and add a new file by clicking the + sign:

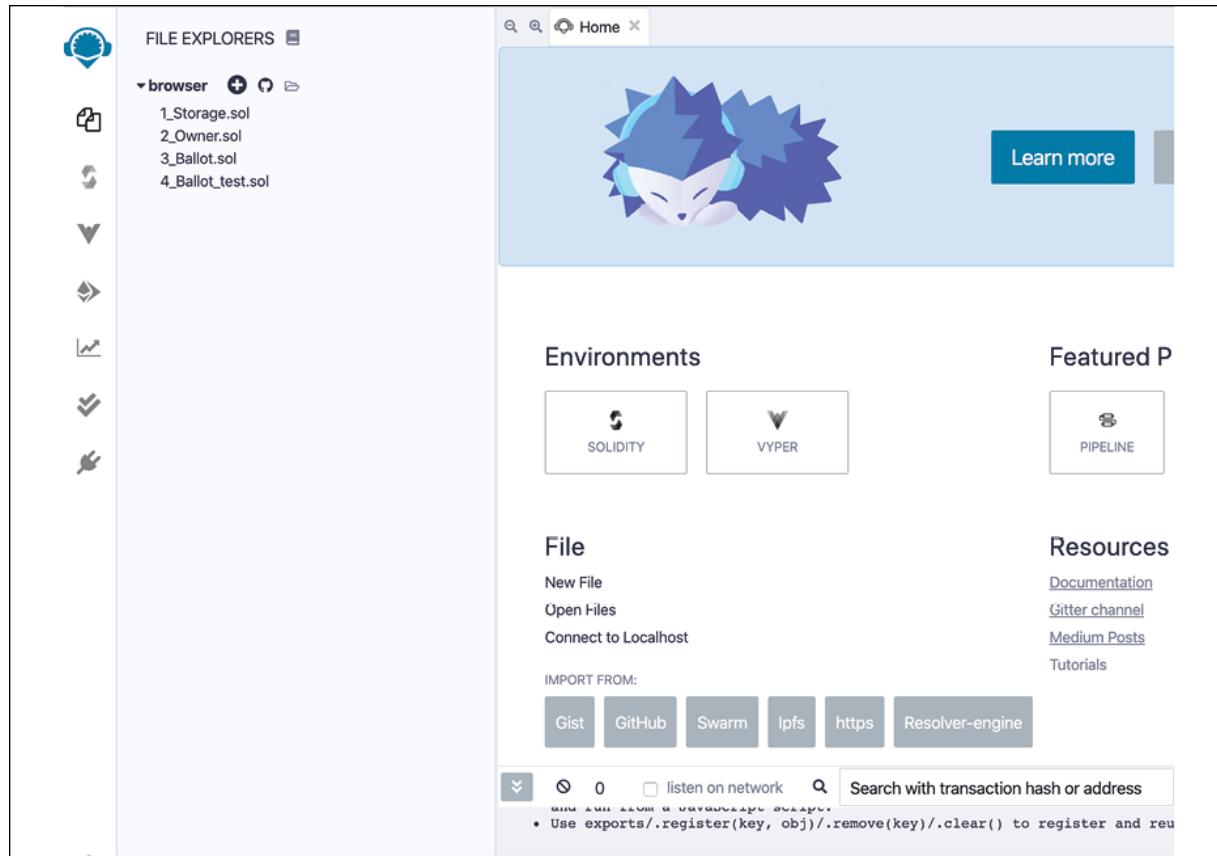


Figure 18.4: Remix IDE file explorer

Create a new file in the Remix IDE named `erc20example.sol`.

When we have created the new file, it will look like the following screenshot:

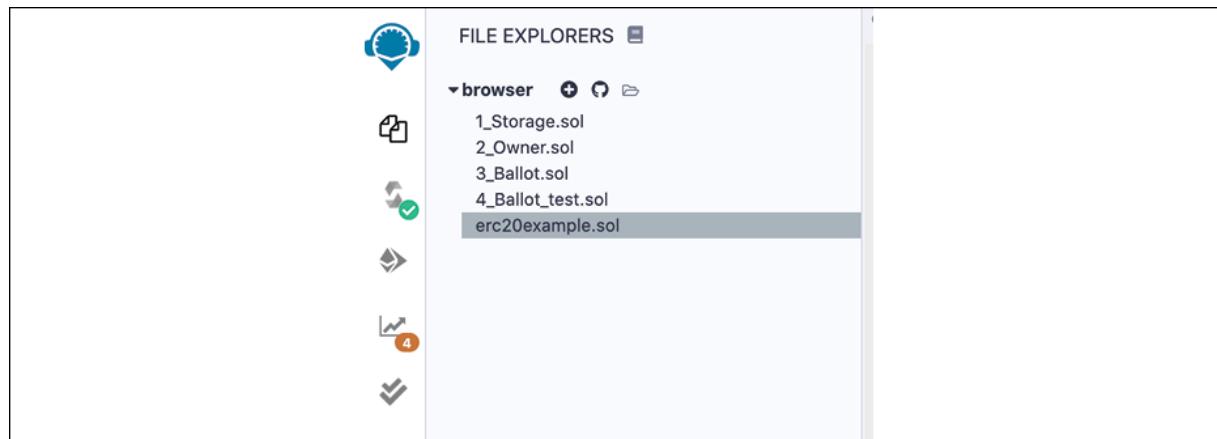
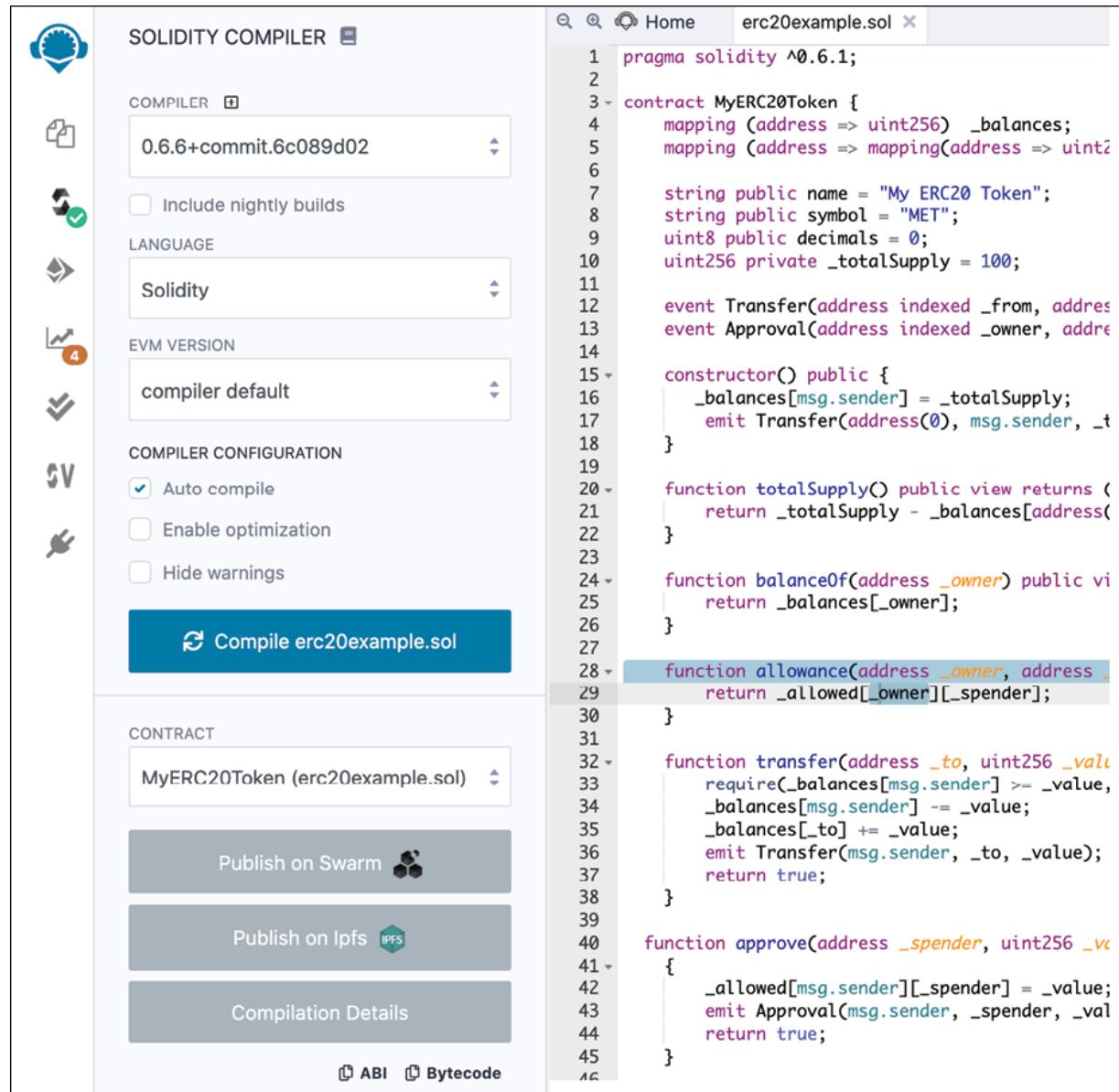


Figure 18.5: New file in Solidity

Now simply write the source code in the IDE, or paste it directly.

The next step is to compile the source code. To do this, click on **Compile erc20example.sol**. Optionally, **Auto compile** can also be selected under **COMPILER CONFIGURATION**, which will compile the code automatically as soon as it is written into the IDE:



The screenshot shows the Solidity Compiler interface in the Remix IDE. On the left, there's a sidebar with various icons. The main area has several dropdown menus and configuration options:

- COMPILER:** Set to "0.6.6+commit.6c089d02".
- LANGUAGE:** Set to "Solidity".
- EVM VERSION:** Set to "compiler default".
- COMPILER CONFIGURATION:** Includes checkboxes for "Auto compile" (which is checked), "Enable optimization", and "Hide warnings".
- Contract:** A dropdown set to "MyERC20Token (erc20example.sol)".
- Buttons:** "Publish on Swarm" and "Publish on ipfs".
- Compilation Details:** Buttons for "ABI" and "Bytecode".

The right side displays the Solidity source code for "erc20example.sol". The code defines a contract named "MyERC20Token" with methods for balance, allowance, transfer, and approve.

```
pragma solidity ^0.6.1;

contract MyERC20Token {
    mapping (address => uint256) _balances;
    mapping (address => mapping(address => uint256)) _allowances;

    string public name = "My ERC20 Token";
    string public symbol = "MET";
    uint8 public decimals = 0;
    uint256 private _totalSupply = 100;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);

    constructor() public {
        _balances[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
    }

    function totalSupply() public view returns (uint256) {
        return _totalSupply - _balances[address(0)];
    }

    function balanceOf(address _owner) public view returns (uint256) {
        return _balances[_owner];
    }

    function allowance(address _owner, address _spender) public view returns (uint256) {
        return _allowances[_owner][_spender];
    }

    function transfer(address _to, uint256 _value) public {
        require(_balances[msg.sender] >= _value, "Insufficient balance");
        _balances[msg.sender] -= _value;
        _balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
    }

    function approve(address _spender, uint256 _value) public {
        _allowances[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
    }
}
```

Figure 18.6: Solidity compiler in Remix

Once compiled successfully, it is ready to be deployed. First, we will deploy it on JavaScript VM available with the Remix IDE to ensure that everything

works. Once we have tested that it can be deployed correctly and works as per our expectations, we could deploy it on the mainnet using MetaMask. In our example, however, we will deploy it on the Goerli testnet using MetaMask.



To deploy on the mainnet, simply choose mainnet from MetaMask after ensuring that enough funds are available to deploy it on mainnet.

Remember that we have some ether left from our exercise in *Chapter 16, Serenity*, and have a test account on the Goerli network. We can use the same account for this example, or create a new account and fund it using the process in *Chapter 16, Serenity*. The aim of this exercise is to understand how MetaMask can be used to deploy our new token smart contract on an Ethereum network. We will also see how we can import ERC-20 tokens in MetaMask and use it to transfer funds from one account to another.

After compilation, we deploy the smart contract using the **Deploy & Run Transactions** interface available within the Remix IDE. This is shown in the following screenshot:

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons. The main area is divided into two sections: 'DEPLOY & RUN TRANSACTIONS' on the left and a code editor on the right.

DEPLOY & RUN TRANSACTIONS:

- ENVIRONMENT:** Set to 'JavaScript VM'.
- ACCOUNT:** An account address '0x24a...236B6 (99.999999!)' is selected.
- GAS LIMIT:** Set to '3000000'.
- VALUE:** Set to '0 wei'.
- CONTRACT:** The contract is named 'MyERC20Token - browser/erc20e...'.
- Deploy:** A brown button.
- PUBLISH TO IPFS:** An unchecked checkbox.
- OR:** A section with 'At Address' and 'Load contract from Address' buttons.
- Transactions recorded:** Shows '0' transactions.
- Deployed Contracts:** A list with a delete icon.
- Note:** A message says 'Currently you have no contract instances to interact with.'

Solidity Code Editor:

```
pragma solidity ^0.6.1;

contract MyERC20Token {
    mapping (address => uint256) _balances;
    mapping (address => mapping(address => uint256)) _allowances;
    string public name = "My ERC20 Token";
    string public symbol = "MET";
    uint8 public decimals = 0;
    uint256 private _totalSupply = 1000000000000000000000000;
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor() public {
        _balances[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
    }

    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address _owner) public view returns (uint256) {
        return _balances[_owner];
    }

    function transfer(address _to, uint256 _value) public returns (bool) {
        require(_value <= _balances[msg.sender]);
        require(_to != address(0));
        _balances[msg.sender] -= _value;
        _balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
        require(_value <= _balances[_from]);
        require(_value <= _allowances[_from][msg.sender]);
        require(_to != address(0));
        _balances[_from] -= _value;
        _balances[_to] += _value;
        _allowances[_from][msg.sender] -= _value;
        emit Transfer(_from, _to, _value);
        return true;
    }

    function approve(address _spender, uint256 _value) public returns (bool) {
        _allowances[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    function allowance(address _owner, address _spender) public view returns (uint256) {
        return _allowances[_owner][_spender];
    }
}
```

Figure 18.7: Deploying and running transactions in Remix

Make sure the environment JavaScript VM is selected and an account is selected from which to deploy, and then select **Deploy**.

Once it's deployed, we will see an output similar to the following screenshot, where the contract will become available under **Deployed Contracts** and, in the logs, we can see relevant details regarding the deployment of the contract:

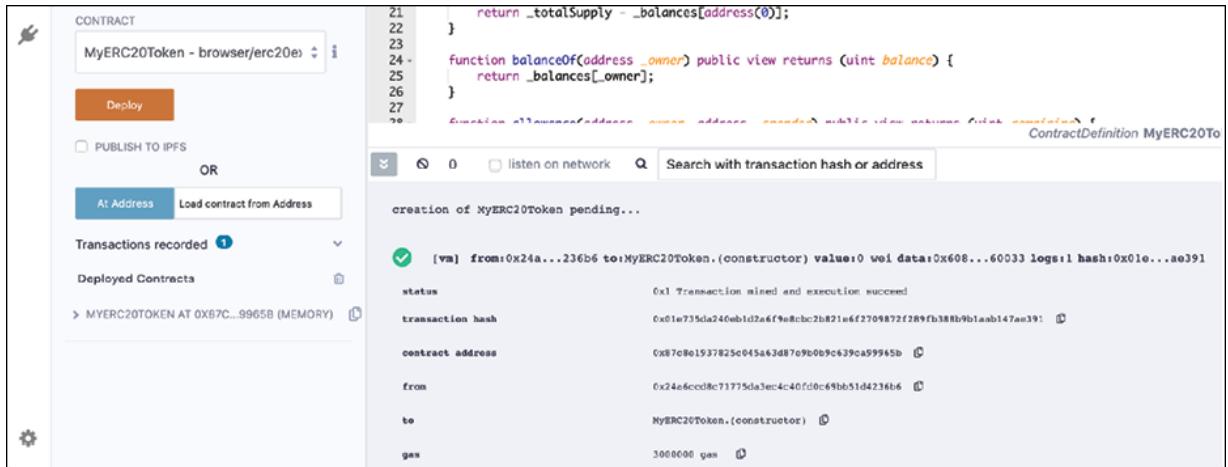


Figure 18.8: Contract creation in Remix

Most importantly, we can see in the logs that when the contract is created, the first event emitted is `'Transfer'`. Here, we can see that all 100 tokens have been transferred to the owner account,

0x87C8E1937825c045a63D87e9b0B9c639ca99965b :

Once it's deployed, we will see under **Deployed Contracts** all the functions exposed by the contract in the Remix IDE:

The screenshot shows the Remix IDE interface with the following components:

- Contract Address:** MYERC20TOKEN AT 0x87C...9965B (MEMORY)
- Exposed Functions:**
 - approve: address _spender, uint256 _value
 - transfer: address _to, uint256 _value
 - transfer_from: address _from, address _to, uint256 _value
 - allowance: address _owner, address _spender
 - balanceOf: address _owner
 - decimals: uint8
 - name: string
 - symbol: string
 - totalSupply: uint256
- Source Code:**

```

4   mapping (address => uint256)
5   mapping (address => mapping(address =>
6     uint8 public decimals = 0
7     uint256 private _totalSupply
8
9     event Transfer(address indexed from, address indexed to, uint256 value)
10    event Approval(address indexed owner, address indexed spender, uint256 value)
11
12    constructor() public {
13      _balances[msg.sender] = _totalSupply;
14      emit Transfer(address(0), msg.sender, _totalSupply);
15    }
16
17    function totalSupply() public view returns (uint256) {
18      return _totalSupply;
19    }
20
21    function balanceOf(address owner) public view returns (uint256) {
22      return _balances[owner];
23    }
24
25    function allowance(address owner, address spender) public view returns (uint256) {
26      return _allowances[owner][spender];
27    }
28
29    function transfer(address to, uint256 value) public returns (bool) {
30      require(value <= _balances[msg.sender], "ERC20: transfer amount exceeds balance");
31      _balances[msg.sender] -= value;
32      _balances[to] += value;
33      emit Transfer(msg.sender, to, value);
34      return true;
35    }
36
37    function approve(address spender, uint256 value) public returns (bool) {
38      require(spender != address(0), "ERC20: approve from the zero address");
39      _allowances[msg.sender][spender] = value;
40      emit Approval(msg.sender, spender, value);
41      return true;
42    }
43
44    function transferFrom(address from, address to, uint256 value) public returns (bool) {
45      require(_allowances[from][msg.sender] >= value, "ERC20: transferFrom value exceeds allowance");
46      require(value <= _balances[from], "ERC20: transferFrom amount exceeds balance");
47      _allowances[from][msg.sender] -= value;
48      _balances[from] -= value;
49      _balances[to] += value;
50      emit Transfer(from, to, value);
51      return true;
52    }
53
54    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
55      require(spender != address(0), "ERC20: increaseAllowance from the zero address");
56      _allowances[msg.sender][spender] += addedValue;
57      emit Approval(msg.sender, spender, _allowances[msg.sender][spender]);
58      return true;
59    }
60
61    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
62      require(_allowances[msg.sender][spender] >= subtractedValue, "ERC20: decreaseAllowance subtractedValue exceeds allowance");
63      _allowances[msg.sender][spender] -= subtractedValue;
64      emit Approval(msg.sender, spender, _allowances[msg.sender][spender]);
65      return true;
66    }
67
68    function name() public view returns (string memory) {
69      return _name;
70    }
71
72    function symbol() public view returns (string memory) {
73      return _symbol;
74    }
75
76    function decimals() public view returns (uint8) {
77      return _decimals;
78    }
79
80    function _mint(address to, uint256 value) internal {
81      require(to != address(0), "ERC20: mint to the zero address");
82      _totalSupply += value;
83      _balances[to] += value;
84      emit Transfer(address(0), to, value);
85    }
86
87    function _burn(address from, uint256 value) internal {
88      require(value <= _balances[from], "ERC20: burn value exceeds balance");
89      _totalSupply -= value;
90      _balances[from] -= value;
91      emit Transfer(from, address(0), value);
92    }
93
94    function _approve(address owner, address spender, uint256 value) internal {
95      require(spender != address(0), "ERC20: approve from the zero address");
96      _allowances[owner][spender] = value;
97      emit Approval(owner, spender, value);
98    }
99
100   function _transfer(address from, address to, uint256 value) internal {
101      require(from != address(0), "ERC20: transfer from the zero address");
102      require(to != address(0), "ERC20: transfer to the zero address");
103      require(value <= _balances[from], "ERC20: transfer value exceeds balance");
104      _balances[from] -= value;
105      _balances[to] += value;
106      emit Transfer(from, to, value);
107    }
108
109    receive() external payable {
110      revert("ERC20: ETH transfers are not allowed");
111    }
112  }
```
- Call History:** Shows two calls to the balanceOf function.

Figure 18.9: Exposed functions in Remix for our contract

We can test the functionality of our contract using this interface. For example, calling `totalSupply` shows a value of `100`, which indicates the number of tokens, and calling `symbol` shows the string `MET`, our token's symbol.

At this point, our contract works and we have tested it locally. Now we can deploy it onto the Goerli testnet. We will use MetaMask for this purpose.

In the Remix IDE, under **DEPLOY & RUN TRANSACTIONS**, select the **Injected Web3** option, as shown in the following screenshot:

```

1 pragma solidity ^0.6.1;
2
3 contract MyERC20Token {
4     mapping (address => uint256) _balances;
5     mapping (address => mapping(address => uint256))
6
7         string public name = "My ERC20 Token";
8         string public symbol = "MET";
9         uint8 public decimals = 0;
10        uint256 private _totalSupply = 100;
11
12        event Transfer(address indexed _from, address indexed _to, uint256 _value);
13        event Approval(address indexed _owner, address indexed _spender, uint256 _value);
14
15    constructor() public {
16        _balances[msg.sender] = _totalSupply;
17        emit Transfer(address(0), msg.sender, _totalSupply);
18    }
19
20    function totalSupply() public view returns (uint256) {
21        return _totalSupply - _balances[address(0)];
22    }
23
24    function balanceOf(address _owner) public view returns (uint256) {
25        return _balances[_owner];
26    }
27
28    function allowance(address _owner, address _spender) public view returns (uint256) {
29        return _allowances[_owner][_spender];
30    }
}

```

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons. The main area has tabs for 'DEPLOY & RUN TRANSACTIONS' and 'Solidity'. In the 'DEPLOY & RUN TRANSACTIONS' tab, the 'ENVIRONMENT' dropdown is set to 'Injected Web3', which is highlighted in blue. Below it, there are fields for 'GAS LIMIT' (set to 3000000) and 'VALUE' (set to 0 wei). Under 'CONTRACT', the dropdown shows 'MyERC20Token - browser/erc20example.sol'. A large orange 'Deploy' button is visible. Below the button are checkboxes for 'PUBLISH TO IPFS' and 'At Address' (which is selected). There's also a 'Transactions recorded' section with a count of 1. At the bottom, there's a dropdown for 'MYERC20TOKEN AT 0X87C...9965B (MEMORY)'.

The right side of the interface is the 'Solidity' tab where the code for 'erc20example.sol' is displayed. The code defines an ERC20 token contract named 'MyERC20Token' with standard functions like 'balanceOf', 'totalSupply', and 'transferApproval'. The 'Injected Web3' environment is noted as being provided by Metamask or similar provider.

Figure 18.10: Remix Inject Web3

This is the execution environment that is provided by enabling `web3` within the browser using the MetaMask plugin.

First, confirm that MetaMask is running and connected to the Goerli test network. This should be easy to check, as we used this same network in *Chapter 16, Serenity*.

If everything is working in MetaMask, it should display a screen similar to the following screenshot. Note that you may have to log in again to MetaMask. Once you're logged in, select the Goerli network and an account that has some ether in it:

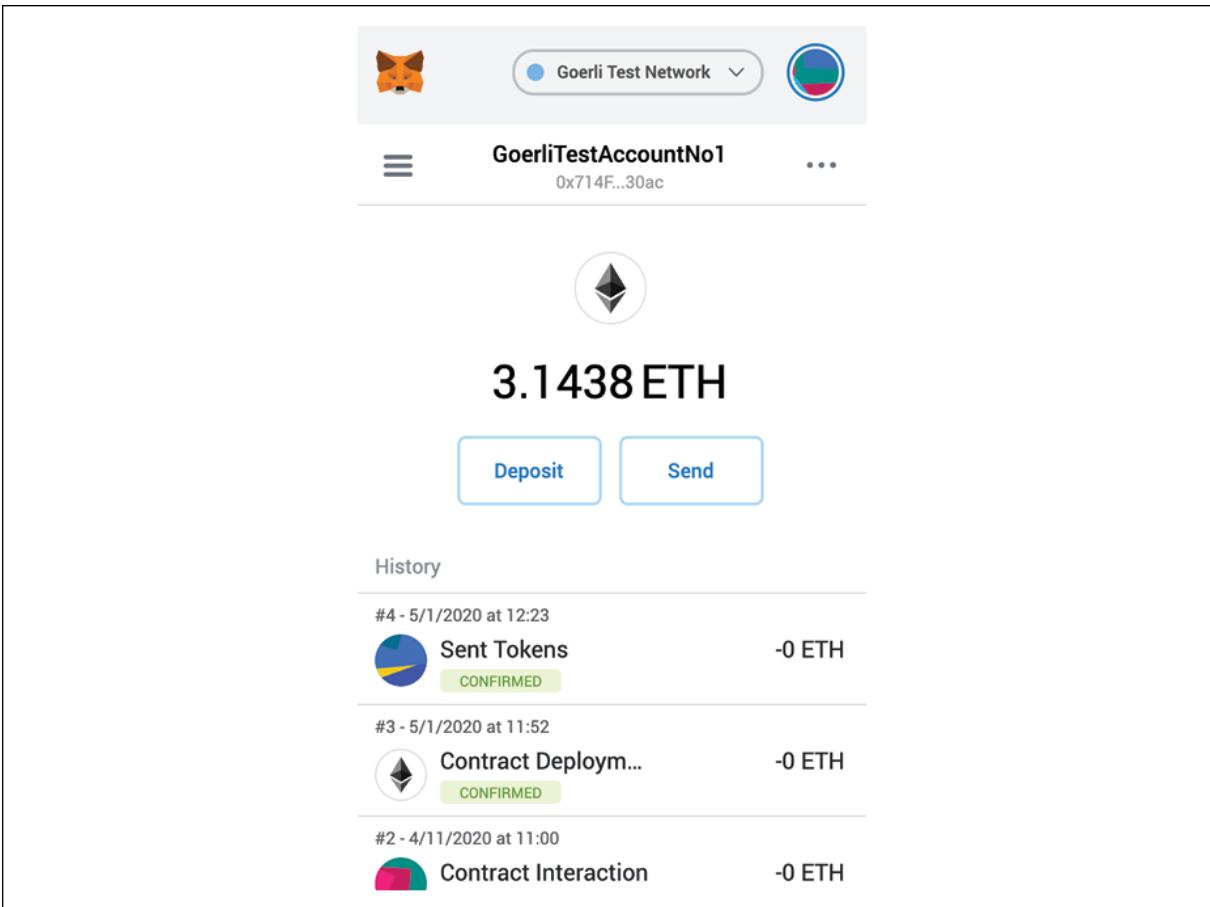


Figure 18.11: Goerli test network in MetaMask

In the Remix IDE, choose **Injected Web3** in the **ENVIRONMENT** field under **DEPLOY & RUN TRANSACTIONS**. This is shown in the following screenshot:

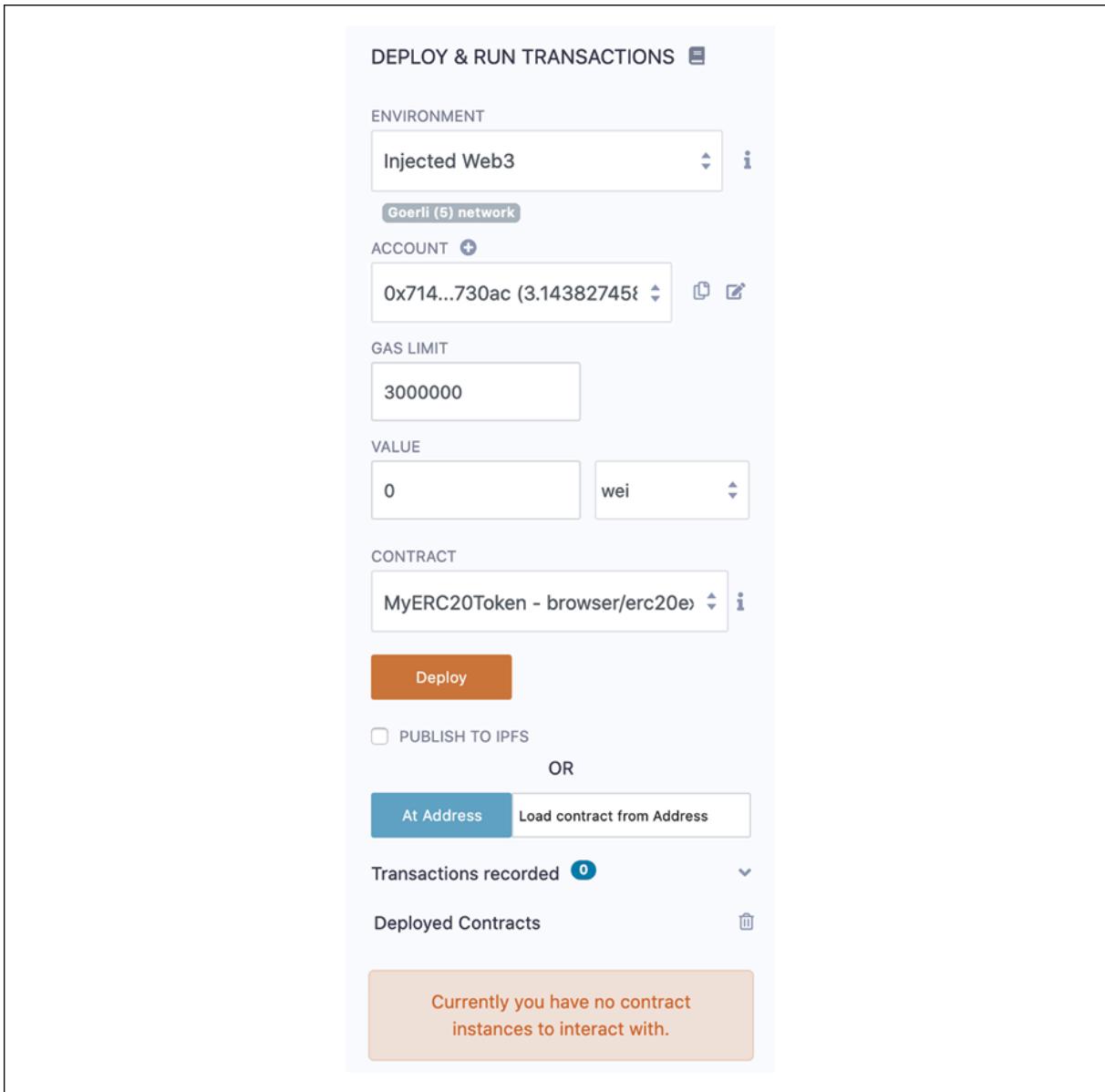


Figure 18.12: DEPLOY & RUN TRANSACTIONS in MetaMask

Note that in the screenshot, the **Goerli (5) network** is selected. It also shows the account that we have set up in MetaMask. Now we are all set to deploy this on the Goerli test network.

Click **Deploy**, which will open the MetaMask window to confirm the transaction, as shown here:

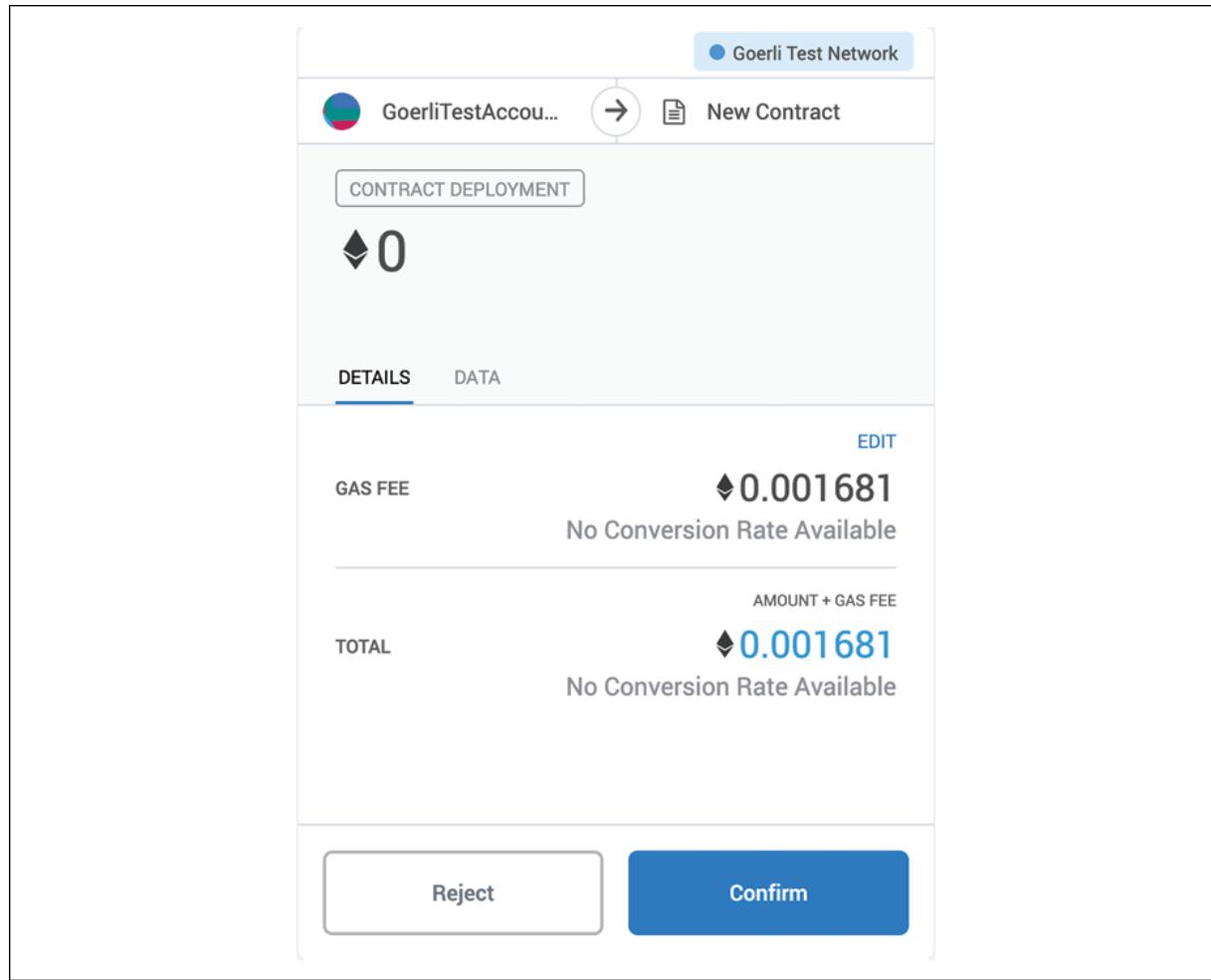


Figure 18.13: Confirm transaction

Click **Confirm**, and the contract will be deployed. We can see this in the history in MetaMask, as shown here:

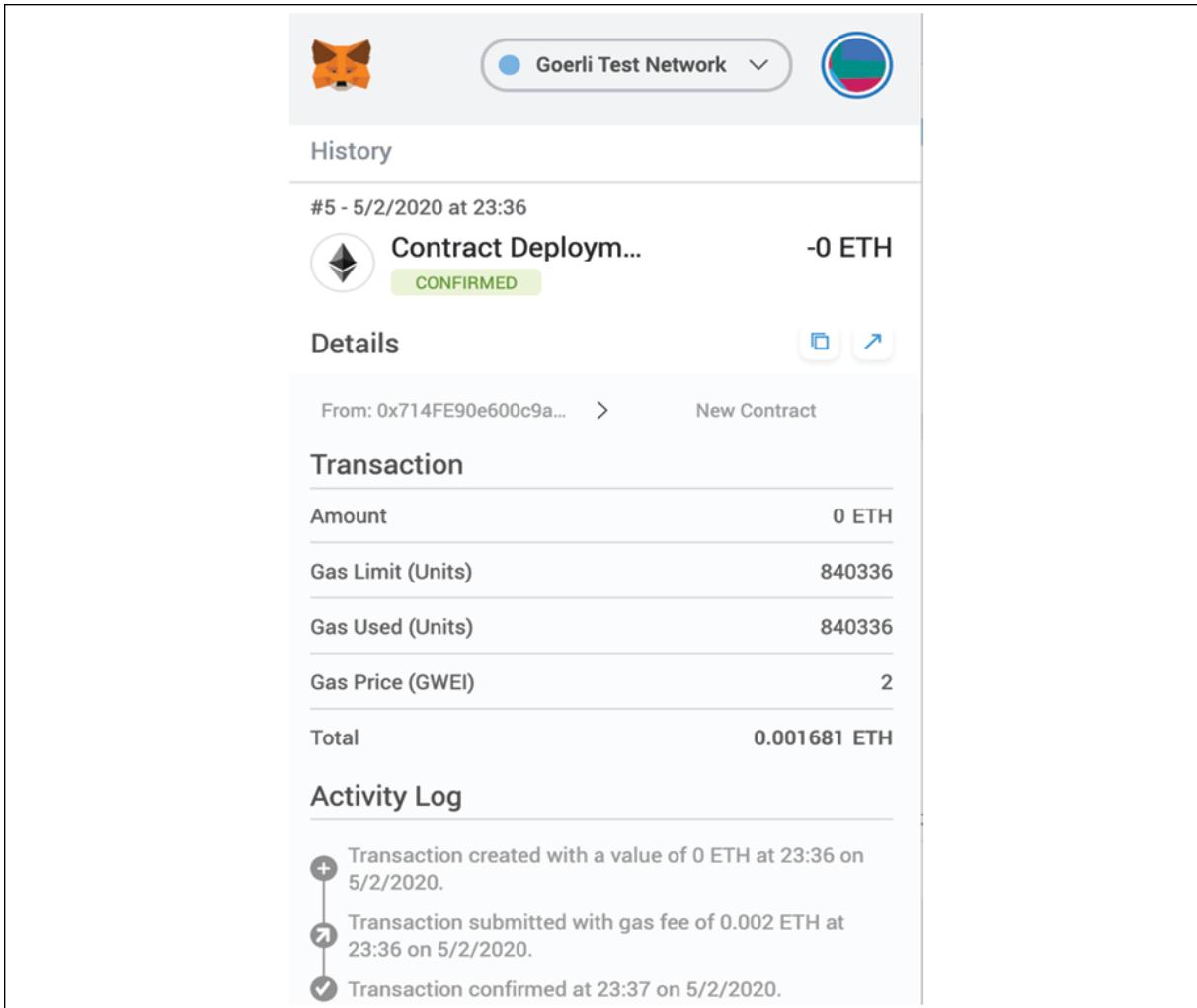


Figure 18.14: My ERC token contract deployment using MetaMask

Similar to the tests that we did earlier in this example when we deployed our contract on the JavaScript VM, we can invoke different functions exposed by our new ERC-20 token directly from the Remix IDE, as shown here:

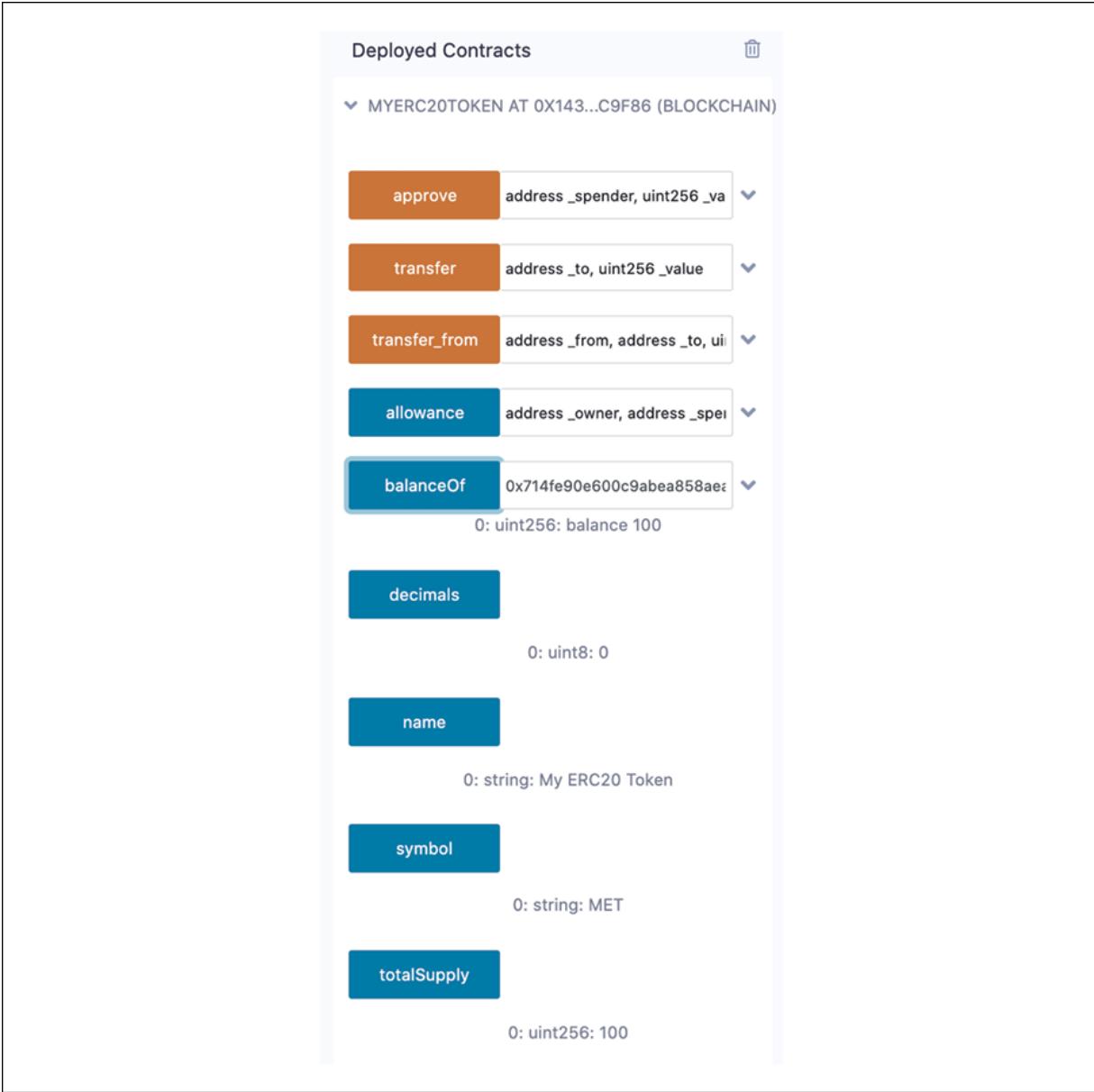


Figure 18.15: Deployed ERC-20 contract in Remix

We can also see our token on the Etherscan token tracker. This is shown by accessing the following link:

<https://goerli.etherscan.io/token/0x1437d9a9825d875cd516f287465f830c65ec9f86>

The Etherscan page can also be seen in the following screenshot:

The screenshot shows the Etherscan token view for the MET token. The token has a total supply of 100 MET, held by 1 address. A single transfer was made from the contract address to another address, amounting to 100 tokens.

Txn Hash	Age	From	To	Quantity
0x8325d0e9b13b2d...	5 mins ago	0x0000000000000000	0x714fe90e600c9ab...	100

Figure 18.16: Etherscan token view

So here we have it, our own MET token deployed on the Goerli test network.

Now, if desired, we could deploy this to the Ethereum mainnet, provided that we have some ether available. We can perform exactly the same steps to deploy it, with the only difference being that in MetaMask, we will choose the Ethereum mainnet as the network.

Adding tokens in MetaMask

Once we have deployed our contract and our ERC-20 token is now on the blockchain (so to speak), unless we are able to view it and perform operations on it, the token on its own is of no real use.

To perform operations on the token, we can manually create commands using the Web3.js `sendRawTransaction` method, and use the JavaScript and command-line `geth` console to interact with it. Alternatively, we can use an easier option and simply add the token to a wallet. Wallets abstract away the complexities associated with transaction creation and management and provide an easy-to-use interface to perform transfers and similar tasks.

MetaMask can serve as a wallet for tokens and provides an interface to add tokens. In this section, we'll see how we can add our MET token to MetaMask and perform some operations on it.

Open MetaMask and find the **Add Token** option, either on the main user interface, or open the account menu and it will display the current account, as shown here:

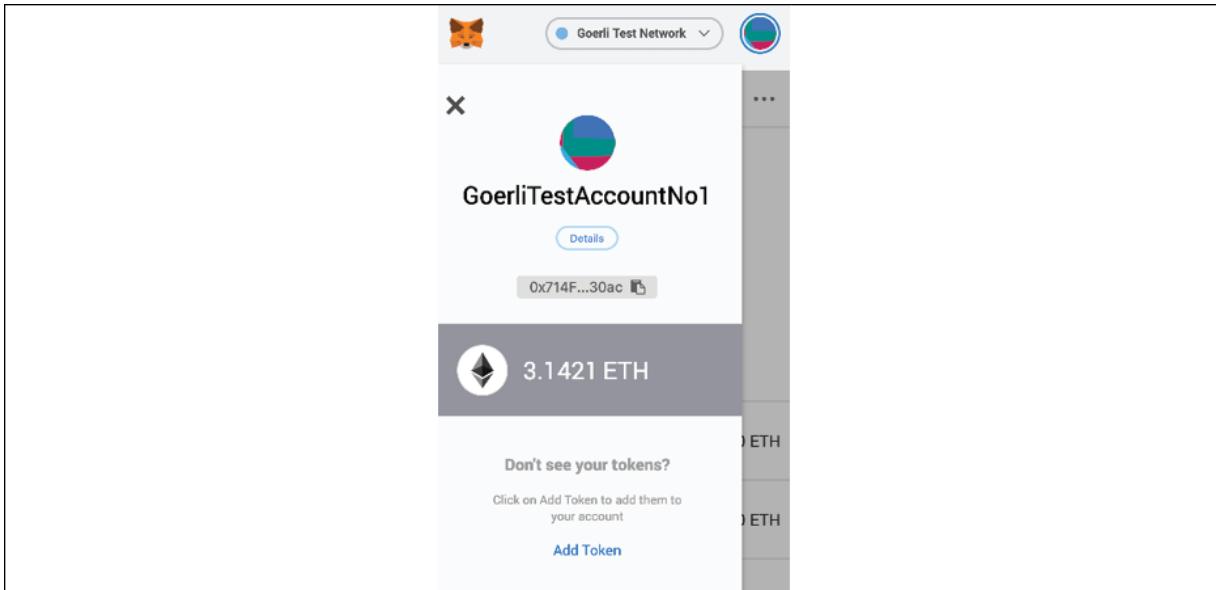


Figure 18.17: Adding a new token in Remix

Click on **Add Token** and select **Custom Token**.

Enter the contract address, `0x1437d9a9825d875cd516f287465f830c65ec9f86`, which will automatically display the **Token Symbol** and the decimal points, as shown in the following screenshot:

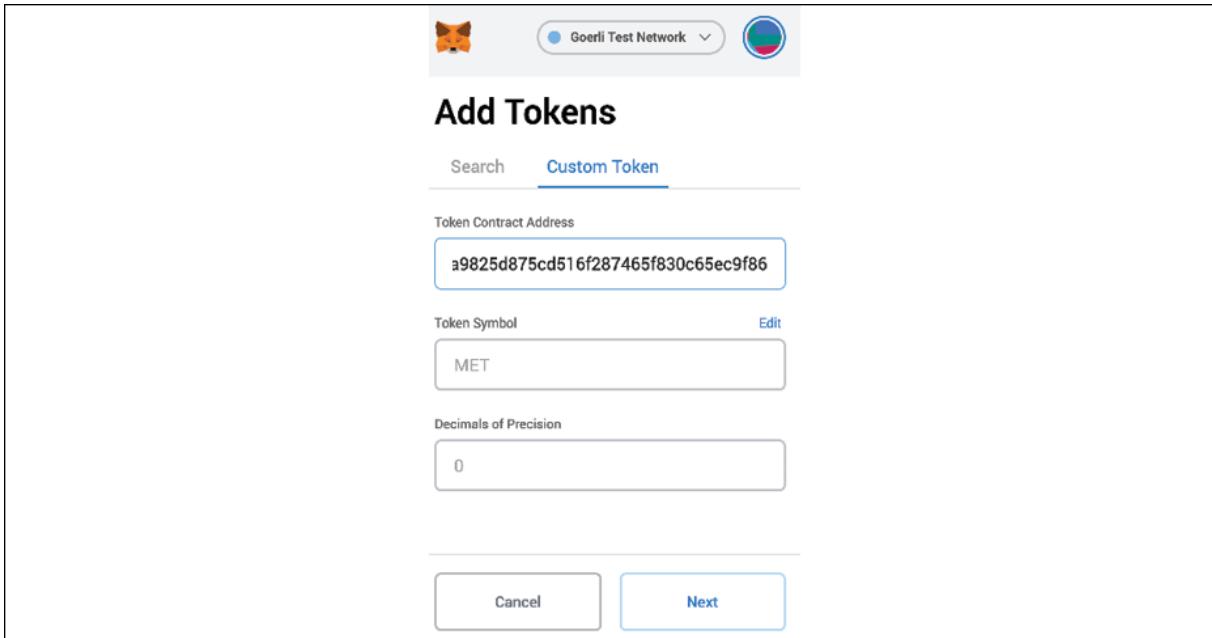


Figure 18.18: Custom token in MetaMask

Press **Next**, and then confirm that you want to add these tokens, as shown here:

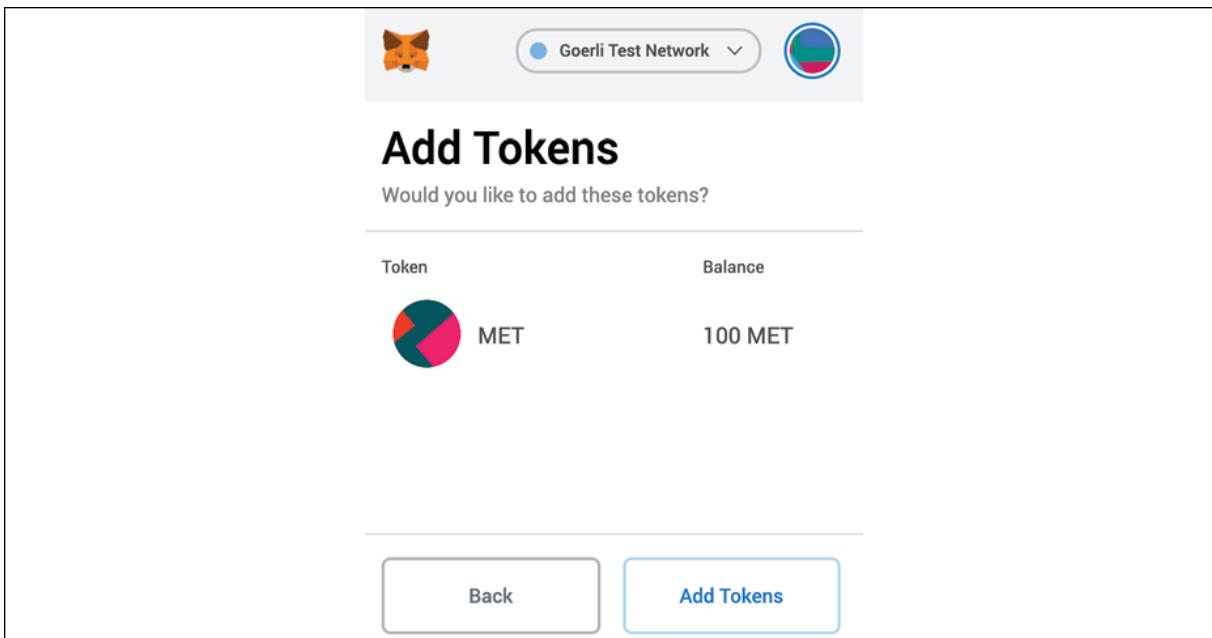


Figure 18.19: Adding a token in MetaMask

Now we have **100 MET** in our MetaMask wallet:

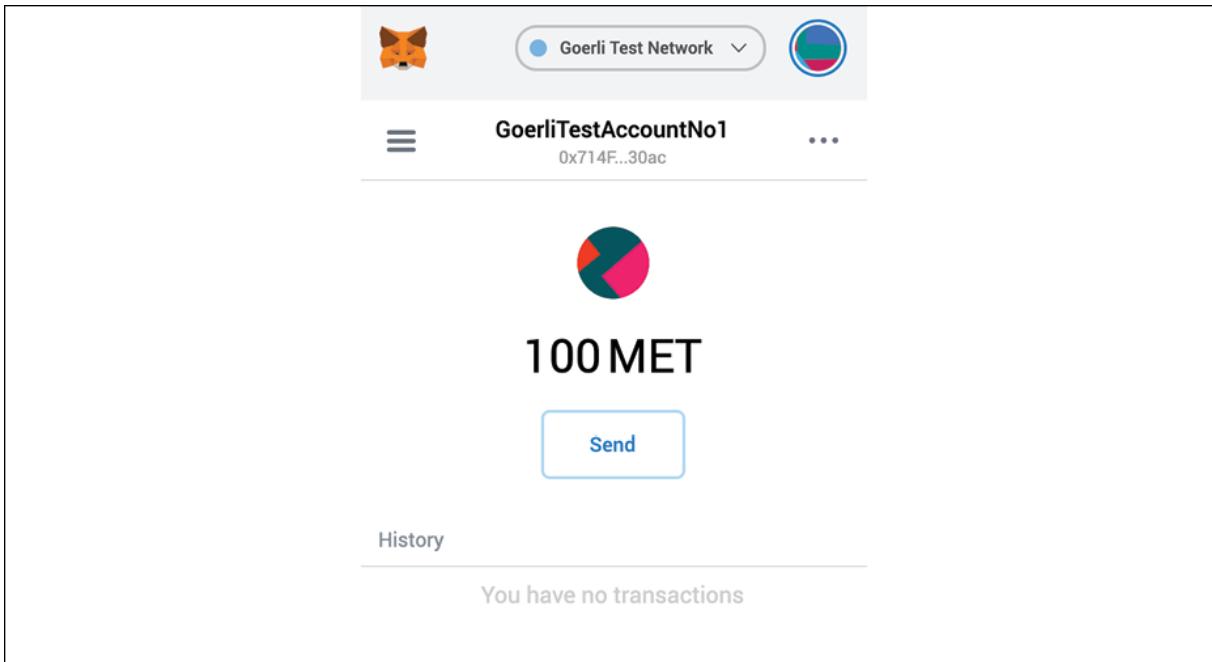


Figure 18.20: METs in MetaMask

Now let's send 10 MET to another account on the Goerli network. Click on **Send**, then enter the target account's details:

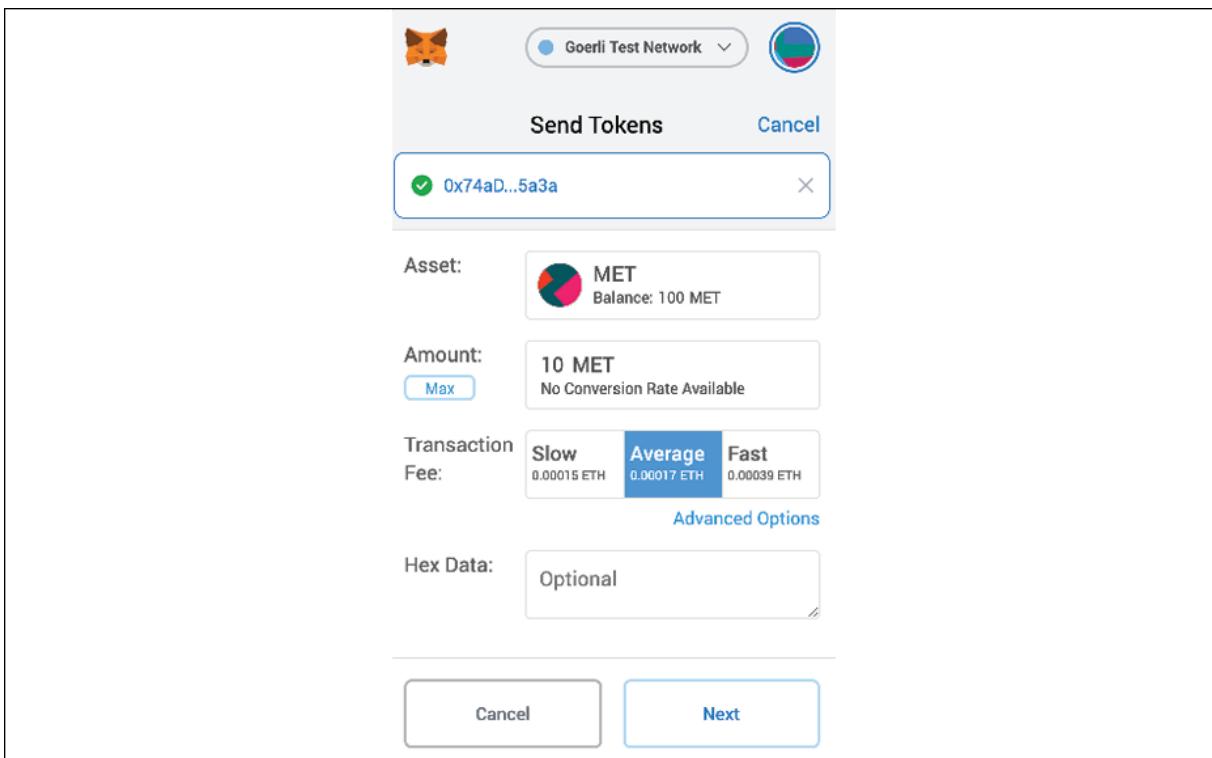


Figure 18.21: Send tokens

Enter **10 MET** and click **Next**:

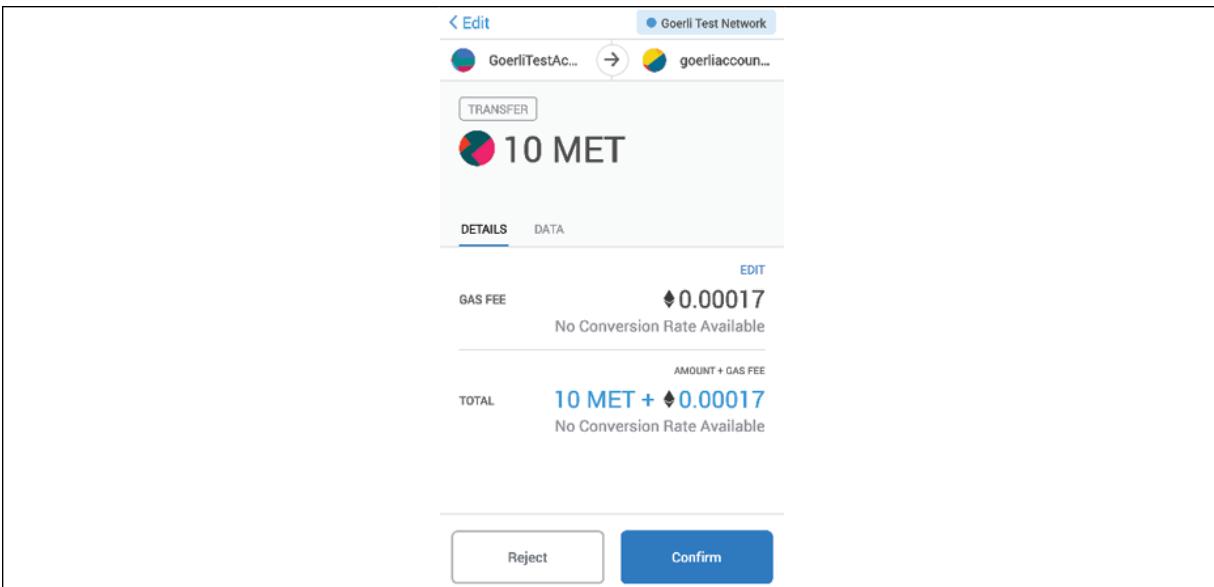


Figure 18.22: Remix confirmation

Click **Confirm**, and the transaction will be processed:

The screenshot shows a transaction history page from Etherscan. At the top, there is a logo of a fox, a dropdown menu set to 'Goerli Test Network', and a circular icon with a blue and red gradient. Below this, the word 'History' is displayed. A transaction entry is shown with the ID '#6 - 5/2/2020 at 23:57'. The transaction details are as follows:

Sent Tokens	-10 MET
CONFIRMED	
From: 0x714FE90e600c9a...	> To: 0x74aD931bCd3365b0...
Transaction	
Amount	0 ETH
Gas Limit (Units)	77341
Gas Used (Units)	51561
Gas Price (GWEI)	2.2
Total	0.000113 ETH
Activity Log	
+ 5/2/2020.	Transaction created with a value of 0 ETH at 23:57 on 5/2/2020.
23:58 on 5/2/2020.	Transaction submitted with gas fee of 170150.2 GWEI at 23:58 on 5/2/2020.
✓ 5/2/2020.	Transaction confirmed at 23:58 on 5/2/2020.

Figure 18.23: Tokens sent

We can see this transfer on Etherscan here:

<https://goerli.etherscan.io/tx/0xefeb5268c7faf0f9ccf2fe32213b279a69db728c0f1fbddccc0a0a8f2e5afad1ad>

This page is also shown in the following screenshot

The screenshot shows the Etherscan interface for a Goerli Testnet transaction. At the top, there's a header with the Etherscan logo, network selection (Goerli Testnet Network), search bar, and navigation links (All Filters, Home). Below the header, the title "Transaction Details" is displayed. A sponsored message from Gitcoin Virtual Hackathons is present. The main content area shows transaction details in a tabular format:

Overview	Event Logs (1)	State Changes
[This is a Goerli Testnet transaction only]		
⑦ Transaction Hash:	0xafeb5268c7faf0f9ccf2fe32213b279a69db728c0f1fbdcc0a0a8f2e5af1ad ⓘ	
⑦ Status:	Success ⓘ	
⑦ Block:	2628521	32 Block Confirmations
⑦ Timestamp:	8 mins ago (May-02-2020 10:58:46 PM +UTC)	
⑦ From:	0x714fe90e600c9ab... ⓘ	
⑦ To:	Contract 0x1437d9a9825d875cd516f287465f830c65ec9f86 ⓘ	
⑦ Tokens Transferred:	From 0x714fe90e600c9ab... To 0x74ad931bcd3365... For 10 ⓘ My ERC20 Tok... (MET)	
⑦ Value:	0 Ether (\$0.00)	
⑦ Transaction Fee:	0.0001134342 Ether (\$0.000000)	

At the bottom, there's a link "Click to see More" with a downward arrow.

Figure 18.24: Etherscan view

Now, coming back to the Remix IDE, we can now see that our balance has been reduced by 10 MET:

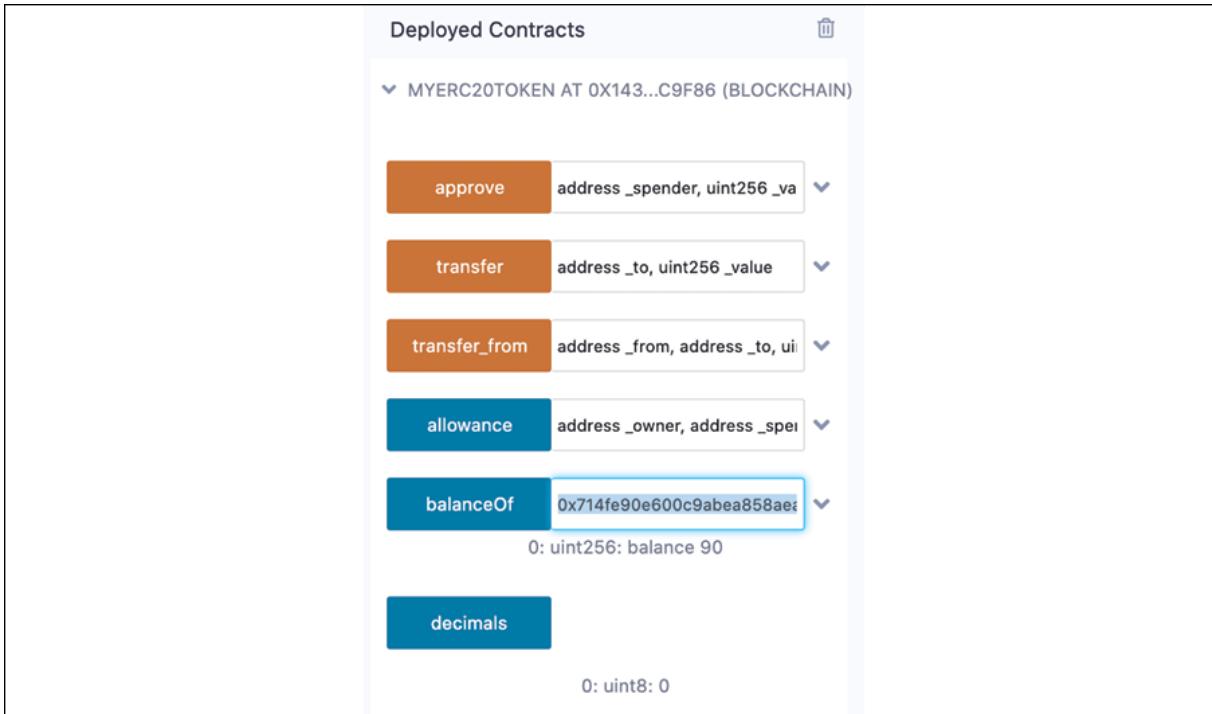


Figure 18.25: Deployed contracts in Remix

Similarly, we can check the balance of the target account to which we transferred 10 MET, which is now **10 MET**:

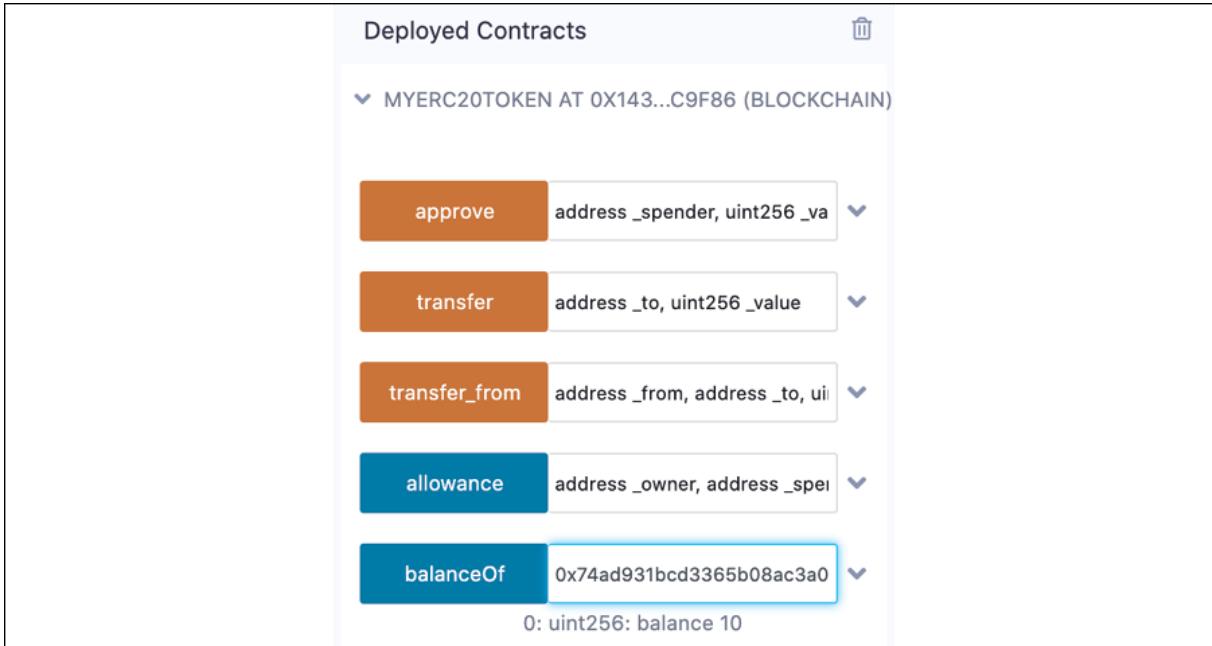


Figure 18.26: Balance of remix

We can see all the transfers of our ERC-20 token, MET, here on Etherscan:

<https://goerli.etherscan.io/token/0x1437d9a9825d875cd516f287465f830c65ec9f86>

This page can also be viewed in the following screenshot:

The screenshot shows the Etherscan interface for the Goerli Testnet Network. The top navigation bar includes 'All Filters', a search bar, and tabs for Home, Blockchain, Tokens, and Misc. A specific search term '0x1437d9a9825d875cd516f287465f830c65ec9f86' is visible in the search bar.

The main content area displays information about the 'Token My ERC20 Token'. It shows the total supply of 100 MET, held by 2 addresses, and 2 transfers. The 'Profile Summary' section shows the contract address as 0x1437d9a9825d875cd516f287465f830c65ec9f86 and decimals as 0.

The 'Transfers' tab is selected, showing a table of 2 transactions. The first transaction is from 0xefeb5268c7faf0f9... to 0x74ad931bcd3365..., quantity 10, occurring 9 mins ago. The second transaction is from 0x8325d0e9b13b2d... to 0x714fe90e600c9ab..., quantity 100, occurring 30 mins ago.

Figure 18.27: Etherscan view of token transfers

With this, we have covered how to create an ERC-20 token from scratch and deploy it on the Ethereum blockchain.

Let's now have a look at some of the novel concepts that are emerging due to the remarkable success of tokenization and the blockchain ecosystem in general.

Emerging concepts

With the advent of blockchain and tokenization, several new concepts have emerged over the last few years. We will introduce some of them now.

Tokenomics/token economics

Tokenomics or token economics is an emerging discipline that is concerned with the study of economic activity, economic models, and the impact of tokenization. It deals with the goods and assets that have been tokenized and the entities that are involved in the entire process of token issuance, sale, purchase, and investment.



You might have heard another term, cryptoeconomics, which is a related but slightly different term. Cryptoeconomics is concerned with the same topics but it is a superset of tokenomics. In other words, tokenomics is a subset of cryptoeconomics. Tokenomics is only concerned with tokens and tokenization ecosystems, but does not include the broader blockchain networks, protocols, and cryptocurrencies.

With the use of the **proof of work (PoW)** mechanism in Bitcoin, it was demonstrated for the first time that computer protocols can be designed in such a way that attacking a system does not result in achieving an uneven advantage or commercial benefit. This concept then further matured into what we call today cryptoeconomics. This can also be understood as a combination of economics, game theory, and cryptography.

Token engineering

Token engineering is an emerging concept that is looking at tokenization from an engineering perspective and is striving to apply the same rigor, systems thinking, and mathematical foundations to tokenization and blockchain in general that a usual engineering discipline has. This subject is still in its infancy; however, good progress has been made toward the development of this new discipline.



More information on token engineering can be found at
<https://tokenengineeringcommunity.github.io/website/>

Token taxonomy

There is a lack of consistent taxonomy for tokens. As such, there are no clear standards defined on how to design and manage tokens. Also, it is not clear how to reuse an already existing and working token design, or if any exist at all.

Therefore, there is a need for a classification system that categorizes all different types of tokens according to different attributes they have. There is also no universal classification of different attributes of tokens such as type, value, and economic attributes. Such a system would benefit the tokenization ecosystem tremendously.



Don't confuse this with the ERC standards we explained earlier; those are development standards and they are certainly useful. But they are limited in scope and are not the universal classification of tokens.

Some work on this was started by Interwork Alliance by producing a **Token Taxonomy Framework (TTF)**. More details on this work can be found on their website: <https://interwork.org>.

Summary

In this chapter, we covered tokenization and relevant concepts and standards. We also covered different types of tokens and related token standards. Moreover, a primer on trading and finance was also provided to familiarize readers with some standard finance concepts, which help us to understand the DeFi ecosystem too, as most DeFi terminology is borrowed from traditional finance.

Also, we introduced a practical example on how to create our own ERC-20-compliant token using the Ethereum platform. Finally, we introduced some emerging ideas related to tokenization.

In the next chapter, we will explore how blockchain can be used outside of the context of its original usage, that is, cryptocurrencies. We will cover various use cases, and consider in particular detail the use of blockchain in IoT.

Blockchain – Outside of Currencies

Digital currencies were the first-ever application of blockchain, which arguably didn't realize the technology's full potential. Although **Bitcoin**, the first major blockchain conceptualization, was introduced in 2008, it was not until years later that blockchain technology's possible applications in industries other than cryptocurrencies were realized. In 2010, discussion started regarding BitDNS, a decentralized naming system for domains on the internet. Then, **Namecoin**

(<https://en.bitcoinwiki.org/wiki/Namecoin>) started in April 2011 with a different vision to Bitcoin, the sole purpose of which was to provision electronic cash. This can be considered the first example of blockchain usage other than purely as a cryptocurrency.

Next, in 2013, the first **initial coin offering (ICO)**, **MasterCoin**, emerged, which paved the way for more ICOs. After that, Ethereum's ICO was hugely successful. With the availability of Ethereum in 2015, a general-purpose smart contract platform, more ideas started to emerge around various applications of blockchain technology. Since then, many use cases of blockchain technology in various industries have been proposed. In this chapter, five main industries have been selected, with the aid of use cases, for discussion:

- The Internet of Things
- Government
- Health
- Finance

- Media

Let's begin with one of the most exciting use cases of blockchain technology: the Internet of Things.

The Internet of Things

The **Internet of Things (IoT)** has recently gained much traction due to its potential for transforming business applications and everyday life. IoT can be defined as a network of computationally intelligent physical objects (any objects, such as cars, fridges, and industrial sensors) that are capable of connecting to the internet, sensing real-world events or environments, reacting to those events, collecting relevant data, and communicating this over the internet.

This simple definition has enormous implications and has led to exciting concepts, such as wearables (such as health trackers or watches), smart homes, smart grids, smart connected cars, and smart cities, which are all based on this basic concept of an IoT device. After dissecting the definition of IoT, four functions come to light as being performed by an IoT device: **sensing, reacting, collecting, and communicating**. All of these functions are performed by using various components on the IoT device.

Sensing is performed by sensors. Reacting or controlling is performed by actuators; collection is a function of various sensors, and communication is performed by chips that provide network connectivity. One thing to note is that all of these components are accessible and controllable over the internet, via the IoT. An IoT device on its own is useful to some extent, but if it is part of a broader IoT ecosystem, it is more valuable.

In the next section, we'll introduce the typical architecture of an IoT-based ecosystem.

Internet of Things architecture

A typical IoT can consist of many physical objects connected to each other, and to a centralized cloud server. This is shown in the following diagram:

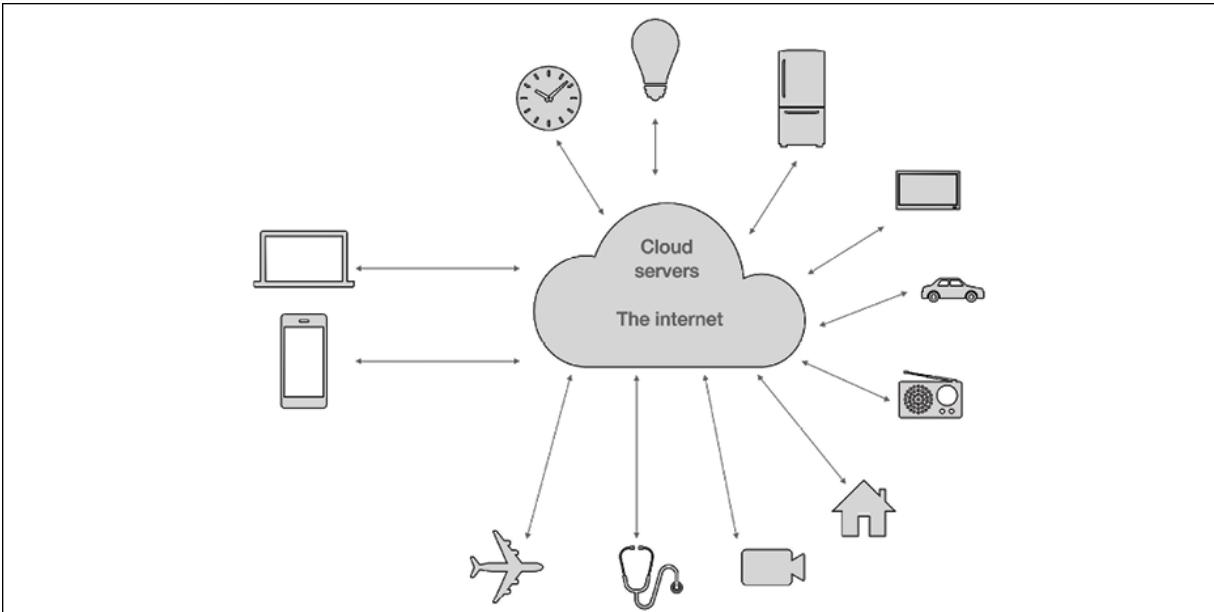
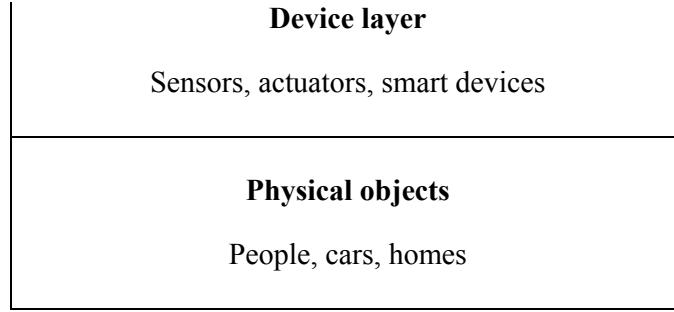


Figure 19.1: A typical IoT network

Elements of the IoT are spread across multiple layers, and various existing reference architectures can be used to develop IoT systems. A five-layer model can be used to describe IoT, which contains a **physical object** layer, **device** layer, **network** layer, **services** layer, and **application** layer. Each layer or level is responsible for various functions and includes multiple components. These are shown in the following table:

Application layer Transportation, financial, insurance, and many others
Management layer Data processing, analytics, security management
Network layer LAN, WAN, PAN, routers



Now, let's examine each layer in detail.

Physical object layer

This layer includes any real-world physical objects. It includes objects like cars, fridges, trains, and homes. In fact, anything that is required to be monitored and controlled can be connected to the IoT.

Device layer

This layer contains things that make up the IoT, such as sensors, transducers, actuators, smartphones, smart devices, and **Radio-Frequency Identification (RFID)** tags. There can be many categories of sensors, such as body sensors, home sensors, and environmental sensors, based on the type of work they perform. This layer is the core of an IoT ecosystem where various sensors are used to sense real-world environments. This layer includes sensors that can monitor temperature, humidity, liquid flow, chemicals, air, pressure, and much more. Usually, an **Analog to Digital Converter (ADC)** is required on a device to turn the real-world analog signal into a digital signal that a microprocessor can understand.

Actuators in this layer provide the means to enable control of external environments; for example, starting a motor or opening a door. These components also require digital to analog converters to convert a digital signal into analog. This method is especially relevant when control of a mechanical component is required by the IoT device.

Network layer

This layer is composed of various network devices that are used to provide internet connectivity between devices and to the cloud, or servers that are part of the IoT ecosystem. These devices can include gateways, routers, hubs, and switches. This layer can include two types of communication.

First, there is the horizontal means of communication, which includes radio, Bluetooth, Wi-Fi, Ethernet, LAN, Zigbee, and PAN, and can be used to provide communication between IoT devices. This layer can optionally be included in the device layer as it physically resides on the device layer where devices can communicate with each other.

Second, we have communication to the next layer, which is usually through the internet and provides communication between machines and people or other upper layers.

Management layer

This layer provides the management layer for the IoT ecosystem. This includes platforms that enable the processing of data gathered from the IoT devices and turning it into meaningful insights. Device management, security management, and data flow management are included in this layer. It also manages communication between the device and application layers.

Application layer

This layer includes applications running on top of the IoT network. This layer can consist of many applications, depending on the requirements, such as transportation, healthcare, financial, insurance, or supply chain management. This list, of course, is not an exhaustive list by any stretch of the imagination; there is a myriad of IoT applications that can fall into this layer.

With the availability of cheap sensors, hardware, and bandwidth, IoT has gained popularity in recent years and currently has applications in many different areas, including healthcare, insurance, supply chain management, home automation, industrial automation, and infrastructure management. Moreover, advancements in technology such as the availability of IPv6,

smaller and more powerful processors, and better internet access have also played a vital role in the popularity of IoT.



IPv6 is the latest version of the internet protocol, which, with a size of 128 bits, offers more address space compared to the 32-bit IPv4. More information on IPv6 is available here:
<https://en.wikipedia.org/wiki/IPv6>

In the next section, we'll discuss some advantages of IoT and blockchain convergence.

Benefits of IoT and blockchain convergence

The benefits of IoT with blockchain technology range from saving costs to enabling businesses to make vital decisions, and thus improve performance based on the data provided by the IoT devices. Even in domestic usage, IoT-equipped home appliances can provide valuable data for cost savings. For example, smart meters for energy monitoring can provide valuable information on how the energy is being used and can convey that back to the service provider. Raw data from millions of IoT devices is analyzed and provides meaningful insights that help in making timely and efficient business decisions.

The usual IoT model is based on a centralized paradigm, where IoT devices usually connect to a cloud infrastructure or central servers to report and process the relevant data back. This centralization poses certain possibilities of exploitation, including hacking and data theft. Moreover, not having control of personal data on a single, centralized service provider also increases the possibility of security and privacy issues. While there are methods and techniques to building a highly secure IoT ecosystem based on the normal IoT model, there are much more specific and desirable benefits that a blockchain-based model can bring to the IoT.

Blockchain for IoT can help to build trust, reduce costs, and accelerate transactions. Additionally, decentralization, which is at the very core of

blockchain technology, can eliminate single points of failure in an IoT network. For example, perhaps a central server might be unable to cope with the amount of data that billions of IoT devices (things) produce at high frequency, whereas a decentralized blockchain serving as a communication layer between these IoT devices enables all the IoT devices on the blockchain to maintain a copy of the data store of their own, which means that this architecture is inherently highly available and resilient.

Some IoT devices losing their database does not result in the entire network coming to a standstill, as might be the case with a centralized server, even with a disaster recovery solution. Also, the **peer-to-peer (P2P)** communication model provided by blockchain can help reduce costs because there is no need to build high-cost centralized data centers, nor implement complex public key infrastructure for security. Devices can communicate with each other directly or via routers.

As an estimate made by various researchers and companies, by 2020, there will be roughly 22 billion devices connected to the internet. With this explosion of billions of devices connecting to the internet, it is hard to imagine that centralized infrastructures will be able to cope with the high demands of bandwidth, services, and availability without incurring excessive expenditure. A blockchain-based IoT will be able to solve scalability, privacy, and reliability issues in the current IoT model.

Blockchain enables things to communicate and transact with each other directly, and with the availability of smart contracts, negotiation and financial transactions can also occur directly between the devices instead of requiring an intermediary, an authority, or human intervention. For example, if a room in a hotel is vacant, it can rent itself out, negotiate the rent, and can open the door lock for a human who has paid the required fee. Another example could be that if a washing machine runs out of detergent, it could order it online after finding the best price and value based on the logic programmed into its smart contract.

The aforementioned five-layer IoT model can be adapted to a blockchain-based model by adding a blockchain layer on top of the network layer. This layer will run smart contracts and provide security, privacy, integrity, autonomy, scalability, and decentralization services to the IoT ecosystem.

The management layer, in this case, can consist of only software related to analytics and processing, and security and control can be moved to the blockchain layer.

This model can be visualized in the following table:

Application layer Transportation, financial, insurance, and many others
Management layer Data processing, analytics, security management
Blockchain layer Security, consensus, P2P (M2M) autonomous transactions, decentralization, smart contracts
Network layer LAN, WAN, PAN, routers
Device layer Sensors, actuators, smart devices
Physical objects People, cars, homes

In this model, other layers are expected to remain the same, but an additional blockchain layer will be introduced as middleware between all participants of the IoT network.

It can also be visualized as a P2P IoT network after abstracting away all the layers mentioned earlier. This model is shown in the following diagram, where all the devices are communicating and negotiating with each other without a central command and control entity:

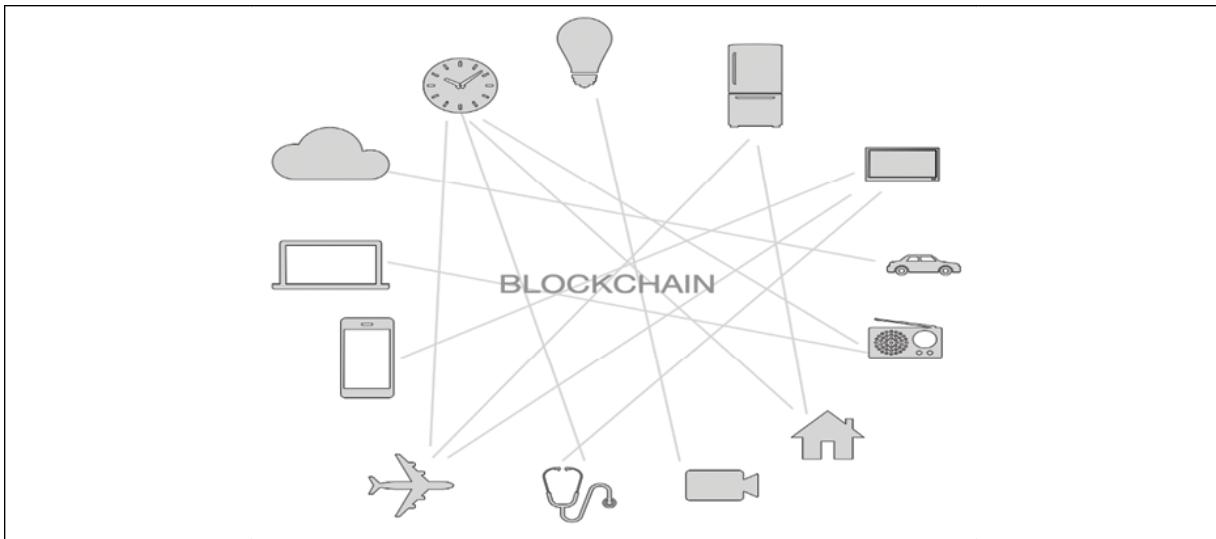


Figure 19.2: Blockchain-based direct (P2P, or machine-to-machine (M2M)) communication model

An IoT-based blockchain can also result in cost savings, due to the easier device management offered by a blockchain-based decentralized approach. The IoT network can also be optimized for performance by using blockchain. In this case, there will be no need to store IoT data centrally for millions of devices. This is because storage and processing requirements can be distributed to all IoT devices on the blockchain. This can result in completely removing the need for large data centers for processing and storing the IoT data.

A blockchain-based IoT can also thwart **denial of service (DoS)** attacks: hackers can target a centralized server or data center more efficiently, but with blockchain's distributed and decentralized nature, such attacks are no longer possible. Additionally, if, as estimated, there will soon be billions of devices connected to the internet, it will become almost impossible to manage security and updates for all those devices from traditional, centrally owned servers. Blockchain can provide a solution to this problem by allowing devices to communicate with each other directly in a secure manner, and even request firmware and security updates from each other. On a blockchain network, these communications can be recorded immutably and securely, which will provide auditability, integrity, and transparency to the system. This mechanism is not possible with traditional P2P systems.

In summary, there are clear benefits that can be reaped with the convergence of IoT and blockchain, and a lot of research and work in academia and the industry are already in progress. Practical use cases and platforms have emerged in the form of **Platform as a Service (PaaS)** for blockchain-based IoT, such as the IBM Watson IoT blockchain. There are various projects that have already been proposed that provide blockchain-based IoT solutions, such as IBM Blue Horizon and IBM Bluemix, two examples of IoT platforms that support blockchain IoT. Various start-ups, such as **Filament**, have already proposed novel ideas on how to build a decentralized network that enables devices on the IoT to transact with each other directly and autonomously, driven by smart contracts.

In the following section, a practical example will be provided on how to build a simple IoT device and connect it to the Ethereum blockchain. This IoT device is connected to the Ethereum blockchain and is used to open a door (in this case, the door lock is represented by an LED) when the appropriate amount of funds is sent by a user on the blockchain. This is a simple example and requires a more rigorously tested version to implement it in production. Nevertheless, it demonstrates how an IoT device can be connected to a blockchain. We will demonstrate how it can be controlled and responded to in response to certain events on an Ethereum blockchain.

Implementing blockchain-based IoT in practice

This example makes use of a **Raspberry Pi** device, which is a **Single Board Computer (SBC)**. The Raspberry Pi is an SBC developed as a low-cost computer to promote computer education, but it has also gained much more popularity as a tool of choice for building IoT platforms. A Raspberry Pi 3 Model B is shown in the following image.

You may be able to use earlier models too, but those have not been tested:

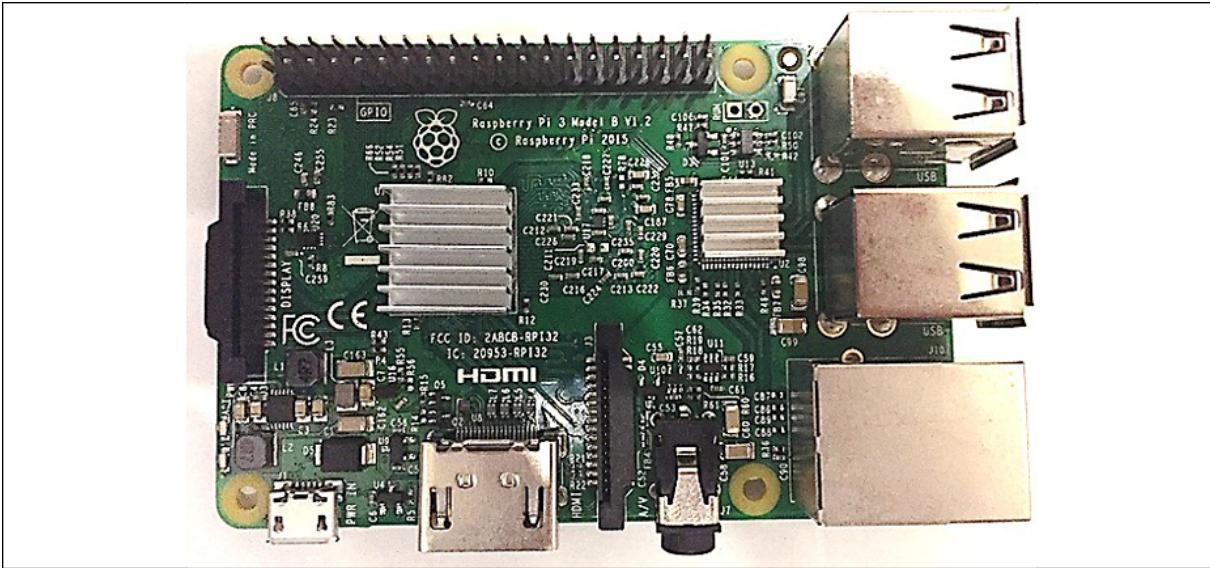


Figure 19.3: Raspberry Pi Model B

In the following section, an example will be discussed where a Raspberry Pi will be used as an IoT device connected to the Ethereum blockchain, and it will act in response to a smart contract invocation.

Setting up Raspberry Pi

First, the Raspberry Pi needs to be set up. This can be done by using NOOBS, which provides an easy method of installing Raspbian or any other operating system.

NOOBS is an abbreviation of **New Out Of Box Software**. It is a user-friendly and easy-to-use operating system installation manager for the Raspberry Pi.

NOOBS can be downloaded and installed from
<https://www.raspberrypi.org/downloads/noobs/>.

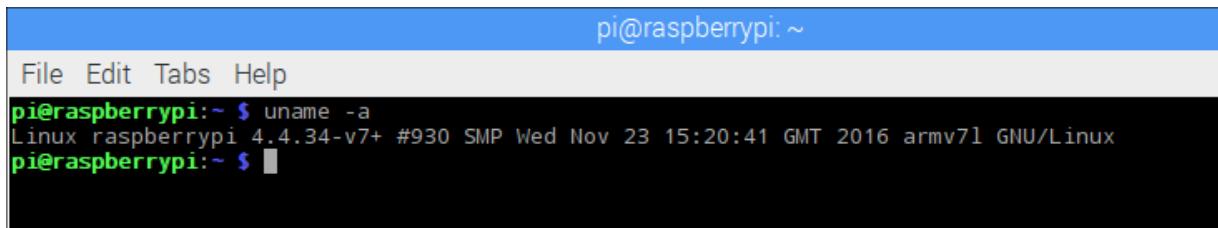


Alternatively, you can only install Raspbian, from
<https://www.raspberrypi.org/downloads/raspbian/>.

Another alternative, available at <https://github.com/debian-pi/raspbian-ua-netinst>, can also be used to install a minimal non-GUI version of Raspbian OS.

For this example, NOOBS has been used to install Raspbian. As such, the rest of the exercise assumes Raspbian is installed on the SD memory card of the Raspberry Pi. The command output in the following screenshot shows which architecture the operating system is running on. In this case, it is `armv7l`; therefore, the ARM-compatible binary for Geth will be downloaded.

The platform can be confirmed by running the command `uname -a` in a Terminal window in the Raspberry Pi Raspbian operating system:



```
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 4.4.34-v7+ #930 SMP Wed Nov 23 15:20:41 GMT 2016 armv7l GNU/Linux
pi@raspberrypi:~ $ █
```

Figure 19.4: Raspberry Pi architecture

Once the Raspbian operating system has been installed, the next step is to download the appropriate Geth binary for the Raspberry Pi ARM platform.

The download and installation steps are described in detail as follows:

1. First, download the Geth binary in Raspberry Pi. We use `wget` to download the `geth` client images:

```
$ wget https://gethstore.blob.core.windows.net/builds/geth-
```



Note that, in this example, a specific version of Geth is being downloaded. Other versions are available that can be downloaded from <https://geth.ethereum.org/downloads/>. However, it's recommended that you download the version that has been used in the examples in this chapter.

2. Unzip and extract this into a directory. The directory named `geth-linux-arm7-1.5.6-2a609af5` will be created automatically with the `tar` command, as shown here:

```
$ tar -zxvf geth-linux-arm7-1.5.6-2a609af5.tar
```

This command will create a directory named `geth-linux-arm7-1.5.6-2a609af5` and will extract the Geth binary and related files into that directory. The Geth binary can be copied into `/usr/bin` or the appropriate path on Raspbian to make it available from anywhere in the operating system. When the download is finished, the next step is to create the genesis block.

3. The same genesis block should be used that we created previously in *Chapter 13, Ethereum Development Environment*. The genesis file can be copied from the other node on the network. This is shown in the following code segment. Alternatively, an entirely new genesis block can be generated. Elements of the genesis file were explained in *Chapter 13, Ethereum Development Environment*, under the section entitled *The genesis file*, which you can refer to as a refresher:

4. Once the `genesis.json` file has been copied onto the Raspberry Pi, the following command can be run to generate the genesis block. It is important that the same genesis block is used that was generated previously; otherwise, the nodes will effectively be running on separate networks:

```
$ ./geth init genesis.json
```

This will show an output similar to the one shown in the following screenshot:

```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5 $ ./geth init genesis.json
I0110 23:37:15.714795 cmd/utils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:37:15.715283 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.715283 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:37:15.794723 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:37:15.923380 core/genesis.go:93] Genesis block already in chain. Writing canonical number
I0110 23:37:15.923895 cmd/geth/chancmd.go:131] successfully wrote genesis block and/or chain rule set: f2b2ffed01967a845a01d1dea21e5a
ec021e8e68b5ec9ffcc082df
```

Figure 19.5: Initializing the genesis file

5. After genesis block creation, we add peers to the network. This can be achieved by creating a list of nodes on our private network. In order to define the list, we create a file named `static-nodes.json` using any text editor, and add the `enode` of the peer that `geth`, on the Raspberry Pi, will connect to for synchronizing.
6. The `enode` URL can be obtained from the Geth JavaScript console by running the following command, which should be run on the peer to which Raspberry Pi is going to connect:

```
> admin.nodeInfo
```

This will show an output similar to the one shown in the following screenshot:

```
> admin.nodeInfo
{
  enode: "enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3
87375e932fb4885885f6452f6efa77f@[::]:30301",
  id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e9
4885885f6452f6efa77f",
```

Figure 19.6: Geth nodeInfo

7. The `static-nodes.json` file should contain the `enode` value, as shown in the following screenshot. We can view the contents using the `cat` command:

```
$ cat static-nodes.json
```

This command will produce the output shown here:

```
pi@raspberrypi:~/ethereum $ cat static-nodes.json
[
  "enode://44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc
57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@192.168.0.19:30301"
]
```

Figure 19.7: Static nodes configuration

After this step, further instructions presented in the following sections can be followed to connect Raspberry Pi to the other node on the private network. In this example, the Raspberry Pi will be connected to the network ID 786 that we created in *Chapter 13, Ethereum Development Environment*. The key is to use the same genesis file created previously and different port numbers. Using the same genesis file will ensure that clients connect to the same network in which the genesis file originated. Different ports are not a strict requirement; however, if the two nodes are running under a private network and access from an environment external to the network is required, then a combination of DMZ, router, and port forwarding will be used. Therefore, it is recommended to use different TCP ports to allow port forwarding to work correctly.

The `--identity` switch shown in the following section, *Setting up the first node*, which hasn't been introduced previously, allows for an identifying name to be specified for the node.

Setting up the first node

First, the `geth` client needs to be started on the first node using the following command:

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

This will give an output similar to the following:

```
imran@drequinox-OP7010:~$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --rpc --rp
capi web3,eth,debug,personal,net --rpcport 9001 --rpccorsdomain "*" --port 30301 --identity "drequinox"
"
I0110 23:26:46.032878 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/
.ethereum/privatenet/geth/chaindata
I0110 23:26:46.072986 ethdb/database.go:176] closed db:/home/imran/.ethereum/privatenet/geth/chaindata
I0110 23:26:46.073243 node/node.go:175] instance: Geth/drequinox/v1.5.2-stable-c8695209/linux/go1.7.3
I0110 23:26:46.073258 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/imran/
.ethereum/privatenet/geth/chaindata
I0110 23:26:46.082654 eth/backend.go:193] Protocol Versions: [63 62], Network Id: 786
I0110 23:26:46.083188 core/blockchain.go:214] Last header: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083203 core/blockchain.go:215] Last block: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083210 core/blockchain.go:216] Fast block: #7991 [999c534f...] TD=11652654509
I0110 23:26:46.083929 p2p/server.go:336] Starting Server
I0110 23:26:48.239776 p2p/discover/udp.go:217] Listening, enode://44352ede5b9e792e437c1c0431c1578ce367
6a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@[::]:3030
1
I0110 23:26:48.239893 p2p/server.go:604] Listening on [::]:30301
I0110 23:26:48.240913 node/node.go:340] IPC endpoint opened: /home/imran/.ethereum/privatenet/geth.ipc
I0110 23:26:48.241212 node/node.go:410] HTTP endpoint opened: http://localhost:9001
I0110 23:42:58.206205 eth/backend.go:479] Automatic pregeneration of ethash DAG ON (ethash dir: /home/
imran/.ethash)
I0110 23:42:58.206217 miner/miner.go:136] Starting mining operation (CPU=8 TOT=9)
```

Figure 19.8: Geth on the first node

Once this has been started up, it should be kept running, and another `geth` instance should be started from the Raspberry Pi node.

Setting up the Raspberry Pi node

On Raspberry Pi, the following command is required to be run to start `geth` and to sync it with other nodes (in this case, only one node). The following is the command:

```
$ ./geth --networkid 786 --maxpeers 5 --rpc --rpccapi web3,eth,de
```

This should produce an output similar to the one shown in the following screenshot. When the output contains a row displaying `Block synchronisation started`, this means that the node has connected successfully to its peer:

```
pi@raspberrypi:~/geth-linux-arm7-1.5.6-2a609af5$ ./geth --networkid 786 --maxpeers 5 --rpc --rpcapi web3,eth,debug,personal,net --rpccorsdomain "*" --port 30302 --identity "raspberry"
I0110 23:38:04.654374 cmd/utils/flags.go:612] WARNING: No etherbase set and no accounts found as default
I0110 23:38:04.654776 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.693111 ethdb/database.go:176] closed db:/home/pi/.ethereum/geth/chaindata
I0110 23:38:04.696937 node/node.go:176] instance: Geth/raspberry/v1.5.6-stable-2a609af5/linux/go1.7.4
I0110 23:38:04.697042 ethdb/database.go:83] Allotted 128MB cache and 1024 file handles to /home/pi/.ethereum/geth/chaindata
I0110 23:38:04.847835 eth/backend.go:191] Protocol Versions: [63 62], Network Id: 786
I0110 23:38:04.849753 eth/backend.go:219] Chain config: {ChainID: 0 Homestead: <nil> DAO: <nil> DAOSupport: false EIP150: <nil> EIP158: <nil>}
I0110 23:38:04.857847 core/blockchain.go:216] Last header: #2668 [6776ef24..] TD=708187563
I0110 23:38:04.858174 core/blockchain.go:217] Last block: #2668 [6776ef24..] TD=708187563
I0110 23:38:04.858349 core/blockchain.go:218] Fast block: #2668 [6776ef24..] TD=708187563
I0110 23:38:04.866705 p2p/server.go:340] Starting Server
I0110 23:38:10.223170 p2p/discover/udp.go:227] Listening, enode://98ba36ecea7ff011803d634da45752abd25101f20a62f23427afc3f280017bc134b195ac6ed59c3b01ca2a3f14638a52697a1bb1bf967fc842740886.15.44.209:30302
I0110 23:38:10.224031 p2p/server.go:608] Listening on [:]:30302
I0110 23:38:10.233788 node/node.go:341] IPC endpoint opened: /home/pi/.ethereum/geth.ipc
I0110 23:38:10.237027 node/node.go:411] HTTP endpoint opened: http://localhost:9002
I0110 23:38:20.225637 eth/downloader/downloader.go:326] Block synchronisation started
I0110 23:38:49.583631 core/blockchain.go:1067] imported 1 blocks, 0 txs ( 0.000 Mg) in 14.018s ( 0.000 Mg/s). #2669 [76077955
I0110 23:38:49.622191 core/blockchain.go:1067] imported 5 blocks, 0 txs ( 0.000 Mg) in 38.520ms ( 0.000 Mg/s). #2674 [76077955
```

Figure 19.9: Geth on the Raspberry Pi

This can be further verified by running commands in the `geth` console on both nodes, as shown in the following screenshot. The `geth` client can be attached by simply running the following command on the Raspberry Pi:

```
$ geth attach
```

This will open the JavaScript `geth` console for interacting with the `geth` node. We can use the `admin.peers` command to see the connected peers:

```
> admin.peers
[{
  caps: ["eth/62", "eth/63"],
  id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f",
  name: "Geth/dreqinnox/v1.5.2-stable-c8695209/linux/go1.7.3",
  network: {
    localAddress: "192.168.0.21:56550",
    remoteAddress: "192.168.0.19:30301"
  },
  protocols: {
    eth: {
      difficulty: 11719415397,
      head: "0x2d32c90b4c9dacea9a109b0ae52c1ebf511915bb618a2d3c55a80a63852e89f6",
      version: 63
    }
  }
}]
```

Figure 19.10: Geth console admin peers command running on the Raspberry Pi

Similarly, we can attach to the `geth` instance by running the following command on the first node:

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

Once the console is available, `admin.peers` can be run to reveal the details about other connected nodes, as shown in the following screenshot:

```
> admin.peers
[{
  caps: ["eth/62", "eth/63"],
  id: "98ba36ecea7ff011803d634da45752abd25101f20a62f23427afc3f280017bc134833dd5ba400bb195ac6ed59c3b01
ca2a3f14638a52697a1bb1bf967fc84274",
  name: "Geth/raspberry/v1.5.6-stable-2a609af5/linux/go1.7.4",
  network: {
    localAddress: "192.168.0.19:30301",
    remoteAddress: "192.168.0.21:56512"
  },
  protocols: {
    eth: {
      difficulty: 11700366137,
      head: "0x1188f58b4900a1d771d333141ea9400d78400bb8e561494ab436519ae64e1e34",
      version: 63
    }
  }
}]
```

Figure 19.11: Geth console admin peers command running on the other peer

Once both nodes are up and running, further prerequisites can be installed to set up the experiment. Installation of Node.js and the relevant JavaScript libraries is required.

Installing Node.js

The required libraries and dependencies are listed here. First, **Node.js** and **npm** need to be updated on the Raspberry Pi Raspbian operating system. For this, the following steps can be followed:

1. Install Node.js on the Raspberry Pi using the following command:

```
$ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E b
```



Note that we are using Node version `7.x` for this example, simply for demonstration. You can use a later version if desired.

This should display an output similar to the following. The output is quite large; therefore, only the top part of the output is shown in the following screenshot:

```
pi@raspberrypi:~/testled $ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
## Installing the NodeSource Node.js v7.x repo...
## Populating apt-get cache...
+ apt-get update
Get:1 http://archive.raspbian.org jessie InRelease [22.9 kB]
```

Figure 19.12: Node.js installation

2. Run the update via `apt-get`:

```
$ sudo apt-get install nodejs
```

Verification can be performed by running the following command to ensure that the correct versions of Node.js and `npm` are installed, as shown in the following screenshot:

```
pi@raspberrypi:~/testled $ npm -v
4.0.5
pi@raspberrypi:~/testled $ node -v
v7.4.0
pi@raspberrypi:~/testled $
```

Figure 19.13: npm and node installation verification

It should be noted that these versions are not a necessity; any of the latest versions of `npm` and Node.js should work. However, the examples in this chapter make use of `npm 4.0.5` and `node v7.4.0` simply for demonstration, so it is recommended that you use the same versions in order to avoid any compatibility issues.

3. Install Ethereum `web3` `npm` using the following command:

```
$ npm install web3@0.18.0
```

Web3 is required to enable JavaScript code to access the Ethereum blockchain.

 Make sure that the specific version of `web3` shown in the screenshot, or a version similar to this (for example, 0.20.2), is installed, instead of the default, version 1.2.11 (at the time of writing). As this example was originally developed and tested using version 0.18.0, it is recommended that `web3` 0.20.2 or 0.18.0 stable

version should be used for this example. The aim of this example, which is to show how Ethereum-based IoT networks can be created, doesn't change regardless of whether a newer or older version of Web3 is used. You can install the recommended version by using `$ npm install web3@0.20.2`.

The output of the command will be similar to the one shown here:

```
pi@raspberrypi:~/testled $ npm install web3
testled@1.0.0 /home/pi/testled
└── web3@0.18.0
    └── bignumber.js@2.0.7  (git+https://github.com/debris/bignumbers.git#94d7146671b9719e00a09c29b01a691bc85048c2)
npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

Figure 19.14: npm install web3

4. Similarly, `npm onoff` can be installed, which is required to communicate with the Raspberry Pi and control GPIO:

```
$ npm install onoff
```

This will result in the following output, indicating successful installation of `onoff`:

```
pi@raspberrypi:~/testled $ npm install onoff --save
testled@1.0.0 /home/pi/testled
└── onoff@1.1.1

npm WARN testled@1.0.0 No repository field.
pi@raspberrypi:~/testled $
```

Figure 19.15: Onoff installation

In this section, we set up a private network with our Raspberry Pi, and another node with `geth`, `nodejs`, `web3`, and the `onoff` library to run our example on.

When all the prerequisites are installed, hardware setup can be performed. For this purpose, a simple circuit is built using a breadboard and a few electronic components.

Hardware prerequisites

The hardware components are listed as follows:

- **LED:** A Light Emitting Diode (**LED**), can be used as a visual indication for an event.
- **Resistor:** A 330-ohm component is required, which provides resistance to the passing current based on its rating. It is not necessary to understand the theory behind it for this experiment; any standard electronics engineering text covers all these topics in detail.
- **Breadboard:** This provides a means of building an electronic circuit without requiring soldering.
- **T-Shaped Cobbler:** This is inserted on the breadboard, as shown in the following image, and provides a labeled view of all **General-Purpose I/O (GPIO)** pins for the Raspberry Pi.
- **Ribbon cable connector:** This is simply used to provide connectivity between the Raspberry Pi and the breadboard via GPIO.

All these components are shown in the following figure:

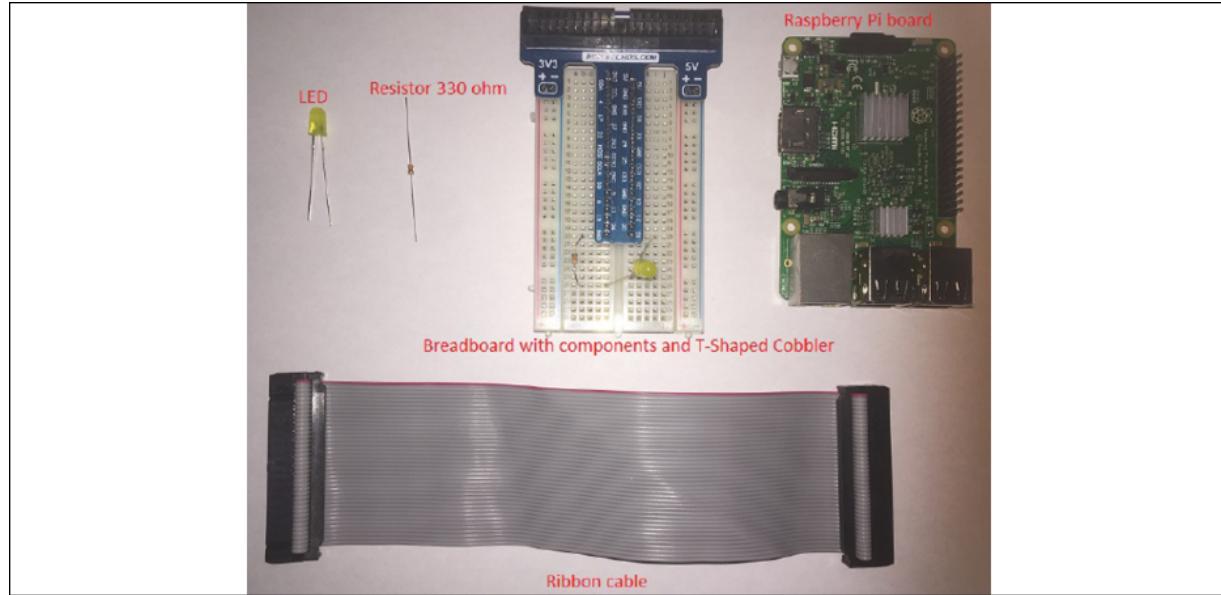


Figure 19.16: Required components

Building the electronic circuit

As shown in the following image, the **positive** leg (long leg) of the **LED** is connected to pin number **21** of the **Breadboard**, and the **negative** (short leg) is connected to the **Resistor**, which is then connected to the **ground (GND)** pin of the **Breadboard**. Once the connections have been set up, the **Ribbon cable** can be used to connect to the GPIO connector on the Raspberry Pi:

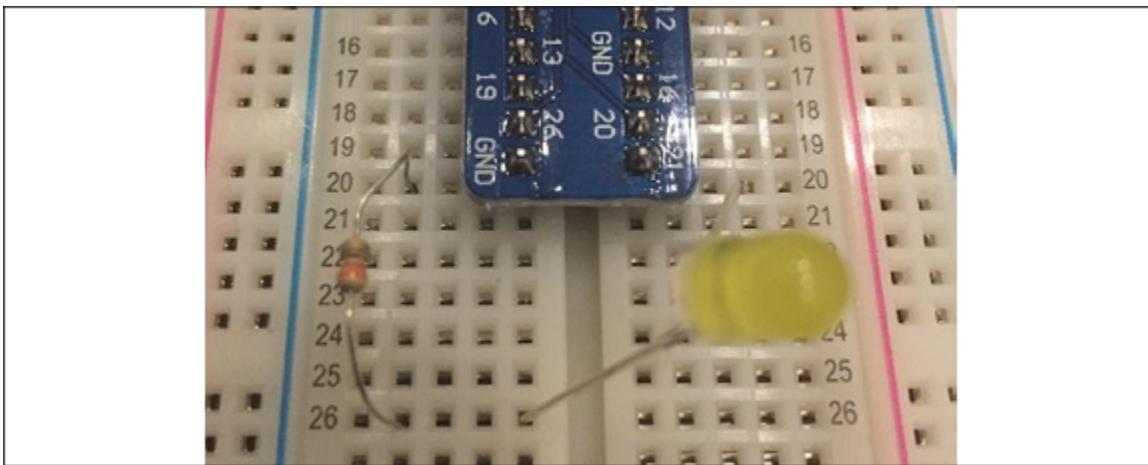


Figure 19.17: Connections for components on the breadboard

Once the connections have been set up correctly and the Raspberry Pi has been updated with the appropriate libraries and Geth, the next step is to develop a simple smart contract that expects a value. If the value provided is not what it expects, it does not trigger an event; otherwise, if the value passed matches the correct value, the event triggers, which can be read by the client JavaScript program running via Node.js.

Of course, the **Solidity** contract can be very complicated and can also deal with the **ether** sent to it. If the amount of ether is equal to the required amount, then the event can trigger. However, in this example, the aim is to demonstrate the usage of smart contracts to trigger events that can then be read by JavaScript running on Node.js, which then, in turn, can trigger actions on IoT devices using various libraries.

The smart contract source code is shown as follows:

```
pragma solidity ^0.4.0;
contract simpleIOT {
```

```
uint roomrent = 10;
event roomRented(bool returnValue);
function getRent (uint8 x) public returns (bool) {
    if (x == roomrent) {
        emit roomRented(true);
        return true;
    }
}
```

The online **Solidity** compiler (**Remix IDE**) can be used to run and test this contract. The **Application Binary Interface (ABI)** required for interacting with the contract is also available in the Remix IDE's solidity compiler section, as shown in the following screenshot:

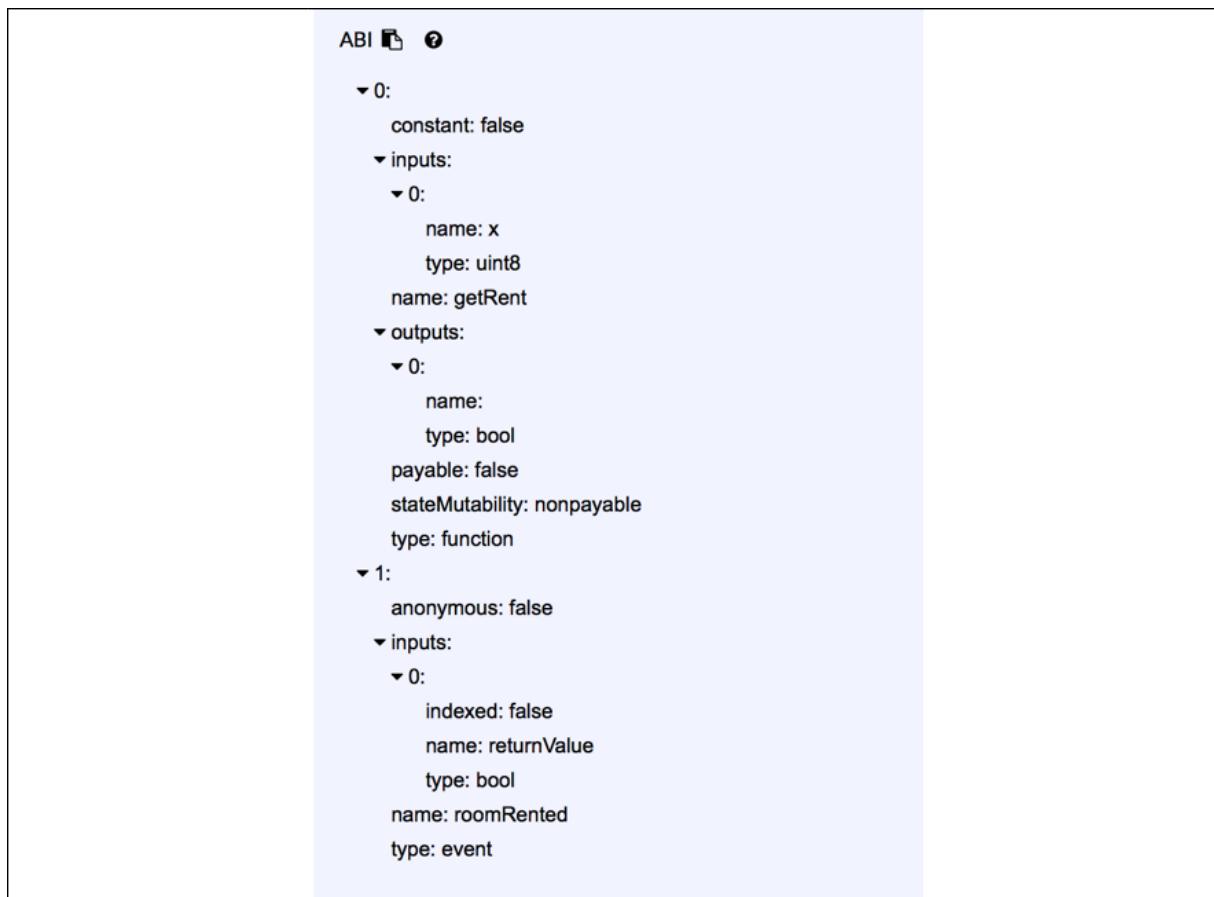


Figure 19.18: ABI from Remix IDE

The following is the ABI of the contract:

```
[
  {
    "constant": false,
    "inputs": [
      {
        "name": "x",
        "type": "uint8"
      }
    ],
    "name": "getRent",
    "outputs": [
      {
        "name": "",
        "type": "bool"
      }
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": false,
        "name": "returnValue",
        "type": "bool"
      }
    ],
    "name": "roomRented",
    "type": "event"
  }
]
```

There are two methods by which the Raspberry Pi node can connect to the private blockchain via the `web3` interface. The first is where the Raspberry Pi device is running its own `geth` client locally and maintains its ledger, as shown in the following diagram:

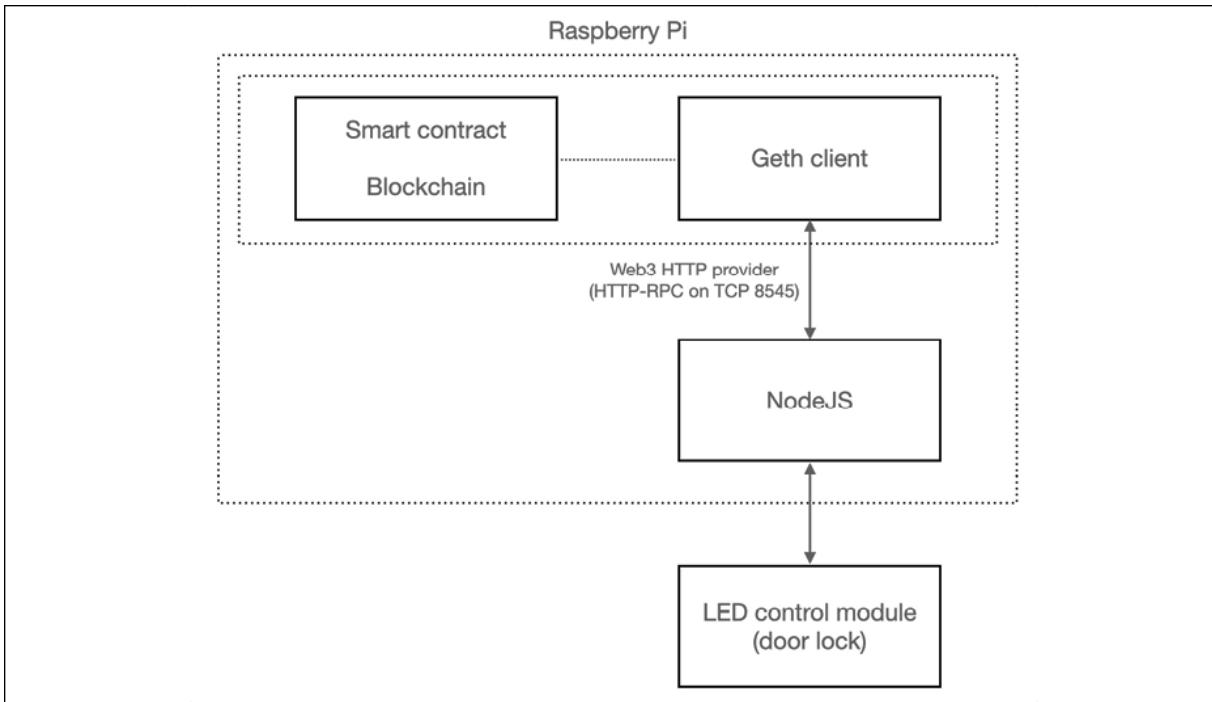


Figure 19.19: Application architecture of the room rent IoT application (IoT device with local ledger)

Sometimes, with resource-constrained devices, it is not possible to run a full `geth` node or even a light node. In that case, the second method, which uses a `web3` provider (via HTTP-RPC server), can be used to connect to the appropriate RPC channel. The following diagram shows the high-level architecture of an IoT application where the Raspberry Pi does not run a Geth instance. Instead, it runs only the minimum software required to provide IoT application functionality to the IoT device. For blockchain-relevant operations, it connects to another blockchain node running on the network via HTTP-RPC:

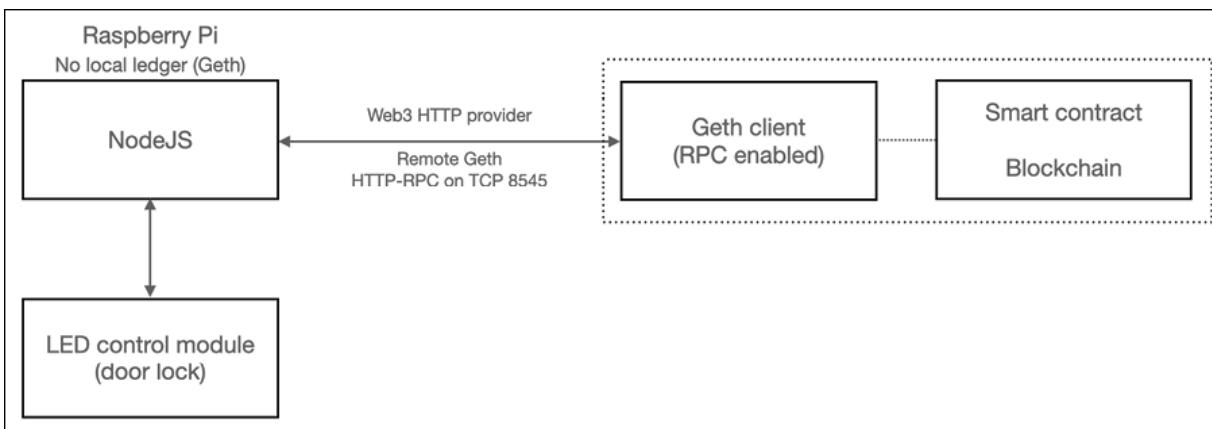


Figure 19.20: Application architecture of the room rent IoT application (IoT device without local ledger)

There are obvious security concerns that arise from exposing RPC interfaces publicly; therefore, it is recommended that this option is used only on private networks. If required to be used on public networks, appropriate security measures need to be put in place, such as allowing only the known IP addresses to connect to the `geth` RPC interface. This can be achieved by a combination of disabling peer discovery mechanisms and HTTP-RPC server listening interfaces.

More information about this can be found using `geth help`. The traditional network security measures such as **firewalls**, **Transport Layer Security (TLS)**, and **certificates** can also be used but have not been discussed in this example for brevity. Now, **Truffle** can be used to deploy the contract on the private network ID `786` to which, at this point, the Raspberry Pi is connected.

Truffle `deploy` can be performed simply by using the following command; it is assumed that `truffle init` and other preliminaries discussed in *Chapter 15, Introducing Web3*, have already been performed:

```
$ truffle migrate
```

This should produce an output similar to the following:

```
imran@drequinox-OP7010:~/iotcontract$ truffle migrate --reset
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0xdd8a88072aa4ff49b62c25d6f6f2207b731aee76
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying simpleIOT...
  simpleIOT: 0x151ce17c28b20ce554e0d944deb30e0447fbf78d
Saving successful migration to network...
Saving artifacts...
```

Figure 19.21: Truffle deploy

Once the contract has been deployed correctly, JavaScript code can be developed that will connect to the blockchain via `web3`, listen for the events from the smart contract in the blockchain, and turn the LED on via the Raspberry Pi. The JavaScript code required for this example is shown here. Simply write or copy and paste the code shown here into a file named

`index.js`:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
} else
{
    web3 = new Web3(new Web3.providers.HttpProvider("http://locat
//http-rpc-port
})
var Gpio = require('onoff').Gpio;
var led = new Gpio(21, 'out');
var coinbase = web3.eth.coinbase;
var ABIString = '[{"constant":false,"inputs":[{"name":"x","type"
var ABI = JSON.parse(ABIString);
var ContractAddress = '0x975881c44fbef4573fef33cccec1777a8f76669
web3.eth.defaultAccount = web3.eth.accounts[0];
var simpleiot = web3.eth.contract(ABI).at(ContractAddress);
var event = simpleiot.roomRented( {}, function(error, result) {
{
    console.log("LED On");
    led.writeSync(1);
}
});
```



Note that, in the preceding code example, the contract address '`0x975881c44fbef4573fef33cccec1777a8f76669c`' for the variable `var ContractAddress` is specific to the deployment; it will be different when you run this example. Simply change the address in the file to what you see after deploying the contract.

Also, note the HTTP-RPC server listening port on which Geth has been started on Raspberry Pi. By default, it is TCP port `8545`. Remember to change this according to your Raspberry Pi setup and Geth configuration. It

is set to `9002` in the preceding example code because Geth running on Raspberry Pi is listening on `9002` in the example. If it's listening on a different port on your Raspberry Pi, then change it to that port:

```
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:9002"))
```

When Geth starts up, it shows which port it has HTTP endpoint listening on. This is also configurable with `--rpcport` in `geth` by specifying the port number value as a parameter to the flag.

This JavaScript code can be placed in a file on the Raspberry Pi; for example, `index.js`. It can be run by using the following command:

```
$ node index.js
```

This will start the program, which will run on Node.js and listen for events from the smart contract. Once the program is running correctly, the smart contract can be invoked by using the Truffle console, as shown in the following screenshot.

In this case, the `getRent` function is called with the parameter `10`, which is the expected value:

```
[truffle(development)> simpleiot.getRent(10)
'0x71f550949a4c5168af7b9f7f84fada99bccc20a123779642e5e8c0c012726ee'
```

Figure 19.22: `getRent` function invocation

After the contract has been mined, `roomRented` will be triggered, which will turn on the LED.

In this example, it is a simple LED, but it can be any physical device, such as a room lock, that can be controlled via an actuator. If all works well, the LED will be turned on as a result of the smart contract's function invocation, as shown in the following image:

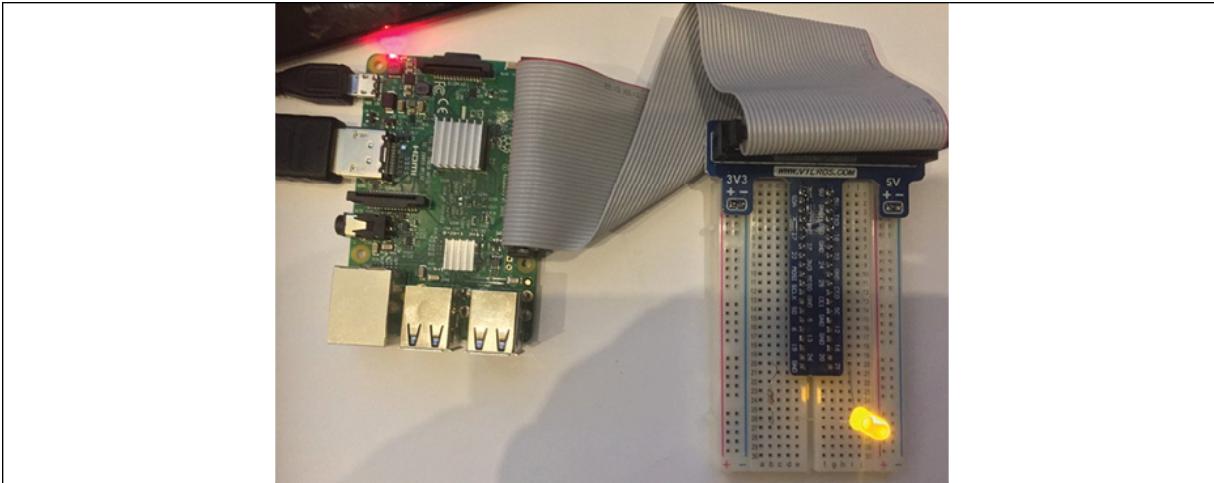


Figure 19.23: Raspberry Pi with LED control

Also, on the node side, it will display an output similar to the one shown here:

```
$ node index.js  
LED On
```

As demonstrated in the preceding example, a private network of IoT devices can be built that runs a `geth` client on each of the nodes, listens for events from smart contracts, and triggers an action accordingly. The example shown is simple by design but demonstrates the underlying principles of an Ethereum network that can be built using IoT devices, along with smart contract-driven control of the physical devices.

In the next few sections, other applications of the blockchain will be discussed, such as in government, finance, and health.

Government

There are various applications of blockchain being researched currently that can support government functions and take the current model of e-government to the next level. First, in this section, some background for e-

government will be provided, and then a few use cases such as e-voting, homeland security (border control), and electronic IDs (citizen ID cards) will be discussed.

Government, or electronic government, is a paradigm where information and communication technology are used to deliver public services to citizens. The concept is not new and has been implemented in various countries around the world, but with blockchain, a new avenue of exploration has opened up. Many governments are researching the possibility of using blockchain technology for managing and delivering public services, including, but not limited to, identity cards, driving licenses, secure data sharing among various government departments, and contract management. Transparency, auditability, and integrity are attributes of blockchain that can go a long way in effectively managing various government functions.

Border control

Automated border control systems have been in use for decades now to thwart illegal entry into countries and prevent terrorism and human trafficking.

Machine-readable travel documents, specifically **biometric passports**, have paved the way for automated border control; however, current systems are limited to a certain extent and blockchain technology can provide solutions. A **machine-readable travel document (MRTD)** standard is defined in document ICAO 9303

(<https://www.icao.int/publications/pages/publication.aspx?docnum=9303>) by the **International Civil Aviation Organization (ICAO)** and has been implemented by many countries around the world.

Each passport contains various security and identity attributes that can be used to identify the owner of the passport, and also circumvent attempts at tampering with these passports. These include biometric features such as retina scan, fingerprints, facial recognition, and standard ICAO specified

features, including **machine-readable zone (MRZ)** and other text attributes that are visible on the first page of the passport.

One key issue with current border control systems is data sharing, whereby the systems are controlled by a single entity and data is not readily shared among law enforcement agencies. This lack of ability to share data makes it challenging to track suspected travel documents or individuals. Another issue is related to the immediate implementation of blacklisting of a travel document; for example, when there is an immediate need to track and control suspected travel documents. Currently, there is no mechanism available to blacklist or revoke a suspicious passport immediately and broadcast it to the border control ports worldwide.

Blockchain technology can provide a solution to this problem by maintaining a blacklist in a smart contract that can be updated as required. Any changes will be immediately visible to all agencies and border control points, thus enabling immediate control over the movement of a suspected travel document. It could be argued that traditional mechanisms like **Public Key Infrastructures (PKIs)** and P2P networks can also be used for this purpose, but they do not provide the benefits that a blockchain can provide. With blockchain, the whole system can be simplified without the requirement of complex networks and PKI setups, which will also result in cost reduction. Moreover, blockchain-based systems will provide cryptographically guaranteed immutability, which helps with auditing and discourages any fraudulent activity.

The full database of all travel documents may not be stored on the blockchain currently due to inherent storage limitations, but a backend distributed database such as **BigchainDB**, **IPFS**, or **Swarm** can be used for that purpose. In this case, a hash of the travel document with the biometric ID of an individual can be stored in a simple smart contract, and a hash of the document can then be used to refer to the detailed data available on the distributed filesystem, such as IPFS. This way, when a travel document is blacklisted anywhere on the network, that information will be available immediately with the cryptographic guarantee of its authenticity and integrity throughout the distributed ledger. This functionality can also

provide adequate support in anti-terrorism activities, thus playing a vital role in the homeland security function of a government.

A simple contract in **Solidity** can have an array defined for storing identities and associated biometric records. This array can be used to store the identifying information about a passport. The identity can be a hash of MRZ of the passport or travel document concatenated with the biometric record from the RFID chip. A simple Boolean field can be used to identify blacklisted passports. Once this initial check passes, further detailed biometric verification can be performed by traditional systems. Eventually, when a decision is made regarding the entry of the passport holder, that decision can be propagated back to the blockchain, thus enabling all participants on the network to immediately share the outcome of the decision.

A high-level approach to building a blockchain-based border control system can be visualized as shown in the following diagram. In this scenario, the passport is presented for scanning to an RFID and page scanner, which reads the data page and extracts machine-readable information, along with a hash of the biometric data stored in the RFID chip. At this stage, a live photo and retina scan of the passport holder is also taken. This information is then passed on to the blockchain, where a smart contract is responsible for verifying the legitimacy of the travel document by first checking its list of blacklisted passports, and then requesting more data from the backend IPFS database for comparison. Note that the biometric data, such as a photo or retina scan, is not stored on the blockchain; instead, only a reference to this data in the backend (IPFS or BigchainDB) is stored in the blockchain.

If the data from the presented passport matches with what is held in the IPFS as files or in BigchainDB and also passes the smart contract logical check, then the border gate can be opened:

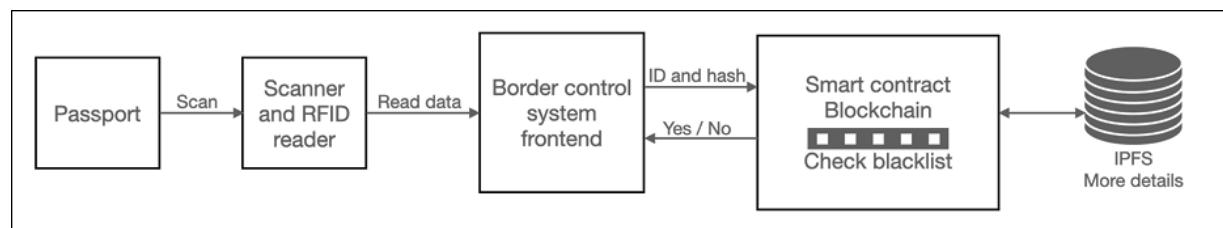


Figure 19.24: Automated border control using blockchain

After verification, this information is propagated throughout the blockchain and is instantly available to all participants on the border control blockchain. These participants can be a worldwide consortium of homeland security departments of various nations.

Voting

Voting in any government is a key function and allows citizens to participate in the democratic election process. While voting has evolved into a much more mature and secure process, it still has limitations that need to be addressed to achieve a desired level of maturity. Usually, the limitations in current voting systems revolve around fraud, weaknesses in operational processes, and especially transparency. Over the years, secure voting mechanisms (machines) have been built that make use of specialized voting machines that promised security and privacy, but they still have vulnerabilities that could be exploited to subvert the security mechanisms of those machines. These vulnerabilities can lead to serious implications for the whole voting process and can result in mistrust in the government by the public.

Blockchain-based voting systems can resolve these issues by introducing end-to-end security and transparency in the process. Security is provided in the form of integrity and authenticity of votes by using public key cryptography, which comes as standard in a blockchain. Moreover, immutability guaranteed by blockchain ensures that votes cast once cannot be cast again. This can be achieved through a combination of biometric features and a smart contract maintaining a list of votes already cast. For example, a smart contract can maintain a list of already-cast votes with the biometric ID (for example, a fingerprint) and can use that to detect and prevent double casting. Secondly, **zero-knowledge proofs (ZKPs)** can also be used on the blockchain to protect voters' privacy. With ZKP, a voter can remain anonymous by hiding their identities, and the vote itself can be kept confidential.



Recently, presidential elections were held in Sierra Leone using blockchain technology, making it the first country to use blockchain technology for elections: <https://www.coindesk.com/sierra-leone-secretly-holds-first-blockchain-powered-presidential-vote/>.

Another example can be seen here:

<https://www.coindesk.com/swiss-city-plans-to-vote-on-blockchain-using-ethereum-digital-id.>

Citizen identification (ID cards)

Electronic IDs or national ID cards are issued by various countries around the world at present. These cards are secure and possess many security features that thwart duplication or tampering attempts. However, with the advent of blockchain technology, several improvements can be made to this process.

Digital identity is not only limited to just government-issued ID cards; it is a concept that applies to online social networks and forums, too. There can be multiple identities being used for different purposes. A blockchain-based online digital identity allows control over personal information sharing. Users can see who used their data and for what purpose, as well as control access to it. This is not possible with the current infrastructures, which are centrally controlled.

The key benefit is that a single identity issued by the government can be used easily, and in a transparent manner, for multiple services via a single government blockchain. In this case, the blockchain serves as a platform where a government is providing various services such as pensions, taxation, or benefits and a single ID is being used to access all these services. Blockchain, in this case, provides a permanent record of every change and transaction made by a digital ID, thus ensuring integrity and transparency of the system. Also, citizens can notarize birth certificates, marriages, deeds, and many other documents on the blockchain tied with their digital ID as a proof of existence.

Currently, there are successful implementations of identity schemes in various countries that work well, and there is an argument that perhaps blockchain is not required in identity management systems. Although there are several benefits, such as privacy and controlling the use of identity information, due to the current immaturity of blockchain technology, perhaps it is not ready for use in real-world identity systems. However, research is being carried out by various governments to explore the use of blockchain for identity management.

Moreover, laws such as the right to be forgotten can be quite difficult to incorporate into blockchain due to their immutable nature.

Other government functions where blockchain technology can be implemented to improve cost and efficiency include the collection of taxes, benefits management and disbursement, land ownership record management, life event registration (marriages, births), motor vehicle registration, and licenses. This is not an exhaustive list and, over time, many functions and processes of a government can be adapted to a blockchain-based model. The key benefits of blockchain, such as immutability, transparency, and decentralization, can help to bring improvements to most of the traditional government systems.

Now, let's see how the health industry can benefit from blockchain technology.

Health

The health industry has also been identified as another major industry that can benefit by adapting blockchain technology. Blockchain can provide an immutable, auditable, and transparent system that traditional P2P networks cannot. Also, blockchain provides a simpler, more cost-effective infrastructure compared to traditional complex PKI networks. In healthcare, major issues such as privacy compromises, data breaches, high costs, and fraud can arise from a lack of interoperability, overly complex processes, transparency, auditability, and control. Another burning issue is counterfeit

medicines; especially in developing countries, this is a major cause of concern.

With the adaptability of blockchain in the health sector, several benefits can be realized, including cost savings, increased trust, the faster processing of claims, high availability, no operational errors due to complexity in the operational procedures, and preventing the distribution of counterfeit medicines.

From another angle, blockchains that are providing a digital currency as an incentive for mining can be used to provide processing power to solve scientific problems. This helps to find cures for certain diseases. Examples include **FoldingCoin**, which rewards its miners with FLDC tokens for sharing their computer's processing power for solving scientific problems that require unusually large calculations.



FoldingCoin is available at <http://foldingcoin.net/>.

Another similar project is called **CureCoin**, which is available at <https://www.curecoin.net/>. It is yet to be seen how successful these projects will be in achieving their goals, but the idea is very promising.

In the next section, we'll explore one of the most talked about and anticipated industries that can benefit from blockchain technology: finance.

Finance

Blockchain has many potential applications in the finance industry. Blockchain in finance is currently the hottest topic in the industry, and major banks and financial organizations are researching to find ways to adopt blockchain technology, primarily due to its highly desired potential to cost-save.

These applications include, but are not limited to, insurance, post-trade settlements, financial crime prevention, and payments.

Insurance

In the insurance industry, blockchain technology can help to stop fraudulent claims, increase the speed of claim processing, and enable transparency. Imagine a shared ledger between all insurers that can provide a quick and efficient mechanism for handling intercompany claims. Also, with the convergence of IoT and blockchain, an ecosystem of smart devices can be imagined, where all these things can negotiate and manage their insurance policies, which are controlled by smart contracts on the blockchain.

Blockchain can reduce the overall cost and effort required to process claims. Claims can be automatically verified and paid via smart contracts and the associated identity of the insurance policyholder. For example, a smart contract, with the help of an **oracle** and possibly IoT, can make sure that when the accident occurred, it can record related telemetry data and, based on this information, release payment. It can also withhold payment if the smart contract, after evaluating conditions of payment, concludes that payment should not be released; for example, in a scenario where an authorized workshop did not repair the vehicle or was used outside a designated area and so on and so forth. There can be many conditions that a smart contract can evaluate to process claims and the choice of these rules depends on the insurer, but the general idea is that smart contracts, in combination with IoT and oracles, can automate the entire vehicle insurance industry.

Several start-ups, such as **Dynamis**, have proposed smart contract-based P2P insurance platforms that run on the Ethereum blockchain. This was initially proposed to be used for unemployment insurance and does not require underwriters in the model.



The Dynamis white paper is available at <http://dynamisapp.com/>.

Post-trade settlement

This is the most sought-after application of blockchain technology. Currently, many financial institutions are exploring the possibility of using blockchain technology to simplify, automate, and speed up the costly and time-consuming post-trade settlement process.

To understand the problem better, the trade lifecycle will be described briefly. A trade lifecycle contains three steps: **execution**, **clearing**, and **settlement**. Execution is concerned with the commitment of trading between two parties and can be entered into the system via front office order management terminals or exchanges. Clearing is the next step, whereby the trade is matched between the seller and buyer based on certain attributes, such as price and quantity. At this stage, accounts that are involved in payment are also identified. Finally, the settlement is where, eventually, security is exchanged for payment between the buyer and seller.

In the traditional trade lifecycle model, a central clearing house is required to facilitate trading between parties, which bears the credit risk of both parties. The current scheme is somewhat complicated, whereby a seller and buyer have to take a complicated route to trade with each other. This comprises various firms, brokers, clearing houses, and custodians, but with blockchain, a single distributed ledger with appropriate smart contracts can simplify this whole process and can enable buyers and sellers to talk directly to each other.

Notably, the post-trade settlement process usually takes two to three days, and has a dependency on central clearing houses and reconciliation systems. With the shared ledger approach, all participants on the blockchain can immediately see a single version of truth regarding the state of the trade. Moreover, P2P settlement is possible, which results in the reduction of complexity, cost, risk, and the time it takes to settle the trade. Finally, intermediaries can be eliminated by making use of the appropriate smart contracts on the blockchain. Also, regulators can view the blockchain for auditing and regulatory requirements.



This can be very useful in implementing MIFID-II regulation requirements (<https://www.fca.org.uk/markets/mifid-ii>).

Financial crime prevention

Know Your Customer (KYC) and **Anti Money Laundering (AML)** are the key enablers for the prevention of financial crime. In the case of KYC, currently, each institution maintains their own copy of customer data and performs verification via centralized data providers. This can be a time-consuming process and can result in delays in onboarding a new client.

Blockchain can provide a solution to this problem by securely sharing a distributed ledger between all financial institutions that contain verified and true identities of customers. This distributed ledger can only be updated by consensus between the participants, thus providing transparency and auditability. This can not only reduce costs but also enable regulatory and compliance requirements to be satisfied in a better and consistent manner.

In the case of AML, due to the immutable, shared, and transparent nature of blockchain, regulators can easily be granted access to a private blockchain where they can fetch data for relevant regulatory reporting. This will also result in reducing complexity and costs related to the current regulatory reporting paradigm. This is where data is fetched from various legacy and disparate systems, and then aggregated and formatted together for reporting purposes. Blockchain can provide a single shared view of all financial transactions in the system that are cryptographically secure, authentic, and auditable, thus reducing the costs and complexity associated with the currently employed regulatory reporting methods.



There are already several solutions available, two of which are listed here:

Crypto-kyc: <https://www.crypto-kyc.com>

KYC-Chain: <https://kyc-chain.com>

There are many other solutions that can be found with a simple search for **Blockchain KYC** on an internet search engine.

Payments

A payment is a transfer of money or its equivalent from one party (the payer) to another (the payee) in exchange for services, goods, or for fulfilling a contract. Payments are usually made in the form of cash, bank transfers, credit cards, and cheques. There are various electronic payment systems in use, such as **Bacs Payment Schemes Limited (Bacs)** and the **Clearing House Automated Payment System (CHAPS)**.

All of these systems are, however, centralized and governed by traditional financial service industry codes and practices. These systems work adequately, but with the advent of blockchain, the potential of technology has arisen to address some of these limitations.

Some of the key advantages that blockchain technology can bring to payments are listed as follows.

Decentralization

Decentralization means that there is no requirement of a trusted third party to process payments. Payments can be made directly between parties without requiring any intermediary. This can result in reduced cost and faster (direct) payments between parties.

Faster settlement

Settlement can be much quicker compared to the traditional network due to the active presence of all parties on the network. Payment data can be shared and seen by all parties at the same time, and due to this settlement becomes quicker and more comfortable. Moreover, there is no requirement of running lengthy reconciliation processes because the data is all there on the blockchain, shared between all parties and readily available, which removes the requirement of the lengthy reconciliation process.

Better resilience

With a payment system running on a blockchain with potentially thousands of nodes around the world, the network becomes naturally resilient. It could also be argued that, with blockchain payments, there is no downtime because blockchain does not rely on traditional **disaster recovery (DR)** practices and is also better protected against malicious and **denial of service** attacks.

With all these advantages, it is easy to see how the payments industry can benefit from blockchain technology.

There is also another branch of payments that deals with international or cross-border payments, and comes with its own challenges. We'll discuss this next.

Cross-border payments

In traditional finance, cross-border payment is a complex process that can take days to process and involves multiple intermediaries. Current mechanisms suffer from delays incurred by multiple intermediaries, enforcement of regulations, differences in terms of regulations between different jurisdictions... the list goes on. All of these issues can be addressed by utilizing blockchain technology. The most significant advantage is decentralization, where, due to the lack of requirement of intermediaries, payments can be made directly between businesses or individuals. Also, due to P2P connectivity, the whole process becomes a lot faster—almost immediate, in fact—which results in more productivity and business agility.

Peer-to-peer loans

Blockchain also enables P2P loans, where lenders and borrowers can deal with each other directly instead of relying on a third party. Currently, in the

decentralized finance (DeFi) landscape, there is a large percentage of lending **DApps** that deal with the lending and borrowing of crypto tokens.

In this section, we covered some common use cases of blockchain in the finance industry. Next, we'll discuss media, another use case where blockchain can provide a cheaper, fair, and transparent media ecosystem.

Media

Critical issues in the media industry revolve around content distribution, rights management, and royalty payments to artists. For example, digital music can be copied many times without any restrictions, and any attempts to apply copy protection have been hacked in some way or other. There is no control over the distribution of the content that a musician or songwriter produces; it can be copied as many times as needed without any restriction, and consequently has an impact on the royalty payments. Also, payments are not always guaranteed and are based on traditional airtime figures. All these issues revolving around copy protection and royalty payments can be resolved by connecting consumers, artists, and all players in the industry, allowing transparency and control over the process.

Blockchain can provide a network where digital music is cryptographically guaranteed to be owned only by the consumers who pay for it. This payment mechanism is controlled by a smart contract instead of a centralized media agency or authority. The payments will be automatically made based on the logic embedded within the smart contract and the number of downloads.



A recent example of such an initiative is Musicoin:
<https://musicoin.org>.

Moreover, illegal copying of digital music files can be stopped altogether because everything is recorded and owned immutably in a transparent manner on the blockchain. A music file, for example, can be stored with

owner information and a timestamp that can be traced throughout the blockchain network. Furthermore, the consumers who own a legal copy of some content are cryptographically tied to the content they have, and it cannot be moved to another owner unless permissioned by the owner. Copyrights and transfers can be managed easily via blockchain once all digital content is immutably recorded on the blockchain. Smart contracts can then control the distribution and payment to all concerned parties.

Summary

There are many applications of blockchain technology and, as discussed in this chapter, they can be implemented in various industries to bring about solutions to existing problems. In this chapter, five main industries that can benefit from blockchain have been discussed. First, IoT was discussed, which is another revolutionary technology in its own right. By combining it with blockchain, several fundamental limitations can be addressed, which brings about tremendous benefits to the IoT industry. More focus has been given to IoT in this chapter, as it is the most prominent and most ready candidate for adapting blockchain technology. Moreover, applications in the government sector were discussed, whereby various government processes such as homeland security, identification cards, and benefit disbursements can be made transparent, secure, and more robust.

Furthermore, issues in the finance sector were discussed, along with possible solutions that blockchain technology could provide. Although the finance sector is exploring the possibilities of using blockchain with high energy and enthusiasm, it is still far away from production-ready, blockchain-based systems. Finally, some aspects of the health sector and music industry were also discussed. All these use cases and much more in the industry stand on pillars provided by core attributes of blockchain technology, such as decentralization, transparency, reliability, and security. However, certain challenges need to be addressed before blockchain technology can be adopted fully. These will be discussed in the next chapter.

20

Enterprise Blockchain

In this chapter, we'll investigate **enterprise blockchains**. What is the standard architecture of enterprise blockchains? Why they are required? We will answer these questions, and will also try to answer the big question of why current public blockchains are not necessarily a suitable choice for enterprise use cases.

We will also introduce several enterprise blockchain platforms, including Quorum and Corda. Along the way, we'll cover the following topics:

- Enterprise solutions and blockchain
- Limiting factors
- Requirements
- Enterprise blockchain versus public blockchain
- Use cases of enterprise blockchains
- Enterprise blockchain architecture
- Designing enterprise blockchain solutions
- Blockchain in the cloud
- Currently available enterprise blockchains
- Enterprise blockchain challenges
- Corda
- Quorum
- Setting up Quorum with IBFT
- Other Quorum projects

We discussed several different blockchain types in *Chapter 1, Blockchain 101*, including permissioned, public, private, and consortium blockchains. Enterprise blockchains are a permissioned **consortium** chain type that primarily tackle enterprise requirements.

Permissioned doesn't mean private. Permissioned blockchains can also be public and allow access only to known participants. Enterprise blockchains are usually private and permissioned, and are run between consortium members. While public blockchains provide integrity, consistency, immutability, and security guarantees, they lack certain features, which makes them less suitable for enterprise usage. We'll discuss these limitations in detail shortly.

First, we'll look into what enterprise solutions are and how blockchain can fit into an enterprise. Secondly, we'll see what questions should be answered before introducing an enterprise blockchain solution to a business.

Enterprise solutions and blockchain

Enterprise solutions integrate different fragments of a business and enable it to achieve its goals by providing business-critical information to the stakeholders. Therefore, an enterprise blockchain solution should be able to achieve this outcome. Here, we'll consider the question: *does blockchain fit this definition and help achieve enterprise business goals?*

If a blockchain solution can integrate different facets of a business and enable it to achieve its goals, then clearly it is an enterprise solution that brings real business value. In order to find out if a blockchain is suitable for an enterprise or not, we can ask some questions. Some of these questions are listed as follows:

- How can business/enterprise processes be improved using blockchain technology?

- Do I really need an enterprise blockchain? For this, we can ask some questions to rationalize whether we need an enterprise blockchain solution or not. Does my use case:
 - Need shared data between participants?
 - Have participants from other organizations that are not necessarily trustworthy and have conflicting interests?
 - Need strict auditing? A blockchain can provide this, as it is an immutable and tamper-proof chain of records that can provide a definite audit trail of activities performed on the chain (enterprise system).
 - Need to ensure the confidentiality of transactions?
 - Need to ensure the anonymity of participants?
 - Need controlled but transparent updates to the ledger?
 - Not need a single trusted authority; instead, the decisions on the network (updates to the ledger) should be consortium member-driven and agreed between members?

If the answer to any of the preceding questions is *yes*, then using an enterprise blockchain solution could be a good option. Otherwise, a traditional database might give a better alternative.

In addition to the questions mentioned here, when proposing an enterprise blockchain solution, we also need to answer some other questions from a business perspective that help establish the overall vision and strategy of the implementation of the blockchain solution:

- What is the overall objective of the proposed blockchain solution? Does it align with the business goals of the enterprise? Is it a **Proof of Concept (PoC)** project, intended just to demonstrate an idea, or a production project with real business deliverables?
- Where would it be deployed—cloud or local hosting? Who will manage the blockchain solution once it is deployed?
- Risk management considerations—does the solution follow any established guidelines for risk management? For example, NIST 800-

37 is a risk management framework that provides a process used to manage security and privacy risk for IT systems and organizations.

- Are there any other projects that this organization may have implemented already using enterprise blockchain? Can we learn from previous experiences and leverage some of the resources and best practices that may have been developed previously?

These questions should be—and are usually—asked when developing any other enterprise solution, but in relation to blockchain, these questions become even more critical due to the nascent and immature nature of enterprise blockchain technology. A clear definition of the objectives, along with clear alignment with business requirements, will result in an implementation that meets business goals. Now, the question arises regarding how blockchain solutions can be developed that can bring real business value. We'll learn later in this chapter how enterprise blockchain solutions can be designed using established architecture frameworks.

So, in summary, after answering the aforementioned questions even informally and then engaging in a formal exercise of designing an enterprise blockchain solution using an enterprise framework, we can answer the question regarding whether a proposed blockchain solution is an enterprise solution that meets business objectives. We'll cover designing enterprise blockchain solutions later in this chapter.

Next, we'll envisage some success factors that can help enterprise blockchains to become successful and adopted in the enterprise.

Success factors

There are some business-oriented factors that should be addressed for a successful enterprise blockchain solution.

Of the utmost importance is the requirement that the blockchain solution should bring some economic value and help to achieve real business goals. Moreover, enterprise solutions should be aligned with business goals.

The primary value of enterprise blockchains is in the property of being a sharable, replicable, permissioned ledger between organizations that immediately results in cost reduction by eliminating the need for data exchange. In doing so, we also eliminate the need for infrastructure and tools to support such exchanges. Also, due to the security, immutability, and auditing provided as inherent features of a blockchain, there is no need to invest separately for these requirements.

Blockchain solutions that integrate easily/seamlessly with existing systems provide better value because already mature legacy systems (at least at this stage) provide a mechanism to connect with the blockchain, read its data, and save it in a known format that the enterprise is already familiar with. Moreover, just a blockchain network on its own is not entirely useful in enterprises if it is not integrated with existing back-office systems such as **Enterprise Resource Planning (ERP)**, backend databases, record reconciliation systems, or other organization reporting tools.

An enterprise blockchain solution should be seen as a complete end-to-end enterprise solution as part of the larger enterprise architecture, instead of only a siloed blockchain network. We'll explore this topic later in this chapter, under the *Designing enterprise blockchain solutions* section.

Enterprise-grade governance, control, and security are also desirable features from a business perspective as they allow business stakeholders to apply already-established organization rules and policies to the blockchain. This can also help to achieve regulatory and compliance requirements.

Now, the following question arises: *with the availability of many different public blockchain platforms, why has the adoption of blockchain technology in enterprises still not been fully achieved?* The reason is that there are several factors that make public blockchains unsuitable for enterprise use cases. In the next section, we'll answer this question and explore why public blockchains are not quite suitable for enterprise use cases.

Limiting factors

We discussed several benefits of blockchain technology in general in *Chapter 1, Blockchain 101*. While all those benefits are attainable using public blockchains, several features are lacking in public blockchains, which makes them unsuitable for use in enterprise use cases. The interest in enterprise blockchain arises from these limitations in public blockchains, along with specific requirements in any business.

We'll describe some of the most common concerns next.

Slow performance

Public blockchains are slow and can process only a few transactions per second. Bitcoin processes 3-4 transactions per seconds, while Ethereum processes around 14. This low transaction rate is not suitable for businesses that usually require high transaction speed. For example, card payment businesses typically need to process thousands of transactions per second.

Lack of access governance

Public blockchains are available for anyone to join, which makes it easier for investors and cryptocurrency enthusiasts to join and helps with the network effect. However, in an enterprise, all participants must be known so that everyone knows who they are dealing with. This lack of an identification and access control mechanism makes public blockchains rather unsuitable for businesses.

Lack of privacy

Public blockchains are inherently transparent, and everything on the ledger is visible to everyone. This means anyone can easily view transaction details and participants involved in a transaction.

Probabilistic consensus

Public blockchains usually use a **Proof of Work (PoW)** type of consensus mechanism, which is inherently a probabilistic protocol that provides probabilistic finality. Even though the confirmations mechanism, as discussed in *Chapter 6, Introducing Bitcoin*, does give a certain level of confidence that a transaction is irrevocable, it is still possible that the chain may fork and transactions can be lost. This issue is particularly a concern for businesses where once a transaction commits, it is deemed final. Imagine receiving a property ownership document on a chain from someone only to discover later that the blockchain has forked, and that you are no longer the owner of the property.

To address this limitation, enterprise blockchains use deterministic consensus algorithms, which provide immediate finality.

Transaction fees

In Ethereum or other similar blockchains, the transaction fee is charged in the native cryptocurrency for each transaction execution. While this mechanism provides incentives for miners and protects against spam, there is no requirement of such an arrangement in enterprise blockchains. If an organization were to use public blockchains for their business transactions, then they'd need cryptocurrency in reserve to pay for operations on the blockchain. This extra cost might be undesirable for some businesses.

While the concerns mentioned here are considered limitations in public blockchains, based on these concerns and limitations, we can derive and define several requirements, or features, of a blockchain that will enable it to become an enterprise blockchain. In other words, the limitations in public blockchains can be seen as requirements of enterprise blockchains.



With all these factors to address, it is essential to remember that, based on the current climate and interest, enterprise blockchain is indeed going to stay. No matter how many challenges and concerns there are, sooner or later, they will be addressed. In fact, there are already production-ready projects

built on enterprise blockchains. We will explore some of those, specifically **Corda** and **Quorum**, later in this chapter.

The next section talks about several features that a blockchain should possess to become suitable for enterprises.

Requirements

In addition to *integrity* and *consistency*, which are also provided by public blockchains, there are several other requirements specifically for enterprise blockchains that make them suitable for enterprise use cases. In some cases, some requirements become even stricter in enterprise blockchains compared to public blockchains. For example, in public blockchains, eventual consistency is acceptable. However, in enterprise blockchains, the moment a transaction is committed, it should immediately finalize and irrevocably become part of the global record (state). Thus, we'll begin by briefly defining integrity and consistency before introducing specific requirements for enterprise blockchains.

- **Integrity:** This attribute of a blockchain is provided by the use of hash functions and digital signatures, and plays a vital role in the overall security of the blockchain. Hash functions allow us to check for any data modifications, whereas digital signatures ensure that the messages that originated from a sender have not been altered.
- **Consistency:** This attribute of a blockchain ensures that all honest nodes agree on the same sequence of blocks. To achieve this, various mechanisms are used, such as PoW or **Byzantine Fault Tolerance (BFT)** consensus protocols.



You can refer to *Chapter 1, Blockchain 101*, for a refresher on this, as we introduced these properties there in more detail.

Now, we'll introduce three fundamental requirements that should be met for a blockchain to become suitable for enterprises. These requirements are **privacy**, **performance**, and **access governance**.

Privacy

Privacy is of paramount importance in enterprise blockchains. Privacy has two facets: first, *confidentiality*, and second, *anonymity*.

Confidentiality

Confidentiality is a fundamental requirement in an enterprise. It is anticipated that, in enterprise blockchains, all transactions hide their payloads so that the amount of transactions is not revealed to anyone who is not privy to the transaction.

Types of private transaction

Private transactions can be defined as transactions that meet the privacy requirements of an enterprise use case. There are two types of private transactions, as defined by **Enterprise Ethereum Alliance (EEA)**:

- **Restricted private transactions:** This type of private transaction is transmitted only to those parties on the blockchain network who are a privy to the transaction.
- **Unrestricted private transactions:** Under the unrestricted private transaction paradigm, private transactions are transmitted to all participants on the network, regardless of whether they are party to the transaction or not. The payload is still encrypted and confidential, but the transaction itself is broadcast to the entire network.

There are many approaches to achieving privacy in enterprise blockchains and blockchain in general. These methods range from an off-chain mechanism like privacy managers (used in Quorum and some other Enterprise chains) to utilizing **zero-knowledge proofs (ZKPs)**, trusted hardware, and **secure multiparty computation (SMPC)**. We'll cover some

of these techniques in the next chapter, *Chapter 21, Scalability and Other Challenges*, but in this chapter, we will mainly focus on privacy manager-based privacy, where an off-chain component is used to provide privacy services. We will discuss the privacy manager-based approach in the *Quorum* section later in this chapter.

Anonymity

Anonymity in enterprise blockchains might not seem a strict requirement at first, because all participants are known and identified. However, it is necessary for scenarios with some competing participants and conducting business on-chain. It could be essential, for example, that in a scenario where parties X and Y are transacting together, party Z doesn't find out which parties are transacting together. Even though the transaction values are not visible, it can still reveal details about the business that the two parties might be doing. If that information is available publicly, then other parties on the consortium chain will gain market intelligence and may try to influence that process via marketing or other methods.

Performance

Due to the high-speed requirements of businesses, enterprise blockchains must be able to process transactions at a high rate.

Scalability/speed

Performance has two facets: scalability and speed. Speed deals with how many transactions can be processed in a given amount of time. It deals with the ability of a system to handle a large volume of transactions at an acceptable speed.

On the other hand, we have the number of participants in a system. Public blockchain can support a large number of users. This is especially true in cryptocurrency blockchains such as Bitcoin and Ethereum. This is possible due to the PoW or **Proof of Stake (PoS)** algorithms for consensus used in these public blockchains. However, in enterprise blockchains, a different

class of consensus algorithms (most commonly BFT) are used, which do not work well with a large number of nodes. This results in limiting the number of users on a consortium chain.

Here, we have to consider a tradeoff. Ideally, an enterprise blockchain should be able to perform well with a large number of users; however, with the tradeoff, enterprise blockchains should be able to process transactions at a high rate. This is what most enterprise blockchains focus on since, in many use cases, the number of nodes is not that high, and speed can be prioritized over scalability.

In public blockchains, sometimes, network congestion caused by high volumes of traffic can cause performance issues and can increase the transaction processing times. This is detrimental to **enterprise DApps**, which need faster transaction processing and response times. Network congestion also results in increased gas prices, which can also be a concern for businesses from a cost perspective.

Access governance

From another angle, being a permissioned blockchain, enterprise-grade access control (in the form of either a new mechanism on the chain, or control driven by an enterprise SSO already in place) is a fundamental requirement in enterprise blockchains. As all participants must be identifiable on a consortium chain, it is essential to build an access control mechanism that facilitates that process. This feature can be achieved by using enterprise-grade access control mechanisms such as **Role-Based Access Control (RBAC)**.



RBAC is an ANSI Standard. More information is available in the ANSI INCITS 359-2004 document. You can find more information on the document and the RBAC standard here:
<https://csrc.nist.gov/projects/role-based-access-control>

The access control mechanism also can address **Know Your Customer (KYC)** requirements.

In addition to privacy, performance, and access governance, which are usually identified as three fundamental requirements of enterprise blockchains, there are also some other requirements that are highly desirable but can be considered somewhat optional, as they are more use case-dependent. We'll describe these next.

Further requirements

In this section we'll present some further requirements that are very useful and can increase the suitability/efficacy of enterprise blockchain solutions. We'll discuss these requirements as follows.

Compliance

A common concern is compliance. Public blockchains are not suitable for enterprise use cases due to strict regulatory and law requirements in almost all sectors such as finance, health, and government. Compliance challenges mainly include regulatory compliance, standards compliance, data sovereignty, and liability concerns. We'll define these briefly next.

Compliance with standards and regulations: Often, in enterprise use cases, compliance with a technical standard or laws is required. For example, compliance with GDPR is mandatory in the European Union and the European Economic Area. Another example is compliance with **Financial Conduct Authority (FCA)** regulations in the UK. Moreover, it might be necessary, in a use case, to comply with technical standards such as cryptography standards published by NIST.



One such example is the use of NIST approved curves, as described here:
<https://csrc.nist.gov/Projects/elliptic-curve-cryptography>.

Data sovereignty: Data sovereignty is a broad topic that mainly subjects data to the laws of the country in which it is located. For example, under GDPR, transferring personal data outside the EU is subject to adequacy decisions (Article 45) and the appropriate safeguards (Article 46).



More on this GDPR guidance can be found here:
<https://gdpr.eu/tag/chapter-5/>.

As public blockchains are borderless and geographically dispersed systems, compliance with such regulations can be challenging.

Liability: In traditional business and IT systems, legal responsibility often lies with a party who provides a specific service; for example, a cloud service provider being responsible for handling data in accordance with local laws and regulations. In a decentralized public blockchain, the data is on a public blockchain, and it therefore becomes challenging to keep a single party responsible for data management or providing services. In case of malicious incidents, again, it is not possible to hold any single party responsible.

This limitation poses a significant challenge in enterprise settings where, usually, a responsible party is in control of service provision and is held accountable. From another angle, we know that, in traditional systems, in case of any problems, a legal system can help. However, in a decentralized blockchain, it can become quite challenging to blame any single party for their actions. Therefore, in enterprise blockchains, the ability to comply with regulations and law becomes a very sought-after attribute.

Interoperable

As the enterprise blockchain ecosystem evolves, the need to be able to exchange data between the disparate enterprise and public blockchains also arises. Lack of standardization also alleviates this problem; however, standardization efforts are underway, such as EEA (discussed in the *Enterprise blockchain challenges* section). There are also interoperability

solutions being developed and available for blockchains that allow interoperability between chains, such as **ION**, **Polkadot**, and **Interledger**.

We will discuss interoperability further in *Chapter 21, Scalability and Other Challenges*.

Integration

No enterprise blockchain is an isolated end-to-end solution. It has to integrate with existing enterprise systems or other off-chain systems that are part of the whole enterprise solution to fulfil business goals. Therefore, enterprise blockchains must provide interfaces for integration. This can be as simple as providing RPC endpoints, or as complex as building blockchain-specific connectors and plugins to integrate with enterprise service bus or other legacy systems—more on this in the *Enterprise blockchain architecture* section.

Integration with security devices such as **Hardware Security Modules (HSMs)** is also quite desirable for many enterprise use cases where strict security is required, or due to regulatory or compliance requirements.

Ease of use

Usually, enterprise systems are easy to deploy and use. Deployment in enterprises easy and quick, and often relies on mature frameworks and tools such as Ansible and other proprietary tools, but this is not the case with blockchain.

Deployment of enterprise systems is a well-studied, understood, and mature area. With enterprise orchestration tools and established techniques over the years, enterprise deployment has become easy. However, blockchains are not as easy to deploy as other enterprise systems. With the availability of **Blockchain as a Service (BaaS)** and deployment automation tools, this is changing. However, there is still some work that needs to be done.

Monitoring

Monitoring using visualization tools plays a vital role in any enterprise solution. Without the ability to monitor and visualize a system, it is almost impossible to ensure the health of the enterprise system. It is also a desirable feature in enterprise blockchain solutions to be able to visualize the blockchain network. Monitoring a blockchain allows an administrator to keep an eye on the network's health and operations. It allows an administrator to monitor and respond to the events of interest, such as a node going down, communication link slowness, a node being unable to sync with the blockchain, and many other scenarios.

Secure off-chain computation

In some scenarios, it is desirable to be able to offload some intensive computation to off-chain systems; for example, if there is a requirement to do some computation that requires **High Performance Computing (HPC)** resources.

This somewhat overlaps with integration, but it's mentioned here separately because of specific security requirements that the off-chain computations must be provably correct with integrity and authenticity guarantees.

Better tools

Usually, in enterprise systems, there are many supporting tools and utilities packaged with the main product to operate the software. For example, these include administration tools, deployment tools, developer utilities, visualization tools, management tools, and end user tools. Blockchain platforms with better tooling are much desirable because of better user support. Tools such as block explorers, user administration modules, and DApps to manage smart contracts are quite useful. They are gradually becoming more mature as the whole blockchain ecosystem is growing.

Now that we understand the features of enterprise blockchains, we will present a comparison between public and enterprise blockchains to help understand the main differences.

Enterprise blockchain versus public blockchain

In this section, we'll provide a comparison between public and enterprise blockchains. Consider the table shown here, which assesses some points of comparison between the two blockchain types. This is not an exhaustive list; however, we'll touch on the most major points:

Aspect	Public chains	Enterprise chains
Confidentiality	No.	Yes.
Anonymity	No.	Yes.
Membership	Permissionless.	Permissioned via voting, KYC, usually under an enterprise blockchain.
Identity	Anonymous.	Known users.
Consensus	PoW/PoS.	BFT.
Finality	Mostly probabilistic.	Requires immediate/instant finality.
Transaction speed	Slower.	Faster (usually, should be).
Scalability	Better.	Not very scalable, usually due to consensus choice. Usually a much smaller number of nodes compared to public chains.
Regulatory compliance	Not usually required.	Required at times.

Trust	Fully decentralized.	Semi-centralized and managed via consortium and voting mechanisms.
Smart contracts	Not strictly required; for example, in the Bitcoin chain.	Strictly required to support arbitrary business functions.

The preceding table compares several key aspects of public and enterprise platforms. Next, we'll briefly consider some of the use cases of enterprise blockchains.

Use cases of enterprise blockchains

Considering the preliminary considerations, limiting factors, and requirements, which we've discussed so far, of employing an enterprise blockchain in an organization, there are several clear use cases of enterprise blockchains. Some of these standout scenarios are listed as follows:

- Payments
- Supply chain
- IoT
- Insurance
- Healthcare

Evidently, enterprise blockchains have many use cases, including but not limited to payments, insurance, KYC, and monitoring supply chains. We introduced some of these use cases throughout this book, especially in *Chapter 1, Blockchain 101*, and some in *Chapter 19, Blockchain – Outside of Currencies*. The use cases discussed in the latter chapter are also applicable to enterprise use cases, such as the Internet of Things scenario, which can be implemented in both public and private blockchains. We will

cover some more use cases in the context of the latest developments in *Chapter 22, Current Landscape and What's Next*.

In the next section, we'll describe enterprise blockchain architecture, which can help us communicate with stakeholders, promote early design decisions, serve as a reusable model for development, build shared understanding, and understand how the system is structured.

Enterprise blockchain architecture

A typical enterprise blockchain architecture contains several elements. We saw a generic blockchain architecture in *Chapter 1, Blockchain 101*, and we can expand and modify that a little bit to transform it into an enterprise blockchain architecture that highlights the core requirements of an enterprise blockchain. These requirements are mostly driven by enterprise needs and use cases:

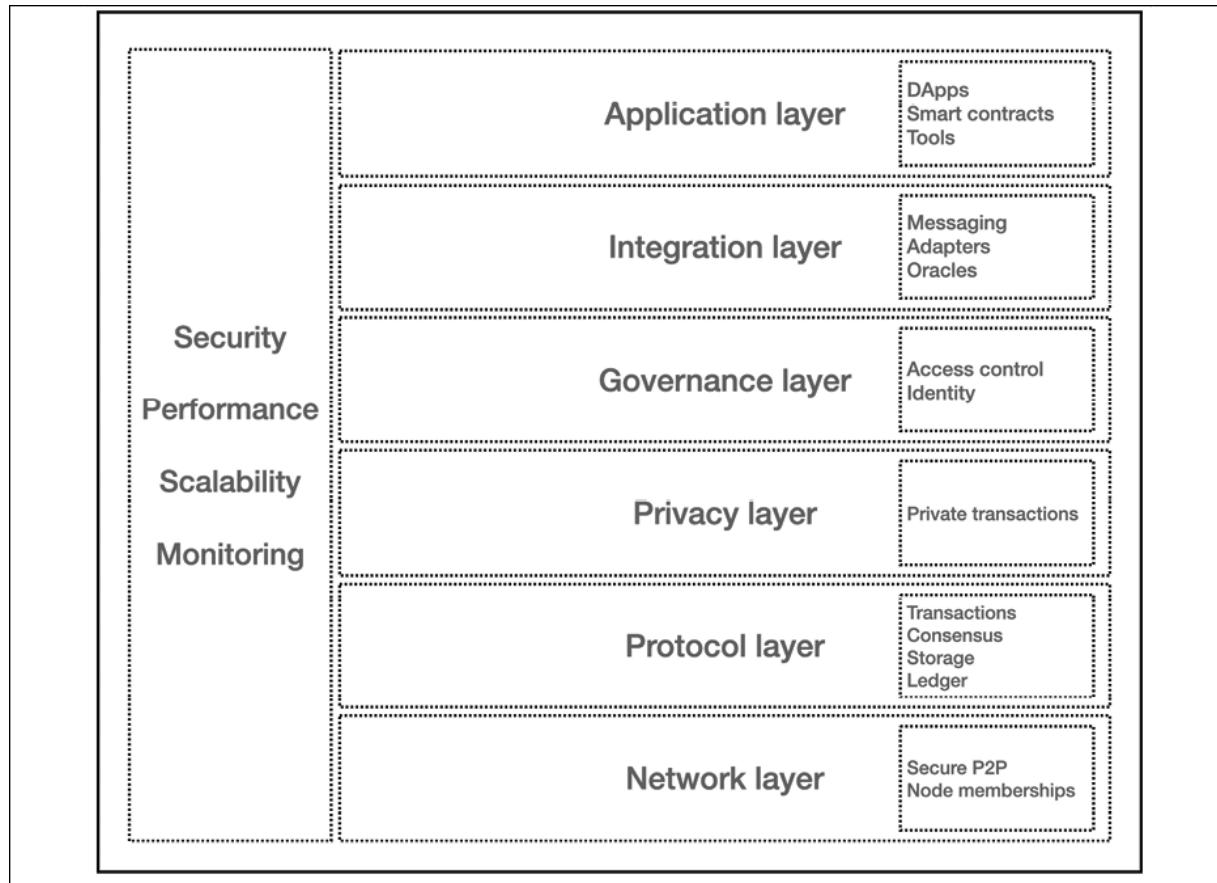


Figure 20.1: Enterprise blockchain layered architecture

We'll discuss each of these layers as follows.

Network layer

The network layer is responsible for implementing network protocols such as **peer-to-peer (P2P)** protocols used for information dissemination.

Protocol layer

This is the actual ledger layer, or blockchain layer, where the core consensus, transaction management, and storage elements are implemented.

Privacy layer

This layer is responsible for providing one of the core features of enterprise blockchain: privacy. There are a number of ways to achieve privacy, including off-chain privacy managers, zero knowledge, and hardware assisted privacy. Hardware-based privacy is usually supported using **trusted execution environments (TEEs)**. On the other hand, software or algorithmic privacy such as **zero-knowledge proofs (ZKPs)** are also quite common in enterprise blockchains.

Governance layer

This layer is responsible for providing the enterprise grade access control mechanism that controls the consortium network membership. This can either be controlled via on-chain permissioning system implemented in smart contracts, as part of the software client, or can be integrated with existing off-chain enterprise permissioning systems such as **Single Sign-On (SSO)** or **Active Directory (AD)**.

Integration layer

This layer provides APIs and a mechanism used to integrate with the legacy or existing back-office systems. This can be as simple as an RPC layer providing APIs over RPC or can constitute built-in connectors and plugins for integrating with the enterprise service bus.

The integration layer is responsible for ensuring integration with back-office, legacy, and existing off-chain systems. It is not part of the core protocol but as part of the holistic view of the blockchain end-to-end solution, this layer is vital for delivering business results. While there are many techniques available, a commonly integration framework used in the enterprise environment is **Apache Camel**. This can also be used in blockchain solutions as it comes with the Ethereum Web3J library component.

Let's provide a quick introduction to Apache Camel, before we move onto other layers of the enterprise blockchain architecture.

What is Apache Camel?

Apache camel is an open source enterprise integration framework. It enables easy integration between various systems by utilizing enterprise integration patterns. It has hundreds of components for different systems such as databases, APIs, and MQs for easy integration between systems.

For example, Web3J connector (Apache Camel Ethereum connector) is a feature-rich connector available in Apache Camel that enables integration between Ethereum chains and other systems.

The Apache Camel Ethereum connector works with Ethereum Geth nodes, Quorum, Parity, and Ganache. It supports JSON RPC API over HTTP and IPC with implementations for different blockchain operations such as `net`, `eth`, `shh`, and so on. It has support for Ethereum filters and **Ethereum Name Service (ENS)**, which can be explored further at <https://ens.domains>. It has support for JSON RPC API and is also a fully tested (unit and integration) solution.

A generic high-level design using Apache Camel is shown in the following diagram:

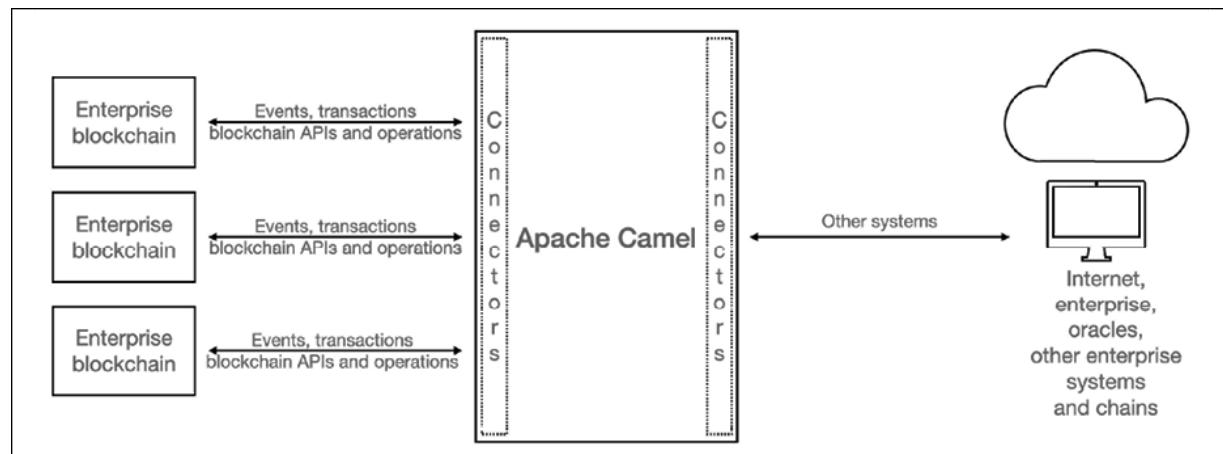


Figure 20.2: High-level design of Apache Camel, which enables blockchain integration

In the preceding diagram, we can see several enterprise blockchains connecting to Apache Camel using Apache Camel Ethereum connectors. Also, other systems connect to Apache Camel via their respective connectors. Apache Camel via connectors is responsible for integrating between all these systems. For example, it is entirely possible to use Apache Camel to take data from enterprise blockchains and store it in traditional SQL databases in the enterprise via suitable connectors.



More details on the Web3J component is available here:

<https://camel.apache.org/components/latest/web3j-component.html>.

Now, let's return to our discussion on the enterprise blockchain layers and introduce the application layer.

Application layer

The application layer, as the name suggests, consists of **DApps**, **smart contracts**, tools, and other relevant software to support enterprise use cases.

Security, performance, scalability, monitoring

On the left-hand side in the architecture diagram (*Figure 20.1*), security, scalability, performance, and monitoring is shown. As each layer in the enterprise blockchain benefits from and indeed requires security, scalability, and performance, this layer is shown as encompassing all layers.

Monitoring also plays a vital role in any enterprise solution. Without effective visualizations in such complex networks, it is almost impossible to keep track of everything. Therefore, this layer is shown as relevant to all enterprise blockchain layers.

Now that we understand the architecture of enterprise blockchains in general, let's dive a little bit deeper into the mechanics of designing enterprise blockchain solutions. In the next section, we'll see which tools and frameworks we can use to build enterprise solutions.

Designing enterprise blockchain solutions

An isolated blockchain in an enterprise is not sufficient enough to solve business problems. In addition to choosing a blockchain platform, there are some other factors to consider while introducing a blockchain in an enterprise. On top of the list of these factors is integration with the existing back-office and legacy systems.

There are already established and mature frameworks like **The Open Group Architecture Framework (TOGAF)** Zachman Framework to facilitate enterprise architecture development. With the advent of enterprise blockchain technology, a question arises regarding whether we can leverage existing frameworks to address enterprise blockchain architectural needs. The answer is *yes*.

In the section, we'll look at TOGAF and its **architecture development method (ADM)**, and consider how blockchain solutions can benefit from the TOGAF framework.



Other models can also be used, but TOGAF is popular and quite commonly used, which is why we are discussing it here.

Without any architecture framework, it becomes quite challenging to have a holistic view of an enterprise and see how all business processes fit together. To address this challenge, we can use enterprise architecture frameworks.

The purpose of enterprise architecture frameworks is to enable an organization to execute its business strategy effectively. It allows an organization to see an organization from different perspectives, including business, information, process, and technology, and make effective business decisions to achieve business goals.

Let's now explore the popular enterprise architecture framework, TOGAF.

TOGAF

TOGAF stands for **The Open Group Architecture Framework**. It is developed by the Open group.



The official TOGAF website can be found here:
<https://www.opengroup.org/togaf>.

It is a framework that enables organizations to systematically design, plan, and implement enterprise solutions in businesses. It has four architectural domains.

Business architecture domain

This domain defines the business strategy, organization structure, business processes, and governance of the organization.

Data architecture domain

This domain describes structures of an organization's data assets and relevant data management resources.

Application architecture domain

This domain describes the enterprise applications, deployment blueprints, relationships, and interactions between applications, along with the

relationships these applications have with business processes within the enterprise.

Technology architecture domain

This domain defines the requirements of the technical architecture implementation of the enterprise applications. This includes the description of hardware, software, middleware, and network infrastructure required for implementation of the enterprise applications.

Now, re-examining each domain with blockchain in mind, we can include blockchain at each layer and make decisions such as business requirements, the application architecture, logical and physical data assets, and finally the infrastructure required for implementing the enterprise blockchain solution.

After this basic introduction to TOGAF, let's dive a bit deeper to understand the method TOGAF uses to develop an IT architecture.

Architecture development method

TOGAF ADM is a method for developing IT architecture that meets organizational business needs. It describes the process of moving from the foundational TOGAF architecture to an organization-specific architecture. This is an iterative process with continuous requirements management, which results in the development of an architecture that is specific to an organization and addresses specific needs. Once the architecture development is complete, the architecture can be published throughout the organization to develop a common understanding.

The ADM is a tested and repeatable process for developing enterprise architectures. The process follows the following steps: establish architecture framework, develop architecture content, architecture governance, and implementation of architectures.

The ADM has nine phases and each phase can be further divided into multiple steps. The ADM model is shown here:

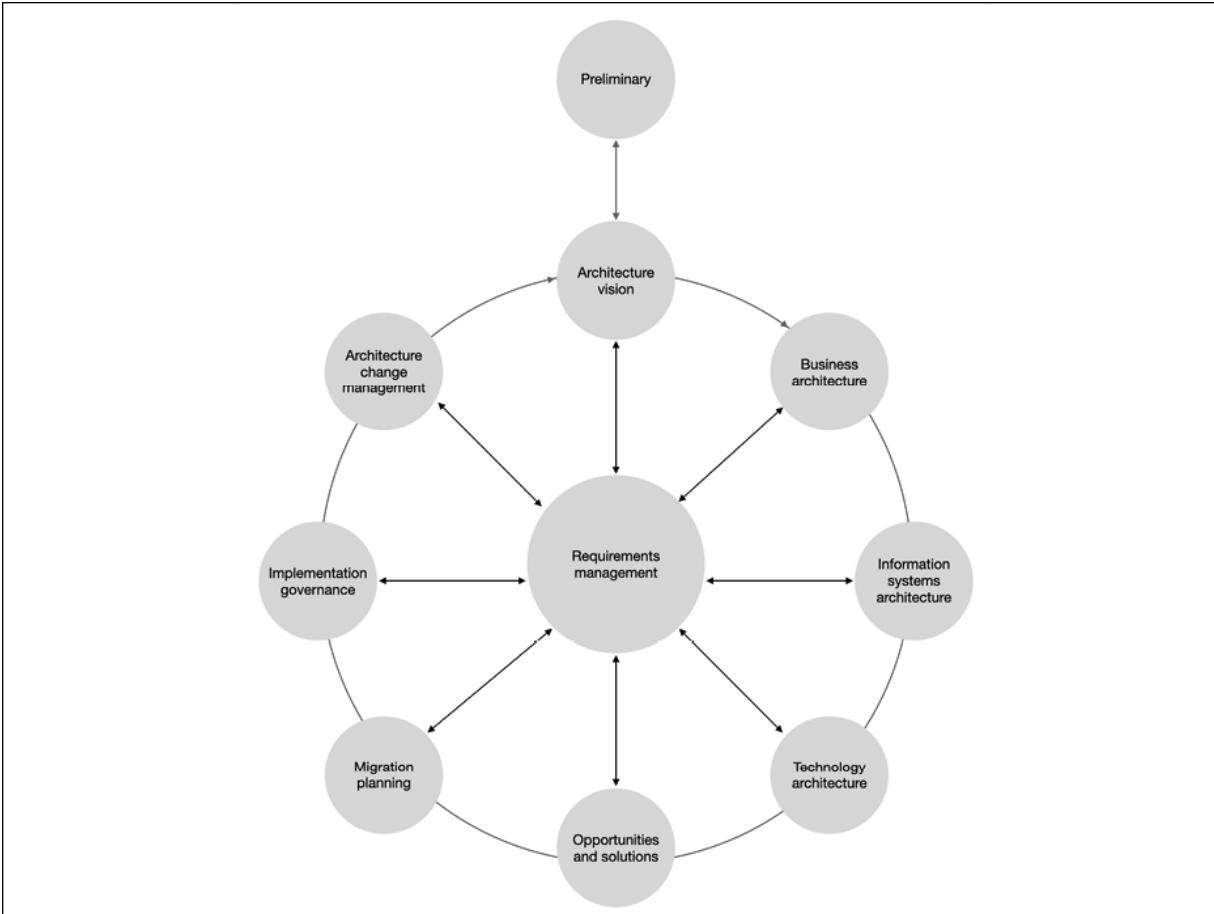


Figure 20.3: ADM model



The original diagram can be found at
<https://pubs.opengroup.org/architecture/togaf9-doc/arch/chap05.html>.

Now, we'll describe each phase in more detail.

Preliminary phase

This phase defines the groundwork of the architecture. It defines methodologies, architecture principles, the architecture scope, and assumptions about the architecture. It also defines the responsible parties for architecture delivery.

From a blockchain perspective, we can define the overall blockchain strategy in this phase.

Architecture vision

This phase validates business principles, goals, and the business strategy. It defines key business requirements, as well as a high-level description of the business value expected from the architecture work. This phase also analyzes the impact of the new architecture on other processes.

Business architecture

This phase proposes the baseline business architecture and develops a target business architecture, along with gap analysis.

In a blockchain scenario, we can define the current state architecture and target state architecture at this stage.

Information systems architecture

This phase defines the data architecture and application architecture. The data architecture includes building the baseline data architecture, data architecture description, building data architecture models, doing impact analysis review reference models, and viewpoints. Application architecture defines the baseline application architecture, builds application architecture models, and proposes applications. Both of these activities also perform gap analysis to validate the architecture being developed, as well as to find any shortfalls between the baseline architecture and the target architecture.

Similar to the business architecture, for an enterprise blockchain solution, we can define the current state architecture and target state architecture.

Technology architecture

This phase is primarily concerned with reviewing the baseline business, data, and application architecture and building a baseline description of the

current technology architecture in the enterprise. It also proposes the target technology architecture.

In this phase, we can propose a target enterprise blockchain platform and the target solution.

Opportunities and solutions

This phase performs evaluations and selects various proposed target architectures. Also, an implementation strategy and plan are proposed at this stage. We can apply this to blockchain in a similar fashion, by evaluating or selecting the target architecture encompassing enterprise blockchain solutions.

When evaluating blockchain solutions, a number of features that we presented earlier in the comparison between enterprise blockchain and public blockchain can also be used; for example, confidentiality, scalability, and finality.

Migration planning

This phase creates and finalizes the comprehensive implementation plan for migrating to the target architecture from the current architecture.

Implementation governance

This phase deals with the implementation of the target architecture. A strategy to govern the overall deployment and migration is developed here. We can also perform blockchain solution testing and devise a deployment strategy during this phase.

Architecture change management

This phase is responsible for creating change management guidelines and procedures for the newly implemented target architecture. From a

blockchain perspective, this phase can provide change management procedures for the newly implemented enterprise blockchain solution.

With this, we've completed our introduction to TOGAF and explored the idea that enterprise blockchain solutions should be viewed through the lens of the enterprise architecture.

The fundamental idea to understand here is that enterprise blockchain solutions are not merely a matter of quickly spinning up a network, creating a few smart contracts, creating a web frontend, and hoping that it will solve business problems. This setup may be useful as a PoC or for an incredibly simple use case but is certainly not an enterprise solution. We suggest that an enterprise blockchain solution must be looked through the lens of the enterprise architecture and regarded as a full grade enterprise solution. This is so that we can effectively achieve the business goals intended to be solved by enterprise blockchain solutions.

Next, let's explore how we could implement a blockchain business solution in an organization that has moved its operations to the cloud.

Blockchain in the cloud

Cloud computing provides excellent benefits to enterprises, including efficiency, cost reduction, scalability, high availability, and security. Cloud computing delivers computing services such as infrastructure, servers, databases, and software over the internet. There are different types of cloud services available; a standard comparison is made between **Infrastructure as a Service (IaaS)**, **Platform as a Service (PaaS)**, and **Software as a Service (SaaS)**. A question arises here: where does blockchain fit in?

Blockchain as a Service, or **BaaS**, is an extension of SaaS, whereby a blockchain platform is implemented in the cloud for an organization. The organization manages its applications on the blockchain, and the rest of the software management, infrastructure management, and other aspects such as security and operating systems are managed by the cloud provider. This

means that the blockchain's software and infrastructure are provided and maintained by the cloud provider. The customer or enterprise can focus on their business applications without worrying about other aspects of the infrastructure.

The following is a comparison of different approaches:

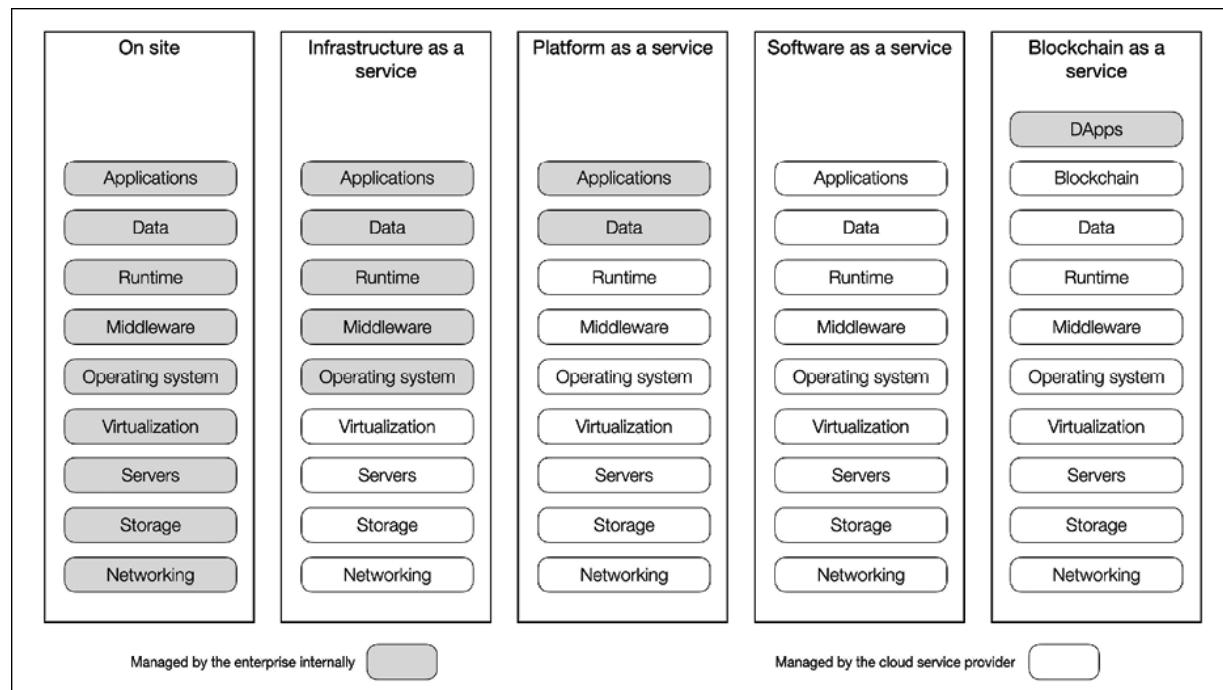


Figure 20.4: Cloud solutions

BaaS can be thought of as a SaaS, where the software is a blockchain. In this case, just like in SaaS, all services are externally managed. In other words, customers get a fully managed blockchain network on which they can build and manage their own **DApps**. Note that in the preceding diagram, under the **Blockchain as a Service** column, **Applications** have been replaced with **Blockchain**, as a differentiator between other cloud services and BaaS. Here, blockchain is the software (application) provided and managed by the cloud service provider. Also, note that **DApps** have been added on top, which are managed by the enterprise internally.

There are many BaaS providers. A few of them are listed as follows:

- AWS: <https://aws.amazon.com/blockchain/>

- **Azure**: <https://azure.microsoft.com/en-gb/solutions/blockchain/>
- **Oracle**: <https://www.oracle.com/uk/application-development/cloud-services/blockchain-platform/>
- **IBM**: <https://www.ibm.com/uk-en/cloud/blockchain-platform>

In the next section, we'll introduce some enterprise blockchain platforms.

Currently available enterprise blockchains

As this is a very ripe area for research, and indeed the market is very active in this space, there have been several enterprises blockchain solutions developed and made available in the last few years. Notably, the year 2019 has been called "the year of the enterprise blockchain" as many startups and enterprises focused on this area emerged.

In this section, we will not cover all these platforms in detail, but will provide a brief introduction and links to more details on these chains.

Corda

Corda is an open source distributed ledger platform intended for businesses. It supports privacy and assured identity, making it scalable and interoperable. We'll explore Corda in more detail later in this chapter.

Quorum

Quorum is an open source enterprise blockchain platform. It aims to support enterprise business needs and allows a business to achieve business

goals (and unlock economic value) by leveraging blockchain technology. It addresses crucial enterprise requirements including privacy, performance, and enterprise permissioning. We will also explore Quorum in greater detail later in this chapter.

Fabric

Fabric is a **Hyperledger** project. It is an enterprise-grade distributed ledger that allows the development blockchain solutions with a modular architecture. It has a permissioned architecture, support modularization, pluggable consensus, and supports smart contracts.

We discussed Hyperledger and other projects under it, such as **Sawtooth**, in detail in *Chapter 17, Hyperledger*.

Autonity

Autonity is an Ethereum-based protocol. It enables permissioned, decentralized, and interoperable enterprise networks. More information can be found here: <https://www.autonity.io/>.

In the next section, we'll provide a comparison of three leading enterprise distributed ledger platforms.

Comparison of main platforms

In this section, we'll provide a comparison of leading enterprise blockchain platforms. We'll cover most of the desirable features of enterprise blockchains. This can be used as a reference for a quick comparison between these platforms. This comparison is based on the current implementations of these platforms available at the time of writing. However, as this is a very rapidly changing area, some features may change over time or new features might be added or improved:

Feature	Quorum	Fabric	Corda
Target industry	Cross-industry	Cross-industry	Cross-industry
Performance (approximate transactions per second, or TPS)	700 (*)	560 (∞)	600 (@)
Consensus mechanism	Pluggable multiple Raft, IBFT, PoA	Pluggable Raft	Pluggable, notary-based
Tooling	Rich enterprise tooling	SDKs	Rich enterprise tooling
Smart contract language	Solidity	Golang	Kotlin/Java
Finality	Immediate	Immediate	Immediate
Privacy	Yes (restricted private transactions)	Yes (restricted private transactions)	Yes (restricted private transactions)
Access control	Enterprise-grade permissioning mechanism	Membership service provides/certificate based	Doorman service/KYC. Certificate-based
Implementation language	Golang, Java	Golang	Kotlin
Node membership	Smart contract and node software managed	Via membership service provider	Node software managed using configuration files, certificate authority controlled

Member identification	Public keys/addresses	PKI-based via membership service provider, supports organization identity	PKI-based, supports organization identity
Cryptography used	SECP256K1 AES CURVE25519 + XSALSA20 + POLY13050 PBKDF2 SCRYPT	SECP256R1	ED255519 SECP256R1 RSA – PKCS1
Smart contract runtime	EVM	Sandboxed in Docker containers	Deterministic JVM
Upgradeable smart contract	Possible with some patterns, not inherently supported	Allowed via upgrade transactions	Allowed via administrator privileges and auto update allowed under administrative checks
Tokenization support	Flexible— inherited from public Ethereum standards	Programmable	Corda token SDK

* TPS results for Quorum are based on
<https://arxiv.org/pdf/1809.03421.pdf>.

∞ TPS results for Hyperledger Fabric are based on
<https://hyperledger.github.io/caliper-benchmarks/fabric/performance/2.0.0/nodeContract/no-deSDK/submit/empty-contract/>.

@ TPS results for Corda are based on <https://www.r3.com/corda-enterprise/>.

This comparison may also be used for the quick evaluation of these platforms and suitability for an enterprise use case.

Now that we've examined some requirements, architectures, and use cases of enterprise blockchain, let's briefly describe some of the challenges that still face this technology in practice.

Enterprise blockchain challenges

While enterprise blockchains have addressed core enterprise requirements (privacy, performance, and governance) to some extent, there are still more challenges that need to be addressed. There is significant progress being made toward solving these issues. However, there is still a lot of work required to be done to adequately address these limitations. Some of these limitations are listed as follows.

Interoperability

Blockchain solutions are built by different development teams with different targets requirements. As a result, there are now many different types of blockchains, ranging from cryptocurrency public blockchains to application-specific blockchains, which are developed for a single business application. Data exchange between these chains is a crucial concern. While blockchain networks continue to grow independently, integration and interoperability between these chains remains a big concern.

This problem also stems out from lack of standardization. If a standard specification is available, then all chains following that standard will become compatible automatically. However, what about those chains that are already deployed in production, including public chains? How do we achieve interoperability between them? Many business use cases have the requirement to get data from one organization to another or from one

blockchain network to another. This is also true in the case of data exchange requirements between a public blockchain and consortium network.

Lack of standardization

Lack of standardization is a commonly highlighted concern in enterprise blockchains and generally in the blockchain ecosystem. Traditional systems are mostly developed in line with standards defined by standards bodies such as NIST FISMA standards for information security.



The **Federal Information Security Management Act (FISMA)** of 2002 is a United States federal law that requires all federal agencies to develop, document, and implement an agency-wide information security program. FISMA was amended in 2014 as the Federal Information Security Modernization Act. More on FISMA here:

<https://www.congress.gov/bill/113th-congress/senate-bill/2521>.

Standards are also essential to achieving interoperability. Several initiatives have been taken to address the challenges mentioned here and also to standardize enterprise blockchain platforms. A leading organization in this field is EEA, Saving a couple of lines here: deleted section heading and joined two small sections. Please link up the sentence on the previous page with this one: "...in this field is EEA, a standards organization run by its members that aims to develop enterprise blockchain specifications.



EEA regularly release technical specifications that can be downloaded at the link provided here:

<https://entethalliance.org/technical-specifications/>.

EEA's official website is <https://entethalliance.org>.

Compliance

There are various compliance requirements, in almost all industries—especially finance, law, and health—where strict guidelines and rules have been mandated by regulatory authorities. Enterprise systems are expected to conform to these requirements. Blockchain initially started with a different focus of building an electronic cash system with egalitarian philosophy. Still, surely enterprises have a different vision and mindset. Some of the regulations include GDPR, SOX, and PCI, which define different requirements for an enterprise to conform to. Also, compliance with technical standards such as the NIST FISMA standard for information security is necessary.

There are also jurisdiction issues. A distributed network with geographically dispersed locations may require different legal requirements to be met in various jurisdictions. For example, a specific type of cryptography may be allowed by law in the US but not in Cuba.

Business challenges

From a business perspective, cost, funding, and governance are some of the challenges that need to be met. If building and implementing an enterprise blockchain solution is prohibitively costly, then the business stakeholder might prefer traditional enterprise systems. Also, from an operational perspective, training staff to operate blockchain solutions might be a concern. We can categorize all blockchain-related costing, funding, and economics under an umbrella term that we call **Enterprise Blockconomics**. Enterprise Blockconomics, (blockchain economics) can be defined as a study of cost and cost models in the context of enterprise blockchains.

With that, we have covered a lot of background material and developed an understanding of the enterprise blockchain requirements, architecture, and challenges. Let's now dive into some examples of enterprise blockchain platforms. First, we'll discuss Corda and then Quorum.

Corda

Corda is not a blockchain by definition because it does not use blocks for batching transactions. Still, it is a distributed ledger, and provides all benefits that a blockchain can. Traditional blockchain solutions have the concept of transactions that are bundled together in a block, and each block is linked back cryptographically to its parent block, providing an immutable record of transactions. This is not the case with Corda.

Corda has been designed entirely from scratch with a new model for providing all blockchain benefits, but without a traditional blockchain with blocks. It was developed initially for the financial industry to solve issues arising from each organization managing its own local ledgers. This means that each organization has its own "view of truth." This situation often leads to contradictions and operational risk. Moreover, data is also duplicated at each organization, which results in an increased cost of managing individual infrastructures and complexity. These are the types of problems within the financial industry that Corda initially aimed to resolve by building a decentralized database platform. Initially, Corda only focused on financial applications but now has expanded into other areas such as government, healthcare, insurance, and supply chains.

Corda has two implementations, **Corda enterprise** and **Corda open source**. Both Corda enterprise and Corda open source are interoperable and compatible with one another, and have the same features. However, the enterprise version is more focused on enterprise requirements. It is a commercial version of the Corda platform targeting business requirements such as privacy, security, and performance. Also, it includes Corda firewall, high availability nodes and notaries, and support for hardware security modules. It provides an enterprise-grade platform for use in businesses.

The Corda platform, on the other hand, allows direct transactions between businesses. This network provides benefits such as privacy and ready sharing of data, which results in the reduction of complexity and costs.

The Corda source code is available at <https://github.com/corda/corda>. It is written in a language

called **Kotlin**, which is a statically typed language targeting the **Java Virtual Machine (JVM)**.

Architecture

The main components of the Corda platform include the Corda network, state objects (contract code and legal prose), transactions, consensus, and flows. We will explore these in more detail now.

Corda network

The Corda network is defined as a fully connected graph. It is a permissioned network that provides direct P2P communication on a "need to know" basis. Unlike other traditional blockchains/distributed ledgers, there is no global broadcast or a gossip protocol. The communication occurs directly on a point to point basis between interested parties. Peers or nodes communicate using the AMPQ serialization protocol. A service called network map service is responsible for publishing a list of peers.

A high-level architecture diagram of the Corda network is shown here, showing different components:

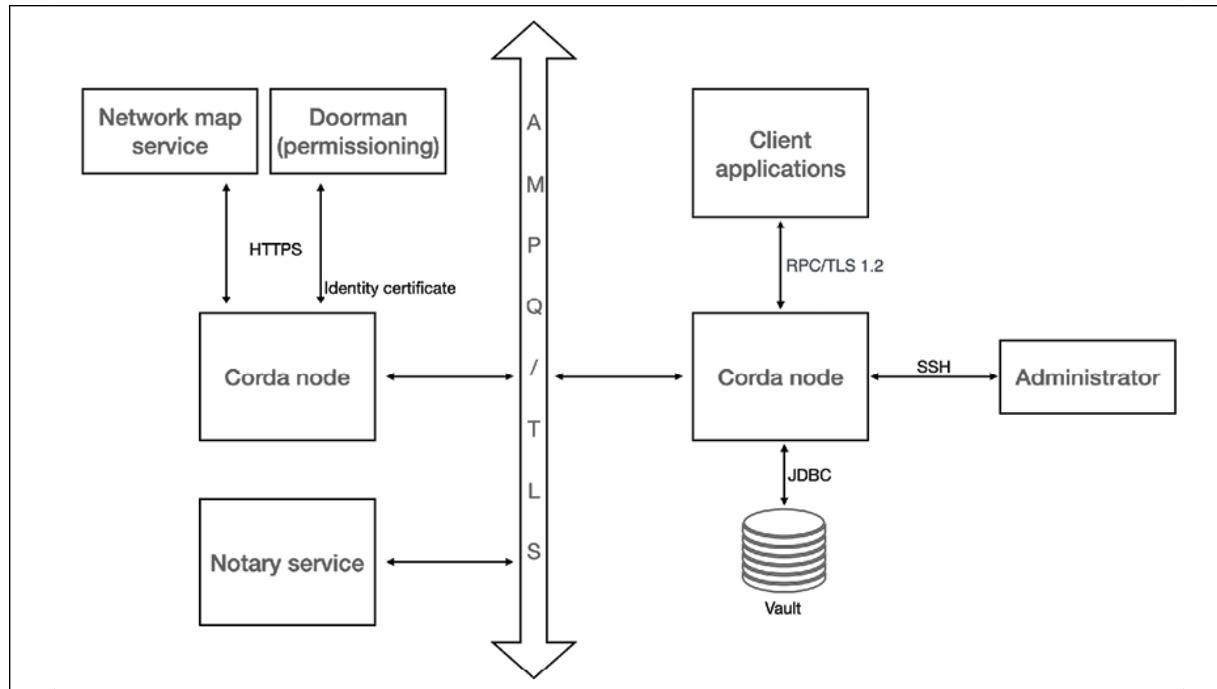


Figure 20.5: Corda high-level network architecture

State objects

State objects represent the smallest unit of data that constitute a financial agreement. These objects are digital documents that describe and fully capture all information about a shared agreement between parties. They are created or deleted as a result of transaction execution. State objects refer to the contract code and legal prose.

Legal prose is optional and provides legal binding to the contract. However, contract code is mandatory to manage the state of the object. It is required to provide a state transition mechanism for the node, according to the business logic defined in the contract code. State objects contain a data structure that represents the current state of the object. A state object can be either *current* (live) or *historic* (no longer valid).

For example, in the following diagram, a state object represents the current state of the object. In this case, it is a simple mock agreement between **Party A** and **Party B** where **Party A** has paid **Party B** 1,000 GBP. This represents the current state of the object; however, the referred contract

code can change the state via transactions. State objects can be thought of as a state machine, which is consumed by transactions to create updated state objects:

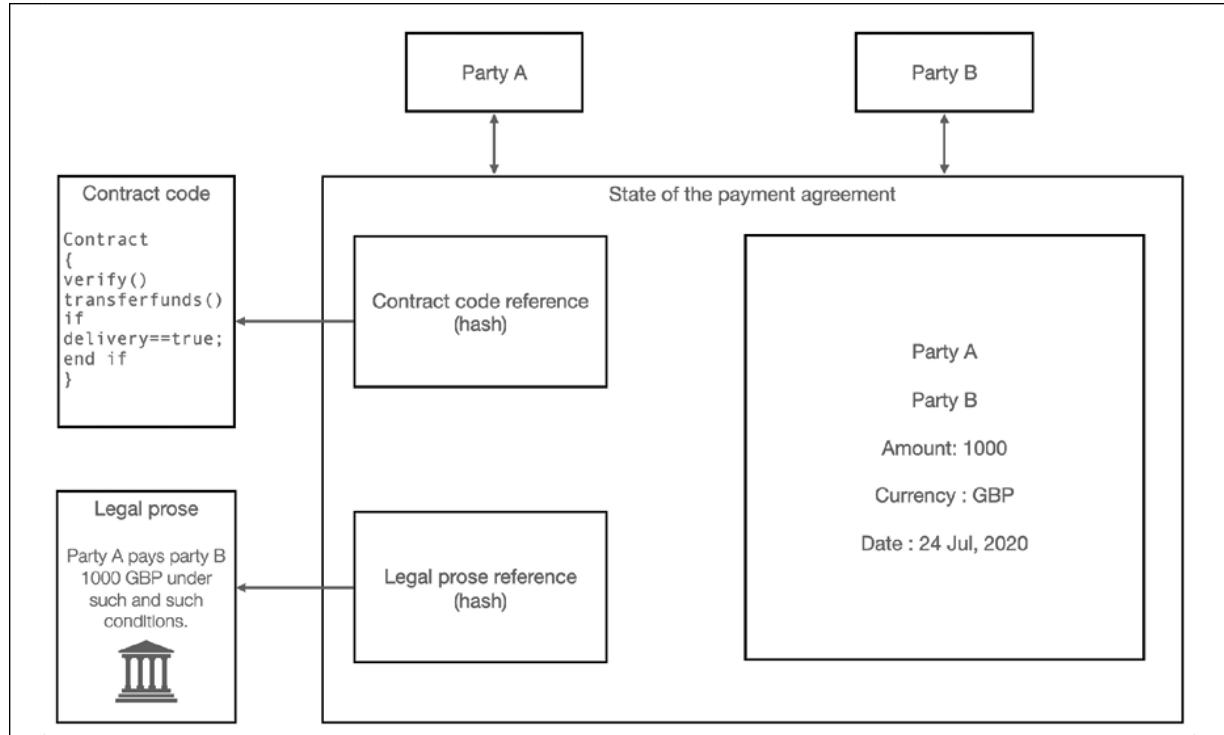


Figure 20.6: An example state object

States define the facts regarding which the agreement is achieved.

Transactions

Transactions are used to perform transitions between different states. For example, the state object shown in the preceding diagram is created as a result of a transaction. Corda uses a Bitcoin-style UTXO-based model for its transaction processing. The concept of state transition by transactions is the same as in Bitcoin. Similar to Bitcoin, transactions can have single, multiple, or no inputs, and single or multiple outputs. All transactions are digitally signed.

Moreover, Corda has no concept of mining because it does not use blocks to arrange transactions in a blockchain. Instead, notary services are used to

provide a temporal ordering of transactions. In Corda, new transaction types can be developed using JVM bytecode, which makes it very flexible and powerful.

Consensus

The consensus model in Corda is quite simple and is based on notary services, which will be discussed in the *Corda* section of this chapter, under *Components*. The general idea is that the transactions are evaluated for their uniqueness by the notary service and, if they are unique (that is, unique transaction inputs), they are signed by consensus services as valid. There can be single or multiple clustered notary services running on a Corda network. Various consensus algorithms like **PBFT** or **Raft** can be used by notaries to reach consensus, but the core idea is to check whether a proposed transaction is a valid update for the ledger or not.

There are two types of consensus in Corda: validity consensus and uniqueness consensus.

Validity consensus

This ensures that:

- The contract of every input and output state accepts the proposed transaction
- The transaction possesses all required signatures

Note that this process doesn't just verify the proposed transaction itself. It also verifies every previous transaction in the entire chain of the transactions. In other words, it validates all transactions behind this transaction that ultimately resulted in the creation of the inputs of the proposed transactions. This is called "walking the chain".

Uniqueness consensus

This provides validity consensus over the state's validity, and consensus over state uniqueness. The first mechanism is concerned with the validation

of the transaction, ensuring that all required signatures are available and the states are appropriate. The second mechanism is a means to detect double-spend attacks and ensures that a transaction has not already been spent and is unique. Uniqueness consensus is only checked by a notary service. In contrast, validity consensus is verified by each required signer before signing the transaction. This ensures that none of the input of a proposed transaction has already been spent (consumed) in another transaction.

Flows

Flows in Corda are a novel idea that allows the development of decentralized workflows. All communication on the Corda network is handled by these flows. These are transaction building protocols that can be used to define any financial flow of any complexity using code. Flows run as an asynchronous state machine, and they interact with other nodes and users. During their execution, they can be suspended or resumed as required.

CorDapps

The core model of Corda consists of state objects (data), transactions, and transaction protocols that, when combined with contract code (allowed operations), APIs, wallet plugins, and user interface components, result in constructing a Corda distributed application.

A **CorDapp (Corda Distributed Application)** is a distributed application that runs on a Corda network. It allows nodes on the network to reach an agreement regarding updates to the ledger. For this purpose, flows are defined that describe a routine for the node to execute.

Smart contracts in Corda are written using Kotlin and/or Java. The code is targeted at the JVM. The JVM in Corda has been modified to achieve deterministic results of the execution of JVM bytecode. There are three main components in a Corda smart contract, as follows:

- **Executable code**, which defines the validation logic to validate changes to the state objects.
- **State objects** represent the current state of a contract and can either be consumed by a transaction or produced (created) by a transaction.
- **Commands** are used to describe the operational and verification data that defines how a transaction can be verified.

In this section, we saw that a CorDapp is a distributed application that can be developed in Kotlin or Java and runs on a Corda network. CorDapps provide a mechanism for nodes to reach an agreement. Next, we'll explore some of the components required in the Corda network.

Components

The Corda network has multiple components. These components are described as follows.

Nodes

A **node** is a JVM runtime with a unique identity on the network. It has a network layer that allows interaction with other nodes and an RPC interface that will enable the node's owner to interact with it. A node hosts **CorDapps** and Corda services.

Nodes in a Corda network operate under a trustless model and are run by different organizations. Each node has an agreed and verified IP address. IP addresses belong to organizations who run nodes and also go through a KYC process. Nodes run as part of an authenticated P2P network and communicate directly with one another using the **Advanced Message Queuing Protocol (AMQP)**. AMQP is an approved international standard (ISO/IEC 19464) and ensures that messages across different nodes are transferred safely and securely. AMQP works over **Transport Layer Security (TLS)** in Corda, thus ensuring privacy and integrity of data communicated between nodes.

Nodes also make use of a local relational database for storage. Messages on the network are encoded in a compact binary format. They are delivered and managed by using the Apache Artemis message broker (Active MQ).

A node can serve as a network map service, notary, oracle, or a regular node. The following diagram shows a high-level view of a Corda node:

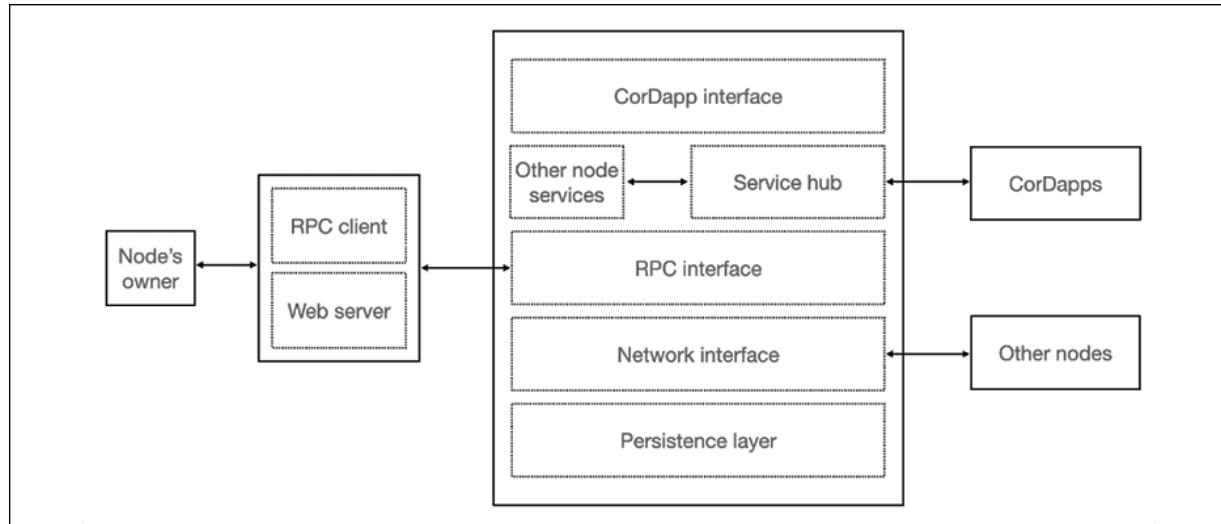


Figure 20.7: Simplified Corda node architecture

The key elements of a Corda node are:

- A **Persistence layer**, which consists of a vault and storage service. It is responsible for storing data in a local SQL database.
- A **Network interface**, which enables communication with other nodes as part of running a flow.
- An **RPC interface** is responsible for providing interface for interaction with the node's owner via RPC clients and/or web servers. It exposes a number of RPC operations for node and flow management, administration, and retrieving information about flows, nodes, and the network.
- A **Service hub** enables flows in the node to call other services (utilities in a node) such as storage and vault services.
- A **CorDapp interface** and provider, which allows the extending of a node with CorDapps.

Next, we'll discuss the permissioning element of Corda, which controls network membership.

The permissioning service

The permissioning service **Doorman** is used to provision TLS certificates for security. Participants are required to have a signed identity issued by a root certificate authority to participate in the network. Identities are required to be unique on the network. The permissioning service is used to sign these identities, and the naming convention applied to recognize participants is based on the X.500 standard. This ensures the uniqueness of the name. Doorman is responsible for performing KYC and permitting nodes to join and use the network.

Network map service

This service is used to provide a network map in the form of a document of all nodes on the network. This service publishes IP addresses, identity certificates, and a list of services offered by nodes. All nodes announce their presence by registering to this service when they first start up, and when a connection request is received by a node, the presence of the requesting node is checked on the network map first. Put another way, this service resolves the identities of the participants to physical nodes.

Notary service

In traditional blockchains, mining is used to ascertain the order of blocks. In Corda, notary services are used to provide transaction ordering and timestamping services. There can be multiple notaries in a network, and they are identified by composite public keys. Notaries can use different consensus algorithms like **BFT** or **Raft**, depending on the requirements of the applications. Notary services sign the transactions to indicate the validity and finality of the transaction, which is then persisted to the database. Notaries can be run in a load-balanced configuration to spread the load across the nodes for performance reasons. The nodes are recommended to be run physically closer to the transaction participants to reduce latency.

Due to the distributed nature of distributed ledgers, there is no exact universal time in the network. Therefore, it is always difficult to say when precisely an event occurred. In Corda, time windows can be used to assert that a transaction happened after a specific event or time. A transaction in Corda can include a time window that enforces timing restrictions on the commitment of the transaction. This means that the transaction can only be notarized and committed during the time window specified in the transaction. The notary can serve as a timestamping authority. Before notarizing a transaction, it can verify that a transaction occurred during a specified time window.

Time windows allow us to build scenarios that can indicate whether a transaction happened before an event or after a specific time. This can be useful in constructing scenarios where we can say, for example, a transaction happened after the expiry of a contract or after some event or on a specific day. It is important that a notary has an accurate and valid time feed for this function to work correctly.

Oracle service

Oracle services either sign a transaction containing a fact if it is true or can themselves provide factual data. They allow real-world feed into the distributed ledgers. Oracles were discussed in *Chapter 10, Smart Contracts*.

Transactions

Transactions in a Corda network are never transmitted globally but in a semi-private network. They are shared only between a subset of participants who are related to the transaction. This is in contrast to traditional blockchain solutions like Ethereum and Bitcoin, where all transactions are broadcast to the entire network globally. Transactions are digitally signed and either consume state(s) or create a new state(s).

Transactions on a Corda network are composed of the following elements:

- **Input references:** This is a reference to the states the transaction is going to consume and use as an input.

- **Output states:** These are new states created by the transaction.
- **Attachments:** This is a list of hashes of attached ZIP files. ZIP files can contain code and other relevant documentation related to the transaction. Files themselves are not made part of the transaction; instead, they are transferred and stored separately.
- **Commands:** A command represents the information about the intended operation of the transaction as a parameter to the contract. Each command has a list of public keys, which represents all parties that are required to sign a transaction.
- **Signatures:** This represents the signature required by the transaction. The total number of signatures required is directly proportional to the number of public keys for commands.
- **Type:** There are two types of transactions, namely normal and notary changing. Notary changing transactions are used for reassigning a notary for a state.
- **Timestamp:** This field represents a bracket of time during which the transaction has taken place. These are verified and enforced by notary services. Also, it is expected that if strict timings are required, which is desirable in many financial services scenarios, notaries should be synchronized with an atomic clock.
- **Summaries:** This is a text description that describes the operations of the transaction.

Vaults

Vaults run on a node and are similar to the concept of wallets in Bitcoin. As the transactions are not globally broadcasted, each node will have only that part of data in their vaults that are considered relevant to them. Vaults store their data in a standard relational database and, as such, can be queried by using standard SQL. Vaults can contain both on the ledger and off ledger data, meaning that it can also have some part of the data that is not on the ledger. Vaults keep track of spent and unspent states.

Other tools

Other important tools in Corda are described as follows.

Application firewall

This is an enterprise feature that operates like a reverse proxy and helps to run Corda behind corporate firewalls. It is an optional component that acts as a reverse proxy for Corda nodes residing behind the **demilitarized zone (DMZ)**, which is usually the case in an enterprise.



DMZ is a network security technique commonly used to protect the internal enterprise network from external attacks. For this purpose, an isolated sub-network is created between the external network (usually the internet) and the internal network, which hosts externally facing servers. On the other hand, the internal network stays behind the firewall. Connectivity is allowed from the DMZ to the external network only, which results in reducing the possibility of direct attacks on the internal network.

The application firewall helps to minimize the amount of code that runs on a network's DMZ, thus reducing the attack surface. It enables secure AMQP/TLS1.2 interaction with Corda peer nodes outside the corporate network. It works as an application-level firewall for internet-facing endpoints on the Corda network.

The firewall has two components, called the bridge and the float. The *bridge* handles outbound connections, whereas the *float* handles the inbound connection. A single instance of the Corda firewall can serve multiple nodes.

Network simulator

This is a visualization tool that shows traffic between nodes and the network set up via a map service.

Node explorer

This is another visualization tool that allows us to view transaction information.

Load tester

This, as the name suggests, helps to stress test nodes under various extreme scenarios.

Transaction flow

In this section, we'll explore how, when a request is sent to a node, Corda accepts the request and responds. The flow process can be summarized as follows:

1. An HTTP request via API or RPC call is made directly to flows in Corda. This is a request to start a transaction.
2. Start a flow. Flows contain the business logic.
3. Perform initial steps such as transaction building and verification.
4. Obtain counterparty signatures.
5. Sign the transaction and check signatures.
6. Obtain notarization/finality.
7. Record the transaction and store it in persistent storage and Vaults.
8. Respond via an HTTP response through API.

This process can be visualized using the following diagram:

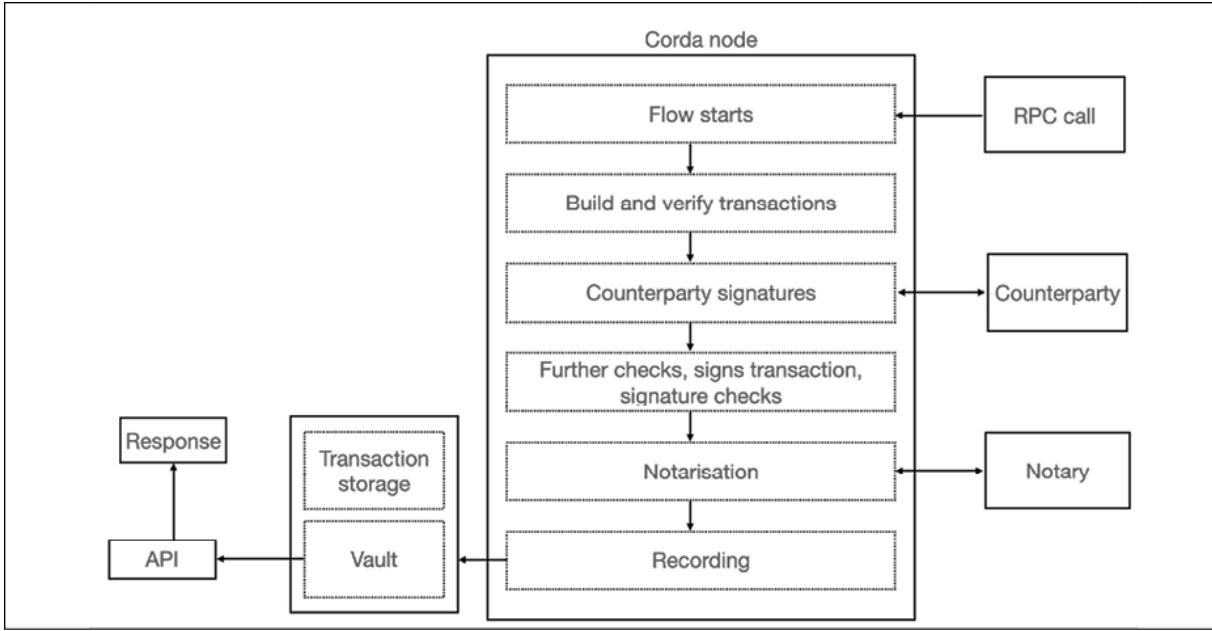


Figure 20.8: Node high-level flow

A CorDapp is a functional aspect of the Corda network and contains the business logic for a use case. The vault (database) is used to store states via JDBC. Corda nodes communicate using AMQP/TLS1.2. Node administrators use SSH to interact with the node. An identity certificate is issued to a node via the Doorman service, which works over HTTPS. Corda nodes discover other nodes on the network via the network map service over HTTPS. Corda nodes and the Corda firewall service make sure that a certificate is currently valid by checking CRLs, which is performed over an HTTP or HTTPS connection.

Now, let's see how we can set up a development environment to explore Corda's code base. We have not provided any specific development application but have provided links where official examples are available.

Corda development environment

The development environment for Corda can be set up easily using the following steps. The required software consists of the following:

- JDK 8 (8u131), which is available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
- IntelliJ IDEA Community edition, which is free and available at <https://www.jetbrains.com/idea/download/>.
- H2 database platform independent ZIP, and is available at <http://www.h2database.com/html/download.html>.
- Git, which is available at <https://git-scm.com/downloads>.
- Kotlin language, which is available for IntelliJ. More information can be found at <https://kotlinlang.org/>.
- Gradle is another component that is used to build Corda. It is available at <https://gradle.org>.

Once all these tools are installed, smart contract development can be started:

- CorDapps can be developed by utilizing an example template available at <https://docs.corda.net/docs/corda-os/4.5/hello-world-template.html>
- Detailed documentation on how to develop contract code is available at <https://docs.corda.net/>.

Corda can be cloned locally from GitHub using the following command:

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

Once entered in the operating system terminal, this command will clone the repository into a local directory named `corda`.

Once the repository is cloned, it can be opened in an IntelliJ development environment for further development. There are multiple samples available in the repository, such as a bank of Corda, interest rate swaps demo, and traders' demo. You may find it useful to explore these demos to understand how CorDapps are developed. These samples are available under the

`/samples` directory, under `corda`, and they can be explored using IntelliJ IDEA IDE.

With this, we've completed our introduction to the Corda platform. Now, we'll cover Quorum, another popular blockchain platform, in detail.

Quorum

Quorum is an open source enterprise blockchain platform. It is a lightweight fork of **Ethereum**. Quorum not only benefits from the innovation and research being done in the upstream public Ethereum (Geth) project, but also many excellent enterprise features have been introduced in Quorum. These enterprise features primarily focus on providing enterprise-grade privacy, performance, and permissioning (access control).

In this section, we'll explore Quorum's architecture in detail and see a practical example of how to set up a Quorum network. First, let's take a look at Quorum's architecture.

Quorum addresses three fundamental issues in public blockchains, which makes Quorum an excellent choice for enterprise use cases:

- Privacy
- Performance
- Enterprise governance

At a high level, the Quorum architecture consists of nodes and their associated privacy managers. The Quorum node is a modified and enhanced version of public `geth` that supports private transactions. Quorum nodes communicate via HTTPS with privacy managers, who, in turn, are responsible for providing privacy by storing the payload in encrypted format in their local storage. These Quorum nodes maintain the public and private states separately.

Architecture

The Quorum architecture can be visualized using the following diagram:

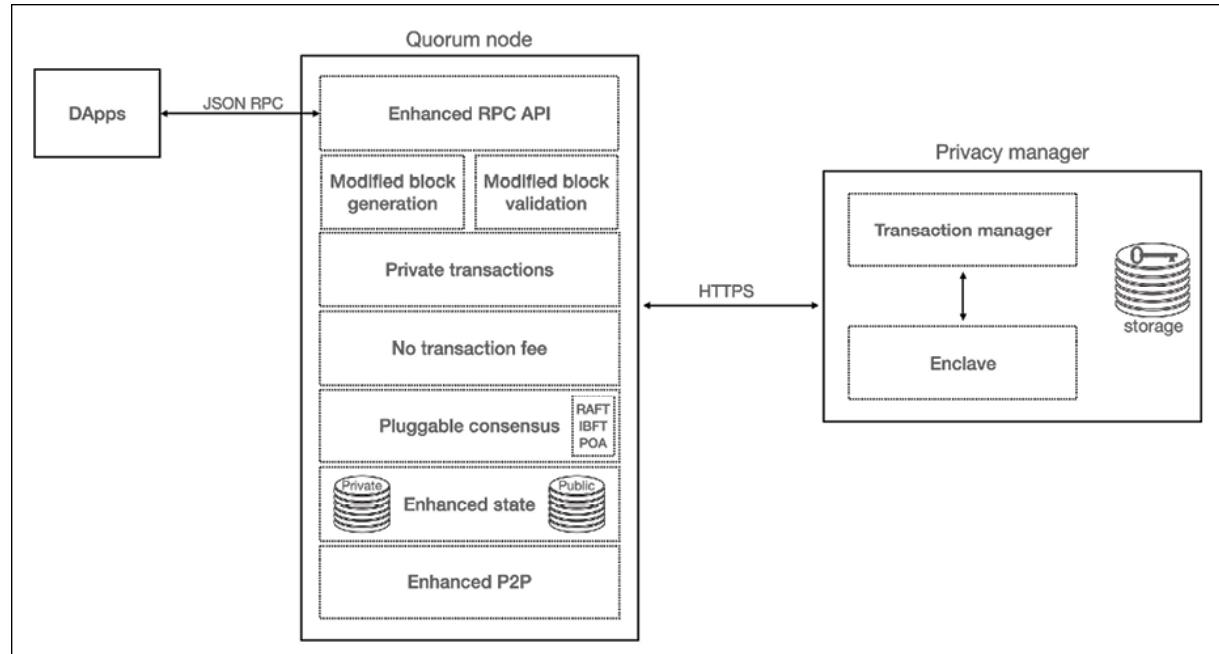


Figure 20.9: Quorum architecture

There are a number of changes that have been made to Quorum client to make it suitable for enterprise use cases. We'll provide an overview of each of those changes here.

Enhanced P2P

This layer is modified to allow connections only between authorized nodes.

Enhanced state (private and public)

In addition to the public state trie, there is an additional Merkle Patricia state trie for the private state.



In Ethereum, a **Modified Merkle Patricia trie (MPT)** is a data structure that is used to provide a cryptographically authenticated key-value store.

More technical details on this are available here:
<https://eth.wiki/en/fundamentals/patricia-tree>.

Pluggable consensus

Quorum supports multiple consensus algorithms such as IBFT, **Proof of Authority (PoA)**, and Raft. Quorum, being an enterprise blockchain, does not need a PoW type of consensus.

No transaction fees

The pricing of gas has been set to zero as there is no need for transaction fees in consortium networks.

Private transactions

Private transactions are identified using [37] or [38] as the `v` value in the transaction. The transaction creation mechanism is modified to enable replacing transaction data (input) with the hash of the encrypted payload.



Remember that in standard Ethereum, as we learned in *Chapter 11, Ethereum 101*, a transaction's `v` value can be either [27] or [28], but in Quorum, that has been changed to [37] or [38] to indicate private transactions. Public transactions are still identifiable by a [27] or [28] `v` value, whereas private transactions are identifiable with a `v` value of either [37] or [38].

Modified block generation mechanism

The logic for generating blocks is modified with a new check for the global public state root instead of the global state root.

Modified block validation mechanism

Block validation logic is modified to replace the global state root check with a global state root check *for public state*—otherwise known as a global public state root.

Enhanced RPC API

Quorum supports additional RPC APIs that help us interact with the enhanced enterprise features of Quorum, such as permissioning and consensus mechanisms.

Now, we'll discuss **privacy manager**, which consists of two components: a transaction manager and an enclave.

Privacy manager

Privacy manager is an off-chain mechanism that provides transaction confidentiality. It is paired on a one-to-one basis with a Quorum node. It allows Quorum nodes to share the transaction payload securely between authorized participants. It is composed of two components: the **transaction manager** and the **enclave**.

Transaction manager

The **transaction manager** is responsible for:

- Storing encrypted transaction payloads
- Managing access to encrypted transaction payloads
- Propagating encrypted payloads to other transaction managers on the network
- Discovering other transaction managers on the network

Quorum has developed two transaction managers. Initially, a Haskell implementation called **Constellation** was made available. However, it is no longer actively developed in favor of a more feature-rich, Java-based

privacy manager called **Tessera**. Tessera is developed in Java. It is used for storage, encryption, decryption, and propagation of private transaction data.

Enclave

An **enclave** is an isolated and independent element that provides cryptography services for transaction payload encryption and decryption. It can only be associated on a one-to-one basis with its own transaction manager.

In order to achieve all the desired privacy features, several cryptographic protocols and primitives have been used in the Quorum platform. We'll provide a quick overview of these in the next section.

Cryptography used in Quorum

Quorum inherits its cryptography stack from public **Ethereum**. It includes standard ECDSA signatures, AES CTR cryptography for wallets and DEVP2P, and Keccak256 hash functions. Privacy manager makes use of Curve25519, **Elliptic-curve Diffie-Hellman (ECDH)**, poly1305, and Xsalsa20 cryptographic operations to provide confidentiality guarantees required by private transactions in Quorum.

Now, we'll explore how each of the main facets—privacy (confidentiality), enterprise-grade membership control (access control, and the permissioning mechanism), and performance—are achieved in Quorum.

Privacy

Quorum supports both private and public transactions. For private transactions, it uses a mechanism called private transaction manager, which is an off-chain component to facilitate the confidentiality of transactions. We will now describe how private transactions work in Quorum.

Suppose there are three parties: *A*, *B*, and *C*. Parties *A* and *B* are privy to a transaction, while party *C* is not. We'll now explore how the private transaction is generated and flows between parties and propagated on the network, while maintaining confidentiality. Let's call this transaction *transaction AB*:

1. To begin private *transaction AB*, party *A* creates a transaction and signs it before sending it to their Quorum node, node *A*. The transaction is composed of a transaction payload and the public key of the intended recipient. This list of public keys of intended recipients is maintained in the *PrivateFor* list. It can be a single public key or multiple, depending on the requirements.



There are two signing mechanisms in Quorum. For public transactions, an EIP55-based mechanism is used and for private transactions, an Ethereum Homestead signing mechanism is used. Transactions in Quorum can also be signed independently, without using Quorum's signing mechanism.

2. Quorum node *A* sends the transaction to transaction manager *A* for processing.
3. Transaction manager *A* makes an encryption request to its enclave, to encrypt the transaction payload.
4. Party *A*'s enclave encrypts the transaction payload and sends it to transaction manager *A*.
5. Transaction manager *A* stores the transaction payload and sends it to other transaction managers; in this case, *B*.
6. A transaction hash is returned to Quorum node *A* by transaction manager *A*, which replaces the original transaction payload with the hash and changes the value `v` of the transaction to `37` or `38` to indicate that the transaction is private.
7. The transaction propagates to other nodes via the normal Ethereum P2P protocol.
8. The block that contains *transaction AB* finalizes and propagates on the network between all nodes (*A*, *B*, and *C*).

9. When the block containing the transaction is received by the Quorum nodes, they recognize the transaction as private because the `v` value of the transaction is either `37` or `38`. If the transaction is identified as private, the Quorum nodes will inquire their respective transaction managers to figure out if they are party to the transaction or not. This is achieved by finding out that if there is an entry available in the database with the transaction hash. Here, parties *A* and *B* will have the transaction hash stored in the transaction manager, whereas party *C* will not.
10. The transaction managers of parties *A* and *B* make a transaction payload decryption request to their respective enclaves.
11. The enclaves of party *A* and *B* decrypt the private transactions.
12. When the transaction managers receive the decrypted payload back from their enclaves, they send this data to their respective Quorum nodes. In our example, the transaction managers of parties *A* and *B* will send the transaction payload to their respective Quorum nodes, *A* and *B*. The Quorum nodes will execute the transaction (contract) via the EVM and update their private state database accordingly. The transaction manager of party *C* will return a message back to its Quorum node, indicating that it is not privy to (or a recipient of) the transaction. This means that the Quorum node of party *C* will simply ignore the transaction.

This process can be visualized using the following diagram:

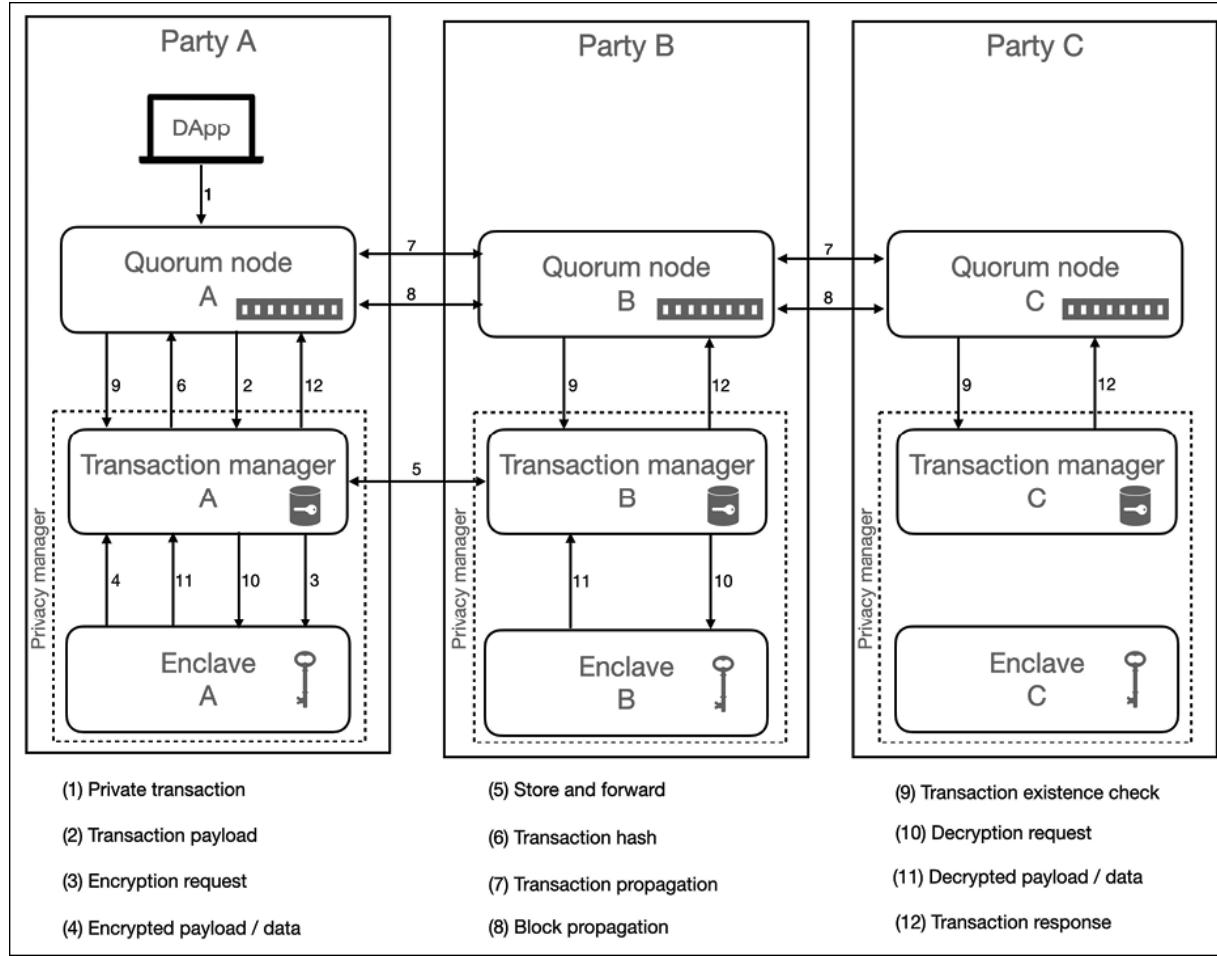


Figure 20.10: Quorum private transaction flow

Let's now expand on *step 4* from the *achieving privacy* process, enclave encryption, as it consists of a number of substeps that are performed within the enclave.

Enclave encryption

When an **enclave** receives a transaction, it performs several steps to encrypt the transaction. All these steps are shown here and are performed by the enclave as part of the transaction payload encryption process:

1. Generates a symmetric key for transaction.
2. Generates two random nonces.

3. Encrypts the transaction payload and one of the nonces with the symmetric key generated in the first step.
4. Encrypts the transaction key for transactions for each recipient. For this purpose, the following steps are performed:
 - a. Generates a shared symmetric key by utilizing ECDH. ECDH will use the sender's private key and the receiver's public key.
 - b. Encrypts the symmetric key for transaction for each receiver separately using the newly generated shared symmetric key and the second nonce.
 - c. This step will be repeated for each recipient.
5. Finally, the transaction manager receives the encrypted transaction payload, all the encrypted symmetric keys for transactions, both nonces, and the public keys of the senders and receivers.

Now, let's see how the transaction manager stores and propagates transactions to the other transaction manager. We'll now expand on *step 5* of the *achieving privacy* process mentioned previously.

Transaction propagation to transaction managers

A **transaction manager** performs the following steps after it receives an encrypted response back from the **enclave**:

1. It generates the SHA3-512 hash of the encrypted payload received from the enclave.
2. It stores the encrypted payload, the hash and encrypted symmetric key for the transaction, both nonces, and the public keys of the sender and receiver.
3. It sends, using HTTPS, the encrypted payload, the hash, and the encrypted symmetric key for the transaction using transaction manager *B*'s public key to transaction manager *B*.
4. Transaction manager *A* awaits an acknowledge message from transaction manager *B*. If acknowledgement is not received, the transaction will not be propagated to the network.

Let's now discover how the decryption process works in enclaves. This corresponds to *step 11* of the *achieving privacy* process.

Enclave decryption

The transaction payload decryption process starts when a transaction payload decryption request is made to an **enclave**. The enclave performs the following steps to decrypt a payload:

1. Derives the shared symmetric key. The key derivation process works as follows:
 - a. Party *A*, being the sender of the transaction, derives the shared symmetric key using its private key and receiver's public key.
 - b. Party *B*, being the recipient of the transaction, derives the shared symmetric key using its private key and the sender's public key.
2. Decrypts the symmetric key for transactions with the shared symmetric key, along with the encrypted payload and nonce, fetched from the database.
3. Decrypts the transaction data with the symmetric key for transactions and the encrypted data and nonce fetched from the database.
4. Finally, the decrypted private transaction data is sent to the transaction manager.



The enclave being separated from the transaction manager addresses **separation of concerns** and allows parallelization in order to improve performance. Separation of concerns is a form of abstraction that allows the segregation of different components of a computer software into separate and distinct units, so that each unit addresses a separate concern. This simplifies design and helps to modularize the software.

The overall process (*enclave encryption, transaction manager storage and propagation to other nodes, and enclave decryption*), corresponding to *steps 4, 5, and 11* in Figure 20.10, can be visualized in the following diagram:

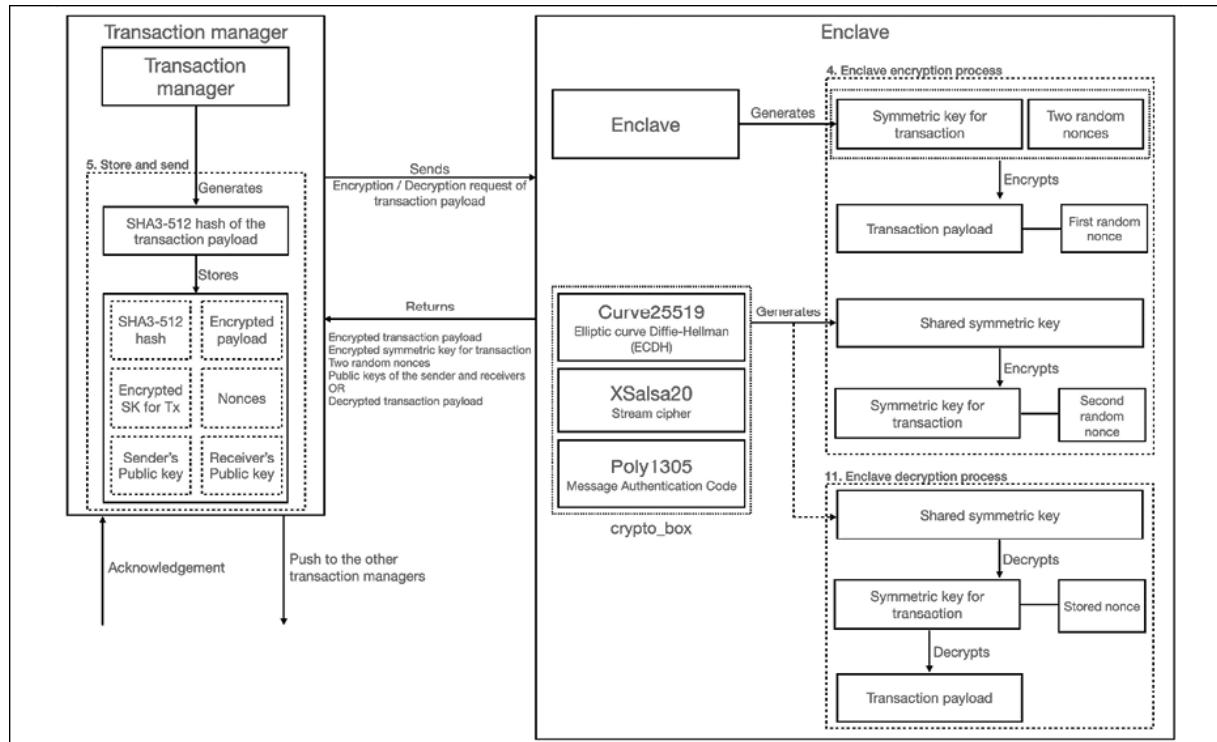


Figure 20.11: Enclave and transaction manager high-level architecture

We've covered quite a lot of theory here, but with this, we have now completed our introduction to Quorum private transactions. Next, we'll explore how access control works in Quorum.

Access control with permissioning

As an enterprise blockchain platform, **Quorum** comes with an enterprise-grade access control mechanism that manages network membership for consortium members. It is implemented in smart contracts written in the **Solidity** language. It supports many features that a typical enterprise-grade permissioning mechanism would. It includes features to support the management of nodes, account-level permissioning, and support for voting-driven decision-making for permissioning actions.

The overall architecture of Quorum's permissioning mechanism can be visualized using the following diagram:

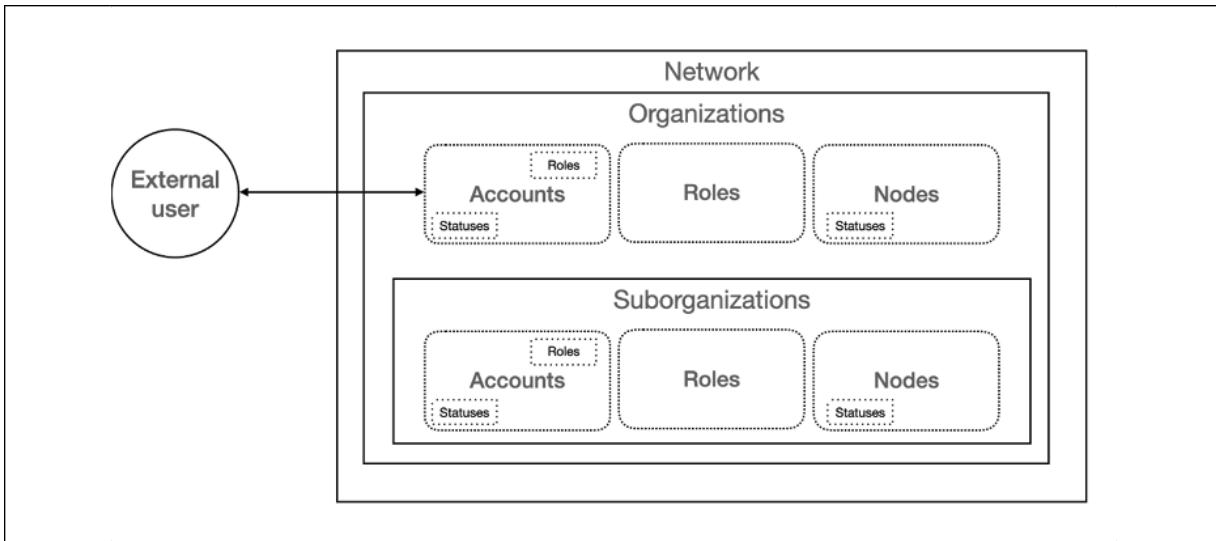


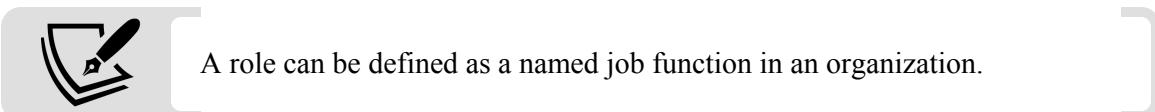
Figure 20.12: Quorum permissioning mechanism

This permissioning mechanism is inspired by a widely accepted industry standard called **Role-Based Access Control (RBAC)**. It is standardized by the **American National Standards Institute (ANSI)** and is used in most enterprise-grade software, such as operating systems and cloud systems.



The features that Quorum's permission mechanism supports are listed here:

- Role management
- Assignment of roles to accounts (subjects)
- Permissions management
- Node management



At a fundamental level, the permissions management model supports granular control over:

- Funds transfer
- Smart contract creation
- Smart contract execution

It controls which account can transfer funds, create smart contracts, or execute smart contracts. These permissions are used to create roles, which are then assigned to accounts to enforce permissions.

Permissions are divided into two broad categories: *account permissions*, which control what functions an account can perform, and *node permissions*, which control membership of nodes on the network.

Accounts can be assigned with different roles and statuses, whereas nodes can be assigned with statuses. Statuses for accounts include several statuses such as active, inactive, suspended, and blacklisted. Nodes can be in approved, deactivated, pending approval, or blacklisted states. The access types can be read-only, transact, contract deploy, and full access.

Now that we have covered how enterprise-grade permissioning is implemented in Quorum, let's now move on to another important aspect of enterprise blockchain: performance.

Performance

Performance is a vast subject and can mean different things in different scenarios. Mostly, it is merely a measure of how many transactions a system can perform in a given interval, usually measured in **transactions per second (TPS)**. It can also mean overall throughput of the system, quality of the service, and the ability to scale. From a business perspective, it could mean business performance. However, here, we will only deal with the TPS scenario and present some studies that have been performed to evaluate the performance of the Quorum blockchain.

The core idea behind performance enhancement in the Quorum blockchain is the usage of deterministic and fast consensus algorithms. Quorum supports various consensus mechanisms suitable for consortium networks. As compared to other public blockchain consensus mechanisms, the algorithms used in Quorum provide better performance.

Several evaluations have been made and reported regarding Quorum's performance. These studies have reported Quorum's performance to be as high as 2,500 TPS.



These studies are available here:

<https://scandiweb.com/blog/jpmorgan-s-quorum-blockchain-performance-testing>.

<https://arxiv.org/pdf/1809.03421.pdf>.

Now, we'll introduce the consensus mechanisms supported by Quorum.

Pluggable consensus

Quorum supports several consensus algorithms that can be plugged in, based on use case requirements. For example, if the requirement is simple crash-tolerance, users can choose Raft. If more security is required and Byzantine faults need to be handled, then users can choose IBFT. We have listed the consensus mechanisms available in Quorum here:

- **Raft**: A crash fault-tolerant consensus algorithm
- **IBFT**: A PBFT-inspired BFT algorithm
- **Clique: Proof of Authority (PoA)** inherited from public Ethereum

We discussed all these algorithms in detail in *Chapter 5, Consensus Algorithms*. You can review these topics in detail there.

Now that we have gone through several features and have built a theoretical understanding of various features of the Quorum enterprise blockchain, let's

now see how we can set up a Quorum network.

Setting up Quorum with IBFT

In this section, we will set up a **Quorum** network with four nodes, demonstrating how an IBFT network can be set up and carry out private transactions in Quorum. These steps can be performed manually, or we can use a tool called **Quorum Wizard** to quickly and easily spin up a Quorum network.



For details on the manual setup, you can refer to the detailed Quorum setup instruction on Quorum's official website here:
<https://docs.goquorum.consensys.net/en/latest/Tutorials/Creating-A-Network-From-Scratch/>.

We will use Quorum Wizard to set up our network.

Quorum Wizard

Quorum Wizard is a command-line tool written in **JavaScript** that enables users to create a local network of Quorum nodes quickly. It runs as an `npm` module and, as such, requires Node.js and NPM to run.

Node.js can be installed from this address:

<https://nodejs.org/en/>.

To check the current version installed, issue the following command:

```
$ npm -v
```

This command will produce the following output, indicating the installed version of node package manager, `npm`:

```
6.14.4
```

This command will show the currently installed version of `node`:

```
$ node -v
```

This will produce an output specifying the version of `node` you have installed:

```
v12.18.0
```

If the expected version number is displayed after executing both of these commands, then we are all set to install Quorum Wizard, which we'll describe in the next section.

Installing Quorum Wizard

Quorum Wizard can be installed using `npm install`. It's an excellent tool and cuts installation time down from hours to minutes:

```
$ npm install -g quorum-wizard
```

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

```
/usr/local/bin/quorum-wizard -> /usr/local/lib/node_modules/quor
+ quorum-wizard@1.1.0
added 155 packages from 143 contributors in 27.897s
```

Once installed, we can run it with the following procedure.

Running Quorum Wizard to create a new network

We will create a 4-node IBFT network in this example. Simply run Quorum Wizard and follow the steps as prompted:

```
$ quorum-wizard
```

Running Quorum Wizard will show the following text:

```
Welcome to Quorum Wizard!
This tool allows you to easily create bash, docker, and kubernetes files to start up a quorum network.
You can control consensus, privacy, network details and more for a customized setup.
Additionally you can choose to deploy our chain explorer, Cakeshop, to easily view and monitor your network.

We have 3 options to help you start exploring Quorum:
1. Quickstart - our 1 click option to create a 3 node raft network with tessera and cakeshop
2. Simple Network - using pregenerated keys from quorum 7nodes example,
   this option allows you to choose the number of nodes (7 max), consensus mechanism, transaction manager, and the option to deploy cakeshop
3. Custom Network - In addition to the options available in #2, this selection allows for further customization of your network.
   Choose to generate keys, customize ports for both bash and docker, or change the network id

Quorum Wizard will generate your startup files and everything required to bring up your network.
All you need to do is go to the specified location and run ./start.sh

(Use arrow keys)
> Quickstart (3-node raft network with tessera and cakeshop)
Simple Network
Custom Network
Exit
```

Figure 20.13: Quorum wizard options

Select **Simple network** and select all the options by using the up and down arrow keys, as shown here. Press Enter after a choice is made, which will move us on to the next question in Wizard:

```
Simple Network
? Would you like to generate bash scripts, a docker-compose file
? Select your consensus mode - istanbul is a pbft inspired algor
? Input the number of nodes (2-7) you would like in your network
? Which version of Quorum would you like to use? Quorum 2.6.0
? Choose a version of tessera if you would like to use private t
? Do you want to run Cakeshop (our chain explorer) with your net
? What would you like to call this network? 4-nodes-istanbul-tes
```

Once all the options have been selected, as shown in the preceding code snippet, the tool will download and install dependencies. This process will produce an output similar to the one shown here. For brevity, the full output is not shown:

```
Downloading dependencies...
.
.
Building network directory...
Generating network resources locally...
Building qdata directory...
Writing start script...
Initializing quorum...
Done
```

Finally, the output shown here is produced, indicating the success of the operation:

```
Quorum network created
Run the following commands to start your network:
cd network/4-nodes-istanbul-tessera-bash
./start.sh
A sample simpleStorage contract is provided to deploy to your ne
To use run ./runscript.sh public-contract.js from the network fc
A private simpleStorage contract was created with privateFor set
To use run ./runscript private-contract.js from the network folc
```

Now, change to the `4-nodes-istanbul-tessera-bash` directory, as shown here, and start the network:

```
$ cd network/4-nodes-istanbul-tessera-bash
$ ./start.sh
```

This will produce an output similar to the one shown here. Some of the output has been truncated for brevity:

```
Starting Quorum network...
.
.
.
All Tessera nodes started
Starting Quorum nodes
Starting Cakeshop
```

```
Cakeshop started at http://localhost:8999  
Successfully started Quorum network.
```

And that's it! We now have a Quorum network with four nodes running on the IBFT protocol. We can also visualize the network using **Cakeshop**, which we'll describe next.

Cakeshop

Cakeshop is a powerful visualization and administration tool with different features such as node management, block explorer, and contract management. Cakeshop is installed as part of the Quorum Wizard network creation process. Once the network runs successfully, we can browse to <http://localhost:8999>, where Cakeshop runs.

Cakeshop is shown in the following screenshot:

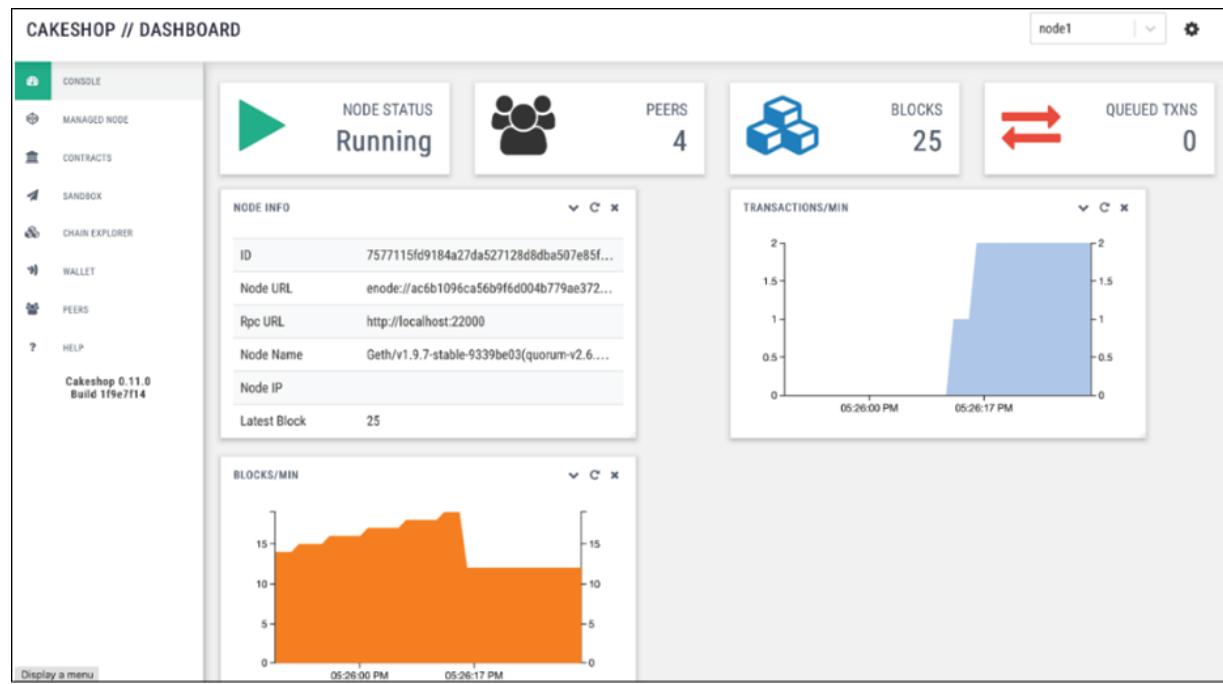


Figure 20.14: Cakeshop

Now, let's do an experiment to see how private transactions can be created and run on the Quorum network. Quorum Wizard has already created the

relevant scripts.

Running a private transaction

In this section, we will explore how private transactions can be created and executed on a Quorum network.

1. From within the `4-nodes-istanbul-tessera-bash` directory, run the command shown here:

```
$ 4-nodes-istanbul-tessera-bash ./runscript.sh private_cont
```

Once the preceding command runs, it will produce an output similar to the one shown here, indicating that the transaction has been sent and that a transaction hash

```
0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc7945  
6bf4 has been produced:
```

```
Contract transaction send: TransactionHash: 0xa58ec5e746612  
True
```

Here, we have basically executed a private transaction and deployed a private smart contract that is only visible to Node 1 and Node 2. Nodes 3 and 4 are not party to this transaction and they shouldn't be able to view the contents of the code or the values (state) of the contract. We will demonstrate in this example that this is indeed the case.

Next, we'll attach Geth to each node, as shown using the commands in the following sections, and run the console commands as shown in the examples for each node.

Node 1

In order to interact with `geth` via IPC, enter the following command in the operating system's Terminal. This will open the Geth JavaScript console,

where users can interact with the blockchain using several methods exposed by Geth:

```
$ geth attach qdata/dd1/geth.ipc
```

Once the `geth` console is open, enter the following command, which declares a variable named `abi`:

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData","c
```

Now, create a new contract instance using the command shown here. We can use this object to access all methods and events of the contract:

```
> var simpleContract = eth.contract(abi)
```

Next, we use the instance object created previously to get a complete abstraction of our contract by using the command shown here:

```
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303")
```

Now, we can use the `simple` object to access all the methods in our smart contract. In the following example, we're using the `get()` method to return the stored value in our contract:

```
> simple.get()
42
```

This command outputs `42`, which is our stored value.

Note that after each statement we enter in the `geth` console, except the last statement, `simple.get()`, we also see that a message of `Undefined` is displayed. Simply ignore this; it is just a standard way in JavaScript of

indicating uninitialized variables, non-existent object properties, or other similar scenarios.

Node 2

Similar to Node 1, open the `geth` console for Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

As we did for Node 1, in the `geth` console, enter the following commands:

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData","type":"uint256"}]
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303")
> simple.get()
42
```

Much like Node 1, the final command outputs `42`, which is our stored value.

Node 3

Again, similar to Node 1, open the `geth` console for Node 3 using the command shown here:

```
$ geth attach qdata/dd3/geth.ipc
```

In the `geth` console, enter the following

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData","type":"uint256"}]
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303")
> simple.get()
0
```

Note that the value returned is `0`, which indicates that Node 3 cannot access the value of `42`, which is expected as Node 3 is not privy to the transaction.

Node 4

Finally, we open the `geth` console for Node 4 using the command shown here:

```
$ geth attach qdata/dd4/geth.ipc
```

In the console, enter the following command as we did for nodes 1, 2, and 3:

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData","c
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303
> simple.get()
0
```

As expected, Nodes 3 and 4, which are not party to the transaction, will see `0` as the transaction value. On the other hand, Nodes 1 and 2 are able to see the transaction value, which is `42`.

In the next section, we'll see how we can use Cakeshop to view the transactions and other relevant blockchain data. Remember, as we discussed earlier in this chapter, we need to monitor and visualize enterprise blockchain networks. Cakeshop fulfills that need.

Viewing the transaction in Cakeshop

In Cakeshop, we can see this transaction and relevant attributes. We simply browse to chain explorer and enter the transaction hash

"`0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf`" from step 1 of this process. Cakeshop will display the status of the

transaction, block ID, block number, contract address, and several other attributes, which makes it easy to explore the transaction process in detail:

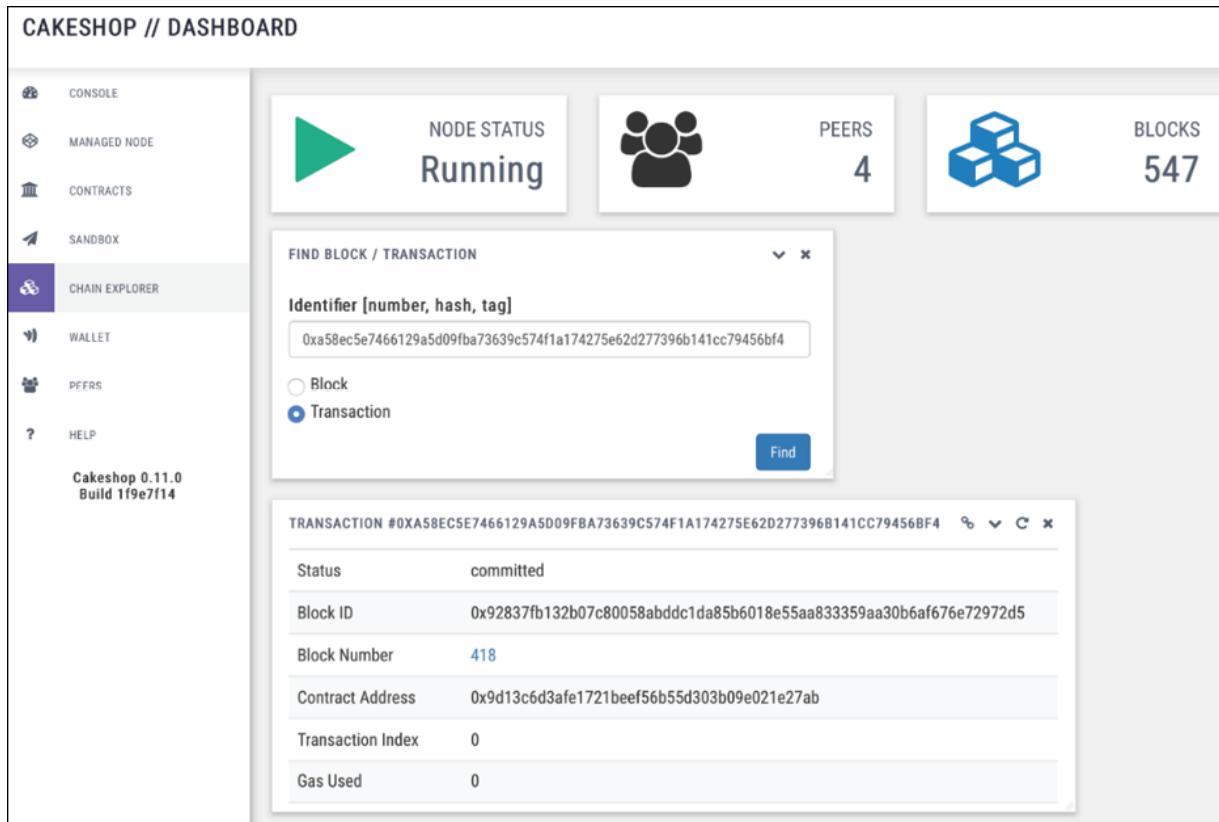


Figure 20.15: Cakeshop chain explorer

Further investigation

We can also see the transaction receipts and contract code using the `geth` console.

For this, we `attach` to `geth` using the following command:

```
$ geth attach
```

We will do this on each node, as follows.

Node 1

First, we open the `geth` console using the following command on Node 1:

```
$ geth attach qdata/dd1/geth.ipc
```

When the `geth` console opens, enter the following statement, which uses the transaction hash from our first step of contract deployment:

```
> eth.getTransaction("0xa58ec5e7466129a5d09fba73639c574f1a174275
```

The output shown here will be displayed:

```
{
  blockHash: "0x92837fb132b07c80058abddc1da85b6018e55aa833359aa3",
  blockNumber: 418,
  from: "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
  gas: 4700000,
  gasPrice: 0,
  hash: "0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b14",
  input: "0x6d99da7776195f534b48d53e8a2e94231299d38967a874115fe3",
  nonce: 2,
  r: "0x3ca297d21fed45e3701d2e3ed9d4e3a144c1350ea5b981005ff734ef",
  s: "0x6233d5d65445ae258e7c04efd085f1cdba42ca5f63c9982b4db779d4",
  to: null,
  transactionIndex: 0,
  v: "0x26",
  value: 0
}
```

In the preceding output, we can see that the `v` value is `0x26` in hexadecimal, which means `38` in decimal. This value of `38` indicates that the transaction is private.

Also, notice the `input` field, which is the encrypted payload of the transaction.

Similarly, we can get the transaction receipt of this transaction by issuing the call shown here:

We can see in the output that the status is `0x1`, indicating the success of the transaction. As we need to get the code of this transaction (smart contract), we need the contract address, which we can see in the `contractAddress` field. We copy that address into the RPC call shown here to fetch the code for this contract:

As we can see in the preceding output, the contract code is visible as bytecode in hex.

Node 2, which is privy to the transaction

Similarly to Node 1, open the `geth` console on Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

In the `geth` console, enter the following statement. This method takes the address of the smart contract and returns the compile smart contract code.

(bytecode):

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

The following output shows the bytecode of our contract at address "0x9d13c6d3afe1721beef56b55d303b09e021e27ab":

As expected, the code is also visible on Node 2, as Node 2 is party to the transaction.

Node 3, which is not privy to the transaction

Now, we open the `geth` console on Node 3:

```
$ geth attach qdata/dd3/geth.ipc
```

In the console, enter the following statement with the address of our smart contract:

```
> eth.getCode("0x9d13c6d3afe1721beeef56b55d303b09e021e27ab")
```

This will show the output shown here:

"0x"

This output indicates that the contract code is only available on the nodes that are party to the transaction. The contract code is not available on other nodes that are not a party to the transaction.

This experiment demonstrates that the contract is only available on the nodes that are party to the transactions. First, we deployed our private

contract from Node 2 with Node 1 as a participant and used the `geth` console to interact with the contract. We saw that the value `42` is only available on the nodes that are party to the transaction. Also, we further experimented and noticed that not only the value, but the smart contract code, is only available on the nodes that are party to our private contract (transaction).

This network is now available for further experiments. You can create your own private contracts with different nodes as parties. You can also explore how other methods such as `debug_traceTransaction` behave when interacting with private contracts. This network can also be used to build some PoCs for an enterprise **DApp**.

After this quick experiment of demonstrating how private transactions work in Quorum, let's explore other Quorum projects. As mentioned earlier, Quorum is a feature-rich platform and is under continuous improvement and development. As a result, new features and projects are introduced regularly. Some of the other projects under Quorum are listed here, along with brief descriptions.

Other Quorum projects

Some of the projects and features are listed here.

Remix plugin

This is a plugin for the popular Remix IDE, which supports private smart contracts on the Quorum blockchain. More information on this is available here:

<https://docs.goquorum.consensys.net/en/latest/Reference/RemixPlugin/Overview/>.

Pluggable architecture

Quorum supports a pluggable architecture that allows us to add new features to the Quorum client as plugins. This approach allows us to keep the core Quorum services separate from the new features, which allows greater modularity and extensibility without modifying the core client.

More information on this feature is available here:

<https://docs.goquorum.consensys.net/en/latest/Concepts/Plugins/>.

Quorum is a large project with many excellent enterprise features. It is not possible to cover all of them in this chapter. However, the material provided should get you started with Quorum blockchain setup, plus an understanding of the concept of private transactions.



There is a wealth of information available in the official Quorum documentation at

<https://docs.goquorum.consensys.net/en/latest/>. You are encouraged to go through the documentation to get a deeper understanding of Quorum. Alternatively, the following article covers a wide range of projects available:

https://medium.com/@mateo_ventures/heres-who-is-building-on-quorum-see-the-list-b18d65aa0a2c.

Quorum has been used in many industries for different projects, including finance, supply chain, healthcare, media, and government sectors. A couple of example uses are:

- Tracking luxury goods: <https://www.coindesk.com/louis-vuitton-owner-lvmh-is-launching-a-blockchain-to-track-luxury-goods>
- Post trade processing platform:
<https://cointelegraph.com/news/oil-trading-blockchain-platform-vakt-launches-with-shell-bp-as-first-users>

Quorum is also available on different cloud platforms, including:

- Azure: <https://azuremarketplace.microsoft.com/fi/marketplace/apps/enterprise-ethereum-alliance.quorum-demo?tab=Overview>
- Kaleido: <https://www.kaleido.io>

Summary

In this chapter, we started with an introduction to enterprise solutions and blockchain. We saw some limiting factors in public chains that make them unsuitable for enterprise use cases. We also looked at some requirements that, when met, will make a blockchain suitable for enterprise use cases.

We also covered how to design enterprise blockchain solutions and made a case to see the enterprise blockchain solutions in the context of the enterprise architecture. Next, we explored cloud computing and the definition of **Blockchains as a Service**. We briefly looked at the efforts being made to standardize enterprise Ethereum specifications. The last sections of this chapter covered enterprise blockchain platforms, including Quorum and Corda, and finally we set up an IBFT network using Quorum to demonstrate how privacy is achieved using Quorum.

In the next chapter, we will introduce scalability, security, and other challenges that blockchains face, how many of these limitations are being addressed, and what the future holds for the blockchain technology.

Scalability and Other Challenges

This chapter aims to explore various challenges that need to be addressed before blockchain can become a mainstream technology. Despite the fact that multiple use cases and **Proof of Concept (PoC)** systems have been developed, and that the technology works well for many scenarios, there still is a need to address some fundamental limitations that are present in blockchains to make the technology more adoptable.

At the top of the list of these issues comes **scalability** and then **privacy**, both of which are significant limitations. The lack of scalability is a general concern, where blockchains do not meet the adequate performance levels expected by users when the chain is used on a large scale. Privacy is also of utmost importance, especially as blockchains are envisioned to be used in privacy-demanding industries. There are specific requirements around the confidentiality of transactions in the finance, law, and health industries, for example. These two issues are becoming inhibiting factors toward blockchain technology's broader acceptance. A review of currently proposed and ongoing research in these two specific areas will be presented throughout this chapter.

In addition to scalability and privacy, other challenges facing blockchain include regulation, integration, interoperability, adaptability, and **security** in general. Although in blockchains generally, security is quite good—for example, Bitcoin has stood the test of time and is still extremely secure—there still are some caveats that may allow security to be compromised. Moreover, there are some reasonable security concerns in other blockchains such as **Ethereum**, regarding **smart contracts**, **denial of service (DoS)**

attacks, and a large **attack surface**. All of these will be discussed in detail in the following sections, which are as follows:

- Scalability
- Privacy
- Security
- Other challenges

Now, we'll start with scalability, one of blockchain's biggest challenges.

Scalability

This problem has been a focus of intense debate, rigorous research, and media attention for the last few years.

This is the single most important problem in blockchain, which could mean the difference between the wider adaptability of blockchains or limited private use only by consortiums. As a result of substantial research in this area, many solutions have been proposed, which are discussed in the following section.

From a theoretical perspective, the general approach toward tackling the scalability issue generally revolves around protocol-level enhancements. For example, a commonly mentioned solution to Bitcoin scalability is to increase its block size. This would mean that a larger number of transactions can be batched in a block, resulting in increasing scalability.



We discussed block size increase and other relevant blockchain solutions in more detail in *Chapter 9, Alternative Coins*.

Other proposals include solutions that offload certain processing to off-chain networks; for example, off-chain state networks.

Based on the aforementioned solutions, generally, the proposals can be divided into two categories: **on-chain solutions**, which are based on the idea of changing fundamental protocols on which the blockchain operates, and **off-chain solutions**, which make use of network and processing resources off-chain in order to enhance the blockchain.

Now, we'll introduce an approach that has been proposed to view blockchains as being composed of different *planes*. We'll then address each plane separately to develop an overall scalability solution.

Blockchain planes

Another approach to addressing limitations in blockchains has been proposed by Andrew Miller et al. and others in their position paper *On Scaling Decentralized Blockchains* (available at https://doi.org/10.1007/978-3-662-53357-4_8) In this paper, it is shown that a blockchain can be divided into various abstract layers called *planes*. Each plane is responsible for performing specific functions. These include the *network plane*, *consensus plane*, *storage plane*, *view plane*, and *side plane*. This abstraction allows limitations at each plane to be addressed individually and in a structured manner, which then results in an overall scalability solution. A brief overview of each layer is given in the following subsections with some references to the Bitcoin system.

Network plane

First, we'll discuss the network plane. The paper *On Scaling Decentralized Blockchains* mentioned previously identifies that, in **Bitcoin**, the network plane underutilizes network bandwidth in regards to transaction propagation, which is one of its key functions. This limitation transpires due to the way the Bitcoin protocol effectively duplicates transaction broadcasts. First, local transaction validation occurs, where the protocol propagates all transactions. Then, after mining, block propagation occurs, which contains previously disseminated transactions again.



It should be noted that this issue was addressed by BIP 152 (*Compact Block Relay*, <https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki>).

Consensus plane

The second layer is called the consensus plane. This layer is responsible for mining and achieving consensus. Bottlenecks in this layer revolve around limitations in **Proof of Work (PoW)** algorithms, whereby increasing consensus speed and bandwidth results in compromising the security of the network due to an increase in the number of forks.

Storage plane

The storage plane is the third layer, which stores the ledger. Issues in this layer revolve around the need for each node to keep a copy of the entire ledger, which leads to certain inefficiencies, such as increased bandwidth and storage requirements. Bitcoin has a method available called **pruning**, which allows a node to operate without the need to keep the full blockchain in its storage. Pruning means that when a Bitcoin node has downloaded the blockchain and validated it, it deletes the old data that it has already validated. This saves storage space. This functionality has resulted in major improvements from a storage perspective.

View plane

Next on the list is the view plane, which proposes an optimization based on the proposal that Bitcoin miners do not need the full blockchain to operate, and that a view can be constructed out of the complete ledger as a representation of the entire state of the system, which is sufficient for miners to function. Implementation of views will eliminate the need for mining nodes to store the full blockchain.

Side plane

Finally, there's the side plane, which represents the idea of off-chain transactions. This is the process whereby the concept of payment or transaction channels is used to offload the processing of transactions between participants. However, it is still backed by the main Bitcoin blockchain.

Clearly, this plane-based model can be used to describe limitations and improvements in current blockchain designs in a targeted, structured manner.

Also, there are several other general strategies that have been proposed over the last few years that can address the limitations in current blockchain designs, such as Ethereum and Bitcoin. These approaches will be characterized and discussed individually in the following section.

Methods for improving scalability

As this is a very active area of research, over the years many techniques and proposals have been made to address the blockchain scalability problem. In this section, we'll introduce many of these techniques.

We can divide approaches to solving scalability issues into three main categories based on the *layer* in the blockchain stack they operate on.



Note that the model described previously, which looks at the blockchain stack as a combination of different planes, is a different concept to the layers model mentioned in this section, which views the blockchain stack slightly differently. There is, however, some overlap—for example, when we say *side plane*, we also mean *Layer 2*. Similarly, the consensus plane can be viewed as a *Layer 1* component.

We describe these categories here:

- *Layer 0* methods, or *network* layer methods, where the mechanisms to improve scalability operate at the network level of the blockchain architecture stack.

- *Layer 1* methods are also called on-chain methods, where the blockchain protocol itself is enhanced to improve scalability. This layer represents the blockchain itself.
- *Layer 2* methods, or off-chain methods, where mechanisms that are not part of the blockchain and exist outside of the main blockchain are used to improve the scalability of the blockchain.

Scalability has two facets. One is increasing the processing speed of transactions, and the other is the increase in the number of nodes on the network. Both are desirable in many situations; however, speed of transactions is more sought after on public networks.

Layer 0 – network solutions

We'll describe some of the Layer 0, or network layer, solutions in the following sections.

Kadcast

This is a new protocol that enables fast, efficient, and secure block propagation for the Bitcoin blockchain network.



More information is available in the paper: Rohrer, E. and Tschorsch, F., 2019, October. *Kadcast: A structured approach to broadcast in blockchain networks*. In Proceedings of the 1st ACM Conference on Advances in Financial Technologies (pp. 199-213).

<https://dl.acm.org/doi/pdf/10.1145/3318041.3355469>

bloXroute

Another network layer solution is bloXroute, which aims to address scalability problems by creating a trust-less blockchain distribution network.



More information about bloXroute is available at
<https://bloxroute.com> and in the following whitepaper:

<https://bloxroute.com/wp-content/uploads/2019/11/bloXrouteWhitepaper.pdf>

Next, let's move on to some core on-chain solutions, or *Layer 1* solutions.

Layer 1 – on-chain solutions

In this section, we'll describe Layer 1, or on-chain, solutions, which target core blockchain elements such as blocks, transactions, and other on-chain data structures to address the scalability problem.

Transaction parallelization

Usually, in blockchain designs, transactions are executed sequentially. For example, in Ethereum, all transaction execution is sequential, which allows it to be safe and consistent. This also raises a question that if, somehow, transaction executions can be done in parallel, without compromising the consistency and security of the blockchain, it would result in much better performance.

Some suggestions have already been made for Ethereum, for example:

- Easy parallelizability, which introduces a new type of transaction:
<https://github.com/ethereum/EIPs/issues/648>.
- Transaction parallelizability in Ethereum, which proposes a mechanism to enable parallel transaction execution in Ethereum. The paper is available here:
<https://arxiv.org/pdf/1901.09942.pdf>.

However, at the time of writing, none of these ideas are production-ready. Currently, Hyperledger Sawtooth is to be the only blockchain that supports parallel transactions and is ready to be used in production. We discussed Sawtooth in detail in *Chapter 17, Hyperledger*.

Increase in block size

This is the most debated proposal for increasing blockchain performance (transaction processing throughput). Currently, Bitcoin can process only about three to seven transactions per second, which is a major inhibiting factor in adapting the Bitcoin blockchain for processing microtransactions. Block size in Bitcoin is hardcoded to be 1 MB, but if the block size is increased, it can hold more transactions and can result in faster confirmation time. There are several **Bitcoin Improvement Proposals (BIPs)** made in favor of block size increase. These include [BIP 100](#), [BIP 101](#), [BIP 102](#), [BIP 103](#), and [BIP 109](#).



An excellent account of historic references and discussion is available at https://en.bitcoin.it/wiki/Block_size_limit_controversy.

In Ethereum, the block size is not limited by hardcoding; instead, it is controlled by a gas limit. In theory, there is no limit on the size of a block in Ethereum because it's dependent on the amount of gas, which can increase over time. This is possible because miners are allowed to increase the gas limit for subsequent blocks if the limit has been reached in the previous block. Bitcoin **Segregated Witness (SegWit)** has addressed this issue by separating witness data from transaction data, which resulted in more space for transactions. Other proposals for Bitcoin include Bitcoin Unlimited, Bitcoin XT, and Bitcoin Cash. You can refer to *Chapter 6, Introducing Bitcoin*, for more details.



For more information on Bitcoin proposals, refer to the following addresses:

<https://www.bitcoinunlimited.info>

<https://www.bitcoincash.org>

Block interval reduction

This is another proposal about reducing the time between each block generation. The time between blocks can be decreased to achieve faster finalization of blocks, but it may result in less security due to the increased number of forks. Ethereum has achieved a block time of approximately 14 seconds.

This is a significant improvement from the Bitcoin blockchain, which takes 10 minutes to generate a new block. In Ethereum, the issue of high orphaned blocks resulting from smaller times between blocks is mitigated by using the **Greedy Heaviest Observed Subtree (GHOST)** protocol, whereby orphaned or stale blocks (also called uncles in the Ethereum chain) are also included in determining the valid chain. Once Ethereum moves to **Proof of Stake (PoS)**, this will become irrelevant as no mining will be required and almost immediate finality of transactions can be achieved.

Invertible Bloom Lookup Tables

This is another approach that has been proposed to reduce the amount of data required to be transferred between the Bitcoin nodes. **Invertible Bloom Lookup Tables (IBLTs)** were originally proposed by Gavin Andresen, and the key attraction in this approach is that it does not result in a hard fork of Bitcoin if implemented. The key idea is based on the fact that there is no need to transfer all transactions between nodes; instead, only those that are not already available in the transaction pool of the syncing node are transferred. This allows quicker transaction pool synchronization between nodes, thus increasing the overall scalability and speed of the Bitcoin network.

Sharding

Sharding is not a new technique and has been used in distributed databases for scalability, such as MongoDB and MySQL. The key idea behind sharding is to split up the tasks into multiple chunks that are then processed by multiple nodes. This results in improved throughput and reduced storage requirements. In blockchains, a similar scheme is employed, whereby the state of the network is partitioned into multiple shards. The state usually includes balances, code, nonce, and storage. Shards are loosely coupled

partitions of a blockchain that run on the same network. There are a few challenges related to inter-shard communication and consensus on the history of each shard. This is an open area of research and has been extensively studied in the context of Ethereum 2.0.

Private blockchains

Most private blockchains are inherently faster because no real decentralization is required and participants on the network do not need to mine using PoW; instead, they can only validate transactions. This can be considered as a workaround to the scalability issue in public blockchains; however, this is not the solution to the scalability problem. Also, it should be noted that private blockchains are only suitable in specific areas and setups such as enterprise environments, where all participants are known.

Proof of Stake

PoS algorithm-based blockchains are fundamentally faster because PoS algorithms do not require the completion of time- and energy-consuming PoW. PoS was explained in detail in *Chapter 5, Consensus Algorithms*.

Block propagation

In addition to the preceding proposal, pipelining of block propagation has also been suggested, which is based on the idea of anticipating the availability of a block. In this scheme, the availability of a block is already announced without waiting for actual block availability, thus reducing the round-trip time between nodes. Finally, the problem of long distances between the transaction originator and nodes also contributes toward the slowdown of block propagation. It has been shown in the research conducted by Christian Decker et al. that connectivity increase can reduce propagation delay of blocks and transactions. This is possible because, if at any one time the Bitcoin node is connected to many other nodes, it can speed up the information propagation on the network.

An elegant solution to scalability issues will most likely be a combination of some or all of the aforementioned general approaches. A number of

initiatives taken in order to address scalability and security issues in blockchains are now almost ready for implementation or already implemented. For example, Bitcoin SegWit is a proposal that can help massively with scalability and only needs a soft fork in order for it to be implemented. The key idea behind so-called SegWit is to separate signature data from the transactions, which resolves the transaction malleability issue and allows block size increase, thus resulting in increased throughput.

Bitcoin-NG

Another proposal, Bitcoin-NG, which is based on the idea of microblocks and leader election, has gained some attention recently. The core idea is to split blocks into two types, namely leader blocks (also called key blocks) and microblocks:

- **Leader blocks:** These are responsible for PoW, whereas microblocks contain actual transactions.
- **Micro blocks:** These do not require any PoW and are generated by the elected leader every block-generation cycle. This block-generation cycle is initiated by a leader block. The only requirement is to sign the microblocks with the elected leader's private key. The microblocks can be generated at a very high speed by the elected leader (miner), thus resulting in increased performance and transaction speed.

On the other hand, the *Ethereum 2.0 Mauve Paper*, written by Vitalik Buterin and presented at Ethereum Devcon2 in Shanghai, describes a different vision of a scalable blockchain.



The mauve paper is available at
https://docs.google.com/document/d/1maFT3cpHvwn29gLvtY4WcQiI6kRbN_nbCf3JlgR3m_8/edit#.

This proposal is based on a combination of sharding and an implementation of the PoS consensus algorithm. The paper defined certain goals such as gaining efficiency via PoS, minimizing block time, and ensuring economic finality, scalability, cross-shard communication, and censorship resistance.



The mauve paper is the vision for Ethereum 2.0, which is now becoming a reality. We discussed Ethereum 2.0, which will be introduced with the Serenity update, in *Chapter 16, Serenity*.

DAGs

DAG stands for **Directed Acyclic Graph**. It is seen as an alternative to blockchain technology. Blockchain is fundamentally a linked list, whereas DAG is an acyclic graph where links between nodes have only one direction. The DAG structure eliminates the need for mining and blocks, which allows parallel creation and confirmation of transactions, thus achieving high transaction throughput. One example of DAG is IOTA cryptocurrency. More information about this can be found at <https://www.iota.org>.

Consensus mechanisms

Traditionally, blockchains are based on PoW or PoS algorithms, which are inherently slow in terms of performance. Especially PoW is able to process only a few transactions per second. If a different consensus mechanism such as Raft or PBFT is used, then the processing speed can increase significantly. We examined these concepts in *Chapter 5, Consensus Algorithms* in greater detail.

So far, we have discussed Layer 0 (network-based solutions) and Layer 1 (on-chain solutions) to the scalability issue. In the next section, we'll introduce another approach that aims to solve the scalability problem by using off-chain or Layer 2 components.

Layer 2 – off-chain and multichain solutions

Layer 2 solutions are based on the idea that instead of modifying the main chain to achieve scalability, the objective should be to offload some of the processing to faster mechanisms outside of the main underlying chain, do processing there, and then write the result back on the main chain as an integrity guarantee. Several such techniques are described here.

State channels

This is another approach proposed for speeding up the transaction on a blockchain network. The basic idea is to use side channels for state updating and processing transactions off the main chain; once the state is finalized, it is written back to the main chain, thus offloading the time-consuming operations from the main blockchain.

State channels work by performing the following three steps:

1. First, a part of the blockchain state is locked under a smart contract, ensuring the agreement and business logic between participants.
2. Now, off-chain transaction processing and interaction is started between the participants that update the state (only between themselves for now). In this step, almost any number of transactions can be performed without requiring the blockchain. This is what makes the process so fast and arguably the best candidate for solving blockchain scalability issues. It could be argued that this is not a real on-blockchain solution such as, for example, sharding, but the end result is a faster, lighter, and more robust network that can prove very useful in micropayment networks, IoT networks, and many other applications.
3. Once the final state is achieved, the state channel is closed, and the final state is written back to the main blockchain. At this stage, the locked part of the blockchain is also unlocked.

This process is shown in the following diagram:

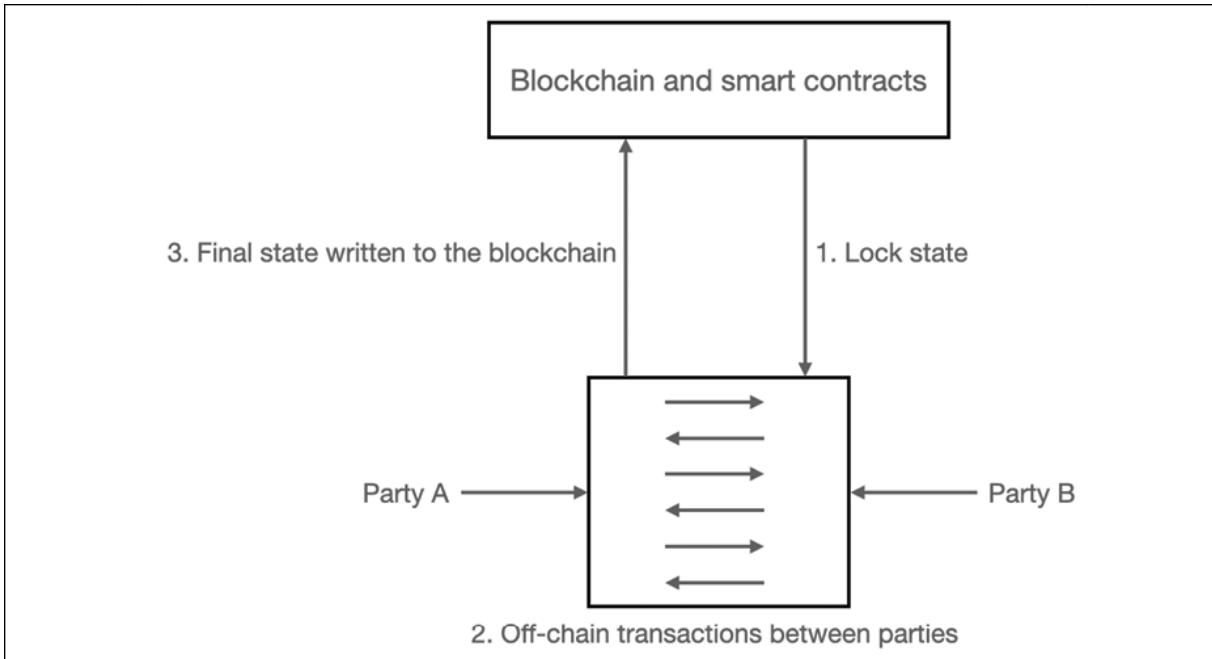


Figure 21.1: State channels

This technique has been used in the **Bitcoin Lightning** network and Ethereum's **Raiden**. The key difference between Lightning and Raiden is that Lightning only works for Bitcoin transactions, whereas Raiden supports all ERC20-compliant tokens, making the Raiden network a more flexible option.

Sidechains

Sidechains can improve scalability indirectly by allowing many sidechains to run along with the main blockchain, while allowing usage of perhaps comparatively less secure and faster sidechains to perform transactions but that are still pegged with the main blockchain. The core idea of sidechains is called a **two-way peg**, which allows the transferal of coins from a parent chain to a sidechain and vice versa.

Sub-chains

This is a relatively new technique recently proposed by Peter R. Rizun and is based on the idea of weak blocks, which are created in layers until a strong block is found.



Rizun's sub-chains research paper is available at:
<https://doi.org/10.5195/ledger.2016.40>. Rizun, P. R.
(2016). *Subchains: A Technique to Scale Bitcoin and Improve the User Experience*. Ledger, 1, 38-52.

Weak blocks can be defined as those blocks that have not been able to be mined by meeting the standard network difficulty criteria, but have done enough work to meet another weaker difficulty target. Miners can build **sub-chains** by layering weak blocks on top of each other, unless a block is found that meets the standard difficulty target.

At this point, the sub-chain is closed and becomes the strong block. Advantages of this approach include reduced waiting time for the first verification of a transaction. This technique also results in a reduced chance of orphaning blocks and a speeding up of transaction processing. This is also an indirect way of addressing the scalability issue. Sub-chains do not require any soft fork or hard fork to implement but need acceptance by the community.

Tree chains

There are also other proposals to increase Bitcoin scalability, such as **tree chains**, which change the blockchain layout from a linearly sequential model to a tree. This tree is basically a binary tree that descends from the main Bitcoin chain. This approach is similar to sidechain implementation, eliminating the need for major protocol change or block size increase. It allows improved transaction throughput. In this scheme, the blockchains themselves are fragmented and distributed across the network in order to achieve scalability.

Moreover, mining is not required to validate the blocks on the tree chains; instead, users can independently verify the block header. However, this idea is not ready for production yet and further research is required in order to make it practical.



The original idea was proposed in the following research paper:
<https://eprint.iacr.org/2016/545.pdf>.

In addition to the aforementioned general techniques, some Bitcoin-specific improvements have also been proposed by Christian Decker (much of Decker's work can be found here:

<https://scholar.google.ch/citations?user=ZaeGlZIAAAAJ&hl=en>

) in his book *On the scalability and security of Bitcoin*. This proposal is based on the idea of speeding up propagation time as the current information propagation mechanism results in blockchain forks. These techniques include minimization of verification, pipelining of block propagation, and connectivity increase. These changes do not require fundamental protocol-level changes; instead, these changes can be implemented independently in the Bitcoin node software.

With regards to verification minimization, it has been noted that the block verification process is contributing toward propagation delay. The reason behind this is that a node takes a long time to verify the uniqueness of the block and transactions within the block. It has been suggested that a node can send the inventory message as soon as the initial PoW and block validation checks are completed. This way, propagation can be improved by just performing the first *difficulty check* and not waiting for transaction validation to finish.

Plasma

Another recent scalability proposal is **Plasma**, which has been proposed by Joseph Poon and Vitalik Buterin. This proposal describes the idea of running smart contracts on the root blockchain (Ethereum mainnet), and having child blockchains that perform high numbers of transactions, to feedback small amounts of commitments to the parent chain. In this scheme, blockchains are arranged in a tree hierarchy with mining performed only on the root (main) blockchain, which feeds the proofs of security down to child chains. This is also a Layer 2 system since, like state channels, Plasma also operates off-chain.





The research paper on Plasma contracts is available at
<http://plasma.io>.

Commit chains

Commit chains are a more generic term for Vitalik Buterin's Plasma proposal. They are also referred to as non-custodial side chains but without a new consensus mechanism, as is the case with sidechains. They rely on main chain consensus mechanism, so they can be considered as safe as the main chain. The operator of the commit chain is responsible for facilitating communication between transacting participants and sending regular updates to the main chain.



More information on commit chains is available here: Khalil, R., Zamyatin, A., Felley, G., Moreno-Sanchez, P. and Gervais, A., 2018. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Report 2018/642, 2018. <https://eprint.iacr.org/2018/642.pdf>.

Zk-Rollups

Zk-Rollups is a Layer 2 solution that allows multiple transactions to be bundled into a single transaction. This approach enables mass transfer processing using a single transaction, which improves scalability.

Trusted hardware-assisted scalability

This technique is based on the idea that complex and heavy computing can be offloaded to off-chain resources in a verifiably secure manner. Once the computations are complete, the verified results are sent back to the blockchain.



A scalable verification solution for blockchains is described in this paper:
<https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf>.

Further information regarding a scalability solution based on trusted hardware can be found here: <https://truebit.io/>.

Other scalability techniques include multichain approaches. One of the prime examples of such techniques is **Polkadot**, which we'll describe briefly later in this chapter in the context of interoperability. However, this mechanism also provides scalability.

With this, we have completed our discussion of some of the many potential scalability improvements of blockchain. In the next section, we'll discuss privacy and some relevant techniques to achieve privacy in blockchain.

Privacy

Privacy in blockchain can be divided into two main categories based on the type of service required. These categories are the anonymity of the users and the confidentiality of the transactions. Anonymity is concerned with hiding the sender's or receiver's identity, whereas confidentiality addresses the requirements of hiding transaction values.

Anonymity

Anonymity is desirable in situations where the identity of users is required to be hidden from other participants on the network. This can be due to regulatory requirements, enterprise requirements, or just due to the sensitive nature of the transactions. For example, in a business network, it might be desirable to hide the dealings between different participants on the network from other competitors to avoid friction or unjustified business competitive advantage. The properties of unlinkability and untraceability are used to achieve anonymity.

The property of untraceability allows for the hiding of the trace of a transaction from one party to another. On the other hand, unlinkability means that an observer is unable to deduce the link between transactions and its participants, or the relationships between transaction participants

(senders and receivers). We also discussed anonymity in *Chapter 9, Alternative Coins*, in greater detail.

Confidentiality

Confidentiality is an absolute requirement in many industries such as finance, law, and health. Similarly, privacy of transactions is a much-desired property of blockchains. However, due to its very nature, especially in public blockchains, everything is transparent, thus inhibiting its usage in various industries where privacy is of paramount importance, such as finance, health, and many others.

There are different proposals that have been made to address the privacy issue and some progress has already been made. Several techniques, such as **indistinguishability obfuscation**, **homomorphic encryption**, **zero-knowledge proofs (ZKPs)**, and **ring signatures**, have all been researched and implemented. All of these techniques have their merits and demerits and will be discussed in the following sections.

Techniques to achieve privacy

In this section, we'll describe different techniques to achieve privacy, including anonymity and confidentiality.

Similar to scalability solutions, we can also divide privacy solutions into three categories, based on the layer within the blockchain architecture stack that they operate:

- Layer 0 methods, or network layer methods, where the mechanism to achieve privacy operates at a network level.
- Layer 1 methods, also called on-chain methods, where the blockchain protocol itself is enhanced to achieve privacy.
- Layer 2 methods, or off-chain methods, where mechanisms that exist outside of the main blockchain are used to achieve privacy on the blockchain.

As many of the techniques described in the following sections can be used at Layers 1 and 2—for example, zero-knowledge proofs—we are not distinguishing between these layers. Instead, we'll only list these solutions and discuss what they can be used for. Furthermore, solutions for anonymity and confidentiality also overlap and techniques used to achieve confidentiality are also more or less applicable in achieving anonymity. We do, however, segregate Layer 0, the *network* layer, and describe some of the network approaches to achieve anonymity, before moving on to introduce a range of privacy mechanisms that are applicable at Layers 1 and 2.

Layer 0

These solutions are network layer-level methods that provide anonymity by using TOR and I2P. These methods allow for hiding the identities of the parties involved.

Tor

Tor, the Onion Router, is a software that enables anonymous communications. More information on Tor is available here: <https://www.torproject.org>. We can use Tor to enable anonymous communication in cryptocurrency blockchain networks. Tor is a common choice to enable anonymous communication and has been used in Monero, Verge and in **Bitcoin** to provide anonymity.

For Bitcoin, see <https://en.bitcoin.it/wiki/Tor>. Monero can also be run with Tor (<https://web.getmonero.org/>). Verge is another example of cryptocurrency making use of Tor for IP obfuscation.

I2P

I2P, Invisible Internet Project, is an anonymous network built on the internet. It enables censorship-resistant peer-to-peer communication. It is used in Monero to provide anonymity services. More information on I2P is available here: <https://geti2p.net/en/>.



Monero and Zcash, which can be considered a lightweight fork of Monero, are examples of cryptocurrency blockchains that support anonymous transactions. Monero makes use of ring signatures, stealth addresses, and confidential transactions, whereas Zcash uses zk-SNARKs.

Layers 1 and 2

Other layers include Layer 1 and Layer 2 solutions, which we'll introduce next.

Indistinguishability obfuscation

This cryptographic technique may serve as a silver bullet to all privacy and confidentiality issues in blockchains, but the technology is not yet ready for production deployments. **Indistinguishable obfuscation (IO)** allows for code obfuscation, which is a very ripe research topic in cryptography. If applied to blockchains, IO can serve as an unbreakable obfuscation mechanism that will turn smart contracts into a black box where the behavior of the obfuscated code is indistinguishable. In simpler words, the inner functionality of the smart contract is totally hidden.

The key idea behind IO is what's called by researchers a *multilinear jigsaw puzzle*, which basically obfuscates program code by mixing it with random elements, and if the program is run as intended, it will produce the expected output. However, any other way of executing it would make the program look like random garbage data. This idea was first proposed by Sanjam Garg et al. in their research paper *Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits*.



This research paper is available at
<https://eprint.iacr.org/2013/451.pdf>.

Homomorphic encryption

This type of encryption allows operations to be performed on encrypted data. Imagine a scenario where the data is sent to a cloud server for

processing. The server processes it and returns the output without knowing anything about the data that it has processed. This is also an area ripe for research and fully homomorphic encryption, which allows all operations on encrypted data, is still not fully deployable in production; however, major progress in this field has already been made. Once implemented on blockchains, it can allow processing on ciphertext, which will allow the privacy and confidentiality of transactions inherently.

For example, the data stored on the blockchain can be encrypted using homomorphic encryption, and computations can be performed on that data without the need for decryption, thus providing privacy service on blockchains. This concept has also been implemented in a project named *Enigma*, which is available online at

<https://www.media.mit.edu/projects/enigma/overview> by MIT's Media Lab. Enigma is a peer-to-peer network that allows multiple parties to perform computations on encrypted data without revealing anything about the data.



The original research is available at
<https://crypto.stanford.edu/craig/>.

Zero-knowledge proofs

ZKPs have recently been implemented in Zcash successfully, as seen in *Chapter 9, Alternative Coins*. More specifically, **Succinct Non-Interactive Argument of Knowledge (SNARKs)** have been implemented in order to ensure privacy on the blockchain.

The same idea can be implemented in **Ethereum** and other blockchains also. Integrating Zcash on Ethereum is already a very active research project being run by the Ethereum Research & Development team and the Zcash company.



The original research paper is available at
<https://eprint.iacr.org/2013/879.pdf>. Another excellent paper is available here:

 <http://chriseth.github.io/notes/articles/zksnarks/zksnarks.pdf>.

There is a recent addition to the family of ZKPs called **zero-knowledge Succinct Transparent Argument of Knowledge (zk-STARK)**, which is an improvement on zk-SNARKs in the sense that zk-STARKs consume a lot less bandwidth and storage. Also, they do not require the initial, somewhat controversial, trusted setup that is required for zk-SNARKs. Moreover, zk-STARKs are much quicker as they do not make use of elliptic curves or rely on hashes.



The original research paper for zk-STARKs is available here:
<https://eprint.iacr.org/2018/046.pdf>.

A privacy preserving mechanism based on zk-SNARKs is called Nightfall. More information on Nightfall is available at
<https://github.com/EYBlockchain/nightfall>.

Bulletproofs are non-interactive zero-knowledge short proofs. They require no trusted setup. This scheme allows a prover to prove that an encrypted number is within a range of numbers without revealing any other information about the number.



More information on Bulletproofs is available here:

Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P. and Maxwell, G., 2018, May. *Bulletproofs: Short proofs for confidential transactions and more*. In 2018 IEEE Symposium on Security and Privacy (SP) (pp. 315-334). IEEE.

<https://electroneopulse.org/public/doc/Bulletproof%20RingCT.pdf>.

State channels

Privacy using state channels is also possible, simply due to the fact that all transactions are run off-chain, and the main blockchain does not see the

transaction at all except for the final state output, which ensures privacy and confidentiality.

Secure multiparty computation

The concept of secure multiparty computation is not new and is based on the notion that data is split into multiple partitions between participating parties, under a secret sharing mechanism, which then does the actual processing on the data without the need to reconstruct data on a single machine.

The output produced after processing is also shared between the parties. Multiple parties carry out the computation mutually without revealing their inputs. Only the output of the computation is revealed.



A secure multiparty computation platform called Enigma was proposed in 2015. The paper is available here:
https://web.media.mit.edu/~guyzys/data/enigma_full.pdf.

Trusted hardware-assisted confidentiality

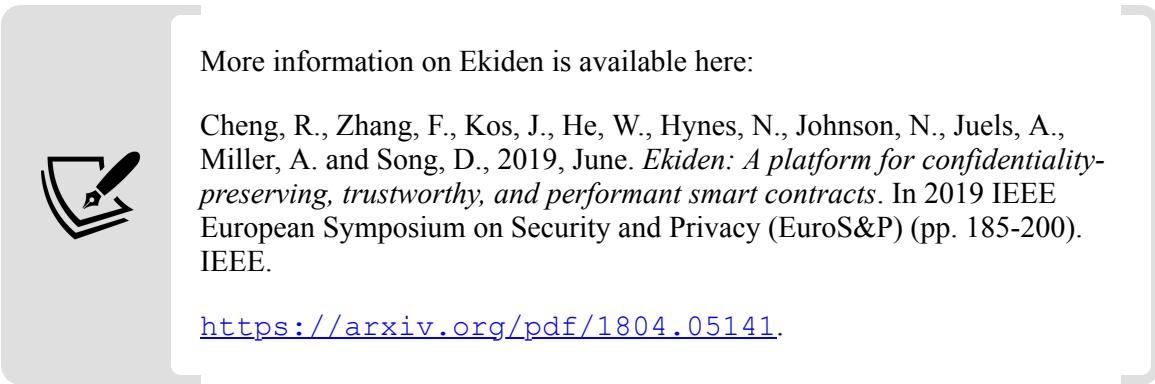
Trusted computing platforms can be used to provide a mechanism by which confidentiality of a transaction can be achieved on a blockchain, for example, by using Intel **Software Guard Extension (SGX)**, which allows code to be run in a hardware-protected environment called an **enclave**. Once the code runs successfully in the isolated enclave, it can produce a proof, called a **quote**, which is attestable by Intel's cloud servers.

It is a concern that trusting Intel will result in some level of centralization and is not in line with the true spirit of blockchain technology. Nevertheless, this solution has its merits and, in reality, many platforms already use Intel chips anyway, so trusting Intel may be acceptable by some in some cases.

If this technology is applied to smart contracts, then once a node has executed the smart contract, it can produce a proof of correctness, thus

proving successful execution, and other nodes will only have to verify it. This idea can be further extended by using any **Trusted Execution Environment (TEE)**, which can provide the same functionality as an enclave and is available even on mobile devices with **Near Field Communication (NFC)** and a secure element.

For example, Ekiden is a platform that makes use of Intel SGX's TEE to run smart contracts while preserving confidentiality.



CoinJoin

CoinJoin is a technique that is used to anonymize **Bitcoin** transactions by mixing them interactively. The idea is based on forming a single transaction from multiple entities without causing any change in inputs and outputs. It removes the direct link between senders and receivers, which means that a single address can no longer be associated with transactions, which could lead to the identification of users.

CoinJoin needs to cooperate between multiple parties that are willing to create a single transaction by mixing payments. Therefore, it should be noted that, if any single participant in the CoinJoin scheme does not keep up with the commitment made to cooperate for creating a single transaction, by not signing the transactions as required, then it can result in a **DoS** attack.

In this protocol, there is no need for a single trusted third party. This concept is different from mixing a service, which acts as a trusted third party or intermediary between the Bitcoin users and allows shuffling of

transactions. This shuffling of transactions results in the prevention of tracing and linking payments to a particular user.

Confidential transactions

Confidential transactions make use of **Pedersen commitments** in order to provide confidentiality. Commitment schemes allow a user to commit to some value while keeping it secret with the capability of revealing it later. Commitment schemes can be constructed simply by using cryptographic hash functions. Two properties that need to be satisfied in order to design a commitment scheme are binding and hiding.

Binding makes sure that the committer is unable to change the chosen value once committed, whereas the **hiding** property ensures that any adversary is unable to find the original value to which the committer made a commitment.

A Pedersen commitment is a type of information hiding, computationally-binding commitment scheme, under a discrete logarithm assumption. Pedersen commitments allow for additional operations and preserve commutative property on the commitments, which make them particularly useful for providing confidentiality in **Bitcoin** transactions. In other words, they support partial homomorphic encryption of values, meaning that using commitment schemes allows the hiding of payment values in a Bitcoin transaction.



This concept is already implemented in the *Elements Project*:
<https://elementsproject.org/features/confidential-transactions>.

MimbleWimble

The **MimbleWimble** scheme was proposed somewhat mysteriously on the Bitcoin IRC channel and since then has gained a lot of popularity.

MimbleWimble extends the idea of confidential transactions and CoinJoin, which allow the aggregation of transactions without requiring any

interactivity. However, it does not support the use of the Bitcoin scripting language, along with various other features of the standard Bitcoin protocol. This makes it incompatible with the existing Bitcoin protocol. Therefore, it can either be implemented as a sidechain to Bitcoin or on its own as an alternative cryptocurrency.

This scheme can address both privacy and scalability issues at once. The blocks created using the MimbleWimble technique do not contain transactions as in traditional **Bitcoin** blockchains; instead, these blocks are composed of three lists: an input list, an output list, and something called *excesses*, which are lists of signatures and differences between outputs and inputs. The input list basically references the old outputs, and the output list contains confidential transactions outputs.

The blocks created using the MimbleWimble scheme are verifiable by nodes by using signatures, inputs, and outputs to ensure the legitimacy of the block. In contrast to Bitcoin, MimbleWimble transaction outputs only contain `pubkeys`, and the difference between old and new outputs is signed by all participants involved in the transactions.

Mixing protocols

A **mixing protocol**, or mixer, is a service that allows users to preserve their privacy by mixing their coins with other users. We discussed mixers in detail in *Chapter 9, Alternative Coins*.

Attribute-based encryption

The **attribute-based encryption (ABE)** is a type of public key cryptography that provides confidentiality and access control simultaneously. The key idea behind this scheme is that the private key of a user and its ciphertext are dependent on the user's attributes such as their location, time zone, or their role in the organization. This means that the decryption is only possible when not only the private key is available but also the matching attributes are present.

Anonymous signatures

Anonymous signatures are types of digital signatures where the signatures do not reveal the identity of the signer. There are primarily two schemes available for anonymous signatures: **group signatures** and **ring signatures**.

Group signatures allow a set of signers to form a group managed by a group manager. Each member of the group is issued with a group signing key by the group manager. This group signing key allows each member of the group to anonymously sign messages on behalf of the group. The group manager is able to figure out who is the signer of the message, whereas external entities cannot. While group signatures can work well, one of the limitations of this scheme is that the group manager is able to identify the users. This issue was addressed with ring signatures.

Ring signatures allows a set of signers to form a group (a ring) of members. Each member in this group is able to sign messages on behalf of the ring. Unlike group signatures, there is no group manager in ring signatures, so no one is able to identify the signers.

While all these techniques are useful and provide many desirable properties of anonymity, there is also a problem of using this technology maliciously. Imagine if anonymity is misused by criminals involved in money laundering or selling illegal drugs by using anonymous cryptocurrency. It would be almost impossible to trace illegal activities back to the originator if absolute anonymity is achieved. Imagine a **Silk Road marketplace** ([https://en.wikipedia.org/wiki/Silk_Road_\(marketplace\)](https://en.wikipedia.org/wiki/Silk_Road_(marketplace))) variant using anonymous cryptocurrency. How could that be traced and stopped?

Privacy managers

Privacy managers provide a mechanism that provides confidentiality of transactions. These are off-chain components that substitute the transaction payload with a hash index, in such a way that only those participants who are party to the transaction are able to find the corresponding encrypted payload represented by the hash index. Other parties who are not privy to the transaction will simply ignore the hash. This concept was discussed in *Chapter 20, Enterprise Blockchains*.

We can visualize many of the blockchain privacy techniques discussed so far in this chapter using the following chart:

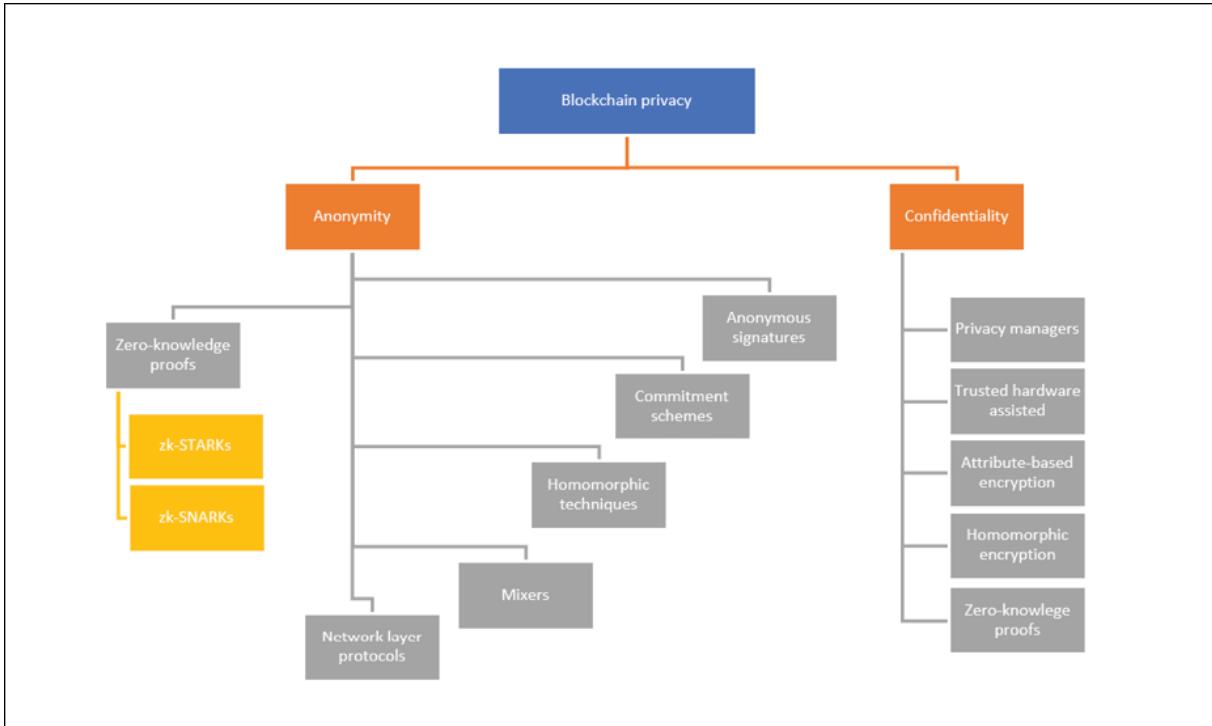


Figure 21.2: High-level taxonomy of blockchain privacy techniques

Other than scalability and privacy, there are several other general security aspects that need to be addressed in blockchain. We'll describe these general security topics next.

Security

Blockchains are generally secure and make use of asymmetric and symmetric cryptography, as required throughout the blockchain network, to ensure that the core layer of the blockchain is secure. However, there still are few caveats that can result in compromising the security of the blockchain.

There are a few examples of **transaction malleability**, **eclipse attacks**, and the possibility of **double-spending** in **Bitcoin** that, in certain scenarios, have been shown to work by various researchers. Transaction malleability opens up the possibility of double withdrawal or deposit by allowing a hacker to change a transaction's unique ID before the Bitcoin network can confirm it, resulting in a scenario where it would seem that transactions did not occur. [BIP 62](#) is one of the proposals, along with SegWit, that has suggested solutions to solve this issue. It should be noted that this is a problem only in the case of unconfirmed transactions; that is, scenarios where operational processes rely on unconfirmed transactions. In the case of normal applications that only rely on confirmed transactions, this is not an issue.

Information eclipse attacks in Bitcoin can result in double-spending. The idea behind eclipse attacks is that the Bitcoin node is tricked into connecting only with the attacker node IPs. This opens up the possibility of a 51% attack by the attacker. This has been addressed to some extent in Bitcoin client v0.10.1—more information on the attack and how it has been fixed is available here: <http://cs-people.bu.edu/heilman/eclipse/>.

Now, before discussing security further in blockchains, let's have a look at the interesting topic of formal verification. This is becoming a hot topic in blockchain space, due to its strong technical merits as a technique to address security issues. Note that this is not a new technique, but it has found very interesting applications in blockchain technology.

Formal verification

Before diving into different formal verification techniques that are available in blockchain space, first, let's develop some understanding of what **formal verification** is, what its types are, and why it is desirable.

Formal methods are the set of techniques used to model systems as mathematical objects. These methods include writing specifications in formal logic, model checking, verification, and formal proofs. Generally,

formal methods can be divided into two broad disciplines called **formal specifications** and **formal verification**. The first discipline is concerned with writing precise and concrete specifications, whereas the latter encompasses the development of proofs to prove the correctness of a specification.

In essence, formal verification is comprised of three steps:

1. First, creating a formal model of the system to be checked.
2. Secondly, write a formal specification of the properties that are expected to be satisfied by our model.
3. Finally, the model is checked to ensure that the model satisfies the specification.

For checking, there are two broad categories of techniques that can be used, namely *state exploration-based* approaches and *proof-based* approaches. There are pros and cons to both:

- State exploration-based approaches are automatic but are inefficient and difficult to scale. A usual problem is state explosion, where the number of states to check grow so exponentially big that the model does not fit in a computer's memory.
- On the other hand, proof-based approaches (theorem proving) are more precise but usually require human interaction and more in-depth knowledge of the proofs and relevant techniques.



Proof-based verification is performed using proof assistants such as Coq (<https://coq.inria.fr/>) and Isabelle (<https://isabelle.in.tum.de/>).

In the next section, we'll introduce a formal verification technique called model checking, which is used to check a system model for its correctness.

Model checking

Model checking can be defined as a technique used to verify finite state systems automatically. The underlying process is based on running an exhaustive search of the state space of the system. With this brute-force approach, it can be determined whether a specification is true or false. This option is becoming more popular because it does not require time-consuming proof-writing.

Instead, this paradigm allows a designer to write the specification formally, and its properties and the model checker will automatically explore the entire state space and evaluate the model against the specified properties. For example, if the specification says that a particular state should never be reached and during state exploration, the model checker finds that there is an execution (trace) that does enter that very state that is undesirable, then it will report that. The designer can then address the problem accordingly.

A visual representation of the model checking process is shown in the following diagram:

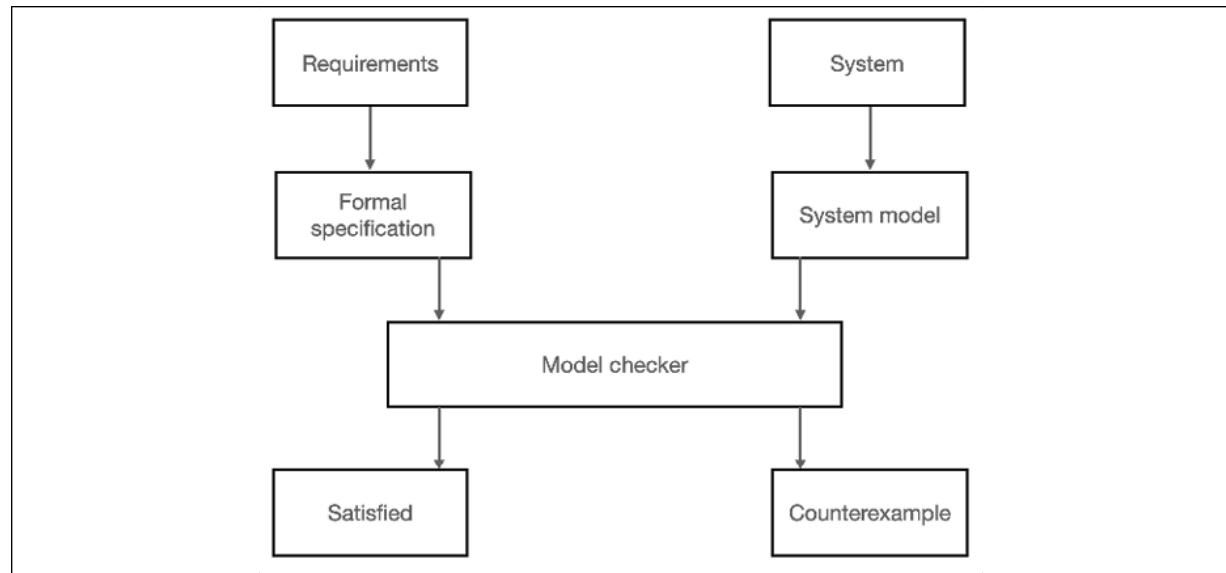


Figure 21.3: Model checking

The process of model checking starts with modeling, where a formal specification is created from an informal design of the model. The specification step comes next, where relevant properties of the design are specified using some logical formalism. Finally, verification of the model is

carried out, which checks all the properties defined in the specification and provides results. In the case of false or negative outcomes, error traces with counterexamples (roughly defined as exceptions) are provided to help the designer trace the error.

Mathematical assertion languages called **temporal logics** are used to describe states of a system over time. Temporal logic is used at the specification step of the model checking process, which represents the evolution of the behavior of a system over time, or in this case, the ordering of events of a system over time. In this paradigm, a specification of the behaviors of the system over time is built using various fundamental properties of individual states.

For example, in a state, it can be said that "something is on" or "something is active." There are also propositional connectors used in the specification's design, such as AND \wedge , OR \vee , NOT \neg , and IMPLY \Rightarrow . We will use these connectors in some basic examples shortly to demonstrate how they can be used in conjunction with formulas to describe system properties.

Finally, there are also **temporal connectives**, which issue statements like "the system can always only have one process active", or "the system can never be in a state where both read and write operations are performed simultaneously."

There are two types of temporal logics, namely **Linear Temporal logic (LTL)** and **Computation tree logic (CTL)**. LTL is concerned with the model properties on a single computation path or execution, whereas CTL is used to express properties over a tree of all possible paths.

There are a few fundamental temporal operators used in LTL to describe a system's behavior. These properties can be used to specify the safety and liveness properties of a system. The formulas are $\Box P$ (always), $\Diamond P$ (eventually), U (Until), and \bigcirc (Next). The always and eventually properties are most commonly used.

For example, $\mu ((\text{Message}) \neg \text{signed} \wedge \neg \text{sealed}) \Rightarrow \bigcirc \neg \text{valid}$ means if a message is not signed or sealed, then it is always the case that the message is not valid. Another example could be $\Box ((\text{broadcast}) \Rightarrow \Diamond$

`received)`, meaning that `broadcast` *always implies* that eventually the message will be `received`. Another general example could be that of a microwave oven ensuring that heating must not start until the door is closed, which can be represented in temporal logic as $\neg \text{Startheating} \sqcup \text{Door(closed)}$.

Formal methods are needed to ensure the safety of a system and have been in use in the avionics, electronics, hardware, and embedded systems industries for quite some time and, most lately, in the software industry. Due to renewed interest in distributed systems research with the advent of blockchain, and several high-profile incidents resulting in financial loss, such as the infamous **DAO hack**, the need to use formal methods in the blockchain space is becoming increasingly prominent. Some initiatives have already taken place; however, there is still a long way to go.

Blockchain needs to leverage on the research that has already been done in distributed systems and formal verification space, so that not only can existing blockchain systems be made safer, but new platforms can be developed based on formal specification. It should be noted that not all aspects of a blockchain need to, or should be, formally verified, due to the time-consuming and highly involved nature of the verification process. Therefore, at times, it is sufficient to model and test the most critical parts of a blockchain, such as consensus and security protocols.

There are many dimensions in a blockchain where model checking and specification using temporal logics can be quite useful. It can be used to describe an aspect of the blockchain platform formally, which can then be verified formally to ensure that all required properties are met. For example, in smart contract development, regardless of the language used, a model could be developed and checked before the actual coding of the smart contract. This approach will ensure that the program code, if programmed correctly according to the verified specification, will behave as it is intended. As smart contracts are used in all kinds of critical use cases, including financial transactions, it is advisable to apply model checking in this domain to ensure fairness and security of the system.

Now, we'll briefly look at how consensus mechanisms can be verified.

Verifying consensus mechanisms

From another angle, a **consensus mechanism**, such as **PBFT**, exists in a blockchain, which ensures that all the nodes in a blockchain network reach an agreement on the proposed values, even in the presence of faulty nodes.



Consensus mechanisms such as PBFT and Raft were described in *Chapter 5, Consensus Algorithms*.

This is an area of prime importance and due to its critical nature is regarded as the most vital core mechanism of a blockchain. Model checking can play a crucial role in the formal description and verification of consensus algorithms, to ensure that the protocols meet the required security properties.

A general approach to verifying properties of a consensus algorithm in blockchain starts from specifying the requirements, required properties, and specification in a formal language.



There are several options available for modeling and verifying programs, but tools such as TLA+ and TLC model checker are making the processes more accessible, and are becoming more prominent in this space. Another popular tool is known as SPIN, which uses PROMELA for writing specifications. However, note that there is no single right answer: the choice of formal verification tools mostly depends on the judgment and experience of the designer, the type and depth of verification required, and, to a certain degree, the usability of the verification tools.

A distributed consensus algorithm is evaluated against two categories of correctness properties, namely *safety* and *liveness*. Safety generally means that nothing bad will happen, whereas liveness implies that something good will eventually occur. Both of these properties, then, have some sub-properties, depending on the requirements. Usually, for consensus mechanisms, we have *agreement*, *integrity*, and *validity* attributes under safety properties and for liveness, often, *termination* is a desired property.

We can say that a program is correct if, in all possible executions, the program behaves correctly according to the specification. The specification is formally defined, which is then verified using a model checker or theorem provers.

Static analysis allows us to check the code against a set of coding rules to find code defects. The code does not execute; instead, it is statically checked. On the other hand, there is a dynamic analysis technique where the code is executed to find bugs and tested against test criteria. Dynamic analysis usually constitutes unit tests and is not considered a formal verification technique. Static analysis using formal techniques is used to formally verify the correctness of the program.

With this, we've completed our basic introduction to formal verification. Now, let's have a look at smart contract security and see what formal tools are available for smart contract security verification.

Smart contract security

Recently, a lot of work has been started in smart contract security and, especially, the formal verification of smart contracts is being discussed and researched. This was all triggered especially due to the infamous DAO hack.

Formal verification is a process of verifying a computer program to ensure that it satisfies certain formal statements. This is not a new concept and there are a number of tools available for other languages that achieve this; for example, Frama-C (<https://frama-c.com>) is available for analyzing C programs.

The key idea behind formal verification is to convert the source program into a set of statements that is understandable by the automated provers. For this purpose, **Why3**, which is a platform for program verification, is commonly used. We'll discuss Why3 at length in the *Why3* topic, later in this section.



Note that an experimental but operational Why3 verifier was available in Remix IDE initially but was later removed. However, now, in the Remix IDE, static analysis options are available. Details can be found here:

https://remix-ide.readthedocs.io/en/latest/static_analysis.html.

We will also present an example of static analysis in the Remix IDE in the next section.

Smart contract security is of paramount importance now, and many other initiatives have also been taken in order to devise methods that can analyze Solidity programs and find bugs. A recent and seminal example is **Oyente**, which is a tool built by researchers to analyze smart contracts.

Several security bugs in smart contracts have been discovered and analyzed in the Oyente white paper, *Making Smart Contracts Smarter*. These include transaction ordering dependence, timestamp dependence, mishandled exceptions such as call stack depth limit exploitation, and reentrance vulnerability. The transaction ordering dependency bug basically exploits the scenarios where the perceived state of a contract might not be what the state of the contract changes to after execution.

This weakness is a type of race condition. It is also called frontrunning and is possible due to the fact that the order of transactions within a block can be manipulated. As all transactions first appear in the memory pool, the transactions there can be monitored before they are included in the block. This allows a transaction to be submitted before another transaction, thus leading to controlling the behavior of a smart contract.

Timestamp dependency bugs are possible in scenarios where the timestamp of the block is used as a source of some decision-making within the contract, but timestamps can be manipulated by the miners. The *call stack depth limit* is another bug that can be exploited due to the fact that the maximum call stack depth of EVM is 1,024 frames. If the stack depth is reached while the contract is executing, then, in certain scenarios, the send or call instruction can fail, resulting in non-payment of funds.



The call stack depth bug was addressed in the EIP 50 hard fork at <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md>.

The **reentrancy bug** was exploited in the **DAO attack** to siphon out millions of dollars into a child DAO. The reentrancy bug essentially means that a function can be called repeatedly before the previous (first) invocation of the function has completed. This is particularly unsafe in ether withdrawal functions in Solidity smart contracts. This is because `block.timestamp`, the withdrawal function, can call back into it repeatedly, without waiting for the first call to complete. More on this shortly in the *Oyente* section.

In addition to the aforementioned bugs, there are several other problems that should be kept in mind while writing contracts, such as the fact that when sending funds to another contract, sending can fail, and even if `throw` is used as a *catch-all* mechanism, it will not work.

Timestamp dependence is another vulnerability that is quite common. Usually, the timestamp of a block is accessed via `now` or `block.timestamp`, but this timestamp can be manipulated by miners, leading to influencing the outcome of some function that relies on timestamps. Often, this is used as a source of randomness in lottery games to select the next winner. Thus, it might be possible for a miner to modify the timestamp in such a way that its chances of becoming the next winner increase.

Other usual software bugs, such as integer overflow and underflow, are also quite significant, and any use of integer variables should be carefully implemented in **Solidity**. For example, a simple program where `uint8` is used to parse through elements of an array with more than 255 elements can result in an endless loop. This occurs because `uint8` is limited to 256 numbers.

Smart contracts and, generally, all aspects of blockchain can be formally verified. In the following sections, examples of contract verification will be shown using Remix IDE, Why3, and Oyente.

Static analysis in Remix IDE

Static analysis of Solidity code is now available as a feature in the Solidity online Remix IDE. The code is analyzed for vulnerabilities and reported in the analysis tab of the Remix IDE:

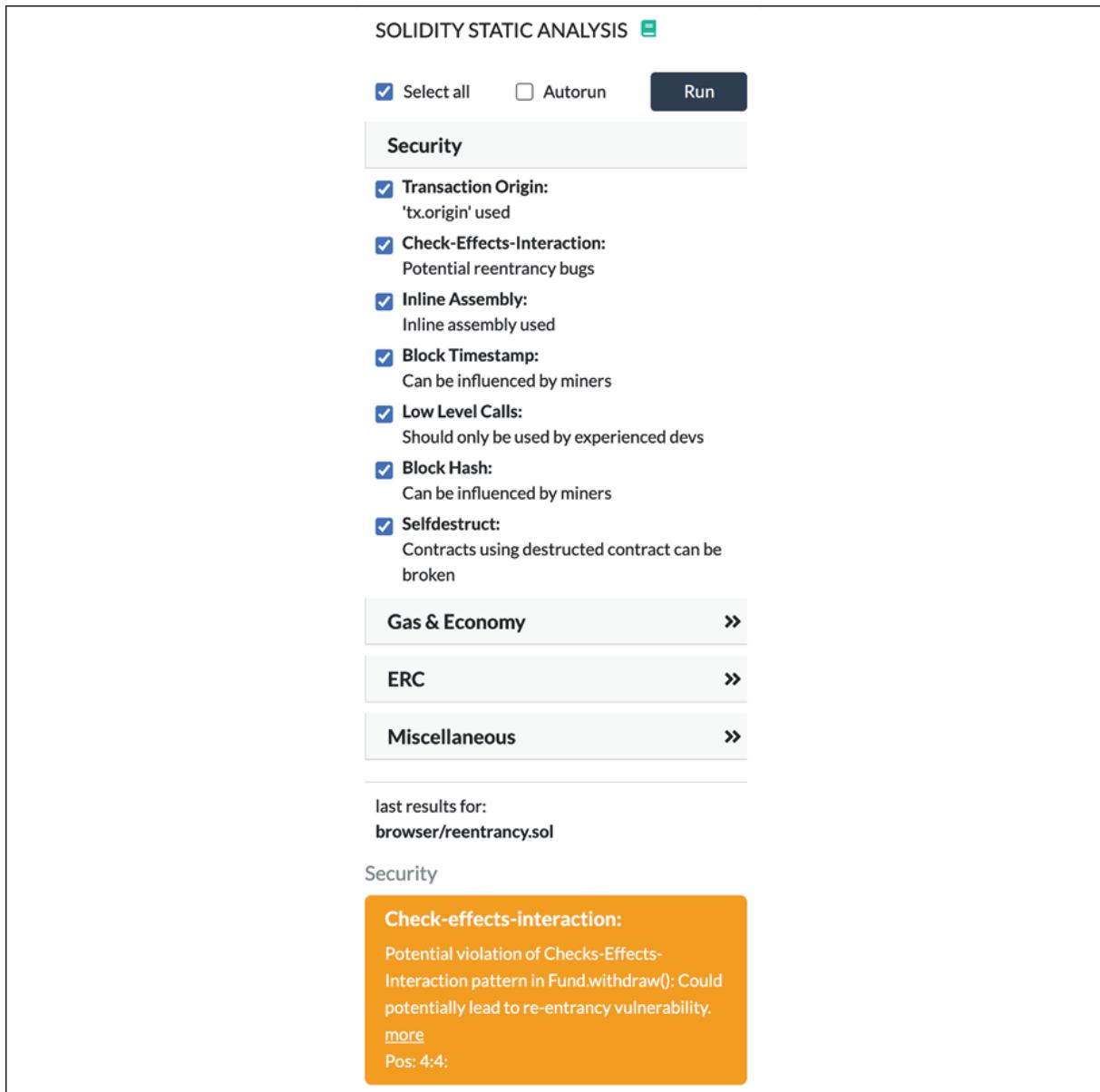


Figure 21.4: Remix IDE analysis options

A sample output of a contract with **reentrancy bug** is shown at the bottom of the preceding screenshot.

Static analysis in Remix IDE analyzes several categories of vulnerabilities, including **Security** and **Gas & Economy**. As shown in the preceding screenshot, the analysis tool has successfully detected the reentrancy bug, details of which are shown at the bottom of the screen.

Other than static analysis capabilities within the Remix IDE, other tools are also available, such as Why3, which we'll discuss next.

Why3

Why3 can also be used for formally analyzing Solidity code.



In the following example, a simple piece of Solidity code that defines the `z` variable as the maximum limit of `uint` is being shown. When this code runs, it will result in returning `0`, because `uint z` will overrun and start again from `0`. This can also be verified using Why3, which is shown here:

A screenshot of the Why3 online compiler interface. On the left, there is a code editor containing the following Solidity code:

```
1 pragma solidity ^0.4.8;
2 contract Overflow {
3     uint z;
4     function x() returns (uint y) {
5         z = 2**256-1;
6         return z+1;
7     }
8 }
```

On the right, there is a results panel with the following text:

This tab provides support for formal verification of Solidity contracts.
This feature is still in development and thus also not yet well documented, but you can find some information [here](#). The compiler generates input. Please paste the text below into <http://why3.lri.fr/try> to actually perform the verification. We plan to support direct integration in the future.

```
(* copy this to http://why3.lri.fr/try/ *)
module UInt256
    use import mach.int.Unsigned
    type uint256
    constant max uint256: int = 0xfffffffffffffffffffffffffffff
    clone export mach.int.Unsigned with
        type t = uint256,
        constant max = max uint256
    end

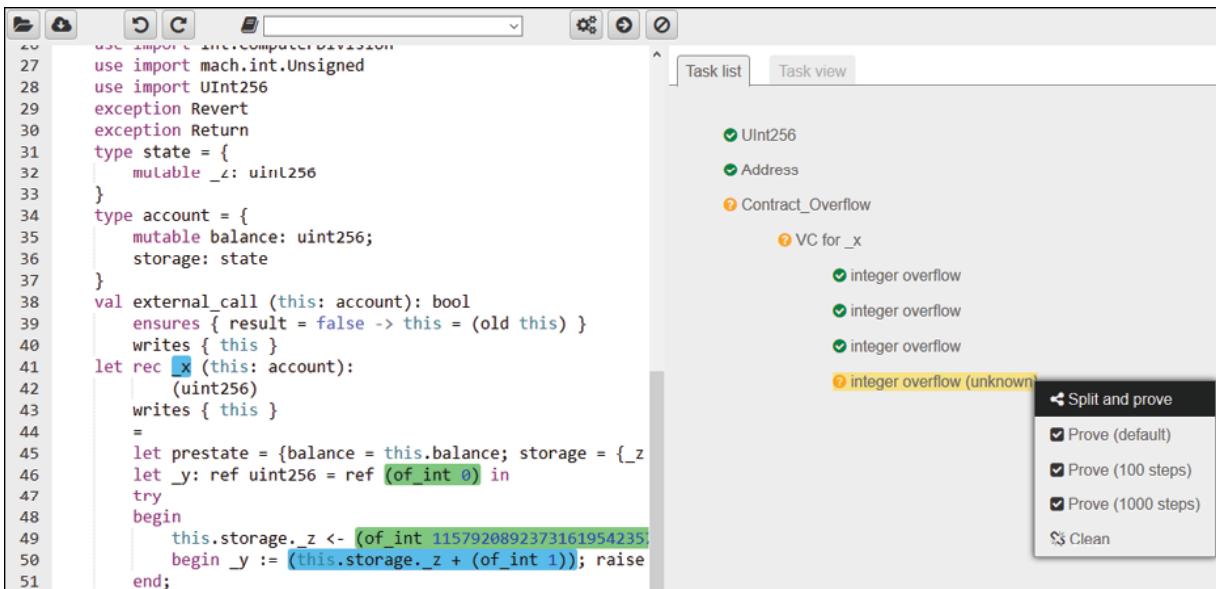
module Address
    use import mach.int.Unsigned
    type address
    constant max address: int = 0xfffffffffffffffffffff
    (* 160 bit = 40 *)
    clone export mach.int.Unsigned with
        type t = address,
        constant max = max address
end
```

Figure 21.5: Solidity online compiler with formal verification

Conversion from Solidity to Why3 compliant code used to be available in Solidity's online compiler, but it is no longer available. Therefore, the following example is only for completeness purposes and to shed light on

an important class of bugs that can go undetected with traditional tools. In this example, integer overflow is shown as an example.

The following example shows that Why3 successfully checks and reports integer overflow errors. This tool is under heavy development but is still quite useful. Also, it should be noted that tool, or any other similar tool, is not a silver bullet. Even formal verification generally should not be considered a panacea. This is because specifications in the first place should be defined appropriately:



The screenshot shows the Why3 IDE interface. On the left, there is a code editor displaying a smart contract in Why3 syntax. The code defines types for state and account, and includes a function that performs a self-call and updates storage. On the right, a 'Task list' panel displays several items: 'UInt256' (green checkmark), 'Address' (green checkmark), 'Contract_Overflow' (yellow question mark), 'VC for _x' (yellow question mark), and three entries under 'integer overflow' (green checkmarks). A context menu is open over the 'Contract_Overflow' item, showing options like 'Split and prove', 'Prove (default)', 'Prove (100 steps)', 'Prove (1000 steps)', and 'Clean'.

```
26 use import int.ComputerDivision
27 use import mach.int.Unsigned
28 use import UInt256
29 exception Revert
30 exception Return
31 type state = {
32     mutable _z: uint256
33 }
34 type account = {
35     mutable balance: uint256;
36     storage: state
37 }
38 val external_call (this: account): bool
39     ensures { result = false -> this = (old this) }
40     writes { this }
41 let rec _x (this: account):
42     (uint256)
43     writes { this }
44 =
45     let prestate = {balance = this.balance; storage = {_z
46     let _y: ref uint256 = ref (of_int 0) in
47     try
48         begin
49             this.storage._z <- (of_int 1157920892373161954235);
50             begin _y := (this.storage._z + (of_int 1)); raise
51         end;
```

Figure 21.6: Why3 analyzing a smart contract for issues

In the preceding screenshot, the Why3 IDE is shown with a smart contract being analyzed for issues, which are reported on the right-hand side. In this case, an **integer overflow** has been found, which is reported.

In the next section, we'll see how a different tool, Oyente, can be used to test a smart contract for security vulnerabilities.

Oyente

Currently, **Oyente** is available as a Docker image for easy testing and installation. It is available at

<https://github.com/melonproject/oyente> and can be downloaded and tested.

In the following example, a simple contract taken from the **Solidity** documentation that contains a reentrancy bug has been tested. First, we will show the code with a reentrancy bug:

```
1 pragma solidity ^0.4.0;
2 contract Fund {
3     mapping(address => uint) shares;
4     function withdraw() public {
5         if (msg.sender.call.value(shares[msg.sender])())
6             shares[msg.sender] = 0;
7     }
8 }
9 }
```

Figure 21.7: Contract with a reentrancy bug, sourced from the Solidity documentation

This sample code contains a reentrancy bug, which basically means that if a contract is interacting with another contract or transferring ether, it is effectively handing over the control to that other contract. This allows the called contract to call back into the function of the contract from which it has been called, without waiting for completion. For example, this bug can allow calling back into the withdraw function shown in the preceding example again and again, resulting in obtaining ETH multiple times. This is possible because the share value is not set to 0 until the end of the function, which means that any later invocations will be successful, resulting in withdrawing again and again.

An example is shown of Oyente running to analyze the contract shown here and, as can be seen in the following output, the analysis has successfully found the reentrancy bug. The bug is proposed to be handled by a combination of the *Checks-Effects-Interactions* pattern described in the Solidity documentation:

```
root@fa9ef6ac8455:/home/oyente/oyente
(venv)root@fa9ef6ac8455:/home/oyente/oyente# python oyente.py a1.sol
Contract Fund:
Running, please wait...
===== Results =====
Callstack Attack: False
THIS IS A CALLLLLLLLL
{'path_condition': [Iv >= 0, init_Is >= Iv, init_Ia >= 0, If(Id_0/
26959946667150639794667015087019630673637144422540572481103610249216 ==
1020253707,
1,
0) != 0, Not(Iv != 0)], 'Is': Is, 'Iv': Iv, 'some_var_1': some_var_1, 'Id_0': Id_0,
'Ia_store_some_var_1': Ia_store_some_var_1, 'Ia': Ia}

This is the global state
{'Ia': {'some_var_1': 0}, 'miu_i': 3L, 'balance': {'Ia': init_Ia + Iv, 'Is':
init_Is - Iv}}
{64: 96, 0: Is & 1461501637330902918203684832716283019655932542975, 32: 0}

CALL params

Is & 1461501637330902918203684832716283019655932542975

Ia_store_some_var_1

=>>>> New PC: []
Reentrancy_bug? True

Added True
Concurrency Bug: False
Time Dependency: False
Reentrancy bug exists: True
===== Analysis Completed =====
(venv)root@fa9ef6ac8455:/home/oyente/oyente#
```

Figure 21.8: Oyente tool detecting Solidity bugs

Oyente is also available in the analysis tools for smart contracts at <https://oyente.melonport.com>. A sample output is shown here:

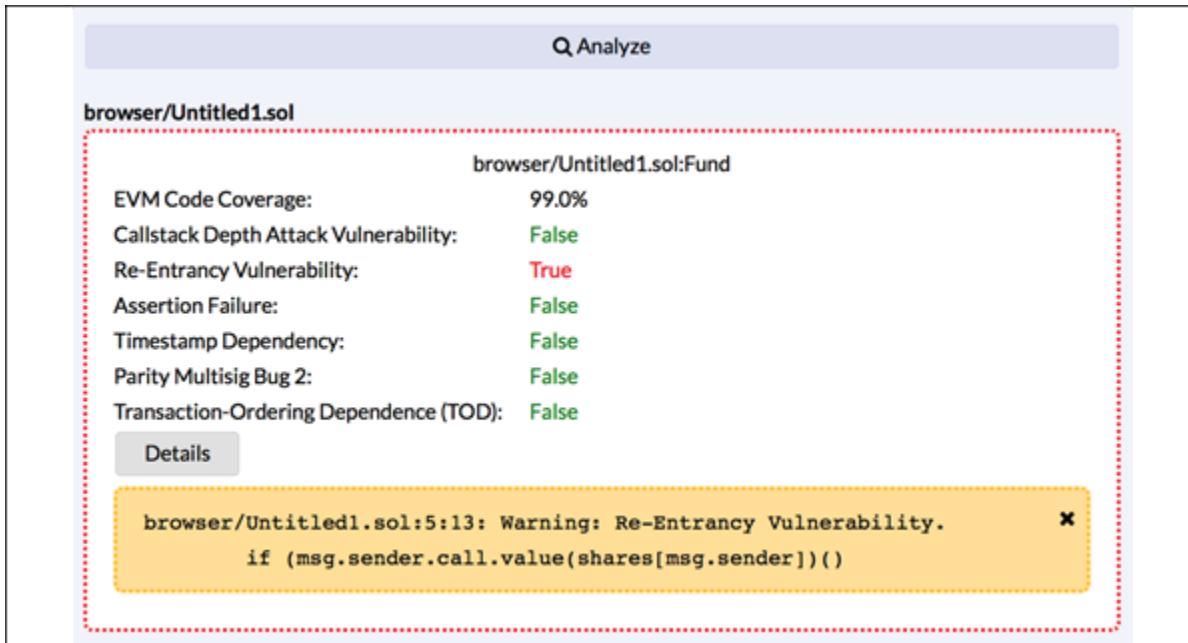


Figure 21.9: Oyente analysis

With this example, we conclude our introduction to security and analysis tools for Solidity. This is a very rich area of research, and more and more tools are expected to be available with time.

There are many experts and researchers in academia and the commercial sector exploring this area and soon, there will be many automated tools available for the verification of smart contracts. There is an online tool available already at <https://securify.ch> that analyzes smart contract code to find security vulnerabilities.

Now, well discuss some other tools that we can use to analyze smart contracts for correctness and security vulnerabilities.

Other tools

Other relevant tools and research in the security arena of blockchain technology include Solgraph, Manticore, and other tools for formal verification of smart contracts.

Solgraph

This tool generates a graph of function control flow of a Solidity contract and finds possible security vulnerabilities. More information on this tool is available here: <https://github.com/rainenorshine/solgraph>.

Manticore

This is a **symbolic execution** framework that can be used for binaries and smart contracts.



More information on Manticore can be found here:

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T. and Dinaburg, A., 2019, November. *Manticore: A user-friendly symbolic execution framework for binaries and smart contracts*. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1186-1189). IEEE.
<https://arxiv.org/pdf/1907.03890.pdf>.

Symbolic execution is a method used to analyze computer programs to determine which part of the computer program executes as a result of what input. In other words, it establishes which input executes which part of the program. In symbolic execution, instead of actual input data, symbolic values are used, which are then used to evaluate the program. Also, an automated theorem prover is used to check whether there are values that result in incorrect behavior of the program or cause it to fail. It is used for debugging, software testing, and security.

Formal verification of smart contracts

A lot of research on formal verification of smart contracts have also been conducted. The proposed techniques include, but are not limited to, model checking, static analysis, dynamic analysis, and verification using proof assistants.

Some of the more relevant papers on the subject include the following:

- Nehai, Z., Piriou, P.Y. and Daumas, F., 2018, July. *Model-checking of smart contracts*. In 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 980-987). IEEE. https://hal.archives-ouvertes.fr/hal-02103511/file/Nehai-Piriou-Dumas%20V18_03_09.pdf.
- Bang, T., Nguyen, H.H., Nguyen, D., Trieu, T. and Quan, T., 2020. *Verification of Ethereum Smart Contracts: A Model Checking Approach*. International Journal of Machine Learning and Computing, 10(4). <http://www.ijmlc.org/vol10/977-AM0059.pdf>.
- Antonino, P. and Roscoe, A.W., 2020. *Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity*. arXiv preprint arXiv:2002.02710. <https://arxiv.org/pdf/2002.02710>.
- Kalra, S., Goel, S., Dhawan, M. and Sharma, S., 2018, February. ZEUS: *Analyzing Safety of Smart Contracts*. In NDSS. http://pages.cpsc.ucalgary.ca/~joel.reardon/blockchain/readings/ndss2018_09-1_Kalra_paper.pdf.
- Amani, S., Bégel, M., Bortin, M. and Staples, M., 2018, January. *Towards verifying Ethereum smart contract bytecode in Isabelle/HOL*. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (pp. 66-77). <https://dl.acm.org/doi/pdf/10.1145/3167084>.
- Chen, X., Park, D. and Roşu, G., 2018, November. *A language-independent approach to smart contract verification*. In International Symposium on Leveraging Applications of Formal Methods (pp. 405-413). Springer, Cham. <http://fsl.cs.illinois.edu/FSL/papers/2018/chen-park-roşu-2018-isola/chen-park-roşu-2018-isola-public.pdf>.

Smart contract security is a very exciting and vast area of research. It is almost impossible to cover all the aspects in this short chapter. You are



encouraged to read the introduction to formal verification earlier in this chapter and then go through some of the preceding papers for further research.

After all this discussion on smart contract security, we should now understand its significance. While all the issues and techniques discussed here to mitigate the security issues in smart contracts are valuable, a question still arises about what else we can do, given that smart contract security is a sensitive topic that can result in serious financial losses. The answer could be that we treat smart contracts and the underlying blockchain as safety-critical systems, and then apply all attributes of safety-critical systems to smart contracts. From an engineering perspective, it will become a subject of safety-critical systems and will be treated as such.

This idea might sound a bit over the top, but imagine if, one day, a smart contract is responsible for generating an event that would trigger a shutdown as a result of overheating in a nuclear reactor. Again, a little exaggerated perhaps, but entirely possible: with IoT and blockchain convergence such scenarios might become a reality, where nuclear reactors are running using a blockchain as a mechanism to control and track associated operations.

Now, we'll introduce some other challenges that blockchain technology faces, which should be resolved in order to enable the mainstream adoption of blockchain.

Other challenges

In addition to key issues such as privacy and scalability, there are a few other challenges that can hinder the adoption of blockchain as a mainstream technology. We'll discuss these challenges and some possible solutions next.

Interoperability

We discussed interoperability briefly earlier in this book. It is sufficient to say here that interoperability is an open challenge and while many initiatives such as **Cosmos**, **Polkadot**, and **Interledger** are in progress to address this limitation, there is still a long way to go. One of the approaches is standardization of blockchain protocols, thus making them compatible with one another. Another approach is to develop protocols that will make existing blockchains interact with each other, such Interledger.

We cover Cosmos and Interledger in detail in this book's bonus content pages, here https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf. In this section, we will briefly describe Polkadot.

Polkadot

Polkadot is a platform that allows connectivity between different private and public chains, oracles, and many other technologies. It is a platform used to enable Web3, a decentralized web, and allows any type of data to be exchanged between any type of blockchain. It offers heterogenous sharding, scalability, upgradability, transparent governance, and cross-chain composability.

The key concept behind Polkadot is a *relay chain*, which is responsible for cross-chain interoperability, consensus, and overall network security. Polkadot allows communication between different independent blockchain shards called *parachains*. Parachains connect to the relay chain after paying a fee. Other components include *bridges*, which are a special type of blockchain that enable connectivity of parachains with outside networks such as Bitcoin and Ethereum.

From a consensus perspective, there are several elements in Polkadot. *Validators* secure the chain by staking tokens called DOTs. DOTs provide governance, staking, and bonding. DOT holders are allowed to participate in governance activities such as protocol upgrades. Staking with DOTs provides a mechanism to ensure the security and stability of the Polkadot network. Also, bonding using DOTs allows new parachains to be added to

the network. Overall, in the Polkadot network, *validators* secure the chain and are selected by *nominators* due to their trustworthiness. *Collators* maintain shards, and so-called *fishermen* operate on the network for monitoring and policing purposes.



More details on Polkadot are available here:
<https://polkadot.network/>.

Lack of standardization

Lack of standardization leads to disparate systems that are unable to communicate with each other. This leads to interoperability issues. Therefore, there is a need to develop a common understanding of the requirements in order to develop blockchain platforms to the same standard. Doing this will ensure that all platforms are interoperable.

Post-quantum resistance

With the advent of quantum computers, it is envisaged that, soon, cryptography used in blockchain will be at the risk of becoming less secure. This is due to the ability of quantum computers to work at phenomenal speed, allowing cryptanalysis of most of the schemes currently in use. There is already some work being done in order to secure blockchains against post-quantum attacks, but there is still a lot to do.



Some of the proposed solutions are listed here:

The quantum-resistant ledger can be found at:
<https://www.theqrl.org>.

Moreover, efforts from NIST are underway, which can be explored here:
<https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization>.

Ethereum 2.0 is also ensuring that its chosen cryptography is either quantum-secure or pluggable with quantum-secure counterparts when available.



An article that explores quantum safety for Ethereum 2.0 is available here:
<https://blog.ethereum.org/2015/12/24/understanding-serenity-part-i-abstraction/>.

Also, quantum-resistant signature schemes such as **Extended Merkle Signature Scheme (XMSS)** and SPHINCS are being investigated for Ethereum 2.0. The beacon chain's RANDAO is also expected to be quantum-resistant.

Compliance and regulation

Compliance and regulation are important aspects. In traditional IT systems, compliance and regulation play a vital role in ensuring that data and information are handled in accordance with the policies, rules, and standards set by regulatory authorities. Adhering to such standards minimizes the possibility of security breaches, unauthorized access to data, and other issues stemming from the improper handling of data. Now, blockchain can, in fact, help to achieve compliance with certain regulations, such as GDPR and other use cases, where tamper-resistant auditing of logs is required.

A lot of work has already been done in the context of cryptocurrencies and tokens, but not so much from the underlying blockchain perspective. Treating cryptocurrencies and tokens as securities (allowing them to be governed by existing securities laws) is one thing, but regulating the underlying blockchain technology in terms of adherence to security standards, data sovereignty, and information-sharing regulations is another. We discussed compliance in detail in *Chapter 20, Enterprise Blockchains*.

With this, we have completed our examination of different challenges that blockchain faces, and also discussed some of the current solutions available

to address these challenges during the next stages of blockchain evolution.

Summary

In this chapter, we introduced the various challenges facing blockchain technology. A major challenge, blockchain's limited scalability, was discussed, along with several methods available to achieve greater scalability. We approached these methods using different models, which divide the blockchain into planes, or layers, that can be improved in different ways.

We also discussed privacy, which is one of the major limiting factors toward adopting public blockchains for various enterprise use cases. Next, we explored smart contract security and various tools to improve it, such as model checking. Formal verification is a deep and extensive subject, but a brief introduction to various aspects was given, which should serve as solid ground for further research in this area. We also saw examples of formal verification to have an idea of what tools are available.

Finally, we discussed some other challenges to consider for the future of blockchain, including the issue of standardization and pluggability, and the inevitable need for quantum resistance.

In the next and final chapter of this book, we will explore the current landscape and future prospects of the blockchain technology.

22

Current Landscape and What's Next

Blockchain technology is changing rapidly, and it will continue to affect the way we conduct our day-to-day business as new developments emerge. It has challenged existing business models, and promised great benefits such as cost-saving, efficiency, and transparency. So far in this book, we've explored the technical underpinnings of blockchain technology, such as cryptography, consensus mechanisms, and distributed systems concepts. We've looked at cryptocurrencies such as Bitcoin, Litecoin, and Zcash, explored Ethereum in detail, and learned how to write smart contracts and DApps. We also saw applications of blockchain technology in the **Internet of Things (IoT)** and the finance, government, and media sectors, and learned how blockchain technology can address challenges in these industries and others. Moreover, we examined Hyperledger, and some innovative enterprise blockchains such as Quorum and Corda. Finally, we looked at some scalability, privacy, and security challenges faced by proponents of blockchain technology, and considered how these challenges can be tackled.

This chapter will explore the latest developments, emerging trends, issues, and future predictions about blockchain technology. We'll also present some topics related to open research problems and improvements related to blockchain technology. Along the way, we'll cover the following main topics:

- Emerging trends
- Areas to address
- Blockchain research topics

- Blockchain and AI
- The future of blockchain

With the advancement of blockchain technology and the rigorous research being conducted in this space, various developments have been seen in recent months and years. We'll explore these emerging trends in the next section.

Emerging trends

Blockchain technology is in a period of rapid change and intense development, due to the deep interest in it by the academic and commercial sectors. As the technology has become more mature, a few trends have started to emerge. For example, **private blockchains** have recently gained a lot of attention due to their specific use cases in finance organizations. Similarly, **enterprise blockchains** aim to develop blockchain solutions that meet enterprise-level efficiency, security, and integration requirements. For an in-depth exploration of the technical details and features of some alternative blockchains, and their relevant platforms, take a look at this book's bonus online content pages here: https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf. These are out of the scope of this chapter, but it is highly recommended to access these pages to explore some recent blockchain solutions in more detail. Now, some of the broader trends are listed and discussed in the following sections.

New implementations of blockchain technology

Since the invention of Bitcoin, new blockchains are being developed at a rapid rate. We'll introduce some of these new trends as follows.

Application-specific blockchains

Currently, an inclination toward **application-specific blockchains (ASBCs)** is evident, whereby a blockchain or distributed ledger is specially developed or makes use of an already existing blockchain platform but with only one application in mind and is focused on a specific industry.

For example, **Everledger** (<https://www.everledger.io>) is a blockchain solution that has been developed for the sole use of providing an immutable tracing history and audit trail for diamonds and other high-value items. This approach thwarts any fraud attempts because everything related to ownership, authenticity, and item value is verified and recorded on the blockchain. This ability would be invaluable for insurance and law enforcement agencies were it to be widely adopted.

Another example of an ASBC is **Digital Trade Chain (DTC)**, in which a group of seven banks agreed to simplify the trade finance process by directly connecting the parties involved in a trade transaction.



DTC-related information is available at:
<http://www.bankingtech.com/2017/10/ibm-and-eight-banks-unleash-we-trade-platform-for-blockchain-powered-commerce/>.

Concrete, real-life, end-to-end implementations of blockchain technology are also now becoming available, even in the finance industry, such as **Australian Securities Exchange (ASX)** replacing its legacy post-trade system (its system to support trade processing after completion, also referred to as trade settlement) with blockchain.



Further information regarding the ASX project is available at:
<https://www.asx.com.au/services/chess-replacement.htm>.

Start-ups

In recent years, many technology start-ups have emerged that are working on blockchain projects. There has also been a significant increase in the number of start-ups that are offering blockchain consultancy services and solutions specific to this technology.



You can visit: <https://angel.co/blockchains>, which currently shows a list of 4,961 blockchain start-ups.

Technology improvements

As well as innovative implementations of blockchain technology, a lot of effort is being made to improve the technology and infrastructure itself. Some of these efforts are listed in the following sections.

Standardization

Blockchain technology is not yet mature enough to be able to readily integrate or interact with existing systems or other blockchains, and as it stands, no two blockchain networks can easily interact with one another. Standardization will help to improve interoperability, adoption, and integration aspects of blockchain technology.

Some attempts have been made recently to address this, the most notable of which being the establishment of ISO/TC 307, which is a technical committee with the aim of standardizing blockchain and associated distributed ledger technology. The aim of the committee revolves around increasing interoperability and data interchange between users, applications, and systems.

Hyperledger, on the other hand, has a reference architecture that can be used to build blockchain systems based on the same modular architecture. Hyperledger is supported by the Linux Foundation and many other participants from the industry, which is a good omen for the progress of blockchain standardization.

Some work on smart contract templates was started with the publication of a seminal paper authored by Christopher D. Clack, Vikram A. Bakshi, Lee Braine et al., which formally defined smart contract templates. It also presented a vision for future research and stipulated certain necessities in smart contract development.



The paper, *Smart Contract Templates: foundations, design landscape and research directions*, produced by Clack et al. is available at <https://arxiv.org/abs/1608.00771v2>.

All of the efforts mentioned here are a clear indication that in the near future, clearly defined standards will emerge that will further advance the adoption of blockchain technology. Standardization will also result in exponential growth of the blockchain industry by eliminating hurdles such as interoperability.

Consortia

Recent years have seen the startup of various consortia and shared open source efforts. This trend is expected to grow in the coming years, and more and more consortia, committees, and open source efforts towards blockchain advancement will likely emerge soon. A prime example is **R3**, which developed **Corda** with a consortium of some of the world's largest financial organizations.

Enhancements

Various enhancements and suggestions to further develop existing blockchains have been made over the last few years. Most of these suggestions have been made in response to specific security vulnerabilities and to address inherent limitations in blockchain technology. There are certain limitations in blockchain technology, such as scalability, privacy, and interoperability, which need to be addressed before it can be adopted as a mainstream field of technology.

Recently, there have been tremendous efforts made toward addressing **scalability** issues in blockchain technology, which were discussed in *Chapter 21, Scalability and Other Challenges*. Also, blockchain-specific improvement proposals such as **Bitcoin Improvement Proposals (BIPs)** and **Ethereum Improvement Proposals (EIPs)** are regularly made by developers to address various concerns in these systems.

Moreover, recent advancements such as **state channels**, an off-chain scalability solution discussed in *Chapter 21, Scalability and Other Challenges*, are evidence of the fact that blockchain technology is improving rapidly, and will, in time, evolve into a mature and practical technology. Using state channels, the **Bitcoin Lightning** network and Ethereum's **Raiden** have been developed to enable near-instant, low-fee, scalable payments that could help address the scalability issue on future blockchain projects.



The Raiden Network client and smart contracts were released for Ethereum's mainnet in May 2020. More details on the Raiden Network can be found here: <https://github.com/raiden-network/raiden>.

Limitations around the **interoperability** of blockchains have also resulted in changes oriented toward the development of systems that can work across multiple blockchains. Some examples of these systems are Interledger, Polkadot and Cosmos. Interledger has also been implemented in Java as Hyperledger Quilt:

<https://www.hyperledger.org/use/quilt>.

Moreover, various new blockchain solutions to the **privacy** issue have emerged, such as Kadena, which directly addressed blockchain confidentiality issues. Other concepts such as Zcash, CoinJoin, Zether, bulletproofs, and confidential transactions have also been developed to address existing blockchain privacy challenges.

Another necessary enhancement is to reduce electricity consumption. It is evident from the global electricity usage of Bitcoin's blockchain that **Proof of Work (PoW)** mechanisms are very inefficient. Of course, this

computation secures the Bitcoin network, but there is no other benefit of this computation, and it wastes a lot of electrical energy. To reduce this waste, there has been more focus on greener options such as **Proof of Stake (PoS)** algorithms, which do not need enormous reserves of resources, like Bitcoin's PoW algorithm does. This trend is expected to grow, especially with the implementation of a PoS mechanism in Ethereum 2.0.

This trend of addressing technical challenges is expected to continue to develop in years to come, and even if almost all fundamental challenges are addressed in blockchain technology, further enhancements and optimization research will continue.

Ongoing research and study

Blockchain technology has stimulated intense research interest, both in academia and the commercial sector. In recent years, interest has dramatically increased, and now major institutions and researchers around the world are exploring this technology. This growth in interest is primarily because blockchain technology can help to make businesses and their operations more efficient, cheaper to run, and more transparent.

The bulk of academic interest lies in addressing problems in the fields of cryptography, consensus mechanisms, and performance limitations in blockchains. As blockchain technology comes under the broader umbrella of distributed systems, many academic researchers focused on distributed computing research have focused their research on blockchain technology. For example, University College London (UCL) has a dedicated department, the UCL Research Centre for Blockchain Technologies, which focuses on blockchain technology research.

Another example of academic groups conducting blockchain research is the ETH Zurich distributed computing group (<https://disco.ethz.ch>), who have published seminal research regarding blockchain technology. A recent journal called the *Ledger Journal* has recently published its first issue of research papers. It is available at <http://www.ledgerjournal.org/ojs/index.php/ledger>.

Another organization, called the **Initiative for Cryptocurrencies & Contracts (IC3)**, is also researching smart contract and blockchain technologies. IC3 aims to address performance, confidentiality, and safety issues in blockchains and smart contracts, and runs multiple projects to address these issues.



More information about IC3's current projects is available online at
<http://www.initc3.org/>.

As well as those mentioned here, there are now teams and departments dedicated to blockchain research and development in various academic and commercial institutes. Although the initiatives mentioned here are not an exhaustive list by any stretch of the imagination, it is still a solid indication that this is a subject of extreme interest for researchers, and more research and development is expected to be seen in 2020 and beyond.

With the progress of blockchain adoption, research in different areas of cryptography has been conducted, with the aim of addressing challenges such as privacy. Some of these research directions are discussed in the following sections.

Cryptography

Even though cryptography was an area of keen interest and research for many decades before Bitcoin's invention, blockchain technology has resulted in a renewed interest in this field. With the advent of blockchains and related technologies, there has been a simultaneous and significant increase in the interest in cryptography. New research is being carried out and published regularly, especially in the area of financial cryptography.

Technologies such as **zero-knowledge proofs (ZKPs)**, fully **homomorphic encryption**, and functional encryption are being researched for their potential use in blockchains. Already, ZKPs have been implemented for the first time at a practical level in the form of Zcash, which has addressed privacy issues in cryptocurrency networks. It's evident that blockchains and

cryptocurrencies have helped with the advancement of cryptography, especially financial cryptography.

Other developments such as Zether, **zk-SNARKs**, **zk-STARKs**, bulletproofs, the Aztec protocol (<https://www.aztecprotocol.com>), Nightfall and Zokrates (<https://zokrates.github.io>) are only the foundations of what the future holds for blockchain cryptography!

Cryptoeconomics

New fields of research are emerging with blockchains, most notably, cryptoeconomics, which is the study of protocols governing the decentralized digital economy. With the continuing development of blockchains and cryptocurrencies, research in this area has also grown. Cryptoeconomics has been defined by Vitalik Buterin as a combination of mathematics, cryptography, economics, and game theory.



There is an excellent presentation by Vitalik Buterin on cryptoeconomics available at <https://www.crypto.berlin/introduction-to-cryptoeconomics-by-vitalik-buterin>.

Hardware development

When it was realized in 2010 that contemporary methods were no longer efficient for mining bitcoins, miners shifted toward optimizing available mining hardware. These initial efforts included the use of GPUs, and then **Field-Programmable Gate Arrays (FPGAs)** were used after GPUs reached their hash rate limit.

Very quickly after that, **Application-Specific Integrated Circuits (ASICs)** emerged, which increased mining power significantly. This trend is expected to grow further with research into optimizing ASICs by parallelizing and decreasing their die sizes. Such optimizations will make ASICs even faster and more efficient.

Moreover, GPU programming initiatives are also expected to grow, with the regular emergence of new cryptocurrencies, many of which make use of PoW algorithms that can benefit from GPU processing capabilities. For example, recently Zcash has spurred interest in GPU mining rigs and related programming using NVIDIA CUDA and OpenCL. The aim with Zcash is to use multiple GPUs in parallel to optimize mining operations.

Also, some research has been done in the field of trusted computing hardware such as Intel's **Software Guard Extensions (SGX)** to address security issues on blockchains. Intel's SGX has also been used in a novel consensus algorithm called **Proof of Elapsed Time (PoET)**, which was discussed in *Chapter 17, Hyperledger*. Another project, the **21 Bitcoin Computer**, was developed, which serves as a platform for developers to learn about Bitcoin technology and develop applications for the Bitcoin platform.

Hardware research and development is expected to continue, and soon many more hardware applications will be explored. Some examples include **physical unclonable functions**, the acceleration of blockchain processes (such as cryptography), IoT hardware, and hardware security module integration for key management.

Formal methods and security

With the realization of security issues and vulnerabilities in smart contract programming languages, and generally in blockchain protocols, there is now a keen interest in the formal verification and testing of smart contracts and other blockchain components before production deployments. For this, various efforts are already underway, many of which we discussed in *Chapter 21, Scalability and Other Challenges*.

New programming languages

There is also an increased interest in the development of programming languages for developing smart contracts. Efforts are more focused on domain-specific languages, for example, **Solidity** for Ethereum and **Pact**

for Kadena. This is just a start, and many new languages are likely to be developed as the available technology advances.

Education and employment within blockchain

While blockchain technology has spurred a great interest among technologists, developers, and scientists throughout almost every industry around the world, initially there was a lack of formal learning resources and educational material available for aspiring blockchain professionals.

However, there has been a recent trend emerging in job markets, whereby recruiters are seeking blockchain programmers and developers. This is especially relevant to the finance industry, in which many start-ups and even large organizations have started to hire blockchain specialists.

There is also concern about the lack of blockchain developers, which will undoubtedly be addressed as the technology progresses, and more developers either gain experience on a self-learning basis or gain formal training from training providers. Consequently, many educational institutes around the world now offer certificates and courses with a focus on blockchain technology. This trend is, of course, expected to grow as the technology matures and is applied in more commercial use cases.

Innovative blockchain applications

Some new ideas for improving the efficacy of blockchains have emerged with the unprecedented development of blockchain technology. Some of these are listed in the following sections.

Blockchain as a Service

With the current maturity level of cloud platforms, many companies have started to provide **Blockchain as a Service (BaaS)**. The most prominent examples are Microsoft's Azure, in which the Ethereum blockchain is provided as a service, and IBM's Cloud platform, which provides IBM's blockchain as a service. This trend is only expected to grow in the next few

years, and more companies will likely emerge that provide BaaS to consumers.

Electronic Government as a Service (eGaaS), which is in fact a form of BaaS, provides application-specific blockchains for governance functions (<http://egaas.org>). This project aims to organize and control activities without document circulation and bureaucratic overhead.

Convergence with other technologies

The convergence of other technologies with blockchain offers major benefits to both technologies. At their core, blockchains provide resilience, security, and transparency, which, when combined with other technologies, results in a very powerful complementary technology. For example, the IoT stands to gain major benefits such as integrity, decentralization, and scalability when implemented via a blockchain. **Artificial Intelligence (AI)** is expected to gain numerous benefits from the implementation of blockchain technology: in fact, within blockchain technology, AI can be implemented in the form of **Autonomous Agents (AAs)**, which can make rational decisions on behalf of humans. Some ideas of the benefits that the convergence of AI and blockchain can provide are listed as follows:

- The blockchain can share machine learning models in a secure and **peer-to-peer (P2P)** manner.
- The blockchain can serve as an auditing layer for decisions made by AI.
- As AI and machine learning capabilities grow, blockchain can provide a mechanism to monitor and control any malicious behavior that an AI may manifest. While AI can be used for good, it can also be used by malicious actors. In this case, the blockchain can serve as a control mechanism to thwart any attacks. For example, all agents on a blockchain can be controlled under a consensus mechanism, and malicious Byzantine behavior can be handled as per the blockchain consensus protocol.
- AI can also benefit blockchain technology by enabling artificially intelligent smart contracts.

- AI can be applied to other components of a blockchain to achieve, for example, adaptive consensus mechanisms. These can, based on network conditions, readjust fault-tolerance requirements and as a result enable a more efficient consensus mechanism.

There are of course many more examples of innovative applications of blockchain in combination with the aforementioned technologies. Some of these will be discussed in detail in the *Blockchain and AI* section of this chapter.

Alternatives to blockchains

With the advancement of blockchain technology in recent years, researchers have started to think about the possibility of creating platforms that can provide guarantees and services that a blockchain provides, but without the need for an actual blockchain. This has resulted in the development of R3's **Corda**, which is not really a blockchain, since it is not based on the concept of blocks containing transactions. Instead, it is based on the concept of a state object that transverses throughout the Corda network, according to the requirements and rules of the network participants that represent the latest state of the network.

Other examples include **IOTA**, an IoT blockchain that makes use of a **Directed Acyclic Graph (DAG)** as a distributed ledger named **Tangle**, instead of a conventional blockchain formed of blocks. This ledger is claimed to have addressed scalability issues and implemented high-level security features that even protect against quantum computing-based attacks. It should be noted that Bitcoin is also somewhat protected against quantum attacks, because quantum attacks can only work on exposed public keys, which are only revealed on the blockchain if both **send** and **receive** transactions are made. Therefore, if the public key is not revealed, which is the case in unused addresses or addresses that may have only been used to receive bitcoin, then quantum safety can be guaranteed. In other words, using a different address for every transaction protects against quantum attacks (to a degree). Also, in Bitcoin, it is relatively easy to change to another quantum signature protocol if required.

Hedera Hashgraph is another alternative to traditional blockchain. It is a distributed ledger that runs an **asynchronous Byzantine fault tolerant (ABFT)** protocol for achieving consensus. It is claimed to be a faster and more efficient alternative to traditional sequential blockchain.



More information about Hedera is available here:
<https://www.hedera.com>.

Some debatable ideas

As with every new idea, some debate is inevitable. Some of the topics that have sparked debate are discussed as follows.

Public versus private on the blockchain

Although private and enterprise blockchains are seen as a solution to achieving privacy, and a way to address other enterprise concerns such as scalability and governance, some blockchain purists are not convinced, since centralization is viewed as a deviation from the core philosophy of blockchains. Decentralization proponents envisage that in the future, public blockchains will be able to provide all of the features that are desirable in enterprise use cases such as privacy, scalability, and governance. Using the Ethereum mainnet for enterprise use cases is also possible—there have been some high-profile activities performed on the public Ethereum network, such as bond settlement.



More on the use of Ethereum for high-profile settlements can be found here:
<https://www.coindesk.com/santander-settles-both-sides-of-a-20-million-bond-trade-on-ethereum>.

With Ethereum 2.0, the need for enterprise blockchains may diminish as the mainnet becomes capable of providing all required enterprise services. Also, with the availability of privacy solutions such as Nightfall, Aztec, and

Zether, public blockchain may become the tool of choice for enterprise use cases one day, and private blockchains may not be as necessary as they were initially thought to be.

On the other hand, there are a number of advantages that private and enterprise blockchains provide, as discussed in *Chapter 20, Enterprise Blockchain*. Therefore, it is not entirely true that private blockchains are redundant. Decentralization is seen as a concern by enterprises, and public chains, with their fully decentralized nature, are deemed unsuitable for enterprise use cases. In traditional systems or enterprise blockchains, participants can be held accountable for their actions, but in public blockchains, there is no centralized control or governance, meaning that in cases of disputes, nobody can be held accountable.

This clash of interests might change, however, if public blockchains could introduce an optional governance mechanism to support enterprise governance needs. This feature could be achievable using smart contracts, where a governance layer is provided on top of the underlying blockchains in order to meet enterprise requirements such as access control, auditing, and security.

Central bank digital currency

With the advent of blockchain and tokenization, there is now a lot of interest in digital currencies issued by central banks. This is a somewhat controversial subject amongst regulators, technologists, and economists, but nevertheless something that could soon be a reality. The key difference between cryptocurrencies and **central bank digital currency (CBDC)** is that CBDC is issued by a central bank as legal tender, declared by a country's government. It is a digital form of fiat money, whereas cryptocurrency is more of a decentralized token of value, which is not backed by government regulation, law, or any monetary body.



China has launched a CBDC project called **Digital Yuan**. More information is available here: <https://www.coindesk.com/coindesk-50-how-peoples-bank-china-became-cbdc-leader>.



The Swedish government has launched a currency digitization project called e-krona. More information on this project is available here:
<https://www.riksbank.se/en-gb/payments--cash/e-krona/>.

As well as the topics we've discussed in this section, there are some further notable projects and relevant development tools that have emerged as a result of rapid development in blockchain technology that are beyond the scope of this chapter. Readers can explore some of these by referring to the bonus content repository for this book at https://static.packtcdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf.

In the next section, we'll introduce some challenges and areas of concern that need to be addressed in order to achieve the mainstream adoption of blockchain technology.

Areas to address

Apart from security and privacy, which were discussed at length in *Chapter 21, Scalability and Other Challenges*, several other less technology-based obstacles should be addressed before broad adoption of blockchains can be achieved. We'll introduce some of these more specific areas to address in the following sections.

Regulation

Regulation is considered one of the most significant challenges to blockchain development. The core issue is that blockchains, and their associated cryptocurrencies, are not recognized as legal systems or forms of currency by any government. Even though in some cases, blockchain tokens have been classified as having monetary value, for example in the US and Germany, they are still far from being accepted as regular currencies.

Moreover, blockchains in their current state are not recognized by financial regulatory bodies as platforms that are stable or reliable enough to be used by financial institutions.

There have been, however, various initiatives proposed by regulatory authorities around the world to research and set regulations. Bitcoin in its current state is fully unregulated, even though some attempts have been made by governments to tax it. In the UK, under the EU VAT directive, Bitcoin transactions are exempt from **Value Added Tax (VAT)**. This may change after the finalization of Brexit, but **Capital Gains Tax (CGT)** may still be applicable in some scenarios. Some attempt by financial regulatory authorities to regulate blockchain technology is expected very soon, especially after the recent announcement by the UK's **Financial Conduct Authority (FCA)** that it may approve some companies that are using blockchain for their business services.

Another regulatory concern is that blockchain technology is not ready for production deployments. Even though the Bitcoin blockchain has evolved into a solid blockchain platform and is used in production, it is not suitable for every scenario. This is especially true in the case of sensitive environments such as finance and health. However, this situation is changing very quickly and this chapter has already explored various examples of new blockchain projects that have been implemented in real life, such as the ASX blockchain post-trade solution. This trend is expected to grow as efforts are made to improve the associated technology, and address technical limitations such as scalability and privacy.

Security is also another general concern that has been highlighted by many researchers, which is especially applicable to the finance and health sectors. A report by the **European Union Agency for Network and Information Security (ENISA)** has highlighted some distributed ledger-specific concerns, including the need for regulation, auditing, control, and governance.



The report is available at
<https://www.enisa.europa.eu/news/enisa-news/enisa-report-on-blockchain-technology-and-security>.

Some concerns highlighted in the report also include smart contract management, key management, **Anti Money Laundering (AML)** regulations, and anti-fraud tools. Clearly, while blockchain is generally seen as a solution to many technical business challenges, it can also be misused. In the next section, we'll see some scenarios in which cryptocurrencies and blockchain networks are being used for illegal activities.

Illegal activity

With the key attributes of censorship-resistance and decentralization, blockchain technology can help to improve transparency and efficiency in many walks of life. However, the somewhat unregulated nature of blockchain technology means that it has the potential to be used for illegal activity. For example, consider a scenario where illegal content is published over the internet. Normally, it can be immediately shut down by approaching the relevant authorities and website service providers, but this is not possible in blockchains. Once something is there on the blockchain, it is almost impossible to revert.

This means that any unacceptable content, once published on the blockchain, cannot be removed. If the blockchain is used for distributing immoral or illegal content, then there is no way for anyone to prevent it. This fact poses a serious challenge, and it seems that some regulation and control would be beneficial in this scenario, but it provokes the critical question: how can a blockchain be regulated? In this case, it may not be prudent to create regulatory laws first, and then see if blockchain technology adapts, because this might disrupt innovation and progress. It may be more sensible to let the blockchain technology grow first in a similar, organic manner to the internet. Then, when its user base reaches a critical mass, governing bodies can call for the application of regulations around the implementation and usage of blockchain technology.

There are various real-life examples where the **dark web** is used in conjunction with **Bitcoin** to perform illegal activities.



Dark web is a term used to describe different networks that exist on the internet but necessitate the use of special hardware, programs, configuration, or credentials for access. More on the dark web can be found here: https://en.wikipedia.org/wiki/Dark_web.

One example is the Silk Road online marketplace, which was used to sell illegal drugs and other contraband over the internet. It used Bitcoin for payments and was hosted on the dark web using URLs only visible with a browser called Tor. Although the Silk Road was shut down after years of effort by law enforcement agencies, other sites have emerged that offer similar services; as such, this activity remains a major concern. Imagine that an illegal website was hosted on IPFS, a P2P distributed and decentralized storage network built on a blockchain; there would be no easy way of shutting it down. It is clear that the absence of control and regulation can encourage illegal activity, meaning criminal enterprises like the Silk Road will keep appearing. Further development of totally anonymous transaction capabilities such as Zcash, while useful in various legitimate scenarios, could provide another layer of protection for criminals.

Clearly, it depends on who is using the technology; anonymity can be good in many scenarios, for example in the health industry, where patient records should be kept private and anonymous. However, it may not be appropriate if it can also be used by criminals to hide their activities. One solution might be to introduce intelligent bots, AAs, or even contracts that are programmed and embedded with regulatory logic. They would most likely be programmed by regulators and law enforcement agencies, and live on the blockchain as a means to provide governance and control. For example, a blockchain could be designed in such a way that every smart contract has to go through a controller contract, which scrutinizes the code logic and provides a regulatory mechanism to control the behavior of the contract.

It may also be possible to get each smart contract's code to be inspected by regulatory authorities, and once a smart contract's code has a certain level of authenticity attached to it in the form of certificates issued by a regulator, it will be deployed on the blockchain network. This concept of **binary signing** is akin to the already established concept of **code signing**, whereby executables are digitally signed as a means to confirm that the code is bona

fide and not malicious. This idea is more applicable in the context of semi-private or regulated blockchains, where a certain degree of control is required by a regulatory authority; for example, in finance. It means, however, that some degree of trust is required to be placed in a trusted third-party regulator, which may not be desirable due to the deviation from full decentralization.

However, to address this, the blockchain itself can be used to provide a decentralized, transparent, and secure certificate issuing and digital signing mechanism.

Privacy or transparency

There needs to be a fine balance between privacy and transparency. Too much transparency can result in undermining personal privacy, however, too much privacy can result in the loss of transparency. The choice between privacy and transparency is use case dependent, however, a state of equilibrium is highly desirable to provide the best solution for both requirements.

Now, let's consider some research topics that are currently being explored by researchers in this space. Any student or researcher could use this list of topics as a starting point for their research.

Blockchain research topics

While some major innovations have been made in blockchain technology in recent years, the area is still ripe for further research. Some selected research topics are listed in this section, with some information about existing challenges and the latest developments. Some ideas are also presented on how to address the issues facing these topics.

Smart contracts

Significant progress has been made in this area, aimed at defining the key requirements of smart contracts, and developing templates for their production. However, further research is required in this area, in order to make smart contracts safer and more secure.

Cryptographic function limitations

Cryptography used in the Bitcoin blockchain is exceptionally secure and has stood the test of time. In other blockchains, similar security techniques are used, which are also very secure. However, specific security issues, such as the possibility of the generation and use of duplicate signature nonces in elliptic curve digital signature schemes (leading to private key recovery attacks), collisions in hash functions, and vulnerability to quantum attacks (that could break the underlying cryptographic algorithms), remain an exciting area of research.

Consensus algorithms

Research in PoS algorithms, or other alternatives to PoW, is also an important area of research. This is especially relevant due to the fact that the Bitcoin network's power consumption is expected to reach almost 69 TWh by the end of 2020, which is equivalent to entire nations in terms of electricity consumption. Alternatives such as PoS algorithms have already gained a lot of traction and are due to be implemented in major blockchains, such as Ethereum's **Casper**. However, so far, PoW remains the best option for securing a public blockchain. It has also been suggested that instead of performing an inefficient or single-purpose form of work, as is the case with Bitcoin's PoW, the power of the network can be used to solve useful mathematical or scientific problems.

Scalability

A detailed discussion on scalability has already been carried out in *Chapter 21, Scalability and Other Challenges*; briefly, it is sufficient to say that while some progress has already been made, there is still a need for more research in order to enable on-chain scalability, and further improve off-chain solutions such as state channels. Some initiatives, like increasing block size and using transaction-only blockchains (without blocks), have been proposed that increase the capacity of the blockchain itself to address scalability issues instead of using side channels.

Examples of implementations without blocks include the previously-mentioned **Tangle**. This utilizes a DAG to store transactions, as compared to traditional blockchain solutions where a block is used to store transactions. This makes it inherently faster as compared to block-based blockchains such as Bitcoin, where the waiting time between block generations is currently approximately 10 minutes. The performance increase comes from the fact that instead of mining a block, in a DAG-based system, transactions are linked to one another and are used for confirming and verifying the next transactions, instead of mining and verifying blocks.

Code obfuscation

Code obfuscation, using **indistinguishability obfuscation**, can be used as a means to provide confidentiality and privacy in the blockchain. However, this is still not practical due to prohibitive computational complexity, and some major research efforts are required to achieve this. Now, let's return to the subject of blockchain technology's convergence with the IoT and AI for a more in-depth analysis.

Blockchain and AI

It is envisaged that other technologies, such as IoT and AI, will converge for the mutual benefit and wider adoption of both blockchain and the other

given technology.

The convergence of blockchain with IoT has been discussed at length in *Chapter 21, Scalability and Other Challenges*. Briefly, it can be said that due to blockchain's authenticity, integrity, privacy, and shared nature, IoT networks would benefit greatly from making use of blockchain technology. This can be realized in the form of an IoT network that runs on a blockchain, and makes use of a decentralized **mesh network** for communication in order to facilitate **Machine-to-Machine (M2M)** communication in real time.



A mesh network is a network topology that allows all nodes on a network to connect with one another in a cooperative and dynamic fashion, to facilitate the efficient routing of data.

All of the data that is generated as a result of M2M communication can be used in **machine learning** processes to augment the functionality of artificially intelligent DAOs or simple AAs. These AAs can act as agents in a blockchain-provided **Distributed Artificial Intelligence (DAI)** environment, which can learn over time using machine learning processes. This would enable them to make better decisions for the good of the blockchain.

AI is a field of computer science that endeavors to build intelligent agents that can make rational decisions based on the scenarios and environment that they observe around them. Machine learning plays a vital role in AI technology, by making use of raw data as a learning resource. A key requirement in AI-based systems is the availability of authentic data that can be used for machine learning and model building. Therefore, the explosion of data coming out of IoT devices, smartphones, and other means of data acquisition means that AI and machine learning is becoming more and more powerful. There is, however, a requirement for data authenticity, which is where the convergence with blockchain comes in. Once consumers, producers, and other entities are on a blockchain, the data that is generated as a result of interaction between these entities can be readily

used as an input to machine learning engines with a guarantee of authenticity.

It could also be argued that if an IoT device is hacked, it could send malformed data to the blockchain. This issue would be mitigated using blockchain technology, because an IoT device would be part of the blockchain (as a node) and would have the same security properties applied to it as a standard node in the blockchain network. These properties include the incentivization of good behavior, rejection of malformed transactions, strict verification of transactions, and various other checks that are part of blockchain protocol. Therefore, even if an IoT device is hacked, it would be treated as a **Byzantine node** by the blockchain network and would not cause any adverse impact on the network.

The possibility of combining intelligent oracles, intelligent smart contracts, and AAs will give rise to **Artificially Intelligent Decentralized Autonomous Organizations (AIDAOs)** that can act on behalf of humans to run entire organizations on their own. This is another side of AI that could potentially become normal in the future. However, more research is required to realize this vision.

The convergence of blockchain technology with various other fields, such as 3D printing, virtual reality, augmented reality, spatial computing, and the gaming industry, is also envisaged. For example, in a multiplayer online game, blockchain's decentralized approach allows more transparency, and can ensure that no central authority is gaining an unfair advantage by manipulating game rules. Each of these topics are currently active areas of research, and more interest and development is expected.

Now, let's discuss the future of blockchain technology, and make some predictions about its development.

The future of blockchain

The year 2019 saw the move from the theoretical **Proof of Concept (PoC)** stages to the production of some real-life blockchain environments. 2019 was also considered the year of enterprise blockchain, and it is expected that before the end of 2020, we will see more organizations implementing full-scale enterprise blockchain-based projects. The start of 2021 will likely see a significant rise in **tokenization** and ecosystems built around tokens. A prime example of mainstream tokenization use is **DeFi**, which currently has billions of dollars' worth of value locked in its ecosystem. In the years to come, this technology is only expected to grow.

In this section, to round off the chapter and give an idea of the scope of blockchain development, there are a few careful predictions, based on current advancements and the speed of progress. All of these predictions are likely to be realized between the years of 2020 and 2050:

- The IoT will run on multiple blockchains and will give rise to an M2M economy. This will include energy devices, autonomous cars, and household accessories.
- Central bank-issued digital currencies will become a reality, and will be applied in day-to-day use within the next two to five years.
- Within the next three to five years, DeFi will be regulated, having become part of regular financial activities, and will have hundreds of billions of dollars' worth of value locked within its ecosystem.
- Medical records will be shared securely, while preserving patient privacy, between various private blockchains run by a consortium of health providers. It may well be a single private blockchain shared among all service providers, including pharmacies, hospitals, and clinics.
- Elections and voting systems will be managed transparently and securely via decentralized web applications, making use of a backend of blockchain technology.
- Financial institutions will run a multitude of private blockchains to share data between participants and to manage internal processes.
- Financial institutions will make use of semi-private blockchains that will provide identity information for controls and functions such as

AML and Know Your Customer (KYC). This will be shared between other financial institutions around the world.

- Immigration, passport control, and border control-related activities will be recorded and conducted on a blockchain, shared between all ports of entry and border agencies around the world.
- Governments will run interdepartmental blockchains to provide core services such as pension and benefits disbursement, and to monitor land ownership records, birth registrations, and other database services. This way, auditability, trust, and a sense of security will develop between citizens and their governments.
- Publicly available, regulated blockchains run by governments will be used on a day-to-day basis by citizens in order to manage their digital identities, and perform their day-to-day activities; for example, tax payments, TV license registrations, and marriage registrations.
- Research in cryptography and distributed systems will reach new heights, and universities and other educational establishments will offer dedicated courses on cryptography, cryptoeconomics, and blockchain technology.
- Artificially intelligent DAOs will become part of blockchains, which will make rational decisions on behalf of humans for the good of the system that they are part of.
- BaaS will be provided as standard to anyone who wishes to run their business or day-to-day transactions on a blockchain. In fact, it's entirely possible that just like the internet, blockchains will seamlessly integrate into our daily lives, and people will use them without knowing anything about the underlying technology or infrastructure.
- Blockchains will commonly be used to provide **Digital Rights Management** services for media and the arts, and will be used to deliver content directly to consumers. This will enable direct communication between the consumer and producer, eliminating the need for a central party to govern the licensing and rights management of valuable goods.
- Existing cryptocurrencies such as Bitcoin will continue to grow in value. With the availability of state channels and scalability efforts, this trend is only expected to grow. Bitcoin's value will likely reach

tens of thousands of US dollars per BTC. As a result, cryptocurrency investment will greatly increase, and a new cryptoeconomic system will emerge.

- Financial institutions and clearing houses will start to introduce blockchain-based solutions as part of the normal process for their customers.

Blockchains are going to change the world. The revolution has already begun, and it is only expected to continue.

Summary

First in this chapter, a few trends were discussed that are expected to continue as the technology progresses. We also explored some areas outside technology that will have to be addressed before the blockchain is accepted into society, such as its use in illegal activity. Some research topics were suggested that are being pursued by researchers and organizations around the world, and should be considered for personal research by any student or academic. Furthermore, convergence with other fields such as the IoT and AI were also explored. Finally, some predictions regarding the growth and adoption of blockchain technology were made. Most of these predictions are likely to come true within the next decade or so, while some may take longer.

Blockchain technology has the potential to change our lives, and the impact is already noticeable. This growth is evident in the form of successful PoC implementations and production deployments. Moreover, the number of enthusiasts and developers taking an interest in this technology is increasing, a reflection of the fact that blockchains are becoming intertwined with our lives just as the internet has been for decades. This chapter was only a modest overview of the vast and tremendous potential of blockchains, and further adoption and applications of this technology are inevitable as it matures.

In this book, we have explored the technical foundations of blockchain, and learned how to build practical real-world decentralized applications. We've explored cryptocurrencies, alternative blockchains, enterprise blockchains, scalability, and privacy, and examined the underlying mechanics of blockchain technology. We've combined these theoretical foundations with practical development, design, and deployment of blockchain networks, smart contracts, and decentralized applications. We learned how to use Truffle, Ganache, Drizzle, and other tools and techniques to build DApps. Moving forward, detailed accounts of tokenization, consensus mechanisms, and recent developments such as Ethereum 2.0 were provided.

Blockchain can arguably be considered the most innovative technology of this decade, and as we've seen throughout this book, it is one of the most active areas of study by researchers, academics... and now, you! You are now capable of applying the knowledge from this book, and continuing your learning to try your hand at entry-level blockchain development or architecture. With the material provided in this book, you are equipped with the necessary skills and knowledge of blockchain technology to participate in further research and development concerning this amazing technology, and be part of the blockchain revolution!

Index

Symbols

Practical Byzantine Fault Tolerance (PBFT)

versus Istanbul Fault Tolerance (IBFT) [169](#)

A

accountability [70](#)

account state, fields

balance [339](#)

code hash [339](#)

nonce [339](#)

storage root [339](#)

accounts type, Ethereum ecosystem

Contract Accounts (CAs) [327](#)

Externally Owned Accounts (EOAs) [327](#)

Active Directory (AD) [670](#)

address [457](#)

AddRoundKey

Advanced Encryption Standard (AES) [88](#)

using, for decryption operations
using, for encryption operations
working with

Advanced Message Queuing Protocol (AMQP) [690](#)

aggregate BLS signatures

reference link [138](#)

aggregates signatures [138](#)

aggregation based oracles [306](#)

altcoins [267, 268](#)

providing, methods [268](#)

altcoins, development [288](#)

block rewards [290](#)

block size [290](#)

coinage [290](#)

consensus algorithms [288](#)

difficulty adjustment algorithms [290](#)

hashing algorithms [289](#)

inter-block time [290](#)

interest rate [290](#)

reward halving rate [290](#)

total supply of coins [291](#)

transaction size [290](#)

Amazon Web Services (AWS) [301](#)

American National Standards Institute (ANSI) [707](#)

Analog to Digital Converter (ADC) [620](#)

Android proof [302](#)

Android proofs [300](#)

anonymity [724](#)

anonymity, Bitcoin transactions

 inherent anonymity [282](#)

 mixing protocols [281](#)

 third-party mixing protocols [282](#)

anonymity phase [285](#)

anonymous signatures [733](#)

Anti Money Laundering (AML) [651](#), [764](#)

Anti-Money Laundering (AML)

AntPool

URL

Apache Camel [670](#), [672](#)

append-only [18](#)

app.js JavaScript file

 creating

Application Binary Interface (ABI) [465](#), [638](#)

 generating [435](#), [436](#)

application firewall [693](#)

bridge [694](#)

float [694](#)

application layer [621](#)

applications on blockchain, Fabric [548](#)

application model [550](#)

chaincode implementation [549](#), [550](#)

application-specific blockchains (ASBCs) [750](#)

Application Specific Integrated Circuit (ASIC)

Application-specific Integrated Circuits (ASICs) [755](#)

Application Specific Integrated Circuits (ASICs)

architecture development method (ADM) [675](#), [676](#)

architecture change management [678](#)

architecture vision [677](#)

business architecture [677](#)

implementation governance [678](#)

information systems architecture [677](#)

migration planning [678](#)

opportunities and solutions [677](#)

preliminary phase [676](#)

technology architecture [677](#)

Arguments of Knowledge (ARK) [132](#)

Aries [526](#), [529](#)

armory_utxsvr [287](#)

Artificial Intelligence (AI) [758](#)

Artificially Intelligent Decentralized Autonomous Organizations (AIDAOs) [767](#)

ASICs

asymmetric cryptography [91](#), [94](#), [95](#), [96](#)

discrete logarithm scheme [96](#)

elliptic curves algorithm [96](#), [97](#)

integer factorization schemes [96](#)

private key [97](#)

public key [97](#)

RSA [97](#), [99](#)

asymmetric key cryptography [91](#)

asynchronous BFT (ABFT) [760](#)

asynchronous stream ciphers [88](#)

atomic broadcast [146](#)

atomic swaps [283](#)

attribute-based encryption (ABE) [733](#)

Augur

Australian Securities Exchange (ASX) [750](#)

about [3](#)

reference link [3](#)

URL [750](#)

authentication [68](#)

Autonity [681](#)

URL [681](#)

autonomous agent (AA) [54](#)

Autonomous Agents (AAs) [758](#)

Availability (A)

avalanche effect [75](#)

Avalon [526, 531](#)

Aztec protocol

URL [755](#)

B

BaaS providers

URLs [680](#)

Bacs Payment Schemes Limited (Bacs) [651](#)

Basecoin

URL [574](#)

Basic Attention Token (BAT)

beacon chain

features [515](#)

beacon node [515](#), [517](#)

versus validator node [521](#)

benchmarking [385](#), [386](#)

Besu [526](#), [528](#)

BFT algorithms

about [159](#)

bid price [584](#)

binary signing [765](#)

Bitcoin [186](#), [188](#), [713](#)

advanced protocols [246](#)

buying [250](#)

client, installation [251](#), [253](#)

client, installation link [251](#)

clients and tools, types [253](#)

coinbase transaction [217](#)

cryptographic keys [196](#)

egalitarianism, versus authoritarianism [187](#), [188](#)

future, reference link [184](#)

history [10](#)

innovation in [245](#)

mining systems

overview [183](#), [184](#), [185](#)

reference link [186](#)

Script language, using [211](#)

selling [250](#)

transaction, verifying [218](#)

used, for solving issues [186](#)

user's perspective [189](#)

Bitcoin addresses

creating [202](#)

encoding, with Base58Check encoding [203](#), [204](#)

multi-signature addresses [205](#)

vanity addresses [204](#), [205](#)

Bitcoin APIs

references

Bitcoin blockchain [219](#), [220](#), [221](#)

genesis block [221](#)

mining [225](#)

network difficulty [224](#)

orphan block [222](#), [223](#)

size [224](#)

stale block [222](#), [223](#)

Bitcoin blockchain, mining

hash rate

miners, tasks [225](#), [227](#)

mining algorithm [227](#)

Proof of Work (PoW) [227](#)

rewards [227](#)

Bitcoin Cash (BCH) [248](#)

URL [249](#)

bitcoin-cli [253](#)

experimenting with [262](#), [263](#), [264](#)

Bitcoin, clients and tools

bitcoin-cli [253](#)

bitcoind [253](#)

bitcoin-qt [253](#), [254](#)

Bitcoin command-line interface (bitcoin-cli) [264](#)

Bitcoin command-line tool

using [264](#), [265](#)

bitcoin.conf

reference link [256](#)

setting up [256](#)

bitcoind [253](#)

Bitcoin ecosystem

transaction data structure [207](#), [208](#)

transaction fees [207](#)

transaction lifecycle [206](#)

transaction pool [207](#)

transactions [206](#)

Bitcoin ecosystem, transaction data structure

inputs [209](#), [210](#)

metadata [209](#)

outputs [210](#)

verification [210](#)

Bitcoin Gold [249](#)

URL [250](#)

Bitcoin improvement proposal

reference link [284](#)

Bitcoin Improvement Proposals (BIPs) [245](#), [716](#), [752](#)

informational BIP [245](#)

process BIP [245](#)

standard BIP [245](#)

types [245](#)

Bitcoin investment [250](#)

Bitcoinj

reference link

Bitcoin Lightning [721](#), [752](#)

Bitcoin limitations [281](#)

anonymity [281](#)

privacy [281](#)

Bitcoin merchant solutions

reference link [244](#)

Bitcoin, mining systems

ASICs

CPU mining

Field Programmable Gate Arrays (FPGAs)

GPU mining

mining pool

Bitcoin network [229](#), [231](#), [232](#), [233](#), [234](#), [235](#), [236](#), [238](#)

bloom filter [238](#), [239](#)

full client [238](#)

SPV client [238](#)

Bitcoin-NG [718](#)

Bitcoin node

setting up [254](#), [255](#)

setting up, in regtest [259](#), [260](#), [262](#)

setting up, in testnet [257](#), [259](#)

Bitcoin payments [243](#), [244](#)

Bitcoin programming

libraries

Bitcoin proposals

reference link [716](#)

Bitcoin protocols

Segregated Witness [246](#), [247](#), [248](#)

bitcoin-qt [253](#), [254](#)

Bitcoin, Script language

contracts [216](#)

miniscript [217](#)

opcodes, using [211](#), [212](#)

types [212](#), [213](#), [215](#), [216](#)

Bitcoins, cryptographic keys

private keys [196](#), [200](#)

public keys [200](#)

Bitcoin source code

setting up [256](#)

Bitcoin testnet

reference link [236](#)

Bitcoin Unlimited [249](#)

URL [249](#)

Bitcoin, user's perspective

payment, sending to someone [189](#), [190](#), [191](#), [192](#), [194](#), [195](#), [196](#)

Bitcoin vulnerabilities

reference link [219](#)

transaction bugs [218](#)
transaction malleability [218](#)
value overflow [218, 219](#)

BitLicense [187](#)

BitPay

URL [244](#)

Blake2 compression function

blind signatures [15, 135](#)

reference link [136](#)

blockchain [341](#)

about [1, 16](#)
advantages [29, 30, 31](#)
and AI [767](#)
append-only [18](#)
applications [758](#)
architecture [19](#)
as business [20](#)
by layers [19, 20](#)
consensus
convergence, with other technologies [758](#)
cryptographically secure [18](#)
disadvantages [29, 30, 31](#)
distributed ledger [18](#)
distributed systems [8, 9, 10](#)

education and employment [756](#)
emerging trends [750](#)
events, history [11](#), [13](#)
features [29](#), [30](#), [31](#)
formal methods and security [756](#)
future
generic elements [21](#), [22](#), [24](#), [26](#)
hardware development [755](#), [756](#)
history [10](#)
implementations [750](#)
implementing, in cloud [678](#), [680](#)
issues [31](#)
new programming languages [756](#)
ongoing research and study [754](#)
peer-to-peer [16](#)
public, versus private [760](#), [761](#)
querying, Geth used [356](#)
start-ups [751](#)
tokenization [569](#)
types [32](#)
updatable, via consensus [18](#)
using, for decentralization [38](#), [39](#), [40](#), [42](#)
working [28](#), [29](#)

blockchain applications

in health industry [647](#), [648](#)

in media

blockchain applications, finance

about [648](#)

cross-border payments

financial crime prevention [651](#)

insurance [648](#), [650](#)

payments [651](#)

payments, better resilience

payments, decentralization

payments, faster settlement

peer-to-peer loans

post-trade settlement [650](#)

blockchain applications, government

about [643](#), [644](#)

border control systems [644](#), [645](#)

citizen identification (ID cards) [646](#), [647](#)

voting [646](#)

blockchain, areas to address [763](#)

illegal activity [764](#), [765](#)

privacy [765](#)

regulation [763](#)

transparency [765](#)

Blockchain as a Service (BaaS) [678](#), [758](#)

blockchain-based IoT implementation

about [624](#), [625](#)

electronic circuit, building [636](#), [637](#), [638](#), [639](#), [640](#), [642](#), [643](#)

first node, setting up [629](#), [630](#)

Raspberry Pi node, setting up [630](#), [632](#)

Raspberry Pi, setting up [625](#), [626](#), [628](#), [629](#)

blockchain convergence

benefits [622](#), [623](#), [624](#)

blockchain, elements

address [21](#)

blocks [21](#)

Merkle root [22](#)

node [26](#)

peer-to-peer network [24](#)

programming language [24](#)

smart contracts [24](#)

state machine [24](#)

timestamp [21](#)

transactions [21](#)

virtual machine [24](#)

blockchain.info

URL

blockchain, layers

application layer [20](#)

consensus layer [20](#)

cryptography layer [20](#)

execution layer [20](#)

Blockchain Open Ledger Operating System (BOLOS) [302](#)

Blockchain oracle problem [304](#)

blockchain oracles [304](#)

aggregation based oracles [306](#)

computation oracles [306](#)

crowd wisdom driven oracles [306](#)

decentralized oracles [308](#)

hardware oracles [306](#)

inbound oracles [304](#)

smart oracles [308](#)

software oracles [304](#)

Blockchain oracle services [309](#)

blockchain planes [712](#)

consensus plane [713](#)

network plane [712](#)

side plane [713](#)

storage plane [713](#)

view plane [713](#)

blockchain research topics [765](#)

code obfuscation [766](#)

consensus algorithms [766](#)

cryptographic function limitations [766](#)

scalability [766](#)

smart contracts [765](#)

blockchains

alternatives [760](#)

Blockchain services, Fabric [542](#)

consensus services [542](#)

distributed ledger [543](#)

ledger storage [544](#)

peer-to-peer protocol [543](#)

blockchain technology

growth [1, 2](#)

increasing, interest [4, 6](#)

progressing [2, 3, 4](#)

blockchain, technology improvements [751](#)

Consortia [752](#)

enhancements [752](#)

standardization [751](#)

blockchain, types

distributed ledgers [33](#)

Distributed Ledger Technology (DLT) [33](#)

permissioned ledger [35](#)

private blockchains [34, 35](#)

- proprietary blockchains [35](#)
- public blockchains [34](#)
- semi-private blockchains [34](#)
- shared ledger [35](#)
- sidechains [34](#)
- tokenized blockchains [35](#)
- token-less blockchains [36](#)

block cipher

- block encryption mode [89](#)
- cipher block chaining (CBC) [91](#)
- counter (CTR) mode
- cryptographic hash mode
- electronic codebook (ECB) [91](#)
- keystream generation mode
- message authentication mode

block cipher encryption

block ciphers [88, 89](#)

block data [543](#)

block difficulty mechanism [347, 348](#)

block encryption mode [89](#)

block finalization [346](#)

- Ommers validation [346](#)
- reward application [346](#)

state and nonce validation [346](#)

transaction validation [346](#)

block header [543](#)

block headers [343](#)

beneficiary field [343](#)

difficulty [343](#)

extra data [344](#)

gas limit [343](#)

gas used [343](#)

logs bloom [343](#)

mixhash [344](#)

nonce [344](#)

number [343](#)

Ommers hash [343](#)

parent hash [343](#)

receipts root [343](#)

state root [343](#)

timestamp [343](#)

transaction root [343](#)

block.io

URL

block metadata [543](#)

block reward [290](#)

blocks [341](#), [527](#)

block validation mechanism [345](#)

bloom filter [238](#)

types [239](#)

bloXroute [715](#)

reference link [715](#)

BLS cryptography

Blum-Blum-Shub (BBS) [72](#)

b-money

reference link [185](#)

Boneh-Lynn-Shacham signatures

Boolean [456](#)

brain wallets [240](#)

breadboard [636](#)

Brownie

reference link [447](#)

BTC

URL

BTC Relay

reference link [287](#)

BTC TOP

URL

Bulletproofs [728](#)

Burrow [526](#), [529](#)

Byzantine fault-tolerance (BFT) [144](#)

Byzantine Fault Tolerance (BFT) [659](#)

Byzantine fault-tolerant (BFT) [529](#)

Byzantine generals problem

about [143](#), [144](#)

Proof-of-Work (PoW), using as solution [184](#)

reference link [144](#)

Byzantium [348](#), [380](#)

C

Cakeshop

private transaction, viewing

Caliper [526](#), [533](#)

Capital Gains Tax (CGT) [763](#)

CAP theorem [10](#)

about

cardinality [94](#)

Casascius physical Bitcoins

reference link [198](#)

Casper [347](#), [376](#), [515](#), [766](#)

reference link [347](#), [376](#)

Casper the friendly finality gadget (Casper FFG) [523](#)

Cello [526](#), [533](#)

central bank digital currency (CBDC) [761](#)

centralized finance (CeFi) [588](#)

token regulation [589](#)

tokens, trading [589](#)

centralized systems [39](#)

certificate authority (CA) [119](#), [542](#)

CEX

URL [250](#)

CFT algorithms [152](#)

chain-based PoS

chaincode [537](#), [541](#)

chained hashing scheme [272](#)

chain growth

Chainlink

URL [309](#)

chain of blocks [16](#)

chain quality

Chain Virtual Machine (CVM) [24](#)

challenge-response protocols [95](#)

checkpointing [160](#)

Chicago Mercantile Exchange (CME) [183](#)

cipher block chaining (CBC) [91](#)

cipher block chaining mode (CBC mode)

Cipher Feedback (CFB)

Clearing House Automated Payment System (CHAPS) [651](#)

CNexchange (CNEX) [580](#)

code signing [765](#)

Coinbase

finding

CoinJoin [282, 732](#)

Coinprism

reference link [287](#)

CoinSwap [283](#)

reference link [283](#)

collision resistance [75](#)

colored coins [286](#)

commit chains , [723](#)

commit sub-protocol algorithm [164](#)

committee-based PoS

committers [546](#)

CommonAccord

URL [296](#)

common language for augmented contract knowledge (CLACK) [297](#)

common prefix

compliance challenges

data sovereignty [664](#)

liability [664](#)

regulatory compliance [663](#)

components, Fabric

channels [546](#)

clients [546](#)

crypto service provider [548](#)

Membership Service Provider (MSP) [548](#)

peers [546](#)

smart contracts [548](#)

transactions [546](#)

World state database [546](#)

components, Sawtooth

client [564](#)

REST API [564](#)

state [565](#)

transaction families [565](#), [566](#), [567](#)

transaction processors [565](#)

validator [564](#)

computational model

used, for describing and analysing consensus algorithms [148](#)

computation oracles [306](#)

Computation tree logic (CTL) [737](#)

confidentiality [68](#), [724](#)

confidential transactions [732](#)

confusion [89](#)

consensus

about [37](#)

categories

in blockchain

states [171](#), [173](#)

consensus algorithm [37](#)

analysing [148](#)

designing [148](#)

finality

performance

scalability

selecting

speed

consensus algorithms

asynchronous systems [148](#)

describing and analysing, with computational model [148](#)

fundamental, requisites [150](#)

partial synchrony [150](#)

processes communicate [148](#)

synchronous system [148](#)

timing assumptions [148](#)

used, for classifying categories [150](#), [152](#)

consensus algorithms, fundamental requisites

liveness [152](#)

safety [150](#)

consensus, Fabric

ordering [552](#)

transaction endorsement [552](#)

validation and commitment [552](#)

consensus mechanism

about [37](#)

requisites [37](#)

types

verifying [738](#)

consensus mechanism, Ethereum [378](#)

consensus mechanisms

Clique [709](#)

IBFT [709](#)

RAFT [709](#)

consensus model, Corda [689](#)

uniqueness consensus [689](#)

validity consensus [689](#)

consensus protocol [26](#)

consensus, Sawtooth [560](#)

PBFT [562](#)

PoET [561](#)

RAFT [562](#)

consensus service [542](#)

consistency [659](#)

Consistency (C)

Constellation [700](#)

Contract Accounts (CAs) [327](#)

contract creation transaction [333](#), [335](#)

parameters [333](#)

co-prime [98](#)

Corda [681, 686, 760](#)

architecture [686](#)

components [690](#)

consensus model [689](#)

CorDapps [689](#)

development environment [695, 696](#)

Flows [689](#)

state objects [687, 688](#)

transactions [688](#)

Corda components

network map service [692](#)

nodes [690, 691](#)

notary service [692](#)

oracle service [692](#)

permissioning service (Doorman) [691](#)

vaults [693](#)

Corda Distributed Application (CorDapps) [690](#)

Corda enterprise [686](#)

Corda network [687](#)

transaction elements [693](#)

transactions [692](#)

Corda node

CorDapp interface [691](#)

Network interface [691](#)

Persistence layer [691](#)

RPC interface [691](#)

Service hub [691](#)

Corda open source [686](#)

Corda smart contract

commands [690](#)

executable code [690](#)

state objects [690](#)

Cords

transaction flow [694](#), [695](#)

Cosmos

Counter block [287](#)

counter (CTR) mode

Counterparty [287](#)

reference link [288](#)

Counterparty coin (XCP) [287](#)

Counterparty server [287](#)

Counter wallet [287](#)

CPU mining ,[**384**](#)

Crash fault-tolerance (CFT) [**144**](#)

Crash Fault-tolerant (CFT) [**557**](#)

crosslink

crowd wisdom driven oracles [**306**](#)

cryptocurrencies [**271**](#)

cryptocurrency [**291**](#)

versus token [**291**](#)

cryptoeconomics [**755**](#)

cryptographically secure [**18**](#)

cryptographic, constructs and blockchain [**124**](#)

commitment schemes [**126**](#), [**127**](#)

homomorphic encryption [**124**](#), [**125**](#)

secret sharing [**125**](#)

signcryption [**125**](#)

cryptographic hash functions

applications [**140**](#)

cryptographic hash functions, applications

distributed hash table (DHT)

merkle trees

cryptographic hash mode

cryptographic key [84](#)

cryptographic primitives [70](#), [71](#)

symmetric cryptography [84](#), [85](#)

cryptography [66](#), [68](#), [755](#)

accountability [70](#)

authentication [68](#)

concepts [68](#)

confidentiality [68](#)

integrity [68](#)

non-repudiation [69](#), [70](#)

cryptography, authentication

data origin authentication [69](#)

entity authentication [68](#), [69](#)

cryptographic primitives

asymmetric cryptography [71](#)

keyless primitives [71](#)

symmetric cryptography [71](#)

CryptoKitties

URL [573](#)

Crypto-kyc

URL [651](#)

CureCoin

URL [648](#)

curl

reference link

current market cap

reference link [270](#)

cyclic group [94](#), [108](#)

D

Dagger-Hashimoto algorithm [380](#)

Dai stable coin

URL [574](#)

Dandelion [284](#), [285](#)

reference link [286](#)

Dandelion++ research paper

reference link [285](#)

Dandelion proposal paper

reference link [285](#)

DAO attack

DAO hack [565](#)

Dark Gravity Wave (DGW) [277](#)

reference link [279](#)

dark web [764](#)

reference link [764](#)

datacopy function

Data Encryption Standard (DES) [88](#)

data integrity service [74](#)

data origin authentication [69](#)

decentralization

framework example [48](#)

methods [42](#)

platforms [61](#)

routes to [46](#)

using [46](#)

with blockchain [38](#), [39](#), [40](#), [42](#)

decentralization, blockchain and full ecosystem [48](#)

communication [50](#), [52](#)

power, computing [52](#), [53](#), [54](#)

storage [50](#)

decentralization, methods

contest driven [43](#), [44](#), [46](#)

disintermediation [43](#)

decentralization, platforms

EOS [63](#)

Ethereum [61](#)

Lisk [61](#)

MaidSafe [61](#)

decentralized application

developing, with Truffle framework

decentralized application (DApp) [298](#)

decentralized applications (DApps) [570](#)

decentralized applications (DAPPs) [56](#)

decentralized applications (DAPPs) [56](#)

design [58](#), [59](#)

operations [58](#)

requisite [58](#)

types [56](#), [57](#)

Decentralized Applications (DApps) [311](#)

decentralized applications (DAPPs), examples [60](#)

KYC-Chain [60](#)

Lazooz [60](#)

OpenBazaar [60](#)

decentralized autonomous corporations (DACs) [56](#)

Decentralized Autonomous Initial Coin Offering (DAICO) [577](#)

decentralized autonomous organization (DAO) [55](#)

Decentralized Autonomous Organization (DAO) , [341](#)

decentralized autonomous organizations (DAOs) [577](#)

decentralized consensus [40](#)

decentralized exchanges (DEX) [589](#)

decentralized exchanges (DEXs)

decentralized finance (DeFi) , [299](#), [583](#), [588](#)

advantages

disadvantages

Decentralized Finance (DeFi)

about [2](#)

URL [2](#)

decentralized identity

decentralized oracles [308](#)

decentralized organizations (DOs) [54](#)

Decentralized Organizations (DOs) [54](#)

decentralized storage

deploying, with IPFS

decentralized system [40](#)

decentralized web [63](#)

stages

decryption [114](#)

using, RSA [113](#)

DeFi

delegated PoS

Delegated Proof of Stake (DPoS)

Delegated Proof of Stake (DPOS) [61](#)

demand-side economies of scale [188](#)

demilitarized zone (DMZ) [693](#)

denial of service attacks [624](#)

Denial of Services (DoS) attacks [565](#)

deployment transactions [546](#)

deposit contracts [521](#)

derivative token [575](#)

deterministic wallets [240](#)

development frameworks

device layer [620](#)

difficulty retargeting algorithms [276](#)

difficulty time bomb [347](#)

diffusion [89](#), [285](#)

DigiShield [279](#)

reference link [279](#)

Digital Asset Holdings (DAH) [527](#)

Digital Rights Management (DRM) [31](#)

digital signature

generating, with OpenSSL [121](#), [122](#)

digital signature algorithm (DSA) [120](#)

Digital Signature Algorithm (DSA) [94](#)

digital signatures [95](#), [117](#), [118](#)

types [134](#)

digital signatures, types

aggregate signatures [138](#)

blind signatures [134](#)

multi-signatures [136](#), [137](#)

ring signatures [139](#)

threshold signatures [137](#), [138](#)

Digital Trade Chain (DTC) [750](#)

URL [750](#)

Digital Yuan [761](#)

Digix gold tokens

URL [574](#)

Directed Acyclic Graph (DAG) [50](#), [381](#), [718](#), [760](#)

disaster recovery (DR)

discrete logarithm scheme [96](#)

disintermediation [43](#)

DiscV4 [390](#)

DiscV5 [390](#)

Distributed Artificial Intelligence (DAI) [767](#)

distributed consensus [37](#)

distributed consensus problem

 overview [141](#), [143](#)

distributed consensus problems

 lower bounds, on number of processors to solve [147](#), [148](#)

Distributed Denial of Service (DDOS)

distributed hash table (DHT)

distributed hash tables (DHTs) [50](#)

Distributed Hash Tables (DHTs) [11](#)

distributed ledger [18](#)

distributed ledgers [33](#)

distributed ledgers, Hyperledger [526](#)

 Besu [528](#)

 Burrow [529](#)

 Fabric [527](#)

 Indy [528](#)

Iroha [528](#)

Sawtooth [527](#)

distributed ledger technology (DLT) [6](#)

Distributed Ledger Technology (DLT) [33](#)

distributed system [39](#)

reference link [145](#)

distributed systems

about [8](#), [9](#), [10](#)

distributive law [92](#)

DLS algorithm

reference link [174](#)

domain-specific, Hyperledger [535](#)

Grid [535](#)

Labs [535](#)

domain-specific languages (DSLs) [297](#)

double-spending [735](#)

DragonMint T1 miner

reference link

Drizzle [447](#)

used, for building User Interface (UI)

Dynamis

URL [650](#)

E

easy parallelizability

reference link [715](#)

eclipse attacks [735](#)

EIP 55

reference link [322](#)

Ekiden [730](#)

electronic cash (e-cash)

about [13](#), [15](#)

accountability [13](#)

anonymity [13](#)

fundamental [13](#)

electronic codebook (ECB) [91](#)

Electronic Code Book (ECB) [89](#)

Electronic Frontier Foundation (EFF)

Electronic Government as a Service (eGaaS)

URL [758](#)

Electrum

URL [240](#)

elliptic curve addition function

elliptic curve cryptography (ECC) [100](#)

discrete logarithm problem [108](#), [109](#), [110](#)

mathematics [100](#)

private key pair [114](#), [115](#)

private key pair, generating [115](#), [116](#), [117](#)

public key pair [114](#)

using OpenSSL [114](#)

Elliptic Curve Cryptography (ECC) [92](#), [196](#), [329](#)

elliptic curve cryptography (ECC), mathematics

point addition [100](#), [102](#), [103](#), [104](#)

point doubling [105](#), [106](#)

point multiplication [107](#), [108](#)

Elliptic-curve Diffie-Hellman (ECDH) [700](#)

Elliptic Curve Diffie-Hellman (ECDH) [96](#), [100](#)

Elliptic Curve Digital Signature Algorithm (ECDSA) [100](#), [120](#), [121](#), [200](#), [324](#)

private key, generating with OpenSSL [122](#), [123](#), [124](#)

Elliptic Curve Digital Signatures Algorithm (ECDSA) [96](#)

Elliptic Curve Integrated Encryption Scheme (ECIES) [391](#)

elliptic curve pairing

elliptic curve public key recovery function

elliptic curves algorithm [96](#), [97](#)

elliptic curve scalar multiplication

Embark [447](#)

URL [447](#)

embedded consensus [287](#)

enclave [700](#), [730](#)

encoding schemes [139](#)

base58 [140](#)

base58, reference link [140](#)

base64 [139](#)

encryption [113](#)

using, RSA [113](#)

encrypt then sign [119](#), [120](#)

endorsers [546](#)

enrolment certificate authority (E-CA) [542](#)

enrolment certificates (E-Certs) [542](#)

enterprise blockchain [653](#), [655](#), [680](#)

architecture [668](#)

Autonity [681](#)

challenges [684](#)

Corda [681](#)

Fabric [681](#)

limiting factors [656](#)

platforms, comparing [681](#), [684](#)

Quorum [681](#)

requisites [659](#)

success factors [655](#)

use cases [668](#)

versus public blockchain [666](#)

enterprise blockchain architecture [668](#)

application layer [672](#)

governance layer [670](#)

integration layer [670](#)

monitoring layer [672](#)

network layer [669](#)

performance layer [672](#)

privacy layer [670](#)

protocol layer [669](#)

scalability layer [672](#)

security layer [672](#)

enterprise blockchain challenges [684](#)

business challenges [685](#)

compliance [685](#)

interoperability [684](#)

lack of standardization [684](#)

enterprise blockchain, requisites

access governance [663](#)

compliance [663](#)

ease of use [665](#)

integration [665](#)

interoperable [664](#)

monitoring [665](#)

performance [661](#)

privacy [659](#)

secure off-chain computation [665](#)

supporting tools [666](#)

enterprise blockchains [750](#)

enterprise blockchain solutions

ADM [675](#)

designing [674](#)

TOGAF [674](#)

Enterprise Blockonomics [685](#)

enterprise DApps [663](#)

Enterprise Ethereum Alliance (EEA) [685](#)

about [5](#)

URL [685](#)

Enterprise Resource Planning (ERP) [656](#)

enterprise solutions

and blockchain [653](#), [654](#)

entity authentication [68](#)

entropy [72](#)

EOS [63](#)

epidemic flooding mechanism [285](#)

equity token offerings (ETOs) [577](#)

ERC-20-compliant tokens

mandatory functions [591](#), [592](#)

optional functions [592](#)

ERC20 standard

ERC-20 token [579](#)

adding, in MetaMask [611](#), [612](#), [613](#), [614](#), [616](#)

building [589](#)

building, pre-requisites [591](#)

ERC-1404

URL [582](#)

estimate gas [435](#)

Eth [351](#)

installing [362](#)

reference link [351](#)

ETH 2 [524](#)

Eth capability protocol [391](#)

Ethash [380](#), [382](#)

Ether currency (ETH) [55](#)

ether (ETH) [341](#)

Ethereum [696](#), [728](#)

implementations of DAOs

overview [311](#), [313](#)

roadmap [511](#)

URL [311](#), [579](#)

user's perspective [316](#), [318](#), [319](#), [320](#)

Ethereum [61](#)

Ethereum 1 to Ethereum 2, transition

merging

stateless clients

Ethereum 2.0

development phases [513](#)

features [511](#)

goals [511](#)

overview [510](#)

Ethereum 2.0 clients

high-level architecture

Ethereum account management

with Geth [355](#)

Ethereum blockchain

account state [339](#)

elements [324](#)

state storage [339](#)

world state [339](#)

Ethereum Blockchain Explorer

reference link [319](#)

Ethereum Classic [341](#)

Ethereum Classic (ETC) [268](#)

Ethereum cryptocurrency [341](#), [342](#)

Ethereum ecosystem

accounts [325](#)

accounts type [325](#)

components [322](#), [324](#)

keys and addresses [324](#)

message [327](#), [332](#)

messages [330](#)

transaction [327](#), [330](#), [332](#)

Ethereum (ETH) [268](#)

Ethereum Homestead [335](#)

Ethereum Improvement Proposal (EIP)

URL [580](#)

Ethereum Improvement Proposals (EIPs) [752](#)

Ethereum keystore [357](#), [359](#), [360](#), [361](#)

Ethereum Name Service (ENS)

URL [670](#)

Ethereum network [320](#)

mainnet [320](#)

private nets [321](#), [322](#)

testnets [320](#)

Ethereum releases

reference link [314](#)

Ethereum Request for Comments

Ethereum smart contract development

compliers [433](#)

deploying [448](#), [450](#)

frameworks [444](#)

languages [432](#)

Solidity compiler [433](#)

Solidity language [451](#)

Solidity source code file, layout [450](#)

testing [449](#)

tools and libraries [437](#)

tools, reference link [432](#)

writing [448](#), [449](#)

Ethereum smart contract development, frameworks

Brownie [447](#)

Drizzle [447](#)

Embark [447](#)

Etherlime [448](#)

OpenZeppelin [448](#)

Truffle [444](#), [446](#)

Waffle [448](#)

Ethereum smart contract development, languages

Low-Level Lisp-Like Language [433](#)

Mutan language [433](#)

serpent language [433](#)

solidity language [432](#)

Vyper language [432](#)

Yul language [433](#)

Ethereum smart contract development, tools and libraries

Ganache [441](#), [442](#), [443](#), [444](#)

Ganache CLI [439](#), [441](#)

Node.js [437](#)

Ethereum tokens [341](#), [342](#)

Ethereum Virtual Machine (EVM) [20](#), [24](#), [310](#), [342](#), [346](#), [433](#), [511](#), [529](#)

execution environment

iterator function

machine state

Ethereum WebAssembly (ewasm)

reference link

Ethereum yellow paper [314](#)

blockchain [315](#)

mathematical symbols [314](#), [315](#)

Etherlime

reference link [448](#)

Ethers [61](#)

Etherscan

URL [580](#)

Ethlance

URL [56](#)

ETH Zurich distributed computing group

URL [754](#)

European Union Agency for Network and Information Security (ENISA) [763](#)

eventual consistency

Everledger

URL [750](#)

EVM, storage types

memory

stack

storage

ewasm [510](#)

exchanges [584](#)

exclusive OR (XOR) [80](#)

execution environment

parameters

Explorer [526](#), [533](#)

extendable-output functions (XOFs) [80](#)

Extended Merkle Signature Scheme (XMSS)

external functions

Externally Owned Account (EOA) [335](#)

Externally Owned Accounts (EOAs) [327](#)

F

Fabric [526](#), [527](#), [541](#), [681](#)

APIs [544](#)

Blockchain services [542](#)

CLIs [544](#)

components [544](#)

consensus mechanism [552](#)
membership services [541](#)
smart contract services [544](#)
transaction life cycle [553](#), [555](#)

Fabric 2.0 [555](#)

better performance [557](#)
enhanced data privacy [557](#)
external chaincode launcher [557](#)
new chaincode application patterns [557](#)
new chaincode lifecycle management [556](#)
Raft consensus [557](#)

fault-tolerant algorithms

CFT algorithms [152](#)
Paxos [152](#), [153](#)
types [152](#)

fault-tolerant consensus

about [144](#)
Byzantine fault-tolerance (BFT) [144](#)
Crash fault-tolerance (CFT) [144](#)
types [144](#)

Federal Information Security Management Act (FISMA) [685](#)

URL [685](#)

Federated Byzantine Agreement (FBA)

Feistel cipher [88](#), [89](#)

Field Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) [755](#)

Filament [624](#)

finality

Financial Conduct Authority (FCA) [663](#), [763](#)

financial DSLs

reference link [297](#)

financial instruments, attributes

counterparty [587](#)

economics [585](#)

general attributes [585](#)

sales [585](#)

financial markets

capital markets [583](#)

credit markets [583](#)

money markets [583](#)

Firechat

reference link [52](#)

FLP impossibility

about [146](#), [147](#)

techniques [147](#)

Fluff phase [285](#)

FoldingCoin

URL [648](#)

foreign exchange (forex) [574](#)

fork

hard forks [223](#)

soft forks [223](#)

temporary forks [223](#)

types [223](#)

forking [378](#)

forks

in blockchain [380](#)

formal specifications [735](#)

formal verification [735](#)

Frama-C

URL [740](#)

Frontier [380](#)

full client [238](#)

function

external functions

internal functions
types

fungible tokens [572](#)

principle, of working [573](#)

G

Galois Counter (GCM)

Galois fields [92](#)

Ganache , [441](#), [442](#), [443](#), [444](#)

download link [441](#)

Ganache CLI [439](#), [441](#)

gas [348](#)

fee schedule [350](#)

GDPR guidance

URL [664](#)

Gemini Dollar (GUSD)

URL [574](#)

General-Purpose I/O (GPIO) [636](#)

general-purpose programming languages (GPLs) [298](#)

genesis block [21](#), [344](#)

genesis file [393](#)

genesis transaction [294](#)

Geth [351](#)

installing [352](#), [354](#)

list of accounts

network is up, checking

reference link [351](#)

synchronization status

used, for deploying smart contract [464](#), [465](#), [467](#)

used, for querying blockchain [356](#)

Web3, exploring with [463](#), [464](#)

Geth account

creating [355](#)

Geth attach [356](#), [357](#)

Geth client version

providing

Geth console [356](#)

Geth JSON RPC API

global public state root [698](#)

Goerli test network [369](#)

Golem network

URL [57](#)

Google RPC (gRPC) [543](#)

GPU mining , [384](#), [385](#)

Greediest Heaviest Observed SubTree (GHOST) [523](#)

Greedy Heaviest Observed Subtree (GHOST) [378](#), [716](#)

GreenAddress

URL [241](#)

Grid [535](#)

group signatures [733](#)

H

hard fork [223](#)

hardware description languages (HDLs)

hardware device-assisted proofs [302](#)

Android proof [302](#)

ledger proof [302](#)

trusted hardware-assisted proofs [303](#)

hardware of security modules (HSMs) [539](#)

hardware oracles [306](#)

Hardware Security Modules (HSMs) [665](#)

hardware wallets [241](#)

hash-based MACs (HMACs) [87](#), [88](#)

hashcash [271](#)

hash collision attack [310](#)

hash function [74](#)

hash functions

OpenSSL example [82](#), [84](#)

hash rate

Hedera Hashgraph [760](#)

hierarchical deterministic wallets [240](#)

High Performance Computing (HPC) [665](#)

homomorphic encryption [724](#), [727](#), [755](#)

HotStuff

about

chain quality

linear view change

model

optimistic responsiveness

working

HTML frontend

HTTP REST interface

using [266](#)

Hyperledger

distributed ledgers [526](#)

domain-specific [535](#)
libraries [529](#)
projects [526](#)
reference architecture [536](#), [537](#)
tools [531](#)

Hyperledger design principles [539](#)

auditability [540](#)
deterministic transactions [540](#)
identity [540](#)
interoperability [540](#)
modular structure [539](#)
portability [540](#)
privacy and confidentiality [539](#)
rich data queries [541](#)
scalability [540](#)

Hyperledger, elements

APIs [538](#)
communication [537](#)
consensus [537](#)
data store [537](#)
policy services [537](#)
SDKs [538](#)
security and crypto [537](#)
smart contracts [537](#)

Hyperledger grid [526](#)

Hyperledger labs [526](#), [535](#)

Hyperledger Quilt

URL [752](#)

Hyperledger Sawtooth [559](#)

Hyperledger Sawtooth, core features [559](#)

dynamic and pluggable consensus algorithms [560](#)

enhanced event mechanism [560](#)

global state agreement [559](#)

interoperability [560](#)

modular design [559](#)

multi-language support [560](#)

on-chain governance [560](#)

parallel transaction execution [559](#)

I

I2P [726](#)

URL [726](#)

ice age [347](#)

identity function

Identity Overlay Network (ION)

iExec

reference link [303](#)

URL [309](#)

inbound oracles [304](#)

indistinguishability obfuscation [724](#), [766](#)

indistinguishable obfuscation (IO) [727](#)

Indy [526](#), [528](#)

information and communication technology (ICT) [39](#)

Infrastructure as a Service (IaaS) [678](#)

Infura

URL [363](#)

Initial Block Download (IBD) [234](#)

Initial Coin Offering (ICO) [63](#)

initial coin offerings (ICO) [576](#)

Initial Coin Offerings (ICOs) [188](#)

initial currency offering (ICO) [576](#)

initial exchange offering (IEO) [577](#)

Initial Exchange Offerings (IEOs)

initialization vector (IV) [85](#)

Initial public offering (IPOs)

initial public offerings (IPOs) [576](#)

Initiative for Cryptocurrencies & Contracts (IC3)

URL [754](#)

inner padding (ipad) [87](#)

integer factorization schemes [96](#)

integers [456](#)

integrity [68, 659](#)

Intel SGX technology

reference link [303](#)

Intel Software Guard Extensions (Intel's SGX) [527](#)

Interledger

internal functions

International Civil Aviation Organization (ICAO) [644](#)

Internet of Things (IoT) , [299](#)

about [618](#)

benefits [622, 623, 624](#)

Internet of Things (IoT) architecture

about [618, 620](#)

application layer [621](#)

device layer [620](#)

management layer [621](#)

network layer [621](#)

physical object layer [620](#)

Internet service providers (ISPs) [52](#)

Inter-Planetary File System (IPFS) [50](#)

Invertible Bloom Lookup Tables (IBLTs) [717](#)

invocation transactions [546](#)

IOTA [760](#)

IPFS

URL

used, for deploying decentralized storage

IPFS package

download link

IPv6

URL [621](#)

Iroha [526](#), [528](#)

Istanbul Fault Tolerance (IBFT)

about [169](#)

versus Practical Byzantine Fault Tolerance (PBFT) [169](#)

working with [169](#), [171](#)

Istanbul Fault Tolerant (IBFT) [141](#)

Istanbul upgrade [348](#)

iterator function

J

JavaScript frontend

JavaScript Object Notation (JSON)

JavaScript runtime environment (JSRE)

Java Virtual Machine (JVM) [686](#)

Jaxx

URL [240](#), [352](#)

Jaxx wallet

reference link [316](#)

JSON RPC interface

using [265](#), [266](#)

Just a Bunch of Key wallets [240](#)

K

Kadcast [714](#)

Kafka [542](#)

key derivation function (KDF) [85](#)

keyed hash functions [87](#)

key establishment mechanisms [95](#)

keyless primitives [71](#)

hash functions [74](#)

random numbers [71](#)

keyless primitives, hash functions

arbitrary messages, compressing into fixed-length digests [74](#)

collision resistance [75](#), [76](#)

Message Digest (MD) [76](#)

one-way functions [74](#)

pre-image resistance [74](#)

second pre-image resistance [74](#)

keyless primitives, random numbers

Pseudorandom Number Generators (PRNGs) [72](#)

Random Number Generators (RNGs) [72](#)

keystore files [417](#)

keystream generation mode [83](#)

Kimoto Gravity Well (KGW) [277](#)

reference link [277](#)

Know Your Customer (KYC) [60](#), [187](#), [589](#), [651](#)

Kotlin [686](#)

Kovan test network [369](#)

KYC-Chain

URL [651](#)

L

Last In, First Out (LIFO) [211, 342](#)

Latest Message Driven Greediest Heaviest Observed SubTree (LMD GHOST) [523](#)

layer 0 techniques, privacy

I2P [726](#)

Tor [726](#)

Layer 1 and Layer 2 solutions, privacy

anonymous signatures [733](#)

attribute-based encryption (ABE) [733](#)

CoinJoin [732](#)

confidential transactions [732](#)

homomorphic encryption [727](#)

indistinguishable obfuscation (IO) [727](#)

MimbleWimble [733](#)

mixing protocol [733](#)

privacy managers [734](#)

secure multiparty computation [730](#)

state channels [728](#)

trusted hardware-assisted confidentiality [730](#)

zero-knowledge proofs [728](#)

Lazooz [60](#)

URL [60](#)

leader blocks [718](#)

ledger decoupling [527](#)

ledger proofs [302](#)

Ledger proofs [300](#)

Legal Knowledge Interchange Format (LKIF) [292](#)

legislation

URL [582](#)

Libbitcoin

URL

libP2P

URL [524](#)

Libra

URL [574](#)

libraries, Hyperledger

Aries [529](#)

Quilt [531](#)

Transact [529](#)

Ursa [531](#)

LibUrsa [531](#)

LibZmix [531](#)

light clients [352](#)

Light Emitting Diode (LED) [636](#)

limiting factors, enterprise blockchain [656](#)

lack of access governance [656](#)

lack of privacy [658](#)

probabilistic consensus [658](#)

slow performance [656](#)

transaction fees [658](#)

limit orders [585](#)

Linear Temporal logic (LTL) [737](#)

Lisk [61](#)

literals

enums [458](#)

hexadecimal literals [458](#)

integer literals [457](#)

string literals [457](#)

types [457](#)

Elliptic Curve Discrete Logarithm Problem (ECDLP) [108](#)

load tester [694](#)

lower bound results [146](#)

Low-level Lisp-like Language (LLL) [433](#)

low watermark [167](#)

M

machine learning [767](#)

machine-readable travel document (MRTD) [644](#)

machine-readable zone (MRZ) [644](#)

machine state

elements

Machine-to-Machine (M2M) [767](#)

Maidsafe [61](#)

URL [61](#)

MakerDAO

reference link [306](#)

management layer [621](#)

Manticore [748](#)

market capitalization

reference link [271](#)

market orders [585](#)

master key [84](#)

mathematical puzzle [225](#)

mathematics [91](#)

abelian group [92](#)

cyclic group [94](#)

fields [92](#)

finite field [92](#)

groups [92](#)

modular arithmetic [91](#)

order [94](#)

prime field [92](#)

ring [94](#)

sets [92](#)

Membership Service Provider (MSP) [548](#)

memory-bound PoW algorithms

memory-hard algorithm [380](#)

memory hard computational puzzles [272](#)

memory pool (mempool) [249](#)

Merkle-Damgard construction [75](#)

Merkle-Patricia tree (MPT) [330](#)

Merkle Patricia tries (MPTs) [346](#)

Merkle root

merkle trees

mesh network [767](#)

mesh networks

using [52](#)

message [328](#), [332](#), [335](#)

call [337](#)

components [337](#)

defining [335](#)

message authentication [69](#)

message authentication code (MAC)

message authentication codes (MACs) [84](#), [87](#)

Message Authentication Codes (MACs) [69](#)

message authentication mode

message call transaction [335](#)

parameters [335](#)

Message Digest (MD) [76](#)

MetaMask [363](#), [415](#)

account, creating [369](#)

account, funding [370](#), [371](#), [372](#), [373](#), [374](#)

ERC-20 tokens, adding [611](#), [612](#), [613](#), [614](#), [615](#)

installing [363](#), [365](#), [366](#), [367](#), [368](#)

URL [363](#)

MetaMask wallet [351](#)

methods, for providing initial number of altcoins

blockchain, creating [268](#)

pegged sidechains [268](#)

Proof of Burn (PoB) [268](#)

proof of ownership [268](#)

Michelson

URL

micro blocks [718](#)

MimbleWimble [733](#)

miners [375](#), [376](#)

minikey [198](#)

mining [37](#)

mining algorithm [227](#)

mining pools

mining rigs [386](#)

mini private key format [198](#)

Miniscript

reference link [217](#)

mixing protocol [733](#)

mobile wallets [241](#), [242](#)

model checking [736](#)

URL [697](#)

modulus [91](#)

Monero [726](#)

reference link [282](#)

URL [726](#)

Muir Glacier [311](#)

Muir Glacier network upgrade [348](#)

multi-factor authentication [68](#)

Multi-Interval Difficulty Adjustment System (MIDAS) [279](#)

multi-party non-repudiation (MPNR) [70](#)

multi-signatures [136](#)

MultiSig (Pay to MultiSig) [213](#)

My ERC20 Token (MET) [589](#)

N

Nakamoto consensus

about [182](#)

versus traditional consensus [184](#)

National Institute of Standards and Technology (NIST)

native contracts

big mod exponentiation function

Blake2 compression function

elliptic curve addition function

elliptic curve pairing

elliptic curve public key recovery function
elliptic curve scalar multiplication
identity/datacopy function
RIPEMD-160-bit hash function
SHA-256-bit hash function

Near Field Communication (NFC) [730](#)

network-assisted proofs [301](#)

network effect [188](#)

network layer [621](#)

network map service [687](#)

network simulator [694](#)

network solutions [714](#)

node [8](#)

node explorer [694](#)

Node.js [437](#)

download link [437](#)

Node Package Manager (npm) [446](#)

nodes [375](#), [376](#)

non-deterministic wallets [240](#)

non-fungible tokens [573](#)

features [573](#)

Non-Interactive (N) [132](#)

non-outsourceable puzzles [275](#)

non-repudiation [69](#)

normal operation [160](#)

not colored coins [286](#)

nothing at stake problem [275](#)

Null data/OP_RETURN [213](#)

O

off-chain and multichain solutions [720](#)

commit chains [723](#)

Plasma [723](#)

sidechains [721](#)

state channels [720](#)

sub-chains [721](#), [722](#)

tree chains [722](#)

trusted hardware-assisted scalability [723](#)

Zk-Rollups [723](#)

off-chain solutions [712](#)

offer price [584](#)

OMNI network

URL [57](#)

on-chain solutions [712](#), [715](#)

Bitcoin-NG [718](#)
block interval reduction [716](#)
block propagation [717](#), [718](#)
consensus mechanisms [720](#)
Directed Acyclic Graph (DAG) [718](#)
increase, in block size [716](#)
Invertible Bloom Lookup Tables (IBLTs) [717](#)
private blockchains [717](#)
Proof of Stake [717](#)
sharding [717](#)
transaction parallelization [715](#)

online wallets [241](#)

opcodes [211](#)

using [211](#), [212](#)

OpenBazaar [60](#)

URL [60](#), [296](#)

OpenEthereum

download link [362](#)
installing [362](#)

OpenSSL

used, for generating digital signature [121](#), [122](#)
used, for generating RSA private key pair [110](#)

used, for generating RSA public key pair [110](#)
using, for elliptic curve cryptography (ECC) [114](#)

OpenSSL command line

reference link [64](#)

working with [64](#), [66](#)

OpenZeppelin

reference link [448](#)

oracle-as-a-service platforms [309](#)

oracles [299](#), [301](#)

references [298](#)

use cases [299](#)

working [300](#)

orderer [542](#)

orderer nodes [546](#)

orders [584](#)

management [584](#)

properties [584](#)

routing systems [584](#), [585](#)

outbound oracles [309](#)

outer padding (opad) [87](#)

Out of Gas (OOG) [333](#)

Output Feedback (OFB) [89](#)

over the counter (OTC) [584](#)

Oyente [745](#), [746](#)

URL [745](#), [747](#)

P

Pacemaker

Pact [756](#)

PageSigner project, TLSNotary

reference link [301](#)

paper wallets [240](#)

Parity [351](#), [380](#)

download link [351](#)

Parity command line

accounts, creating [363](#)

partially homomorphic [124](#)

partially homomorphic encryptions (PHEs) [124](#)

Partition tolerance (P)

Password-Based Key Derivation Function 1 (PBKDF1) [85](#)

Patricia trees

Paxos

about [152](#), [153](#)
reference link [152](#)
safety and liveness, achieving [156](#)
working with [153](#), [155](#)

Paxos (PAX)

URL [574](#)

Pay to Pubkey [213](#)

Pay-to-Public-Key Hash (P2PKH) [212](#)

Pay-to-Script Hash (P2SH) [213](#)

pay to script hash - pay to witness Pubkey hash (P2SH-P2WPKH) [248](#)

pay to script hash - pay to witness script hash (P2SH-P2WSH) [248](#)

Pay to Witness public key hash (P2WPKH) [248](#)

Pay to Witness Script hash (P2WSH) [248](#)

PBFT [542](#), [562](#), [738](#)

PBFT protocols

advantages, and disadvantages [168](#)
certificates [165](#)
checkpoint sub-protocol [167](#)
messages, types [165](#)
view-change [166](#), [167](#)

Pedersen commitment

reference link [127](#)

peer-to-peer [16](#)

peer-to-peer (P2P) [76](#), [529](#), [622](#), [758](#)

peer-to-peer (P2P) protocols [669](#)

pegged sidechains [268](#)

performance, facets

scalability [661](#)

speed [661](#)

permissioned ledger [35](#)

permissions

account permissions [708](#)

node permissions [708](#)

Phase 0, Ethereum 2.0 [513](#)

beacon chain [515](#)

beacon node [515](#), [517](#)

BLS cryptography

deposit contracts [521](#)

ETH 2 [524](#)

fork choice [523](#)

P2P interface (networking) [524](#)

Simple Serialize (SSZ)

validator node [517](#), [519](#), [520](#)

Phase 1, Ethereum 2.0

Phase 2, Ethereum 2.0

Phase 3, Ethereum 2.0

physical object layer [620](#)

physical unclonable functions [756](#)

plain text (P)

planes [712](#)

Plasma [723](#)

Platform as a Service (PaaS) [624](#), [678](#)

PoET [561](#)

Devmode [562](#)

PoET CFT [562](#)

PoET SGX [562](#)

point doubling [105](#)

Point of Sale (POS) [243](#)

Polkadot [724](#)

Practical Algorithm to Retrieve Information Coded in Alphanumeric (Patricia)

Practical Byzantine Fault Tolerance (PBFT) , [8](#)

 about [159](#), [161](#)

 commit [163](#)

prepare [163](#)
pre-prepare [161](#)
sub-protocols [160](#)
working with [163](#), [165](#)

Practical Byzantine Fault Tolerance (PBFT), protocol

commit sub-protocol algorithm [164](#)
prepare sub-protocol algorithm [164](#)
pre-prepare sub-protocol algorithm [163](#)

Practical Byzantine Fault Tolerant (PBFT) [141](#)

precompiled contracts

pre-image resistance [74](#)

prepare sub-protocol algorithm [164](#)

pre-prepare sub-protocol algorithm [163](#)

pre-requisites, for building ERC-20 token

MetaMask [591](#)

Remix IDE [591](#)

price feed oracle

reference link [306](#)

prime field [92](#)

privacy [659](#)

privacy and scalability, challenges

compliance and regulation
interoperability
lack of standardization
post-quantum resistance

privacy, blockchain [724](#)

anonymity [724](#)
confidentiality [724](#)
techniques [725](#), [726](#)

privacy, facets

anonymity [661](#)
confidentiality [659](#)

privacy manager, Quorum

enclave [700](#)
transaction manager [700](#)

privacy managers [734](#)

private blockchains [34](#), [750](#)

private graph construction phase [285](#)

private key [97](#)

private nets [321](#), [322](#)

private network [395](#), [396](#), [397](#)

private transaction [659](#)

restricted private transactions [659](#)

unrestricted private transactions [661](#)

private transactions, Quorum

enclave decryption [704](#), [707](#)

enclave encryption [703](#)

propagation, to transaction managers [703](#)

working [700](#), [701](#), [702](#)

Program Counter (PC)

progress freeness [272](#)

Proof of Activity (PoA) , [275](#)

proof of authenticity [301](#)

Proof of Authority (PoA) , [698](#)

Proof of Burn (PoB) [268](#), [275](#)

Proof of Capacity (PoC) , [288](#)

proof of coinage [274](#)

proof of concept implementation, TumbleBit

reference link [284](#)

Proof of Concept (PoC)

Proof of Concept (PoC) project [655](#)

Proof of Deposit (PoD) , [275](#)

Proof of Elapsed Time (PoET) , [527](#), [756](#)

Proof of Importance (PoI)

proof of ownership [268](#)

proof of retrievability [274](#)

Proof of stake (PoS)

about

chain-based PoS

committee-based PoS

delegated PoS

types

working

Proof of Stake (PoS) , [58](#), [141](#), [274](#), [275](#), [347](#), [510](#), [529](#), [661](#), [716](#), [754](#)

Proof-of-Stake (PoS) [313](#)

Proof of Storage [274](#)

Proof of Storage (PoS)

proof of validity [301](#)

Proof of Work (PoW) , [38](#), [141](#), [185](#), [227](#), [249](#), [271](#), [275](#), [344](#), [511](#), [713](#), [754](#)

alternatives [272](#)

Proof-of-Work (PoW) [15](#), [77](#), [313](#)

about [182](#)

CPU bound

used, as solution to Byzantine generals problem [184](#)

variants

working [183](#)

proof of work (PoW) mechanism

Proofs of Knowledge (PoK) [531](#)

protocol messages

types [231](#)

Provable

URL [309](#)

pruning [713](#)

Pseudorandom Number Generators (PRNGs) [72](#)

pseudo-random numbers (PRNGs) [74](#)

public blockchain

versus enterprise blockchain [666](#)

public blockchains [34](#)

public key [97](#)

public key cryptography [15](#)

public-key cryptography [94](#)

public key infrastructure (PKI) [138](#)

Public Key Infrastructures (PKIs) [644](#)

puzzle-promise protocol [283](#)

Pycoin

reference link

Q

Quadratic Arithmetic Program (QAP) [132](#)

Quilt [526](#), [531](#)

Quorum [681](#), [696](#)

access control, with permissioning [707](#), [708](#)

architecture [697](#)

cryptography [700](#)

performance [708](#), [709](#)

pluggable consensus [709](#)

privacy manager [698](#)

private transactions, working [700](#)

setting up, with IBFT [709](#)

use cases [696](#)

Quorum architecture [697](#)

enhanced P2P [697](#)

enhanced RPC API [698](#)

enhanced state (private and public) [697](#)

modified block generation mechanism [698](#)

modified block validation mechanism [698](#)

no transaction fees [698](#)

pluggable consensus [698](#)

private transactions [698](#)

Quorum projects

pluggable architecture

Remix plugin

URL

Quorum, setting up with IBFT [709](#)

cakeshop, using

contract code availability, checking on nodes

private transaction, running

private transaction, running on nodes

private transaction, viewing in Cakeshop

Quorum Wizard, using [709](#)

URL [709](#)

quorum voting [562](#)

Quorum Wizard

installing

private transaction, running [710](#)

running, to create new network

quote [730](#)

R

Radio-Frequency Identification (RFID) [620](#)

Radix tree

RAFT [542](#), [562](#)

RAFT protocol

about [157](#)
log replication [159](#)
sub-problems [158](#)
working with [158](#), [159](#)

Raiden [752](#)

URL [752](#)

Random Number Generators (RNGs) [72](#)

Raspberry Pi node setup, blockchain-based IoT implementation

hardware prerequisites [636](#)
Node.js, installing [633](#), [634](#)

real randomness [72](#)

Recursive Length Prefix (RLP) [332](#)

defining [332](#), [333](#)
reference link [333](#)

Redundant Byzantine Fault Tolerance (RBFT) [528](#)

reentrancy bug , [741](#)

regtest

Bitcoin node, setting up in [259](#), [260](#), [262](#)

Remix IDE

static analysis [742](#), [743](#)
URL [591](#)

Remix JavaScript virtual machine

Solidity contract, deploying [597](#), [598](#), [599](#), [600](#), [602](#), [603](#), [604](#), [605](#), [606](#), [607](#), [608](#), [610](#), [611](#)

Remix plugin

URL

Remote Procedure Call (RPC) [322](#)

Remote Procedure Calls (RPCs) [235](#)

replicated state machine (RSM) [158](#)

replication

about [144](#)

replication techniques

active replication [144](#)

passive replication [144](#)

types [144](#)

reputation-based mechanisms

resistor [636](#)

return on investment (ROI) [576](#)

Reusable Proof of Work (RPOW) [185](#)

reverse oracles [309](#)

ribbon cable connector [636](#)

Ricardian contracts [293](#), [295](#), [296](#)

properties [294](#)

Ricardo [293](#)

ring signatures [139](#), [724](#), [734](#)

Rinkeby test network [369](#)

RIPEMD-160-bit hash function

Role-Based Access Control (RBAC) [663](#)

URL [663](#)

Ropsten test network [369](#)

RSA

using, for decryption [113](#)

using, for encryption [113](#)

RSA digital signature algorithm [118](#), [119](#)

encrypt then sign [119](#)

sign then encrypt [119](#)

RSA private key pair [111](#)

generating, with OpenSSL [110](#)

RSA public key pair [111](#), [112](#)

exploring [113](#)

generating, with OpenSSL [110](#)

RSA puzzle solver [283](#)

runtime bytecode

S

Sabre [529](#)

safe curves

reference link [110](#)

SAFE network

URL [57](#)

Sawtooth [526](#), [527](#), [681](#)

components [563](#)

consensus [560](#)

transaction lifecycle [562](#), [563](#)

Sawtooth development environment

PBFT, using

PoET, using

pre-requisites

setting up

Sawtooth network

setting up

S-boxes [89](#)

scalability [712](#)

improving, methods [714](#)

Scalability Trilemma [510](#)

Script language

using [211](#)

second pre-image resistance [74](#)

secret key cryptography [84](#)

secret key (KEY)

secret prefix [87](#)

secret sharing [15](#)

secret suffix [87](#)

Secure Access for Everyone (SAFE) [61](#)

secure element (SE) [241](#)

Secure Hash Algorithms (SHAs) [76](#), [77](#)

design [77](#)

secure multiparty computation [730](#)

Securities and Exchange Commission (SEC) [188](#)

security, blockchain [735](#)

formal verification [735](#), [736](#)

smart contract security [739](#), [741](#), [742](#)

security protocol [70](#)

security token offerings (STOs) [577](#)

Security Token Offerings (STOs)

security tokens [575](#)

seed [72](#)

Segregated Witness [246](#), [247](#), [248](#)

Pay to script hash - pay to witness Pubkey hash (P2SH-P2WPKH) [248](#)

Pay to script hash - pay to witness script hash (P2SH-P2WSH) [248](#)

Pay to Witness public key hash (P2WPKH) [248](#)

Pay to Witness Script hash (P2WSH) [248](#)

types [248](#)

Segregated Witness (SegWit) [716](#)

semi-private blockchains [34](#)

separation of concerns [705](#)

Serenity [376](#)

Serpent

Seth [529](#)

SHA3 (Keccak)

design [80](#), [82](#)

SHA-256

design [77](#), [79](#)

hash computation [77](#)

pre-processing [77](#)

SHA-256-bit hash function

shard chains

sharding [510](#), [511](#), [717](#)

shared key cryptography [84](#)

shared ledger [35](#)

sidechains [34](#), [721](#)

signcryption [125](#)

Silk Road marketplace [734](#)

simple payment verification (SPV) [229](#)

Simple Payment Verification (SPV) [352](#)

Simple Serialize (SSZ)

Single Board Computer (SBC) [624](#)

single-factor authentication [68](#)

Single Sign-On (SSO) [670](#)

slashing [519](#)

smart contract [54](#), [537](#)

compiling, with Truffle framework

deploying, with Geth [464](#), [465](#), [467](#)

displaying, in binary format [434](#)

interacting, via frontends

interacting, with geth via JSON RPC over HTTP

list of accounts, retrieving

- migrating, with Truffle framework
- POST requests
- testing, with Truffle framework
- Web3 method, availability checking

smart contract engines [529](#)

smart contracts [291](#)

- definitions [290](#)
- deploying [310](#)
- deploying, with Truffle framework
- formal verification [748](#)
- history [289](#), [290](#)
- interacting, via web frontend
- interacting, with Truffle framework
- properties [293](#)
- testing, with Truffle framework
- using [46](#)

smart contract services, Fabric [544](#)

- events [544](#)
- secure container [544](#)
- secure registry [544](#)

smart contracts, interacting via frontends

- HTML frontend
- JavaScript frontend
- Web3.js JavaScript library, installing

smart contracts, interacting via web frontend

app.js JavaScript file, creating
contract functions, calling
Web3 object, creating

smart contract templates [297](#), [298](#)

smart oracles [308](#)

soft fork [223](#)

Software as a Service (SaaS) [678](#)

software-assisted proofs [301](#)

Software Guard Extensions [561](#)

Software Guard Extension (SGX) [730](#)

Software Guard Extensions (SGX) , [756](#)

software oracles [304](#)

solc

Solgraph [747](#)

URL [747](#)

Solidity [298](#), [756](#)

Solidity compiler [433](#)

functions [434](#)

installation [433](#)

Solidity compiler, functions

Application Binary Interface (ABI), generating [435](#), [436](#)
compilation [436](#), [437](#)
estimate gas [435](#)
smart contract, displaying in binary format [434](#)

Solidity contract

building [591](#)
deploying, on Remix JavaScript virtual machine [597](#), [598](#), [599](#), [600](#),
[602](#), [603](#), [604](#), [605](#), [606](#), [607](#), [608](#), [610](#), [611](#)
source code [592](#), [594](#), [595](#), [596](#), [597](#)

Solidity language [451](#)

control structures [459](#), [460](#)
data types [455](#), [456](#)
error handling
events [460](#)
functions [462](#)
inheritance [461](#), [462](#)
libraries [462](#)
variables [451](#)

Solidity language, data types

reference types [458](#)
reference types, arrays [458](#)
reference types, data location [459](#)
reference types, mapping [459](#)
reference types, structs [458](#)

value types [456](#)
value types, address [457](#)
value types, Booleans [456](#)
value types, integers [456](#)
value types, literals [457](#)

Solidity language, variables

global variables [453](#)
local variables [453](#)
state variables [453](#), [455](#)

Solidity source code file

comments [450](#)
components [450](#)
importing [450](#)
layout [450](#)
version pragma [450](#)

SOLO [542](#)

sponge and squeeze construction [80](#)

sponge construction [76](#)

spreading phase [285](#)

SPV client [238](#)

stable tokens [574](#)

algorithmically stable [574](#)
commodity collateralized [574](#)

crypto collateralized [574](#)

fiat collateralized [574](#)

stakes [274](#)

stale chip problem [561](#)

state

state channels [720](#), [728](#), [752](#)

state machine replication (SMR)

about [145](#), [146](#)

fundamental ideas [145](#)

status

URL

Steemit

URL

Stem phase [285](#)

Stratum [231](#)

reference link [231](#)

stream ciphers [88](#)

asynchronous stream ciphers [88](#)

synchronous stream ciphers [88](#)

strong collision resistance [75](#)

sub-chains [721](#)

Substitution-Permutation Network (SPN) [88](#)

Succinct Non-Interactive Argument of Knowledge (SNARKs) [728](#)

succinct (S) [132](#)

Sumeragi [528](#)

Swarm

symmetric cryptography [64](#)

about [84](#), [85](#)

block ciphers [88](#), [89](#)

hash-based MACs (HMACs) [87](#), [88](#)

message authentication codes (MACs) [87](#)

stream ciphers [88](#)

synchronous stream ciphers [88](#)

sync node [234](#)

T

Tangle [760](#), [766](#)

T-Certs [542](#)

temporal connectives [737](#)

temporal logics [737](#)

Tendermint

about [174](#)

implementing [182](#)

properties [176](#)
reference link [182](#)
state variables [178](#), [179](#)
system model, elements [176](#)
types, of messages [177](#), [178](#)
working [179](#), [181](#), [182](#)

Tendermint Core

reference link [182](#)

Terawatt hash (TWh) [273](#)

Tessera [700](#)

testnet

Bitcoin node, setting up in [257](#), [259](#)

testnets [320](#)

test network [388](#)

Tether

URL [57](#)

Tether gold

URL [574](#)

The Open Group Architecture Framework (TOGAF)

application architecture domain [675](#)
data architecture domain [675](#)
technology architecture domain [675](#)

The Open Group Architecture Framework (TOGAF) [674](#)

business architecture domain [675](#)

URL [674](#)

The Realitio project

URL [309](#)

thin block [249](#)

threshold signatures [137](#), [138](#)

timestamp dependence [742](#)

time warp attacks [277](#)

TLS-N based mechanism [302](#)

TLSNotary [301](#)

URL [300](#)

token

versus cryptocurrency [291](#)

token economics (tokenomics)

token engineering

tokenization

process [575](#), [576](#)

tokenization, advantages

decentralization [570](#)

faster transaction processing [569](#)

flexibility [569](#)
fractional ownership [570](#)
innovative applications [570](#)
low cost [569](#)
low entry barrier [570](#)
more liquidity [571](#)
security [570](#)
transparency [570](#)
trust [570](#)

tokenization, disadvantages

legality of tokens [571](#)
regulatory issues [571](#)
security issues [572](#)

tokenization, on blockchain

advantages [569](#), [570](#)
disadvantages [571](#)

tokenized blockchains [35](#)

tokenized IPO [577](#)

token-less blockchains [36](#)

token offerings

about [576](#), [578](#)
decentralized Autonomous Initial Coin Offering (DAICO) [577](#)
equity token offerings (ETOs) [577](#)

initial coin offerings (ICO) [576](#)
initial exchange offering (IEO) [577](#)
security token offerings (STOs) [577](#)

tokens

fungible tokens [572](#)
non-fungible tokens [573](#)
security tokens [575](#)
stable tokens [574](#)
types [572](#)
versus coin [572](#)

token standard EIP

URL [582](#)

token standards

about [579](#)
ERC-20 [580](#)
ERC-223 [580](#)
ERC-721 [580](#)
ERC-777 [580](#)
ERC-884 [581](#)
ERC-1400 [582](#)
ERC-1404 [582](#), [583](#)

token taxonomy

Token Taxonomy Framework (TTF)

URL

tools, Hyperledger [531](#)

Avalon [531](#)

Caliper [533](#)

Cello [533](#)

Explorer [533](#)

Tor

URL [726](#)

total order broadcast [146](#)

Town Crier

URL [309](#)

trade

components [585](#)

life cycle [587](#)

underlying instrument [585](#)

trade life cycle

confirmation [587](#)

end-of-day processing [587](#)

execution and booking [587](#)

market manipulation [588](#)

order anticipators [587](#)

post-booking [587](#)

pre-execution [587](#)

settlement [587](#)

trade ticket [585](#)

trading [583](#)

traditional consensus

versus Nakamoto consensus [184](#)

Transact [526](#), [529](#)

transaction [22](#), [328](#), [332](#)

executing [338](#)

validating [338](#)

transaction families [527](#)

transaction life cycle, Fabric [553](#), [555](#)

transaction lifecycle, Sawtooth [562](#), [563](#)

transaction malleability [735](#)

transaction parallelizability

reference link [715](#)

transaction receipts [340](#), [341](#)

transaction receipts, elements

bloom filter [341](#)

gas, using [340](#)

log set [340](#)

post-transaction state [340](#)

transactions

contract creation transaction [329](#)

message call transaction [329](#)

transactions per second (TPS) [63](#)

transaction, standard fields

data field [330](#)

gas limit field [329](#)

gas price field [329](#)

init field [330](#)

nonce field [329](#)

signature field [329](#), [330](#)

To field [329](#)

value field [329](#)

transaction substate [338](#)

log series [338](#)

refund balance [338](#)

suicide set or self-destruct set [338](#)

touched accounts [338](#)

transaction trie [330](#)

Transport Layer Security (TLS) [301](#), [640](#), [690](#)

tree chains [722](#)

Trezor

URL [240](#)

Trinity [351](#)

download link [351](#)

Trinity client

reference link [351](#)

Triple DES (3DES)

Truebit

URL [306](#)

TrueBit

URL [309](#)

Truffle , [444](#), [446](#), [640](#)

URL [444](#)

Truffle framework

initializing

installing

used, for compiling smart contract

used, for deploying smart contracts

used, for developing decentralized application

used, for interacting with smart contracts

used, for migrating smart contract

used, for testing smart contract

used, for testing smart contracts

Trusted Compute Specifications [533](#)

trusted execution environments (TEEs) [670](#)

Trusted Execution Environment (TEE) , [303](#), [527](#), [730](#)

trusted hardware-assisted confidentiality [730](#)

trusted hardware-assisted proofs [300](#), [303](#)

trusted hardware-assisted scalability [723](#)

T-Shaped cobbler [636](#)

TumbleBit

reference link [284](#)

TumbleBit++

reference link [284](#)

tumbler [283](#)

two-phase commit

reference link [155](#)

two-way peg [268](#), [721](#)

two-way pegged sidechain [34](#)

U

UK Jurisdiction Taskforce (UKJT) [291](#)

uncle block [343](#)

Uniform Resource Identifier (URI) [243](#)

University College London (UCL) [754](#)

unsolvability results [146](#)

Unspent Transaction Output (UTXO) [209](#)

Ursa [526](#), [531](#)

user interfaces (UIs) [463](#)

User Interface (UI) [447](#)

building, with Drizzle

V

validator node [517](#), [519](#), [520](#)

versus beacon node [521](#)

Value Added Tax (VAT) [763](#)

value types [456](#)

venture capital funds (VCs)

Verge [726](#)

view change [160](#)

virtual mining [274](#)

virtual read-only memory (virtual ROM)

Vitalik Buterin

URL [311](#)

Vyper

W

Waffle

reference link [448](#)

walking the chain concept [689](#)

wallet [351](#)

Wallet Import Format (WIF) [196](#)

wallet software [239](#)

wallet software, Bitcoin

brain wallets [240](#)

deterministic wallets [240](#)

hardware wallets [241](#)

hierarchical deterministic wallets [240](#)

mobile wallets [241](#), [242](#)

non-deterministic wallets [240](#)

online wallets [241](#)

paper wallets [240](#)

weak collision resistance [74](#)

Web3

exploring, with Geth [463](#), [464](#)

Web3 JavaScript API

web3J component

URL [672](#)

Web3.js JavaScript library

installing

Web3 methods

availability, checking by calling

Web3 object

creating

Weierstrass equation [100](#)

Whisper

whistleblowing validator [520](#)

Why3 [741](#), [744](#), [745](#)

URL [744](#)

Wireshark

reference link [236](#)

Witnet

URL [309](#)

World of Accountancy [294](#)

World of Law [294](#)

X

XBT terminal [243](#)

Y

YUL language

reference link

Z

Zcash [726](#)

URL [30](#), [127](#)

zcoin

reference link

zero-knowledge proofs (ZKP) [282](#)

zero-knowledge proofs (ZKPs) [127](#), [128](#), [130](#), [131](#), [646](#), [670](#), [724](#), [755](#)

phases [130](#)

Zero-knowledge range proofs (ZKRP) [134](#)

Zero-knowledge Scalable Transparent Arguments of Knowledge (zk-STARKs) [133](#), [134](#)

zero-knowledge Succinct Transparent Argument of Knowledge (zk-STARK) [728](#)

zero knowledge (zk) [132](#)

Zk-Rollups [723](#)

ZK-Rollups

zk-SNARKs [755](#)

zk-STARKs [755](#)

zk-STARKS

Zokrates

URL [755](#)

Zooko's Triangle [53](#)

EXPERT INSIGHT

Mastering Blockchain

A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more



Third Edition



Imran Bashir

Packt